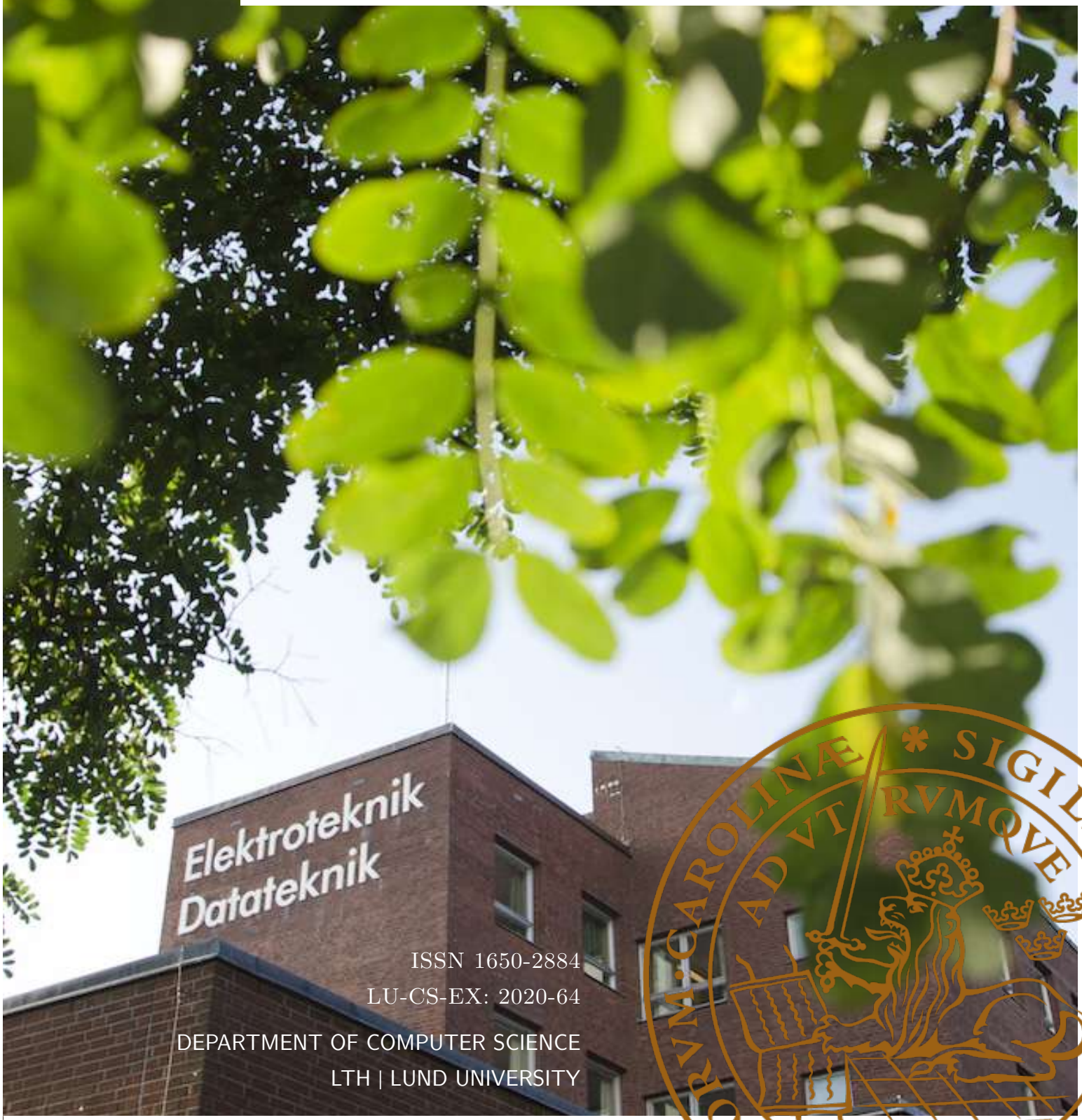
MASTER'S THESIS 2020

Performance Optimizations for Foveated Real-Time Raytracing

Kalle Andersson, Tom Hansson



ISSN 1650-2884
LU-CS-EX: 2020-64

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2020-64

Performance Optimizations for Foveated Real-Time Raytracing

Prestandaoptimeringar för fovea-styrd strålspårning i realtid

Kalle Andersson, Tom Hansson

Performance Optimizations for Foveated Real-Time Raytracing

Kalle Andersson
dat15kan@student.lu.se

Tom Hansson
dat13tha@student.lu.se

November 16, 2020

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisor: Michael Doggett, michael.doggett@cs.lth.se

Examiner: Flavius Gruian, flavius.gruian@cs.lth.se

Abstract

Advancements in ray tracing hardware and techniques make ray tracing a feasible option when rendering real-time computer graphics. However, ray tracing is still a computationally demanding task, which is why methods that reduce the computational load is important. Foveation is a technique that reduces the number of computations by lowering the image quality in the periphery of a user's vision, where the perceived level of detail is lower.

This thesis tests different performance optimizations for foveated rendering while maintaining sufficient image quality. We have developed a foveated renderer using log-polar transformation. By using different optimizations, a speedup by a factor of two is reached.

One of the major optimizations we tested is frustum tracing but we realized after developing different prototypes that it was not a feasible method to achieve performance gain in our implementation.

Keywords: ray tracing, foveation, log-polar, temporal anti-aliasing, frustum tracing

Acknowledgements

We would like to thank our supervisor Michael Doggett for supporting us and helping us come up with ideas and solutions to different problems. We also want to thank him for lending us graphics cards with ray tracing capabilities. We want to thank Pierre Moreau for setting up and testing the frustum tracer on an RTX graphics card. We also want to thank our parents for supporting us.

Contents

1 Introduction **7**
- 1.1 Project Definition . . . 7
- 1.2 Research Questions . . . 8
- 1.3 Ethical Issues . . . 8
- 1.4 Contributions . . . 8
- 1.5 Division of Work and Writing . . . 8
- 1.6 Outline . . . 8

2 Background and Theory **11**
- 2.1 Ray Tracing with DirectX . . . 11
- 2.2 Foveation . . . 12
- 2.3 Temporal Anti-Aliasing . . . 13
- 2.4 Error Metrics . . . 14
- 2.5 Frustum tracing . . . 14
  - 2.5.1 Acceleration structure . . . 15
- 2.6 Related work . . . 16

3 Approach **17**
- 3.1 Basic Ray Tracer . . . 18
- 3.2 Ground Truth Ray Tracer . . . 18
- 3.3 Foveated Ray Tracer . . . 18
  - 3.3.1 Log-polar Foveation . . . 18
  - 3.3.2 Kernel Foveation . . . 19
  - 3.3.3 Composite Foveation . . . 20
  - 3.3.4 Log-Polar Aspect Ratio . . . 21
  - 3.3.5 Log-Polar Transform Formulas . . . 23
- 3.4 TAA . . . 24
  - 3.4.1 Depth Based Rejection . . . 25
  - 3.4.2 Neighborhood Clamping . . . 25
- 3.5 Gaussian Filter . . . 26

3.6 Error Metrics . . . 26
3.7 Frustum Tracing . . . 27
3.7.1 Resizing the AABB . . . 27
3.7.2 Ray-Box Intersection . . . 28
3.8 Acceleration Structure . . . 28

4 Implementation **31**
4.1 Rendering Loop . . . 31
4.2 Foveation Rendering Loop . . . 32
4.2.1 TAA . . . 32
4.2.2 Composite Foveation . . . 33
4.3 Error Metrics . . . 33
4.3.1 MSE Shader . . . 33
4.3.2 SSIM Shader . . . 34
4.3.3 Ranges . . . 34
4.4 Modeling the Acceleration Structure . . . 34
4.5 Frustum Tracing . . . 38
4.5.1 TLAS Tracer . . . 39
4.5.2 BTLAS Tracer . . . 40
4.5.3 BBLAS Tracer . . . 41

5 Results and Discussion **43**
5.1 Input Parameters . . . 43
5.1.1 The Tuning Process . . . 43
5.1.2 Modes . . . 44
5.2 Matching Radial and Angular Resolution . . . 45
5.2.1 Tuning . . . 46
5.3 Running Our Foveated Ray Tracer . . . 47
5.3.1 Performance . . . 47
5.3.2 Ghosting and Flickering . . . 49
5.3.3 Error Heatmap . . . 50
5.3.4 Error Ranges . . . 51
5.3.5 Trade-off Between Image Quality and Performance . . . 56
5.4 Running Our Frustum Tracer . . . 58

6 Conclusion **61**
6.1 Threats to Validity . . . 63
6.2 Future Work . . . 63
6.2.1 Head-Mounted Display . . . 63
6.2.2 User Study . . . 63
6.2.3 Animated Scenes . . . 64

References **65**

Chapter 1
Introduction

Ray tracing is a rendering method well suited for realistic-looking imagery. However, ray tracing in real-time applications has been avoided due to the number of computations that are required. Over the years, as hardware has become more powerful, there has been a push for using ray tracing in real-time computer graphics. One example is Nvidia releasing the RTX graphics cards with built-in support for ray tracing.

Even though new support for ray tracing is released, the computational load is still large. Because of this, techniques that reduce the number of computations are useful for improving performance. Foveation is one such method, and we investigate foveation with different optimizations.

1.1 Project Definition

Foveation is a method that can reduce the number of computations required during ray tracing. We explore how different optimizations can be used to improve the performance of foveated ray tracing. The first optimization is frustum tracing with the purpose of accelerating the ray collision search [19]. The second optimization is a buffer containing the pixels close to the point of gaze to prevent oversampling. The third and last major optimization is using the kernel function introduced by Meng et al. [21].

While increasing the performance using optimizations, we also want to preserve acceptable image quality. Our foveated ray tracer uses TAA (Temporal Anti-Aliasing) and ray launch offsets to combat aliasing and improve image quality. We evaluate visual quality using three different error evaluation metrics: RMSE (root-mean-square error), PSNR (peak signal-to-noise ratio), and SSIM (structural similarity index measure). We also include our subjective opinions when evaluating image quality.

To summarize : This project's purpose is to explore performance optimizations for foveated ray tracing while maintaining acceptable image quality.

1.2 Research Questions

Taking into consideration both the performance as well as image quality the following research questions were developed.

- How much can the performance of ray tracing be improved by using log-polar foveation?
- How much can the performance of foveated ray tracing be improved by using foveation with optimizations?
- How large is the difference in image quality between foveated ray tracing and a ground truth image?

1.3 Ethical Issues

We have no specific ethical issues to mention in relation to this thesis.

1.4 Contributions

Our contributions include: developing and evaluating a frustum tracer in DXR (DirectX Raytracing) [6], how we implement composite foveation by combining the log-polar foveation with a separately rendered image at the point of gaze, the error evaluation method evaluating the foveated images in ranges, and the results of comparing and evaluating different foveation optimization settings against each other.

1.5 Division of Work and Writing

Kalle Andersson has been in charge of developing the foveated renderer, the TAA algorithm, and the image error evaluation system. Tom Hansson has been in charge of developing the frustum tracer, modeling the acceleration structure, and developing the composite foveated renderer. We have also had discussions about the different parts and helped each other when needed. Some parts such as the basic and ground truth ray tracers were developed entirely together.

The writing has been done mostly together, especially chapters 1, 5, and 6. In chapters 2, 3, and 4, the parts we were in charge of developing were generally also the sections we wrote about.

1.6 Outline

The report is divided into the following chapters: Introduction, Background and Theory, Approach, Implementation, Results, and Conclusion. A short description of each chapter is provided here. In chapter 1 (Introduction), we present the purpose of our project as well as limitations and contributions. In chapter 2 (Background and Theory), we introduce concepts

linked to the project and also discuss previous work in the area. In chapter 3 (Approach), we provide a detailed description of what we have done in the project. Chapter 4 (Implementation) contains information about how our program was implemented. Chapter 5 (Results and Discussion) consists of the different data obtained in tests when running the application, as well as analysis and discussion of the results. In chapter 6 (Conclusion), we summarize the project and answer the research questions presented in section 1.2.

Chapter 2
Background and Theory

In this chapter, relevant background and theory are presented. The theory covers the base subjects that are closely linked with the project. A basic understanding of the concepts presented in this chapter is useful when reading this report. We start by discussing ray tracing and how it works in DirectX (section 2.1), which is important since we use terms introduced in that section later in the report. The following section 2.2 describes what foveation is and gives an introduction to how it works. Foveation is the main concept of this report. The next section 2.3 explains temporal anti-aliasing which is a method used in our project to reduce aliasing. Section 2.4 describes the different error metrics we use to evaluate the image quality. Section 2.5 explains the concept of frustum tracing as well as the data structures it uses. The last section of this chapter (section 2.6) covers previous works and how they relate to this thesis.

2.1 Ray Tracing with DirectX

Ray tracing is a rendering method that works by simulating firing a ray in 3D space and calculating what geometry it intersects. In our project, we use DirectX [6] with an extension for ray tracing, called DXR. In this section, we cover some aspects of DirectX and DXR that we mention later in the report.

DirectX makes use of shaders, which are pieces of code that run on the GPU (Graphics Processing Unit). There are several different types of shaders used in our implementation and we describe two of them in this section. The first shader type is called ray generation shader and is part of DXR. Ray generation shaders are used to set up the origin and direction of the rays. They also begin the ray tracing process, which results in a color output. The color output obtained from the ray tracing process can be displayed on the screen to present a view of the scene. The second shader type is called compute shader and is part of DirectX. A compute shader is a general-purpose shader used for various computation tasks on the GPU.

2.2 Foveation

Foveation is a rendering method used to increase performance at the cost of lowering image quality. The idea is to use eye-tracking to maintain image quality around the point of gaze and only reduce the image quality in the periphery. This technique is well suited for the case of using an HMD (Head Mounted Display) equipped with an eye tracker. This is because a large proportion of the screen is viewed in the user's periphery and can be reduced to low quality. Another advantage of using an HMD is that it guarantees that only one user is viewing the screen at a time. Having multiple users would create a conflict because their points of gaze could be at different locations of the screen at the same time. However, we evaluate the image quality on computer screens in this project. This is because we did not have access to HMDs.

The reduction of quality in the periphery should, in a best-case scenario, be done in a way that does not affect the user's perceived quality of the image. There are other reports that discuss the optimal ways of doing this, such as Patney et al. [2].

There are multiple different ways to create a foveated image. In this project we use log-polar transforms [21] to transform coordinates between screen space and log-polar space. In screen space, a pixel position is defined by a horizontal distance (usually denoted as x) and a vertical distance (usually denoted as y). In log-polar space, the position of a pixel is instead defined by the logarithmic distance from the origin (usually denoted as u) and an angle (usually denoted as v). The log-polar coordinate system has a radial and an angular axis.

Figure 2.1: The left image is ray traced using one ray per pixel. The right image shows the same view in log-polar space.

The log-polar space transformed image (the right image in figure 2.1) is stored in a buffer. The size of the buffer is usually smaller than the screen space image, which is not the case in figure 2.1. The buffer size is independent of the size of the screen space image. Decreasing the size of the buffer lowers the quality of the foveated image, but increases performance.

Figure 2.2 shows two images of the same view. The left image is in screen space and the right image is in log-polar space. Color-coded highlights of certain areas have been added to both images. Areas with the same color in the two images correspond to the same parts of the scene view. The purpose of the highlights is to give a better understanding of how the two spaces correspond. In the screen space image, the red area corresponds to an area around the point of gaze which is in the middle of the screen. The same part of the view is colored red in the log-polar image, which shows that the region around the point of gaze corresponds to the left side of the log-polar image. The green area shows that the right side of the log-polar

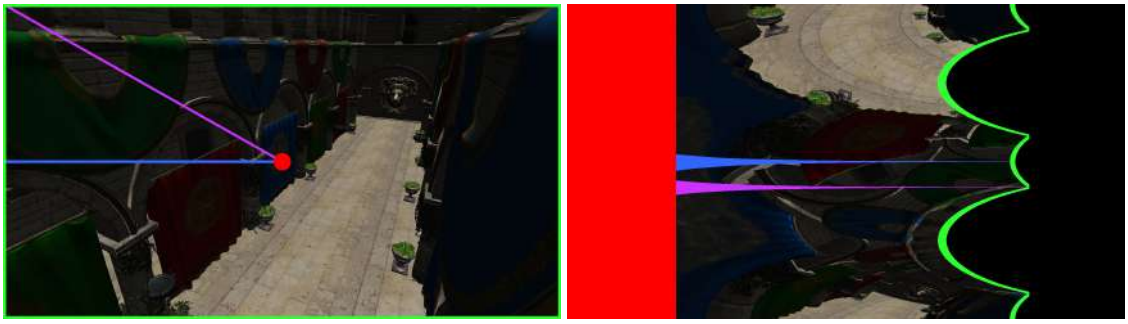


Figure 2.2: The colored parts of the images correspond to the same area of the scene in screen space (left image), and log-polar space (right image).

image corresponds to the edges of the image in screen space. The blue and purple lines have a constant thickness in the screen space image, but appear tapered in the log-polar image. Their varying thickness shows that parts of the scene that are near the point of gaze take up a larger area in the log-polar image. Hence, when transforming back to screen space from log-polar space the log-polar transform ensures that a region around the point of gaze has a higher sample count, and by extension quality, than the edges as illustrated by figure 2.3.

Figure 2.3: An example foveated image of the view in figure 2.1.

2.3 Temporal Anti-Aliasing

Temporal anti-aliasing, or TAA, is an anti-aliasing method that we use in our project to combat visual artifacts such as flickering. TAA uses data from previous frames to improve the current frame's image quality. This can be achieved by having a history buffer containing

the previous output image. The data in the previous image is weighted together with the data from the current frame to produce the final image. TAA allows pixels to get more color samples over time, which improves the quality. It also prevents sudden changes in the image, which mitigates flickering.

When moving and/or rotating the camera, pixels in subsequent frames will not contain data corresponding to the same position in world space. This means that the color data of a pixel in the history buffer image will not correspond to the same pixel position in the current frame. This is solved by using re-projection, which projects the position of the current pixel into the previous frame to find the correct corresponding data. There are also cases when new geometry, previously hidden behind other geometry, is revealed by the moving camera. In these cases, the history will contain the wrong data which causes a phenomenon known as ghosting. Ghosting is when previous frame data lingers in cases where it should not. To solve this issue the history needs to be restricted. There are multiple ways of restricting history such as using a depth buffer [10]. If the current world space depth is different from the stored previous depth, previously occluded geometry is hit which causes the history to be rejected. Neighborhood clamping [4] is another method used to restrict the history by clamping it to the neighboring pixels in the current frame.

2.4 Error Metrics

We use three different error metrics to evaluate image quality: RMSE, PSNR, and SSIM. We decided to use more than one metric to gain a more robust result. By using multiple metrics, one metric could showcase results that other metrics might have missed. We decided to use the three aforementioned metrics since they are widely known and used metrics in computer graphics.

RMSE measures the root of the sum of all squared differences in color space for each pixel. PSNR is used to measure the noise of the image in decibel. The SSIM comparison function is composited of three different functions each comparing different aspects of the image quality. The three composite functions are luminance, contrast, and structure. SSIM generates an index quantifying the similarity of the two compared images.

2.5 Frustum tracing

We use a technique called frustum tracing in an attempt to speed up rendering. Frustum tracing is a technique that decreases the rendering time by culling parts of the scene. This is done by dividing the screen into tiles and calculating what parts of the scene intersect a frustum behind each tile. A tile and the corresponding frustum are visualized in figure 2.4. Any geometry that does not intersect the frustum is culled from the scene when rendering the corresponding tile. In terms of ray tracing, culling parts of the scene reduces the search space during intersection testing. Intersection testing is a step of ray tracing that consists of calculating what part of the scene a ray intersects.

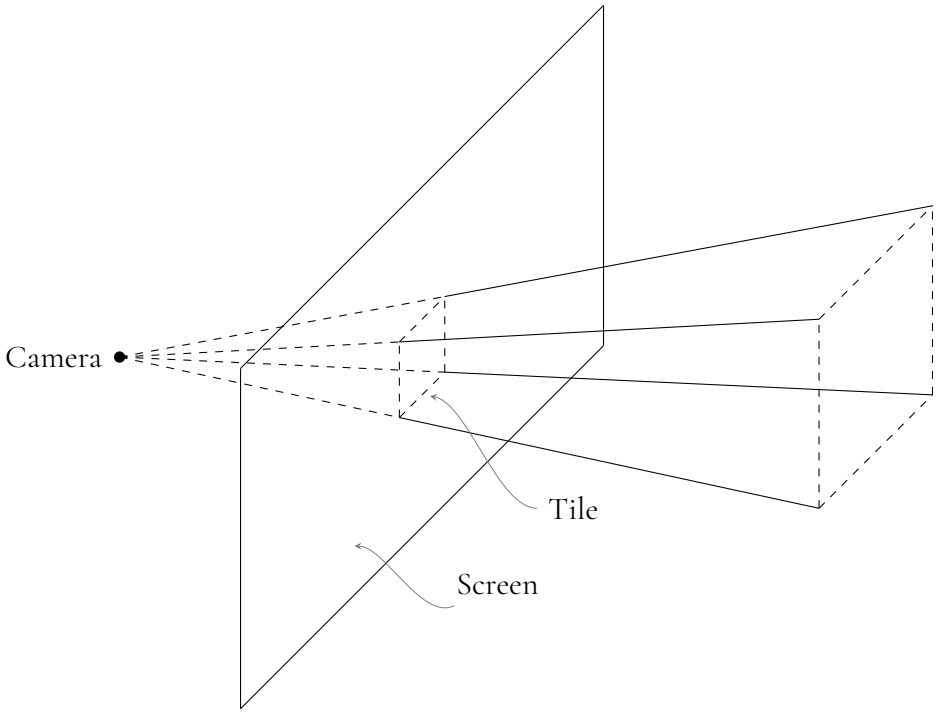


Figure 2.4: A frustum in 3D space behind a tile on the screen.

2.5.1 Acceleration structure

Performing frustum tracing involves modifying the scene data, or its usage, in some way to achieve the culling. In DXR scene data is stored using a format called acceleration structure.

An acceleration structure is divided into two parts, a bottom-level and a top-level. A BLAS (bottom-level acceleration structure) contains primitives, which are simple shapes such as triangles or spheres. Often, a BLAS makes up a single 3D model such as a character or a building, but it is possible for the programmer to merge or split 3D models into one or several BLAS respectively. A TLAS (top-level acceleration structure) contains instances. An instance is a BLAS placed somewhere in the scene. A TLAS makes up a scene.

Bounding Volume Hierarchy

The acceleration structure utilizes a BVH (bounding volume hierarchy) [3] to achieve fast intersection testing. The BVH is the part of interest of the acceleration structure for performing frustum tracing. A BVH is a tree-shaped data structure containing geometric objects. Our frustum tracer uses the BVH of an acceleration structure to perform culling.

The geometric objects are leaves in the tree structure and are grouped together in nodes. A bounding volume is fit to each node so that it completely encloses the objects within. These nodes are further grouped together in larger nodes, whose bounding volume covers those of its child nodes. This structure is repeated until the root node is reached, forming the tree structure. Using a BVH, the intersection testing is carried out by calculating if a ray intersects a node's bounding volume. If it does not intersect, no further tests have to be carried out further down that path of the tree. If it does intersect, however, the process is repeated for each of the node's children. This is repeated until the leaves are reached, at which

point ray-triangle intersection tests is performed for the remaining triangles.

2.6 Related work

In the article *Kernel Foveated Rendering*, by Meng et al. [21], an expansion of the normal log-polar transform is suggested. They propose adding a kernel function to control how many rays are close to the point of gaze compared to the periphery. Hence, being able to increase the detail in the periphery without losing a significant amount of detail near the point of gaze. Our foveated renderer is based on the kernel foveated renderer proposed by Meng et al.

In the article *Towards Foveated Rendering for Gaze-Tracked Virtual Reality*, by Patney et al. [2], they use contrast enhancement to improve the perceived quality in the periphery. They also try to mimic the quality falloff towards the periphery to the same falloff as the human eye. While our foveated renderer is not based on that work, their report served as a second source of information and ideas about the concept of foveation.

In the article *Foveated 3D Graphics*, by Guenter et al. [5], the rendering is done in three different qualities that are layered together. The highest quality layer is only rendered close to the point of gaze. We took inspiration from this work to develop a composite foveated renderer that combines a high-quality layer and a log-polar layer.

In the article *Hierarchical Visibility for Virtual Reality*, by Hunt et al. [19], a raycaster is implemented from the ground up. Their raycaster uses frustum tracing to accelerate the object collision search for rays. This article served as a base when implementing our frustum tracer with DXR.

Karis [4] suggests several different approaches for improving TAA. The Halton sequence is suggested as a method to ensure that samples are well distributed. Neighborhood clamping is suggested as a history restriction method to eliminate ghosting. We use the Halton sequence with a configuration based on Karis' recommendations. We also tried using neighborhood clamping as an alternative to depth-based rejection.

In the article *A Survey of Temporal Antialiasing Techniques*, by Yang et al. [10], different TAA methods are presented and described. Our TAA is optimized using ideas and inspiration from this article.

Chapter 3

Approach

In this chapter, a detailed description of the project approach is provided. We start by describing three different ray tracers we have implemented: the basic ray tracer, the ground truth ray tracer, and the foveated ray tracer. The ray tracers described in this thesis do not trace any secondary rays. The ray tracers' functionality is therefore more akin to a raycaster. However, we will refer to them as ray tracers in this thesis.

The basic ray tracer, section 3.1, is used as a baseline to compare the foveated ray tracer's performance against. The ground truth ray tracer (section 3.2) implementation is based on the basic ray tracer but uses super-sampling to create a high-quality image. As the name suggests, it is used as the ground truth for evaluating the image quality of the foveated ray tracer.

The foveated ray tracer, section 3.3, renders a foveated image by utilizing log-polar transforms. We modify the foveated ray tracer using different optimizations described in separate sections. There are two main optimizations we use that change the log-polar transform: kernel foveation and composite foveation. They are described in sections 3.3.2 and 3.3.3 respectively.

In section 3.4 we describe how our TAA algorithm works and also how we restrict the history by using a depth buffer. Section 3.5 describes a Gaussian filter that we apply to reduce visual artifacts. In section 3.6 we describe our process for calculating errors of foveated imagery.

The last two sections, 3.7 and 3.8, are not connected to foveation. They are instead connected to the third optimization we test, frustum tracing. The frustum tracer was implemented separately and was supposed to be integrated with the foveated ray tracer when it was finished. However, the frustum tracer achieved poor performance which is why we never integrated it with the foveated ray tracer.

3.1 Basic Ray Tracer

The first step of the project was to implement a basic ray tracer. This ray tracer was developed to serve as a base upon which all other ray tracers and functionality could be built. The basic ray tracer traces one ray in the middle of each pixel of the image. This basic ray tracer is used as a baseline for all performance comparisons in this thesis. The basic ray tracer is based on the example implementation provided by NVIDIA in the tutorial written by Lefrançois and Gautron [11].

3.2 Ground Truth Ray Tracer

The main purpose of the ground truth ray tracer is to create a baseline image for comparison of image quality. The ground truth ray tracer is a version of the basic ray tracer that uses super-sampling to create high-quality images. Super-sampling is achieved by tracing multiple rays in each pixel. We decided to super-sample with 16 rays per pixel, because increasing the number of rays further did not provide any significant difference in the resulting image in our tests. The rays in each pixel are offset from the center using the first 16 points of the 2,3 Halton sequence, as proposed by Karis [4]. The Halton sequence ensures that the samples are well distributed and not clumped together [12].

3.3 Foveated Ray Tracer

The foveated ray tracer is the main ray tracer of this project. It is this tracer we want to optimize and evaluate.

The foveated ray tracer is implemented in three steps. In the first step, we add the basic functionality of log-polar foveation, described in section 3.3.1. The second step adds the first optimization, kernel foveation (proposed by Meng et al. [21]), described in section 3.3.2. In the third step, we add the second optimization, that we call composite foveation, described in section 3.3.3. There is also a section 3.3.4 which covers the concept of changing the log-polar aspect ratio. Section 3.3.5 describes the log-polar transform formulas we use after integrating the optimizations.

We are not using eye-tracking to track the point of gaze in our application. Instead, we have implemented a movable point, called the focal point, that simulates where the point of gaze currently is. The reason for doing this is that we did not have access to eye-tracking equipment.

3.3.1 Log-polar Foveation

As stated in section 3.1, our basic ray tracer shoots one ray per pixel. We wanted to reduce the number of rays traced which in turn also reduces the number of shaded pixels. When performing foveated ray tracing, we transform the pixels' coordinates in the log-polar buffer from log-polar space into screen space Cartesian coordinates. The screen space coordinates obtained from the transform is where we shoot the rays. Hence, we shoot a number of rays that are proportional to the size of the log-polar buffer.

When the rays are traced, the color values obtained from the closest hit are stored in the log-polar buffer. However, some rays' screen space coordinates appear outside the screen after the transform, and we can detect these rays before the ray tracing process begins. We never trace these rays, because their data is not used when creating the screen space image. They are represented with black color in the log-polar buffer, this black area can be seen towards the right in the log-polar space image in figure 2.1.

After the rays are traced, a second pass of the data is needed to transform the data back from log-polar space to screen space. The second pass transforms all pixels in screen space to log-polar coordinates, which are used to access color data from the log-polar buffer.

We use the log-polar transform equations provided by Meng et al. [21] to calculate the rays' launch origin and direction, and to create the screen space image. The log-polar transform distributes the rays in a way that prioritizes a higher sampling rate near the point of gaze.

The log-polar foveation has one input parameter, σ, which controls the size of the log-polar buffer using equations (3.1).

$$\begin{aligned} w &= \frac{W}{\sigma} \\ h &= \frac{H}{\sigma} \end{aligned} \tag{3.1}$$

Where (W, H) represents the width and height of the application window, measured in pixels, and (w, h) represents the width and height of the log-polar buffer, also measured in pixels. A higher value of σ decreases the size of the log-polar buffer, which increases performance but lowers image quality.

A behavior of the log-polar transform is that a small area in Cartesian coordinates near origo maps to a larger area in log-polar space. In our case, this means that a pixel near the focal point can map to an area of the log-polar buffer several pixels in size. This problem is known as oversampling. The severity of the oversampling can be reduced by increasing σ. However, σ affects the quality of the entire image, which is a problem in the periphery where the sampling rate is already low.

3.3.2 Kernel Foveation

Because of the oversampling that occurs with the log-polar foveation, reducing the number of rays near the focal point would increase our performance without reducing image quality. Kernel foveation is an idea, proposed by Meng et al. [21], that shifts ray positions on the screen from the focal point toward the periphery. This is achieved by augmenting the log-polar transforms using a kernel function. The kernel function is the first optimization we introduce to our foveated ray tracer.

The definition of the kernel function K that we use is provided in equation 3.2.

$$K(x) = x^{\alpha} \tag{3.2}$$

This is the same kernel function that Meng et al. [21] used. The kernel function adds a second parameter to the foveated ray tracer, α. A higher α increases the effect of the kernel function, which means rays are shifted further towards the periphery.

3.3.3 Composite Foveation

The second optimization, composite foveation, is our own approach in attempting to solve the problem with oversampling of the log-polar foveation.

Guenter et al. [5], use an approach to foveation that differs from log-polar foveation. They render three layers of varying resolution, each containing a different region of the screen, and combine them into a foveated image. This inspired us to separately render the oversampled region, near the focal point, by removing it from the foveated image. In screen space, the removed region is circular and centered at the focal point, which is illustrated in figure 3.1. In log-polar space, the region corresponds to a rectangular area at the left side of the log-polar buffer. We remove the area by modifying the log-polar transforms in a way that shifts the area out from the log-polar buffer.

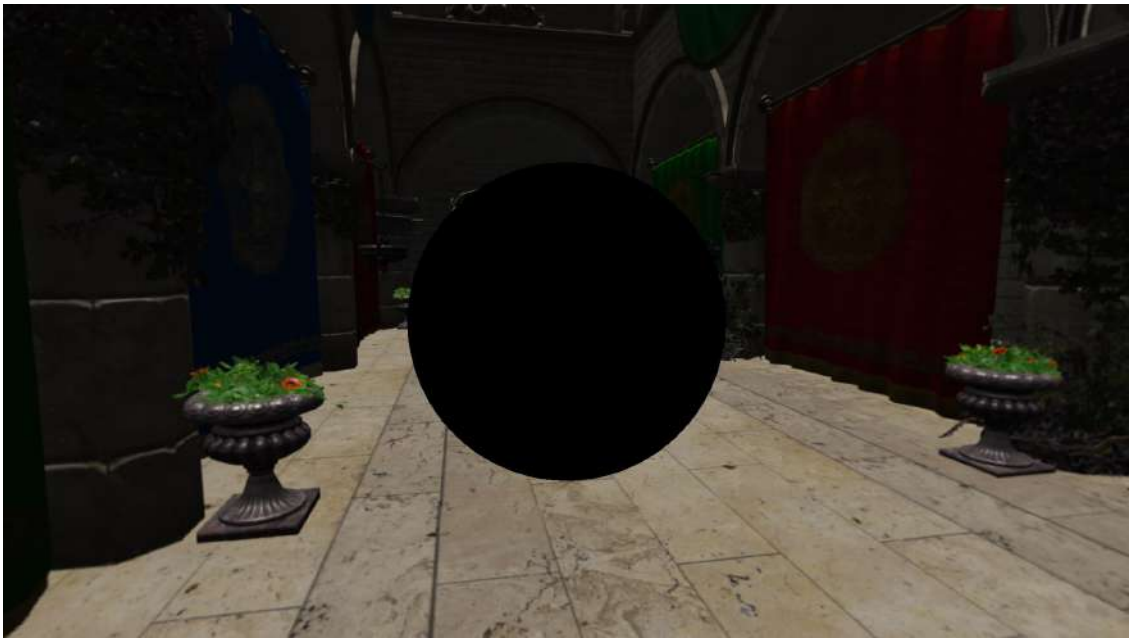


Figure 3.1: The oversampled region has been removed from this foveated image.

The removed region is rendered using one ray per pixel and stored in a buffer called the gaze buffer, illustrated by figure 3.2. We are going to refer to the resulting circular image, contained in the gaze buffer, as the gaze image.

Composite foveation introduces a third input parameter, a_G, to the foveated ray tracer. The parameter a_G is called gaze angle and is used to calculate the radius of the gaze image (r_G), as demonstrated in equation (3.3).

$$r_G = d \cdot \tan(a_G) \tag{3.3}$$

Where d is the distance between the user and the screen. The gaze image radius is used to calculate a radial coordinate, u_G, that maps to the edge of the gaze image. u_G depends on

Figure 3.2: A view of the gaze buffer.

r_G according to equation (3.4).

$$\begin{aligned} u_G &= \frac{w \cdot G}{1 - G} \\ G &= \left(\frac{\log(r_G + 1)}{L} \right)^{\alpha} \end{aligned} \tag{3.4}$$

Where L is the logarithm of the screen diagonal. L is described further in section 3.3.5. Any pixel in the log-polar buffer with a u-coordinate smaller than u_G is covered by the gaze image. We modify the log-polar transforms to remove these pixels from the log-polar buffer. This is done by adding u_G to both the u-coordinate and the log-polar buffer width, as shown in equation (3.5). Adding u_G to the width results in a horizontal stretching of the image in the log-polar buffer, which causes the right side of the image to be stretched out from the buffer. Then, adding u_G to the u-coordinate shifts the image in the log-polar buffer to the left, so that the left side is shifted out from the log-polar buffer, and the right side, which previously got stretched out from the buffer, gets shifted back in. The result is that all u-coordinates less than u_G are removed from the buffer.

$$\begin{aligned} u_C &= u + u_G \\ w_C &= w + u_G \end{aligned} \tag{3.5}$$

The variables u_C and w_C appear in our log-polar transform equations in chapter 3.3.5.

3.3.4 Log-Polar Aspect Ratio

We have a view that shows at which position each ray in the current frame was shot. This view was used during development to make sure that the ray distribution pattern behaved correctly. When examining this view we noticed that the distance between rays was different along the radial and angular axes. We suspected that distributing the rays evenly along both the radial and angular axes could improve the image quality. After testing this idea we confirmed visually that the perceived quality did improve and that the flickering in the periphery was reduced.

The ray density along the radial and angular axes is proportional to the width and height of the log-polar buffer respectively. If the shape of the log-polar buffer is changed, but its area is kept constant, ray density is shifted between the two axes. For example, by increasing the height of the log-polar buffer and decreasing the width, we get more samples along the

angular axis and fewer samples along the radial axis. The ratio between width and height is called aspect ratio.

We added a parameter controlling the aspect ratio of the log-polar buffer called (`lp_aspect`) and explored how it can be altered to improve the ray distribution. We can see in the left image of figure 3.3 that the rays are more tightly packed along the radial axis than the angular axis. By decreasing `lp_aspect` we obtain the evenly distributed ray pattern in the right image of figure 3.3. The shape of the log-polar buffers corresponding to figure 3.3 can be seen in figure 3.4. After introducing `lp_aspect` the definition of the width and height of the log-polar buffer (w, h) was redefined from equation (3.1) as follows in equation (3.6).

$$\begin{aligned} h &= round\left(\frac{\sqrt{\left(\frac{W \cdot H}{lp\_aspect}\right)}}{\sigma}\right) \\ w &= round(h \cdot lp\_aspect) \end{aligned} \tag{3.6}$$

Where (W, H) are defined as the width and height of the application window.

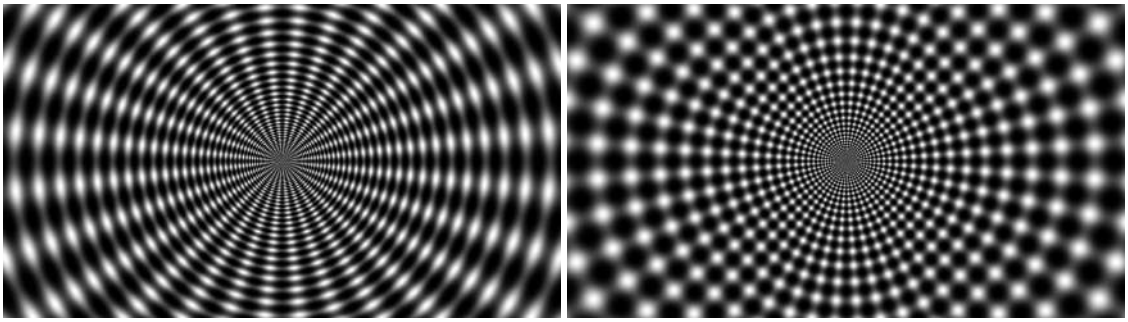


Figure 3.3: The white dots show where the rays were shot in the scene, one ray in the middle of each dot. $\sigma = 30$ (low amount of rays). In the left image `lp_aspect` = 3 and in the right image `lp_aspect` = 1.2.

Figure 3.4: The shape of the log-polar buffers corresponding to the two `lp_aspect` settings in figure 3.3.

3.3.5 Log-Polar Transform Formulas

As previously mentioned, the log-polar transform formulas that we use are based on those provided by Meng et al. [21], including the addition of a kernel function. We provide the log-polar transform equations in this section, explain the variables, and highlight our modifications.

Equations (3.7) convert log-polar coordinates to Cartesian coordinates.

$$\begin{aligned} x &= \exp\left(L \cdot K^{-1}\left(\frac{u_C}{w_C}\right)\right) \cdot \cos\left(\frac{2\pi}{h} \cdot v\right) + \mathring{x} \\ y &= \exp\left(L \cdot K^{-1}\left(\frac{u_C}{w_C}\right)\right) \cdot \sin\left(\frac{2\pi}{h} \cdot v\right) + \mathring{y} \end{aligned} \tag{3.7}$$

Equations (3.8) convert Cartesian coordinates to log-polar coordinates.

$$\begin{aligned} u &= K\left(\frac{log\|x',\, y'\|_2}{L}\right) \cdot w_C - u_G \\ v &= \left(\arctan\left(\frac{y'}{x'}\right) + 1\,[y' < 0] \cdot 2\pi\right) \cdot \frac{h}{2\pi} \\ 1\,[y' < 0] &= \begin{cases} 1, & y' < 0 \\ 0, & y' \geq 0 \end{cases} \end{aligned} \tag{3.8}$$

In equations (3.7) and (3.8), (u, v) are log-polar coordinates, (x, y) are Cartesian coordinates, and (x', y') is the distance to the focal point, as defined in equation (3.9).

$$\begin{aligned} x' &= x - \mathring{x} \\ y' &= y - \mathring{y} \end{aligned} \tag{3.9}$$

Where $(\mathring{x},\, \mathring{y})$ are the coordinates of the focal point. Continuing with equations (3.7) and (3.8), K is the kernel function from equation (3.2), (w, h) are the dimensions of the log-polar buffer. However, w is not present in the equations because of one of our modifications: All instances of u and w have been replaced with u_C and w_C from equation (3.5) respectively. In the first equation of (3.8), u_C has been substituted for $u + u_G$ in the left-hand side since we want to solve for u.

The variable L controls the maximum distance, from the focal point, that can fit into the log-polar buffer. L appears in the transform equations by Meng et al. [21] as the logarithm of the maximum distance from the focal point to one of the four corners of the screen. Using their definition, the value of L changes as the focal point moves across the screen. We noticed that the resolution of the log-polar foveation is affected by the value of L. We found this undesirable when using composite foveation since L then affects the resolution of everything but the gaze image. We want a smooth transition in resolution at the edge of the gaze image. Therefore, we change the definition of L to depend on the screen diagonal, as seen in equation (3.10), instead of the distance between the focal point and screen corners.

$$L = \log \sqrt{W \cdot W + H \cdot H} \tag{3.10}$$

A side effect of our redefinition of L is that the entire width of the log-polar buffer is not used unless the focal point is in a corner of the screen. We adjust for this by making the log-polar buffer wider, and not tracing any rays for the unused area.

Besides our change to $\boldsymbol{L}$ and the ones introduced by the composite foveation (u_C and w_C), we reiterate that we also changed the definition of the log-polar buffer's dimensions, (w, h), by introducing log-polar aspect ratio in section 3.3.4. This change also affects w_C, since it is dependent on w.

3.4 TAA

The TAA algorithm is applied during the second pass of the data, mentioned in section 3.3.1, after transforming from log-polar space to screen space. The TAA algorithm, as stated in section 2.3, uses data from previous frames to enhance the current frame. However, if the camera is stationary no new data will enter the image. This is because we use static scenes (without animations) when running our ray tracers. The result of this is that the history will converge towards the current frame and the image will become jagged in the periphery. To combat this jagged pattern we use a sampling pattern to offset the rays. By sampling subsequent frames at different offsets within the same pixel, more data will enter the history buffer. We are using the first sixteen samples of the 2,3 Halton sequence, recommended by Karis [4], to create our pattern. This improved the detail in the periphery considerably, see figure 3.5. However, the offsets are applied before the log-polar transform which means that the offsets are larger further from the focal point in screen space. The large offsets in the periphery create flickering, which is mitigated by the TAA but not removed completely. Another problem introduced by the pattern was a slight twist of the foveated image. When researching the cause of the twist we found that the average offset, along the x and y axes, of the 16 Halton points we used was close to 0.5 but not exactly. We calculate two scaling values `scale_x` and `scale_y` using the formula 3.11.

$$\begin{aligned} \texttt{scale\_x} &= \frac{0.5}{\texttt{offset\_average\_x}} \\ \texttt{scale\_y} &= \frac{0.5}{\texttt{offset\_average\_y}} \end{aligned} \tag{3.11}$$

Multiplying all Halton points with the scales causes the average of the offsets to be exactly 0.5. This solved the twist problem.

Figure 3.5: In the left image, there is no variation of the rays. The right image uses the Halton sampling pattern. Both images show the same section of the periphery from one specific view.

3.4.1 Depth Based Rejection

To restrict the TAA history during re-projection, we are using depth-based history rejection. When the camera is stationary, the rejection causes a problem when used together with the offset rays described in the previous section. The problem is that the rays can hit different geometry with different depths while still corresponding to the same pixel in the log-polar image. When geometry with different depth is found, the history is rejected, which causes unnecessary flickering. Our solution is to not reject any history when the camera is stationary. This works because there is no occluded geometry that becomes visible in subsequent frames when the camera is stationary (and the scene is static). The only other time history can be rejected is when the pattern offset causes the rejection, which we wanted to prevent.

Age

When the depth test fails and history is rejected, subsequent frames have less reliable data in the history buffer. This is because the buffer has not had enough time to accumulate new frame data. This is why we implemented an age factor as recommended by Yang et al. [10]. When the history is rejected the age is reset to 1. The age is then incremented by 1 during each subsequent frame until a new reset occurs. The output color is calculated using the formula (3.12).

$$output = history \cdot (1 - 1/age) + current \cdot (1/age) \tag{3.12}$$

As the age increases the history becomes more reliable which means that less of the current frame data is used. As recommended by Yang et al. [10] we have a limit for how large the age variable can be. Without a limit, the pixels on the screen will converge and consist of only history and no new data. We are limiting the age to a maximum of 20. This means that the maximum value of the history is 95%. We have a high limit to make as much use of the history as possible. When raising the limit above 20, the high reliance on the history causes ghosting to become noticeable.

Flickering

When the camera is moving, flickering caused by the history rejection becomes visible. The flickering is mostly a problem further away from the focal point where the quality of the image is worse. We had an idea that ghosting might be less noticeable than flickering in the periphery. This is why we disable history rejection when the distance to the focal point is larger than a selected fraction of the application window diagonal. This reduces the flickering in the periphery at the cost of introducing ghosting. Instead of using the rejection, we update the age variable to 1 which causes the ghosting to converge away faster. After testing different values of the fraction, and evaluating the image visually, we arrived at setting it to 1/3. The ghosting introduced is notable, which is why we did not want to apply this method close to the focal point, only far away in the periphery.

3.4.2 Neighborhood Clamping

Neighborhood clamping is a history rejection method recommended by Karis [4] which we tried using as an alternative to the depth-based rejection. We tried using neighborhood

clamping both in the final image and in the log-polar space buffer. The neighborhood clamping removes all ghosting but does not mitigate the flickering well in either case. It works better when used on the log-polar space buffer since the neighboring pixels are smaller compared to the squares in the periphery of the foveated image. However, even though it works better, the flickering is still prevalent compared to when using depth-based rejection. This is why we decided to use depth-based rejection over neighborhood clamping.

3.5 Gaussian Filter

We use a Gaussian filter which is applied during the second pass, mentioned in section 3.3.1. Meng et al. [21] use a Gaussian filter that is applied on the right half of the log-polar buffer (towards the edges of the screen space image). The purpose of that filter is to reduce noise and flickering in the periphery. We also use our Gaussian filter for the same purpose, but we apply it in a different way by interpolating between the 3x3 identity matrix and a 3x3 Gaussian kernel (3.13).

$$Identity\ kernel = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \ Gaussian\ kernel = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \tag{3.13}$$

The Gaussian filter is not applied near the focal point because we want to preserve sharp details there. Hence, we start the linear interpolation when the distance from the focal point is larger than 1/6 of the application window diagonal. We arrived at this value by testing different values and evaluating the image visually. The interpolation creates a smooth transition between the area close to the focal point, where no Gaussian filter is applied, and higher intensity further away from the focal point.

3.6 Error Metrics

In addition to visual evaluation, the error metrics (RMSE, PSNR, and SSIM) described in the previous chapter are used to compare images obtained from the foveated ray tracer to images produced by the ground truth ray tracer. There is an inherent problem when comparing the foveated image to the ground truth image directly. The problem is that the foveated image has higher quality at the point of gaze and lower quality in the periphery. Comparing the images this way, the error calculation would not be able to yield results reflecting how the quality changes at different parts of the image. Instead of calculating errors by comparing the images as a whole, we decided to calculate the errors in rings, at different viewing angles, around the point of gaze. The size of the circular area to be compared by the error metrics is selected by specifying a fraction of a variable called `max_view_angle`. The circular area is then split into a selected number of rings. The ring sizes are decided by splitting the viewing angle, within the circular area, in equal parts, and calculating the rings' inner and outer radius. A view of the rings is demonstrated in figure 3.6. Calculating errors in this way should yield results suited to evaluate a foveated image. We will be referring to this error calculation method as error ranges in the report.

Another idea on how to evaluate the errors could be to evaluate them directly in the log-polar buffer. However, we apply image quality preserving methods after the transformation from log-polar space to screen space. TAA and the Gaussian filter are both applied after the transformation and hence the log-polar image would not give an actual representation of the final quality.

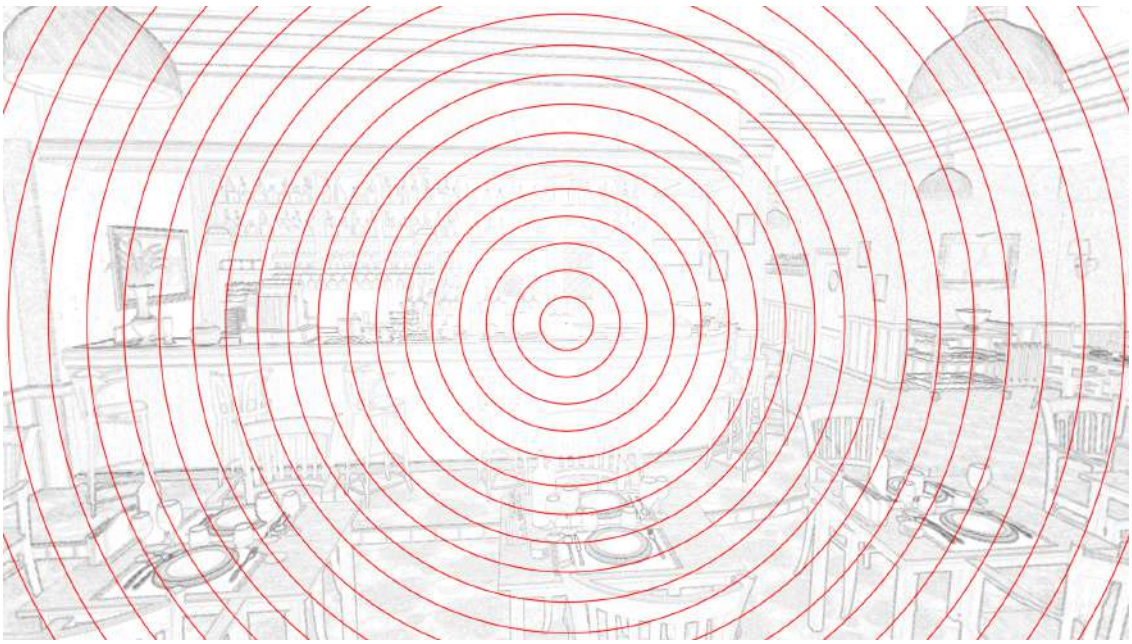


Figure 3.6: A view of 20 ranges covering the application window. The focal point is set in the center of the application window.

3.7 Frustum Tracing

Our approach to frustum tracing consists of two steps. In the first step, we make use of the fact that all bounding volumes in the acceleration structure's BVH are AABBs (axis aligned bounding boxes) to convert the problem at hand from frustum-box intersection to ray-box intersection. The second step is to perform the ray-box intersection test. The reason for this approach is to let us make use of an efficient ray-box intersection algorithm.

3.7.1 Resizing the AABB

The idea behind converting frustum-box intersection to ray-box intersection is to add the width and height of the frustum to the AABB, and then calculate whether or not the AABB intersects a ray along the center of the frustum. The width and height of the frustum vary with the distance from the camera. Rotating the camera, and thereby the frustum, changes the alignment between the frustum's width and height, and the AABB's sides.

We let a frustum be defined by the origin $\boldsymbol{O_F}$ and direction $\boldsymbol{D_F}$ of its frustum ray, which is a ray that originates from the camera and traverses through the center of the frustum's tile. Further, we specify the frustum's rotation around $\boldsymbol{D_F}$ by using the orientation of the

tile. These tiles' axes are aligned with those of the screen, which is specified by the camera's orientation. The camera's orientation is specified by three orthogonal vectors: $\boldsymbol{R}$, $\boldsymbol{U}$, and $\boldsymbol{V}$. The right vector $\boldsymbol{R}$ and the up vector $\boldsymbol{U}$ points along the x-axis and y-axis of the screen (and tiles) respectively. The view vector $\boldsymbol{V}$ points toward the camera's view direction, straight into the screen. Finally, a width factor w_F and height factor h_F are specified. The cross-sectional width w_C, and height h_C, of a frustum and a plane normal to $\boldsymbol{V}$, at a distance d from the camera, is calculated according to (3.14).

$$\begin{aligned} w_C &= d * w_F \\ h_C &= d * h_F \end{aligned} \tag{3.14}$$

Next, we want to calculate the bias to add to an AABB. The point on an AABB furthest from the camera is projected onto $\boldsymbol{V}$ to obtain the value of d. To account for frustum orientation, the width and height are scaled based on the orientation of the vectors $\boldsymbol{R}$ and $\boldsymbol{U}$, and then summed to determine the bias along each axis, as shown in equation (3.15), where x_B, y_B and z_B specify the bias along the x, y, and z axes respectively.

$$\begin{aligned} x_B &= |w_C \cdot \boldsymbol{R}.x| + |h_C \cdot \boldsymbol{U}.x| \\ y_B &= |w_C \cdot \boldsymbol{R}.y| + |h_C \cdot \boldsymbol{U}.y| \\ z_B &= |w_C \cdot \boldsymbol{R}.z| + |h_C \cdot \boldsymbol{U}.z| \end{aligned} \tag{3.15}$$

With the bias values added to the AABB size. Ray-box intersection with the frustum ray approximates frustum-box intersection for the unmodified AABB.

3.7.2 Ray-Box Intersection

While the intersection testing can be specified with an intersection shader in DXR, it only allows for specifying how the intersection between the ray and a primitive works, not how it intersects the bounding boxes of the acceleration structure. For this reason, we use a ray-box intersection software implementation for the second step of frustum tracing. In the article *An Efficient and Robust Ray–Box Intersection Algorithm* by Williams et al. [20] one such implementation is provided. We modify that code to add the bias to the box size. We provide pseudocode of the algorithm in listing 3.1, where `float3` specifies a vector with three components, `low` and `high` are the bottom-left and top-right corners of the AABB respectively, `bias` is the bias as calculated according to equation (3.15), `O` is the frustum ray origin $\boldsymbol{O}_F$, and `D` is the frustum ray direction $\boldsymbol{D}_F$. The function returns true if the frustum ray intersects the AABB specified by `low` and `high`, and with `bias` added to its size.

3.8 Acceleration Structure

To achieve frustum tracing we need to either modify the acceleration structure data or modify the way that we use the acceleration structure. We start by ruling out the second of these two alternatives because the only way to use an acceleration structure is by providing it to a `traceRay` call in a shader, thus we cannot modify the usage. There is a problem with the first alternative as well: The acceleration structure data format is undocumented [7]. Since we need to modify the data, we first create our own model of the format. This is done by

```
float3 iD = 1.0 / D; // computed once per frame

bool ray_box_intersection(float3 low, float3 high, float3 bias) {
    float t1 = (low.x - bias.x - O.x) * iD.x;
    float t2 = (high.x + bias.x - O.x) * iD.x;
    float t3 = (low.y - bias.y - O.y) * iD.y;
    float t4 = (high.y + bias.y - O.y) * iD.y;
    float t5 = (low.z - bias.z - O.z) * iD.z;
    float t6 = (high.z + bias.z - O.z) * iD.z;
    float tmin = max(max(min(t1, t2), min(t3, t4)), min(t5, t6));
    float tmax = min(min(max(t1, t2), max(t3, t4)), max(t5, t6));
    return tmax >= 0 && tmin <= tmax;
}
```

Listing 3.1: Ray-box intersection testing.

inspecting the acceleration structure data from different scenes. We compare the data, form hypotheses, and witness the effect of editing segments of the data in attempts to confirm or deny our hypotheses.

Chapter 4
Implementation

In this chapter, we provide details about our implementation. We explain what our rendering loop looks like in section 4.1. In section 4.2 we describe the implementation of the foveated ray tracer and how it affects the rendering loop. Section 4.3 details the implementation of the three error metrics we use. The last two sections are related to frustum tracing. In section 4.4 we describe our model of the acceleration structure that we use to implement our frustum tracer. In section 4.5 we cover our frustum tracer implementation.

Our code is written in the programming language C++. We use the API (Application Programming Interface) Direct3D 12, which is a part of DirectX [6]. The shader code is written in the programming language HLSL. Assimp (Open Asset Import Library) [14] is used to load 3D models into the application.

4.1 Rendering Loop

When we are rendering using the basic ray tracer or the ground truth ray tracer, rays are dispatched using a ray generation shader called `RayGen`. `RayGen` uses two input constant buffers to decide where the rays should be dispatched. The first buffer contains input parameters concerning the current rendering mode and the size of the application window. The second buffer contains transformation matrices corresponding to the current view. The parameters in these buffers decide where the rays in the current frame should be generated. The parameter containing information about the current rendering mode is used to decide if `RayGen` should behave as the basic ray tracer or the ground truth ray tracer.

After tracing the rays, the obtained data is stored in an output UAV (unordered access view) [18]. This UAV contains the output image that is displayed on the screen after the execution of `RayGen` is finished. A basic view of the dataflow when rendering using the basic ray tracer or the ground truth ray tracer can be seen in figure 4.1.

Figure 4.1: The dataflow when rendering using `Raygen`. The output image is either the ground truth image or the basic ray tracer's output image, depending on which of them is in use.

4.2 Foveation Rendering Loop

When we use the foveated ray tracer, `RayGen` is replaced by another ray generation shader called `LPRayGen`. `LPRayGen` has access to the same input constant buffers as `RayGen` but additionally has access to the four input parameters (σ, α, a_G, and `lp_aspect`), presented in section 3.3, controlling the log-polar transform. `LPRayGen` applies the log-polar transform to decide where the rays are generated. After tracing the rays, the obtained data is stored in a UAV. This UAV is the log-polar buffer containing the log-polar space image described in section 2.2. Additionally, when TAA is active, another UAV containing the depth in log-polar space is needed. After storing the data in the aforementioned UAVs the execution of `LPRayGen` is finished.

As mentioned in section 3.3.1, a second pass is needed to transform the data obtained from `LPRayGen` back from log-polar space. The second pass is executed by a compute shader called `Pass2`. Before `Pass2` is invoked, the log-polar buffer's data format is changed from UAV to SRV (shader resource view) [18]. This is done since we want to use a linear sampler to sample the color from the log-polar buffer which is not possible in the UAV format. `Pass2` uses the log-polar transform, controlled by the same input parameters as `LPRayGen`, to transform the screen space coordinates to log-polar space and sample color from the log-polar buffer. The obtained color is stored in an output UAV which, after the execution of `Pass2` is finished, is displayed on the screen. A basic view of the dataflow when rendering using the foveated ray tracer can be seen in figure 4.2.

4.2.1 TAA

The second pass also optionally applies TAA and the Gaussian filter (section 3.5) on the data. When TAA is enabled, `Pass2` uses additional output and input resources. The additional resources are an input SRV containing the accumulated TAA history from the previous frames and an output UAV where the current frame's data is stored to be used as history in the following frames. We also use a UAV containing the previous frame's depth as well as a UAV where we store the current frame's depth. The depth UAVs are used during the re-projection and depth-based rejection stages of the TAA. After the execution of `Pass2` is finished, the resulting image is stored in the output UAV which is displayed on the screen. The SRV and UAV containing the previous and current history swap place. This is done so the current history becomes the input previous history to the following frame. The same swap happens with the UAVs containing the previous and current depth.

4.2.2 Composite Foveation

When composite foveation is used, the shader `CompositeRayGen` is invoked at the same time as `LPRayGen`. The shader `CompositeRayGen` traces the rays corresponding to the relevant circular area in the gaze buffer, figure 3.2. The data obtained when tracing the rays is stored in a UAV, the gaze buffer. The depth is also stored in a UAV. These two UAVs are used as additional input to the shader `Pass2` (in addition to the output from `LpRayGen`). When `Pass2` transforms the data in the log-polar buffer back from log-polar space it also uses the gaze buffer to create the final image, which is then stored in the second pass' output UAV. The dataflow when rendering using the foveated ray tracer with composite foveation is showcased in figure 4.2.

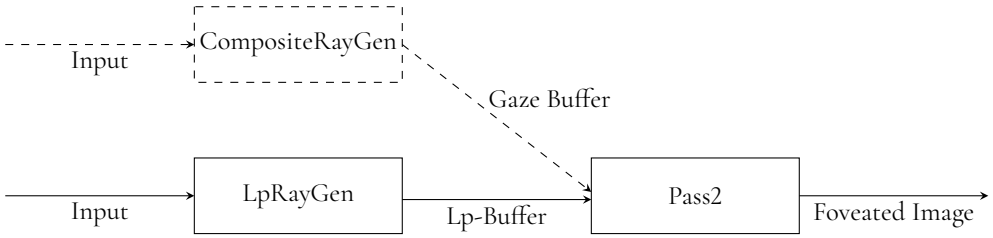


Figure 4.2: A basic view of the dataflow between shaders when rendering using the foveated ray tracer. The `CompositeRayGen` shader is dashed because its execution is optional depending on if composite foveation is used or not.

4.3 Error Metrics

The calculations of the three error metrics we use are implemented in two different compute shaders, called `MSE` and `SSIM`. When calculating errors, both the ground truth ray generation shader (`RayGen`) and the foveation shaders (`LPRayGen` and `CompositeRayGen`) are invoked. This is because we need access to both the foveated image and the ground truth image for the current frame when we do the error calculations. After the foveated image and the ground truth image are obtained, depending on the selected error type, one of the compute shaders `MSE` or `SSIM` are invoked to compare the ground truth image to the foveated image. The dataflow when calculating errors is illustrated in figure 4.3.

4.3.1 MSE Shader

The MSE shader calculates the mean square error in color space. The compute shader is dispatched with threads that each calculate the total error in 10x10 pixel areas. This is done to reduce the computation time in comparison to dispatching one thread for each pixel. The width and height values of the 10x10 area are selected to be ten since the width and height of computer screens usually are divisible by ten. The sum of errors obtained in each 10x10 area is stored in a buffer. The data in the buffer is used in the C++ code to calculate either the RMSE or PSNR. The individual error of each pixel is also stored to be viewed as a heatmap. The heatmap is displayed to evaluate and validate the distribution of the errors visually.

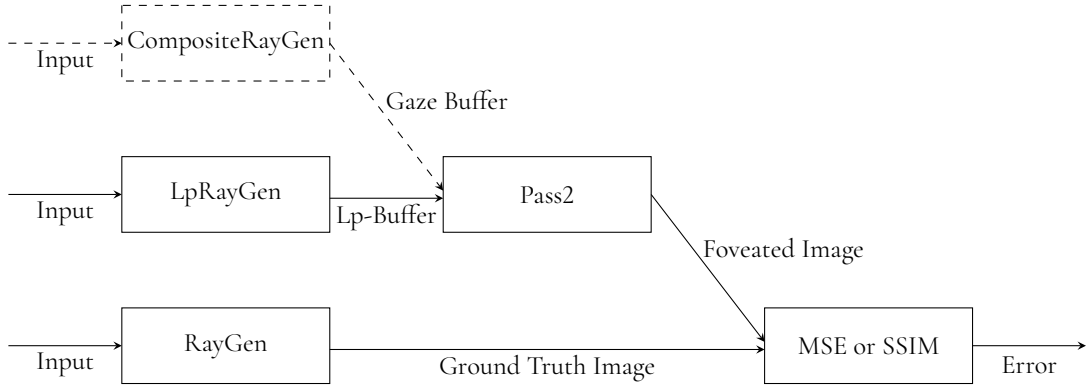


Figure 4.3: A basic view of the dataflow between shaders when calculating errors of a foveated image compared to the ground truth image.

4.3.2 SSIM Shader

Our SSIM shader calculates the SSIM index of all pixels using a neighborhood of size 11x11 [16]. SSIM uses luma instead of color to calculate errors. We convert the images from RGB color space to luma using the following weights [15] in equation 4.1.

$$Luma = 0.2126 \cdot R + 0.7152 \cdot G + B \cdot 0.0722 \tag{4.1}$$

The calculated indices are stored in a buffer. The average of all the indices is then calculated in the C++ code.

4.3.3 Ranges

When ranges are used, the selected error calculation shader is initiated with a number of dispatch calls equal to the number of ranges. On each dispatch call, the errors within the current range are calculated and then averaged. Figure 4.4 illustrates splitting the screen into 20 ranges. We split the screen into 20 ranges since we want enough ranges to be able to showcase the error at differing distances from the focal point while maintaining a sizeable amount of pixel samples in each range. The variable `max_range` controls the size of the area to be split into ranges.

4.4 Modeling the Acceleration Structure

All information about the acceleration structure presented in this section was obtained by inspecting the data of acceleration structures built on an Nvidia Titan V. The data has also been inspected on an Nvidia Titan X to check for similarities and differences.

Our frustum tracer implementation relies on our findings of the acceleration structure. The frustum tracer was tested using the aforementioned GPUs as well as an Nvidia RTX 2080 Ti, with no behavioral differences detected. All statements about the acceleration structure data format are based on our own conclusions. Our descriptions of some data sections are

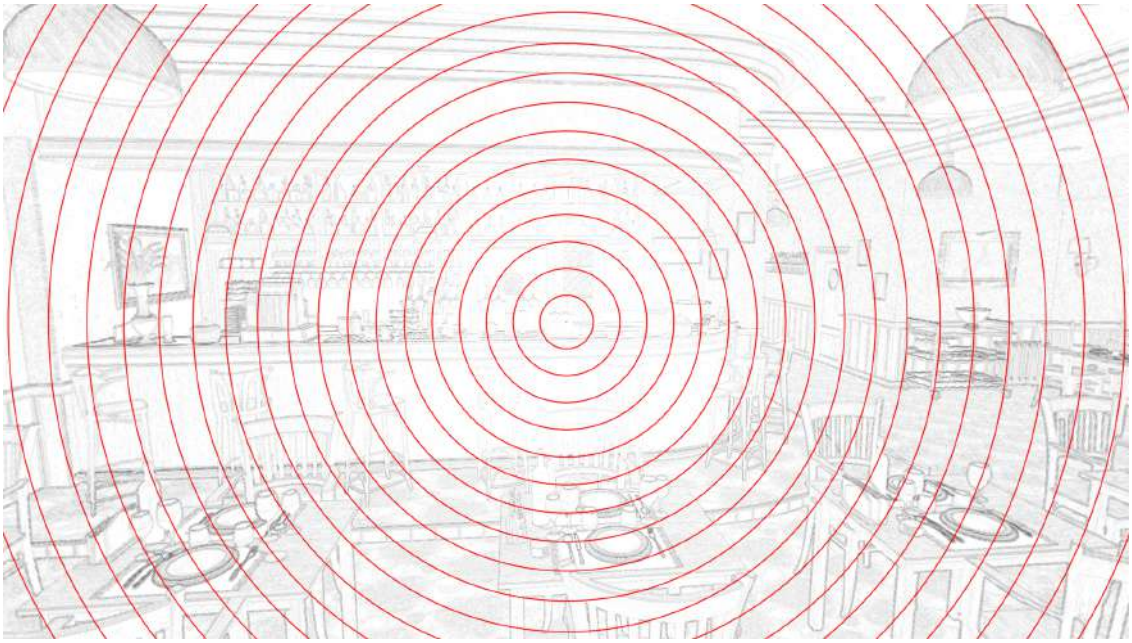


Figure 4.4: A view of the error heatmap using 20 ranges with `max_range` specified as 0.682 (the focal point is set in the center of the application window).

incomplete and we skip other sections entirely. Our goal was not to gain a complete understanding of the data format but instead gain the information necessary to implement the frustum tracer. Once a data section, or part of a section, was deemed unnecessary for our progress toward frustum tracing, our attention was directed elsewhere. Everything we specify about the data format is obtained by investigating TLAS, but we would like to note that BLAS data does appear similar to TLAS.

The data types encountered in the acceleration structure are identical to the C++ types float, UINT8 (8-bit unsigned integer), and UINT32 (32-bit unsigned integer). When we are unsure if we encounter a UINT32 or a UINT64 (64-bit unsigned integer) value, we treat it as a UINT32 value. The reason we settle with UINT32 in all cases is that it works well in our tests. Problems occur if the higher four bytes of a UINT64 are used while the data is being treated as a UINT32, but even in cases where we tested acceleration structures containing over a million instances, no issues were detected. The data uses little-endian ordering [8].

After building a TLAS using DXR functions, it is located in a part of GPU memory called the default heap. This memory is not accessible by the CPU. We copy the acceleration structure to the readback heap for CPU access. The procedure for copying occurs twice. First, four specific bytes describing the acceleration structure size are copied. The size value is then used to copy the entire acceleration structure.

The first section of the acceleration structure data is a header. A description of the content in the header section can be found in table 4.1.

The Instance section contains a set of sequential entries. The instance entries are 0x80 bytes in size and the number of entries is equal to the number of instances specified in the header section. The transform matrix data for an instance is located in their entry.

The BVH section consists of a series of BVH entries or nodes. The first node appears at the beginning of the section. This is the root of the BVH and has an index of 0. The size of

Offset	Type	Description
10	UINT32	Acceleration structure size in bytes
24	UINT32	Number of instances
28	UINT32	BVH section offset in bytes
30	UINT32	Instance section offset in bytes

Table 4.1: Description of fields in the acceleration structure header section. The offsets are provided as hexadecimal numbers.

Offset	Type	Description
00	float	node lower x coordinate
04	float	node lower y coordinate
08	float	node lower z coordinate
0C	UINT8	node width
0D	UINT8	node height
0E	UINT8	node depth
0F	UINT8	node branch mask
10	UINT32	node branch index
18	8xUINT8	8x child branch value
20	8xUINT8	8x child AABB lower x coordinate
28	8xUINT8	8x child AABB lower y coordinate
30	8xUINT8	8x child AABB lower z coordinate
38	8xUINT8	8x child AABB upper x coordinate
40	8xUINT8	8x child AABB upper y coordinate
48	8xUINT8	8x child AABB upper z coordinate

Table 4.2: Description of fields of an entry in the BVH section. The offsets are provided as hexadecimal numbers.

```
ce00:  75 83 66 c2 82 62 99 c0
ce08:  b0 a6 04 c2 7e 7d 7e d0
ce10:  01 00 00 00 00 00 00 00
ce18:  20 21 23 00 3c 22 3e 3f
ce20:  03 18 00 00 32 1c 1d 1d
ce28:  00 01 ae 00 4b a8 0b 0e
ce30:  03 19 00 00 1f 1c 1e 1f
ce38:  dd c7 e0 00 b0 c3 c3 79
ce40:  b1 a8 ba 00 af bb 78 89
ce48:  87 72 8a 00 5b 6d 6a 6a
```

Figure 4.5: A BVH node.

one node is 0x50 bytes and the section consists of an array of nodes. The data of a node is described in table 4.2, and an example node can be seen in figure 4.5. The node's lower x, y, and z coordinates make up its bottom-left corner. The top-right corner of the node can be computed using the bottom-left corner, as well as the width, height, and depth, using formula

(4.2). The bottom-left corner and top-right corner make up the node box. The node box is an axis-aligned box, but we do not use it as a bounding box. The node's AABB is defined in its parent entry in relation to the parent's node box. In other words, a node box is used to define the AABBs of its children.

$$top\_right = bottom\_left + \left(2^{width-77}, 2^{height-77}, 2^{depth-77}\right) \tag{4.2}$$

All field descriptions starting with "8x" are eight sequential UINT8 values, one for each of the node's eight potential children. Since there are up to eight children per node the BVH tree structure is an octree [13]. The children's lower x, y, and z coordinates make up the bottom-left relative corner called C_L, and their upper x, y, and z coordinates make up their top-right relative corner called C_H. C_L and C_H specify coordinates within the node box. The children's bounding volume coordinates are calculated using C_L, C_H, and the node box coordinates in equation (4.3). The $\circ$ sign is the Hadamard product [9].

$$\begin{aligned} bottom\_left_{AABB} &= bottom\_left + \frac{C_L \circ (top\_right - bottom\_left)}{256} \\ top\_right_{AABB} &= bottom\_left + \frac{C_H \circ (top\_right - bottom\_left)}{256} \end{aligned} \tag{4.3}$$

Each child has a branch value of which the lower 3 bytes specify the child's branch number and the upper 5 bytes specify the child's branch type. A branch type of 0x00 means there is no child, 0x20 means that the child is a leaf and 0x38 means that the child is a node. The node branch mask is used to compute 8 values called branch offset according to code listing 4.1. The node branch index, together with the branch offset, specify a child's entry index. The entry indices of children with branch type node are obtained by `entry_index[i] = branch_index + branch_offset[branch_number]`.

```
UINT8 i = 0;
while (i < 8 && (branch_mask & (1 << i)) == 0) {
  branch_offset[i] = 0;
  ++i;
}
if (i < 8) {
  branch_offset[i] = 0;
  ++i;
}
while (i < 8) {
  branch_offset[i] = branch_offset[i - 1]
          + !!(branch_mask & (1 << i));
  ++i;
}
```

Listing 4.1: This code uses the branch mask to compute the eight branch offset values.

Using the obtained information about the acceleration structure the BVH can be traversed. A simple AABB highlighter was implemented in order to evaluate the correctness of the AABB coordinate computations. An image of the highlighter can be seen in figure 4.6, which indicates correctness because the AABBs fit closely around the bunnies.

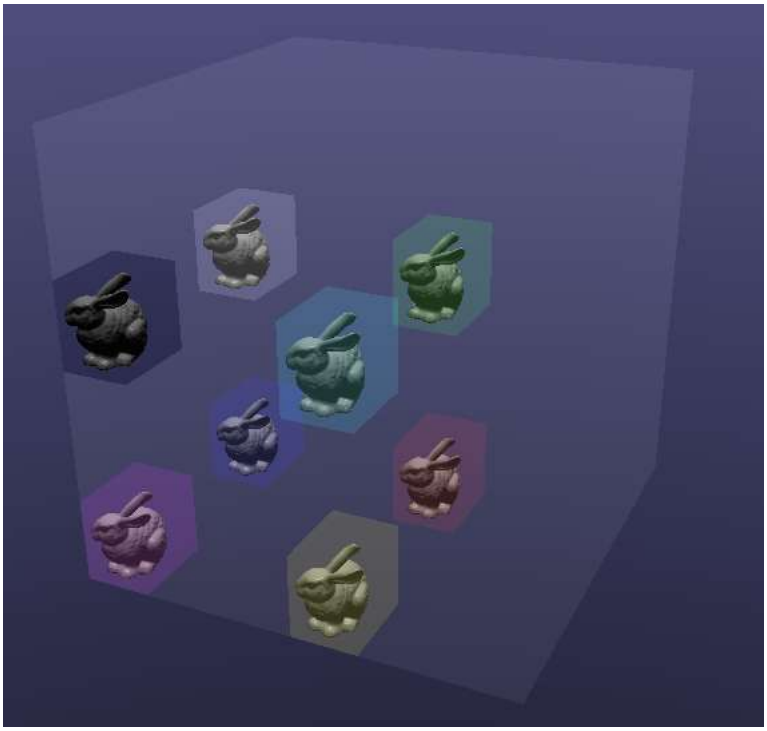


Figure 4.6: Highlighting of bounding boxes. The large faint box is the node box, the smaller colored boxes are the node's eight children's AABB.

4.5 Frustum Tracing

At the beginning of the program, an acceleration structure containing the scene is built, which we will refer to as the original TLAS. The frustum tracer creates modified versions of this acceleration structure to be used when ray tracing. The ray generation shader has access to the original TLAS as well as an array of acceleration structures modified by the frustum tracer, one per tile. The ray generation shader uses screen coordinates to determine which tile the current ray belongs to and uses the corresponding acceleration structure when tracing the ray.

When developing the frustum tracer we took inspiration from Hunt et al. [19]. They calculate an entry point of the BVH for each frustum. The BVH traversal for rays traced within a frustum start from its entry point instead of the root node. DXR limits us from modifying where the BVH traversal begins. Instead, we create copies of the acceleration structure and modify each copy.

We create three different versions of frustum tracing. The main version is called TLAS tracer and is described in section 4.5.1. The second version is called BTLAS tracer. It is used as a baseline in performance comparisons with the TLAS tracer and it is described in section 4.5.2. Both the TLAS tracer and BTLAS tracer culls whole instances from a TLAS. The last version is called BBLAS tracer. It modifies a BLAS to cull individual triangles and is covered in section 4.5.3.

4.5.1 TLAS Tracer

The TLAS tracer uses a copy of the original TLAS for each tile. The BVH of each copy is traversed and frustum-box intersection testing approximated according to our approach described in section 3.7. Every time an intersection test fails the branch type for the corresponding child is set to zero, and the child node is not traversed further.

The TLAS tracer was made to be used as a real-time frustum tracer. This means being executed every frame in a real-time application, which requires an execution time below 16.7 ms to reach 60 fps. However, as our results in section 5.4 show, the TLAS tracer fails to improve rendering time (which does not include the time spent culling). Because of this, we do not optimize the culling for speed, which could be achieved by parallelizing it per tile. This causes the TLAS tracer to become a bottleneck if the number of tiles enters the hundreds.

After implementing the BVH traversal and pruning, we added three optimizations to further reduce the size of the BVH: Trimming, entry point, and bridging. These optimizations can be toggled on and off.

Trimming

Trimming consists of removing nodes that are left without children after pruning and it occurs during the BVH traversal. A node box always fully contains the AABBs of its children. But the union of the children's AABBs almost never entirely fill their parent's AABB, illustrated in figure 4.6 by the empty space in the node box. This means that the TLAS tracer can traverse down a branch, and encounter a situation where a node is visited, but none of the children's bounding boxes intersect the frustum. The frustum tracer will then remove each of the entry's children. Having a node with no children serves no purpose, and the node can be removed from its parent to further reduce the size of the BVH as illustrated in figure 4.7.

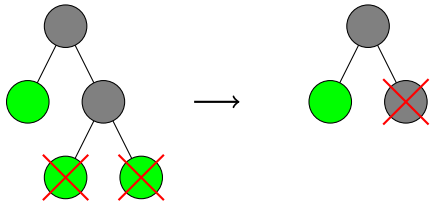


Figure 4.7: Visualization of a BVH before and after trimming. The gray circles are nodes and the green circles are leaves. In this case, the frustum intersected the node in the second row, but neither of its children's AABB. The node is therefore useless and removed.

Entry Point

Similar to Hunt et al. [19], we calculate an entry point. This occurs after the BVH traversal is completed. We achieve the entry point optimization by replacing the root node with its child node in cases where the root node only has one child remaining. This is repeated until the root node has multiple children. The process of setting an entry point is visualized in figure 4.8.

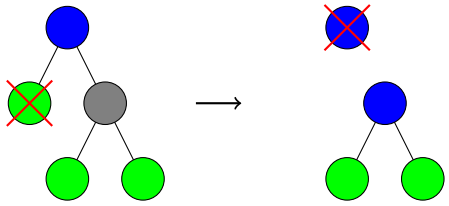


Figure 4.8: Visualization of a BVH before and after setting an entry point. The gray circles are nodes, a blue circle represents the root node and the green circles are leaves. In this case, the frustum only intersected one of the root node's children's AABB. That child is assigned to be the new root node.

Bridging

Our last optimization is called bridging. It is performed after the entry point has been set by traversing the BVH an additional time. Any nodes in the BVH that have only one child remaining is removed by bridging. The child of the removed node replaces the removed node, as demonstrated in figure 4.9. The child's AABB coordinates are recalculated to be relative to its new parent.

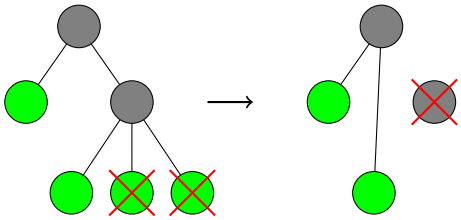


Figure 4.9: Visualization of a BVH before and after bridging. The gray circles are nodes and the green circles are leaves. In this case, the frustum intersects the node on the second row, but only one of its children's AABB. That child can therefore replace the node.

4.5.2 BTLAS Tracer

We implement another version of the frustum tracer to function as a baseline for calculating the performance gain in the near-optimal case for the TLAS tracer. The BTLAS tracer (Building TLAS tracer), unlike the TLAS tracer, does not access the acceleration structure data. Instead, it considers every model in the scene when performing culling. All models that are intersecting the frustum are collected and used to build a new TLAS using DXR functions. The resulting TLAS should be more optimized than the one edited by the TLAS tracer. The reason for this is that the number of branches from a node is typically reduced by our TLAS tracer, making the BVH tree taller than necessary, which increases the length of traversal from the root node to leaves.

This version requires a few seconds to perform culling and is not meant for real-time use. We only use it to show an upper limit of performance gain for the TLAS tracer.

4.5.3 BBLAS Tracer

We implement a third version of the frustum tracer. The BBLAS tracer (Building BLAS tracer) works by restructuring the BLAS. The idea is to have one BLAS for each frustum that contains all triangles within the current frustum. Like the BTLAS tracer, the BBLAS tracer does not access the acceleration structure data.

The reason for creating the BBLAS tracer is to investigate if there is any potential for performance gain by culling individual triangles instead of whole instances, like the TLAS and BTLAS tracers both do.

The time required to perform culling with the BBLAS tracer ranges from several seconds to hours, depending on the number of tiles and size of the scene, to execute and is not meant for real-time use.

Combined model

Before the BBLAS tracer is executed, all models in the scene are combined into one model. This is done once at the start of our application. The BBLAS tracer then culls this combined model. Apart from convenience for the BBLAS tracer algorithm, which considers every triangle in the scene when culling, the combined model serves one more purpose: We create a combined BLAS that contains the combined model, which is used in testing to investigate the effect of combining all BLAS data into one BLAS. Both the combined BLAS and the tile specific BLAS provided by the BBLAS tracer use a TLAS with a single instance.

Culling the combined model

The BBLAS tracer creates a new BLAS for each tile, consisting of all the triangles in the combined model that overlap their tile. The process of creating one of these BLAS starts by projecting each vertex of the combined model onto the screen and then recording whether they are in the tile or not. Then the BLAS is built using all triangles from the combined model that has at least one vertex in the tile. A small bias is added when determining if a vertex is considered to be in tiles to make the culling more lenient. This is to avoid culling triangles that intersect a tile without having any vertex inside it. Finally, one TLAS is built for each BLAS, containing only one instance of their corresponding BLAS.

Chapter 5
Results and Discussion

In this chapter, the results obtained by running our ray tracing implementations are presented. We evaluate the foveated ray tracer in different modes, where each mode specifies different values of the input parameters. We also explain the tuning process of the parameters. The ray tracers are evaluated on different GPUs and in different scenes. We present the rendering times and image quality errors obtained for the different modes. The results of running the frustum tracer are also presented.

5.1 Input Parameters

As described in chapter 3, there are four different input parameters in our application that affect foveation in different ways. σ changes the size of the log-polar buffer as described in section 3.3.1. α tunes the sampling to be distributed more towards the periphery (section 3.3.2). a_G changes the size of the composite foveation gaze buffer (section 3.3.3). `lp_aspect` changes the aspect ratio of the log-polar buffer (section 3.3.4). To narrow down the number of settings possible we have created four different modes in addition to ray tracing with one ray per pixel. Each mode serves to evaluate different aspects of the application. The values of the parameters were selected by tuning them for each mode and evaluating the result.

5.1.1 The Tuning Process

The parameters were tuned during run-time and evaluated visually until satisfactory quality was achieved. In this case, satisfactory quality entails keeping the buffers, and by extension the number of samples, as small as possible while the image quality reduction in the periphery remains imperceptible. We wanted to tune α, a_G, and `lp_aspect` in a way that made the radial and angular resolution match. As described in section 3.3.4 we observed less flickering and also improved perceived image quality when the rays were evenly distributed along the radial and angular axes. The function `mid_range_res`, described in section 5.2, was used to

calculate these resolutions. By matching the angular and radial resolution we ensure that the rays are well distributed along both the radial and angular axes.

The resolution plots by mid_range_res could possibly have been used to match the resolution of our presets as closely as possible to the perceived detail of human vision. In Patney et al. [2], they show that the level of detail that can be resolved by human vision differs from what can be detected. This means that we can not match the resolution in regards to both of these aspects without the use of contrast preservation methods, which Patney et al. use. That is beyond the scope of this thesis.

When evaluating the quality we viewed the screen from a specific distance calculated using formula (5.1).

$$distance\_from\_screen = \frac{screen\_diagonal\_length}{\tan(max\_view\_angle)} \tag{5.1}$$

During the evaluation process, we swapped between the ground truth and a selected foveation mode and checked if we could see any difference between them. There was a delay (showing nothing on the screen) between the two versions of the image. This delay is present to prevent us from noticing the sudden change caused by swapping directly between the two modes. Both Meng et al. [21] and Patney et al. [2] use a similar approach when evaluating image quality in their user studies. All tests were done with the focal point at the center of the screen and the application window set to the same size as the screen. The quality of the tuned modes was then compared to each other using the error metrics (section 5.3.4), to confirm that all modes had a similar image quality.

5.1.2 Modes

Mode	σ	α	a_G	`lp_aspect`
Fov	0.75	1.0	0	1.2
K Fov	1.0	3.0	0	0.70
C Fov	1.4	1.0	0.20	0.33
C K Fov	1.3	2.0	0.15	0.35

Table 5.1: The values chosen for σ, α, a_G and `lp_aspect` for each mode.

The four selected modes and their corresponding input parameter settings are collected in the table 5.1. The modes will be abbreviated, as in table 5.1, when used in some tables to save space. Fov stands for foveation, K stands for kernel and C stands for composite.

For clarification purposes:

- σ controls the size of the log-polar buffer. A larger σ yields a smaller log-polar buffer, which reduces the number of rays traced.
- α controls the effect of the kernel foveation. A larger α leads to rays being shifted away from the focal point more significantly, while $\alpha = 1$ disables the kernel foveation.
- a_G controls the size of the gaze buffer. A larger a_G leads to a larger gaze buffer, while $a_G = 0$ sets the size to zero and thus disables the composite foveation.

- `lp_aspect` is the aspect ratio of the log-polar buffer. A higher `lp_aspect` increases the width and reduces the height of the log-polar buffer.

The modes were selected to showcase how the optimizations, kernel foveation, and composite foveation, used separately and together, compare to each other as well as foveation without optimizations.

5.2 Matching Radial and Angular Resolution

The method `mid_range_res` is used to calculate the radial and angular resolution at set distances from the focal point. These set distances are located in the middle of the ranges used by the error calculations, hence, the name `mid_range_res`. By calculating these resolutions at increasing distances from the focal point we can create graphs, 5.1, illustrating how the angular and radial resolution changes depending on the distance to the focal point. The resolution along each axis is obtained by calculating how many pixels two adjacent rays span. More rays along one of the axes equal better resolution on that axis.

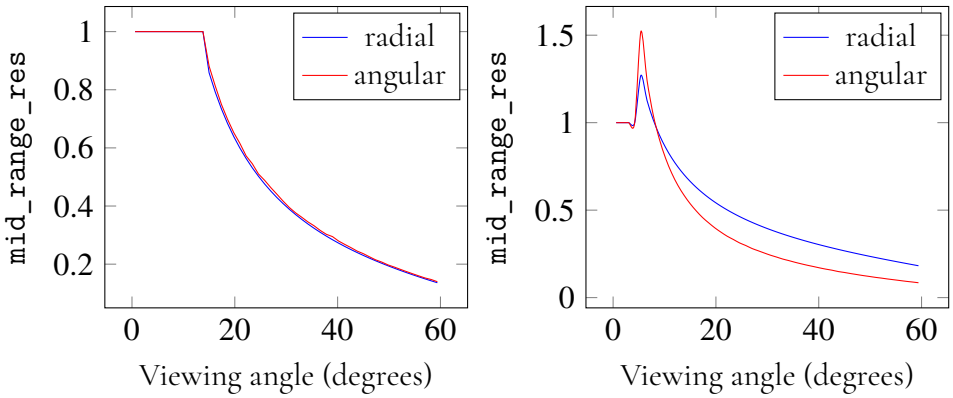


Figure 5.1: Graphs produced by the method `mid_range_res`. The left graph illustrates an example case where the angular and radial resolution is similar and decreasing further away from the focal point. The right graph illustrates an example case where the angular and radial resolution is poorly matched.

When using composite foveation the resolution inside the gaze buffer will always be treated as 1 by the `mid_range_res` method. This is because one ray is shot in each pixel inside the gaze buffer which makes the resolution 1 (ray per pixel). This is a special case since the resolution is neither angular nor radial but instead aligned along the x and y axes of the screen.

In both graphs in figure 5.1 and we can see that the gaze buffer is in use since the resolution starts at a stable 1 in both cases. In the right graph, we can see that the radial and angular resolution does not match and also that oversampling occurs when transitioning from the gaze buffer to the log-polar buffer. Using the graphs produced by `mid_range_res` we tune the parameters α, a_G and `lp_aspect` to reduce the oversampling and closely match the radial and angular resolution as seen in the left graph of figure 5.1.

5.2.1 Tuning

After tuning the input parameters according to the radial and angular resolutions the following four graphs (figure 5.2) were obtained for the different modes.

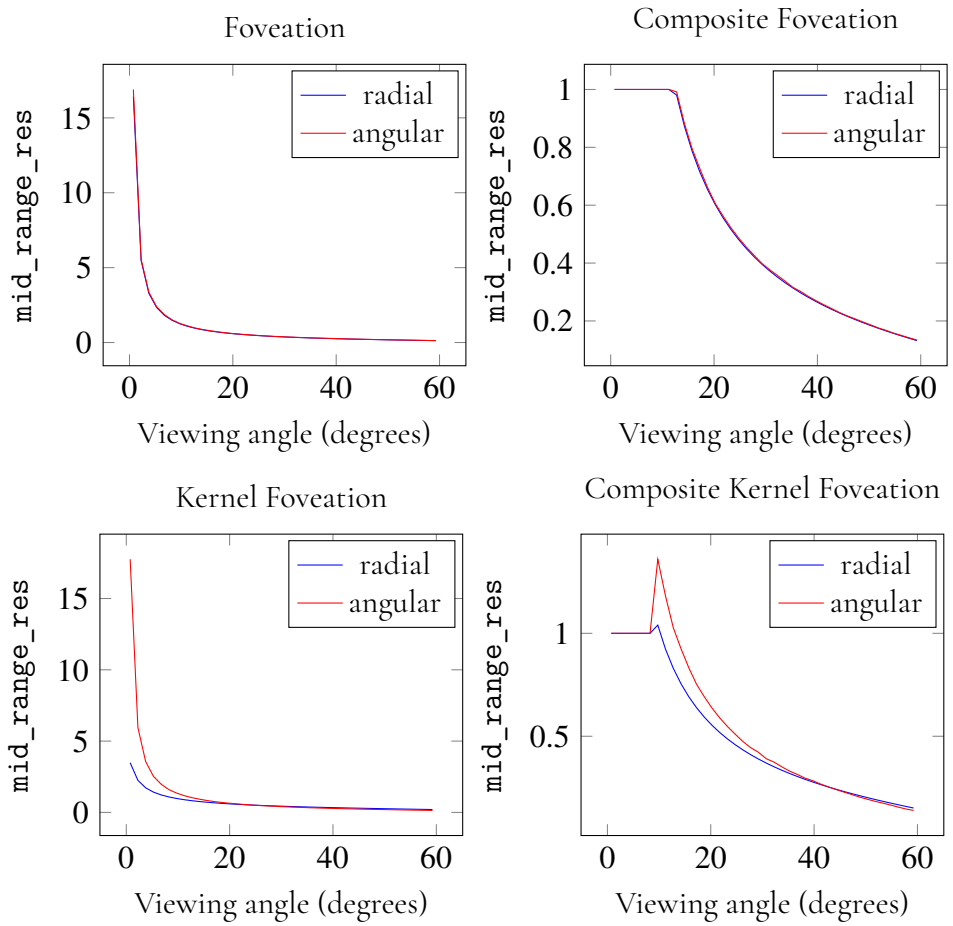


Figure 5.2: Four graphs obtained by running `mid_range_res` on the four modes.

In the two top graphs, we can see that the radial and angular resolution are almost identical. This is possible by using `lp_aspect` to change the aspect ratio of the log-polar buffer which also changes the ratio between angular and radial resolution. We can also see in the top right graph that the resolution of the log-polar transform is closely matched to 1 when transitioning from the gaze buffer. The modes in the bottom two graphs both use the kernel function. The kernel function pushes samples from the focal point towards the periphery which changes the slope of the radial resolution. This change makes it impossible to completely match the angular and radial resolution using `lp_aspect`. This is why the bottom two graphs are tuned in a way that makes the resolutions differ close to the focal point (where the number of samples is high) and are matched towards the periphery (where the number of samples is lower). We prioritize matching the resolution (syncing the graphs) in the periphery because we noticed, during visual evaluation, that the visual quality in the periphery was the limiting factor for how much we could reduce the number of rays.

5.3 Running Our Foveated Ray Tracer

In this section, we present the results obtained when running and evaluating our foveated ray tracer. We tested our foveated ray tracer on two different GPUs, NVIDIA TITAN X and NVIDIA TITAN V. The reason for using these two GPUs is that they were the ones we had access to with support for DXR. We use them both since we wanted to see if there were any interesting differences in performance when using two high-end GPUs that were released with over a year between them (Titan V is newer and more powerful). The different tests of the foveated ray tracer modes use two different scenes shown in figure 5.3. The scenes are Bistro (interior) [1], and Sponza. Bistro consists of 1,046,609 triangles and Sponza consists of 262,267 triangles. We use Bistro and Sponza when evaluating the rendering time of the foveated ray tracer. We also use Bistro when computing errors.

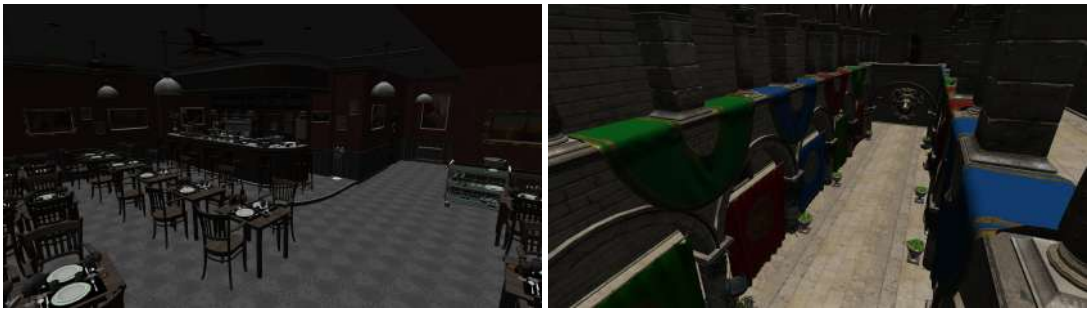


Figure 5.3: The left image is the scene Bistro and the right image is Sponza. Both images are rendered using the ground truth ray tracer.

5.3.1 Performance

The rendering times obtained from running the basic ray tracer, introduced in section 3.1, and the four foveation modes can be seen in table 5.2. The modes are tested both with and without TAA enabled.

The number of rays dispatched and traced in each mode can be seen in table 5.3. TAA is excluded from the table since it does not change the number of rays. The number of rays dispatched is displayed in percentage compared to the number of rays the basic ray tracer dispatches (1 ray per pixel). The column "Rays Traced" contains the percentage of rays that are traced compared to the basic ray tracer. The basic ray tracer traces all rays it dispatches. One case where a ray is dispatched but not traced, when using the foveated modes, happens when it is outside the screen/application window after transforming from log-polar coordinates. This phenomenon is discussed in section 3.3.1 and is demonstrated by the black color at the right side of the log-polar buffer in figure 2.1. The other case happens when the coordinates for a ray is outside the gaze image circle in the gaze buffer, demonstrated by the black color towards the corners in figure 3.2.

Discussion of Performance Data

Using the data from both tables 5.2 and 5.3 we can see that more rays dispatched/traced corresponds directly to longer rendering time. We can also conclude that the TAA algorithm only

Preset	Titan V Rendering Time in Sponza (ms)	Titan X Rendering Time in Sponza (ms)	Titan V Rendering Time in Bistro (ms)	Titan X Rendering Time in Bistro (ms)
Basic Ray Tracer	6.52	14.28	8.73	20.43
Fov	10.58	22.42	11.33	24.83
K Fov	5.62	11.70	6.49	14.49
C Fov	3.98	7.72	4.61	10.09
Fov TAA	11.07	22.75	11.78	25.23
K Fov TAA	5.91	11.97	6.79	14.79
C Fov TAA	4.21	7.87	4.87	10.20
C K Fov TAA	4.21	8.09	4.90	10.23

Table 5.2: The average rendering time of a single frame on each GPU for the scenes Bistro and Sponza.

Preset	Rays Dispatched (%)	Rays Traced (%)
Basic Ray Tracer	100	100
Foveation	162	154
Kernel Foveation	75.4	65.7
Composite Foveation	48.3	37.5
Composite Kernel Foveation	47.6	37.7

Table 5.3: A table containing the number of rays dispatched and traced in each mode. The percentages of rays dispatched and traced are given in comparison to the basic ray tracer, which both dispatches and traces 1 ray in each pixel.

introduces a small overhead calculation time when compared to the total rendering time. The rendering times for Sponza are faster than Bistro since Bistro is a more complex scene. Titan V is faster than Titan X in all of our tests, which is expected. The speedup is also similar. An interesting result obtained from the tables is that the foveated mode both dispatches/traces more rays and is slower than the basic ray tracer. This is because when tuning the foveated ray tracer we had to lower σ until the quality in the periphery was acceptable. When the periphery achieved sufficient quality the pixels towards the focal point are oversampled. This issue is mitigated by both the Kernel function and the composite foveation which is why they achieve better results.

The fastest mode is composite foveation which also traces the least amount of rays. The kernel foveation also achieves better results than the normal foveation. The number of rays traced has a more significant effect on the rendering time compared to the number of rays dispatched since the shading process is computation heavy. This is illustrated further by the composite kernel foveation which dispatches the least amount of rays but traces more rays than the composite foveation. Composite foveation and composite kernel foveation trace a similar number of rays and have similar rendering time. Hence, it is difficult to say for certain if one mode is strictly better than the other, especially since small changes in the input parameters could change the result. It should also be noted that the kernel function

and the gaze buffer strives to solve the same problem, which is why, when combined, there is no significant increase in performance when compared to only using the gaze buffer.

The resulting image when running the program in composite foveation mode, with a stationary view and TAA enabled, can be seen in figure 5.4. The other modes aim for similar quality as illustrated by the graphs in section 5.3.4.

Figure 5.4: Composite foveation with a stationary camera and TAA enabled. The other foveation modes have similar quality. The focal point is at the center of the image and there is lower quality towards the edges.

5.3.2 Ghosting and Flickering

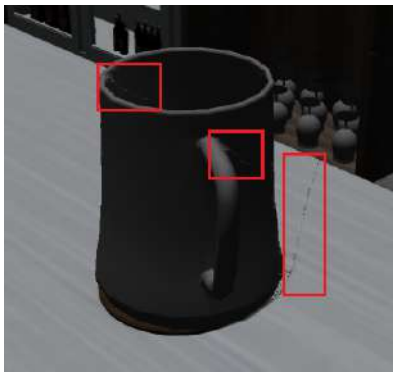


Figure 5.5: Figure illustrating the ghosting, contained in the red rectangles. The camera is moving towards the right.

When evaluating the program we noticed that some ghosting was persisting through the depth check when the camera is in motion. This ghosting is demonstrated in figure 5.5. This

ghosting is minimal and less prevalent towards the focal point but it still affects the results. This is the main reason why we decided to evaluate the application in two different scenarios with TAA active (stationary camera and moving camera), explained further in section 5.3.4. We have concluded that the ghosting is caused by the depth rejection being too lenient in some cases. However, in other cases, it is also too strict. Further away from the focal point, where the rays are more sparse, flickering appears when the rejection is more strict. This means that there is a trade-off between flickering and ghosting. Since flickering is a problem further from the focal point we decided to allow more ghosting in the periphery to reduce the flickering there.

Video

This video showcases the flickering on the geometry where it is most prevalent in the Bistro scene. The flickering is notable on the edges of the rolling table in the right part of the image.
`https://youtu.be/NSQar3p961g`

5.3.3 Error Heatmap

The figure 5.6 illustrates the error obtained when running the composite kernel foveation mode. Darker color indicates a higher error.

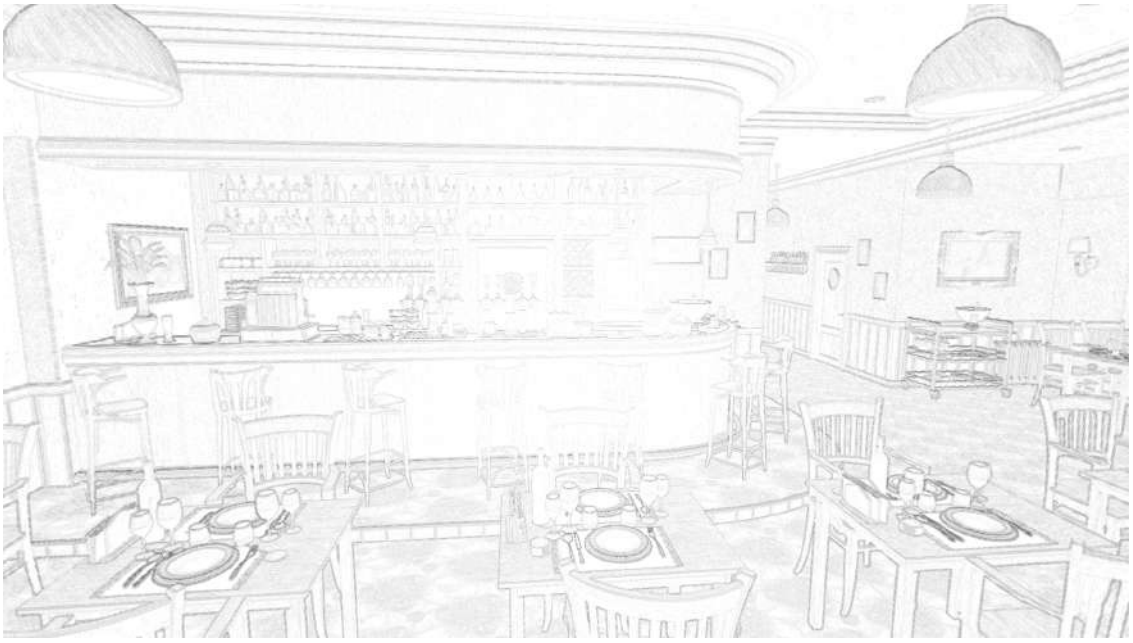


Figure 5.6: Error heatmap created using C K Fov TAA settings (with the focal point in the center of the application window).

Two important observations can be made from viewing this heatmap. The error is more prevalent further from the focal point than close. This is expected because there are fewer samples. The other observation is that the error only appears towards the edges of different objects in the scene. This is also an expected result since the error calculations operate on color and luma. The color and luma change drastically between different objects in the image which causes the errors to be prevalent in those areas.

5.3.4 Error Ranges

All graphs presented in this section are obtained by testing the selected foveation modes' output images against images produced by the ground truth ray tracer (described in section 3.2). As mentioned in section 5.3.2, we limit the tests with TAA active to two different scenarios. The first scenario is when the camera is stationary and the second is when the camera is moving. The errors vary depending on where the camera is looking, which is why, to achieve consistent results, we decided to use a set path for the moving camera tests. This way, the different modes compare the same views in all tests when the camera is in motion. We also noticed that there were small discrepancies in the results between different runs with the same settings. To avoid this we ensure that all tests start on the same index of the Halton sequence that offsets the rays. After this change, the tests produced consistent results on subsequent runs. The errors are calculated every 60th frame to allow the camera to move along the path to a view with differing scenery, to provide more variation to the measured images.

The scene Bistro is used for our error tests since it is the most detailed scene. When running the error tests we set `max_range` to 0.682 (a value selected so the error calculations cover the screen) and split that area into 20 ranges as shown in figure 4.4. The final error in each range is the average of the errors obtained in all evaluated frames. In addition to calculating errors using SSIM, PSNR, and RMSE, we also calculate the standard deviation (SD) of the errors in each range. The focal point is always in the middle of the screen during the tests. When visually evaluating the application we noticed that the blur caused by the Gaussian filter, presented in section 3.5, introduced tunnel vision. This is why we decided against using the Gaussian filter in the error tests in this section.

Without TAA

The results of running the foveation, kernel foveation, composite foveation, and basic ray tracer with no TAA are shown in the graphs in figure 5.7. The camera is moving along the set path, but when TAA is turned off there is no distinction in image quality between using a moving and a stationary camera. This is because the previous frames have no effect on the current frame. Hence, the results in this section are comparable to those in both section 5.3.4 and 5.3.4, which use TAA with stationary and moving camera respectively. The errors of 20 different views along the path were averaged when producing the graphs. An SSIM index close to 1, a high PSNR value, and a low RMSE value all indicate that the images compared are similar.

The basic ray tracer shoots one ray per pixel and should in an ideal case produce errors that are consistent over all ranges. We can see that the SSIM error is relatively consistent but RMSE and PSNR demonstrate a more fluctuating result close to the focal point. The inconsistencies in these results are caused by the differences in the views where the error measurements were taken. The ranges close to the focal point and towards the corners are also more inconsistent since they contain fewer pixels, as illustrated by figure 4.4. We can also see that the errors of the modes are similar with the exception of composite foveation close to the focal point. However, the composite foveation errors follow the basic ray tracer close to the focal point which means that the sudden changes in quality are caused by the specific views chosen. The shape of the standard deviation graph is similar to that of RMSE.

This means that the deviation of our errors is closely correlated with the size of the errors. The same observation holds true for the tests with TAA activated in the following sections.

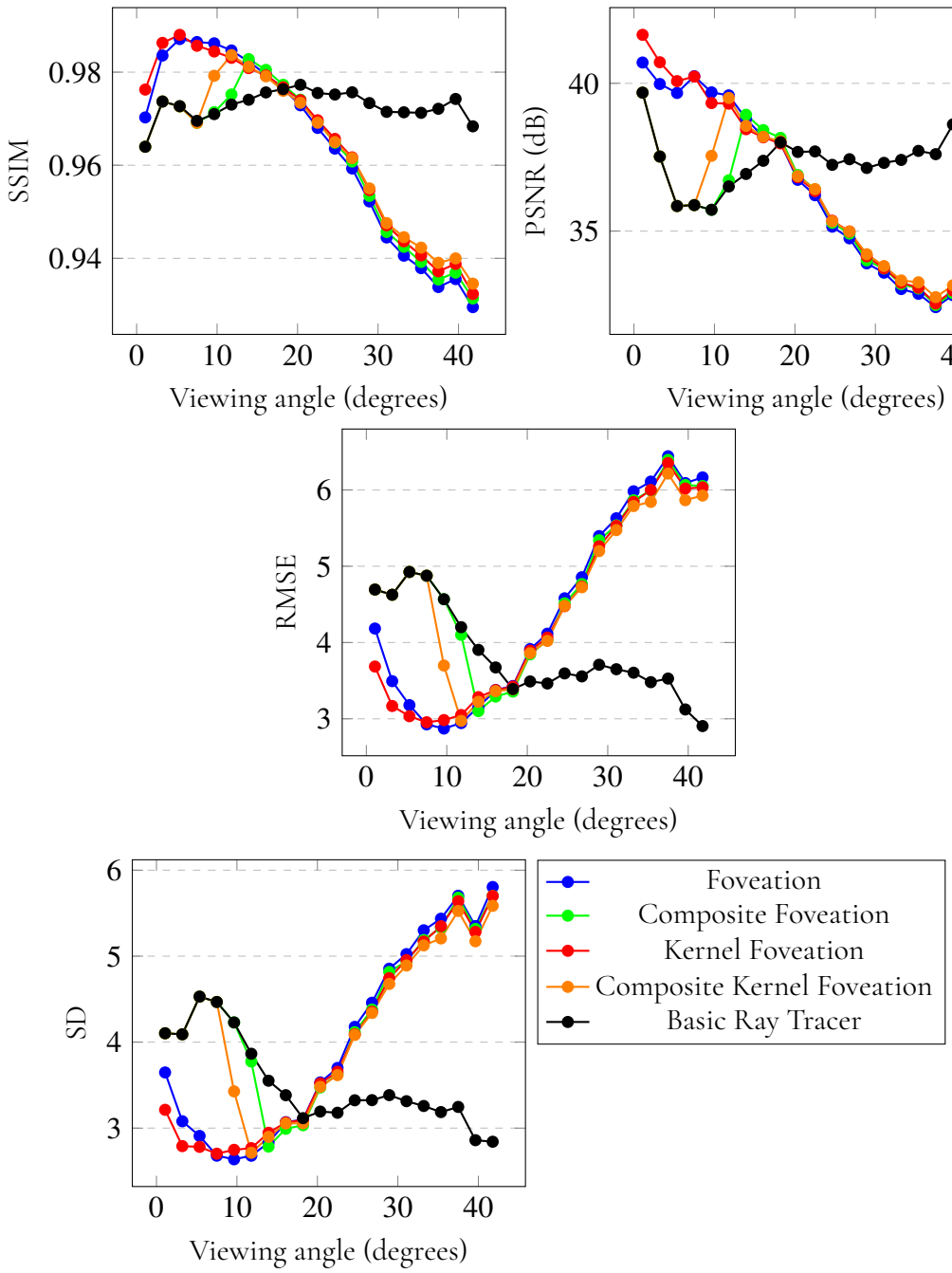


Figure 5.7: Errors obtained with TAA turned off. The errors are obtained as an average of 20 different camera views.

TAA and Stationary Camera

In the graphs 5.9 the errors of one selected view are presented with TAA turned on. We can see that the errors towards the periphery are not steadily increasing. Between 30 and 40 degrees, we see that the errors decrease instead of increase. By investigating the RMSE and PSNR of the basic ray tracer we can see that it changes in the same way but less pronounced. This means that the change in error is caused by the specific view we selected for the test. Hence, the view we used has less detail at about 30-40 degrees. In the graph 5.8, we ran the composite foveation mode in two different views. View 1 is the same view we used in our tests and the graph shows that view 2 does not have the same distribution of errors at around 30-40 degrees. This further supports the claim that the discrepancy in error at around 30-40 degrees is caused by the specific view we chose.

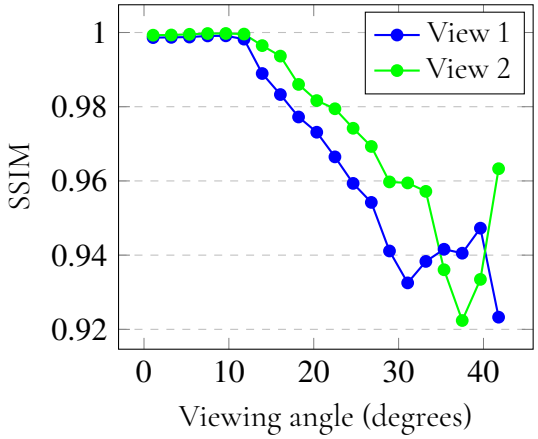


Figure 5.8: The SSIM values obtained when running composite foveation in two different views. View 1 is the same view we use during the stationary camera tests and view 2 is a different arbitrarily chosen view.

We observe that the composite modes perform better than the foveation and kernel foveation modes close to the focal point with TAA turned on. We can see that both the foveation and the kernel foveation modes' quality drop close to the focal point even though they have more samples in this region. The same behavior is observed in the graphs 5.7, where TAA is turned off. This is an effect caused by the oversampling. The oversampling causes a pixel close to the focal point in the log-polar buffer to map to a smaller area in the final image as the number of samples increases. Because of this, the linear sampler that extracts color from the log-polar buffer will linearly sample from pixels whose neighboring pixels contain essentially the same data. Hence, the linear sampler works more akin to a point sampler when the oversampling is present. When using TAA, the same phenomenon happens with the Halton pattern. Since a pixel close to the focal point in the log-polar buffer maps to a smaller area in the final image, the offset caused by the Halton pattern will also be small. When reading from a pixel in the log-polar buffer it will contain essentially the same data regardless of the pattern. For the composite modes, the pixels in the gaze buffer map 1:1 to the final image. Hence, when the Halton pattern is used, the offsets provide more information when compared to the foveation and kernel foveation modes.

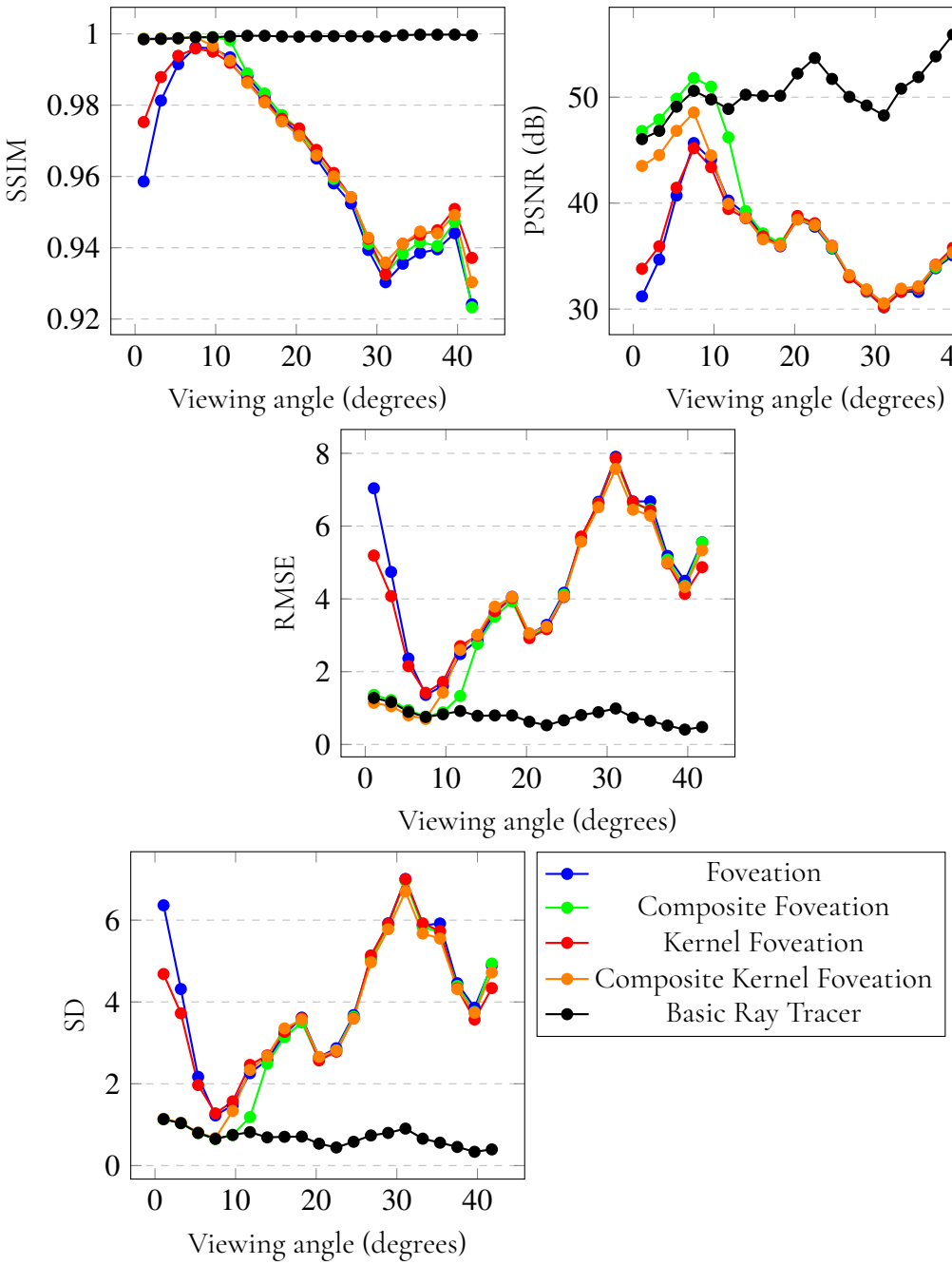


Figure 5.9: Errors obtained when using a stationary camera and TAA in one selected view.

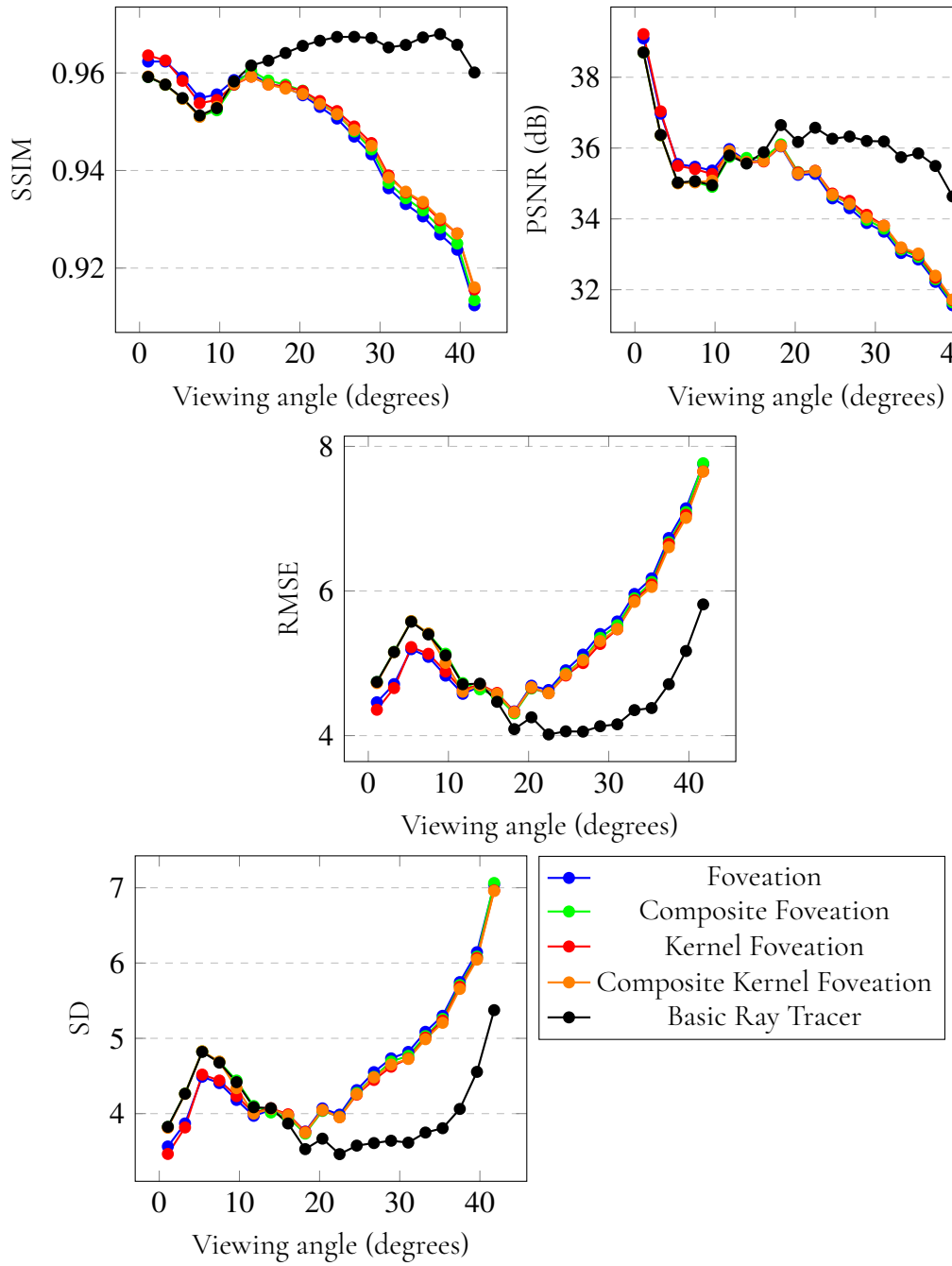


Figure 5.10: Errors obtained when using a moving camera and TAA. The camera is moving along a set path. The errors are obtained as an average of 20 different frames along the path.

TAA and Moving Camera

In the graphs 5.10 we see that the errors generally are higher when moving than in the graphs 5.9 with a stationary camera. This happens because of the history rejection when previously occluded geometry becomes visible which causes the image to contain less information when moving. Another cause is turning off the Halton pattern when moving (to reduce flickering) and only relying on the information that enters the history naturally when moving around. The last cause is the minor ghosting that appears when the wrong color is sustained through the depth rejection as explained in section 5.3.2.

The modes are showing similar error readings across the ranges which means that the tuning of the modes was successful. The errors are also higher towards the periphery as expected. The only discrepancy being the drop in quality at around 5 degrees from the focal point. This is caused by the specific views we average in the scene as demonstrated by the basic ray tracer which has a similar shape close to the focal point.

It is interesting that the error of the basic ray tracer has an increasing value towards the periphery. We suspect that this is caused by new geometry entering the view at the edges of the screen, which happens when the camera moves or rotates. Since there is no history present for this geometry, the quality is lower.

5.3.5 Trade-off Between Image Quality and Performance

In this section we calculate rendering speed and errors for one selected mode, using three different σ values. We use the composite foveation mode because we found it to be the best performing mode based on our results in section 5.3.1. The composite foveation mode used $\sigma = 1.4$ in the tests presented in sections 5.3.1 and 5.3.4. In the following tests we set $\sigma = 1.7$ and $\sigma = 2.0$ in addition to the previously tested $\sigma = 1.4$. We arrived at $\sigma = 1.7$ by testing what the smallest noticeable difference to σ was for us, and then $\sigma = 2.0$ by adding the same step size again. Running the same mode with a higher σ increases the rendering speed and reduces the image quality. Table 5.4 contains the performance data and the graphs 5.11 show the errors.

Preset	Titan V Rendering Time (ms)	Titan X Rendering Time (ms)	Rays Dispatched (%)	Rays Traced (%)
$\sigma = 1.4$	4.87	10.20	48.3	37.3
$\sigma = 1.7$	3.78	7.98	34.2	27.1
$\sigma = 2.0$	3.12	6.29	24.9	20.1

Table 5.4: The average rendering time in the scene Bistro of a single frame in the composite foveation mode with differing σ values.

The perceived loss in quality depends on whether the camera is stationary or moving. While the camera is stationary, the visual effect of increasing σ is that it induces a feeling of tunnel vision due to loss of contrast in the periphery. While the camera is moving, the already pronounced flickering artifacts become worse as σ increases. It is difficult to decide on an acceptable level of quality because of this flickering.

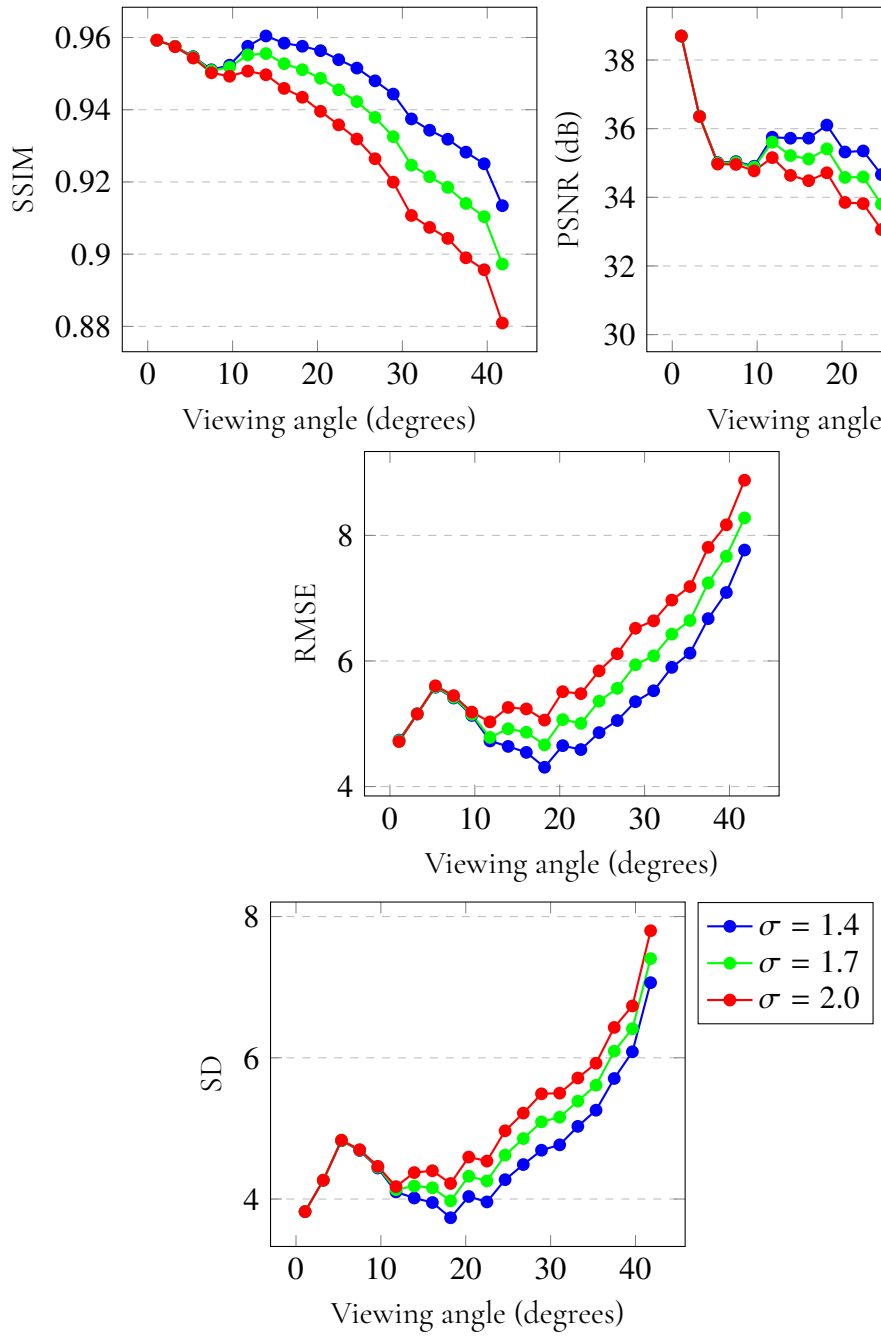


Figure 5.11: Graphs showcasing the errors of composite foveation with differing σ values. The errors are calculated in the scene Bistro with TAA activated and the camera moving along a set path.

Since it is difficult to decide exactly what sufficient quality entails, we decided to run these tests with varying σ to showcase how the rendering time changes when we allow lower quality. As stated before, we decided on sufficient quality based on our subjective opinion. If we had enough time and resources, a better way to decide what sufficient quality entails would be to match the image quality to the perceived level of detail of human vision. This can for example be done by using a perceptual target image, as Patney et al. [2] did. However, there would still be an issue with temporal aliasing because of the flickering. A sufficiently temporally stable image could be decided by carrying out a user study.

5.4 Running Our Frustum Tracer

In this section, we present the results obtained when testing our frustum tracer versions. We tested our frustum tracer on three different GPUs. Two of them are NVIDIA TITAN X and NVIDIA TITAN V, which are the same ones that we used when testing our foveated renderer in section 5.3. The third GPU is a GEFORCE RTX 2080 Ti. We included the RTX card in these tests because it has hardware support for ray tracing, which affects the performance of intersection testing. This is interesting because frustum tracing alters the intersection testing by culling the scene.

During the development of the frustum tracer, we noticed that it did not improve the performance. This caused us to set up test conditions that are favorable for frustum tracing to see if there is any scenario in which it can be used for performance gain. We create test scenes that consist of a large number of instances of the same 3D model. The model we use is the Stanford bunny [17], which consists of 65,630 triangles. The Stanford bunny can be seen in figure 5.12. The reason for having many instances is so that intersection testing takes

Figure 5.12: The bunny model we use.

up a relatively large proportion of the time spent rendering a frame. This makes it so that any improvement of performance for intersection testing is more noticeable. The reason for using the same 3D model for all instances is to increase cache efficiency. We arrange the instances in a square formation and view it from the side during testing. The reason for this layout is that it appears flat in this view, and because of this, rays intersect a small amount of geometry, which lets the frustum tracer cull larger sections of the BVH. Due to varying demands of memory between the BBLAS tracer and the other versions, we create two different test scenes. The first test scene is made for testing the TLAS tracer and BTLAS tracer and it consists of 16384 instances. The second test scene is made for testing the BBLAS tracer and contains 144 instances. We arrived at these numbers by increasing the number of

instances in each scene close to the point where the memory usage became an issue for their intended frustum tracing versions.

We measure the average time required to render a frame for each version of frustum tracing. We also compute the total memory usage of the acceleration structures used by each version. For comparison, we include rendering using an unmodified acceleration structure (normal AS). The poor performance of the frustum tracer caused us to never optimize the time spent culling, which could have been done by parallelizing the algorithm. The measured times therefore only consider the time spent rendering a frame. The time spent modifying the acceleration structures to perform culling is not included. This is also done to highlight any potential performance gain for intersection testing.

Preset	TLAS size (MB)	BLAS size (MB)	Titan V (ms)	Titan X (ms)	RTX 2080 Ti (ms)
Normal AS	2.5	6.0	14.48	15.83	2.7
TLAS Tracer	2400	6.0	14.55	15.96	2.8
Optimized TLAS Tracer	2400	6.0	14.63	15.72	2.8
BTLAS Tracer	12.7	6.0	14.34	15.43	2.5

Table 5.5: Test data of TLAS frustum tracing. The optimized TLAS tracer includes the three optimizations trimming, entry point, and bridging.

As seen in table 5.5, the TLAS tracer fails to yield faster rendering than the basic ray tracer. The BTLAS tracer shows that some gain is possible. However, it is not of sufficient significance since the time spent performing culling is omitted. The memory usage of the BLAS is the same for all three tests because it is never modified. As mentioned in section 2.5, the screen is divided into tiles. Having many, small tiles is desirable because it allows for more instances to be culled. Here we used 960 tiles. The reason we did not use more tiles is because of memory constraints. The TLAS tracer uses one copy of the original TLAS for each tile. The copies are edited, but their size remains unchanged. This results in a total memory usage increase of a factor of 960 for the TLAS tracer in our tests. The BTLAS Tracer also uses 960 TLAS, but each is smaller than the original TLAS since they are created with only essential data. In our tests, the BTLAS uses five times as much memory for TLAS than the normal AS. The results show that our optimizations for the TLAS tracer have little effect. We did expect the rendering time of the optimized TLAS tracer to be somewhere between the unoptimized TLAS tracer and the BTLAS tracer, but that is not the case for the tests using Titan V, which performed worse with the optimizations enabled.

Preset	TLAS size (KB)	BLAS size (MB)	Titan V (ms)	Titan X (ms)	RTX 2080 Ti (ms)
Normal AS	24	6.0	14.39	14.06	1.9
Combined BLAS	1.3	867	22.30	15.46	17
BBLAS Tracer	19	868	22.14	15.36	16

Table 5.6: Test data of BLAS frustum tracing.

Combining the entire scene into a single BLAS (Combined BLAS) decreases our rendering speed compared to the normal AS, as seen in the last three columns of table 5.6. Using

the per-tile acceleration structures, provided by the BBLAS tracing algorithm, yielded faster rendering than the combined BLAS. Still, it is slower than the normal AS rendering time. The TLAS size is reduced by combining the BLAS. This is because a new TLAS containing a single instance is used. The BLAS size of the combined BLAS and the BBLAS tracer are similar. They both contain all the bunnies. In the combined BLAS, the data is in a single BLAS, while the BBLAS tracer splits the data over several BLAS, one for each tile. In these tests, we used only 15 tiles due to the culling algorithm being slow.

Chapter 6
Conclusion

We have developed a foveated ray tracer, which uses different optimizations to improve performance and image quality. We use kernel foveation, composite foveation, and log-polar aspect ratio to improve and control the ray distribution pattern of the log-polar transform. We use TAA to reduce aliasing such as flickering. We also wanted to improve the performance of our foveated ray tracer by using frustum tracing. For this purpose, we implemented and evaluated a frustum tracer separately with the intention of combining it with the foveated ray tracer. However, because our frustum tracer alone did not achieve performance gain, we did not combine it with the foveated ray tracer.

We created a number of foveated ray tracing modes that use different combinations of the optimizations. We tuned the image quality of the modes to a similar level by evaluating them visually and by using error metrics. The performance of each mode is presented and we achieved the best results using composite foveation. Although, composite kernel foveation also achieved similar results.

We will now discuss the results of our testing in correlation with the research questions presented in section 1.2.

How much can the performance of ray tracing be improved by using log-polar foveation? Our log-polar transform-based foveation without optimizations suffers from oversampling near the focal point and low sampling rate in the periphery. This causes our foveation to perform worse than the basic ray tracer, which shoots one ray per pixel, when maintaining sufficient quality in the periphery. It is possible to tune the foveated ray tracer in a way that improves performance, compared to the basic ray tracer, while maintaining sufficient quality near the focal point. However, the limiting factor while reducing the number of samples is the periphery. The image quality preservation methods that we use (TAA) do not sustain the quality in the periphery enough when the sample count decreases. To ensure sufficient quality in the periphery we increase the size of the log-polar buffer to increase the number of samples in the periphery. The number of rays traced then becomes 154% compared to the basic ray tracer.

One reason for the poor performance could be that we evaluate our application on com-

puter screens. We suspect that by evaluating the program on an HMD, where the field of view is larger, the ray count could be lowered. When the field of view is large, the outer edges of the screen is further out towards the periphery of the user's vision, which means that the reduction of quality is less noticeable.

How much can the performance of ray tracing be improved by using foveation with optimizations? Using optimizations such as the kernel function and gaze buffer the oversampling can be reduced while also increasing the sampling rate in the periphery. This allows us to reduce the size of the log-polar buffer and the number of rays traced while maintaining the same image quality. Our experiments show an increase in speed by a factor of two compared to the basic ray tracer, using optimized foveated rendering. We achieved our best results using composite foveated rendering, and the hybrid composite kernel foveated rendering, both of which create stationary imagery that to us is indistinguishable from the basic ray tracer while tracing less than 38% of the rays.

A major part of our project has been to investigate the potential of using frustum tracing with DXR for foveated rendering. An issue with using this technique in DXR is the requirement of multiple acceleration structures. This increases the amount of memory used by a large factor depending on the number of tiles used. Our tests point to frustum tracing being of no use with DXR. While testing in favorable conditions, and not measuring the time spent performing the frustum tracing algorithm, the rendering times did not improve. We suspect several reasons for the poor rendering time results. One reason could be that often accessing different acceleration structures reduces cache efficiency. It might be that intersection testing only takes up a small portion of the rendering time, which would limit the potential gains from performing this step faster. Another reason could be optimizations performed by DXR or graphics drivers. Such optimizations might be unknown to us and could conflict with our assumptions. The TLAS tracer failed to yield faster rendering times than ray tracing using the unmodified acceleration structure. While the BTLAS tracer did provide 1-3% faster rendering, it is not feasible to use it in a real-time application. The BBLAS tracer suffered performance-wise from using the combined BLAS.

How large is the difference in image quality between foveated ray tracing and a ground truth image? We have measured image quality using three different error metrics: RMSE, PSNR, and SSIM. In this section, we focus on the SSIM metric since all three metrics show similar trends in the graphs in section 5.3.4.

When the TAA is turned off, our foveation modes, that use the gaze buffer (composite foveation and composite kernel foveation), achieve an SSIM index of around 0.97 close to the focal point. The modes that do not use the gaze buffer (foveation and kernel foveation) have higher quality ranging from an SSIM index of 0.97 to 0.99. The quality in the periphery is similar for all modes and becomes lower towards an SSIM index of about 0.93. However, since TAA is not active, severe flickering is observed when the camera is in motion.

When the camera is stationary with TAA active, the modes using the gaze buffer achieve an SSIM index close to 1 near the focal point. This value shows that the image quality is almost identical to that of the ground truth image. The modes that do not use the composite buffer have lower quality towards the focal point, with an SSIM index of 0.97-0.99. However, this loss of quality is not observable to us during visual evaluation. The quality in the periphery becomes lower towards an SSIM index of about 0.93-0.94. There are no notable flaws in perceived image quality in this case (stationary camera with TAA active).

When the camera is in motion and TAA is active, the quality of all modes is similar, but

generally lower than in the case of using a stationary camera. Close to the focal point, we observe an SSIM index of about 0.96. Towards the periphery, we observe an SSIM index of about 0.92. These values show that, when the camera is in motion, the quality is generally lower when TAA is on compared to off. However, as previously stated, the flickering is severe when TAA is turned off. This means that even though the quality is better according to the error metrics when TAA is turned off, the flickering seen during visual evaluation shows that TAA has to be active. This conclusion also highlights that flickering is not well represented by the error metrics we use.

6.1 Threats to Validity

In this section, we discuss different threats to the validity of our research and results.

The first threat is connected to the fact that we have not conducted a user study and instead based the visual evaluations on our own perceptions of the images. If we had more people evaluate the images, the claims about sufficient quality would be more robust.

Another threat is artifacts such as flickering and ghosting and also that the error metrics do not reflect the flickering well in the image quality graphs.

The tuning of the input parameters of the foveated ray tracer could be another threat to the validity of our results. There could be more optimal values for each parameter which could be showcased by a potential user study.

Our implementation of frustum tracing relies on assumptions. We do not know exactly how an acceleration structure is used when ray tracing is performed on the GPU.

6.2 Future Work

In this section, we present ideas for how our evaluation methods and application could be improved in the future.

6.2.1 Head-Mounted Display

As stated in section 2.2 foveation's main use case is speeding up rendering on HMDs. We evaluated the application on computer screens and it would be interesting to implement the program on an HMD and evaluate it in the correct environment.

6.2.2 User Study

The error metrics work well to demonstrate that different foveation modes have similar quality. However, they can show inconsistent results in combination with the error ranges calculation used in our project. These inconsistencies are caused by the scene and views chosen for the tests. It is also a problem to quantify the quality and deduce what qualifies as sufficient quality using the errors alone. The metrics neither reflect flickering well in the graphs. This is why conducting a user study, where different users test and evaluate the image quality, would be a good idea in the future. This would be useful to better motivate an optimal choice of input parameter values.

6.2.3 Animated Scenes

Our TAA implementation assumes that the scenes used are static. Adding support for animated scenes could widen the scope of the project and would require support for motion vectors in the TAA algorithm.

References

[1] Amazon Lumberyard Bistro. https://developer.nvidia.com/orca/amazon-lumberyard-bistro. [Online; accessed September-2020-22].

[2] Anjul Patney, Marco Salvi, Joohwan Kim, Anton Kaplanyan, Chris Wyman, Nir Benty David Luebke and Aaron Lefohn. 2016. Towards Foveated Rendering for Gaze-Tracked Virtual Reality. *ACM Trans. Graph.* 35, 6, Article 179 (November 2016), 12 pages. https://research.nvidia.com/publication/2016-12_Towards-Foveated-Rendering.

[3] Bounding Volume Hierarchy. https://en.wikipedia.org/wiki/Bounding_volume_hierarchy. [Online; accessed September-2020-22].

[4] Brian Karis. 2014. High quality temporal supersampling. Advances in Real-Time Rendering in Games, SIGGRAPH Courses 1 (2014).

[5] Brian Guenter, Mark Finch, Steven Drucker, Desney Tan and John Snyder. 2012. Foveated 3D Graphics. *ACM Trans. Graph.* 31, 6, Article 164 (November 2012), 10 pages. https://dl.acm.org/doi/10.1145/2366145.2366183.

[6] DirectX documentation. https://docs.microsoft.com/en-us/windows/win32/directx. [Online; accessed September-2020-22].

[7] DirectX Raytracing (DXR) Functional Spec. https://microsoft.github.io/DirectX-Specs/d3d/Raytracing. [Online; accessed September-2020-22].

[8] Endianness. https://en.wikipedia.org/wiki/Endianness. [Online; accessed September-2020-22].

[9] Hadamard product. https://en.wikipedia.org/wiki/Hadamard_product_(matrices). [Online; accessed November-2020-5].

[10] Lei Yang, Shiqiu Liu and Marco Salvi. 2020. A Survey of Temporal Antialiasing Techniques. Computer Graphics Forum 2020 The Eurographics Association and John Wiley

Sons Ltd. 39, 2, (July 2020), 15 pages. https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.14018.

[11] Martin-Karl Lefrançois and Pascal Gautron. DX12 Raytracing tutorial. NVIDIA. https://docs.microsoft.com/en-us/windows/uwp/graphics-concepts/shader-resource-view--srv-, [Online; accessed September-2020-22].

[12] Matt Pharr, Wenzel Jakob and Greg Humphreys. 2018. Physically Based Rendering: From Theory to Implementation. Morgan Kaufmann 2017 (2018), ISBN 9780128007099. http://www.pbr-book.org/.

[13] Octree. https://en.wikipedia.org/wiki/Octree. [Online; accessed September-2020-22].

[14] Open Asset Import Library. https://www.assimp.org/. [Online; accessed September-2020-22].

[15] Relative Luminance. https://en.wikipedia.org/wiki/Relative_luminance. [Online; accessed September-2020-22].

[16] Structural similarity. https://en.wikipedia.org/wiki/Structural_similarity. [Online; accessed October-2020-19].

[17] The Stanford 3D Scanning Repository. http://graphics.stanford.edu/data/3Dscanrep/. [Online; accessed September-2020-22].

[18] UAV and SRV documentation. https://docs.microsoft.com/en-us/windows/uwp/graphics-concepts/shader-resource-view--srv-. [Online; accessed September-2020-22].

[19] Warren Hunt, Michael Mara, and Alex Nankervis. 2018. Hierarchical Visibility for Virtual Reality. *Proc. ACM Comput. Graph. Interact. Tech.* 1, 1, Article 8 (May 2018), 18 pages. https://doi.org/10.1145/3203191.

[20] Amy Williams, Steve Barrus, R. Keith, and Morley Peter Shirley. An efficient and robust ray-box intersection algorithm. *Journal of Graphics Tools*, 10:54, 2003.

[21] Xiaoxu Meng, Ruofei Du, Matthias Zwicker and Amitabh Varshney. 2018. Kernel Foveated Rendering. Proc. ACM Comput. Graph. Interact. Tech. 1, 1, Article 5 (May 2018), 20 pages. https://xiaoxumeng1993.wixsite.com/xiaoxumeng/kernel-foveated-rendering.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2020-10-9

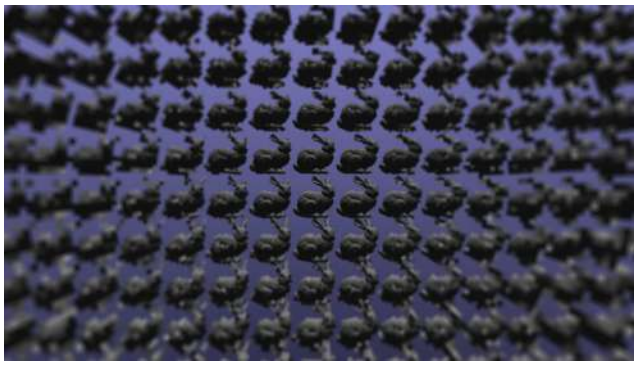
EXAMENSARBETE Performance Optimizations for Foveated Real-Time Raytracing
STUDENTER Kalle Andersson, Tom Hansson
HANDLEDARE Michael Doggett (LTH)
EXAMINATOR Flavius Gruian (LTH)

Förbättrad prestanda för foveation

POPULÄRVETENSKAPLIG SAMMANFATTNING **Kalle Andersson, Tom Hansson**

Metoder som minskar mängden beräkningar för att generera bilder på datorskärmar är användbara för att öka prestanda. Foveation är en sådan metod och fungerar genom att reducera bildkvaliteten i användarens periferiseende. Detta arbetet undersöker olika metoder för att göra foveation snabbare.

Att generera bilder av 3D scener som visas på datorskärmar kan kräva många beräkningar. Ett sätt att minska antalet beräkningar är att generera bilder i lägre upplösning. Foveation är en metod som används för att minska upplösningen utan att en användare märker någon skillnad på bildkvaliteten. Detta görs genom att enbart minska upplösningen i användarens periferiseende, där ögonen uppfattar detaljer sämre.



För att kunna reducera upplösningen enbart i användarens periferiseende måste blicken spåras. Därmed är denna metod bäst lämpad för virtuell verklighet, vid användning av en huvudmonterad skärm försedd med blickspårare. Vi använder dock vanliga datorskärmar istället för huvudmonterade skärmar när vi evaluerar våra bilder.

I vårt examensarbete undersöker vi olika metoder som förbättrar prestandan för foveation samt deras påverkan på bildkvaliteten. Vi implementerar olika optimeringar som förändrar algoritmen som kontrollerar hur vi uppnår foveation. Vi testar olika modifieringar av algoritmen mot varandra och jämför deras prestanda och bildkvalitet. Vi implementerar även en optimering som försöker öka prestandan genom att dela upp scenen i mindre delar och generera varje del av bilden för sig med färre beräkningar. Vi evaluerar bildkvaliteten visuellt men även genom att räkna ut olika felvärden i foveation bilden jämfört med en annan bild som är genererad i hög kvalitet utan foveation.

Våra resultat visar att optimeringen som delar upp scenen i mindre delar inte kan användas för att öka prestandan för foveation i de användningsområden vi testade. De andra optimeringarna som förändrar foveation-algoritmen ger dock bättre resultat. Vi lyckas öka prestandan och generera bilder på hälften så lång tid jämfört med vår utgångspunkt, samtidigt som bildkvaliteten hålls på en hög nivå. Vår implementation har dock vissa problem med flimmer i periferiseendet när vyn är i rörelse.