Real-Time Foveated Rendering In Virtual Reality For Ray-tracing

Andreas Polychronakis

January 2020

Electrical And Computer Engineering,
Technical University Of Crete

Thesis Committee
Associate Professor Katerina Mania, Thesis Supervisor
Professor Konstantinos Mpalas
Associate Professor Antonios Delligianakis

Acknowledgement

This thesis would have been impossible without the support of many people:

First of all, I would like to thank my supervisor Katerina Mania, for trusting me and giving me the opportunity to work on Foveated Ray-Tracing and for always being available for valuable help and suggestions.

Second, i have to pay credits to George Koulieris for always helping out once help was needed and giving me different ideas to how to approach my thesis.

Similarly, i have to thank Gregory D. and Salva K. for their help and support they provided me when working in the lab on my thesis and being good friends for me.

Furthermore, i would like to thank my friends Magda A., Manos A., Zisis C., Aneza D., Evaggelos G., Ardit N., Lina T. and Andreas S. for their moral support, university related collaboration and most of all, all the fun we had throughout these years.

Finally, and most importantly, I would like to thank my family who had to bear the extra stress of having me write this thesis, and without whose great patience and support this thesis would never have been possible.

Περίληψη

Σκοπός της παρούσας διπλωματικής εργασίας είναι η ανάπτυξη λογισμικού διαδικασίας παραγωγής τρισδιάστατων φωτορεαλιστικών γραφικών τα οποία απεικονίζονται σε μάσκες Εικονικής ή Επαυξημένης Πραγματικότητας. Η διαδικασία αυτή χρησιμοποιεί τον αλγόριθμο παρακολούθησης ακτινών (ray-tracing) ενώ ταυτόχρονα εκμεταλλεύεται τη γνώση του οπτικού πεδίου κίνησης του ματιού του χρήστη (eye tracking) για οθόνες τοποθετημένες στο κεφάλι Εικονικής Πραγματικότητας. Οι αλγόριθμοι Γραφικής που εκμεταλλεύονται την ακολουθία κίνησης του ματιού (foveated rendering) χρησιμοποιούν ανιχνευτές ματιού οι οποίοι είναι ενσωματωμένοι σε μάσκες Εικονικής ή Επαυξημένης Πραγματικότητας για να παραχθούν εικόνες με λιγότερη λεπτομέρεια στην εξωτερική περιοχή του οπτικού πεδίου του ματιού. Αυτό γίνεται ώστε να μειωθεί το υπολογιστικό κόστος της διαδικασίας παραγωγής της εικόνας. Σκοπός είναι να υπάρξει υψηλότερος ρυθμός παραγωγής εικόνων χωρίς να γίνεται αισθητή η μείωση της ποιότητας από τον χρήστη. Αυτό επιτυγχάνεται με την χρήση της νέας γενιάς επεξεργαστών για γραφικά (RTX 2060 GPU card) που έχει προτείνει η NVidia και με τη χρήση του λογισμικού OptiX, ώστε να βελτιστοποιηθεί η κατανομή πόρων για τον αλγόριθμο παρακολούθησης ακτινών. Στην μάσκα Εικονικής Πραγματικότητας που χρησιμοποιήθηκε έχουν ενσωματωθεί ανιχνευτές του ματιού από την Arrington Research σκοπεύοντας να γίνει η παρακολούθηση της κατεύθυνσης του ματιού και η εκτίμηση του οπτικού πεδίου του σε πραγματικό χρόνο. Για την επίτευξη της βελτιστοποίησης στην εφαρμογή μειώνεται ο αριθμός των ακτινών που παράχθηκαν με τον αλγόριθμο παρακολούθησης ακτινών στην περιφερειακή όραση παράγοντας ακτίνες για μικρότερο αριθμό pixel. Τέλος, χρησιμοποιείται μια μέθοδο επεξεργασίας της εικόνας για να καλυφθούν τα κενά καθώς και τυχόν ατέλειες λόγω της μειωμένης ποιότητας. Η αξιολόγηση της μεθόδου έδειξε ότι επιτυγχάνει ικανοποιητική ταχύτητα απεικόνισης γραφικών σε πραγματικό χρόνο ενώ οι οπτικές διαφορές μεταξύ κεντρικού οπτικού πεδίου και περιφερειακού δε γίνονται αντιληπτές. Τέτοιες τεχνικές μπορούν να εφαρμοστούν για αλληλεπίδραση με βάση το βλέμμα (gaze interaction) σε ανάπτυξη λογισμικού για νέες εφαρμογές Εικονικής και Επαυξημένης Πραγματικότητας. Οι δοκιμές εφαρμόζονται σε τρισδιάστατες σκηνές που αναπαραστούν τρισδιάστατες ανακατασκευές αποτύπωσης μνημείων πολιτιστικής κληρονομιάς με σκοπό την ανάδειξή τους για εφαρμογές Εικονικής ή Επαυξημένης Πραγματικότητας.

Abstract

The goal of this thesis is the development of software supporting 3D photorealistic graphics for Virtual or Augmented Reality environments. Foveated rendering utilizes an eye tracker embedded in a Virtual Reality headset to produce images with progressively less detail in the peripheral vision located outside the zone gazed by the fovea, based on eye tracking input. By doing that, the computational cost of rendering is reduced without any noticeable change of image quality. The ray-tracing algorithm produces superior rendering quality compared to rasterization, however, its computational cost is high because of intense ray intersection calculations with the 3D scene. Real-time ray-tracing is challenging. We propose a ray-tracing rendering pipeline for which we apply foveated rendering so that a higher frame rate without perceived reduction in quality.

This is achieved by using Nvidia newest generation of graphic process unit card, the RTX 2060, and OptiX software to have the highest available performance for ray tracing. Arrington Research's Viewpoint EyeTracker embedded in the NVIS SX111 VR Head Mounted Display monitored gaze position in real-time. In our work, we take advantage of the eye tracking capabilities of our HMD to separate the center of the user's field of view from the periphery. We then reduce the sampling done using ray tracing in the periphery, taking samples for only a smaller amount of pixels. Next, we duplicate neighboring pixels to cover all blank spots using the information of our closest neighbors. Finally, we use Gaussian blurring to cover all the imperfections in the recreated frame.

Evaluations and user tests show that our method achieves real-time frame rates, while visual differences compared to fully rendered images are hardly perceivable. Our approach, therefore, achieves foveated real time ray-tracing as displayed in a binocularly eye-tracked Head Mounted Display, for complex 3D scenes, without any perceived visual differences compared to full scale ray-tracing. Such techniques could be applied for gaze-based interaction in 3D scenes displayed in developed software for Virtual or Augmented Reality applications. Testing such techniques is demonstrated in scenes which represent 3D reconstructions of cultural heritage sites aiming to pursue their scientific documentation in Virtual or Augmented Reality applications.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Purpose

The quest for high fidelity, photorealistic 3D computer graphics has amplified shading complexity and lighting. The associated computational cost has also increased as a result of the high pixel density of most displays. As Virtual Reality (VR) training and gaming applications demand both high quality imagery and real-time performance, high update rates are crucial. Minimizing system latency and associated perceived motion sickness is a prerequisite for the success of the VR industry [5].

Foveated rendering has the potential to decrease the computational needs of a rendering pipeline. It exploits the principle that the human visual acuity falls off rapidly as eccentricity increases. Foveated rendering uses this attribute of the Human Visual System (HVS) to maintain high quality of rendering in the retina center (the fovea) and to decrease it at the periphery where the acuity of the human eye falls sharply.

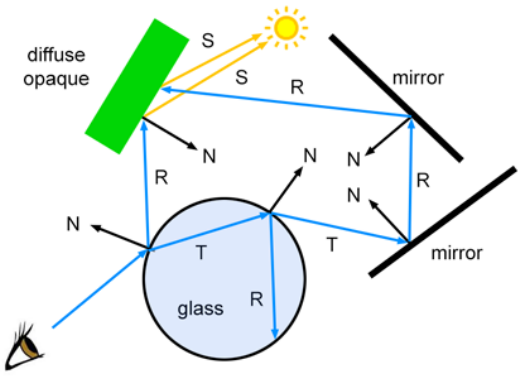


Figure 1: Tracing a ray, S: shadow-ray, R: reflection-ray, T: refraction-ray, N: normal of surface [18]

The Ray Tracing algorithm is based on backwards tracing of rays which are cast from the camera to the scene in order to calculate the color for each pixel as shown in Figure 1. The advantage of this method is that it produces high quality reflections, refractions and hard shadows, resulting in a realistic image and lighting but at the great cost of real-time performance. As shown in Figure 1, the ray hitting a scene object produces a shadow as well as reflection and refraction rays. For each ray, we must calculate its intersection with objects in the 3D world. Intense intersection calculations for each pixel between rays from the camera directed to the scene and associated decisions about which object is hit, is the main reason why ray-tracing in real-time is hard.

Nvidia for the last ten years has developed its own software for ray tracing rendering, e.g., the Optix. In addition, it has released the new generation of graphics

process units, e.g., the RTX series cards which support ray tracing using the RT cores that have been designed to maximize the ray intersection process in a hardware level. Both the software and hardware provided by Nvidia has made ray tracing rendering run in real-time on the newest generation of computers.

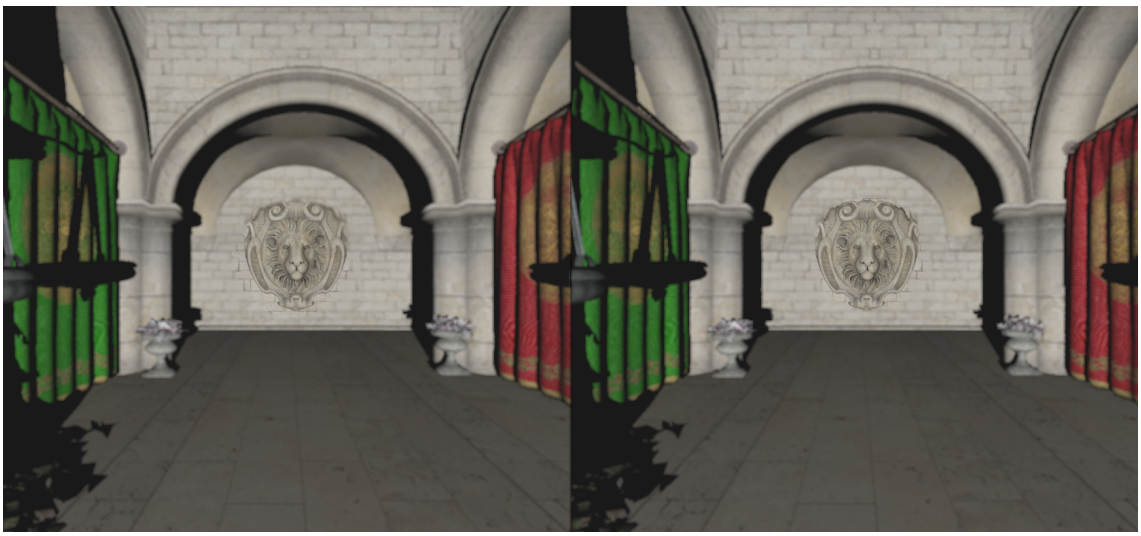


Figure 2: Example of the frame produce for our Head Mounted Display NVIS SX111 (simulation of cultural heritage scene)

In this theses, we create a foveated rendering ray-tracing algorithm for Head Mounted Displays with embedded eye tracking, aiming to produce high quality images at a faster frame rate by distributing computational load where it matters, e.g., where a user is looking at based on eye tracking input. To create the system we used the Nvidia Optix Software together with the newest generation graphics process unit provided from Nvidia (RTX series) as well as the head tracker device InertiaCube3 and the ViewPoint EyeTracker software provided by Arrington Research. The Head Mounted Display utilized is the NVIS SX111 with embedded binocular eye tracking, provided by Arrington Research.

Our system provides a technique for foveated ray-tracing using a probability model based on human eye visual acuity. Previous work used a linear probability model [2] which had a smoother decrease in resolution that does not represent the human eye as accurately as our approach. Using our visual acuity model we separated the visible area into three regions: the foveal, the periphery and the "outside zone", which lies farther than the periphery until the edges of the display. While we use full ray tracing inside the foveal, where the user can see details with high precision, we reduced the amount of rays sampled inside the periphery and the outside zone to one ray for a set of pixels. To cover the imperfections in the regions outside the foveal, we duplicated the values of one ray to a set of pixels and then added a Gaussian blurring mask, which recreated a blurry but complete image. Since the eye cannot distinguish details outside the foveal as well as inside the foveal, it cannot distinguish the difference in quality. In Figure 2, we can see that the quality in the center(considering that the eye is fixated on the lion in the center of the frame) is higher and the quality in the periphery is reduced, but the blurring is unnoticeable

by the user.

The benchmark evaluation showed that as the samples of ray per pixel increase, our technique has good speed up values except from when we have one sample per pixel. As ray tracing is implemented for at least four samples per pixel, our method achieves overall good performance (48 fps against 26 fps for full ray tracing, which results in an 84% increase in performance for a highly detailed scene). Although acceptable performance is achieved, we didn't achieve the 120 fps speed needed for real-time rendering in immersive VR so that motion sickness is improved for some users. This is tested employing a scene resembling a cultural heritage site.

We conduct a user-study to investigate whether our foveated ray-tracing rendering technique is perceptible by users when accuracy of rendering drops outside the foveal region. Because of technical issues related to the specific eye tracking device, we anticipate that if these are solved, our results will be even better, making our foveated rendering technique almost imperceptible. We nevertheless show that our method, for specific values of eccentricity, achieves similar evaluation results when compared with an image rendered at uniformly maximum quality.

1.2 Thesis Structure

This thesis is organized to provide a continual narrative to be followed. There are six chapters that are set out as follows.

After this Introduction, the 2nd chapter acts as a prologue and introduces the reader to the fields of Virtual Reality, the human eye's anatomy, eye/head-tracking, types of virtual lights, Phong illumination model, ray-tracing, Gaussian blur, visual angle, euclidean distance and Foveated rendering.

The 3rd Chapter introduces the software, API's and tools, used for the implementation. Those are the specific eye-tracking and head-tracking and ray-tracing software.

In the 4th Chapter, the implementation of the foveated ray-tracing algorithm is presented, as devised for the NVIS SX111 Head Mounted Display with embedded binocular eye tracking by Arrington Research. The steps taken to create the 3D scenes and in general the adjustments that were applied within our application as well as the API's and tools used are presented. Also, technical issues that occurred are explained as well as decisions taken to address them. Certain basic examples and code are demonstrated.

The 5th Chapter describes the tests devised in order to check the speed of our foveated ray-tracing rendering algorithm and its formal evaluation by users.

The 6th Chapter includes some conclusions and future ideas for of our work.

2 Background

2.1 Virtual Reality

Virtual reality (VR) is an interactive computer-generated experience taking place within a simulated environment. It incorporates mainly sound and visual feedback, but may also allow other type of sensory feedback like haptic. This virtual environment can be similar to the real world or it can be fantastical. Current VR technology most commonly uses virtual reality headsets or multi-projected environments, sometimes in combination with physical environments, to generate realistic images, sounds and other sensations that simulate a user's physical presence in a virtual or imaginary environment. A person using virtual reality equipment is able to look and move around the artificial world and interact with virtual features or items. This effect is commonly created by VR headsets consisting of a head-mounted-display with two small screens in front of the eyes, but can also be created through specially designed rooms with multiple large screens. VR systems that include transmission of vibrations and other sensations to the user through a game controller or other devices are known as haptic systems.

2.1.1 Immersion

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. Immersion can also be defined as the state of consciousness where a person awareness of physical self is transformed by being surrounded in an artificial environment; used for describing partial or complete suspension of disbelief, enabling action or reaction to stimulation's encountered in a virtual or artistic environment. The degree to which the virtual or artistic environment faithfully reproduces reality determines the degree of suspension of disbelief. The greater the suspension of disbelief, the greater the degree of presence achieved. To achieve a sense of full immersion, we must make the human senses (sight, sound, touch, smell, taste) perceive the digital environment to be physically real. The current technology can fool the senses using panoramic 3D displays (visual), surround sound acoustic (auditory), Haptics and force feedback (tactile), smell replication (olfactory) and taste plication (gustation).

2.1.2 Hardware

In this section, we will describe the input devices that are utilized in VR. The most common device is the Head-Mounted Display (HMD). There also other input devices such as the virtual glasses or goggles, data gloves, data suit, joystick, keyboards, but also the haptic devices which they enable the sense of touch when a user interacts with an object in VR. In this thesis, we use a Head Mounted Display

equipped with binocular eye tracking (NVIS SX111, eye tracking by Arrington research).

2.1.3 Head Mount Display

The HMD is a display device, worn on the head, that has a little display optic before one (monocular HMD) or each eye (binocular HMD). A HMD has numerous uses, including in gaming, aviation, engineering and medicine lift. An HMD is the essential part of virtual reality headsets. The most current plugins coming with HMD is eye tracking, where eye tracking provide the program with data from user's eye (gaze position, pupil size etc.). In the market currently, there are cheap commercial HMDs such as the Oculus Rift, HTC Vive and Vive PRO eye. Also, there are cheap HMDs with eye tracking capabilities such as the FOVE 0. In this thesis, we use a high-end expensive HMD with binocular eye tracking, e.g. the NVIS SX111.

Figure 3: Oculus Rift [19]

The Oculus Rift has two Pentile organic light-emitting diode (OLED) displays, a resolution 1080 x 1200 pixels per eye (2160 x 1200 pixels combined), a refresh rate of 90 Hz, a field of view (FOV) at 110 degrees, six degrees of freedom and can track a space that's 9 x 9 ft. As you can see in Figure 3, the Oculus has two touch controllers and two sensors. The touch controllers are a pair of tracked controllers that provide intuitive hand presence in VR. Providing the feeling that they virtual hands are actually your own. The two sensors also known as Rift sensors track constellations of infrared(IR) light emitting diodes(LEDs) to translate the movement of user into VR whether you 're sitting down or standing up. In addition, Oculus integrated headphones provide a 3D audio effect. Oculus rift does not support eye tracking.

Figure 4: HTC Vive [20]

THe HTC Vive has a Dual Active-Matrix Organic Light-Emitting Diode (AMOLED) 3.6" diagonal display, a resolution of 1080 x 1200 pixel per eye (2160 x 1200 pixels combined), a refresh rate of 90 Hz, a FOV at 110 degrees and it can track a space that's 15 x 15 ft (Figure 4). In addition, HTC vive does not support eye tracking. Also, we see that HTC has two controllers and two sensors and it's working the same way that Oculus rift works, however, the sensors of the HTC are wireless Oculus's ones are wired.

As the need for intuitive interaction in VR resulted to eye gaze potentially serving as one affordance for interaction, companies producing HMDs tried to incorporate eye tracking into HMDs.

Figure 5: HTC Vive Pro [20]

The HTC vive pro eye power the Eye-Tracking technology for VR, enabling tracking and analysis of eye movements. The embedded eye tracker of HTC vive pro eye has an accuracy of $0.5^o - 1.1^o$ visual arc and a frequency for gaze data output at 120Hz for both eyes(binocular) and it has a trackable FOV of 110^o. It has a Dual AMOLED 3.5" diagonal, a resolution of 1440 x 1600 pixels per eye (2880 x 1600 pixels combined), a refresh rate is at 90 Hz, a FOV of 110 degrees. As we can see in Figure 5, the HTC vive pro eye is almost look the same as HTC vive (Figure 4)

but with better characteristics and is slightly different in the appearance.

Figure 6: Fove ([21])

The FOVE 0 HMD with eye tracking capability has a dual OLED display, a resolution of 1280 x 1440 pixel per eye (2560 x 1440 pixels combined), a refresh rate of 90 Hz, a FOV at 100 degrees, an IR-based position tracking and an infrared eye tracking system per eye with accuracy less than 1^o of visual arc which is though, tested by the community and has proved inadequately robust in relation to eye tracking capabilities.

Figure 7: NVisor SX111

The HMD used in this thesis is a high-end expensive HMD with binocular eye tracking capability. The NVIS SX111 has a dual Liquid crystal on silicon (LCOS) display with the resolution for each display be at 1280 x 1024 (2560 x 1600 pixels combined) with a total viewing covering 102 degrees' horizontal by 64 degrees vertical and with 111 degrees across the diagonal. An eye tracking system has been installed on the HMD by Arrington research so it can track the gaze of the human eye. The eye tracker has an accuracy $0.25^o - 1.0^o$ of visual arc and selectable frequency by the user between 60 Hz and 30 Hz. In addition, a 3-degree of freedom (rotational) tracker was install to provide the rotation of the user head.

In summary, in this thesis, the NVIS SX111 is the HMD we used in our foveated rendering system because as we explain above the Oculus Rift and HTC Vive don't offer a system for eye tracking. The newer HTC Vive Pro Eye is the most recent HMD with eye tracking capability which hasn't been tested yet in our lab.

2.2 Human Eye

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 is one of the most complex organs 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.

2.2.1 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, know as the fibrous tunic, is composed of the cornea and sclera. The middle layer, known as the vascular tunic or unvea, 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 viterous body, and the flexible lens. The aqueous humor is a clera 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 transaprent 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.

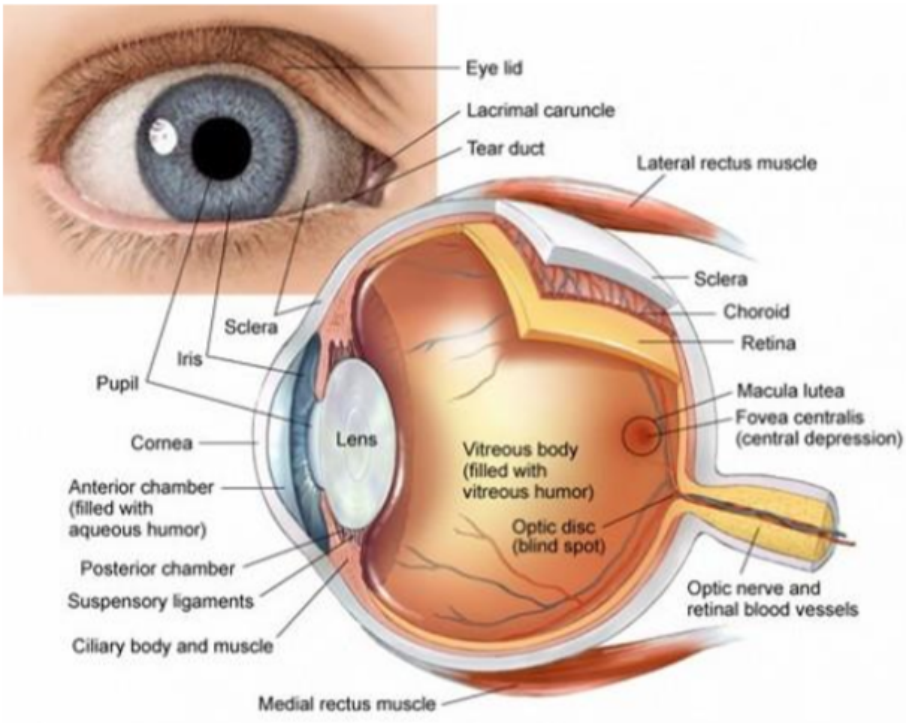


Figure 8: Anatomy of the Human Eye [22]

Below, the basic front layer parts of the eye are listed and described.

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.

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.

2.2.2 Vision: The Eye's Functionality

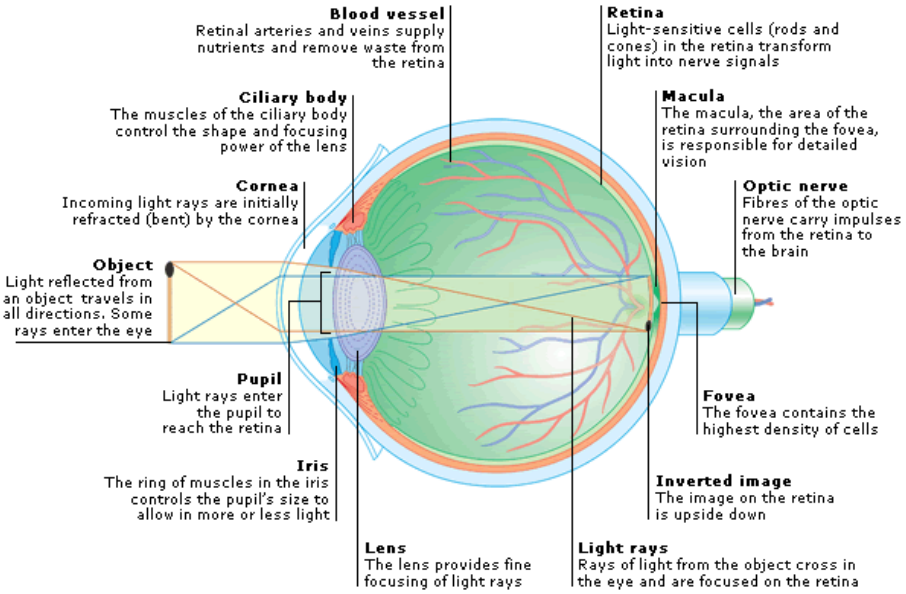


Figure 9: How Each Part of the Eye Interacts and Adjusts to the Incoming Light/Image Display [23]

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.

2.2.3 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.

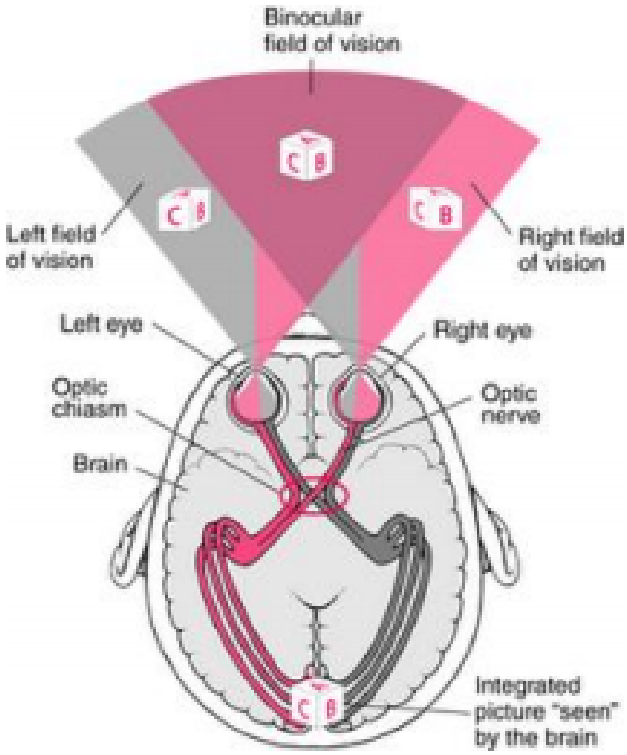


Figure 10: Binocular vs. Monocular Vision

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.

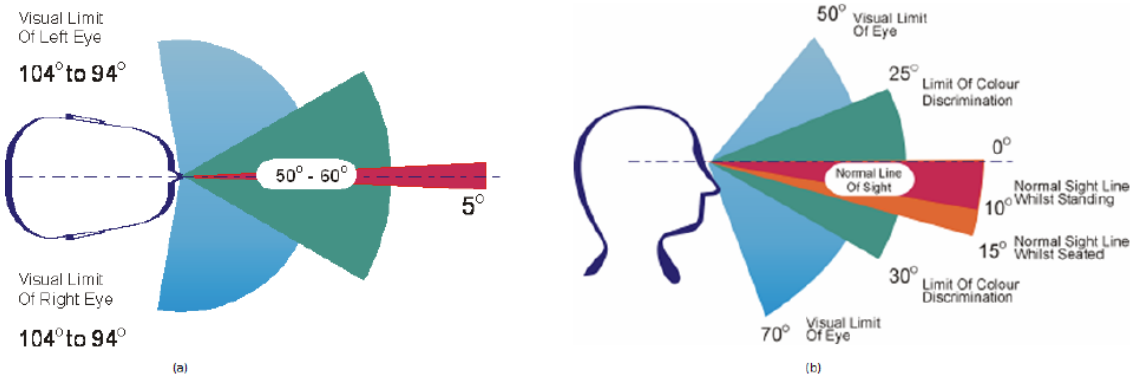


Figure 11: (a) Horizontal Field of View (b) Vertical Field of View

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.

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.

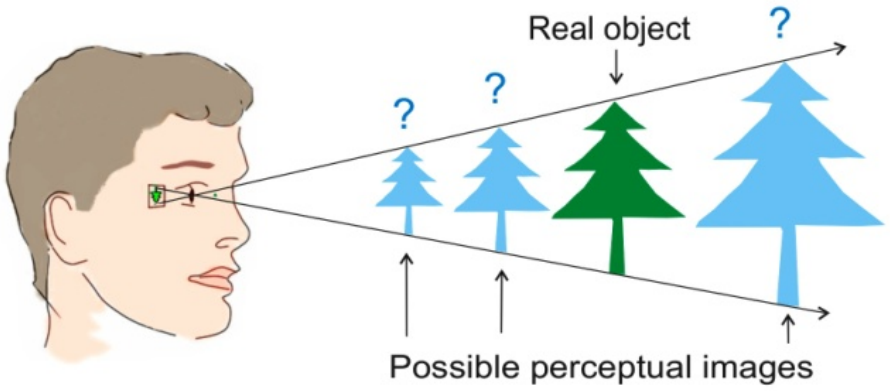


Figure 12: Perception of a plane object in monocular vision [24]

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".

In this thesis, we want to created the stereopsis phenomenon for virtual reality experiences in order to increase the immersion of the user. The two displays of our HMD must render the same image but with minor differences in order to create the stereopsis phenomenon and provide the user with the illusion that there is depth in our 3D environment. In order to create the stereopsis phenomenon, we placed two virtual cameras, one for each eye, in the virtual environment with a short distance between them based on the average human's Inter-Pupilary Distance (IPD)

2.2.4 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 this thesis will be described. The main types of eye movements are the following saccade, pursuit, smooth, blinking, and compensatory .

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. 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.

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.

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.

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 offcenter 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.

2.2.5 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 exist 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 Virtual Environment (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 immersive virtual environment (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.

In summary, we attempt to render a 3D environment with similar parameters to the visual model of the human eye. We combine ray tracing rendering to create a realistic environment with foveated rendering in order to optimise the rendering process by reducing the resolution of the image in the areas where the human eye cannot discern smaller details.

2.3 Types of Virtual Lighting

While there are many potential choices, a handful of visual light types consistently see use in 3D games. Certain visual lights globally affect the entire scene, whereas other visual lights affect only the area around the light.

2.3.1 Ambient Light

Ambient light is a uniform amount of light applied to every single object in a scene. The amount of ambient light might differ for different levels in a game, depending on the time of day. A level set at night will have a much darker and cooler ambient light than a level set in the daytime, which will be brighter and warmer.

Because it provides an even amount of lighting, ambient light does not light different sides of objects differently. It is a global amount of light uniformly applied to every part of every object in a scene. This is akin to the sun on a cloudy day in nature.

2.3.2 Directional Light

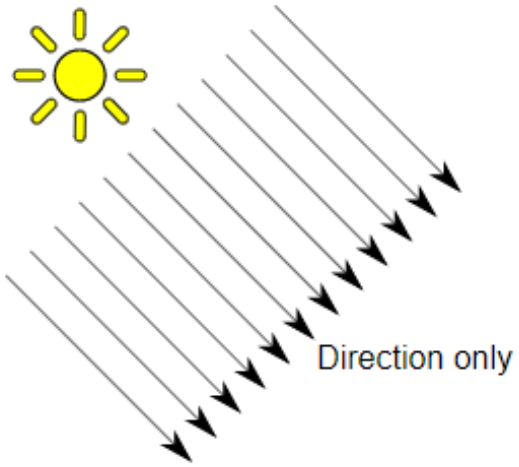


Figure 13: Directional Light [25]

A directional light is a light emitted from a specific direction. Like ambient light, directional light affects an entire scene. However, because a directional light comes from a specific direction, it illuminates one side of objects while leaving the other side in darkness. An example of a directional light is the sun on a sunny day. the direction of the light depends on where the sun is at that time day. The side facing the sun is bright, while the other side is dark, as seen in Figure 13.

2.3.3 Point Light

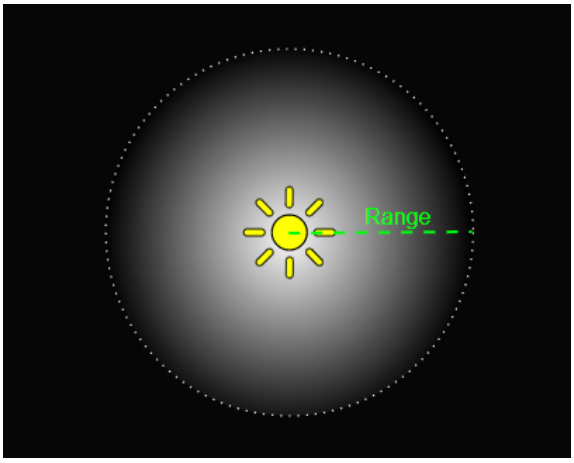


Figure 14: Point Light [25]

A point Light exists at a specific point and emanates in all directions from that point. Because it starts at specific point, a point light also illuminates only one side of an object. Usually, a point light also has a radius of influence. There's visible light in the area immediately around the light, but it slowly dissipates until it no longer adds light. The point light doesn't go on infinitely, as seen in Figure 14.

2.3.4 Spotlight

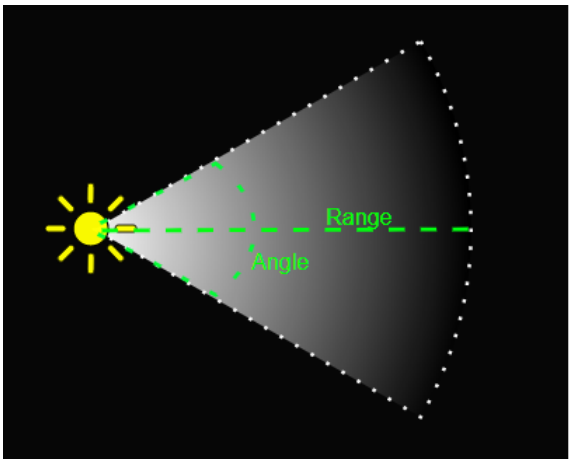


Figure 15: Spot Light [25]

A spotlight is much like a point light, except that instead of traveling in all directions, it's focused in a cone. To simulate a spotlight, you need all the parameters of point light and additionally the angle of the cone, as seen in Figure 15.

In this thesis, we created a combination from the above lights to be used as a light source of our virtual environment. The light source that has been created, has infinity range like the directional light and is emitted to all directions like the point light.

2.4 Phong Illumination Model

To simulate lights, not only do you need their associated data, you also need to calculate how the lights affect the objects in the scene. A tried-and-true method for approximating light is a bidirectional reflectance distribution function (BDRF), which is a function that approximates how light bounces off surfaces. There are many different types of BDRFs, but a classic one is the Phong reflection model.

The Phong model is a local lighting model because it doesn't calculate secondary reflections of light. In other words, the reflection model lights each object as if it's the only object in the entire scene.

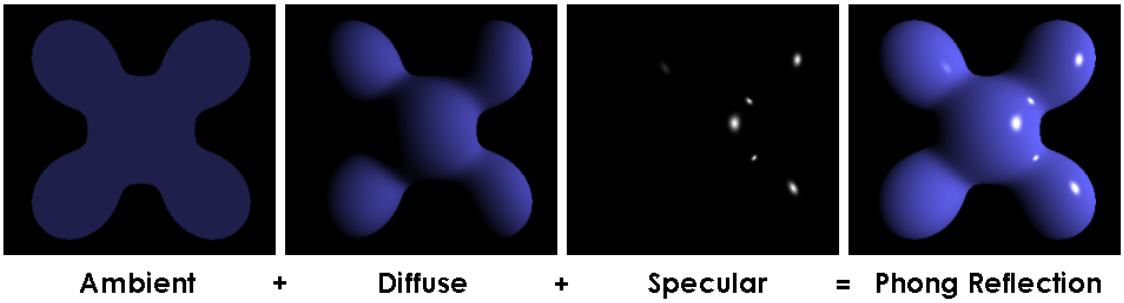


Figure 16: Phong reflection model [26]

The Phong model divides light into three distinct components: ambient, diffuse and specular, as seen in Figure 16. All three components consider the color of the surface as well as the color of the light affecting the surface.

The ambient component is the overall illumination of the scene. Thus, it makes sense to directly tie the ambient component to the ambient light. Because ambient light applies evenly to the entire scene, the ambient component is independent of any others lights and the camera.

The diffuse component is the primary reflection of light of the surface. Any virtual light affecting the object affect the diffuse component. The diffuse component calculation uses both the normal of the surface and a vector from the surface to the virtual light. The position of the camera does not affect the diffuse component.

The final component in the Phong models is the specular component. This approximates shiny reflections off a surface. An object with a high degree of specularity, such as polished metal object, has stronger highlights than an object painted in matte black. As with diffuse component, the specular component depends on

both the light vector and the normal of the surface. However, specularity also depends on the position of the camera. This is because looking at a shiny object from different angles change the perceived reflections.

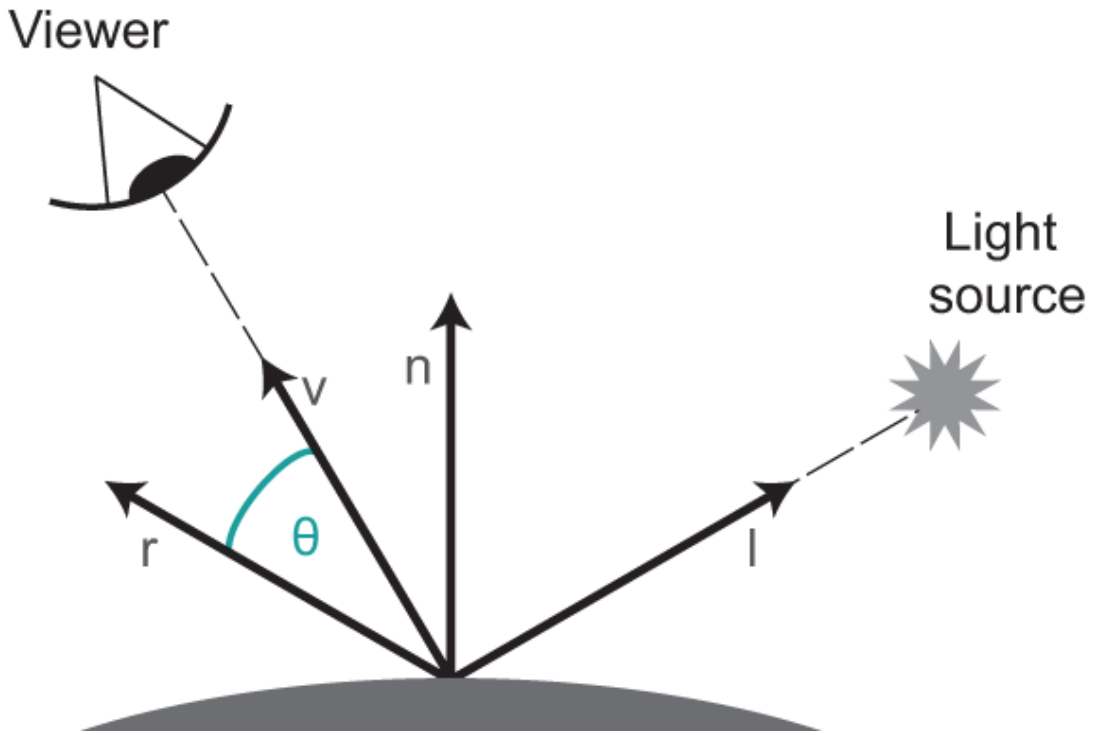


Figure 17: Diagram of Phong reflection calculations [29]

Figure 17 shows the Phong reflection from a side view. Computing Phong reflection requires a series of calculations that include several variables:

- $\hat{n}$ - Normalized surface normal
- $\hat{I}$ - Normalized vector from the surface to the light source
- $\hat{v}$ - Normalized vector from the surface to the camera(eye) position
- $\hat{r}$ - Normalized reflection of -$\hat{I}$ about $\hat{n}$
- α - Specular power(which determines the shininess of the object)

In addition, there are colors for the surface:

- K_a - Ambient color
- K_d - Diffuse color
- K_s - Specular color
- i_a - Ambient Light Color
- i_m - Light Color of light source

In the Phong reflection model, you calculate the light applied to a surface as follows:

$$I_p = K_a i_a + \sum((K_d(\hat{I} \cdot \hat{n}) i_m + K_s(\hat{r} \cdot \hat{v})^a i_m)) \quad (1)$$

It should be noted that the diffuse and specular component for a 3D object is computed for all virtual light sources in the scene, but there's only one ambient component. In addition, the $\hat{n} \cdot \hat{I}$ calculation when is greater than zero ensures that the light affects only surfaces facing the light.

In this thesis, our foveated ray-tracing rendering technique requires the use of a local illumination model in order to find the global illumination for each pixel. To calculate the global illumination for each pixel, we added the local illumination for each object that a ray has intersected as its traveling in the 3D environment. In order too calculate the local illumination we used the Phong model, where for each object that has intersected we calculated the contribution of each light source and estimate the shiny reflections of the 3D object.

2.5 Ray-Tracing

2.5.1 Introduction

The algorithm of direct imaging (like rasterization) are working on the based model which must be represent to the screen filling with colors in random position to the image register. Generally, they synthesize the image running from the 3D environment to the screen 2D environment. The removal of hidden surface is happening to the screen 2D environment with the z-buffer algorithm. Ray tracing is a single general and flexible design algorithm, which is synthesize the image from the 2D environment to the 3D environment.

In Ray tracing, we follow the path along a ray starting from the center of view of the camera, passing through each pixel and interact with the 3D scene to render what the viewer is seeing to that direction [6]. When a ray interact with object surface it absorbed, reflect, refract or weaken based on the material properties of the geometry. The removal of the hidden surfaces is happening to 3d space, where the object exist, instead of the screen and that is an inherent function of ray tracing, as a ray meet first surfaces close to the camera when is traveling to the 3d world.

The idea for tracing the path that light travel and the calculation of its behavior as it interact with the surfaces of different material was exist before the age of computer graphics. The theory of transmission of electromagnetic waves, but especially optical geometry and the laws of reflection and refraction, provided the background for physics studying interaction of light and objects before the confection of ray-tracing as algorithm in the early 80s.

In his general form, the direct imaging in real-time doesn't relate the coloring and the shading of a surface area with the existence of other object in the same environment. The display of shadows, reflections and refraction's light in the objects of the environment must be calculated separate and integrate as a color information in the calculation of the local lighting model when scanning the scene. On the other, ray tracing integrate all the calculations with include the direction of transmitting light in single elegant recursive algorithm: recursive ray-tracing.

In this thesis, the ray-tracing algorithm is used to calculate the global illumination of the 3D environment and to produce accurate hard shadows but at the great cost of real-time performance. We use foveated rendering in order to increase the performance of ray tracing, which reduces the quality outside the foveal region by rendering the 3D environment based on the way the human eye works.

2.5.2 Recursion Ray-Tracing Algorithm

The idea for casting rays from the camera to the scene for the construction of an image with the correct depth without the use of the Z-buffer algorithm was first introduced by Appel [6], Goldstein and Nagel [7]. A complete method for the recursive ray-tracing, where rays are reflected and refracted in the scene was proposed later by Whitted [8]. This method combine the previous algorithm, where they send primitives rays form the camera to the scene until they find an intersection with a surface. When they find an intersection they shade the intersection points base on local model of light with they recursive born of new rays from these points.

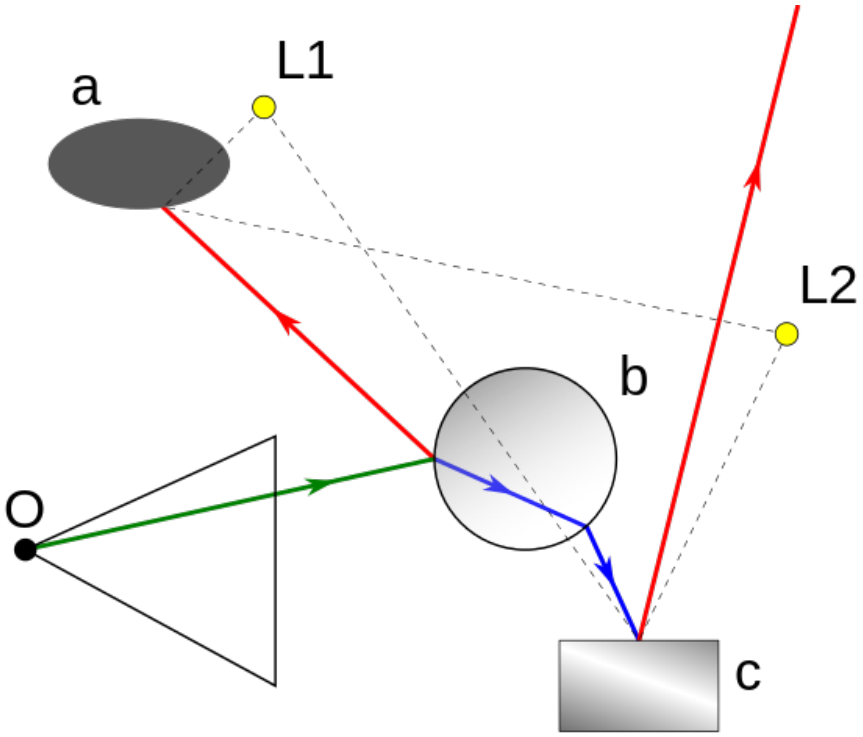


Figure 18: Recursion Ray-Tracing Algorithm in a scene. Green ray is the first ray shoot form point O. Red, blue and dashed lines are rays generated after intersect with a surface. Red rays are reflect from geometry b and c . Blue Rays are refracted form geometry b. Dashed Rays are rays cast toward the light source L1 and L2.

The Ray tracing algorithm is simple: For each pixel a ray is generated (primitive ray) with starting point the camera where pass through the center of the pixel. The ray is looking for intersection with the scene geometry in order to find the closest intersection of ray with scene based on the starting point, as seen in Figure 18. When a valid intersection point has been detected, a local lighting model is applied on to be shaded base on the light sources that are visual contact with the intersection point. If an intersection is not detected, the ray takes the color of the background. If the material of the surface where the ray stops is transparent, a second transparent ray is generated from the intersection point. If the surface is reflective, a second ray is generated towards the direction of perfect reflection. Both the secondary rays are treated as the primitive ray meaning from the point of intersection we look for new intersection with the scene geometry. In case of hitting a surface, shading is calculated again with a local illumination model and new rays are generated based on the material of the surface.

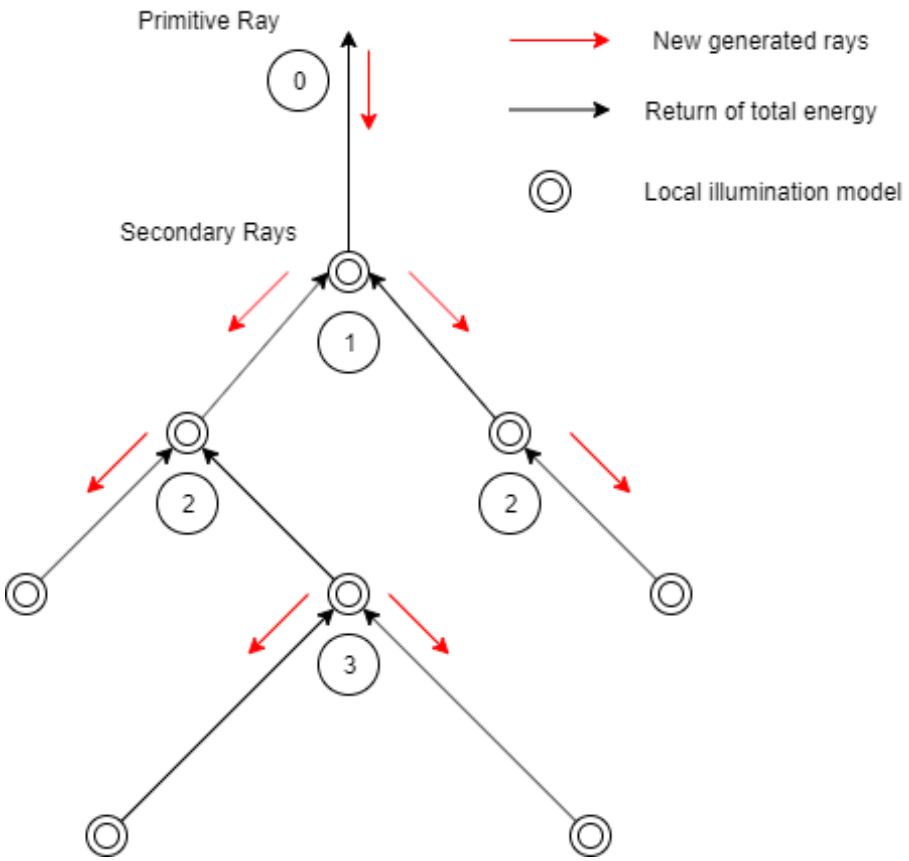


Figure 19: Diagram of recursive ray-tracing algorithm

Each time where a ray meet a surface, the local color for the intersection points is calculated. This color is the sum of local color for the lighting model and the contributions of color returning from the reflected and refracted rays which start from this intersection point. Therefore, each time where the algorithm return from a recursive step is carrying the cumulative color where have been calculated from this recursive step and below as seen in Figure 19. This color is added to the local lighting color of the above level depending on the value of the refraction and reflection factors and is forward to the upper level. After all recursives calls are returned, the final color is what the camera perceives in the direction of primitive ray, which is assigned to the corresponding pixel.

The depth of recursion, meaning how many new rays will be generated, depends on three factors. Initially, if the ray hit a surface where is not transparent or reflective then no new rays are generated. Second, if the contribution of rays has a significant decrease there is no point to continue to accumulate light for the specific path after there will be a not meaningful contribution in the final color of the pixel where the path starts. Finally, to avoid an uncontrollable state of generated rays in very reflective or transparencies environments, is usually determined a specific max depth of recursive.

In this thesis, we implemented a recursive ray tracing process to render our 3D environment with realistic lighting. We used recursive ray-tracing as an easy way to create new rays based on the surface of a 3D Object. Our foveated rendering

technique is applied on the primitive rays that are sent through the pixels in order to cutoff some rays based on the way the human eye works. The secondary rays that are created based on the 3D object surface remain unaffected by our process. The secondary rays are created based on the Phong model that we implemented, where we create a ray toward each light source (shadow ray) to see if any other object is intersected between the point the new ray was generated and the light source. In addition, in order to calculate the reflections from other surfaces we generate a ray from the point that our ray has intersected and we trace it similarly to a primitive ray.

2.5.3 Accelerated structures

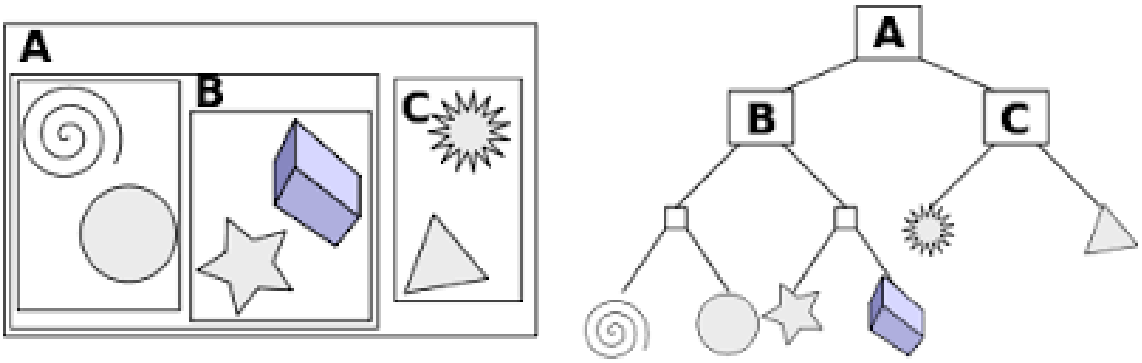


Figure 20: Graph example of Bounding Volume hierarchy - BVH [31]

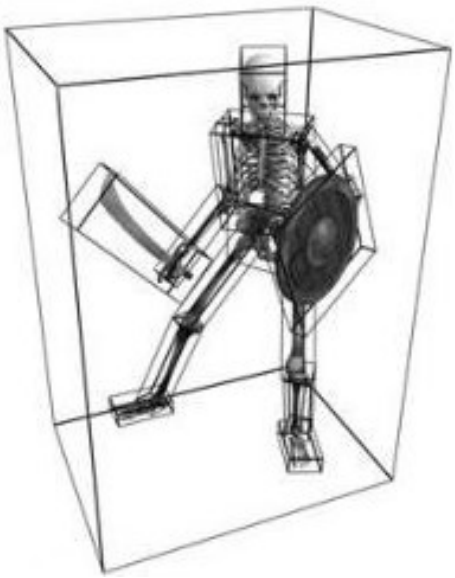


Figure 21: Graphical example of Bounding Volume Hierarchy - BVH

We can benefit from the Hierarchical organizing of a scene to accelerate the process of checking for intersections of a ray with the scene. Instead of an exhaustively checking for intersections of ray with each geometry element of scene we check first

the ray with an organized structure scene. That can be a bounding volume hierarchy, a grid or a combination of these forms of representation (Figure 20,21).

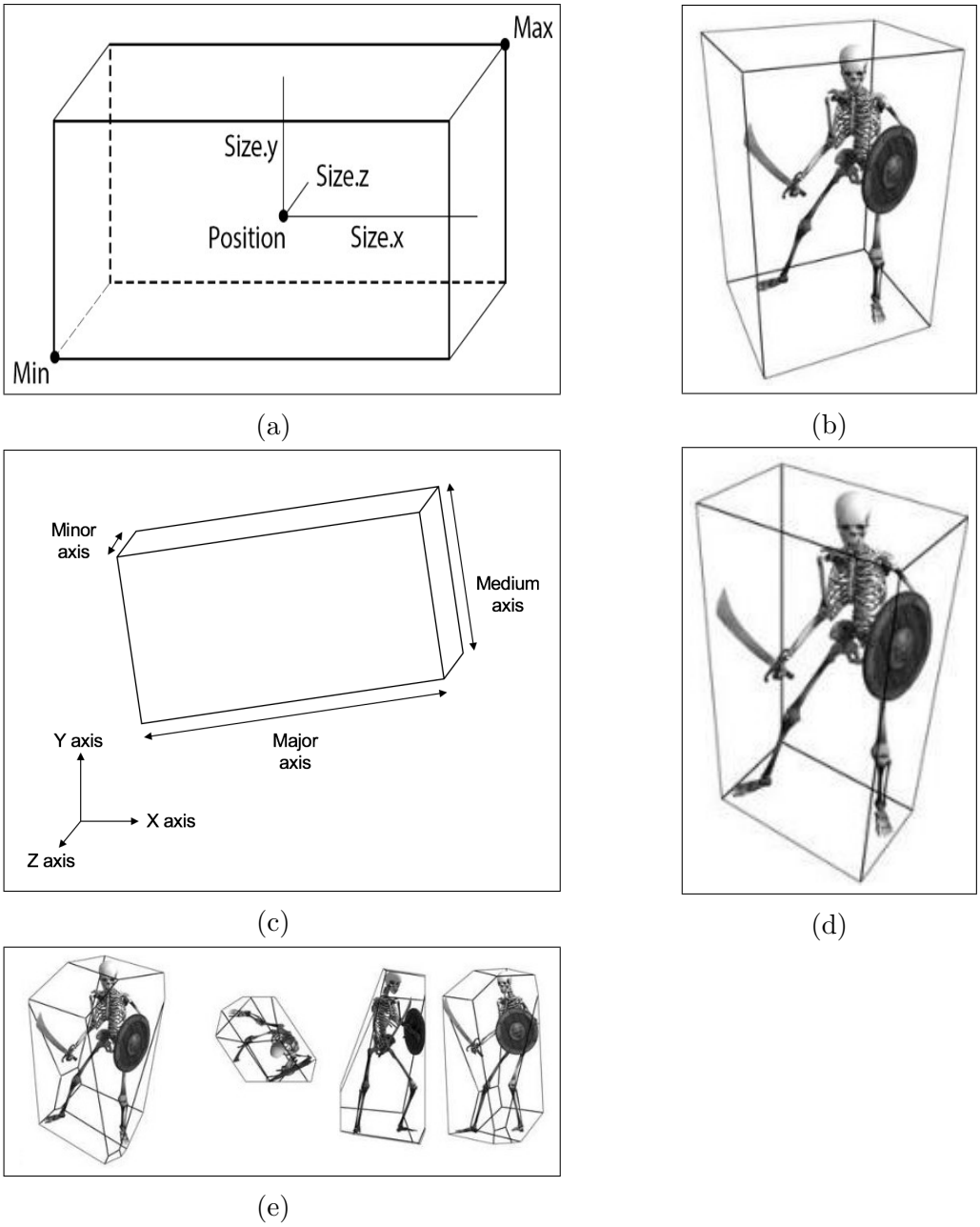


Figure 22: Bounding Volume Hierarchy Structures (a) Axis-Aligned Bounding Boxes - AABB (b) Skeleton example Axis-Aligned Bounding Boxes - AABB (c) Oriented Bounding Boxes - OBB (d) Skeleton example Oriented Bounding Boxes - OBB (e) Skeleton example Bounding Slabs - BS
[28]

The idea of checking a ray for intersection with a simple bounding Volume (meaning a box) which include other geometry elements instead of checking for intersections with all geometry elements in the scene was introduced by Whitted and Clark [8],[9]. For speeding-up the process of ray-tracing the most common type of bounding volume is axis-aligned bounding boxes (AABB), oriented bounding boxes (OBB)

and bounding slabs [10], as seen in Figure 22. The restriction of AABB is to be parallel with main axis but OBB doesn't have this restriction, the last can bound objects with letting smaller empty space, if we choose carefully the direction of them. If we replace the three levels perpendicular to each other that form the box of an OBB, with a number of pairs of parallel planes, the geometry is enclosed by a sum of bounding slabs, which provide even less space in the bounding volume, as seen in Figure 22e).

When the scene is organize in the form of graph, the bounding volume of each node can be use to take a first result of the intersection of ray with the geometries they include [11]. In the case of positive result, the control is done retrospectively at the child nodes. At the level of the hierarchy leaf the geometry elements are all check based on the logic of the basic ray tracing algorithm, after all geometry elements belong to nodes where they don't intersect with the ray have been reject.

An important factor which affects the performance of the bounding volume hierarchy as a mechanism of prime rejection of intersections of rays, is the empty space inside on them. A scene with a big bounding volume tend to let enough empty space between objects.This result in many rays intersections with the bounding volumes without hitting the actual geometry of the scene. That's the reason why bounding slabs are are more efficient in shape compare to the equivalent AABB. Goldsmith and Salmon[12] also showed that the intersections of ray with the bounding volumes where terminate faster when the bounding volume have the minimum possible surface.

Nevertheless, whether fewer rays intersect with a bounding volume is not the only criterion for which type of bounding volume will be choose for bounding an object or a node. The computational complexity of the intersection of a ray with the geometric also plays a very important role, due to the huge amount of rays that we need to check in ray tracing. The search for intersections in AABB is relatively cheap in terms of computational cost. Even in the case we use OBB for bounding volume, we can transform the rays into the object local coordinate system and then check for intersection as if we had AABB.

A different approach for the speedup of ray-tracing is a grid structure. The space which include the scene is fragmented in a number of voxels, which are parallelepiped oriented towards axis. Each of them maintain a reference towards to all geometric elements that are inside or intersected with the voxel. When a ray is cast, it calculate which voxels are intersected with her toward the direction which is traveling and is looking for intersections of the geometric elements belong to the voxel he interact. When a ray enters in the first voxel it meet, we check for intersection with all geometric elements that voxel has or alternatively the process is repeated recursively, if the voxel contains another hierarchy from smaller subdivisions of it. If an intersection not found in that voxel then we visit the next one.

A major benefit to ray-tracing is that when it uses non-overlapping voxels resulting from a uniform distribution of space is that it perform a classification of intersections at the voxels level if it is run when the ray is generated. When the

closest intersection is found in one of the voxels, the search may end. This gives a significant boost in speed to the grid structure in relation to a bounding volume hierarchy. It should be noted, however, that the voxels of the grid structure are filled with reference to the geometric elements and that takes some time since they have to find what elements are the reference in the voxels. The fact that a pre-process is needed to create the grid needs to be taken into account if we plan to design scenes with many moving objects. Dynamic scenes require recalculation of the grid structures in every frame.

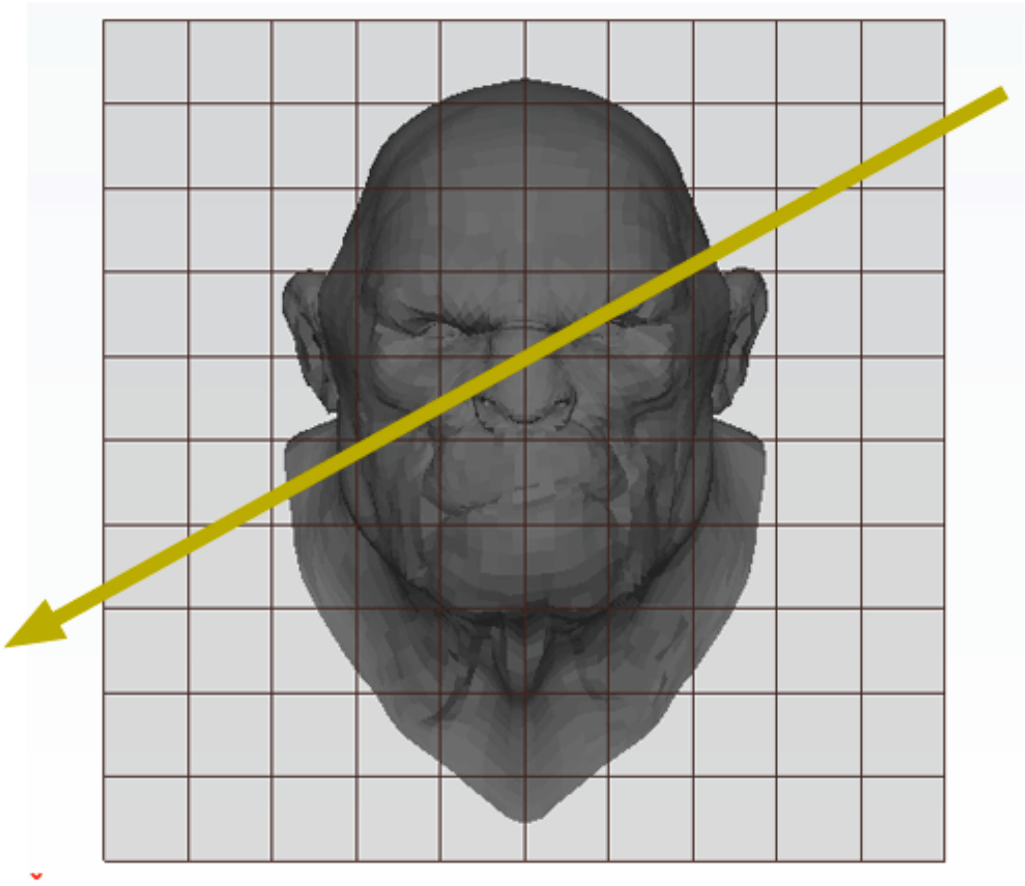


Figure 23: Uniform Distribution [18]

The simplest form of grid for ray-tracing is the uniform distribution of space, where it contain the scene and split it in voxels of same dimensions, as seen in Figure 23. First, we pre-process all the geometric elements to decide which voxels intersect with them. In each voxels which contain or intersect with a geometric element a reference is added in that. In the phase of ray-tracing, the rays passing through a voxel are detected and after the geometry elements contained in the voxel are checked for intersections.

The number of voxels and they size of them, play a significant role in their performance of uniform distribution. With few voxels but big in size we take fewer

intersections between voxels and rays meaning smaller probability a geometric element to intersect with multiple voxels therefore less unnecessary controls. On the contrary, with smaller voxels the number of geometric elements which are include in them are limited, giving a shorter intersection time with each voxels.

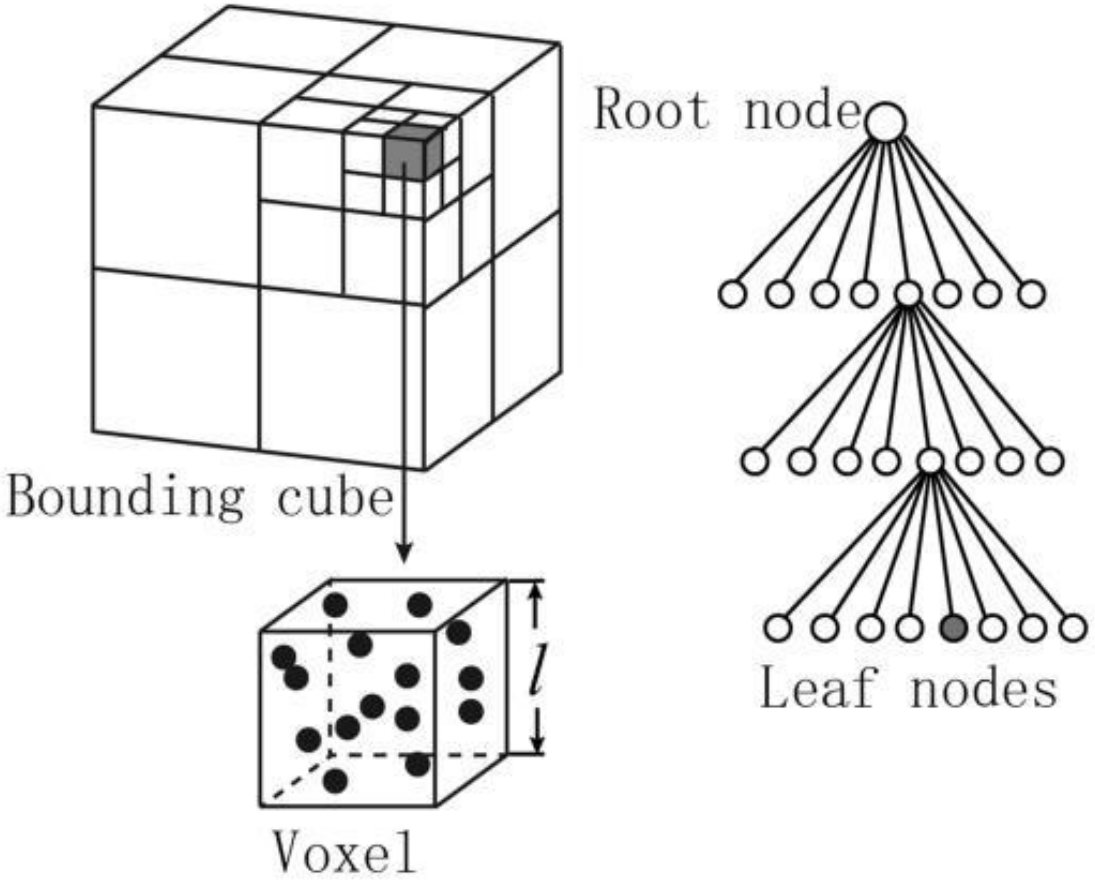


Figure 24: Example 2 of Octree [30]

As mentioned, the voxels of a grid structure can be hierarchically subdivided. Usually, one reason we want something like this is to end up in voxels with an even distribution reference in geometric materials. In Figure 24, we see that many voxels remain empty compare to others which contain many elements. That happen because of the uneven distribution of the elements in the space.

The most commonly organized hierarchy shape for grid structure in ray-tracing is the Octree [13]. the first-level voxel is evenly divided into eight cells. The voxels that have been produced are not divided if they don't contain geometric elements else they divided with the same logic as seen in Figure 24. The division stops when we exceed a maximum number of divisions, or when the number of geometric elements containing a cell is small enough.The maximum number of divisions specify the depth of the tree. As opposed to uniform partitioning, the detection of intersections with the voxels of Octree is not weighed. However, the distribution of controls on the hierarchical structure leafs is much more uniform.

In this thesis, the accelerated structure is used to increase the performance of the ray tracing algorithm by reducing the intersections of a ray with the 3D environment, which also increases the overall performance of our foveated ray tracing technique. In order to increase the performance, we used a bounding volume hierarchy. The alternative would be to use a grid structure over bounding boxes, which is not currently supported by OptiX. Even without using grid structures, which would be the optimal accelerated structure for our scene, we still achieved better rendering times compared to execution without an accelerated structure.

2.6 Gaussian-Blur

In image processing, a Gaussian blur is the result of blurring an image by a Gaussian function. It is a widely used effect in graphics software, typically to reduce image noise and reduce detail. The visual effect of this blurring technique is a smooth blur resembling that of viewing the image through a translucent screen, distinctly different from the bokeh effect produced by an out-of-focus lens or the shadow of an object under usual illumination.

Mathematically, applying a Gaussian blur to an image is the same as convolving the image with a Gaussian function. This is also known as a two-dimensional Weierstrass transform. By contrast, convolving by a circle (i.e., a circular box blur) would more accurately reproduce the bokeh effect. Since the Fourier transform of a Gaussian is another Gaussian, applying a Gaussian blur has the effect of reducing the image's high-frequency components; a Gaussian blur is thus a low pass filter.

The Gaussian blur is a type of image-blurring filter that uses a Gaussian function for calculating the transformation to apply to each pixel in the image. The formula of a Gaussian function in one dimension is:

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}} \tag{2}$$

In two dimensions, it is the product of two such Gaussian functions, one in each dimension:

$$G(x) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{3}$$

where x is the distance from the origin in the horizontal axis, y is the distance from the origin in the vertical axis, and σ is the standard deviation of the Gaussian distribution. When applied in two dimensions, this formula produces a surface whose contours are concentric circles with a Gaussian distribution from the center

point. Values from this distribution are used to build a convolution matrix which is applied to the original image. Each pixel's new value is set to a weighted average of that pixel's neighborhood. The original pixel's value receives the heaviest weight and neighboring pixels receive smaller weights as their distance to the original pixel increases. This results in a blur that preserves boundaries and edges better than other, more uniform blurring filters.

Applying successive Gaussian blurs to an image has the same effect as applying a single, larger Gaussian blur, whose radius is the square root of the sum of the squares of the blur radius that were actually applied. Because of this relationship, processing time cannot be saved by simulating a Gaussian blur with successive, smaller blurs the time required will be at least as great as performing the single large blur.

Gaussian blurring is commonly used when reducing the size of an image. When down-sampling an image, it is common to apply a low-pass filter to the image prior to re-sampling. This is to ensure that spurious high-frequency information does not appear in the down-sampled image (aliasing). Gaussian blurs have nice properties, such as having no sharp edges, and thus do not introduce ringing into the filtered image.

In this thesis, the Gaussian blur is used as a post-process effect to cover the imperfections that are produce from our foveated ray-tracing technique. The reason is that foveated rendering reduces the sampling of rays outside the foveal region in order to reduce the rendering time process. By reducing the sample rays, we need to interpolate a single ray's value over sets of pixels, which results in reduced resolution to our produced image. By applying a Gaussian blurring mask to the lower resolution outside the Foveal we can cover imperfections in the sub-sampled image reducing the chance that it will distract the users while observing the 3D environment.

2.7 Euclidean Distance

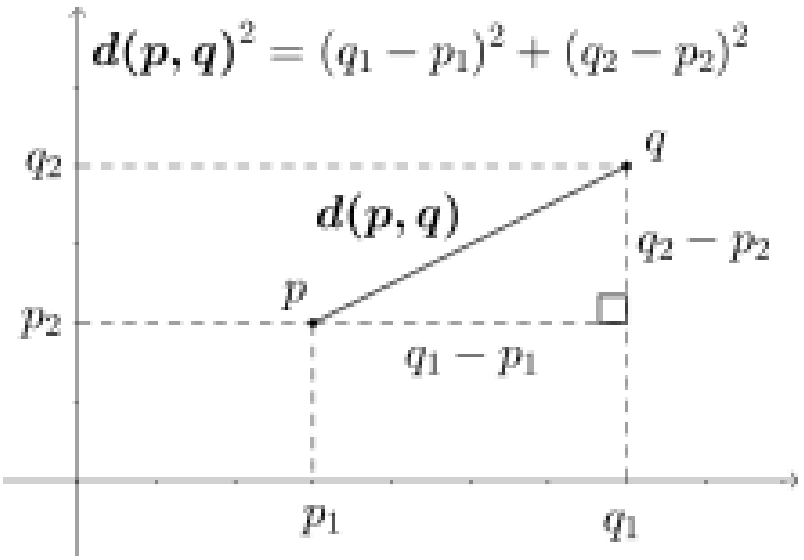


Figure 25: Euclidean Distance [27]

Euclidean distance is the straight-line distance between two points in two- or three-dimensional space. For two dimensions, if p = (p1, p2) and q = (q1, q2) then the distance is given by:

$$d(p,q) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2} \tag{4}$$

In this thesis, we used the euclidean distance to calculate the distance between the pixel that the eye is fixating and for all other pixels of the display. The reason we need to find the distance is because we need the value to calculate the angle for each pixel from the center of the foveal region.

2.8 Visual Angle

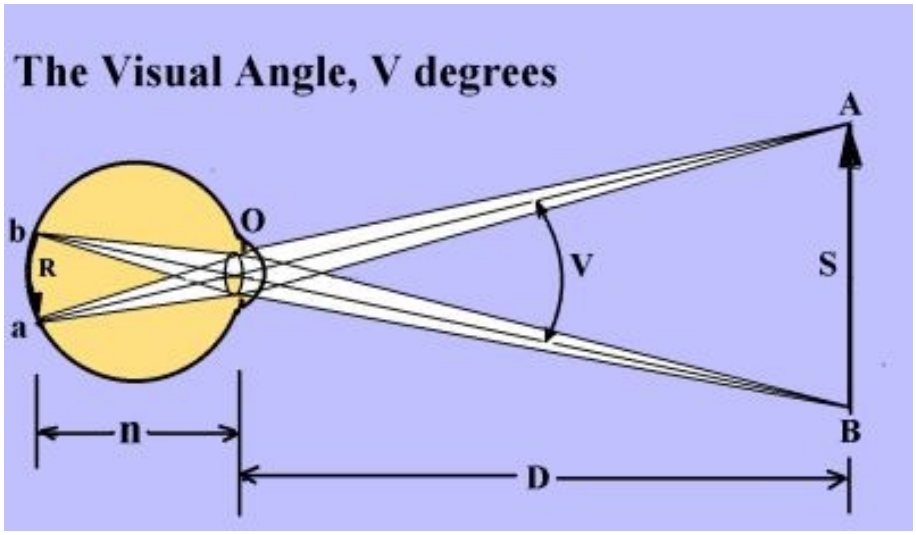


Figure 26: Visual Angle

The visual angle is the angle a viewed object subtends at the eye, usually stated in degrees of arc. It also is called the object's angular size. The Figure 26 shows an observer's eye looking at a frontal extent that has a linear size S, located in the distance D from point O. For present purposes, point O can represent the eye's nodal points at about the center of the lens, and also represent the center of the eye's entrance pupil that is only a few millimeters in front of the lens. The three lines from object endpoint A heading toward the eye indicate the bundle of light rays that pass through the cornea, pupil and lens to form an optical image of endpoint A on the retina at point a. The central line of the bundle represents the chief ray. The same holds for object point B and its retinal image at b. The visual angle V is the angle between the chief rays of A and B.

$$V = 2\arctan(\frac{S}{2D}) \tag{5}$$

In this thesis, we used the visual angle to calculate the angle that is created on the eye between the pixel that the eye is fixated on and all other pixels. The visual angle is used in calculating in which region each pixel belongs (foveal, periphery or outside). In addition, by knowing the angle for each pixel we can use the visual probability model we created to calculate a probability for each pixel based on their angle.

2.9 Foveated Rendering in Virtual Reality

In this section, we include an overview of related research in foveated rendering for rasterization and ray-tracing. The foveated rendering presented by Guenter et. al. [1] is a system rendering three eccentricity layers (inner/foveal, middle and outer layer), representing the angular distance away from the central gaze direction, around the tracked gaze point, as shown in Figure 27. The inner layer has the smallest angular diameter and is being rendered at the highest resolution and finest level of detail (LOD). The middle and outer layer have larger angular diameters compared to the inner layer, rendered at progressively lower resolution and LOD while also updating half the temporal rate of the inner layer. After the process has ended, they interpolate these layers up to native display resolution and smoothly blend them.

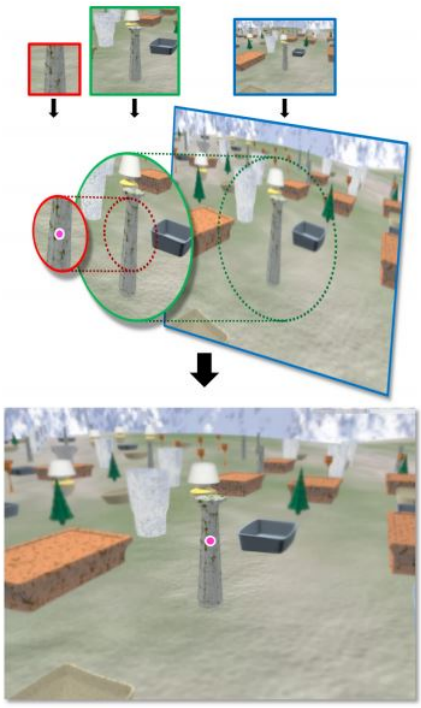


Figure 27: Foveated rendering by Guenter et. al. [1]

In that work they faced two main difficulties, e.g., aliasing in the periphery and system latency [1]. The rendering process relies on point sampling, however, their system has reduced sampling density in the periphery. The use of super sampling to bring back to native resolution is eliminating the computational advantage of foveated rendering. To solve this, they brought forward amortizing the cost of anti-aliasing for peripherals layers over multiple frames, using multi-sample anti-aliasing (MSAA), temporal re-projection and temporal jitter of the spatial sampling grid. The first system resulted in high latency and the update of the fovea region from low to high resolution was visible. Employing anti-aliasing, controlling latency and selecting the right size and resolution layers for the eccentricity layers, they manage to create a seamless image of uniformly high resolution and accelerate the graphics

computation by a factor of 5-6.

The problem with the approach of Patney et. al. was that they focus on reducing computational cost without explicitly identifying and minimizing perceptible artifacts resulting to head and gaze temporal aliasing which is distracting, breaking the sense of immersion. Patney et al. [14] attempted to solve this problem by employing a post process Gaussian blur with a progressively increasing filter width based on eccentricity. This resulted in another issue because large radius blurs induce a sense of tunnel vision, e.g., peripheral pixels appearing blurry. The tunnel vision problem arises from missing contrast, as filtering typically reduces image contrast. They solved this problem by enhancing contrast which restored most of the peripheral detail. The result allowed them to double the rate of blurring the peripheral vision before inducing any noticeable tunnel vision.

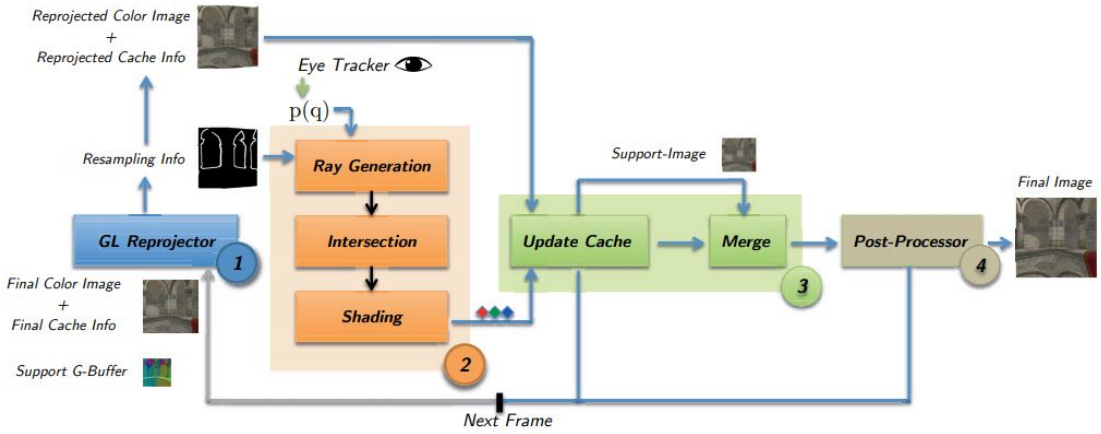


Figure 28: Foveated rendering system for ray-tracing by Weier et. al. [2]

Research analyzed above [1] [14] introduced us to the generic foveated rendering principles and how to apply these principles for rasterization. It is unclear, though, how foveated rendering would be applied to other rendering methods such as ray-tracing. Weier et. al. [2] proposes a foveated system which is using the ray-tracing algorithm, as shown in Figure 28. By using the dynamic adaptation of rendering based on the user's gaze (foveated rendering) they reduce the computational cost for the rendering process and enable the ray-tracing method to produce high quality images fast enough. They reduce the sample density by adopting the ray generation to the foveal receptor density. To limit the detection of visual artifacts, they reconstruct images at high quality to improve perception. To reconstruct the image, they put forward two methods. The first is by using a support image that is guaranteed to sample the full scene using a lower uniform resolution. The second method is taking information from re-projected frames to improve the quality of the reconstructed image.

Our technique compared to Weier et al. [2], as seen in Figure 28, is much simpler because we don't keep records of previous frames or using a G-Buffer and the only input we give to our ray generation program is the information about the eye fixation points on the two displays of our HMD. Also, Weier et al. used a liner probability model to generate rays as opposed to the model we used, which is based

on the visual acuity of the human eye. We simply generate rays using our model and employ one extra condition stating that we generate a ray if our model hasn't generated a ray for every four pixels in the periphery region or every sixteen pixels in the outside zone. Furthermore, the pixels that have remained blank are duplicated using the values of their neighboring pixels, and by using a Gaussian blurring effect we cover the imperfections that have been produced by our method. We expect that our technique will reduce the number of rays per second, which will lead to an increase on the frames per second, while the users won't be able to notice any difference between our foveated ray-tracing rendering technique and full ray-tracing rendering.

2.10 Augmented Reality

Augmented Reality is the projection of digital artifacts on real environments. Augmented reality aim to enhance reality compare to Virtual reality where it aim to immerse the user in a virtual environment. Current Augmented Reality technology most commonly uses specific design glasses or smartphones to generate the digital artifacts on the real environment. A person using augmented reality equipment is able to look through the device and see the digital artifacts on the real environment and interact with them, usually with hand gestures recognition (Magic leap, Hololens) or by touching the screen of a smartphone.

2.11 Foveated Rendering in Augmented Reality

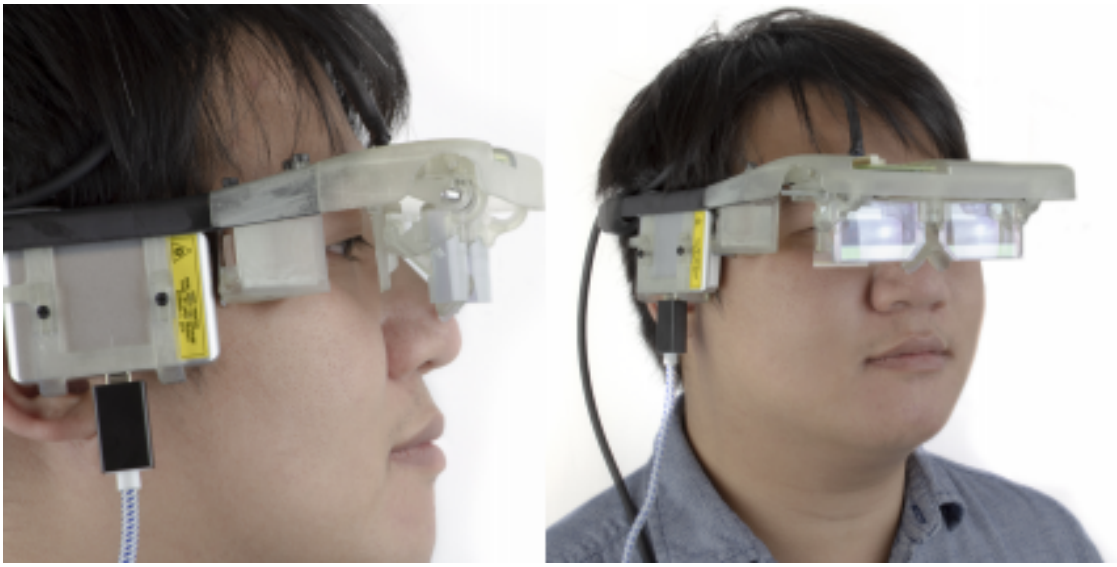


Figure 29: Nvidia wearable Prototype for Augmented Reality. Kim et. al [3]

In this chapter, we include an introduction of related research in foveated rendering for augmented reality displays. Kim et. al.[3] have introduced and analyzed a near-eye foveated display system for augmented reality (see Figure 29), which support a simultaneous wide field of view (FOV) over 100^o, has a high foveal resolution

of 60 circles per degree (cpd) and a large Eyebox of size 12 mm x 8 mm.

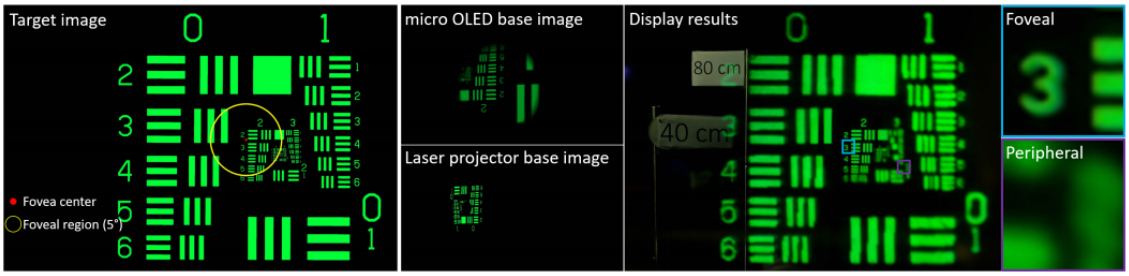


Figure 30: Example of Kim et al. [3] system for a target image and how the system has generated base images for foveal and peripheral displays and the display results.

The main innovation of their system is the use of a holographic element which can change position dynamically driven by gaze tracking, as seen in Figure 29. In order to enable a dynamic position and varifocal system, they used a low latency gaze tracking system, motors, and rendering system. In addition, to achieve a high resolution and a wide field of view for their system they combine a Holographic Optical Element (HOE) and an Organic Light-Emitting Diode (OLED) projector. The phase-conjugate HOE method they developed could produce an element for a 130^o horizontal, monocular FOV, however, the projector would then be blocked by the user's eyelashes and the HOE would be too close to the eye and making them uncomfortable. They have managed to simulate the defocus blur effect for the foveal and periphery area but they haven't simulated the chromatic aberration in the periphery. Furthermore, they haven't yet provide a solution for the focus depth, however they propose creating a specialize gaze tracking network to estimate it rather than predicting focus depth by separately-tracking pupils to estimate focus depth from binocular vergence with this method lacking in accuracy. Finally, the mechanical complexity of their prototype system makes it not feasible for marketing purposes and mass production.

Our foveated ray-tracing rendering technique may be able to be combined with the work of Kim et. al. [3] to reduce the rendering cost of their system and to provide photo-realistic 3D holographic elements by calculating the contribution of virtual and real light of their environment to improve the mixed reality.

3 Software Architecture and Development

3.1 Nvidia OptiX

3.1.1 Overview

Parker et. al. [4] have created the Optix engine which uses a small set of programmable operations in order to implement most ray tracing algorithms. Nvidia achieved this by identifying an abstract model that can be used for almost all ray tracing algorithms. The programmable operations are associated with each ray, where for each ray model a user-defined data structure (payload) has been set.

In this thesis, we used Nvidia Optix to implement the ray-tracing algorithm because it provides us with high performance and an easy implementation of a multi-pass rendering pipeline for our Foveated ray-tracing rendering technique. The OptiX Engine has also been programmed to take advantage of the perofirmance of the newest generation of graphics cards, e.g., the RTX series. The RTX series architecture has a specific type of processing core known as an RT core, which is designed to speed up the intersection of a ray with the 3D environment at the hardware level.

3.1.2 Programs

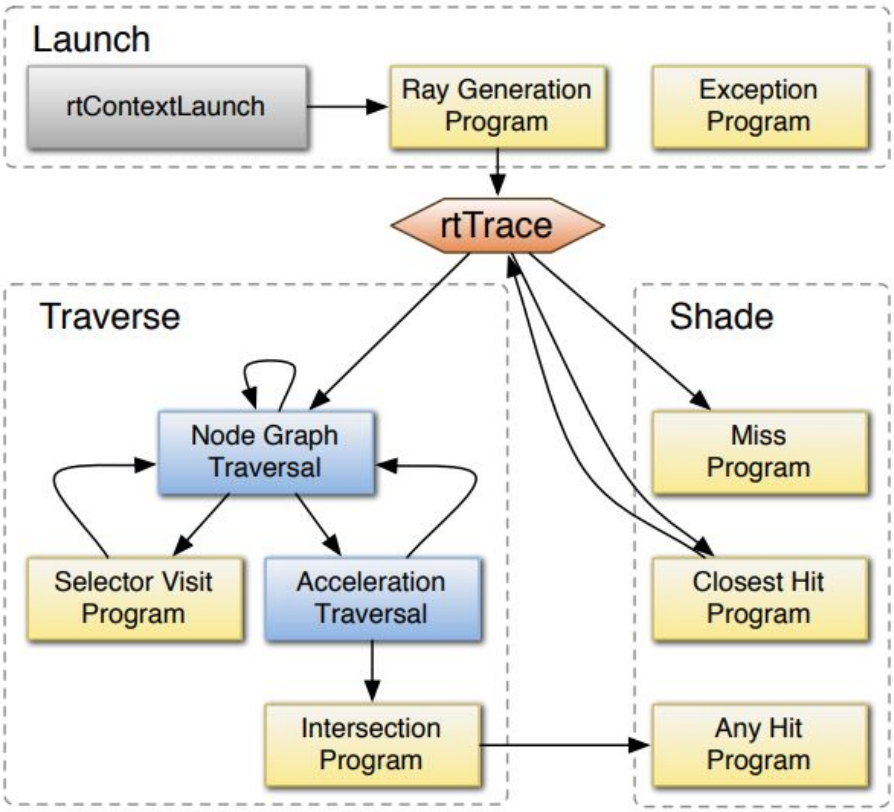


Figure 31: Execution Flow of Optix Ray-Tracing pipeline. Parker et. al. [4]

The OptiX engine has seven different programs: Ray generation program, Bounding box program, Intersection program, Closest hit program, Any hit program, Miss program, and Exception programs. These programs are specified from the users and each of them operates on a single ray at a time. The OptiX engine's core operation is the rtTrace, which is responsible to locate an intersection of a ray and to respond to that intersection, as seen in Figure 31.

The ray generation program is responsible to shoot one or multiple rays and to initialize the process of tracing these rays. The system defines the number of ray generation programs it wants to create (usually for all pixels of a display) and with a single invocation of the rtContextLaunch, it creates them.

The Bounding box program is responsible for calculating the boundaries for each geometry in order to be used for an acceleration data structure, which accelerates the intersection process.

The Intersection program is responsible for implementing the calculations to test if there is any intersect between the ray and the geometry. These calculations are used to find whether and where a ray touches the object. It then computes the normals, texture coordinates, or other attributes based on the hit position.

The closest hit program is invoked when the closest intersection between a ray and the scene geometry has been found. The closest hit program performs operations based on the surface of the geometry that the ray has intersected. These operations, for example, are used to calculate the shading of the 3D object surface, cast new rays according to the geometry surface properties, and store the calculated result data in the ray payload.

The Any hit program is called when an intersection has been found between a ray and 3D object. The difference of the Any hit program with the closest hit program is that it allows the material of a 3D object to participate in the intersection decisions without having to calculate the shade of the geometry. This mechanism is frequently used as an early ray termination for shadow rays.

The Miss program is executed when the ray does not intersect with any geometry. They can be used to implement a background color or an environmental map lookup.

The Exception program in the OptiX engine is invoked when a critical condition on the engine has risen, for example, an overflow in the memory. The exception program can be used to write a special value to an output pixel buffer in order to visualize the critical condition.

In this thesis, we specified almost all the programs of the Optix engine expect the intersection program and the bounding program, which are used in order to create our foveated ray tracing rendering technique.

3.1.3 Scene representation

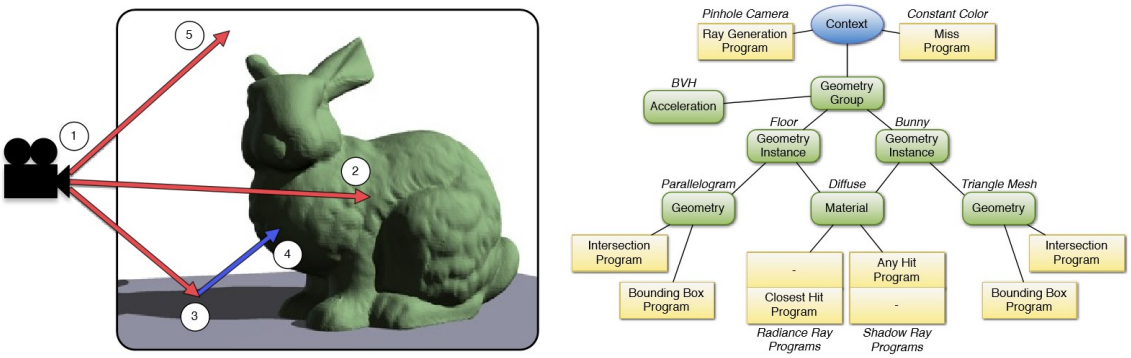


Figure 32: Optix Context Diagram For Scene. Parker et. al. [4]

The Optix engine has a structure graph that can be used to represent the scene information and associated programmable operations, where a container object may call the context and store the scene's information as the head of the graph. As seen in Figure 32, the graph has an amount of different nodes.

We analyzed the main node types that are used in this thesis the Geometry Group, Geometry Instance, Geometry and Material.

The Geometry Group node contains the geometry instance node, material node, and associated acceleration structure, where the geometry instance and material node can have multiple parents to share geometry information.

The Geometry Instance node binds a geometrical object to a set of material objects, as seen in Figure 32.

The Geometry node stores all the geometric primitives and is linked to a bounding box program and an intersection program, where these programs are shared for all the primitives inside the geometry node.

The Material node includes programs called for each intersection that has been found (any hit program) and for the closest intersection to the origin of a ray (closest hit program). In addition, it holds information about the shading technique that has been implemented.

In this thesis, we used the OptiX engine to construct our scene for our foveated ray-tracing rendering application. Our scene is constructed by using many 3D objects, where all our 3D objects are built as sets of triangles. In order to create the scene based on the OptiX engine, we used the geometry provided by the OptiX engine to build the geometry for our 3D objects consisting of triangles. We created some materials based on the need of our scene. Finally, we created the graph of our scene based on the OptiX engine to run our Foveated ray-tracing rendering for this scene.

3.1.4 Acceleration Structures

The OptiX engine includes an interface to control its acceleration structures. This interface can create several structures over different regions of the scene, for example, the scene may have animation components which require an acceleration structure to rebuild the scene every frame. In order to not reconstruct the whole acceleration structure, it creates separate structures for the static regions and the animated regions, which increases the calculation efficiency.

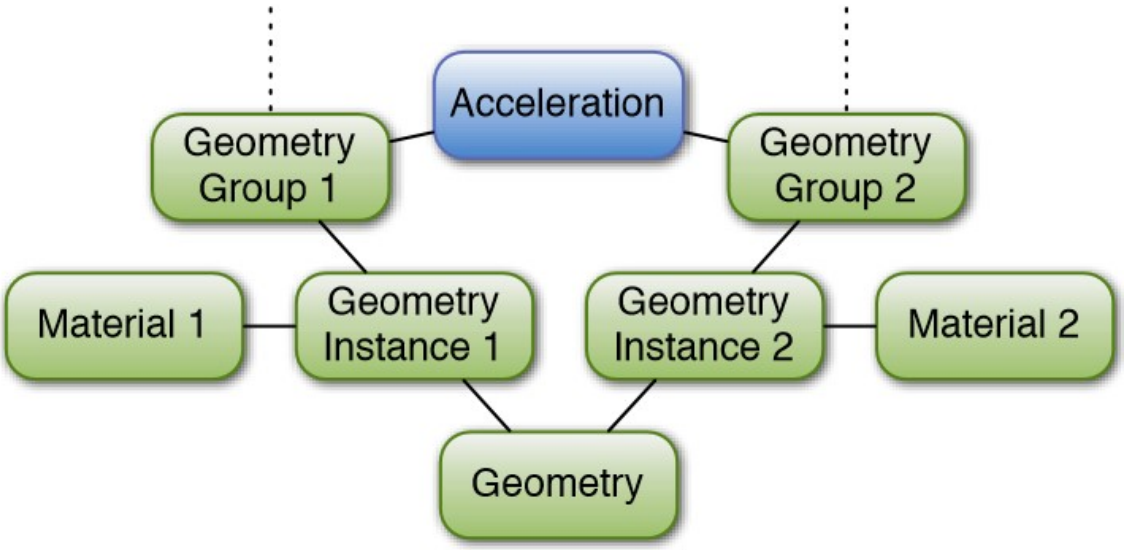


Figure 33: Node graph with instancing. Parker et. al. [4]

The Optix engine has made it possible to attach a single acceleration structure to multiple nodes in order to avoid the construction of the same data structure, as seen in Figure 33. In addition, the Optix engine can add and remove instances at runtime.

There is no single acceleration structure that is optimal for every ray-tracing algorithm under all conditions. The typical tradeoff between the different variants is ray tracing performance versus construction speed, and each application has a different optimal balance. Therefore, OptiX provides many different acceleration structure types and the programmers can select the appropriate one for their application, for example, the LBVH [15] is focused on the construction speed of the scene and the SBVH [16] is focused on the quality of the hierarchy.

An acceleration structure in the OptiX engine is constructed by using the bounding boxes of the referenced geometry. Initially, it used the bounding program for each geometry (see sub-chapter 3.1.2), which is included in the acceleration structure, to calculate the bounding boxes. The building process of the acceleration structure starts after having acquired the bounding boxes. Finally, the constructed acceleration structure data is placed in the device memory to be used for the ray traversal code.

In this thesis, we used the OptiX engine in order to create an accelerated structure for the 3D environment of our foveated ray-tracing rendering application. We used an accelerated structure to reduce the intersections test of a ray with our 3D environment, which increases the performance of our foveated ray-tracing rendering.

3.2 Eye Tracing Software

3.2.1 Overview

The software used regarding to 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.

3.2.2 Viewpoint EyeTracker® Program

Once the program is running, it displays several windows arranged as shown in the below Figure 34.

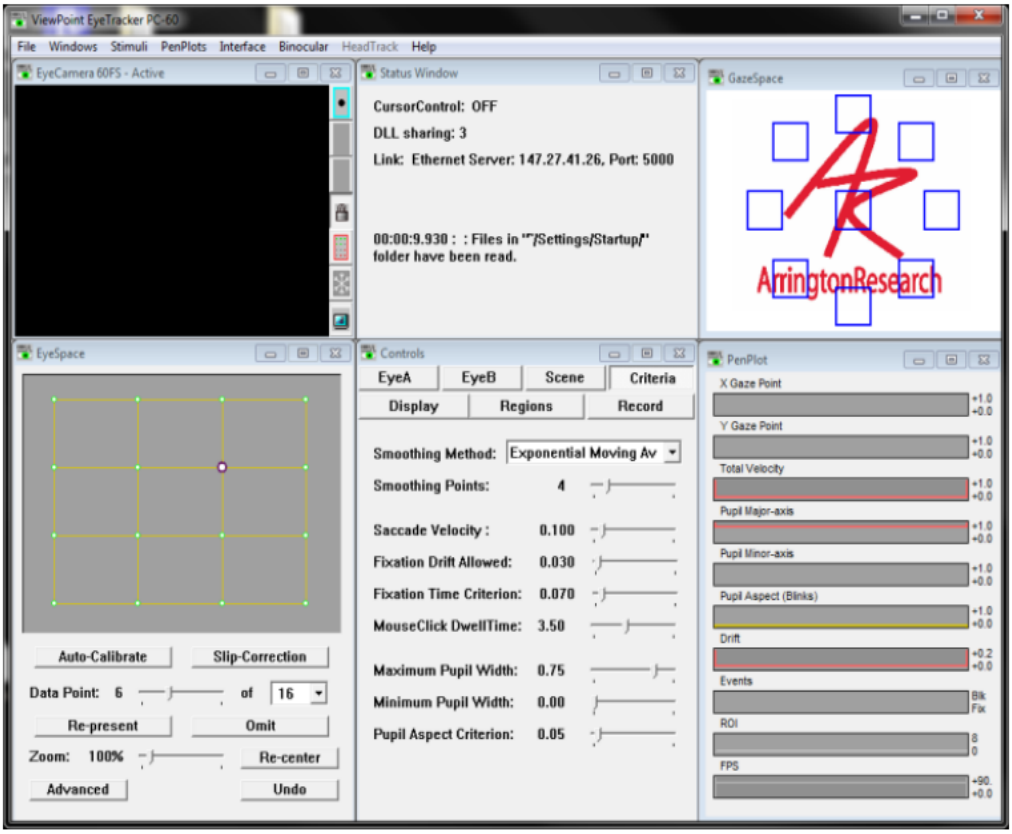


Figure 34: Viewpoint EyeTracker® Software Interface [34]

There is the EyeCamera window, the EyeSpace window, the Status window, the Controls window, the GazeSpace and the PenPlot windows.

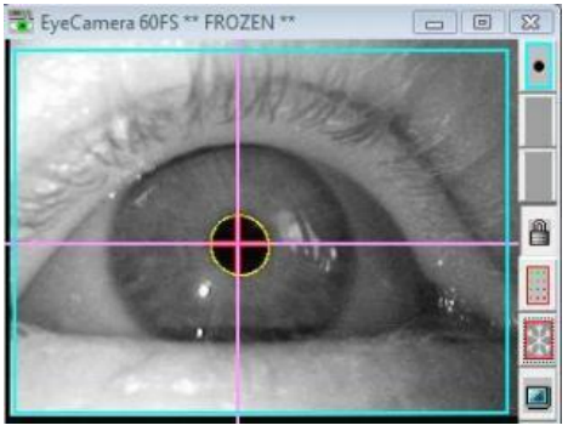


Figure 35: EyeCamera Window [34]

The EyeCamera window displays the video image of the eye and the image analysis graphics, as seen in Figure 35. 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 within the software searches for the pupil and corneal reflection to exclude extraneous reflections and shadows.

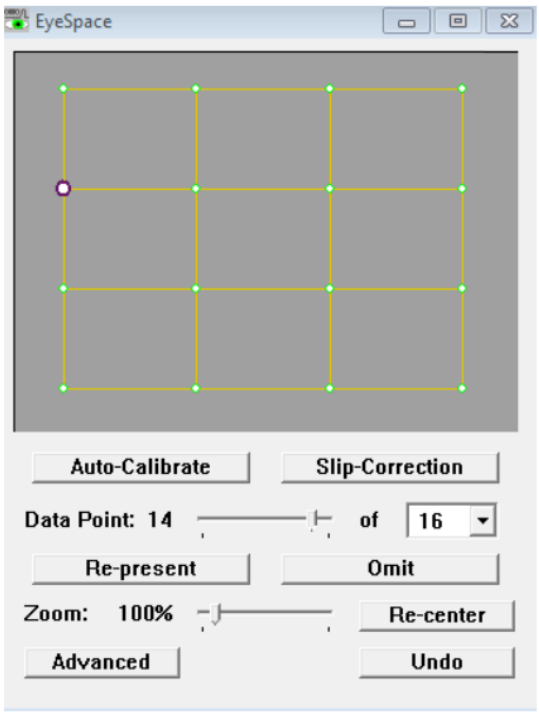


Figure 36: EyeSpace Window [34]

The EyeSpace window corresponds to the geometry of the EyeCamera image, as seen in Figure 36. It's an easy, flexible and intuitive calibration procedure. The developer can decide the number of calibration points to use, their color, and how fast to present them to the subject or stick with the defaults set within ViewPoint. The EyeSpace window provides a visualisation of the calibration mapping so that the developer can quickly and easily tell if the calibration is accurate. Outliers can

be repeated individually if necessary to avoid recalibration. Individual points are easily represented and perform slip correction at any point. There are options to save and reload individual user's calibration settings for use in subsequent trials.

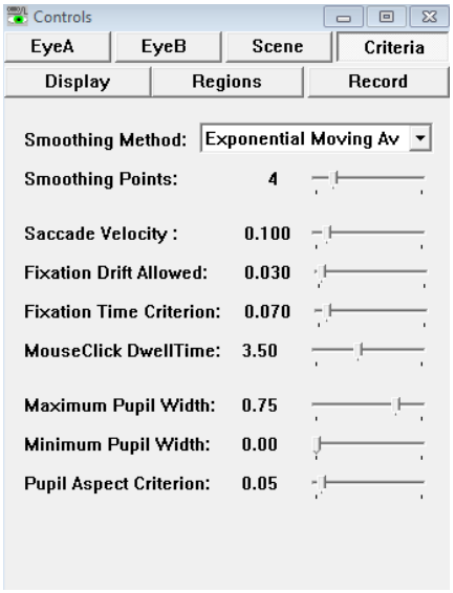


Figure 37: Controls Window [34]

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, as seen in Figure 37. Eye Image quality adjustments can be made and the tacking method can be specified. Also, smoothing and other criteria can be applied to data such as defining parameters to setup the regions of interest and calibration regions as well as adjust brightness, contrast, hue, saturation of the scene as well as open, pause, close data files or insert markers.

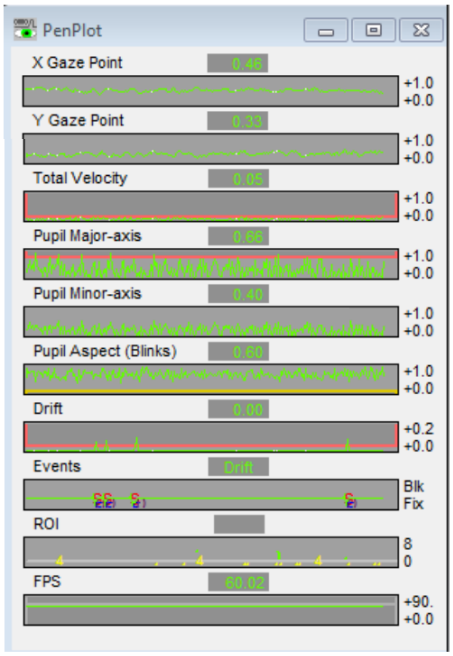


Figure 38: Pen Plot Window [34]

The Pen Plot window displays plots of X and Y position of gaze, velocity, ocular torsion, pupil width, pupil aspect ratio, drift, etc. in real time, as seen in Figure 38.

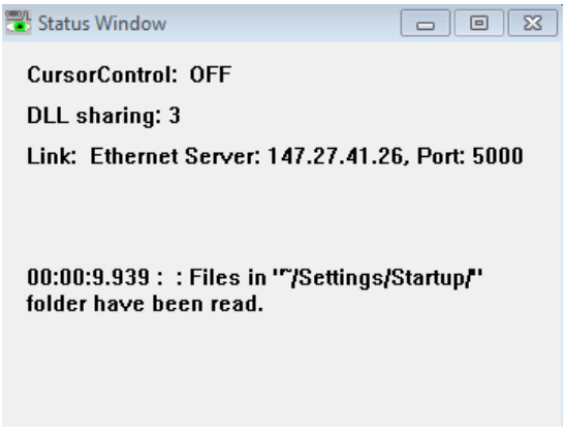


Figure 39: Status Window [34]

The Status window shows details about processing performance and measurements, as seen in Figure 39. Also, it shows the applications that share the same .dll, thus, the threads that the ViewPoint communicates with.

Figure 40: Stimulus Window [34].

The Stimulus window is a new window that pops up when calibration starts, as seen in Figure 40. It is designed to be full-screen, preferably on a second monitor. Upon this window, the subject's calculated position-of-gaze information may be displayed as well as region of interest boxes.

3.3 Eye Tracking Method

In this section, the way the Viewpoint EyeTracker® works in a typical head fixed configuration will be described. The item numbers in this section refer to the components of the system as shown in the Figure below 41.

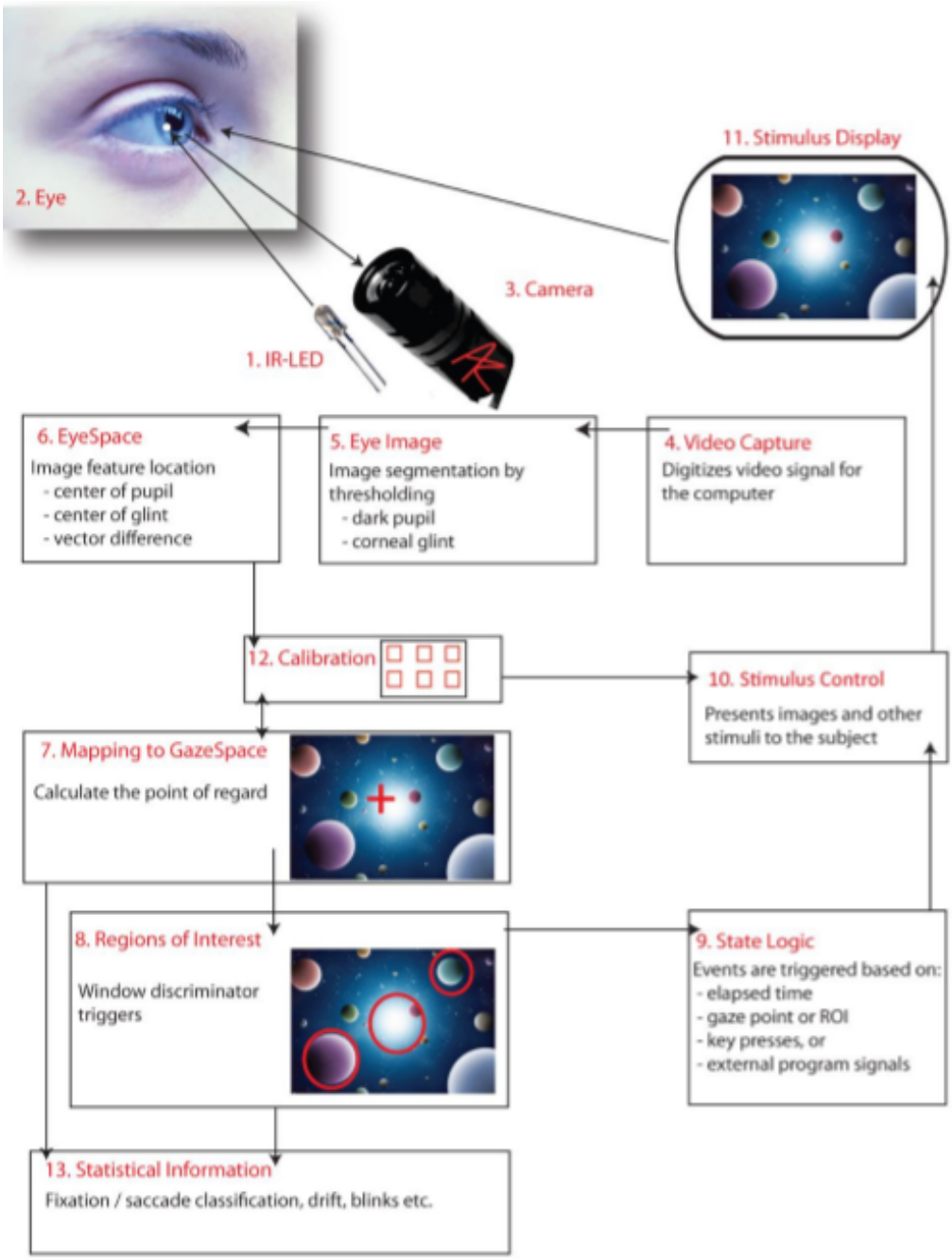


Figure 41: Eye Tracker Diagram [34]

The infrared light source (item 1) serves to both illuminate the eye (item 2) and also 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" effects 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 a computer. 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.

3.3.1 Binocular Method

The eye-tracker used supports both monocular and binocular eye tracking. By default, ViewPoint is set for monocular eye tracking. So, we have to switch from monocular to binocular operation.

In our application, we applied binocular eye tracking in order to render each screen of our HMD using our foveated rendering technique. Both eyes point gaze data are used to map where the user is looking at our virtual scene and render based on our foveated model. Our foveated ray-tracing rendering technique creates three layers in the screens of the HMD which represent the foveal, periphery and outside zone of the human eye. In addition, each layer is rendered different for example the foveal layer is rendered at the best quality that a system can provide but the other zones are rendered with less quality in order to reduce the rendering time.

3.3.2 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, how the eye looks like, when the user 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 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.

In our situation, calibration takes approximately 1-2 minutes to complete and consists of a square green target that is displayed at different locations of the screen on a blank background. The user is warned with a "Get Ready" message on the screen getting the user's attention to start the calibration process. During the calibration process, we must ensure that the pupil is accurately located at all times by monitoring the green dots and the yellow oval, i.e., monitoring the image segmentation. As for the calibration points, we must use at least 9 but tests showed us that best calibration results are provided by using 16 or 20 points; for this thesis we used 20. 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 shown in the EyeSpace window.

Stray calibration data points can be identified and re-calibrated or omitted. The EyeSpace window allows the user to select stray calibration points to be recalibrated. If a point is selected to be re-calibrated, then it will be re-presented on the screen and the user will be asked to look at the center of it; to calibrate. This can be repeated with many calibration data points as necessary. If the calibration points are not rectilinear, for example, there are lines crossing, then complete re-calibration is necessary. If a particular point cannot be recalibrated, then that specific point can be omitted.

3.3.3 SDK and Communication

In order to implement eye tracking in our application, we should establish a communication between ViewPoint software and our application.

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.

3.4 Head Tracking Device-Software

3.4.1 Overview

Figure 42: The Inertia Cube 3 [35]

The head tracking device used in this thesis is the InertiaCube3. 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.

3.4.2 SDK and Communication

The head tracking device has a test software and the InterSense Software Development Kit (SDK). The core of all InterSense software that are 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 provide a standard interface for the device.

Before the tracker is used, it must be configured and a diagnostic tool must be run. During this thesis, these tests were run once, right before the testing and experiment phase. This testing tool named the ISDEMO, that is provide by InterSence, 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 the tracker's sensor parameters can be modified. In addition, tools such as self-system test and compass calibration are provided to see that the device is working properly.

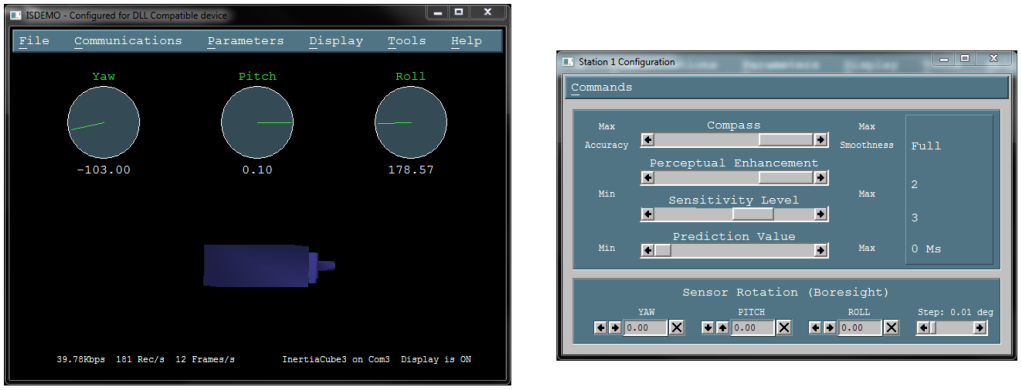


Figure 43: Left-ISDEMO application showing real-time Yaw,Pitch and Roll data. Right-Setting Adjustments for the head tracking device [35]

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. 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 through the InterSense DLL.

4 Implementation

After having explained the background and the necessary theoretical foundation and methods that are used in this thesis, we explain the implementation of our foveated ray-tracing rendering technique. But, first, we need to explain certain novel equations created and used in our technique.

In our approach, we created a probability model based on eccentricity(ω), which we used for the ray generation process. Our model was based on the work of Mandelbaum et. al. [17] who had measured the visual acuity for different values of the eccentricity of the human eye. Using the values that Mandelbaum et. al. extracted, we estimated the equation that could closely match the data-set with an accuracy of over 95% and from that we created the following equation:

$$V(\omega) = 0.90964e^{\frac{\omega}{2.9661}} + 0.00792 \tag{6}$$

, which represents how the visual acuity for the human eye changes based on eccentricity, as eccentricity increase the visual acuity fall.

Based on equation 6 we then created a probabilistic model equation, as seen below:

$$P(\omega) = \begin{cases} 1 & ,\omega <= \omega_o \\ 0.90964e^{\frac{\omega}{2.9661}} + 0.00792 & ,\omega > \omega_o \end{cases} \tag{7}$$

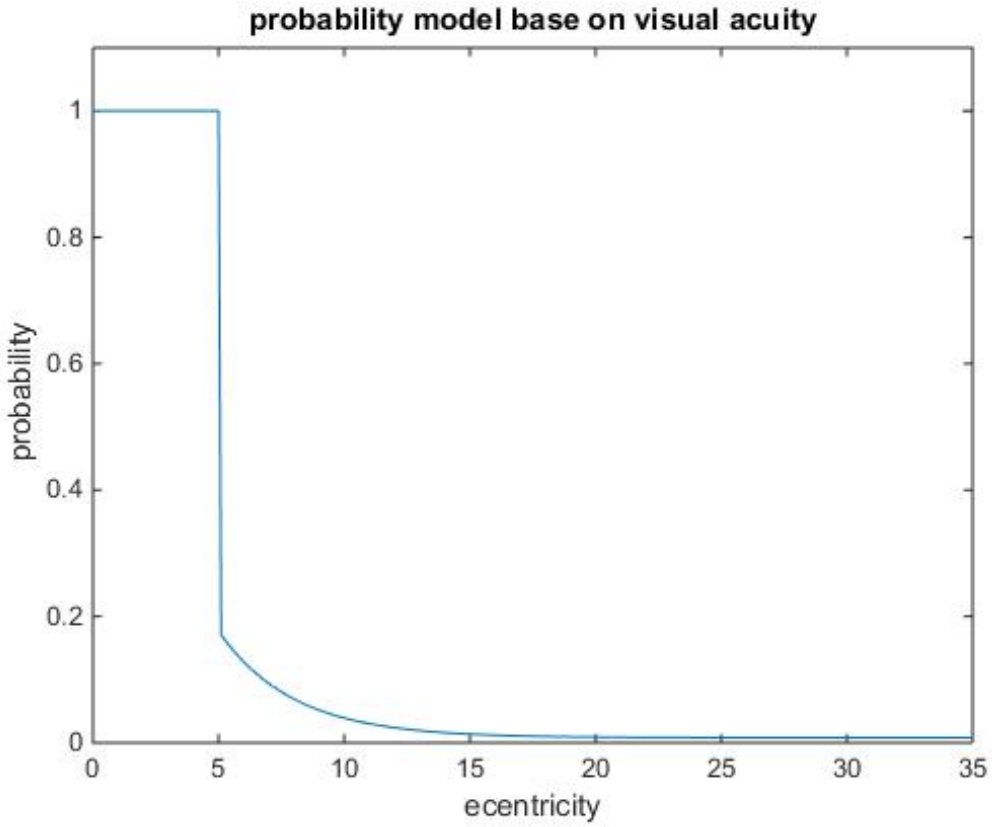


Figure 44: Probability model Visual representation for $\omega_o = 5^o$.

, where the ω_o is the eccentricity value we set to represent the size of the fovea of the human eye. When ω is less or equal than ω_o the probability is equal to one else the model is based on the equation given a probability according to the ω, as seen in Figure 44. This model was used to decide the probability of producing a new ray at a certain pixel based on the angle of each pixel from the center of the human eye. The ω is the eccentricity angle of each pixel from the center of they human eye. To calculate the ω for all pixels we calculate the distance between the pixel that the eye is fixating and all the pixels of the display. The distance is calculated using the euclidean distance equation 4. Having calculated the distance, we can use the visual angle equation 5 to find the ω while having prior knowledge that the distance between the eyes and the displays of our HMD has been set at 24mm.

After having created our model we started designing our Virtual Reality application in C++ language using the Optix engine, InterSense and Viewpoint SDK.

4.1 Eye Tracker

As mentioned in the previous chapters, the data from the infrared cameras of the Eye-tracking device are provided through the SDK that Arrington Reserach's ViewPoint EyeTracker® provided for third party applications. More specifically, the SDK comes with a Dynamic Link Library (DLL) named VPX_InterApp.dll which allows any third party application to interact with the eye–tracking device using the

ViewPoint EyeTracker®. By, importing the VPX_InterApp.dll we have access to the ViewPoint EyeTracker® application.

```
int theCallbackFunction2(int msg, int subMsg, int param1, int param2, void* usrPtr);
int theCallbackFunction2(int msg, int subMsg, int param1, int param2, void* usrPtr)
{

    int qualityData;
    VPX_GetDataQuality2(EYE_A, &qualityData);
    if (qualityData == 1 || qualityData == 0 || qualityData == 2)
    {
        VPX_GetGazePointSmoothed2(0, &gp);
        gp.x = gp.x * (width / 2) + (width / 2);
        gp.y = (1-gp.y) * height;
    }

    VPX_GetDataQuality2(EYE_B, &qualityData);
    if (qualityData == 1 || qualityData == 0 || qualityData == 2)
    {
        VPX_GetGazePointSmoothed2(1, &gp1);
        gp1.x = gp1.x * (width / 2);
        gp1.y = (1-gp1.y) * height;

    }

    return 0;
}
```

Figure 45: Head Tracking code for reading process from the eye tracking device

Afterwards, we defined a callback function which is managed by the library of the Viewpoint the VPX_InterApp.dll. The callback function is responsible for getting the gaze position corresponding to the screens from the eye tracking device and pass it to our application, as seen in Figure 45.

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.

Table 1: Quality Codes and the Respectively Information about the New Data Received [34]

Before we receive the data from the eye tracking device we undergo a quality check for their integrity. The ViewPoint EyeTracker® has six levels of quality for the data, as seen in table 1. We are accepting data that have a quality code of lesser or equal than two.

```
//Pass Data about they eyes positions on the screen.
context["Gaze_point_X_Left"]->setFloat(gpl.x);
context["Gaze_point_Y_Left"]->setFloat(gpl.y);
context["Gaze_point_X_Right"]->setFloat(gp.x);
context["Gaze_point_Y_Right"]->setFloat(gp.y);
```

Figure 46: Eye Tracking data code stored in context to be passed to the GPU.

The callback function is called once per frame. We pass the data we receive from the tracking device to the GPU using the Optix context, as seen in Figure 46. This is used in order to render the frame based on the gaze position, as foveated ray tracing rendering requires the center of the eye's attention.

4.2 Head Tracker

```
handle = ISD_OpenTracker (NULL, 0, false, true);

if (handle > 0){
    Tracker = ISD_TRACKER_INFO_TYPE();
    ISD_GetTrackerConfig(handle,&Tracker, true);
    lastTime = ISD_GetTime();
    ISD_GetStationConfig(handle,&Station[station - 1], station, true);
    data = ISD_TRACKING_DATA_TYPE();
    ISD_ResetHeading(handle,station);
}
```

Figure 47: Head Tracking code.

The data is passed from the head tracking device through the InterCube3 software's SDK. The first thing we do when our application starts is to connect to InterCube3 using the ISD_OpenTracker() function. The application seeks for the tracker that is connected to the computer and if a tracker is detected, then, a timer is set to count the connection duration and with ISD_ResetHeading we complete the synchronisation process, as seen in Figure 47.

After we have successfully connected to the tracker, head tracking data are passed from the device to our application by calling the ISD_GetTrackingData() function once per frame. The data received by InterCube3 is the yaw, pitch and roll of the orientation and they are used for each virtual camera in our scene in order to rotate them corresponding to where the HMD has turned. Finally, when we exit the application, the ISD_CloseTracker function is called, which is responsible for terminating the connection.

4.3 Virtual reality and scene creation - Host

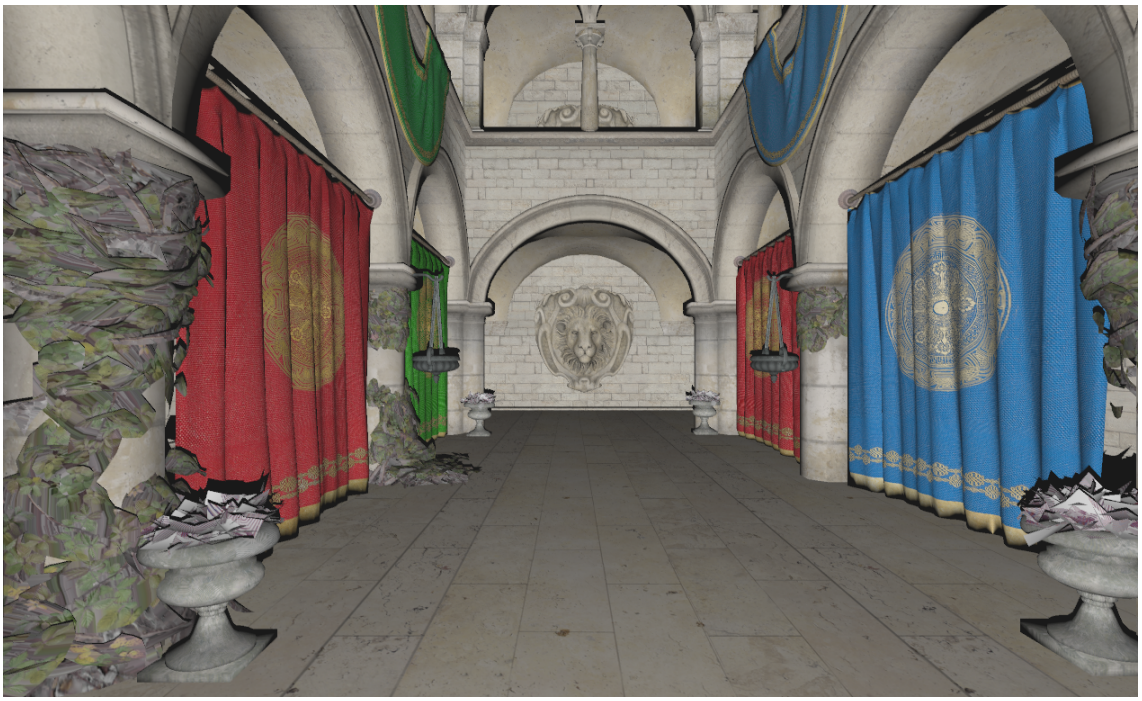


Figure 48: Sponza full ray tracing image for one display.

In this chapter, we explain how we created the Virtual Reality scene for our application. The scene that was created is the Sponza palace, which is one of the most iconic 3D environments used in computer graphics to illustrate rendering techniques. The Sponza palace scene can be seen in Figure 48.

```
context = Context::create();
context->setRayTypeCount(2);
context->setEntryPointCount(3);
context->setStackSize(2800);
context->setMaxTraceDepth(5);
```

Figure 49: Context code.

In the host program, as seen in Figure 50, we created a "context" as Optix requires using the Optix SDK to create it, as seen in Figure 49. An OptiX context provides an interface for controlling the setup and subsequent launch of the ray tracing engine, while it is also responsible for creating the virtual environment. A new context is created using the Context::create() function.

OptiX supports the notion of ray types, which is useful to distinguish between rays that are traced for different purposes. For example, a renderer might distinguish between rays used to compute color values and rays used exclusively for determining the visibility of virtual light sources (shadow rays). Proper separation of such

conceptually different ray types not only increases program modularity but also enables OptiX to operate more efficiently. The total number of ray types for a given context can be set with context→setRayTypeCount(), as seen in Figure 49. For our application, we set the number of different types of rays to two, one to compute color values and one for the shadow rays.

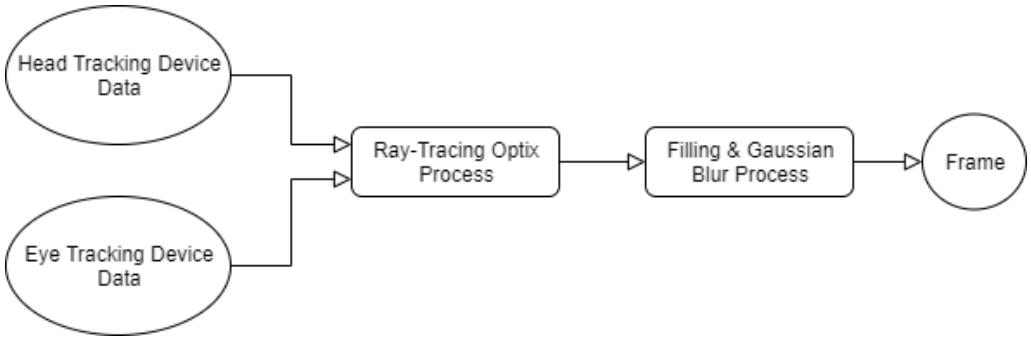


Figure 50: Our Foveated Ray-Tracing Pipeline.

Each context may have multiple computation entry points. A context entry point is associated with a single ray generation program as well as an exception program. The total number of entry points for a given context can be set with context→setEntryPointCount(), as we see in Figure 49. Our pipeline has two passes: the ray tracing Optix and the filling-Gaussian blur process (as seen in Figure 50), so we set the entry points to two.

```
// Ray generation program
const char* ptx = sutil::getPtxString(Name, "accum_camera.cu");
context->setRayGenerationProgram(0, context->createProgramFromPTXString(ptx, "pinhole_camera"));
context->setRayGenerationProgram(1, context->createProgramFromPTXString(ptx, "Filling_Pixel"));
context->setRayGenerationProgram(2, context->createProgramFromPTXString(ptx, "black_screen"));
// Exception program
Program exception_program = context->createProgramFromPTXString(ptx, "exception");
context->setExceptionProgram(0, exception_program);
context["bad_color"]->setFloat(1.0f, 0.0f, 1.0f);
// Miss program
context->setMissProgram(0, context->createProgramFromPTXString(sutil::getPtxString(Name, "accum_camera.cu"), "envmap_miss"));
context["bg_color"]->setFloat(0.34f, 0.55f, 0.85f);
//HDR enviroment
const float3 default_color = make_float3(1.0f, 1.0f, 1.0f);
std::string texture_path = std::string(sutil::samplesDir()) + "/data";
const std::string texpath = texture_path + "/" + std::string("cape_hill_4k.hdr");
context["envmap"]->setTextureSampler(sutil::loadTexture(context, texpath, default_color));
//phong Material
ptx = sutil::getPtxString(Name, "phong.cu");
phong_ch = context->createProgramFromPTXString(ptx, "closest_hit_radiance");
phong_ah = context->createProgramFromPTXString(ptx, "any_hit_shadow");
//Triangle geometry
ptx = sutil::getPtxString(Name, "triangle_mesh.cu");
mesh_intersection = context->createProgramFromPTXString(ptx, "mesh_intersect");
mesh_bounds = context->createProgramFromPTXString(ptx, "mesh_bounds");
mesh_intersection_refine = context->createProgramFromPTXString(ptx, "mesh_intersect_refine");
```

Figure 51: Program Link code.

In Figure 51, we present how we link the user-specified programs to the context

as Optix requires. The functionality of the user-specified programs will be analyzed in the next subsection. Because we have two entry points, as we explain above, we link each point to the associated ray generation program. The first variable in the setRayGenerationProgram() shows which ray generation program is scheduled to run first. As we saw in Figure 50 the first program to run is the ray tracing process and then the Filling-Gaussian Blur process. The same thing must be done for the exception and miss program and we set the variables accordingly. We calculate the closest hit program and any hit program that has been linked to the context created based on the Phong illumination model and from these we keep an instance of them so they can be associated with each 3d object of our scene. Finally, we link the necessary programs for the geometry of the triangles, the mesh intersection program, and the mesh bounding program.

```
//Buffer creation
Buffer buffer = sutil::createInputOutputBuffer(context, RT_FORMAT_FLOAT4, width, height, use_pbo);
context["output_buffer"]->set(buffer);
context["filter_buffer"]->set(buffer);
num_of_rays = context->createBuffer(RT_BUFFER_INPUT_OUTPUT, RT_FORMAT_INT, width, height);
context["number_of_rays_buffer"]->set(num_of_rays);
```

Figure 52: Buffer creation code.

OptiX uses buffers in order to pass data between the host and the device. Buffers are created by the host prior to the invocation of Context→Launch() using the sutil :: createInputOutputBuffer() function or the context→creteBuffer(). The Create Input Buffer function is provided by Optix SDK as an implementation for the GLUT library, which is responsible for the creation and management of any created windows. The Create Buffer function is responsible for creating a buffer which doesn't need to be associated with any window. The buffer type determines the direction of the data flow between host and device. The size of the dimensions of a buffer is determined by the width and height of the window we want to create, as seen in Figure 52.

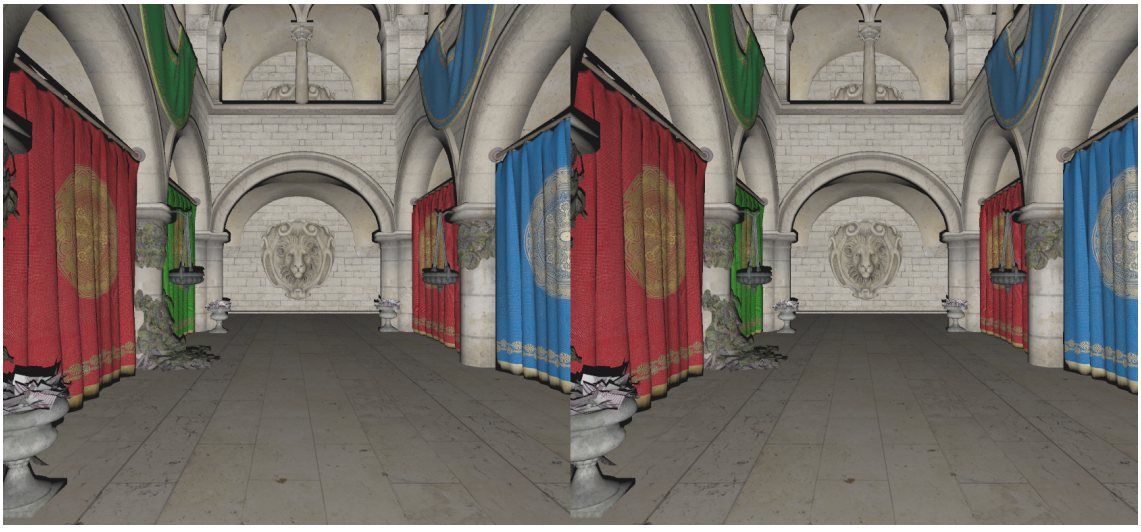


Figure 53: Virtual reality output for our HMD. Non-foveated.

In our case, our HMD has a total resolution of 2560 x 1024, and thus we create a two-dimensional buffer for each pass of our pipeline, where each dimension has the size of the total resolution of our HMD, instead of using two separate buffers of size 1280 x 1024, which is the resolution for each individual display of our HMD, because GLUT for Optix couldn't show the two buffers at the same time for two different windows. This solution works because we can connect our HMD to the computer like having two monitors in a row. Then, we created a window with size equal to the total resolution of our HMD and we placed it on the upper left of the left display. By doing that, the rendering frame which is stored in the buffer contains two images, one for each display, and matches it correctly to the displays of our HMD. The ray generation program is responsible for creating the two images and storing them into one buffer, as seen in Figure 53. We also created a buffer to count the number of rays that our app has generated with the same configurations used by the previous buffer. This new buffer was not connected to GLUT because we do not want to render it and so we used the context→creteBuffer().

Our HMD uses the partial overlap method to achieve stereoscopic vision. To achieve that, we created two pinhole cameras in our virtual scene. Based on the HMD's specifications, the field of view of each camera-eye was set to 90^o and the cameras were rotated by 13^o left and right respectively, while the distance between them was based on the average pupil distance measurement, which is at 64mm.

```
GeometryGroup Load3dObjects(const std::string& filename)
{

    GeometryGroup geometry_group = context->createGeometryGroup();
    geometry_group->setAcceleration(context->createAcceleration("Trbvh"));
    OptiXMesh mesh;
    std::string mesh_file;

    for (int i = 0; i < 382; i++)
    {
        if (i != 2 && i!=4) {
            mesh_file = std::string(sutil::samplesDir()) + "/data/Sponza-master/sponza_" + std::to_string(i)+".obj";
            mesh.context = context;
            mesh.use_tri_api = use_tri_api;
            mesh.ignore_mats = ignore_mats;
            mesh.intersection = mesh_intersection_refine;
            mesh.bounds = mesh_bounds;
            loadMesh(mesh_file, mesh, Matrix4x4::identity().scale(make_float3(2)));
            aabb.set(mesh.bbox_min, mesh.bbox_max);
            geometry_group->addChild(mesh.geom_instance);
            number_of_triangles += mesh.num_triangles;
            }
        }


    return geometry_group;
}
```

Figure 54: Code for loading 3D objects in the virtual scene.

The Load3dObject() is responsible for reading the 3D objects and create a geometry group, while setting the acceleration structure for this geometry group as Optix requires. We also declared a variable named "mesh" which uses the OptixMesh structure to store the necessary variables of the 3D object we read. To read each 3D object of our scene we run a loop for a total of three hundred eighty-two

steps, which is the number of objects the sponza palace has. For each object, we define the file path of the mesh structure and we link it to the context. In the mesh structure, we also define if we want to ignore the materials of the object and link the intersection and bounding programs. Using the LoadMesh() function provided by Optix we give as input the mesh structure, the file location of each 3D object and if we want to scale or rotate or transform the specific object when we load each object. The same function is responsible for reading the files of the objects and takes any information they have on vertices, normals, texture coordinates, triangle indexes, and material indexes associated with the object and stores them in the appropriate buffers. Also, each material that is created is also connected with the closest hit program, passing the variables that the Phong illumination model needs. Finally, we add an instance of the mesh as a child to the geometry group.

```
void setupScene(const std::string& filename)
{
    GeometryGroup geometry_group_gg  = loadMesh(filename);
    // Create a top-level Group to contain the two GeometryGroups.
    Group top_group = context->createGroup();
    top_group->setAcceleration(context->createAcceleration("Trbvh"));
    //Add the geometry Groups for our scene.
    top_group->addChild(geometry_group_gg);
    context["top_object"]->set( top_group );
    context["top_shadower"]->set( top_group );
}
```

Figure 55: Code for setting up the scene.

The SetupScene() function creates a geometry group and sets an acceleration structure to BHV and is then added as the topmost child to the geometry group that was created at the LoadMesh() function. The final step is for the context to set the topmost group to the variable top_object.

```
void setupLights()
{

    BasicLight lights[] = {
        { make_float3(0.0, 20.0f, 0.0f ), make_float3( 0.7f, 0.7f, 0.7f), 1 }
    };

    Buffer light_buffer = context->createBuffer( RT_BUFFER_INPUT );
    light_buffer->setFormat( RT_FORMAT_USER );
    light_buffer->setElementSize( sizeof( BasicLight ) );
    light_buffer->setSize( sizeof(lights)/sizeof(lights[0]) );
    memcpy(light_buffer->map(), lights, sizeof(lights));
    light_buffer->unmap();

    context[ "lights" ]->set( light_buffer );

}
```

Figure 56: Code for creating the light sources and store it in the appropriate buffer.

The setupLights() function creates a buffer that contains all the information of the light sources, where each light source has information about its position in the

scene and its color. Next, we pass the light buffer to the GPU where it is used by the closest hit program for the calculations of the Phong model. In our scene, we only used one light, which works as a point light, which emits in all directions and has infinite range, similar to directional lights. If we use more than one light, the frames per second (fps) drop because our scene was complex (256 thousand triangles) and using two lights has a great impact in performance.

4.4 Ray-Tracing Optix User-Specified Programs - Device

Our pipeline (figure 50) uses the Optix engine for the ray-tracing process as the first step. In order to use the Optix engine for they ray tracing, we must define six different programs: Ray Generation program, Intersection program, Bounding program, Closest Hit program, Any Hit program, Miss program, and Exception program. In the Ray Generation program, we created our foveated technique because the program is responsible to generate the rays therefore this the program where we must implement our technique. The Intersection program is responsible to find the closest intersection between a ray and the 3D environment and the bounding program creates the boundaries for each 3D object of our 3D environment, which are used by the accelerated structure. The Closest hit program is used to calculate the shading of the surface that is hit by a ray by using the Phong model. The Any hit program is called when a shadow ray hit a 3D object. The Miss program handles all the rays that didn't hit any 3d object in our 3D environment. The Exception program is called when there is and overflow in the local stack.

4.4.1 Ray Generation Program

In order to create our foveated rendering technique, we created three different layers (Foveal, Periphery, and outside region) for each display, which are centered around the pixel that each eye is focused on. In the Foveal region, we use full ray-tracing for each pixel. In the other two regions, we reduce the amount of rays sampled to one ray for a set of pixels. To classify the pixels into regions we used two equations: the euclidean distance equation 4 and the visual angle equation 5. Initially, for each pixel we used the euclidean distance to find the distance between the pixel that the eye is focused on and the pixel that has been schedule to be ray-traced. After having calculated the distance between the two pixels, we used the visual angle equation to calculate the angle that is created between the user's eye, the pixel the eye is focused on and the pixel that is to be ray traced. The host program provides the angle of the Foveal and periphery regions and by comparing the angle we have calculated with the angle of the regions we can check at which region the pixel belongs to.

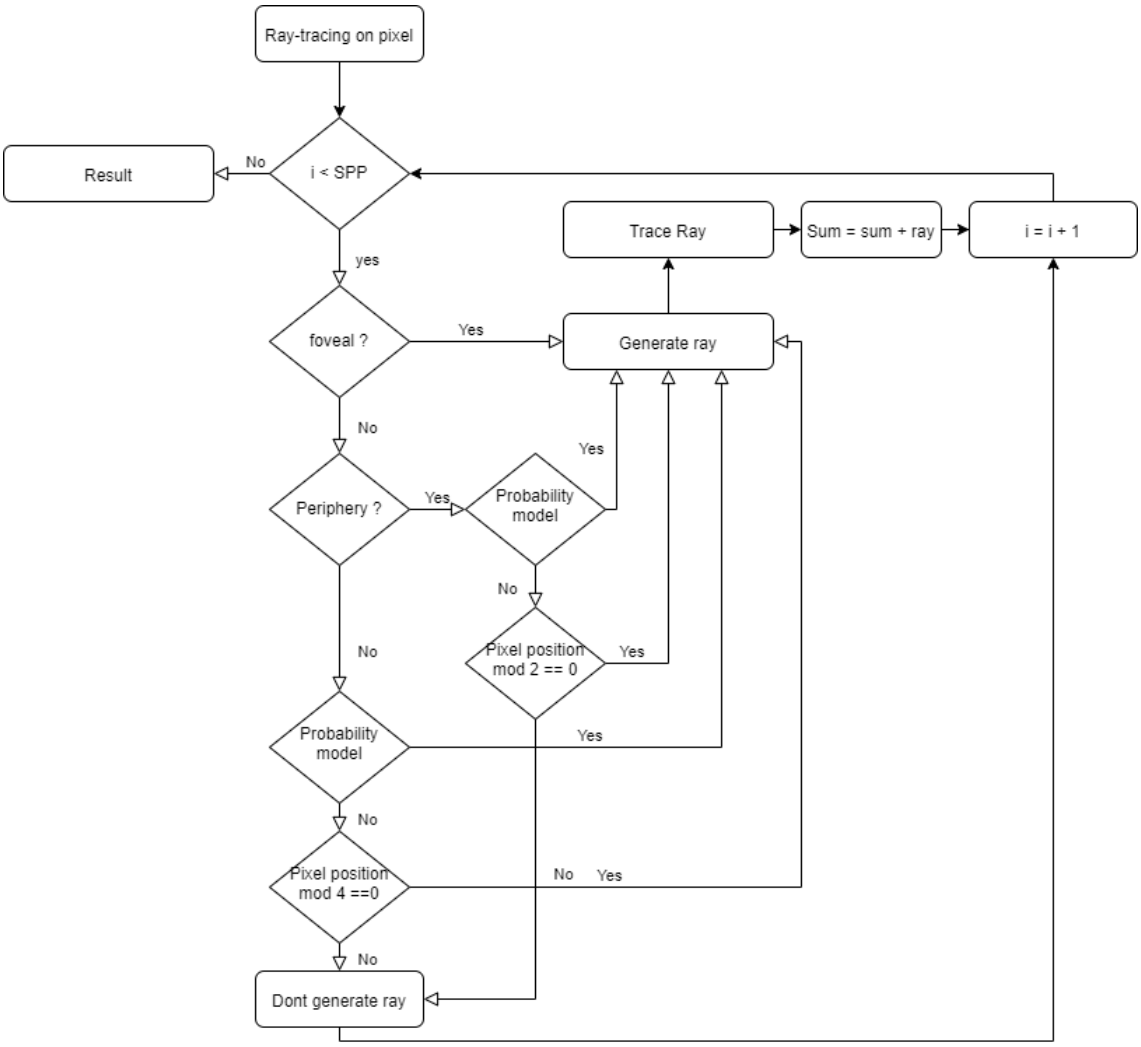


Figure 57: The flow chart of our algorithm for the ray generation procedure for each eye.

The execution flow chart for the ray generation process can be seen in Figure 57. Initially, we run the Ray Generation process a set number of times based on the Sample Per Pixel (SPP) parameter, which is a predefined value provided by the host program. This means that for one pixel we will create one or more rays. If we create one ray it's sent through the center of the pixel, while multiple rays per pixel are sent from different positions inside the same pixel. Before we generate any ray, we estimate the region the pixel resides in (fovea, periphery or the outside zone). If the pixel is inside the foveal we generate the ray without any other conditions. Otherwise, if the pixel is inside the peripheral region or the outside region we add two more conditions either of which determines if we will generate a ray. The first condition is based on the probability model we created. Each pixel is given a probability based on its angle and using that probability we randomly decide if a ray will be generated on that pixel. The second condition determines if each pixel is the selected pixel to cast a ray for its set of pixels. For pixels inside the periphery we shoot one ray for every 2x2 square while on the outside zone for every 4x4 square. In each set of pixels we only cast a ray at the top-left corner of

the group. If both conditions are met, we generate a ray and schedule it for tracing. When the tracing procedure ends, we store the color that the ray carries to the output buffer, which stores the colors for each pixel of the current frame. In Figure 58, we see the frame that is produce by the ray tracing program using our model.

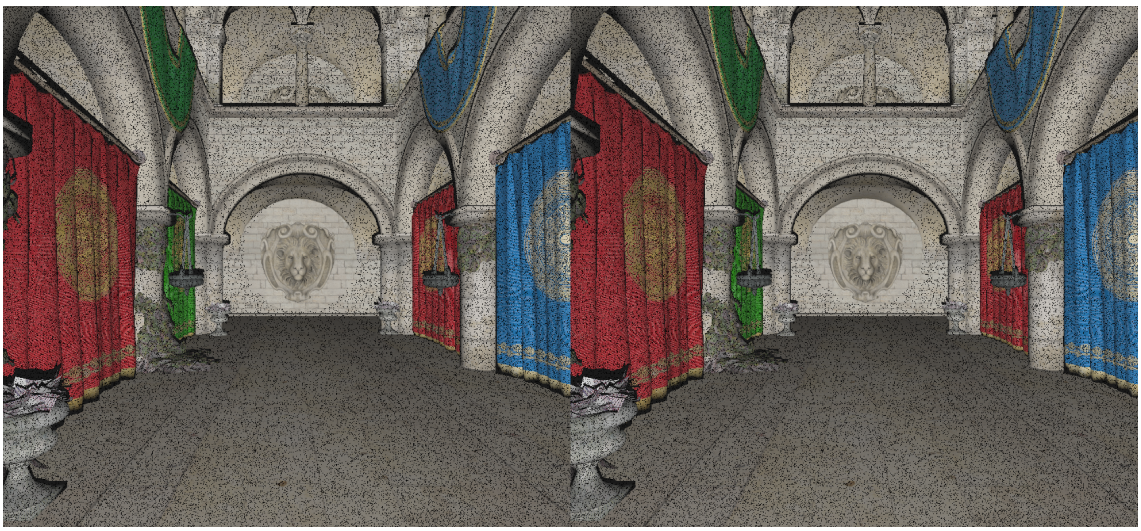
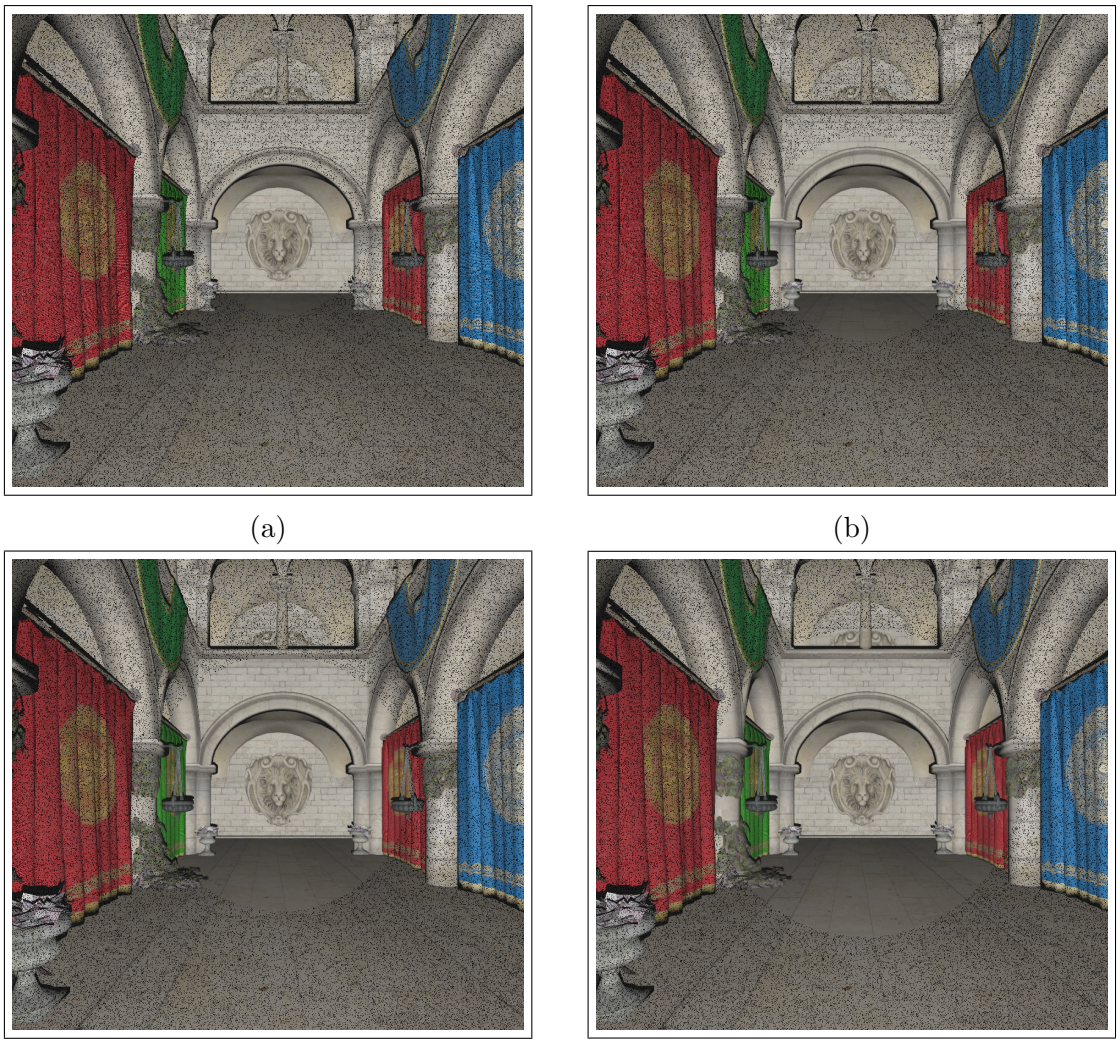


Figure 58: Foveated ray tracing after the Ray Generation program has ended. Foveal = 5^o, Periphery = 10^o.



(a) (b)

(c) (d)

Figure 59: Example of foveated ray tracing after Ray Generation for one eye with (a) foveal=7.5^o , periphery= 15^o (b) foveal = 10^o, periphery = 20^o (c) foveal = 12.5^o and periphery = 25^o (d) foveal = 15^o and periphery = 30^o

4.4.2 Intersection Program

In the Intersection and Bounding program, we used the example provided by the Optix SDK to create the Bounding program and the Mesh Intersection program. The Mesh Intersection program looks for a potential intersection in the closest triangle of a 3D object. If there is an intersection, it is responsible for providing the closest hit program with the texture coordinates and the normal of the triangle. The Bounding program computes an axis-aligned bounding box for its primitives which is used by OptiX in the Acceleration Structure process.

4.4.3 Closest & Any Hit Program

In the Closest Hit program, we choose the Phong illumination model to be implemented as our local illumination model. The Closest Hit program is called for each ray that hit a surface. In the host program, we have linked the necessary variables for each 3D object as the Phong model required.

The Any Hit program is called only when a shadow ray hits a 3D object and simply stores a zeroic value to the payload of the shadow ray. The Any hit is responsible for terminating the shadow ray (see Figure 60).

```
RT_PROGRAM void any_hit_shadow()
{
    // this material is opaque, so it fully attenuates all shadow rays
  prd_shadow.attenuation = make_float3(0.0f);
  rtTerminateRay();
}
```

Figure 60: Any and Closest Hit Program code

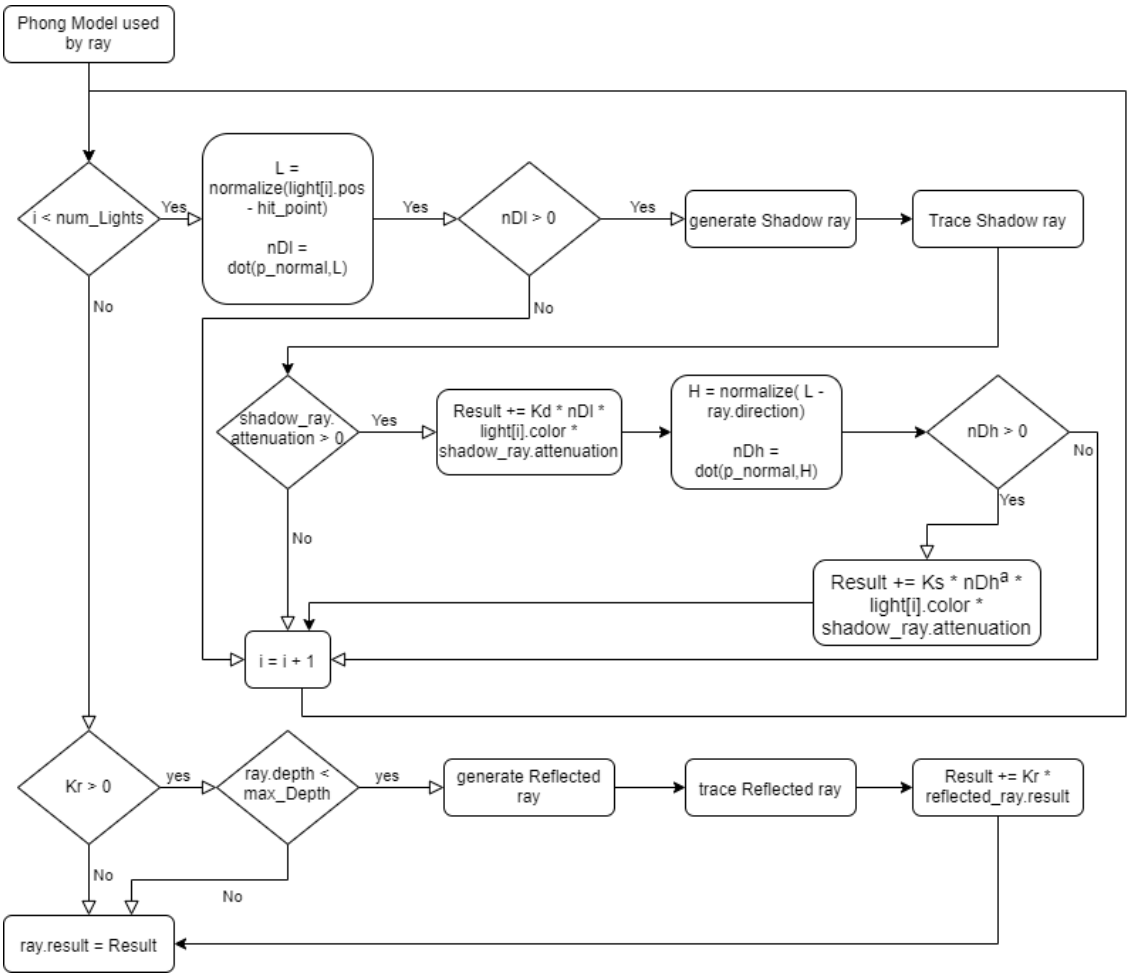


Figure 61: Flow Chart of the Phong Illumination model.

The above flow chart (Figure 61) shows the algorithm implementation for the Phong illumination model. We calculate two contributions on the 3D object the specular and diffuse values as a sum for each independent light source (in our case we have one light source but without loss of generalization). Initially, we calculate $\hat{L}$ for each light source, which is the normalized vector from the object surface to the light source and we calculate the nDl which is the dot product between the $\hat{p\_normal}$(surface normal) and the $\hat{L}$ which represent the cosine angle between them . If the nDl is negative it means that the light source is behind the surface and if equal to zero the vectors of are codirectional. Thus, we generate a shadow ray only when nDl is greater than zero we generate and we trace it. If the Any hit program is not executed, we set the shadow ray payload equal to nDl. If the nDl is positive we calculate the diffuse contribution of each light . The diffuse contribution is calculated by multiplying the Kd (diffuse color of the 3D object) with the color of the virtual light and the payload of the shadow ray. The next step is to calculate the specular contribution of the 3D object's surface. In order to do that, we compute the halfway vector ($\hat{H}$) that lies halfway between $\hat{L}$ and the negated ray direction. Then, we calculate the nDh which is the dot product between the $\hat{p\_normal}$(surface normal) and the $\hat{H}$ which represents the cosine angle between them. If the nDh is greater than zero we calculate the specular contribution. The nDh is raised to a specular power based on the shininess of the 3D object, which controls the sharpness of the highlights. The specular contributions is calculated by multiplying the Ks (specular color of the 3D object) with the color of the virtual light and the payload of the shadow ray.– and after that, we move to the next light. The final step is to calculate the reflections on our object's surface based on reflections from other surfaces that our ray hit. If the Kr (roughness of the 3D object surface) is greater than zero and if we haven't surpass the max depth we generate a reflective ray. The reflective ray is traced and we calculate the contribution by multiplying the Kr with the payload of the reflective ray and added to the payload of the ray.

4.4.4 Miss & Exception Program

```
rtTextureSampler<float4, 2> envmap;
RT_PROGRAM void envmap_miss()
{
  float theta = atan2f( ray.direction.x, ray.direction.z );
  float phi   = M_PIf * 0.5f -  acosf( ray.direction.y );
  float u     = (theta + M_PIf) * (0.5f * M_1_PIf);
  float v     = 0.5f * ( 1.0f + sin(phi) );
  prd.result = make_float3( tex2D(envmap, u, v) );
}
```

Figure 62: Miss Program code

The miss program is executed when a ray doesn't hit anything and when it is executed it returns a default pre-defined value based on the environment map (an

image) which is used as background. The value for the envmap is set by the host and can be modified between different frames of the raytracer. On the host, this texture is bound to an image read from a file. Then we modify the miss program to compute the latitude and longitude of the ray's direction and lookup the color from an environment map, as seen in Figure 62

```
RT_PROGRAM void exception()
{
  const unsigned int code = rtGetExceptionCode();
  rtPrintf( "Caught exception 0x%X at launch index (%d,%d)\n", code, launch_index.x, launch_index.y );
  output_buffer[launch_index] = make_float4( bad_color,0.0f );
}
```

Figure 63: Exception Program code

The Exception program is executed when the local stack overflowed. If this occurs, all processing for the current ray generation program is aborted and an exception program is executed. In this application, we just set the output buffer to a special color (also set by the host) to alert the user that an exception has occurred. As seen in Figure 63, the output_buffer holds the information for the frame we calculate and when an exception is raised we store the value of the bad_color to the buffer.

4.5 Filling Process & Post-Processing

Our pipeline (figure 50) uses a second pass for post-processing, which is responsible for covering the blank pixels that have been created and covers the imperfections that are produce from the process. To cover the imperfections, we used a Gaussian blur mask. In order to reduce the rendering time for each frame, we combine the two post-processing effects in one pass in our rendering pipeline.

The post-processing pass receives the output buffer as an input, which stores the values of each pixel after the ray-tracing pass. For each pixel we apply a 5x5 Gaussian blurring mask. Since a large amount of pixels in the covered region of the Gaussian blur will not have a value, we use the value of the top left corner pixel of the pixel set for each blank pixel, which will always have a value based on our ray generation process. When the loop ends the flling-Gaussian process has finished and the output buffer that is produced is displayed on our HMD, as seen in Figure 64.

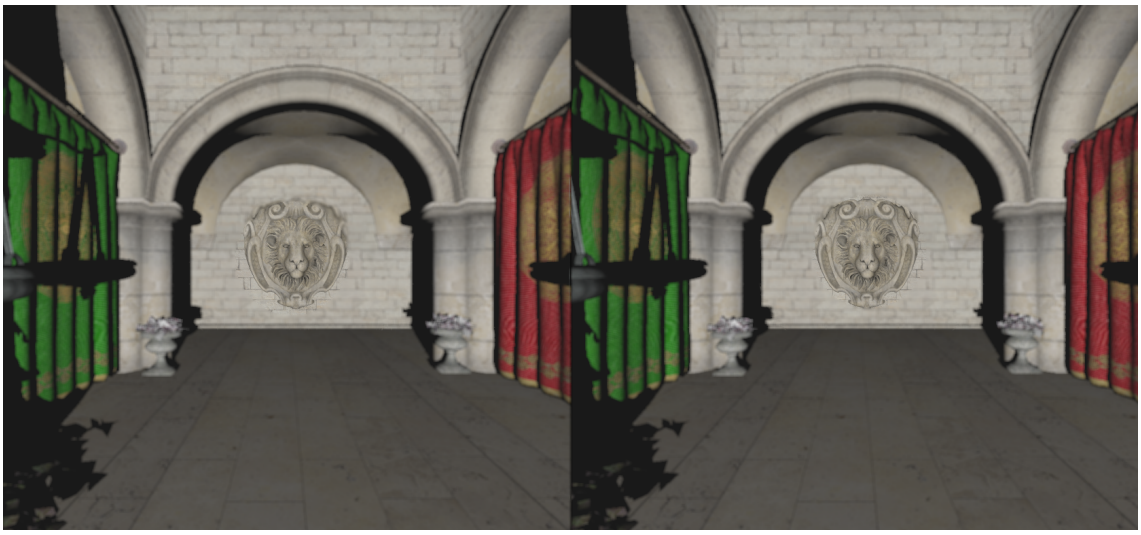
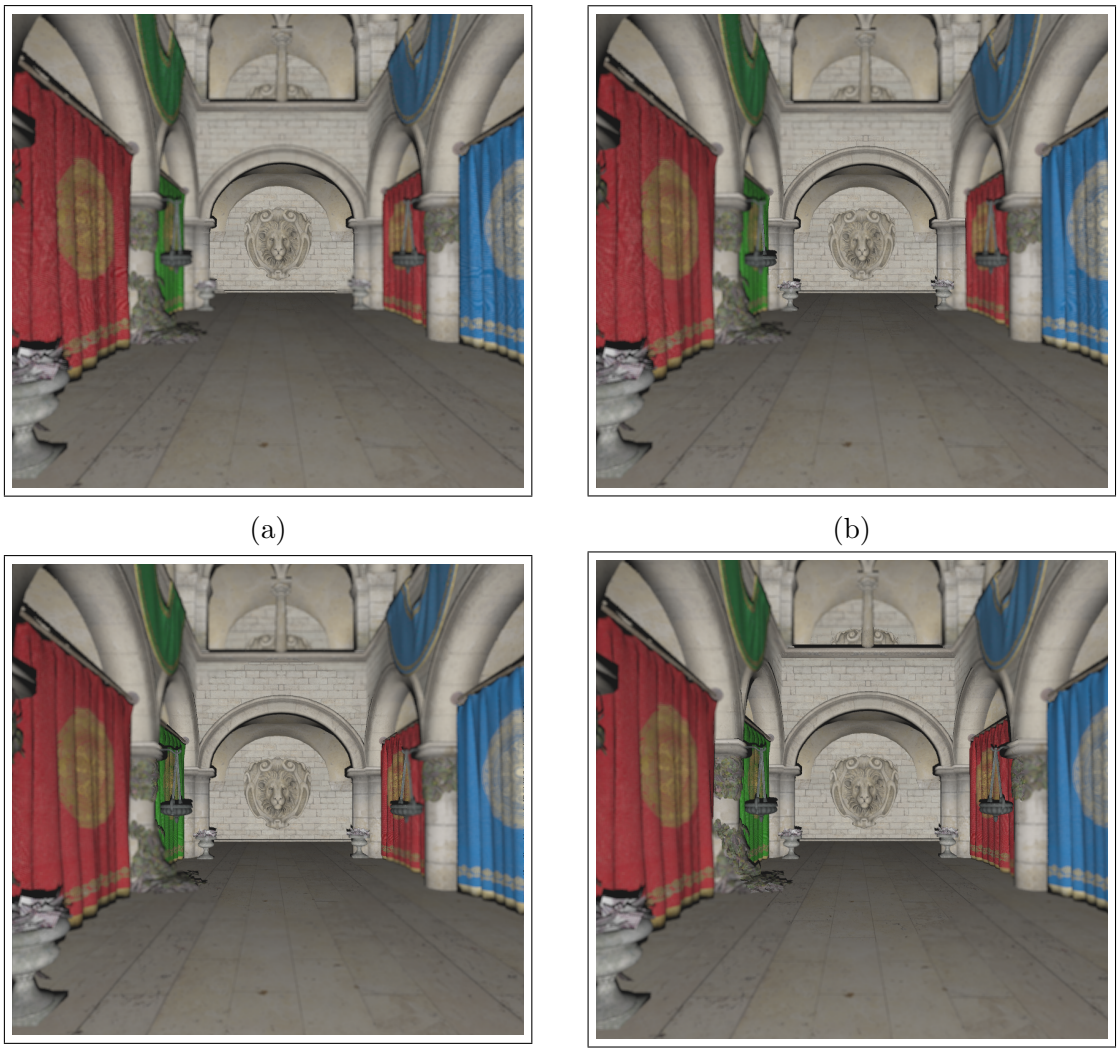


Figure 64: Filling-Gaussian process for both eyes. Foveal = 5^o, Periphery = 10^o



(a) (b) (c) (d)

Figure 65: Example of foveated ray tracing after Filling-Gaussian process has ended for one eye with (a) foveal=7.5^o , periphery= 15^o (b) foveal = 10^o, periphery = 20^o (c) foveal = 12.5^o and periphery = 25^o (d) foveal = 15^o and periphery = 30^o

5 Evaluation

In order to evaluate our technique we want to calculate the performance increase factor of our technique over using full scale ray Tracing. Due to our sub-sampling of areas outside the fovea we also want to make sure that the reduction in quality does not hinder user experience, or ideally is not noticeable at all. To calculate the performance of our approach we compare two numerical values: the Rays per frame and the application's maximum possible Frames Per Second while running under the same conditions. To calculate the optimal boundaries for the fovea and periphery where the reduction in quality was not perceived and empirical user study was organised.

5.1 Benchmark evaluation

$$Percentage = (\frac{Foveated}{non - Foveated} - 1) * 100\% \tag{8}$$

To see how efficient our technique is we calculated how many rays per frame (RPF) are generated and how many frames per second (FPS) can be achieved for various different values of eccentricity in the foveal and periphery regions as well as for different samples per pixel, meaning how many rays were shot for a single pixel. Then, we calculate the reduction in rays and the performance in FPS we gained by comparing each value with the full ray tracing process without subsampling any regions outside the foveal, as seen in equation 8. The following results have been calculated for the Sponza scene we created and for the resolution of the Nvis SX111 HMD (2560x1024 for two eyes).

Eccentricity		SPP			
Foveal	Peripheral	1	4	9	16
5^o	10^0	4.47	5.47	7.13	9.46
7.5^o	15^0	4.49	6.74	10.48	15.71
10^o	20^0	4.52	8.53	15.20	24.54
12.5^o	25^0	4.56	10.76	21.09	35.55
15^o	30^0	4.60	13.11	27.30	47.17
Full	-	5.21	20.85	46.91	83.39

Table 2: RPF $\cdot 10^6$ for different SPP values and different sizes of eccentricity on the foveal and peripheral regions.

As seen in Table 2, as the SPP, foveal and periphery values increase we have more rays. The reason for that is that as the value of eccentricity increases the foveal circle covers a larger region of the screen, meaning more rays will be generated. As the SPP increase, each pixel corresponds to more rays shot compared to

smaller values of SPP.

		Reduction Percentage			
Eccentricity		SPP			
Foveal	Peripheral	1	4	9	16
5^o	10^0	14.2%	73.76%	84.8%	88.65%
7.5^o	15^0	13.81%	67.67%	77.65%	81.16%
10^o	20^0	13.24%	59.08%	67.59%	70.57%
12.5^o	25^0	12.47%	48.39%	55.04%	57.36%
15^o	30^0	11.7%	37.12%	41.80%	43.43%

Table 3: Reduction percentage on RPF for different SPP values and different sizes of eccentricity on the foveal and peripheral regions

As seen in Table 3, we calculated the reduction of RPF as percentage using the equation 8. We notice that as the SPP value increase the reduction percentage is also increased. The reason for that is that as the SPP increase each pixel corresponds to more rays shot compared to smaller values of SPP and the rays that are cutoff by our foveated rendering technique increase even more. We notice that as the eccentricity increased the reduction percentage drop. This happen because as the value of eccentricity increases the foveal circle covers a larger region of the screen, meaning more rays will be generated.

Eccentricity		SPP			
Foveal	Peripheral	1	4	9	16
5^o	10^0	60.16	57.84	53.79	48.15
7.5^o	15^0	59.98	54.02	46.22	39.40
10^o	20^0	59.74	50.27	40.29	31.57
12.5^o	25^0	59.07	48.39	34.29	25.11
15^o	30^0	58.64	47.26	28.90	19.84
Full	-	103.88	26.12	17.24	10.05

Table 4: FPS $\cdot 10^6$ for different SPP values and different sizes of eccentricity on the foveal and peripheral regions.

As seen in Table 4, as the SPP increases we notice an FPS drop. In addition, as the eccentricity increases, we also notice another FPS drop. This happens because as the number of rays increases more processes are needed to calculate them, which results in longer rendering times. As a special case, we see that for SPP equal to one the full ray tracing process has higher FPS than the foveated ray tracing. This happens because our pipeline uses a two pass rendering process, and the cost of scheduling the second pass is higher than the cost of full ray-tracing. Although in this case foveated rendering has lower performance, the quality of the image for SPP equal to one is much poorer than that of higher SPP values, in which cases our

approach has a significant increase in FPS compared to the full ray tracing process.

		Performance Gained			
Eccentricity		SPP			
Foveal	Peripheral	1	4	9	16
5^o	10^0	-42.0.8%	121.43%	212%	379.10%
7.5^o	15^0	-42.26%	106.81%	168.09%	292.03%
10^o	20^0	-42.49%	92.45%	133.7%	214.12%
12.5^o	25^0	-43.13%	84.87%	98.54%	149.85%
15^o	30^0	-43.55%	80.93%	67.63%	97.41%

Table 5: Speedup on FPS for different SPP values and different sizes of eccentricity on the foveal and peripheral regions.

As seen in Table 5, we calculated the performance gained as a percentage in FPS using the equation 8. As the SPP increase we notice an increase in the performance we gain. As a special case, we see that for SPP equal to one the performance is negative. This happens because our pipeline uses a two pass rendering process, and the cost of scheduling the second pass is higher than the cost of full ray-tracing. We notice that as the eccentricity increased the performance drop. This happens because as the number of rays increases more processes are needed to calculate them, which results in longer rendering times.

As seen in both Table 3, 5, we achieved good performance gains in the FPS and a good reduction percentage for RPF using our Foveated ray-tracing rendering technique. Despite all this, the FPS values are far away for optimal because for virtual reality applications is recommended to have a performance of 120 FPS so the users don't get dizzy during the use of a virtual reality application. One of the reasons we didn't achieved higher FPS is because of the highly complex scene that was created. Another reason is that the GPU card we use for this application was RTX 2060 which is not a VR ready card compare to the RTX 2070,2080,2080Ti.

5.2 User evaluation

To see if our technique is perceptible from the users we conduct three different experiments. In order to find, if our foveated rendering technique produces acceptable quality compare to non-foveated rendering. We conduct a pair test, a ramp test, and a slider test. The number of users that participate in the experiments is 15. The experiments that we mentioned were conducted for the Sponza scene we have created and for the Nvis SX111 HMD (2560x1024 for two eyes) and for SPP equal to 4.

The pair test presented each user with pairs of short animated sequences and separated by a short interval(0.5s) of black. One piece of the pair was the non-foveated rendering and the other used foveated rendering at Foveal eccentricity levels from

Foveal = 5^o to Foveal = 15^o and corresponding periphery eccentricity levels from periphery = 10^o to periphery = 30^o. Pairs of all level were presented twice, in both orders (non-foveated then foveated, and foveated then non-foveated). After showing each pair to the user they were asked to report whether the first rendering was better, the second was better, or the two were the same quality. The experiment was designed to interrogate what foveation quality level was comparable to non-foveated rendering.

The ramp test presented each user with a set of short sequences were initially they were presented with a non-foveated animation as a reference and then the Foveal and Periphery levels incrementally ramped either up or down from a foveal eccentricity of Foveal =5^o or Foveal = 15^o. Users were then asked whether the quality had increased, decreased, or remained the same over each sequence. Each ramp was presented in both directions (5^o to 15^o and 15^o to 5^o), sampled using 5 discrete steps, each 5 seconds long and separated by a short interval of black. The study was conduct to find the lowest Foveal level quality perceived to be equivalent to the non-foveated reference.

The slider test let users change the foveal and periphery eccentricity level themselves. Users were first presented with a non-foveated animation as a reference. Then starting at a low level of foveation eccentricity level (Foveal = 5^0 and Periphery = 10^o), users could increase the level, show the non-foveated reference again, or decrease the level, with the task of finding a quality level equivalent to the non-foveated reference.

Because of a technical problem with our eye tracking device, we were not able to have an accurate gaze position of the eyes in the displays of our HMD. To conduct the experiments, we asked from the users during the experiment to look on a specific point which we manually placed the center for each foveal region to match they gaze for each eye to that point. The technical problem was that the mirror that each camera use to capture the image of the eye was damaged resulting in the Viewpoint EyeTracker® software not been able to locate their gaze positions correspond to the displays correctly.

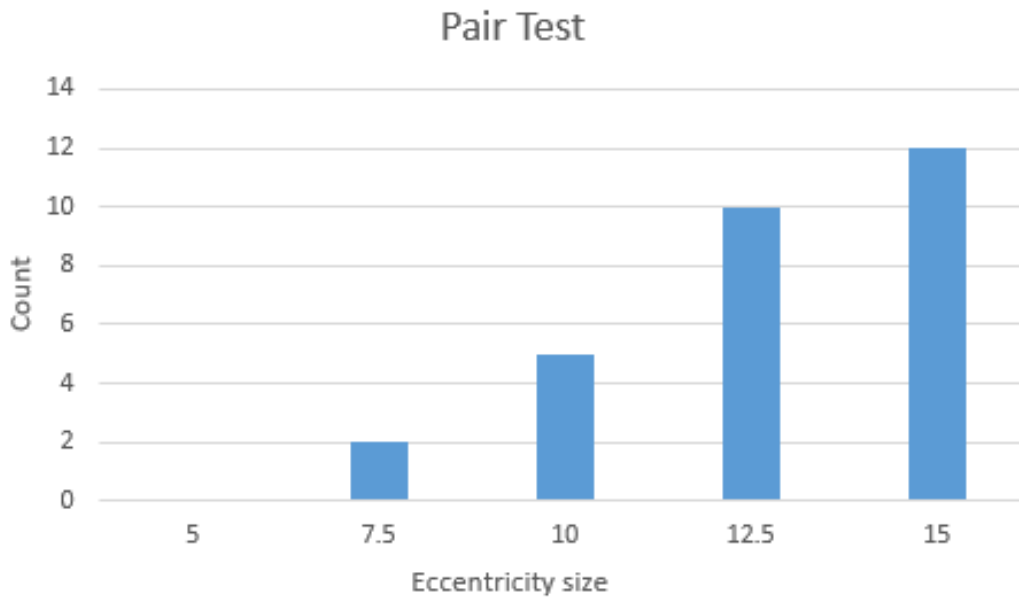


Figure 66: Pair Test

In the pair test, we recorded which of the user's report that foveated and non-foveated rendering was the same and produce the diagram seen in Figure 66, where for Foveal eccentricity equal to 12.5^o and 15^o most users weren't able to distinguish a difference between foveated with non-foveated. A small number of users report that in all different eccentricity levels where able to compare foveated with non-foveated.

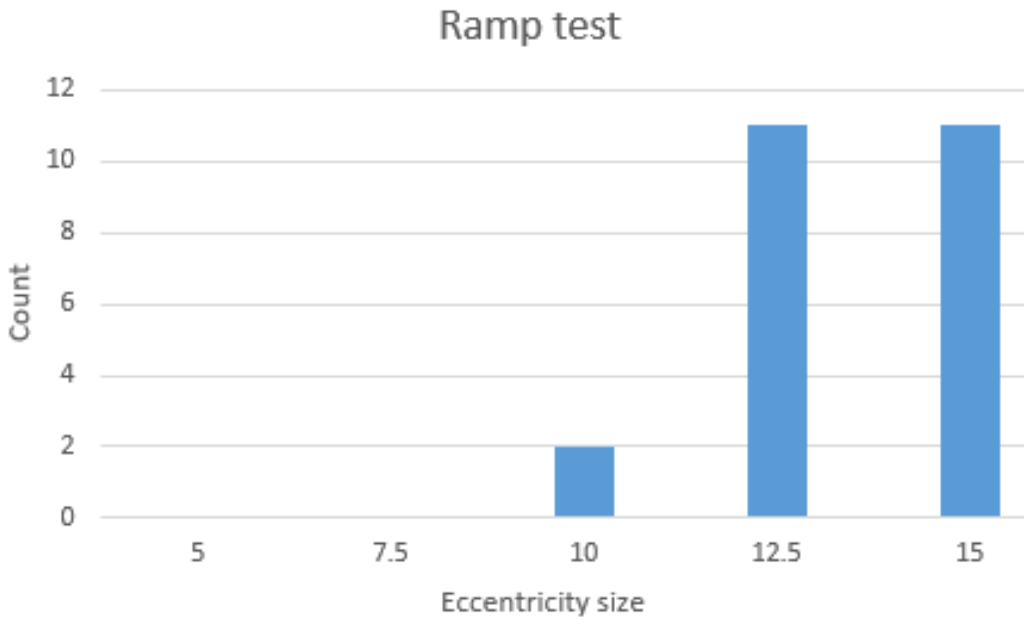


Figure 67: Ramp Test

In the ramp test, we recorded which of the user's report for each sequence if the quality was the same compare to the reference quality (non-foveated rendering)

and produce the diagram seen in Figure 67, where we see that when eccentricity for Foveal is equal to 12.5^o and 15^o users found quality be the same with the non-foveated reference.

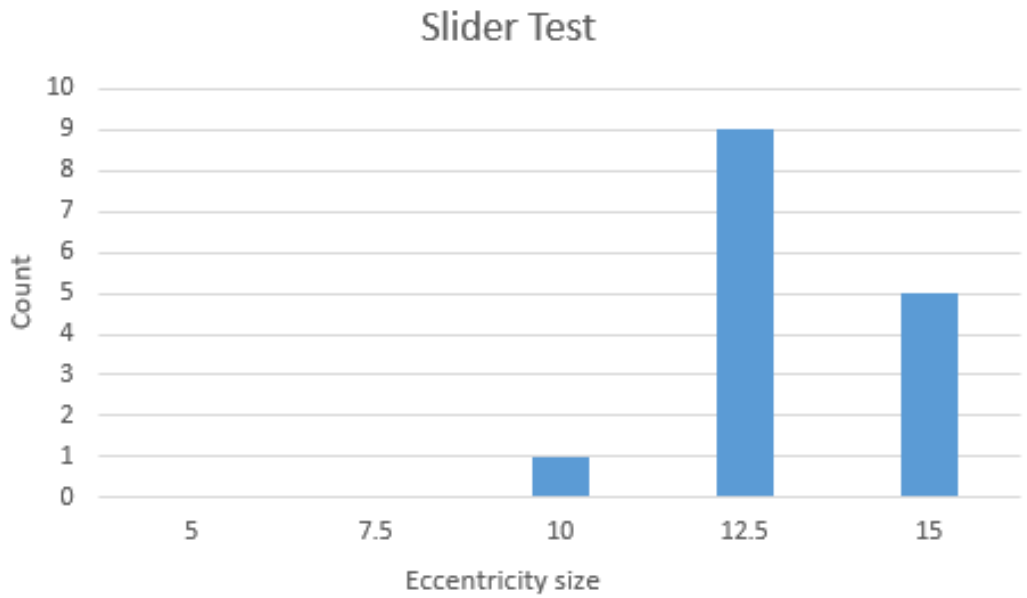


Figure 68: Slider Test

In the slider test, we recorded the lowest Foveal eccentricity level at which users report to be equivalent to the non-foveated reference then we created the diagram seen in Figure 68. As seen in Figure 68, most of the users report 12.5^o as they preference value for foveated rendering.

From the results, we found that most users find foveated rendering to be the same with then non-foveated for eccentricity equal to 12.5^o and higher without finding any difference in quality. If our equipment wasn't damaged, we hope to achieved a lower Foveal eccentricity level, which will result in higher FPS.

6 Conclusion & Future work

6.1 Conclusion

In this thesis, we proposed a different approach to foveated ray-tracing rendering for virtual reality. Ray tracing is aiming to become the new way to calculate immersive lighting and shaders, but the amount of calculations that is required make optimisations mandatory. Foveated rendering research also aims to become the norm, as eye tracking is slowly being integrated in everyday platforms like smartphones and VR HMDs and promises increased performance without perceived reduction in quality by the end user.

One of the main contributions of our work is the use of a probability model based on visual acuity instead of a linear probability model like the one used in the work of Weiner [2]. Using this we achieved to render an image in the way the visual acuity of human eye works and to reduce even more the rays that we need to calculate. In addition, to cover the empty pixels that have not been calculated, we propose that instead of re-projecting or keeping track from the previous frame to copy from their neighbor pixel and by using a post-process effect like Gaussian blur to cover the imperfections that arise to the frame. This allows us to fill the black spots of the frame but without keeping a buffer over previous frames resulting in memory loss. Additionally, by using Gaussian blurring we are using the pixels of the current frame, eliminating potential misleading artifacts from previous frames, which would result in a fragmented image.

We tested our technique in a user evaluation and concluded that our technique has significant increase in performance as the SPP increase making it efficient, while the users did not perceive significant reduction on the overall image. The user evaluation has yielded good results, even with the technical difficulties we engaged, making it feasible for use in a Virtual Reality environment.

As far as we are aware, the presented technique is the first approach aiming to create a foveated ray-tracing rendering application, therefore, our approach is novel and has large room for future work and optimisations.

6.2 Future work

As this work covers a relatively new area of expertise with few publications, mostly in the last couple of years, there are multiple potential ideas for future work.

This thesis has been mainly focused on the construction of a foveated ray-tracing model using the visual acuity. In our work we used a basic gaming hardware setup to execute our application. An alternative setup we would consider interesting is using two GPUs, one for each eye. In order to do that, the GPUs must process data in parallel for each frame produced, commonly achieved using a Scheduling Linked

Interface (SLI). Sadly, OptiX does not currently support SLI processing, and optimising our code to work in such a system could greatly change the performance of our algorithm.

Another idea would be to create a foveated rendering technique that combines ray-tracing and rasterization rendering. This idea may work by having ray-tracing used for transparent materials and rasterization for opaque materials. In addition, we could have different levels of detail to reduce the intersection cost for ray tracing for each region of the foveated rendering (foveal, periphery and outside zone).

A good extension of our work would be to add support for rendering transparent materials. Currently, our method only supports opaque and reflective materials, which are simpler to calculate.

Finally, we could reduce the complexity of our environment, which would allow to increase the virtual light sources and make a more realistic scene.

Furthermore, the HMD used in this thesis is outdated based on today's standards. The use of the new generation of HMDs, Like the HTC Pro Eye, will improve the quality of our rendered frames due of the higher resolution and the high dynamic color (HDR) they provide. This may also require more computational cost per frame due to the higher resolution, reducing our method's performance. In addition, the newest eye-tracking system will have lower latency and higher accuracy in the eye tracking process compared to the eye-tracking device we used in this thesis, which will significantly improve the perceived image when using foveated rendering.

Finally, the performance of our foveated ray-tracing rendering technique has not been compared with other approaches on foveated rendering. The main reason for that was their high complexity and the lack of time to implement them for our system. In the future, by reproducing the previous approaches and running comparative evaluation between ours and them (benchmark and user evaluation) using the same equipment and API tools, we can compare them to find which technique is more efficient to use.

7 Bibliography

References

[1] Brian Guenter, Mark Finch, Steven Drucker, Desney Tan, and John Snyder. Foveated 3d graphics. *ACM Trans. Graph.*, 31(6):164:1–164:10, November 2012.

[2] Martin Weier, Thorsten Roth, Ernst Kruijff, André Hinkenjann, Arsène Pérard-Gayot, Philipp Slusallek, and Yongmin Li. Foveated real-time ray tracing for head-mounted displays. *Computer Graphics Forum*, 35(7):289–298, October 2016.

[3] Jonghyun Kim, Youngmo Jeong, Michael Stengel, Kaan Akundefinedit, Rachel Albert, Ben Boudaoud, Trey Greer, Joohwan Kim, Ward Lopes, Zander Majercik, and et al. Foveated ar: Dynamically-foveated augmented reality display. *ACM Trans. Graph.*, 38(4), July 2019.

[4] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. Optix: A general purpose ray tracing engine. *ACM Trans. Graph.*, 29(4):66:1–66:13, July 2010.

[5] Rachel Albert, Anjul Patney, David Luebke, and Joohwan Kim. Latency requirements for foveated rendering in virtual reality. *ACM Trans. Appl. Percept.*, 14(4):25:1–25:13, September 2017.

[6] Arthur Appel. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 37–45, New York, NY, USA, 1968. ACM.

[7] Robert A. Goldstein and Roger Nagel. 3-d visual simulation. *SIMULATION*, 16(1):25–31, 1971.

[8] Turner Whitted. An improved illumination model for shaded display. *SIGGRAPH Comput. Graph.*, 13(2):14–, August 1979.

[9] James H. Clark. Hierarchical geometric models for visible surface algorithms. *Commun. ACM*, 19(10):547–554, October 1976.

[10] Timothy L. Kay and James T. Kajiya. Ray tracing complex scenes. *SIGGRAPH Comput. Graph.*, 20(4):269–278, August 1986.

[11] Steven M. Rubin and Turner Whitted. A 3-dimensional representation for fast rendering of complex scenes. *SIGGRAPH Comput. Graph.*, 14(3):110–116, July 1980.

[12] J. Goldsmith and J. Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications*, 7(5):14–20, May 1987.

[13] A. S. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, 4(10):15–24, Oct 1984.

[14] Anjul Patney, Marco Salvi, Joohwan Kim, Anton Kaplanyan, Chris Wyman, Nir Benty, David Luebke, and Aaron Lefohn. Towards foveated rendering for gaze-tracked virtual reality. *ACM Trans. Graph.*, 35(6):179:1–179:12, November 2016.

[15] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast bvh construction on gpus. *Computer Graphics Forum*, 28(2):375–384, 2009.

[16] Martin Stich, Heiko Friedrich, and Andreas Dietrich. Spatial splits in bounding volume hierarchies. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 7–13, New York, NY, USA, 2009. ACM.

[17] Joseph Mandelbaum and Louise L Sloan. Peripheral visual acuity*: With special reference to scotopic illumination. *American Journal of Ophthalmology*, 30(5):581–588, 1947.

[18] https://www.scratchapixel.com/index.php?redirect

[19] https://www.oculus.com

[20] https://www.vive.com

[21] https://www.getfove.com/

[22] https://www.vedantu.com/biology/structure-of-eye

[23] https://whyamiachristian.com/2016/04/15/science-helps-me-understand-the-complexity-of-t

[24] https://www.intechopen.com/books/advances-in-stereo-vision/perception-and-reality-in-ster

[25] https://docs.unity3d.com/Manual/Lighting.html

[26] https://en.wikipedia.org/wiki/Phong_reflection_model

[27] https://en.wikipedia.org/wiki/Euclidean_distance

[28] https://subscription.packtpub.com/book/game_development/

[29] https://www.geertarien.com/

[30] https://www.researchgate.net/

[31] https://techreport.com/review/34095/popping-the-hood-on-nvidias-turing-architecture/

[32] https://developer.nvidia.com/optix

[33] https://en.wikipedia.org/wiki/Eye_tracking

[34] http://arringtonresearch.com/

[35] https://www.intersense.com/

8 Appendix

User Evaluation

Age: .....
Gender: .....
Eye Color: .....
Eye Problem: .....

Pair Test:

First non-Foveated then Foveated:

Foveated Parameters: Foveal = 5^o, Peripheral = 10^o.

- First better ☐
- Second better ☐
- Same ☐

First Foveated then non-Foveated:

Foveated Parameters: Foveal = 5^o, Peripheral = 10^o.

- First better ☐
- Second better ☐
- Same ☐

First non-Foveated then Foveated:

Foveated Parameters: Foveal = 7.5^o, Peripheral = 15^o.

- First better ☐
- Second better ☐
- Same ☐

First Foveated then non-Foveated:

Foveated Parameters: Foveal = 7.5^o, Peripheral = 15^o.

- First better ☐
- Second better ☐
- Same ☐

First non-Foveated then Foveated:

Foveated Parameters: Foveal = 10^o, Peripheral = 20^o.

- First better ☐
- Second better ☐
- Same ☐

First Foveated then non-Foveated:

Foveated Parameters: Foveal = 10^o, Peripheral = 20^o.

- First better ☐
- Second better ☐
- Same ☐

First non-Foveated then Foveated:

Foveated Parameters: Foveal = 12.5^o, Peripheral = 25^o.

- First better ☐
- Second better ☐
- Same ☐

First Foveated then non-Foveated:

Foveated Parameters: Foveal = 12.5^o, Peripheral = 25^o.

- First better ☐
- Second better ☐
- Same ☐

First non-Foveated then Foveated:

Foveated Parameters: Foveal = 15^o, Peripheral = 30^o.

- First better ☐
- Second better ☐
- Same ☐

First Foveated then non-Foveated:

Foveated Parameters: Foveal = 15^o, Peripheral = 30^o.

- First better ☐
- Second better ☐
- Same ☐

Ramp Test:

Quality when Foveated circle increased:

First sequence:

Foveated Parameters: Foveal = 5^o, Peripheral = 10^o.

- Increase ☐
- Decrease ☐

- Same ☐

Second sequence:

Foveated Parameters: Foveal = 7.5^o, Peripheral = 15^o.

- Increase ☐
- Decrease ☐
- Same ☐

Third sequence:

Foveated Parameters: Foveal = 10^o, Peripheral = 20^o.

- Increase ☐
- Decrease ☐
- Same ☐

Fourth sequence:

Foveated Parameters: Foveal = 12.5^o, Peripheral = 25^o.

- Increase ☐
- Decrease ☐
- Same ☐

Fifth sequence:

Foveated Parameters: Foveal = 15^o, Peripheral = 30^o.

- Increase ☐
- Decrease ☐

- Same ☐

Quality when Foveated circle Decreased:

First sequence:

Foveated Parameters: Foveal = 15^o, Peripheral = 30^o.

- Increase ☐
- Decrease ☐
- Same ☐

Second sequence:

Foveated Parameters: Foveal = 12.5^o, Peripheral = 25^o.

- Increase ☐
- Decrease ☐
- Same ☐

Third sequence:

Foveated Parameters: Foveal = 10^o, Peripheral = 10^o.

- Increase ☐
- Decrease ☐
- Same ☐

Fourth sequence:

Foveated Parameters: Foveal = 7.5^o, Peripheral = 15^o.

- Increase ☐

- Decrease ☐
- Same ☐

Fifth sequence:

Foveated Parameters: Foveal = 5^o, Peripheral = 10^o.

- Increase ☐
- Decrease ☐
- Same ☐

Slider Test:

User Preference:

- Foveal: .....
- Peripheral: .....