DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2020

Impact of Foveated Rendering on Path Tracing Frame Rate in Head-Mounted VR Displays

TOM AXBLAD

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Impact of Foveated Rendering on Path Tracing Frame Rate in Head-Mounted VR Displays

TOM AXBLAD

Master in Computer Science
Date: July 1, 2020
Supervisor: Björn Thuresson
Examiner: Tino Weinkauf
School of Electrical Engineering and Computer Science
Swedish title: Påverkan av Synadaptiv-rendering på Path Tracing Frame Rate i VR Headsets

Abstract

The combination of eye tracking and HMDs benefit each other in many ways, one such benefit being *foveated rendering*, which reduces latency in software made for VR especially well. Real time ray tracing and path tracing have gained interest from companies, developers and consumers after recent research and hardware improve both technologies significantly. For this thesis a path tracing rendering system made for VR is built, and an evaluation is made to find the impact of foveated rendering on the frame rate for the built system. Results show that depending on the features included and where the gaze point is located, foveated rendering provides an increase in frame rates between 1.59 and 10.22 times higher than the same system without foveated rendering. To conclude, foveated rendering provides a significant impact on frame rates for path tracing in VR. However, further iterations of the implementation and evaluation is necessary to reach more accurate conclusions to the research question.

Sammanfattning

Kombinationen av eye tracking och VR headsets ger många fördelar till båda teknologierna. En sådan fördel är *foveated rendering*, vilket reducerar latensen i programvara för VR särskilt bra. Just nu växer intresset hos företag, utvecklare och konsumenter gällande ray tracing och path tracing i realtid, efter att ny forskning och hårdvara öppnat upp nya möjligheter för teknologierna. För detta projekt byggdes ett ray-tracing-renderingssystem för VR, med vilket en utvärdering genomförts för att hitta påverkan av foveated rendering på latensen för detta system. Resultaten visar att beroende på vad som ingår i scenerna och var blickpunkten är placerad, ger foveated rendering en ökning i bildhastigheter mellan $1,59$ och $10,22$ gånger högre än samma system utan foveated rendering. Avslutningsvis konstanteras att foveated rendering ger en betydande ökning av bildhastigheterna för path tracing i VR. Dock krävs ytterligare iterationer av implementeringen och utvärderingen för att uppnå säkrare slutsatser om forskningsfrågan.

Contents

Glossary **vii**

1 Introduction **1**
1.1 Research Question 3
1.2 Delimitations 3

2 Background **5**
2.1 Rendering 5
2.2 Visual Field 6
2.3 Virtual Reality 6
2.4 Ray Tracing 8
2.5 Acceleration Structures 9
2.6 Path Tracing 9
2.7 Eye Tracking 11
2.8 Foveated Rendering 11
2.9 Ray Tracing and Foveated Rendering in Virtual Reality 12

3 Methodology and Implementations **15**
3.1 Evaluation 15
3.1.1 Scenes 16
3.1.2 Evaluation measurements 19
3.1.3 Evaluation summary 22
3.2 Implementation 22
3.2.1 Ray buffer with gaze point offset 23
3.2.2 Path tracing 25
3.2.3 Interpolating between target pixels 28
3.2.4 Merging textures 29

4 Results **31**
4.1 Control 31
4.2 Triangle Intersections 33
4.3 Multiple Objects 36
4.4 Ray Reflections 37

5 Discussion **39**
5.1 Scientific Question 39
5.2 Future of Path Tracing and Foveated Rendering in Virtual Reality 42
5.3 Method Criticism 43
5.4 Future Work 45

6 Conclusion **47**

Bibliography **48**

A Rendering Differences **52**

B Rendered Figures **58**

Glossary

compute shader

A type of shader that allows other kinds of data outside of graphics to be executed on the GPU. Examples of this include sorting a list of values, or simulating particles effects. 3

CPU

A CPU, or *Central Processing Unit*, is the hardware that executes instructions, reads and writes data to memory, and controls processes in a computer. A modern processor often has multiple cores which can work on separate tasks at the same time. 22, 23

foveated rendering

An optimisation technique which adapts quality based on where the user is looking. Better quality at the focus point, or fovea, of the user, and lower quality further away. iii, iv, 2–4, 6, 11–17, 21, 22, 28, 31, 39–43, 45, 47

fps

Fps, or *Frames Per Second*, is a measurement of how many frames per second a rendering system is able to calculate. 15, 31–38, 40, 41

gaze point

The exact point on a screen the user is looking at. iii, 3, 11–13, 16, 19–25, 28, 31, 39–41, 44

GPU

A GPU, or *Graphics Processing Unit*, is a unit of hardware in computers dedicated to accelerate the computation of visual data. A GPU can additionally be used for computing tasks in parallel if the task can be split into simple sub-tasks. 13, 22, 23, 25, 42

HMD

A HMD, or a *Head Mounted Display*, is a *Virtual Reality* headset that is mounted on top of the head. It usually provides one screen for each eye, headphones or a headphone jack, and a microphone. Most HMDs are connected to a computer by cable, but some contain a processing unit able to drive the unit without external computer. iii, 1–4, 11–13, 25, 30, 42–44

path tracing

An application of ray tracing which provides a global illumination to the rendering. Path tracing works by simulating many rays of light for each pixel and bounce them in random directions depending on the material hit in the scene. It gives a very realistic rendering result and is thus often used in animation and special effects.. iii, iv, 2–5, 9, 10, 14–16, 18, 23, 25, 26, 28, 42–45, 47

ray tracing

A rendering technique which simulates rays of light moving in a scene. The rays collect information about the scene and provides a result that may give better quality in comparison to other rendering techniques such as rasterisation.. iii, iv, 2, 4, 8, 13, 42, 43, 45

VR

VR, or *Virtual Reality*, is a technology that provides true 3D visualisations. Most virtual reality hardware also provides tracked movement in six degrees of freedom to the head and hands of the user. iii, iv, 1–7, 11, 13–15, 21, 25, 39, 42, 43, 47

Chapter 1

Introduction

The field of modern *Virtual Reality* (VR) is a technology that is rapidly evolving. During 2016 new consumer hardware was introduced in the form of PC connected, or mobile HMDs. The popularity of mobile VR products such as *Google Cardboard* or *Samsung GearVR* grew very fast, but this popularity disappeared just as quickly as it started. It seemed that VR was going to disappear immediately after it appeared, either because of disappointing hardware and lackluster software, or because of media who promised better experiences, interactions, and immersion than VR could fulfill as a technology. Some instead saw the potential of VR as a new visual medium, what it could become, and the hardships and hurdles VR needed to be surpass in order to become a stable platform for visualisation. Today in 2020, VR is starting to become a stable platform for visualisation, and has started growing in popularity again.

VR is harder to use for consumers, because it requires a larger amount of computer resources to work in comparison to classical screens. Since *Head Mounted Displays* (HMDs) uses one screen per eye, much like headphones uses one speaker per ear, the perceived resolution per eye is half of the actual resolution that the HMD uses. In comparison to the classic displays, this two screen system effectively doubles either the rendering computations, or the amount of data needed to display information on HMDs. The payoff for this is true 3D imagery, without any limitations of a cinema screen or a TV. Between the years 2016 and 2020 a lot has changed in the field of VR. New cheaper computer hardware has been introduced that reduces the initial price of VR setups that need a PC. VR HMDs have gotten higher field of views, higher resolution, higher refresh rates on its screens, and some even run without a connected PC while keeping all important features, such as the *Oculus Quest*.

And at the same time, the prices of HMDs are dropping, and user friendliness of HMDs are improving. One significant feature that has been introduced in a limited number of VR HMDs is eye tracking.

The combination of eye tracking and HMDs is desired by users, developers, and researchers alike because of how the two technologies can benefit each other in multiple ways. One such benefit is in social interactivity. Gaze movements from eye tracking hardware in combination with other social queues, or independent of other queues, can have significantly positive impacts on immersive interactions in virtual environments [8]. In comparison to hearing voices and seeing emotionless faces, social VR experiences with eye tracking allows for communication using the eyes which organically occurs during conversations. Another aspect of eye tracking that benefits VR is the use of optimisations based on where the user is looking, such as *foveated rendering*.

A separate area in computer graphics that is rising currently is the real time application of ray tracing and path tracing. Because of companies such as NVIDIA and AMD, new hardware specifically built to render and calculate rays in virtual scenes are being distributed into modern computers, and next generation game consoles such as PlayStation 5, and Xbox One Series X. This new hardware allows for significantly more rays to be used per frame of simulation, which in turn opens up the ability of using ray tracing and path tracing technologies into real time applications outside of movie effects and animated films.

The commercial use of path tracing in VR is so far very limited. Today in 2020, ray tracing can be used for simple point reflections, distance calculations, or other similar computations. Because of the complex algorithms used to simulate rays of light moving in a virtual world, calculating one or more rays for all pixels in HMDs while keeping above the recommended frame rate for VR, is a very hard problem, and path tracing in VR is magnitudes harder in comparison. However, because of the newer hardware that facilitates the use of ray tracing and path tracing. Using path tracing in VR is no longer an impossibility.

This thesis explores a *foveated path tracing rendering system* built for VR headsets. This system is optimised using methods normally used in path tracing rendering systems, such as an bounding box system for models. On top of this, the rendering system will also use the mentioned optimisation called

foveated rendering, where eye tracking hardware is used to determine where the user is looking in order to reduce resolution where the user is not looking.

1.1 Research Question

This thesis aims to expand the research area of path tracing and foveated rendering in VR, by answering the following research question

What impact on frame rate does gaze point adaptive resolution have on a photo realistic ray based rendering system for VR, in comparison to the same rendering system with non adaptive resolution?

Modern VR headsets has two screens, which usually have a resolution per screen that exceeds *HD* resolutions of 1920×1080 pixels, which makes it difficult to achieve high frame rates in VR applications. The research question is of relevance as it explores how foveated rendered path tracing impacts frame rates for an HMD. As of writing this in 2020, there has been very limited research conducted around path tracing in VR, however one notable example is the paper by Koskela et al. [12] which explores the topic theoretically and through literature studies.

1.2 Delimitations

This thesis is limited to a quantitative evaluation of a path tracing rendering system in Unity3D implemented as a compute shader. The results of the thesis are derived from a comparison between this rendering system with and without foveated rendering, and does not include qualitative data in the results. The reason why qualitative data is excluded is because perceived user quality does not relate to the scientific question this thesis sets out to answer. Foveated rendering is already a well researched topic, both in VR and in general applications, which is why perceived quality is not a focus of this study. A few examples of research in foveated rendering containing qualitative results are the papers by Guenter et al. [10], Weier et al. [25], and Roth et al. [20].

This thesis is limited to evaluating the implemented rendering system performance compared to itself numerically. In other words, the update frequency of the rendering system is the major focus of the thesis. The paper is not comparing the rendering system to other rendering counterparts, such as rasterisation,

since it does not contribute data towards the answer of the scientific question.

The reason for using a compute shader instead of a ray tracing shader is because a compute shader allows for complete control over the all steps in the path tracing algorithm, which facilitates the implementation of foveated rendering. The implemented compute shader does therefore not use the hardware made specifically for ray tracing that for example can be found in *Nvidia RTX* technologies.

The paper is limited to the evaluation of a single implementation in Unity3D. Multiple implementations in different engines may provide different results, but considering the time needed to implement a rendering system using path tracing, only one implementation is feasible for the thesis. The reason to use Unity3D specifically was to minimize time spent learning another engine, or time spent on building a new 3D engine.

This thesis is limited to the hardware available at hand. Provided by the VIC studio from KTH is the VR HMD *Vive Pro Eye*, which includes eye tracking in the HMD. The computer used to gather results from scenarios is a custom built computer containing a Intel Core i7 4770K processor with 4 cores and 8 threads with a 4.2 GHz clock speed, 16 GB of DDR3 RAM, and a Nvidia GTX 1080 graphics card.

Chapter 2

Background

This section aims to provide background for the separate technical areas necessary to understand foveated path tracing rendering for VR.

2.1 Rendering

To go from simulated geometry to something that can be visualised on a screen, a graphical calculation called rendering is used to turn the simulation into a representation that can be displayed. A rendering equation needs information about the simulation in order to compute an image. This information includes how many dimensions the simulation uses, where the simulated camera is pointed, the field of view for the camera, object geometry data, textures for geometry, lighting angles, and more. There are many rendering equations that uses different methods to compute the parameters based on what the desired result is. To achieve a more photo realistic result, equations solve the so called *rendering equation* where the light leaving a point is the sum of emitted light plus all reflected light [16]. This fundamental function dictates how a light is reflected from a point depending on multiple parameters about the material on said point. The rendering equation as shown by Kajiya [11] can be written as

$$L(x, \vec{\omega}_o) = L_e(x, \vec{\omega}_o) + \int_\Omega f_r(x, \vec{\omega}_i, \vec{\omega}_o)(\vec{\omega}_i \cdot \vec{n}) L(x, \vec{\omega}_i) d\vec{\omega}_i \tag{2.1}$$

where $\vec{\omega_o}$ is the direction of reflected light from x to the eye, $\vec{\omega_i}$ is the direction of incoming light, $L(x, \vec{\omega})$ is a function describing the light going from x in the direction $\vec{\omega}$, $L_e(x, \vec{\omega})$ is light emission from x to $\vec{\omega}$, $\vec{n}$ is the normal direction from x, and Ω is all space. Lastly, $f_r(x, \vec{\omega}_i, \vec{\omega}_o)$ is the bidirectional reflectance distribution function, or *BRDF*, it describes how a material will reflect light

based on its properties. A BRDF is limited to emission and reflection, it cannot handle subsurface scattering of light, which means that transparent or semi transparent materials needs a different function to describe its reflective properties. The rendering equation is visualised in figure 2.1.

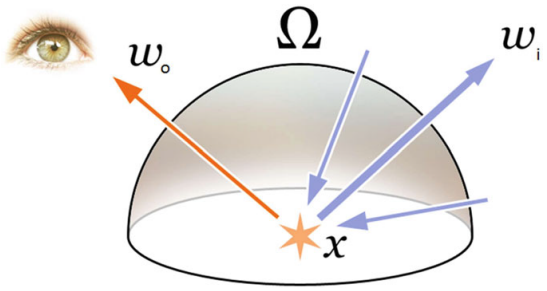

Figure 2.1: The rendering equation. From [16]

2.2 Visual Field

The visual field of humans is split up into multiple sections where visual information is picked up in different amounts. How these sections are decided changes between person to person depending on types and densities of photo receptors in their eyes [23]. Figure 2.2 visualises these sections of the eye with a possible angles for a human eye. The central $5°$ section of figure 2.2 is often called the fovea, which is where most visual information is received and is the part of the eye with the highest number of photo receptors [23]. It is mostly the macular and the near peripheral sections of the visual field which provide useful visual information [3]. Further out in the mid and far peripheral sections, both color and detail information becomes highly limited [3]. This limitation in human vision can be exploited in multiple ways to reduce rendering latency in graphical applications, which is further explained in section 2.8 about foveated rendering.

2.3 Virtual Reality

Virtual Reality (VR) is referring to a type of hardware that can be worn on the head and visualises virtual realities to the user. This hardware has one or two screens placed in front of the user's eyes which allows the user to perceive three dimensional rendered image, very similar to how a pair of headphones can play audio in close proximity to the ears of a listener. Figure 2.3 visualises how a

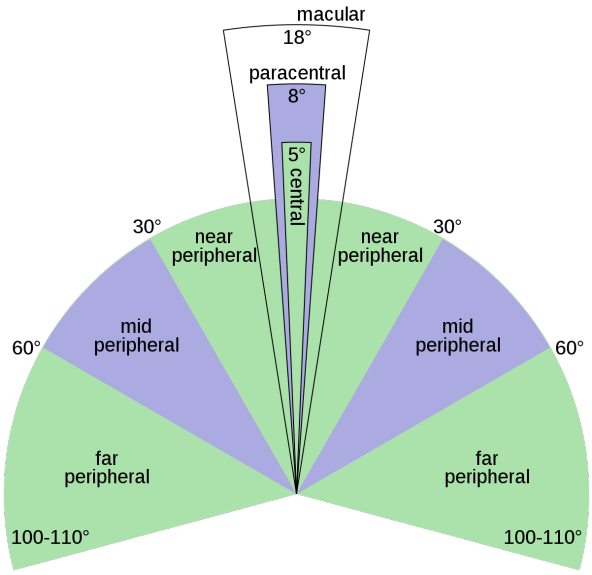

Figure 2.2: Sections of the visual field for a human eye. From [23].

feature can be rendered into two discrete image planes, which represents the screens of a VR headset. Modern VR headsets includes positioning systems which allows users to move around in a VR scene by physically moving in the real world. Also common in modern VR systems are controllers which also are tracked. This allows for movement and interaction with the user's real hands in VR. In addition to screen and positioning technologies, some other common features in modern VR hardware are microphones, and headphones. These additions allows developers to create some features in games that can only work with the mentioned hardware, such as communication systems in social applications that needs the use of microphones. Since VR makes use of cheap hardware for hand movement interactions and motion tracking, it has opened up new possibilities in areas such as gaming, media development, teaching, designing, architecture, and socialisation.

There are multiple methods to track the position of the VR headset and controllers. The most common one is called *outside in* tracking [9]. This method uses a combination of sensors and lights to accurately calculate the position of position tracked hardware. The headset and the controllers can either house the lights or the sensors, and the counterpart is then placed on a wall or a desk in direct line of sight to the headset and controllers. One outside in tracking system known for high accuracy is the *SteamVR Lighthouse* system. The lighthouse system uses spinning light sources to emit infrared light, which is picked up by sensors on the headsets and peripherals that needs to be tracked. Using

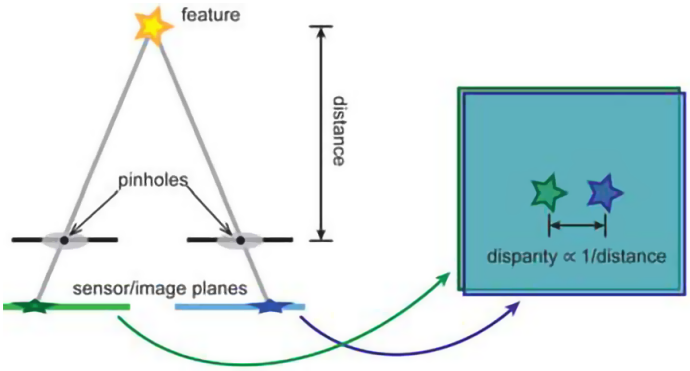

Figure 2.3: Stereo rendering of feature (star) based on two pinhole cameras. From [9].

this information, the hardware then calculate their position in relation to the lighthouses which gives a highly accurate result [14].

2.4 Ray Tracing

Ray tracing is a rendering technique where a ray of light is simulated in a virtual space. Information about the ray is gathered by computing information about the surfaces which the ray bounces between. Using ray tracing, perfect reflections and refraction between surfaces can be calculated. Even if those surfaces are dented, bent, or rounded, reflection and refraction can be calculated using ray tracing where other rendering methods, such as rasterisation, would struggle with the conditions. Ray tracing can also be used to fully render images, where one ray is cast from the camera through all pixels until they hit different points in the scene as visualised in figure 2.4.
To decide what color a pixel on the screen receive through ray tracing, the renderer has to find what specific points, called intersections, the ray hits in the simulated scene. Then for all intersections, the renderer get all available surface information to calculate the respective pixels color. From their respective intersection points, new rays can be cast towards each light source to calculate the level of illumination for that specific point, which creates shadows in the scene. Depending on how the geometry is defined there are different intersection formulas.

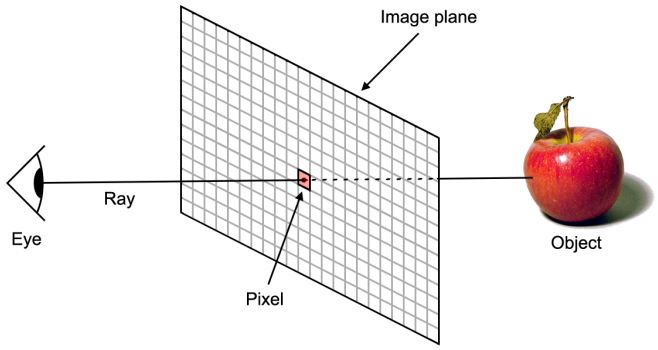

Figure 2.4: A ray that is traced through the center of a pixel in the image plane into the scene. The intersection information is used to shade the pixel red.

2.5 Acceleration Structures

Due to the high number of polygons in modern geometry and meshes, the task to find all the exact points of intersection for rays would normally be to complex for real time applications, and would hence take too much time. That is why acceleration structures were introduced. An acceleration structure is a data structure that divides the geometry of a simulated scene into different sections. Rays of light can find a subsection of all polygons that it can intersect with, using its direction and origin with this sectioned structure. Since all polygons outside of the found subsections cannot intersect with the ray, they do not need to be included in the intersection calculation, which greatly reduces calculation time.

In figure 2.5 there are two different types of acceleration structures which divides the objects in the scene into different sub structures. A KD-tree divides the total space into sectors, and labels which objects are in which sector. A BVH structure instead surrounds the objects in a hierarchy of bounding boxes which instead acts as the separators [2].

2.6 Path Tracing

Path tracing refers to a rendering system where rays can bounce in random directions as real photons would. As long as the ray has energy, the ray bounces from the initial hit point to another point in the scene, then calculates the illumination there, to then bounce again, and so on. The randomness of the bounce

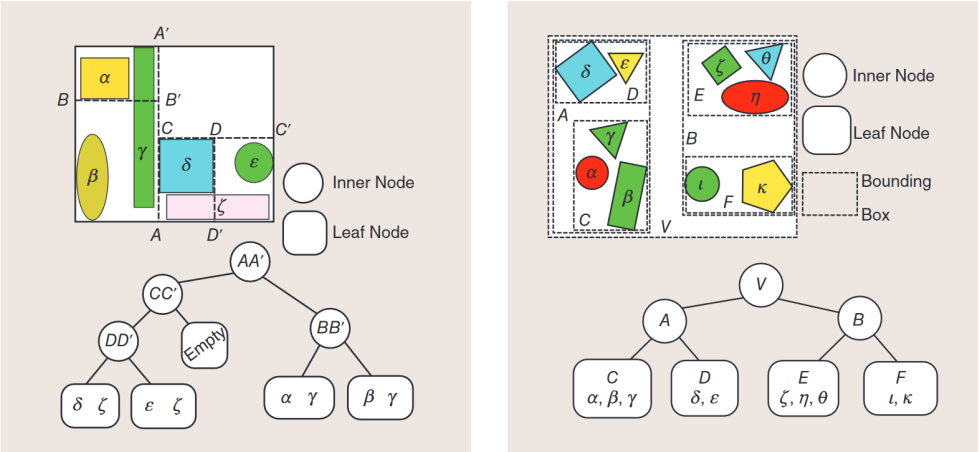

Figure 2.5: Acceleration structures which divides objects into subsets. KD-tree structure (left) and BVH structure (right). From [2].

direction is determined by how diffuse or glossy the material of the intersected object is, which is visualized in figure 2.6. A perfect mirror will bounce the ray exactly as the reflection over the surface normal. Path tracing often creates a rendered image which looks a lot more realistic in comparison to other rendering methods since it uses a more physically accurate lighting simulation and commonly concatenate values between frames. With path tracing, a number of effects can be included in the rendering without the need for extensive additional implementations, such as ambient occlusion and caustics.

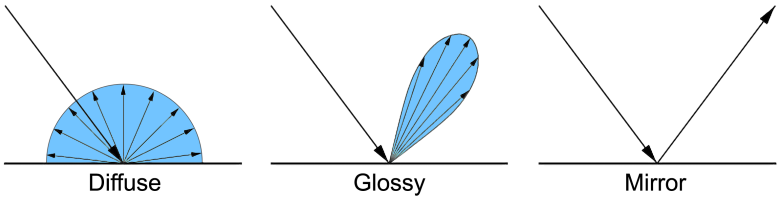

Figure 2.6: How path traced rays bounce depending on surface material. The blue sections represents all possible ray directions after intersecting with the surface. Modified image of `https://help.autodesk.com/cloudhelp/2016/ENU/mental-ray-docs/mr_docs/shaders/layering/images/dgs.png`.

2.7 Eye Tracking

Eye tracking refers to the technologies that tracks the eyes of a user and calculates where they are looking, or what they are looking at. The point where the user is looking is often called the gaze point, and will be referred to as such in this thesis. Modern iterations of eye tracking hardware have either high accuracy and lower update frequency, or vice versa, depending on the purpose of the hardware. The accuracy of Eye tracking refers to how much the hardware can minimise the positional error of the user's visual focus. For foveated rendering, a high frequency is more important than good accuracy in order to improve the effectiveness of foveated rendering [10].

Eye tracking in HMDs can be used to calculate where the user is looking in virtual space. Because VR is three dimensional by nature, a singular gaze point can be located in 3D space inside the virtual environment. This is visualised in figure 2.7, where the gaze point is at position (x_g, y_g, z_g). From this single gaze point, two more gaze points can be defined by (x_l, y_l) and (x_r, y_r) as seen in figure 2.7. These two gaze points represents the positions where the line of sight goes through the left and right screens respectively, which are the gaze points used in foveated rendering.

2.8 Foveated Rendering

Foveated rendering is a optimisation technique where the rendering system reduces the rendered fidelity outside of the user's gaze point. The gaze point of the user is in the middle of their fovea, as discussed in section 2.2, which is why the technique is called foveated rendering. As described in section 2.2, the eye can be divided into multiple sections where the eye receives different amounts of visual information based on the angular distance from the fovea. This information can then be used to decrease the rendered fidelity for areas outside of the fovea, where the fidelity decrease on the screen is dependant on the distance from the gaze point. There are different methods of foveated rendering where different aspects of fidelity are adapted. One common use of foveated rendering is to blend between higher to lower resolution images of different sizes. This method is visualized in figure 2.8. Another method is to adapt the polygon count of mesh models based on the gaze point.

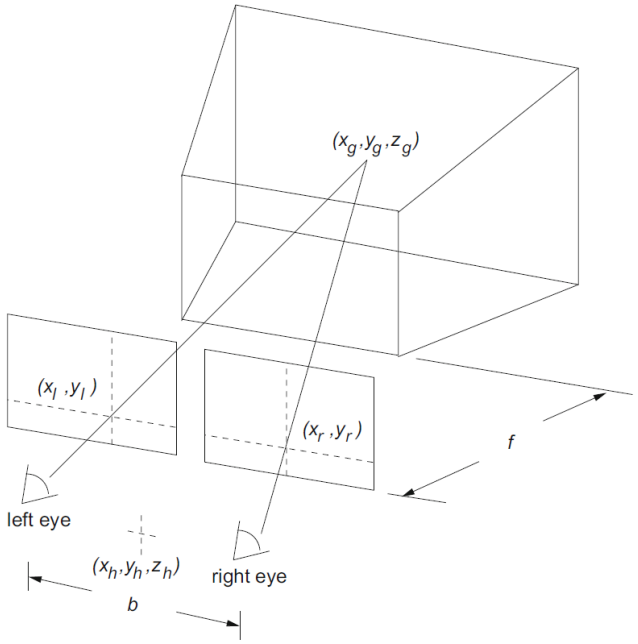

Figure 2.7: Stereo eye tracking inside a *Head Mounted Display* (HMD). From [3].

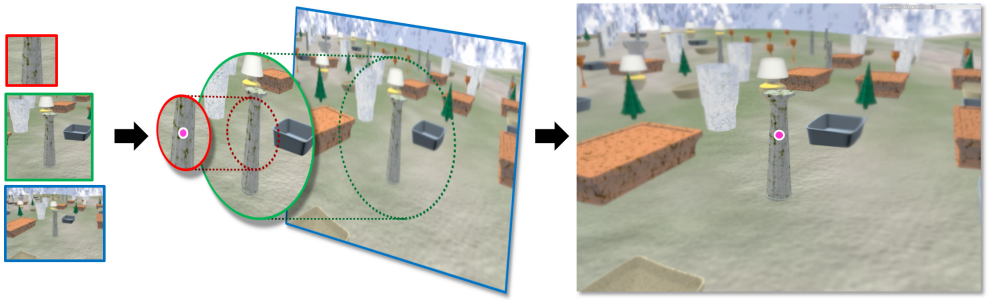

Figure 2.8: A foveated rendering of a scene using three layers Red layer is high resolution, green medium resolution, and blue low resolution. From [10].

2.9 Ray Tracing and Foveated Rendering in Virtual Reality

Early rendering systems that utilised foveated rendering did so without the eye tracking component, instead assuming a gaze point of the user [7], or by using attention prediction based on the rendered content [26]. In 1997 Watson et al.

[24] used this simplified foveated rendering inside a HMD to measure how search performance was affected by lower visual complexities and color detail in the periphery of the user, the result of this study showed that color or spatial visual complexities could be lowered by almost half without reducing search performance.

Instead of using limited prediction models, research in foveated rendering instead started relying on eye tracking to get a more reliable gaze point. An early look at modern foveated rendering systems was made by Levoy and Whitaker [13] where a head mounted eye tracking system was used together with a normal monitor that the user looked at. Their conclusion showed that while users was aware of the detail drop off thresholds, the high resolution section of the screen followed the users gaze point to a promising degree. Later use of eye tracking allowed for the use of foveated rendering in 3D graphics [15], as well as other types of applications, such as controlling the level of detail in gaze-contingent displays [4]. The use of foveated rendering with rasterization was initially complicated since rasterization can only render for a fixed resolution, this issue was however solved by the use of deferred shading systems. One such system was introduced by Stengel et al. [22].

Eye tracking inside HMDs was used in 2001 [5] for inspection training, and again in 2015 [21] where the focus instead was reducing the costs of the eye tracking system. Since then there have been numerous uses of eye tracking and foveated rendering in HMDs for different research topics, such as finding latency requirements [1], where to disable rendering for unseen parts of the HMD screens [19], or how to combine lens astigmatism optimisation with gaze point rendering to achieve even higher frame rates [18].

Since HMDs in general use two high resolution screens, rendering for VR is a challenge even when using less complex rendering methods. Additionally, ray tracing has long been held back because of low performance in comparison to rasterisation, even though it provides some distinct advantages when it comes to effects. Achieving ray tracing in VR is therefore an even more challenging task. Weier et al. [25] shows in their paper from 2016 that real time ray tracing in VR can be achieved at higher frame rates, which they made possible by using a combination of foveated rendering and a re-projection sampling method where data from previous frames are extracted and reused for the current frame. Another way of achieving real time ray tracing in HMDs is to make use of multiple GPU devices and a parallel Open-CL ray tracer, as done

by Fujita and Harada [6].

A theoretical evaluation of the impact path tracing has on VR technologies was done in 2016 by Koskela et al. [12]. The paper is a literature review of existing techniques that can be used to optimise path tracing with foveated rendering. The paper conclude that a theoretical ray path reduction of 94% is possible, and argues that path tracing will most likely meet the rendering requirements of VR in the near future.

Chapter 3

Methodology and Implementations

This chapter details the methodology and systems necessary to evaluate the research question and arrive at the results in Chapter 4. Section 3.1 describes how the scientific question will be evaluated, and how the evaluation will be presented in this thesis. Section 3.2 reveals how this system is implemented to fulfill the requirements set by the evaluation, and exactly how it functions in great detail.

3.1 Evaluation

In order to address the question introduced in section 1.1, an evaluation between path tracing systems is made where at least one of the systems include foveated rendering. Because of this, two systems are used in the evaluation, one with foveated rendering, and the other without. The main characteristic compared between these two systems is execution latency, where latency describes how much time a process takes to execute, often in milliseconds. In the topic of rendering, a common format to describe latency as is *Frames Per Second*(fps), where fps is measured as

$$f_{FPS}(l) = \frac{1000}{l} \tag{3.1}$$

where l is the latency in milliseconds of the rendering system. By comparing the differences in fps between the systems, a conclusion can be made for how big of a significance the foveated rendering optimisation provides to frame rates of the first system compared to the second.

The two rendering systems are made for VR in order to correctly evaluate the

scientific question. This means that each system needs to render two images for each update of the simulation, which doubles the latency for each system. It is hence of importance to minimise as many latency inducing factors as feasible to reduce any advantages that foveated rendering may have over systems with naive implementations.

There are multiple factors in a scene that may introduce more latency to the rendering systems. These factors exists because of the nature of path tracing, which is discussed in section 2.6. To include these different factors in the comparison between systems, multiple types of scenarios are used in order to evaluate the systems. The latency factors that is examined and targeted with these scenarios are

- Triangle intersections
- Intersections with multiple objects
- Reflective surfaces
- Gaze point of the user

All of these latency factors except for the gaze point affects both systems, and hence three types of scenarios are made based on these factors. The reason why the gaze point specifically is a latency factor is because of the adaptive fidelity depending on distance from the gaze point, as explained in 2.8. To include this factor in the evaluation, each scenario is evaluated with two gaze point targets, the first gaze target is directly in the middle of the screen where the main features of the scenario are most often placed, such as a single or multiple objects. The second gaze target should be away from the features of the scenario to keep them in the periphery of the user, and should hence be a corner of the screen. To keep consistency between all scenarios, the decided second gaze target is in the top left corner of the screen for all scenarios.

3.1.1 Scenes

There are multiple scenes used to evaluate the different latency factors, in this section all five scenes used for the evaluation will be introduced and explained. All scenes are meaningful to the evaluation for the three defined scenarios, which is discussed in later sections for each respective scene.

Control Scene

One essential scene for the evaluation is a control scenario. In this scene, there is nothing except for a background. This scene is important for the evaluation because the results show how the systems behaves without any sort of latency factors included. If there are different results in this scene for the different systems, it is necessary to have that data to discuss.

Triangle Intersection Scenes

The first item in the list of latency factors is triangle intersections. Triangle intersections affect latency as the intersection algorithm is computationally expensive. Three dimensional objects in modern real time applications uses thousands of triangles, if not many more. For a naive implementation, all triangles need to be intersected for all rays to find the closest intersections. The difference between the systems reveal how foveated rendering facilitate triangle intersection cost. Two scenes are used to evaluate this scenario, the first scene uses an object with 292 triangles, a rendering of the scene can be seen in figure 3.1.

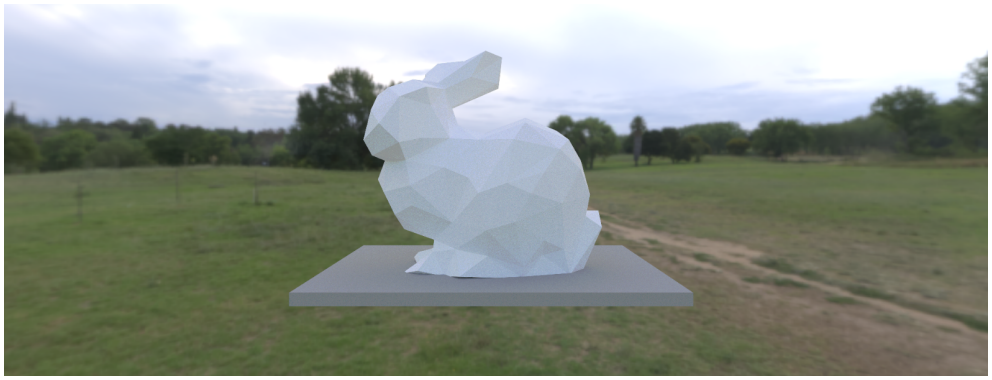

Figure 3.1: Scene with a bunny as object. The object has 292 triangles.

The second scene is also responsible for the triangle intersection factor. It is very similar to the first scene, the difference is that a mesh object with 4968 triangles is used instead. The reason why the same scenario is repeated using an object with more triangles is to be able to compare the results from the first scene with the second. This difference is discussed as it provides information about how the results for the two systems change for two objects with a big difference in triangles. The second scene can be seen in figure 3.2.

Figure 3.2: Scene with a bunny as an object, containing 4968 triangles.

Multi-Object Intersection Scene

Intersections with multiple objects are relevant to the evaluation because of how the two systems handle separate objects. Since this is an important factor in latency, a scene with multiple objects is used to evaluate the systems. The scene used can be seen in figure 3.3. The scene uses 441 axis aligned cubes.

Figure 3.3: Scene with 441 objects in total. All objects used are axis aligned cubes.

The reason why axis aligned cubes is preferred over triangle mesh objects is because triangle intersections is not of interest in this scenario, and it is far easier to render an axis aligned cube in comparison to a triangle mesh object.

Reflection Scene

The final latency factor in path tracing rendering is reflections. To correctly evaluate this factor between the two systems a scene is made which pushes the limits of reflections in the systems. The scene made is a mirror room where the walls are not only mirrors, but also slightly illuminating, this is to reduce the

effect of rays loosing energy for each bounce on mirrors, and also to illuminate the inside of the mirror room. The scene is visualised in figure 3.4. The reason

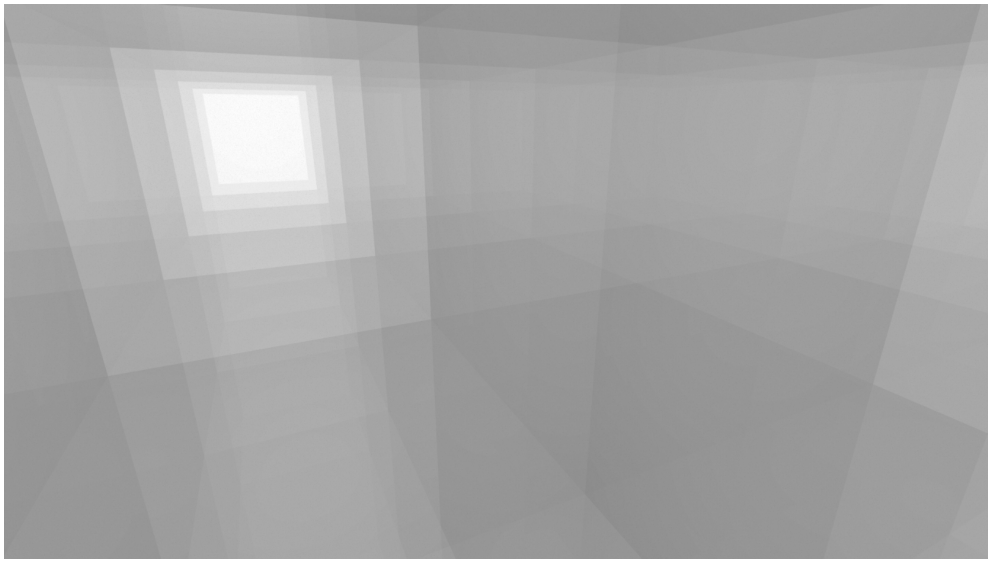

Figure 3.4: A mirror room with slightly illuminated mirrors. The rendering makes the room appear multiple times in different directions because of the reflective walls.

why the reflection latency factor is necessary for the evaluation tests is because each time a ray intersect with something in a scene it will bounce, which means that new rays needs to be calculated each time a ray intersects an object until the ray loses all energy. A mirror creates a perfect reflection, and thus the rays does not loose any energy, instead they bounce until a set maximum bounce limit of eight is reached. Since the mirror walls also increases the energy of the rays for each intersection, the rays all bounce until eight bounces is reached, and thus the reflection latency is as high as possible.

3.1.2 Evaluation measurements

The foveation system uses multiple layers with different densities of targeted pixels for each layer. These foveation layers covers the different visual fields of the human eye as shown in figure 2.2. The visual fields of a human eye vary slightly from person to person, and hence the angular size of the layers will be padded as slightly above the average sizes of the parts of the visual fields. Four layers in total are used to evaluate the foveated system, and of these four, the three first layers change position based on the gaze point of the user, while

the last layer act as a background layer. Foveation layers are discussed in more detail in 3.2.1. For the moving layers, the angular sizes from smallest to largest are $6°, 10°$, and $20°$. These sizes are chosen in order to cover the central, para-central, and macular areas respectively of the visual fields, which are the major fields used to perceive detailed visual information [3], this is discussed in more detail in section 2.2. The final layer covers the rest of the visual field of the user. The relation between angles in the foveation layers can be seen in figure 3.5.

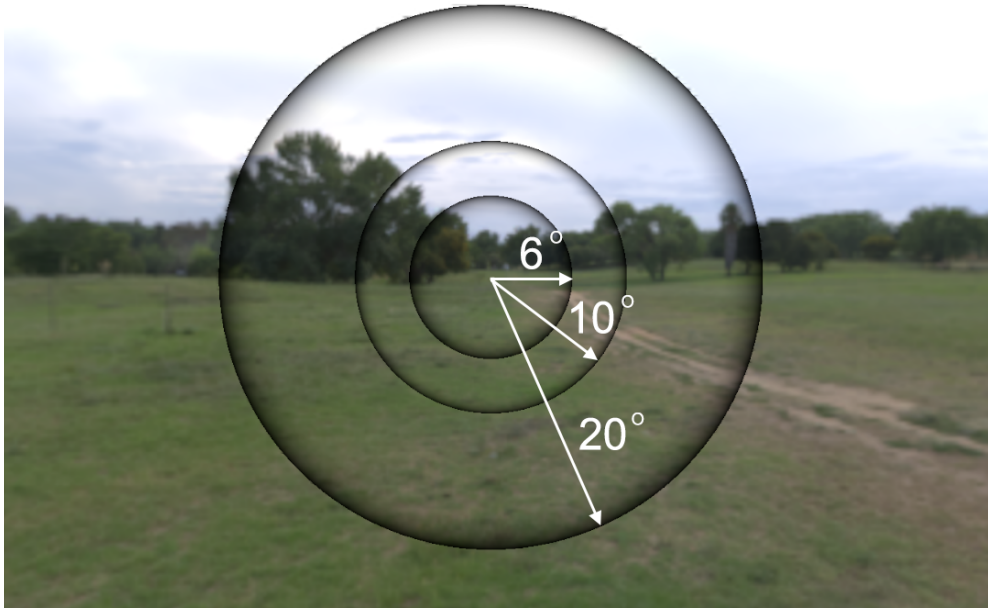

Figure 3.5: The foveation layers used in the evaluation of the foveated system. The angles are $6°, 10°$, and $20°$ respectively. The dark edges between the layers are the blend mask between layers as discussed in section 3.2.4.

The different foveation layers have different target pixel densities which decides how much of the layer is rendered and how much is not. The density is based on how fast the perceived visual information declines the further out from the gaze point a point is, which is discussed in the papers by Guenter et al. [10] and Weier et al. [25]. This density can be described as target pixel distance for a specific foveation layer, where the distance is measured between the target pixels for the foveation layer, with the unit length as the length of the side of a pixel. This concept is described in more detail in section 3.2.1.

For simplicity, a distance of $1, 2, 3, 4$ is used for the three respective foveation layers and the background layer.

Each target pixel may have one or more rays contributing to the pixels final result. Each of these rays are referred to as pixel samples and is discussed in more detail in section 3.2.2. As argued in the paper by Fujita and Harada [6], 16 or more samples should be used as a minimum to avoid visual artifacts in the rendered image. Because of limitations in the Unity3D engine, no more than four samples per pixel can be used, and is hence the chosen number of samples per pixel for all evaluation setups.

The results from the evaluation as shown in chapter 4 are presented in two types of visualisations, one is the frame rate for all measurements each frame, and the other is average frame rate for all types of measurements with the standard deviations. For these visualisations, three types of values are highlighted, where these three types are the regular rendering, the foveated rendering with forward gaze point, and the foveated rendering with corner gaze point. The first visualisation presents all measurements of the three types in a line graph, and the second visualisation is a custom graph made with dots and whiskers. A line graph with all measurements provides a detailed view of how the frame rate varies for each frame, which reveals the effect certain frames can have on the average values and how the system behaves over time. The average frame rate visualisation instead reveals a general performance of the rendering system, which reduces visual complexity and makes it easier to understand and discuss the presented information.

The VR headset used for the evaluation has a static screen frequency of 90Hz. To avoid *tearing* artifacts in the headset, the VR software truncate application update frequency to the highest possible factor of 90, depending on the current frame rate, such as 45, 30, 22.5, and 15 frames per second. Therefore, to decrease the risk of presenting truncated results, and decrease the risk of anomalies due to non-static gaze points, all test are performed for 100 frames, and tests that uses foveated rendering are measured 10 times, which provides enough data to find an average between test values, to avoid the issue with truncated values, and to avoid gaze anomalies. The tests without foveated rendering are tested once for each scene over 100 frames.

3.1.3 Evaluation summary

There are four scenarios which are represented in five scenes for the evaluation process. For these five scenes, there are two systems evaluated, one system with foveated rendering and the other without. The system using foveated rendering is evaluated in two ways, one with a gaze point in the middle of the screen, and one with a gaze point in the top left corner of the screen. The foveated system uses four foveation layers, where the three layers moving based on gaze point are $6°, 10°$, and $20°$ degrees in radius. The four layers uses $1, 2, 3, 4$ target pixel distances respectively. Both the foveated and regular system uses four samples per targeted pixel. There are hence 15 unique types of tests, where 10 types uses the foveated rendering system, and the other 5 uses the regular one. All tests are performed over 100 frames, where foveated test types are performed 10 times to avoid anomalies. The results from the 15 types of tests are presented using two different visual formats, one which includes all measurements for each scene, and one which displays the average values for each scene with the standard deviation.

3.2 Implementation

The rendering system used to answer the research question is split into several sections which each describes a vital part of the rendering system. The order of sections are the same as the order of steps executed for the rendering system.

One important element of the implementation are *foveation layers*, which are discussed in section 2.8. Each foveation layer has a unique radius measured in degrees, higher layers have smaller radius and vice versa, which is visualised in figure 3.5.

Different parts of the system are either computed on the *Central Processing Unit* (CPU), or the *Graphics Processing Unit* (GPU). The steps executed on the GPU are processed in *compute shaders*, which are programs that allows for many different types of calculations on the GPU. All compute shaders makes use of the parallel processing nature of GPUs, resulting in a much lower latency in comparison to calculating the same task on a CPU, which has a relatively small amount of threads that can be utilised for parallel computing. Everything is not calculated on the GPU as the scripts used to sync and dispatch the compute shaders need to be executed on the CPU. A flow chart of the rendering system can be seen in figure 3.6.

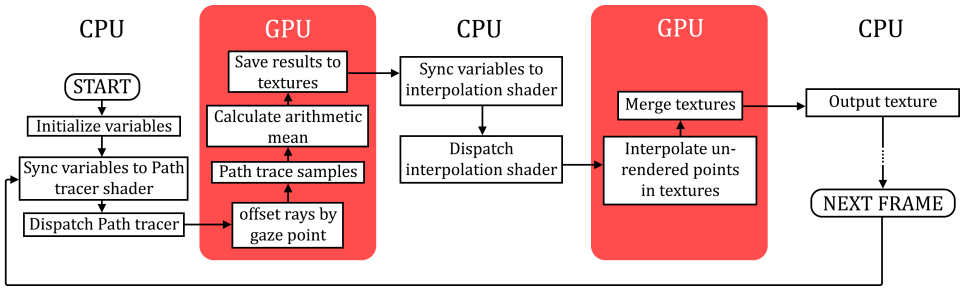

Figure 3.6: A flow chart of the rendering system. The red parts indicate that the specific steps are computed on the GPU using parallelism. All other steps are computed on the CPU.

3.2.1 Ray buffer with gaze point offset

The path tracing shader is executed on the GPU using a ray buffer, foveation layers, and a gaze point. Each foveation layer has a texture, where all rays belonging to that layer renders to. Each ray in the ray buffer is computed by a unique GPU thread, where each thread finds its foveation layer, and consequently its texture, by comparing its thread index against a buffer containing the index ranges for the different foveation layers. Each thread gets values from the ray buffer using its thread index. These values are the ray pixel position, and ray sample index. If multiple rays are rendering to the same pixel on the same foveation layer, they each have a unique sample index.

Each foveated layer has a set of targeted pixels where all targeted pixels are defined inside the radius of the layer. All thread pixel positions are in this set of targeted pixels. Each layer has a *distance* between its targeted pixels, where the distance is the horizontal and vertical length between one target pixel to a neighboring target pixel, as can be seen in figure 3.7. This distance is different for each foveation layer, and can be defined by

$$d(n) = \begin{cases} N - n & n > 0 \\ D_0 & n = 0 \end{cases} \tag{3.2}$$

Where $d(n)$ is the distance between targeted pixels on both the x and y axis, N is the number of foveation layers, n is the foveation layer index where $0 \leq n < N$, and D_0 is the distance between targeted pixels on layer index zero. The smallest distance is hence the highest layer which has a distance of 1, which means all pixels are targeted.
Each thread, except for the ones in layer zero, uses the gaze point information to offset its pixel position, as is shown in figure 3.8.

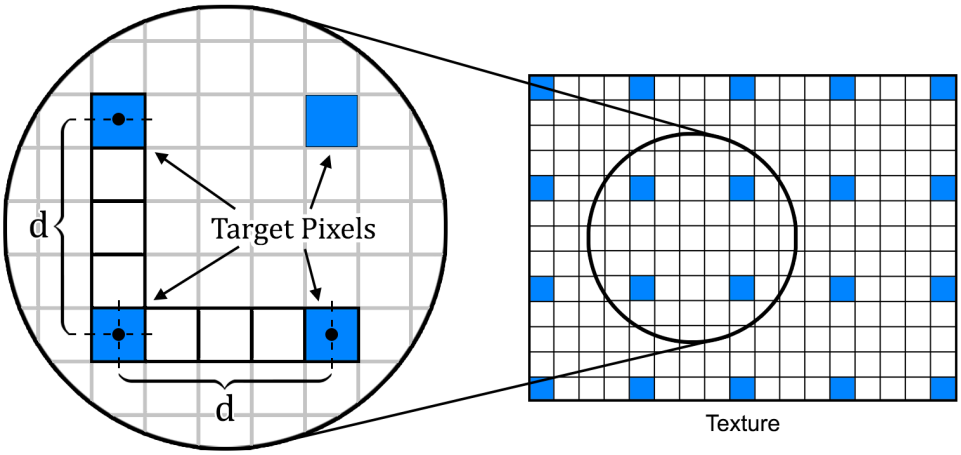

Figure 3.7: The distance d between pixels for a foveation layer. In this specific case $d = 4$. All target pixels are blue, all pixels not targeted are white.

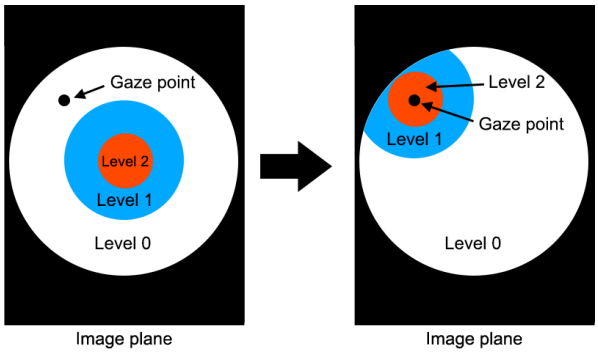

Figure 3.8: How different foveation layers are offset by the gaze point for each frame. Layer one and two in the blue and orange circle respectively are offset to place the gaze point in the center. Layer zero (white) is not affected by the gaze point. Threads outside of the radius of layer zero are not processed, hence visualised as layer one cut outside of layer zero.

The radius for foveation layer zero is used as an occlusion mask, giving the effect that threads are not processed if its pixel position is outside of foveation layer zero´s radius. All targeted pixels are not defined by the gaze point for each frame as it requires a lot more computational resources in comparison to applying the gaze offset for each frame.

A samples per pixel value of n, where $n \geq 1$ creates n threads for each targeted pixel. All treads rendering to the same pixel on the same texture calculate an average between their rendered color values and saves this mean value to the texture. More than one sample per pixel is desired as it reduces visual noise from path tracing. An additional offset away from the center of the target pixel for each sample also reduces aliasing artifacts in the rendered images.

Each of the two screens in VR uses a unique value for the gaze point that corresponds to the left and right eye. The pair of image planes for the two HMD screens are rendered individually for each frame, thus the rendering algorithm is implemented to compute one single image plane. The path tracing step of the rendering algorithm is computed in a *compute shader*. This compute shader is executed by defining thread groups with a fixed size of 128, where all thread groups computes all 128 threads contained in parallel. $G(c)$ in the equation

$$G(c) \leq \frac{c}{T} \tag{3.3}$$

describes the maximum number of thread groups can be computed in parallel, where c is the number of cores for the GPU, and T is the size of the thread groups. Since the GPU used for this thesis has 2560 cores, it can compute up to 20 threads groups, or up to 2560 rays in parallel.

3.2.2 Path tracing

Each ray in the ray buffer is calculated by a unique GPU thread, and drawn to the correct texture based on the index of the ray in the ray buffer. If a subset of all threads draws to the same texture and pixel, each individual thread contributes with one sample for a Monte Carlo integration. Hence, the color vector for the pixel is calculated by the arithmetic mean of all samples from the subset of threads. This is visualised in figure 3.9.
The general formula for Monte Carlo integration is

$$F_N \approx \frac{1}{N} \sum_{n=0}^{N} \frac{f(x_n)}{p(x_n)} \tag{3.4}$$

where $f(x_n)$ is a random sample, and $p(x_n)$ is the probability for that sample to occur. In the context of path tracing, each sample $f(x_n)$ is a result of a path traced ray. As the direction is decided by random hemisphere sampling, the

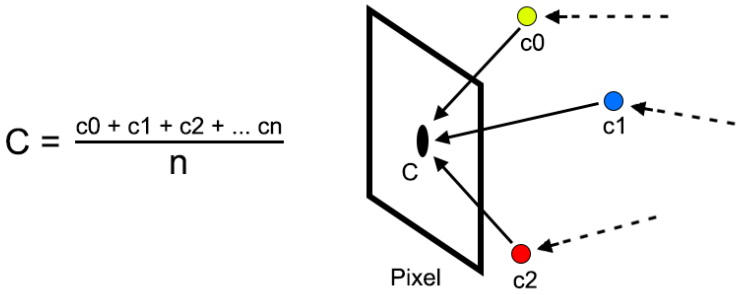

Figure 3.9: A visualisation of the path tracing algorithm. Each color $\{c_0, c_1, ..., c_n\}$ is a sample from a thread for the Monte Carlo integration, which approximates the color of C

probability of all ray samples is

$$p(x_n) = \frac{1}{2\pi} \tag{3.5}$$

In combination with this constant probability, the Monte Carlo integration put into the rendering equation, as shown in section 2.1, gives that the equation can be rewritten as

$$L(x, \vec{\omega}_o) \approx L_e(x, \vec{\omega}_o) + \frac{1}{N} \sum_{n=0}^{N} 2\pi f_r(x, \vec{\omega}_i, \vec{\omega}_o)(\vec{\omega}_i \cdot \vec{n}) L(x, \vec{\omega}_i) \tag{3.6}$$

The last part of the equation $L(x, \vec{\omega}_i)$ is all the incoming light to the intersection point x, it can be calculated by tracing a new ray in the direction from x to $\vec{\omega}_i$ and recursively calculate the rendering equation for the new intersection point.

The BRDF, which is $f_r(x, \vec{\omega}_i, \vec{\omega}_o)$ in the equation, is picked as one out of two BRDFs which focuses on diffuse reflection and a specular reflection respectively. The diffuse reflection is the Lambert Diffuse BRDF, which is described as

$$f_r(x, \vec{\omega}_i, \vec{\omega}_o) = \frac{k_d}{\pi} \tag{3.7}$$

where k_d is the albedo color vector of the intersection point. The specular reflection used is the Phong specular BRDF which was introduced by Phong [17]. The Phong specular BRDF can be described as

$$f_r(x, \vec{\omega}_i, \vec{\omega}_o) = k_s \frac{\alpha + 2}{2\pi} (\vec{\omega}_r \cdot \vec{\omega}_o)^\alpha \tag{3.8}$$

where k_s is the specular color vector of the point, $\vec{\omega}_r$ is the reflection of $\vec{\omega}_o$ on the point, and α is a value based on the smoothness of the point. Figure 2.6 shows how rays reflect in the Phong BRDF based values in α from low to high respectively. The BRDF used for each ray intersections is chosen randomly depending on the specular and albedo values of the intersection point as shown below. The diffuse and specular color vectors are both three dimensional vectors. Two values a_s and a_d are the average values for red, green, and blue, for the specular and diffuse vectors respectively. They are defined as

$$a_s = k_s \cdot [1/3, 1/3, 1/3]^\top \tag{3.9}$$

$$a_d = k_d \cdot [1/3, 1/3, 1/3]^\top \tag{3.10}$$

With this the values p_S and p_D can be defined as

$$p_S = \frac{a_s}{a_s + a_d} \tag{3.11}$$

$$p_D = \frac{a_d}{a_s + a_d} \tag{3.12}$$

where p_S and p_D both assume a value in the range $[0, 1]$ and $p_S + p_D = 1$. Using a random value $R \in [0, 1]$, the BRDF is chosen comparing the value of p_S to R as

$$L(x, \vec{\omega}_o) \approx \begin{cases} L_e(x, \vec{\omega}_o) + \frac{1}{N} \sum_{n=0}^{N} 2k_d L_i & R \geq p_S \\ L_e(x, \vec{\omega}_o) + \frac{1}{N} \sum_{n=0}^{N} k_s(\alpha + 2)(\vec{\omega}_r \cdot \vec{\omega}_o)^\alpha L_i & R < p_S \end{cases} \tag{3.13}$$

where L_i is defined as

$$L_i = (\vec{\omega}_i \cdot \vec{n}) L(x, \vec{\omega}_i) \tag{3.14}$$

As mentioned above, the rendering equation is used recursively to find the best possible value for $L(x, \vec{\omega}_o)$. The recursions continue until it has reached a recursion depth of 8, or until the energy of the ray has reached zero. The energy for each recursion step is calculated based on the energy from the last step as

$$E_{i+1} = \begin{cases} E_i \cdot p_D^{-1} k_d & R \geq p_S \\ E_i \cdot p_S^{-1} k_s \cdot f_c((\vec{n} \cdot \vec{\omega}_i) \frac{\alpha+2}{\alpha+1}) & R < p_S \end{cases} \tag{3.15}$$

where f_c clamps a value between $[0, 1]$ so that

$$f_c(x) = \begin{cases} 0 & x < 0 \\ x & 0 \leq x \leq 1 \\ 1 & x > 1 \end{cases} \tag{3.16}$$

When the end of the recursion is reached, the thread has finalised the sample, which then contributes to the final color vector for the targeted pixel.

3.2.3 Interpolating between target pixels

Different foveation layers compute results with different distances between target pixels in their respective texture, from large distances in lower layers, to small distances in higher layers. This is done in order to minimise calculations far outside of the gaze point of the user, which in turn lowers the latency for each frame. In order to merge textures in the next rendering step, all non targeted pixels necessary in the merge step needs to be interpolated for all textures. This can not be done with extra path tracing calculations, since that would negate the latency reduction benefit of foveated rendering. The only texture which does not need to be modified is the texture of the highest foveation layer, since all relevant pixels are already rendered with path tracing.

All textures already have the correct resolution in memory. For the previous step in the rendering process, a subset of all pixels in the texture are path traced in a lattice pattern, leaving non-rendered sections between all rendered points in the texture, as shown in figure 3.7.

All non-targeted pixels are approximated by *Bi-Cubic Interpolation*. Cubic interpolation uses four values in a series to interpolate values between the second and third value. The algorithm for cubic interpolation is written as

$$\begin{aligned} f_{ci}(v_0, v_1, v_2, v_3, t) = &(-\tfrac{1}{2}v_0 + \tfrac{3}{2}v_1 - \tfrac{1}{2}v_2 + \tfrac{1}{2}v_3)t^3 \\ &+ (v_0 - \tfrac{5}{2}v_1 + 2v_2 - \tfrac{1}{2}v_3)t^2 \\ &+ (-\tfrac{1}{2}v_0 + \tfrac{1}{2}v_2)t \\ &+ v_1 \end{aligned} \tag{3.17}$$

where $\{v_0, v_1, v_2, v_3\}$ are the values used for the interpolation. t is a value in the range $[0, 1]$ which describes the position of interpolation between v_1 and v_2. A t value of 0 results in v_1, a t value of 1 results in v_2, and all other values of t results in a value between v_1 and v_2.

Using cubic interpolation, any color in two dimensional textures can be interpolated using x and y positions by first interpolating four values horizontally with the x position, and then using these four values to interpolate one single value vertically using the y position. This method of using cubic interpolation in two dimensions is called bi-cubic interpolation, which is shown in figure 3.10. Using bi-cubic interpolation, all pixels in all textures receive values and can thus be merged in the following rendering step.

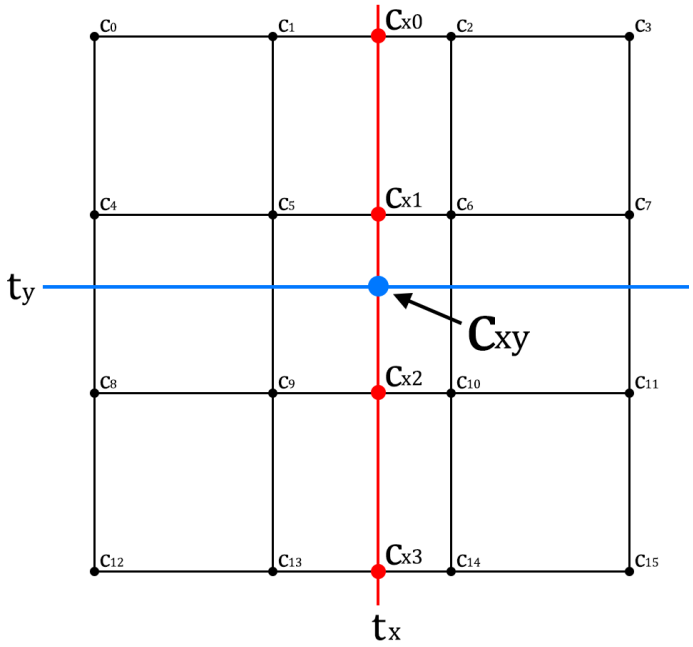

Figure 3.10: A Bi-Cubic interpolation to calculate C_{xy}. $\{C_{x0}, C_{x1}, C_{x2}, C_{x3}\}$ are each calculated by cubic interpolation with four individual values from $\{C_0, C_1, ..., C_{15}\}$, using t_x as the fifth value in the function. C_{xy} is calculated by cubic interpolation with $\{C_{x0}, C_{x1}, C_{x2}, C_{x3}\}$ using t_y as fifth value in the function.

3.2.4 Merging textures

Each foveation layer has a unique texture. To merge these textures into one, a new texture can be defined in which, for each pixel, a value is chosen from the highest possible foveation layer where the pixel position is inside the foveation layers radius. However, unless an extra interpolation pass is introduced, aliasing artifacts would be visible at the edges of foveation layers. To merge these textures while avoiding visible edge artifacts, a blend mask is applied between layers which interpolates one texture into the next texture [10]. These blend masks are visualised in figure 3.11.

For each pixel (x, y) where the value v_b of the blend mask is $0 < v_b < 1$, the color value of the pixel can be calculated with a linear interpolation func-

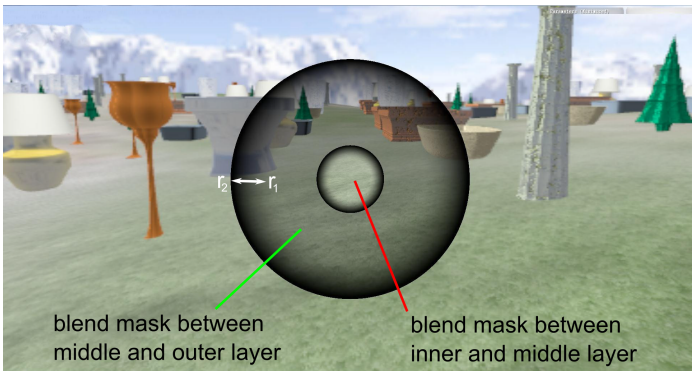

Figure 3.11: Two blend masks between middle and outer foveation layers, as well as between middle and inner foveation layers. between r_2 and r_1 the merged texture fades from the outer layer texture and the middle layer texture. From [10]

tion $L_i(x, y, v_b)$ as

$$L_i(x, y, v_b) = (1 - v_b) \cdot F_0(x, y) + v_b \cdot F_1(x, y) \tag{3.18}$$

where $F_1(x, y)$ is the color value of the pixel in the highest foevation layer which includes the pixel in its radius, and $F_0(x, y)$ is the color value of the pixel in the foveation layer one index below the highest one. All textures are merged together with linear interpolation using blend masks into a final texture. This final texture is the result of the rendering system for the current frame and is displayed in the HMD on one of the two screens.

Chapter 4

Results

This chapter will present the results gathered from the evaluation. These results are presented in two types of visualisations as described in section 3.1.2. Each section presents the results for the different latency factors introduced in section 3.1. Each section contains a small description of the scene or scenes evaluated to reach the results, a more detailed description for each scene can be found in section 3.1.1.

All graphs are measured in the metric *frame rate*, also called *frames per second* (fps), which is a measurement to find how many frames the rendering system computes during one second, as described in section 3.1. Graphs include the three different types of tests, which are the rendering system without foveated rendering, referred to as *regular*, the foveated rendering system with a gaze point in the center of the screen, referred to as *forward*, and finally the foveated rendering system with a gaze point in the top left corner of the screen, referred to as *corner*. Also included in Appendix A is figures showcasing the difference between regular and foveated rendered images for all scenes in the evaluation.

4.1 Control

The control scene is an empty scene where only the background is rendered by the system. Figure 4.1 shows the frame rate for 100 measured frames. Observe that this graph begins at 75 fps at the bottom of the y axis. In this figure it can be observed that the regular, forward and corner gaze (black, red and blue lines respectively) keep around the same frame rate of around 90 fps, with small variations of about 5 fps away from 90 fps for different frames.

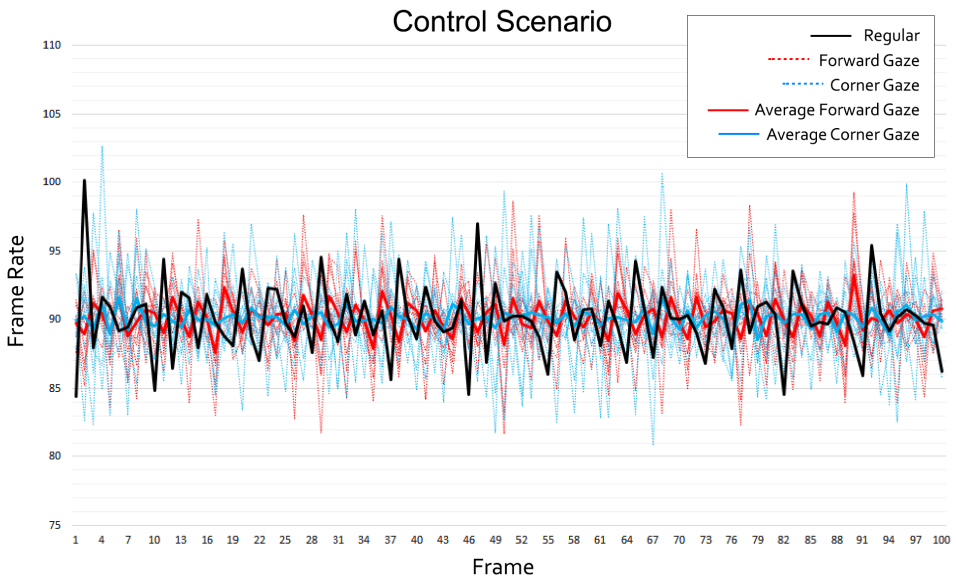

Figure 4.1: The frame rates of regular, forward, and corner tests for the control scenario. Measured over 100 frames. This graph begins with the value 75 for the lowest point in the y axis.

Figure 4.2 displays the average values for the three types of tests together with the standard deviation. Observe that the x-axis start at a value of 87 fps. It can be observed that all average values for regular, forward, and corner types all are very close to 90 fps. The standard deviation is different for the three types of data. The regular type standard deviation of 2.68 fps is a bit bigger in comparison to the two other types, where the forward standard deviation is 2.26 fps, and the corner standard deviation is 2.59 fps.

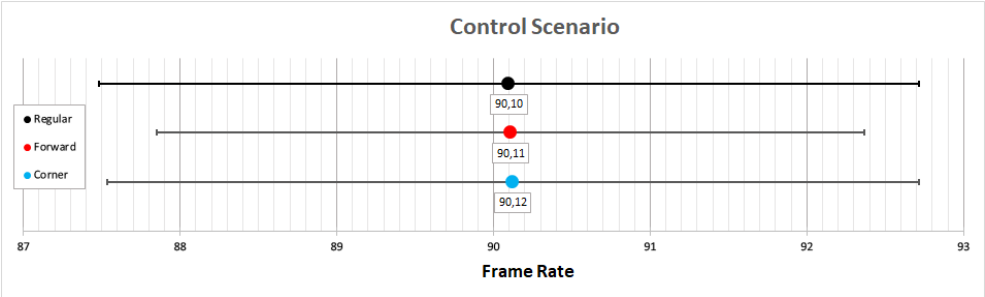

Figure 4.2: The average frame rates and standard deviations for Regular, forward, and corner tests for the control scenario. Observe that the x-axis starts with 87 at the leftmost position.

4.2 Triangle Intersections

The two scenes considering the triangle intersection latency factor both only include one object as described in section 3.1.1. One scene using an object of 292 triangles, referred to as *Low* by the graphs, and the other with an object of 4968 triangles, referred to as *High* by the graphs.

Figure 4.3 shows the frame rate for one hundred frames for scene *Low*. It can be observed that the regular frame rate is varying quite often between values of 18 fps and around 28 fps. During some frame sections, the regular frame rate is stable around 22 fps. The forward frame rates (dashed red lines) are for the most part stable at 45 fps during the tests, jumping between values of 30 and 90 fps for some frames. The corner frame rates (dashed blue lines) are centered more around 90 fps through all frames for all tests, diverging away from 90 fps by small amounts between frames. Sometimes the corner frame rates jumping to 45 fps by one frame before returning to around 90 fps the frame after.

Figure 4.4 displays the average values for the *Low* scenario for all three types of tests. It can be observed that the regular average value is 22.45 fps with a standard deviation of 2.89 fps. The forward average value is 44.89 fps with a standard deviation of 7.69 fps. And finally the corner average value is 89.69 fps, with a standard deviation of 4.8 fps.

The line diagram for the *High* scene can be observed in figure 4.5. The regular frame rate is somewhere around 3 fps for the entire test period. Forward frame rates varies between 5 and 6 fps for most of the hundred frames, with

Figure 4.3: The frame rates of regular, forward, and corner tests for the *low triangle intersection scenario*. Measured over 100 frames.

Figure 4.4: The average frame rates and standard deviations for Regular, forward, and corner tests for the *low triangle intersection scenario*.

one frame going up to 11 fps. The corner frame rates jump between around 20 to around 34 fps for many of the frames in the measurement. Most frames for corner frame rates are however contained within a range between 23 and 30 fps.

Figure 4.6 display the average values for the *High* scenario for all three types of tests. As can be observed in the graph, the regular average value is 2.81 fps

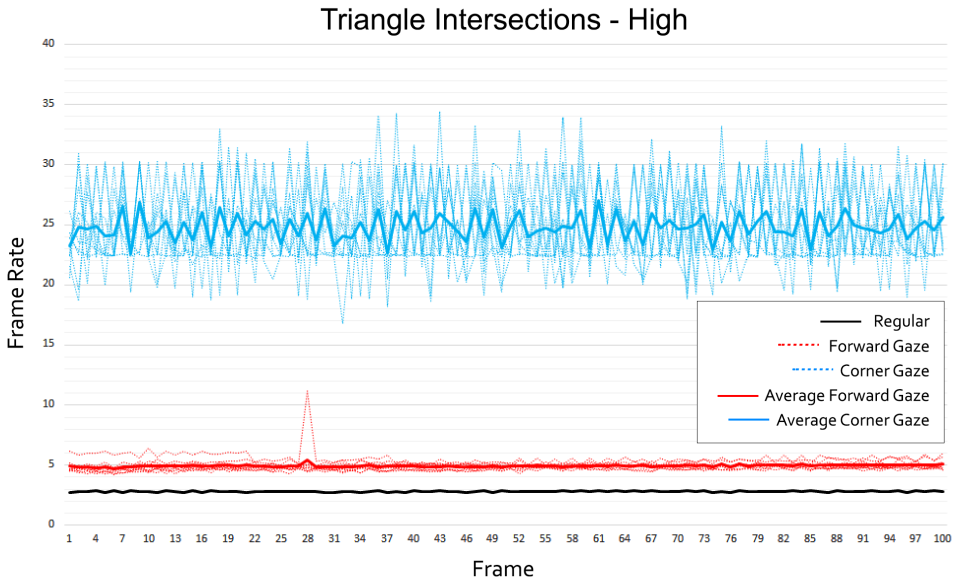

Figure 4.5: The frame rates of regular, forward, and corner tests for the *high triangle intersection scenario*. Measured over 100 frames.

with a standard deviation of 0.05 (not visible in the visualisation). The forward average value is 4.94 fps, with a standard deviation of 0.38 fps. As can be seen in the graph, the corner average value is 24.81 fps with a standard deviation of 3.4 fps, which are the highest average and standard deviation values for the three types in this result.

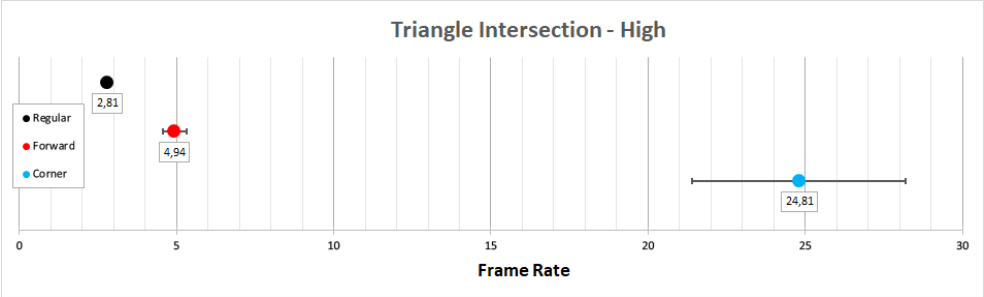

Figure 4.6: The average frame rates and standard deviations for Regular, forward, and corner tests for the *high triangle intersection scenario*.

4.3 Multiple Objects

The scene with multiple objects revolves around the latency factor where rays need to take multiple objects into consideration when calculating intersections. This is done in a single scene as described in section 3.1.1.

Figure 4.7 show the line graph for the scene with multiple objects for regular, forward, and corner test values. The regular frame rate is stable throughout the test, staying around 3 fps. Forward values vary between 15 to 18 fps through all frames. Corner values also vary through all frames, in this case in a range of about 22 to 30 fps, with some outliers above and below this range.

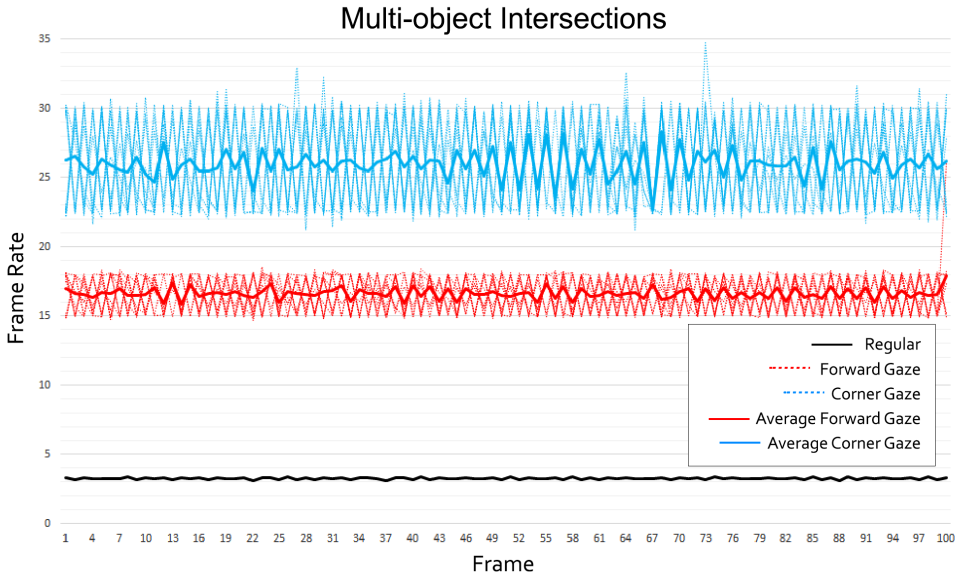

Figure 4.7: The frame rates of regular, forward, and corner tests for the *multiple object scenario*. Measured over 100 frames.

Figure 4.8 displays the average values for the three types of tests together with the standard deviation. For the regular type, The average value is 3.25 fps, and the standard deviation is 0.07 fps (not visible in the graph). The forward type has an average value of 16.61 fps, and a standard deviation of 1.4 fps. The corner type has an average value of 25.96 and a standard deviation of 3.56 fps.

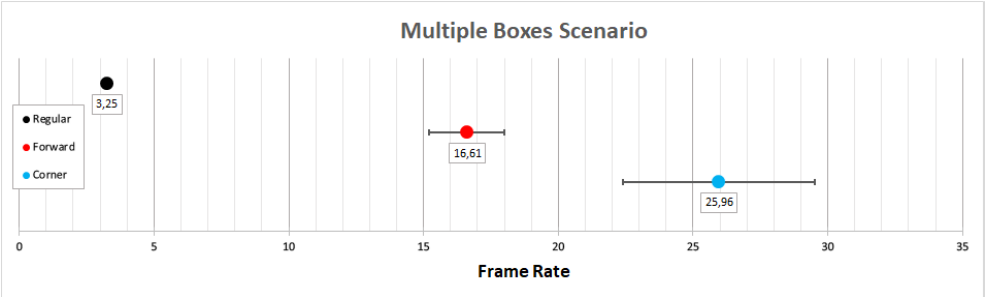

Figure 4.8: The frame rates of regular, forward, and corner tests for the multiple object scenario. Measured over 100 frames.

4.4 Ray Reflections

The reflection scene measures how the systems behave in a room where all rays are reflected until they reach the maximum amount of ray bounces, as described in section 3.1.1.

The line graph for the reflection scene is in figure 4.9. As can be observed, the regular values are centered around 11 fps. Both the forward and corner values vary for all frames between a range of about 45 to 90 fps.

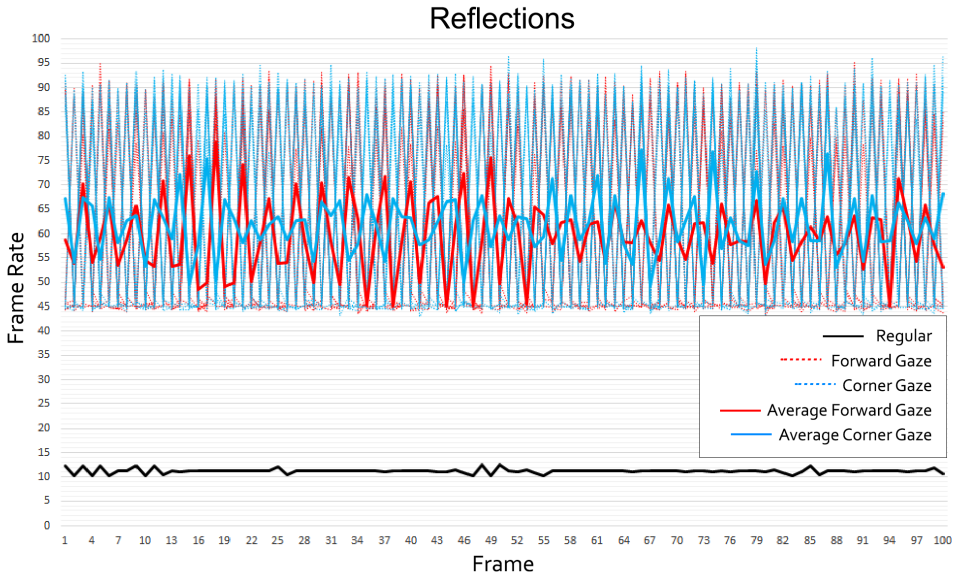

Figure 4.9

The box graph for the reflection can be observed in figure 4.10. The average regular value is 11.25 fps with a standard deviation of 0.47 fps, this standard deviation goes in line with the stability shown in figure 4.9 for the regular values. The Forward type average value is 59.57 fps, and the standard deviation is 20.32. The Corner type average value is 61.81 with a standard deviation of 21.57 fps. Both the forward and corner types varies with high values in comparison to the regular type for this scenario.

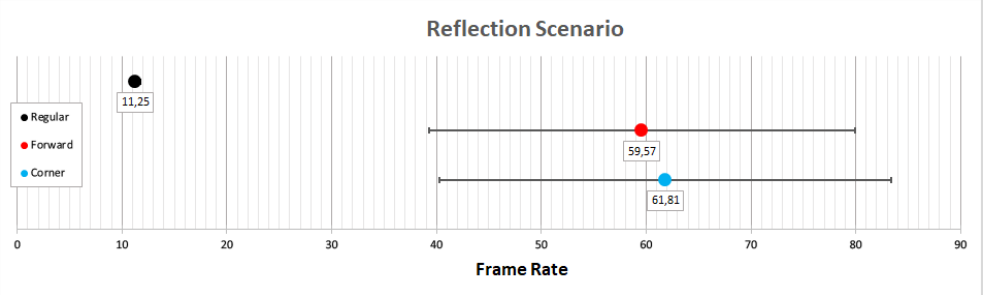

Figure 4.10: The frame rates of regular, forward, and corner tests for the reflection scenario. Measured over 100 frames.

Chapter 5

Discussion

This chapter aims to discuss the results presented in section 4, reflect about the results in relation to the research question, bring up multiple discussion points derived from the results, criticise of the methods used in this thesis, and finally present potential future work.

5.1 Scientific Question

This section raises discussion around the research question of this paper. A repetition of the full scientific question is first necessary to refresh exactly what this discussion aims to answer.

What impact on frame rate does gaze point adaptive resolution have on a photo realistic ray based rendering system for VR, in comparison to the same rendering system with non adaptive resolution?

To be able to answer this question, there is a need to find and discuss changes in frame rate performance with foveated rendering in comparison to non foveated rendering. Considering the results in chapter 4, most results show that the foveated rendering system built for this paper has a positive impact in form of performance in comparison to the non foveation counterpart. This performance increase is different for all the four different scenarios, with only the control scenario having arguably no performance gain. Foveated rendering has thus generally a positive impact on the rendering, which is expected since foveated rendering is a technique that lowers rendering latency by design. The positive impact is hence in itself not a satisfying answer to the scientific question. To reach a satisfying conclusion to the scientific question, the results for

all scenarios needs to be discussed in relation to the question in order to quantify the impact for the different scenarios. The four different scenarios handle different types of latency factors, as discussed in section 3.1, which results then reveal how foveated rendering impacts the specific factors targeted. The different impacts of the foveated rendering system is discussed in the form of *impact factor*, which is defined by the frame rate divided by the regular average frame rate for the foveated rendering cases. The *impact factor* is thus deemed a good indication of how big of an impact the foveated rendering has on frame rate for the specific scenarios.

The control scenario has approximately the same performance between all the three types of tests, with minor differences between standard deviations. The standard deviations for the different types range from 0.09 to 0.4 frames per second (fps), which can be considered small and not of significance. These results show thus that in this specific scenario, foveated rendering has no impact on the rendering latency. This is somewhat expected for this control scenario because of the lack of performance factors in the scene, as mentioned in section 3.1.1.

For the triangle intersection scenario, the impact of foveated rendering is a positive improvement for both cases in comparison to the control scenario, although with differences between scenes. In the case of the triangle intersection scene with the lower amount of triangles, it can be derived from figure 4.4 that the forward and corner average impact factors are 1.99 and 3.99 respectively. Taking standard deviations into account, the impact factor for the forward type ranges between 1.47 minimum and 2.69 maximum, while the corner impact factor is instead ranges between 3.35 minimum and 4.83 maximum. For the second scene with the higher maount of triangles, it can be derived from figure 4.6 that the forward and corner average frame rates have an impact factor of 1.75 and 8.83 respectively. With standard deviation in account, the forward impact factor ranges between 1.59 minimum and 1.93 maximum, while the corner impact factor ranges between 7.49 minimum and 10.22 maximum. This shows a significant impact on frame rate for both types of scenes. These results indicate that the frame rate improves for foveated rendering over regular rendering, even when looking directly at an object in the scene. Since this improvement is smaller in the second scene in comparison to the first for the forward gaze point, it seems as if the improvement decreases for objects with increasing triangle count. The corner average value show a greatly increased improvement in the second scene in comparison to the first, going from an av-

erage impact factor of 3.99 to 8.83 times the frame rate of the average regular frame rate. This seems to indicate that with objects of increasing triangles, the frame rate increases significantly in comparison to regular rendering as long as objects are in the periphery of the user.

For the multiple objects scenario, the average regular frame rate is 3.25 fps in comparison to the 16.61 fps of average forward, and 25.96 fps of average corner frame rates. The improvement factor for foveated rendering in this scenario is thus 5.11 for forward type, and 7.99 for corner type. The difference between forward and corner improvement factors is thus not as big in comparison to the triangle intersection scenes, which seems to indicate that foveated rendering has a bigger positive impact on object management in comparison to triangle intersections. This result is however heavily impacted by the current implementation of the acceleration structure, which could be iterated upon. The acceleration structure is discussed further in section 5.4.

For the reflection scenario results in figure 4.10, it can be derived that the forward average impact factor is 5.30 and the corner average impact factor is 5.49. Standard deviation is very high for this scenario, which gives an impact factor range of 3.35 to 7.41 to forward type, and 3.43 to 7.73 for corner type. For this case, no big difference was expected between forward and corner gaze point tests, which is exactly what the results present. The reason for the small difference is that there is no point in the scene which is more or less difficult for the renderer to compute. There is still a small improvement in corner type over forward type, and the reason for this is that some of the defined rays for the corner gaze point is outside of the occlusion mask, which is discussed in section 3.2.1. These results indicate that the reflection latency factor does significantly impact performance, and that a ray reduction by foveated rendering provides a performance increase. The exact impact factor of foveated rendering is hard to discern in this case because of the high standard deviations and unstable frame rates displayed in figure 4.9, but from the lowest to highest range, the factor can be interpreted as somewhere between 3.35 to 7.73 times higher the frame rate of standard rendering.

5.2 Future of Path Tracing and Foveated Rendering in Virtual Reality

Foveated rendering is currently not a standard technology in real time applications. The reasons for this are many, the eye tracking hardware is difficult to develop to work correctly for all types of people, the hardware is expensive, and it is often awkward or cumbersome to use. When using the HMDs of today, you already have to deal with very similar issues, which makes HMDs a great fit for foveated rendering since users does not need to sacrifice more comfort in order to add the technology necessary. It is of my belief that the future of VR and foveated rendering is very bright. It seems inevitable that foveated rendering is going to be used if it can be used, because of the seemingly guaranteed latency reduction for little to no visual sacrifice in quality. Some HMDs already has eye tracking hardware installed, such as the headset used for this thesis, the *Vive Pro Eye*, and many more HMDs including eye tracking is currently in development.

One possible consequence of foveated rendering on the industry is the quality of VR experiences exceeding classical monitors in visual fidelity. What that means is not that VR can give a unique experience, but that the perceived quality of the user is factually better in VR than outside it with foveated rendering. Of course the same effect can be adapted and used outside of VR, but that would mean adding additional eye tracking hardware to monitors and TVs which introduces higher costs, as well as security and privacy issues. VR is currently a luxury product, which means additional cost does not change the targeted audience. Additionally, eye tracking in HMDs can guarantee that only the eyes are possible to track using cameras since the sensors are placed inside the HMDs, which resolve most security and privacy issues.

Ray tracing and path tracing are very mature rendering methods used in everything from special effects, animation, to simulations. Computer hardware was for a long time simply not powerful enough to enable real time applications using ray tracing and path tracing, but in the last few years that has changed very fast. With the introduction of technologies that enables ray tracing to be computed in hardware rather than software, such as *Nvidias RTX GPUs*, or AMDs upcoming *RDNA 2 GPUs*, more and more computers and consoles will be able to include ray tracing in their real time applications. Some games such as *Minecraft* and *Quake II* already have support for real time path tracing today,

but with the performance improvement of foveated rendering, these hardware and software optimisations may lead to ray tracing and path tracing becoming even easier to implement in real time applications. As a consequence of this, the future of real time experiences might provide the same quality as modern special effects and non-real-time simulations.

As VR and eye tracking technologies become more researched, developed and cheaper, the rest of the tech industry will develop in parallel. In time, it may enable users to engage in virtual worlds with a fidelity hard to distinguish from reality, at a cost not higher than buying a modern game console. Backing up this claim is already existing standalone HMDs with all features of PC connected ones for a price of around 400€, such as the *Oculus Quest*. These technologies may change not only how people find entertainment, but also how we communicate, how we interact with computers, how we teach, and how we learn.

Foveated rendering not only enables greater quality in powerful hardware, it also allows for the same quality in cheaper and less powerful hardware. This opens up the technology for more mobile and less powerful HMDs, which may affect the environment in a positive way because of the increased power efficiency of the system. It also may open up the hardware for more people in society, moving out of the luxury product category, and into a more general electronics category, as TVs and cell phones once did.

5.3 Method Criticism

This section will describe the criticisms directed towards this study. Note that some criticism is introduced and discussed instead in section 5.4, as they reveal what can be done in the future for improved measurements, and improved implementation.

The inclusion of VR in this thesis is done because of converging technologies. Eye tracking and VR are already combined in some products on the market, and is included in current development of future VR products. However, the use of foveated rendering is not exclusive to VR, and can be used for path tracing also without VR technologies. The inclusion of VR can hence be criticised as unnecessary in order to evaluate the basic concept of the scientific question, which is the impact of foveated rendering on path tracing.

The use of Unity3D as an engine can be criticised since it can introduce extra latency factors outside the control of the thesis. Another choice could instead be the use of a custom engine specially built for the thesis. This is however not in the scope of this thesis, and therefore a more complete engine was chosen instead. Another method which can be critiqued is the use of only one engine. Multiple engines may perform differently, leading to different results that can be compared to each other. For example, one engine that could be used is the *Unreal Engine*.

The evaluation of the research question is done through a single subject who wears the HMD and performs the defined steps of the tests. This can be criticised because of the introduction of a human factor in the results, which could potentially introduce errors. Another solution to this would be to implement an automatic testing suit, which firstly places the HMD and gaze points in predefined places in the virtual scene automatically, secondly it collects all results from the rendering calculations in the correct format, and thirdly it records this data in a combined file to be processed.

The separation of the rays into threads in the compute shaders (as discussed in section 3.2.1) is made to divide and distribute computation over more threads in the GPU, which is generally what is preferred over more computation per GPU thread. However, this decision lead to the limitation of how many rays per pixels is used in the evaluation. Because Unity3D limits the number of thread groups defined per compute shader dispatch, the maximum number of rays per pixels possible for the HMD resolution is four. This issue can be bypassed by increasing the number of threads used per thread group, however, this is not a good solution to the issue because larger thread groups introduces extra latency in the implementation. It was thus decided that 4 rays per pixel is used for each frame in the evaluation, which can be criticised because a ray per pixel value at least 16 would be preferred. 16 or more rays per pixel is preferred because previous works of Weier et al. [25] and Fujita and Harada [6] show that path tracing in real time applications needs at least 16 rays per pixel to give users a perceived good render quality.

The lack of evaluation on different hardware can be criticised because of how this may affect results. Different hardware would reveal how the rendering performance changes depending on improvements or deterioration in processing power.

5.4 Future Work

In this thesis, the research question, and subsequently the evaluation, is focused solely on finding numerical impacts of foveated rendering on path tracing. This can however be expanded on to find more information about the quality of foveated rending in path tracing. Previous work in the area of foveated rendering include qualitative research as discussed in section 1.2, which is a good starting point in this extension. The topics that can be included in this qualitative evaluation includes the perceived quality for different foveation layer amounts and angles, different ray per pixel values for the layers. Another topic for future work in quality evaluations, is different sizes of ray distributions for targeted pixels.

The current implementation is made from the ground up using compute shaders. This means that it does not interface with the newest and upcoming GPUs from Nvidia and AMD. In a future iteration of this thesis, changing the implementation to use the *DirectX 12* API, or using the built in ray tracing shader in the Unity3D Engine would allow the use of this specialised hardware. This can increase the general performance of the implementation and introduce the possibility to use the implementation in real time applications targeting higher frame rates.

One topic that can be expanded upon in this project is a better acceleration structure. The current implementation has two levels of bounding boxes, one that surrounds all objects, and one for each individual object in the scene. This works as a simple version of acceleration structure, however an iteration of this current structure is needed to improve the performance. This iteration can be done in one out of two ways. Either an acceleration structure is built from scratch that intersects with the current implementation, or existing implementations is used to handle the structure. One good example of an existing implementation is the hardware from Nvidia in their latest RTX GPU architecture.

In this thesis, parameters are set to static values in order to focus more on the different scenarios which affect frame rates in different ways. To iterate and improve the quantitative evaluation of the systems, there is a need to perform more tests with more variation of these parameters. The different parameters introduced into the system include the number of foveation layers, different layer angles, rays per pixel, and the resolution of the final image. One may

also introduce more scenes for the targeted scenarios with different setups to improve the comparison of results between them.

Chapter 6

Conclusion

In three of the four evaluated scenarios, there is a clear improvement with foveated rendering over a standard path tracing rendering. The conclusion to the question is thus, excluding the control scenario, that the impact of foveated rending on frame rates ranges between 1.59 to 10.22 times higher than the non-foveated system frame rate. The impact depends on what latency inducing factors are included in the scene, where results indicate that features with stronger detrimental effects on latency leads to a lower performance increase when looking directly at the feature, but higher performance increase when the feature is in the periphery of the user. In short, for most cases foveated rendering gives a significantly positive impact on frame rate for path traced rendering in VR.

This estimated impact will change if future iterations of the project are made, which would be necessary to give a more accurate answer to the research question. An iteration of the project could include the inclusion of specialized hardware, new implementations, and variation over the system parameters.

Bibliography

[1] Rachel Albert et al. “Latency Requirements for Foveated Rendering in Virtual Reality”. In: *ACM Transactions on Applied Perception* 14.4 (Sept. 2017), 25:1–25:13. ISSN: 1544-3558. DOI: 10.1145/3127589.

[2] Alfonso Breglia et al. “Comparison of Acceleration Data Structures for Electromagnetic Ray-Tracing Purposes on GPUs [EM Programmer’s Notebook]”. In: *IEEE Antennas and Propagation Magazine* 57.5 (Oct. 2015), pp. 159–176. ISSN: 1558-4143. DOI: 10.1109/MAP.2015.2470685.

[3] Andrew T. Duchowski. *Eye Tracking Methodology*. Springer International Publishing, 2017. ISBN: 978-3-319-57881-1. DOI: 10.1007/978-3-319-57883-5. URL: http://link.springer.com/10.1007/978-3-319-57883-5.

[4] Andrew T Duchowski and Arzu Çöltekin. “Foveated gaze-contingent displays for peripheral LOD management, 3D visualization, and stereo imaging”. In: *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)* 3.4 (2007), pp. 1–18.

[5] Andrew T Duchowski et al. “Binocular eye tracking in VR for visual inspection training”. In: *Proceedings of the ACM symposium on Virtual reality software and technology*. 2001, pp. 1–8.

[6] Masahiro Fujita and Takahiro Harada. “Foveated Real-Time Ray Tracing for Virtual Reality Headset”. In: (), p. 1.

[7] Thomas A Funkhouser and Carlo H Séquin. “Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments”. In: *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. 1993, pp. 247–254.

[8] Maia Garau et al. “The Impact of Avatar Realism and Eye Gaze Control on Perceived Quality of Communication in a Shared Immersive Virtual Environment”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’03. Ft. Lauderdale, Florida, USA: Association for Computing Machinery, 2003, pp. 529–536. ISBN: 1581136307. DOI: 10.1145/642611.642703. URL: https://doi-org.focus.lib.kth.se/10.1145/642611.642703.

[9] Michael J. Gourlay and Robert T. Held. “Head-Mounted-Display Tracking for Augmented and Virtual Reality”. In: *Information Display* 33.1 (2017), pp. 6–10. ISSN: 2637-496X. DOI: 10.1002/j.2637-496X.2017.tb00962.x.

[10] Brian Guenter et al. “Foveated 3D graphics”. In: *ACM Transactions on Graphics* 31.6 (Nov. 2012), 164:1–164:10. ISSN: 0730-0301. DOI: 10.1145/2366145.2366183.

[11] James T. Kajiya. “The Rendering Equation”. In: *SIGGRAPH Comput. Graph.* 20.4 (Aug. 1986), pp. 143–150. ISSN: 0097-8930. DOI: 10.1145/15886.15902. URL: https://doi-org.focus.lib.kth.se/10.1145/15886.15902.

[12] Matias Koskela et al. “Foveated Path Tracing”. In: *Advances in Visual Computing*. Ed. by George Bebis et al. Cham: Springer International Publishing, 2016, pp. 723–732. ISBN: 978-3-319-50835-1.

[13] Marc Levoy and Ross Whitaker. “Gaze-directed volume rendering”. In: *Proceedings of the 1990 symposium on Interactive 3D graphics*. I3D ’90. Association for Computing Machinery, Feb. 1990, pp. 217–223. ISBN: 978-0-89791-351-5. DOI: 10.1145/91385.91449. URL: http://doi.org/10.1145/91385.91449.

[14] Diederick C Niehorster, Li Li, and Markus Lappe. “The accuracy and precision of position and orientation tracking in the HTC vive virtual reality system for scientific research”. In: *i-Perception* 8.3 (2017), p. 2041669517708205.

[15] C O’Sullivan, J Dingliana, and S Howlett. “Gaze-contingent algorithms for interactive graphics”. In: *The Mind’s Eyes: Cognitive and Applied Aspects of Eye Movement Research, J. Hyönä, R. Radach, and H. Deubel, Eds. Elsevier Science, Oxford* (2002).

[16] Jon Peddie. “The Continuum”. In: *Ray Tracing: A Tool for All*. Ed. by Jon Peddie. Springer International Publishing, 2019, pp. 33–64. ISBN: 978-3-030-17490-3. DOI: 10.1007/978-3-030-17490-3_4. URL: https://doi.org/10.1007/978-3-030-17490-3_4.

[17] Bui Tuong Phong. “Illumination for Computer Generated Pictures”. In: *Commun. ACM* 18.6 (June 1975), pp. 311–317. ISSN: 0001-0782. DOI: 10.1145/360825.360839. URL: https://doi-org.focus.lib.kth.se/10.1145/360825.360839.

[18] Daniel Pohl, Xucong Zhang, and Andreas Bulling. “Combining eye tracking with optimizations for lens astigmatism in modern wide-angle HMDs”. In: *2016 IEEE Virtual Reality (VR)*. Mar. 2016, pp. 269–270. DOI: 10.1109/VR.2016.7504757.

[19] Daniel Pohl et al. “Concept for using eye tracking in a head-mounted display to adapt rendering to the user’s current visual field”. In: *Proceedings of the 22nd ACM Conference on Virtual Reality Software and Technology*. VRST ’16. Association for Computing Machinery, Nov. 2016, pp. 323–324. ISBN: 978-1-4503-4491-3. DOI: 10.1145/2993369.2996300. URL: http://doi.org/10.1145/2993369.2996300.

[20] Thorsten Roth et al. “An analysis of eye-tracking data in foveated ray tracing”. In: *2016 IEEE Second Workshop on Eye Tracking and Visualization (ETVIS)*. Oct. 2016, pp. 69–73. DOI: 10.1109/ETVIS.2016.7851170.

[21] Michael Stengel et al. “An affordable solution for binocular eye tracking and calibration in head-mounted displays”. In: *Proceedings of the 23rd ACM international conference on Multimedia*. 2015, pp. 15–24.

[22] Michael Stengel et al. “Adaptive Image-Space Sampling for Gaze-Contingent Real-time Rendering”. In: *Computer Graphics Forum* 35.4 (2016), pp. 129–139. ISSN: 1467-8659. DOI: 10.1111/cgf.12956.

[23] Fanghella Valero and Alfredo José. *Eye-tracking Based Video Streaming System with Fast Quality Switching*. 2017. URL: http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-215718.

[24] Benjamin Watson et al. “Managing level of detail through peripheral degradation: effects on search performance with a head-mounted display”. In: *ACM Transactions on Computer-Human Interaction* 4.4 (Dec. 1997), pp. 323–346. ISSN: 1073-0516. DOI: 10.1145/267135.267137.

[25] Martin Weier et al. “Foveated Real-Time Ray Tracing for Head-Mounted Displays”. In: *Computer Graphics Forum* 35.7 (2016), pp. 289–298. ISSN: 1467-8659. DOI: 10.1111/cgf.13026.

[26] Hector Yee, Sumanita Pattanaik, and Donald P Greenberg. “Spatiotemporal sensitivity and visual attention for efficient rendering of dynamic environments”. In: *ACM Transactions on Graphics (TOG)* 20.1 (2001), pp. 39–65.

Appendix A

Rendering Differences

This appendix includes the differences for all scenes used to evaluate the thesis. All figures include the regular rendering, the forward gaze point foveated rendering, and the differences between them. The differences between renderings is visualised as black pixels on a white background.

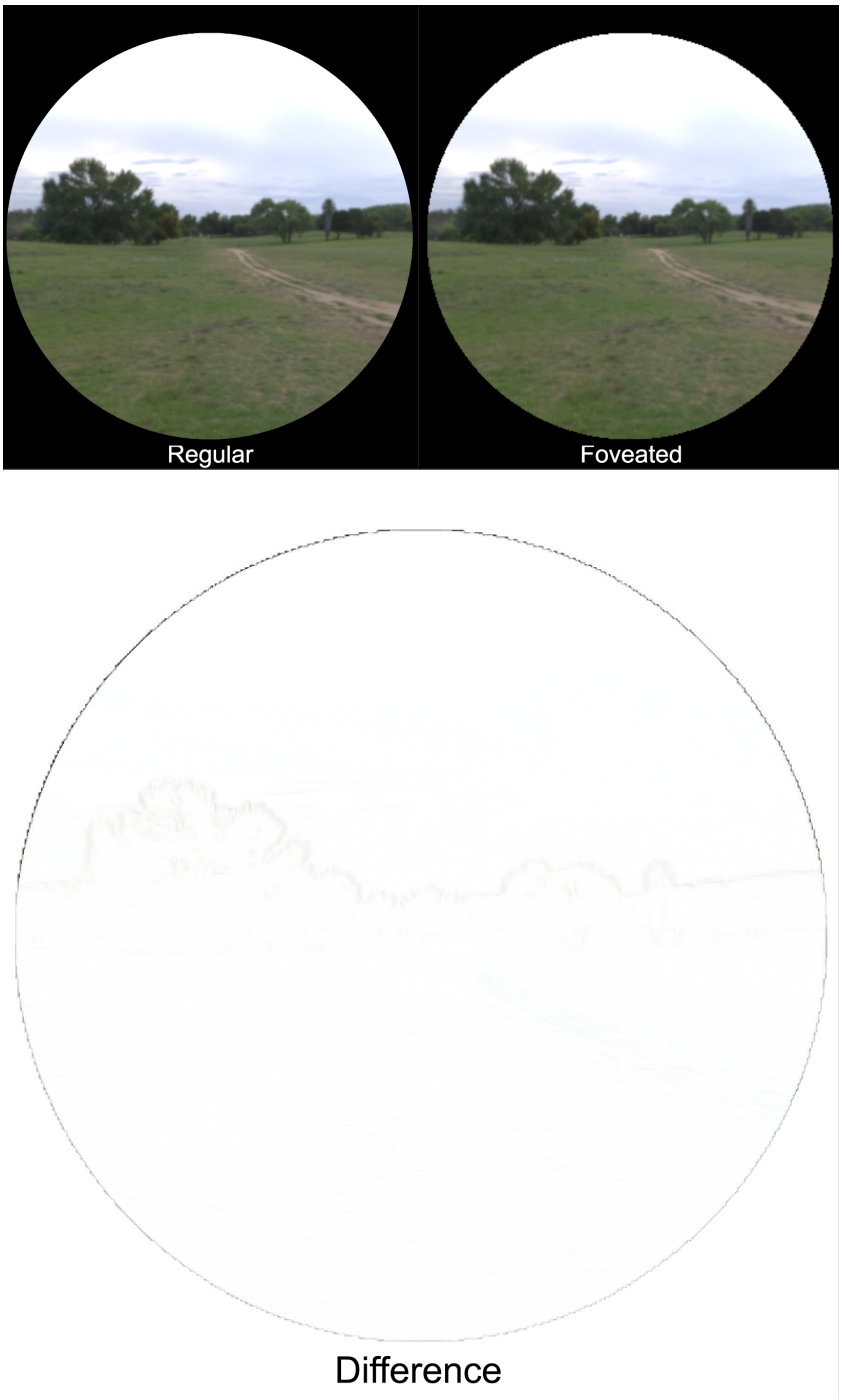

Figure A.1: Differences in the control scene. Top left is regular rendering, top right is forward gaze point foveated rendering, bottom is the differences between the two.

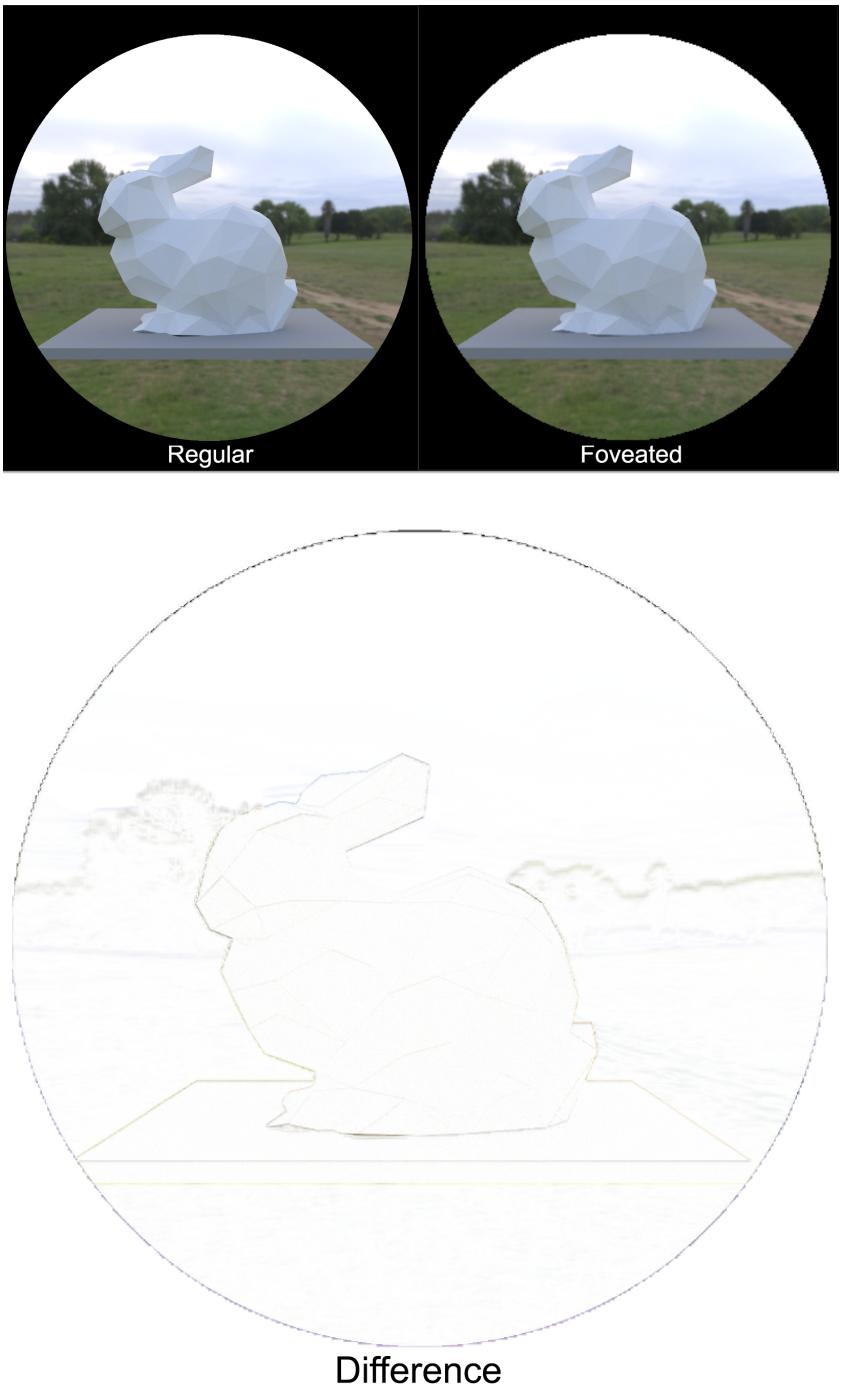

Figure A.2: Differences in the Triangle Intersection Low scene. Top left is regular rendering, top right is forward gaze point foveated rendering, bottom is the differences between the two.

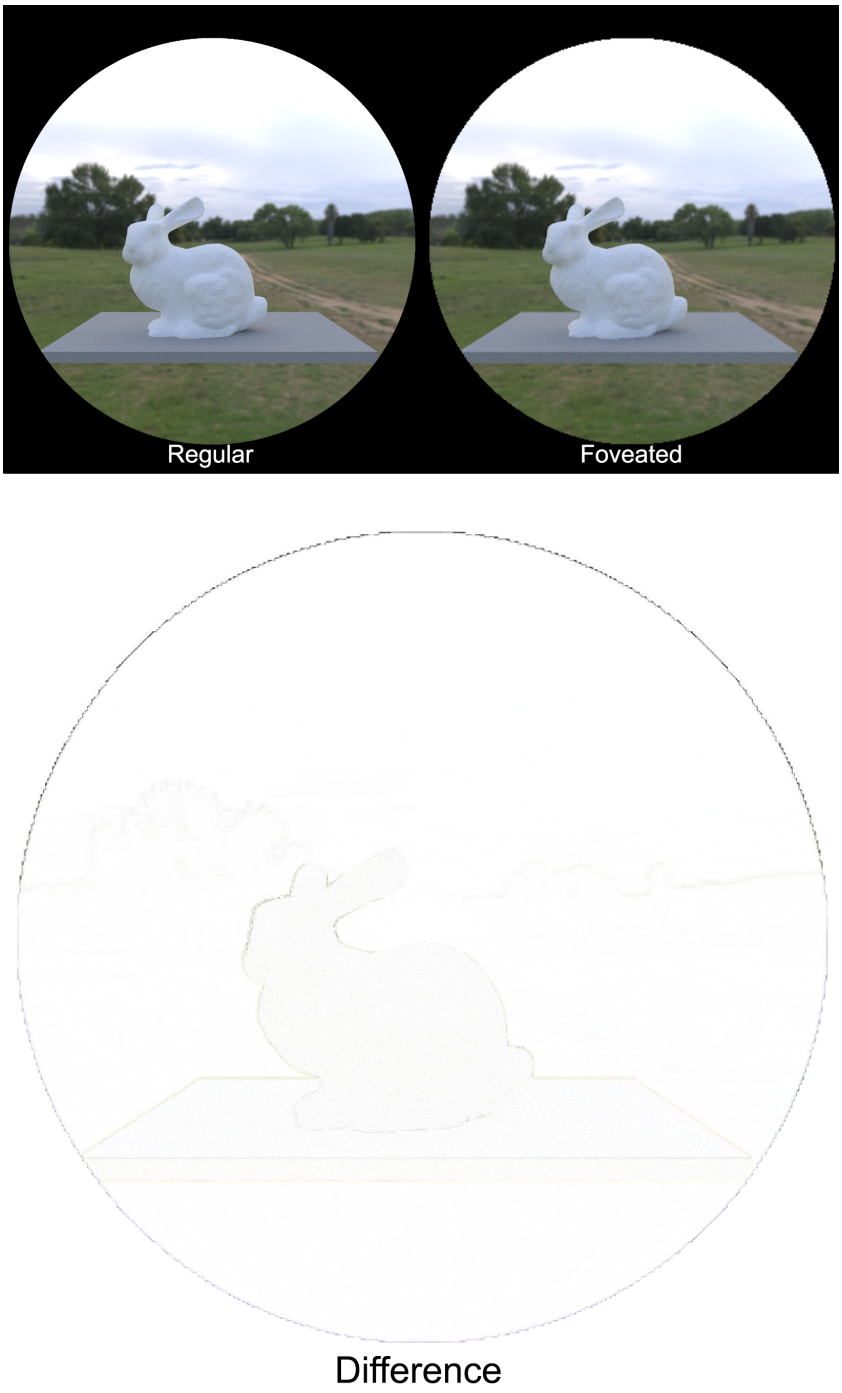

Figure A.3: Differences in the Triangle Intersection High scene. Top left is regular rendering, top right is forward gaze point foveated rendering, bottom is the differences between the two.

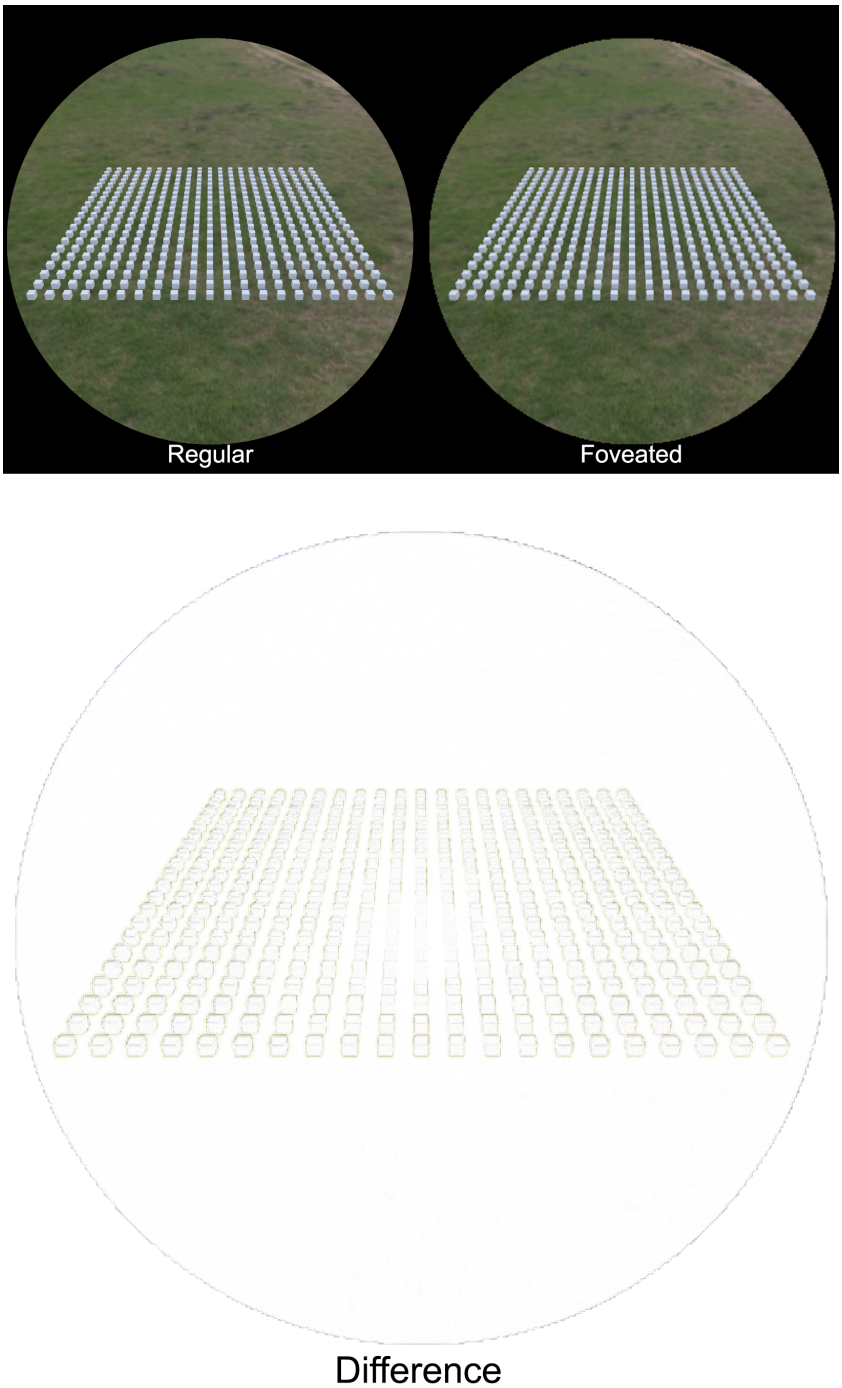

Figure A.4: Differences in the Multi-Object scene. Top left is regular rendering, top right is forward gaze point foveated rendering, bottom is the differences between the two.

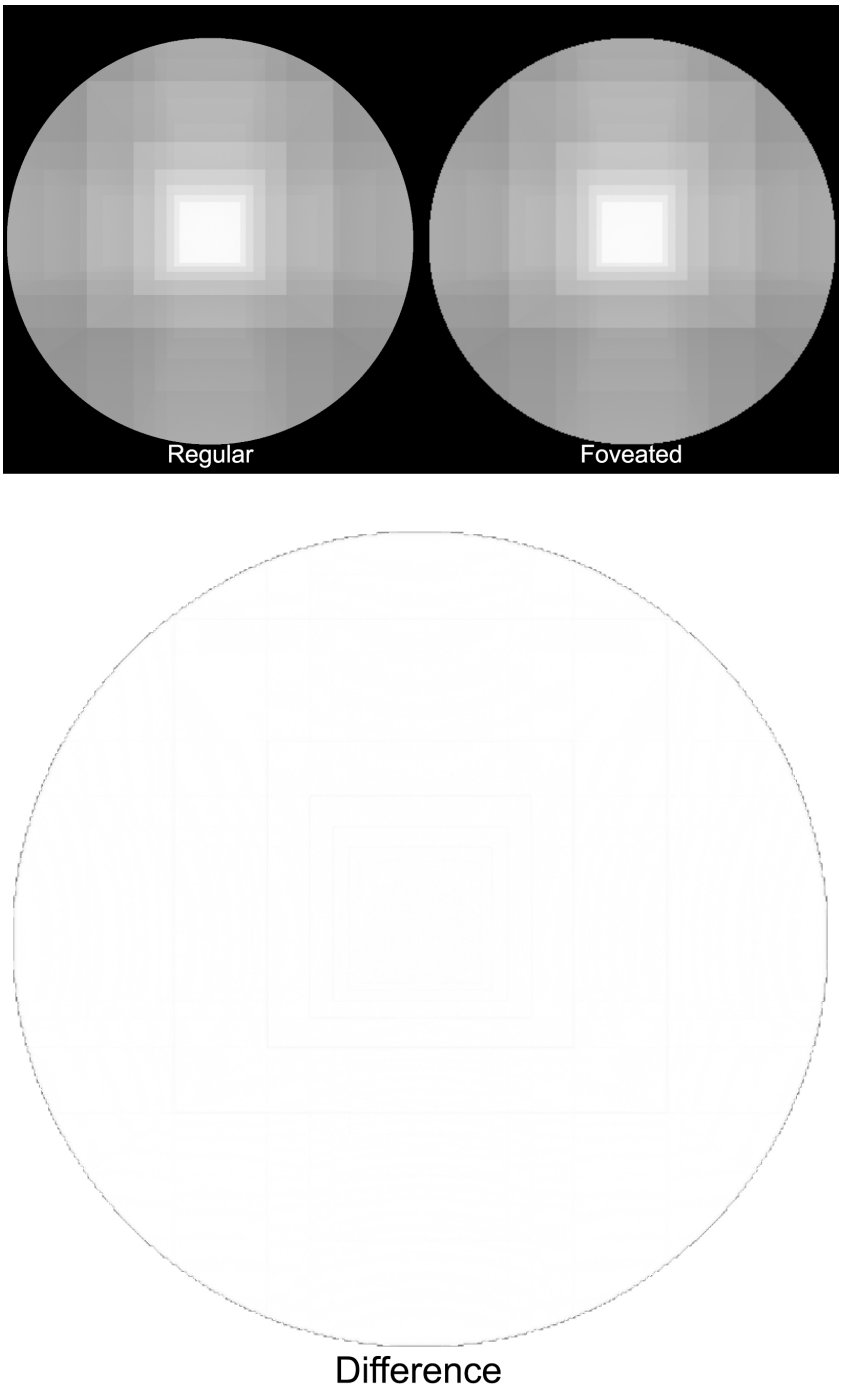

Figure A.5: Differences in the Reflection scene. Top left is regular rendering, top right is forward gaze point foveated rendering, bottom is the differences between the two.

Appendix B

Rendered Figures

This appendix includes some extra figures of the rendering system. These are not of importance for the thesis, but instead showcase the visual results possible from the rendering system.

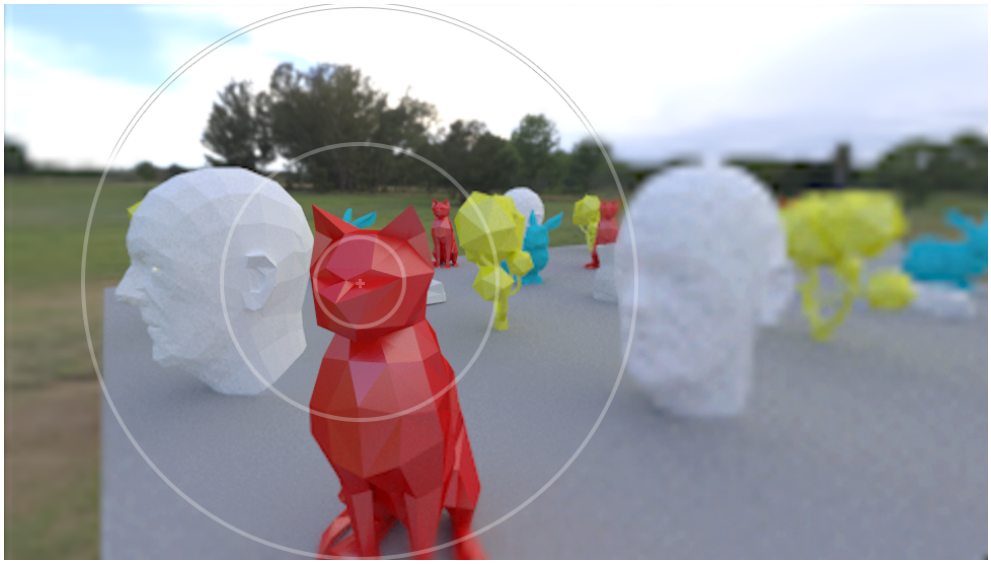

Figure B.1: The image used for the front page of this thesis.

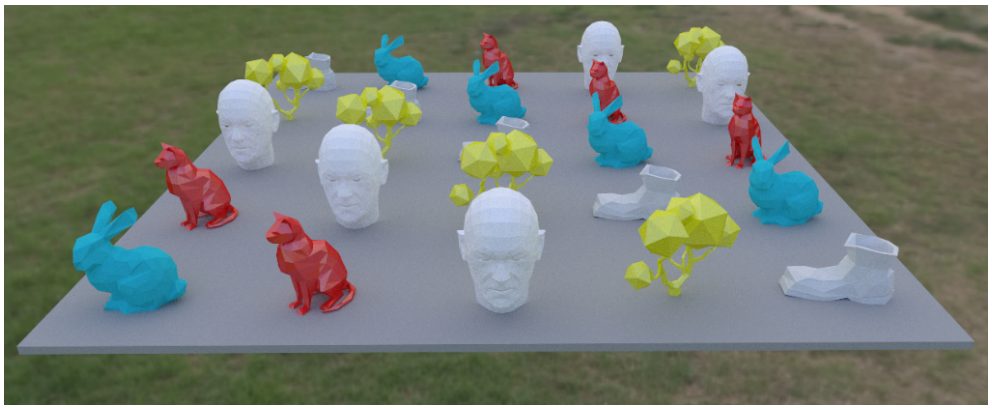

Figure B.2: A scene that originally was going to be used for the evaluation, in the end it was replaced with the multi box scene.

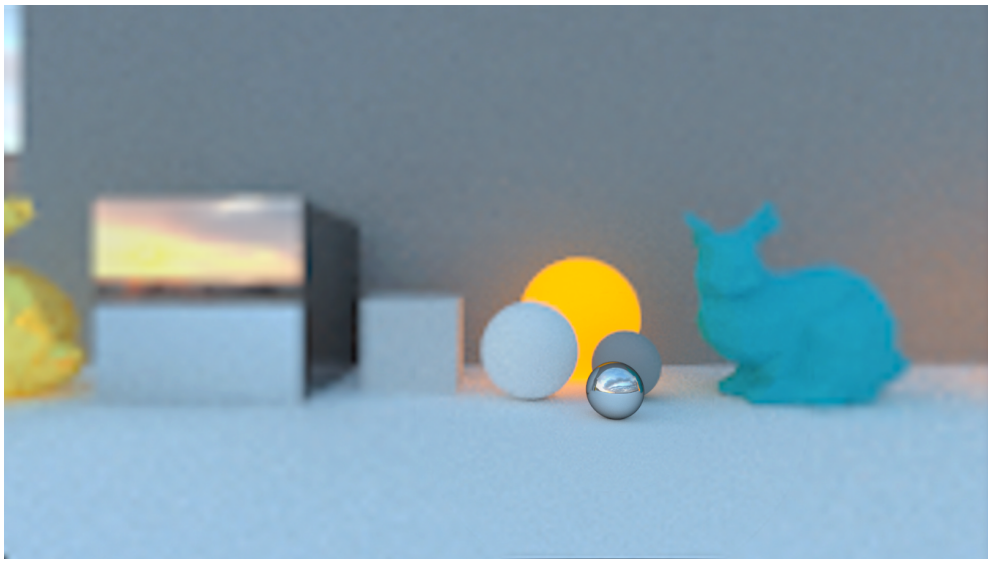

Figure B.3: A rendering where the foveated effect is particularly visible. The gaze point is located directly on the center of the small reflective sphere on the ground.

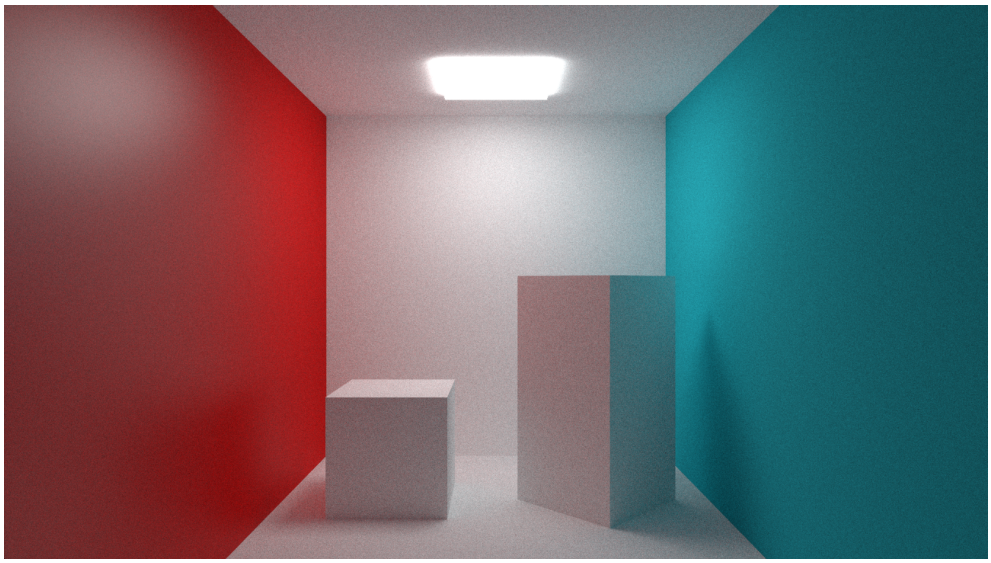

Figure B.4: A rendering similar to the cornell box test. This showcases the achieved global illumination when using path tracing.