MASTER'S THESIS 2022

Image Upscaling for Ray Traced Foveated Rendering

Charlie Mrad

ISSN 1650-2884
LU-CS-EX: 2022-35

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2022-35

Image Upscaling for Ray Traced Foveated Rendering

Bildförstoring för strålspårad fovearad rendering

Charlie Mrad

Image Upscaling for Ray Traced Foveated Rendering

Charlie Mrad
ch3045mr-s@student.lu.se

June 20, 2022

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

Foveated rendering is a potential optimization that can have a big impact on render times for computer graphics, and we have seen in recent years that image upscaling and AI driven supersampling are getting more popular. Therefore we investigate the relationship of these two areas and how well they work together by testing Nvidia's Deep Learning Super Sampling (DLSS) with foveated rendering. We look at the image quality and performance and visually inspect the results to see if the combination is useful. In the process we find that the two technologies are not initially compatible and produce subpar images without applying other methods to reduce this issue. For this purpose we use a method that makes use of temporal anti-aliasing to stabilize the resulting image and find that we can get upwards of a 1.75X increase in foveated rendering performance for a small reduction in image quality, all the while maintaining a temporally stable image. This seems like a promising result for future applications of foveated rendering for areas such as virtual reality where we typically renders one screen per eye and ideally at very high resolutions. In such use cases, there is a big need for performance optimizations wherever possible and taking advantage of image upscaling through DLSS would be ideal.

Keywords: ray tracing, DXR, real-time rendering, foveation, kernel log-polar, DLSS

Acknowledgements

I would like to thank Micheal Doggett for all the help during this project, and for providing the ray tracing hardware necessary to do it. I would also like to thank Anders Bruce for his help setting up remote access which greatly increased my productivity. And of course, I would also like to thank my friends and family for their support.

Contents

Chapter 1
Introduction

Foveated rendering is an important rendering technique for virtual reality (VR) graphics applications, as it allows reducing the amount of work needed to get a good quality image rendered. The principal idea is that areas where the eye is not focused can be rendered at lower sample rates. Today, relatively high-quality real-time ray tracing is possible due to advancements in graphics hardware. One such advancement is the use of hardware implemented neural networks that can be used to generate super sampled and/or upscaled images from relatively low sample resolution input images. Nvidia's version of this is called deep learning super sampling (DLSS) which is what we focus on in this project.
Since both the field of VR and ray tracing are seeing more widespread popularity and technological advancement, it can be important to study how compatible these fields are and what kinds of problems are encountered when they interplay. Thus, we have looked at the effects of applying DLSS to a foveated input image to determine the compatibility of current DLSS technology and VR-optimizing foveated rendering techniques.

1.1 Project scope

For this project we implement a real time ray tracer that supports sampling according to the foveal falloff of the human eye. We use a kernel log-polar transformation [19] to perform such sampling. For the rest of this thesis, this kind of sampling method will be referred to as foveated sampling for the sake of brevity. The ray tracer is built on DirectX 12 and uses the DirectX Raytracing (DXR) API to do a simple form of Whitted ray tracing [29] with some minor additional features such as cone sampled shadows that produce smoother shadow results. The goal of the project is to evaluate the performance of Nvidia's DLSS feature which is an AI powered upscaling and anti-aliasing algorithm [28, 6]. We look in particular at the results when DLSS is combined with the kind of foveated rendering technique mentioned above.

The ray tracer is implemented to allow for some flexibility when adjusting settings but

we primarily use four major setting configurations for this project. We will refer to a setting configuration for the ray tracer as a ray tracer mode for the remainder of the thesis. The four primary ray tracer modes are firstly, a foveated rendering mode using the foveated rendering technique described in sections 2.2 and 4.2. Secondly, we use two modes that are identical to the first except for the fact that they either have enabled DLSS or have enabled both DLSS and a temporal anti-aliasing (TAA) feature which is discussed in greater detail in sections 2.4.2 and 4.4. Finally, we have the ground truth mode that does not apply any foveated rendering, instead sampling uniformly across screen space and using a higher sample rate per pixel. In our case we use 36 samples per pixel in this mode to provide an accurate approximation of the optimal visual quality we can achieve.

The results are evaluated on the basis of performance of the rendering in terms of time taken per frame, split up over the various stages of rendering. Additionally, we measure the quality of the images taken with the various modes of ray tracing. To do this we use the ꟻLIP metric which provides a perceptible difference evaluation [4].

1.2 Research questions

For this project we have chosen to mainly focus on the following research questions:

- What is the render time reduction gained by introducing DLSS to foveated ray traced rendering?
- What is the visual quality loss of using DLSS with foveated ray traced rendering as compared to the ground truth?
- How temporally stable is the foveated render result when DLSS is applied? Meaning, how many visual artifacts do we see that manifest over time?

1.3 Contributions

This project contributes the following to the fields of real time ray tracing and eye-resolution dependent rendering:

- A measurement of the trade off between visual quality and render time when applying Nvidia's DLSS technology to foveated rendering in a ray tracer.
- A method to selectively apply temporal anti-aliasing to improve the temporal stability of the peripheral region of the rendering before applying DLSS.

Chapter 2
Background and theory

In this chapter we will cover any prerequisite knowledge required to understand the implementation of our ray tracer and the results we present. In order, we go over fundamental computer graphics concepts and techniques, how foveated rendering is done, and how ray tracing works, especially in the context of DXR. Finally we talk about what DLSS does and how it works to the best of our knowledge.

2.1 Computer graphics basics

In order to understand the techniques employed in our ray tracer and also to understand the problems that need addressing for applications of computer graphics we cover some relevant topics below. This includes an overview of the rasterization rendering method, the rendering equation and Phong shading.

2.1.1 The rasterized graphics pipeline

Most modern graphics cards rely on a rasterization pipeline to produce images that can be put on screen. This has been the norm for decades now and as a result the rasterization pipelines built into graphics cards have become complex and also quite well optimized. In this following section we will give a basic overview of this rasterization pipeline by splitting it into three basic stages, geometry processing, rasterization, and finally color processing. This will provide a decent baseline understanding of basic computer graphics rendering and some of the various terms used with it, as well as allowing us to compare this commonly used rendering method to ray tracing later on. For this subsection we draw heavily from resources provided by Akenine-Möller et al. [1], Lengyel [15] and Gregory [9].

Geometry processing

The first thing that happens in the rasterization pipeline is processing of geometry input. Usually that comes in the form of a set of triangles, but other primitives and methods of interpreting the geometry exist. The graphics card generally expects this geometry to be passed in as a list of vertices and potentially also a list of indices pointing to various locations in the vertex list that describes how to put these vertices together into some primitive shape. For simplicity we will assume all geometry is always interpreted as triangles for the rest of this section. Now, during this stage the graphics programmer has to provide the GPU with a custom program called a vertex shader. This program allows for manipulation of the vertices being processed, such that you could for example offset the vertices of a highly tessellated plane to create hills and valleys. The vertex shading step is also crucially responsible for transforming the vertices into clip space. In this space, any vertex that falls within a unit cube with its extreme points at (-1, -1, -1) and (1, 1, 1) is in view of the camera. Thus, the responsibility of defining what is in view of the camera falls on the programmer providing the vertex shader.

Once the vertex shading is complete the transformed vertices are passed on to the rest of the geometry processing stage which will proceed to clip the geometry. This means that any geometry which falls outside of the unit cube and therefore, outside the view of the camera, is discarded and the edge cases are cut and split into smaller triangles. The effect is that there is a reduced number of triangles that have to be processed in subsequent stages.

Finally, all the remaining geometry is transformed into screen coordinates that represent the position on the actual screen in pixels. The z coordinate of each vertex is also remapped to a predefined range, usually [0, 1]. The screen pixel coordinates and the remapped z coordinate, also known as the depth value, are then passed onto the next stage.

The rasterizer

The triangles that are output from the last stage now need to be mapped to sets of pixels on the screen. Whilst the triangles are represented in screen space coordinates at this point, the process of determining exactly which pixels that should be covered by each triangle is not trivial. This is what rasterization solves and we will be covering a basic overview of the steps involved below.

The rasterizer has to first take each triangle and determine which pixels should be considered for further processing. One way of determining this is to check whether a pixel's center lies within the triangle or not, if it does then it should be considered part of the triangle. The rasterizer would therefore have to traverse some subset of pixels and check each one and this traversal can be done in multiple ways. The most important aspect of the traversal is to limit how many pixels you need to check and there are different approaches to accomplish this, which we will not write about but commonly used in hardware today is a tile-based rasterization system which splits the screen into tiles and selects tiles to process for each triangle.

The rasterizer is also responsible for interpolating values across the pixels of the triangle. Triangles consist of three vertices each containing at a minimum the position of the vertex, but they can also contain additional attributes such as normals, colors, texture coordinates and so on. Therefore, in order to get a smooth final result we have to interpolate all of these values (including the vertex positions) along the triangle for each pixel that is covered by

the triangle. The way this is done is by using the triangles' perspective corrected barycentric coordinates at the pixel center; using them as weights for the values from each corresponding vertex.

Another important job of the rasterizer is to determine pixel visibility, that is to check if the current pixel is occluded for a given triangle. To do this the rasterizer uses something called depth tests, which make use of the depth value calculated during geometry processing. The depth test algorithm works as follows; first, we check the depth of the current pixel for the current triangle and compare it to the minimum depth encountered at the current pixel so far; then, if the depth is greater than the minimum it is occluded and therefore we discontinue processing the pixel for the current triangle, otherwise, if the depth is less than or equal to the depth we continue processing and store a new minimum depth for the current pixel. This process ensures that where triangles overlap in screen space, they still appear in the correct order.

Pixel processing

Each time a pixel gets processed and passes the depth test it, along with all of its interpolated vertex attributes are passed along to finally calculate the pixel color and write that color to a color buffer. This step is usually the bottleneck of the rasterization pipeline as the programmable pixel shaders that calculate the pixel color can often get complex and are also often called several times per pixel.

The pixel processing stage is required to output a color that can be presented on screen and the way the color is calculated is very dependent on what effect is desired. The complexity of a pixel shader can be as simple as outputting a single color for all pixels or it can be more involved, using textures, normals, material properties and more to create more complex effects.

Below we briefly cover two commonly used techniques that we implement for this project; those are normal mapping and mipmapping, which are explained in great detail by Akenine-Möller et al. [2]. We also make use of Phong shading which is another common technique and we cover that in section 2.1.2.

Normal mapping is used to give surfaces a detailed look without greatly increasing the complexity of the geometry. The idea is to use a texture which encodes the normals on a per pixel level along the surface. This allows using complicated normals on simple geometry that when combined with lighting can give perfectly flat surfaces the illusion of depth and detail. The encoded normals are usually stored as vectors in tangent space which is the space with a basis consisting of the normal, tangent and binormal at the respective point on the surface.

Mipmapping is a technique that reduces texture bandwidth and texture aliasing due to subsampling when several texture pixels map to a single screen pixel. The way mipmapping works is by creating several versions of a texture at different resolutions and using smaller resolution versions the more severe the subsampling is. As an example if you have a texture with a resolution of 256×256 you could for example create versions of this texture at the resolutions 128×128, 64×64, 32×32 and so on down to 1×1. Each successively smaller version is downsampled from the previous version, usually applying some kind of filter in the process. Another consequence of using smaller resolutions for your textures in some areas is that not all textures have to be loaded at full resolution at all times and can therefore save on bandwidth when textures are uploaded to the GPU. We use this technique to reduce texture

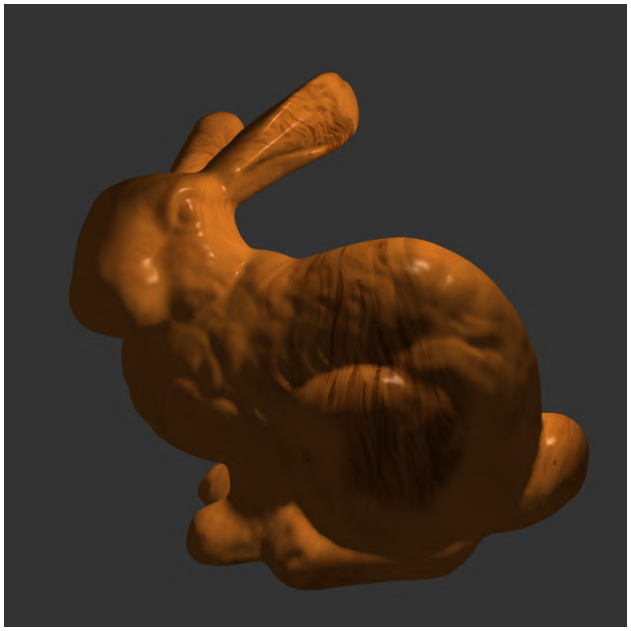


Figure 2.1: Phong shaded Stanford bunny with a diffuse texture. Here $k_a = (0.3, 0.3, 0.3)$, k_d is obtained from the diffuse texture and multiplied with $(0.710, 0.396, 0.114)$ to obtain the orange tint, $k_s = (0.3, 0.3, 0.3)$ and $\alpha = 70$

aliasing only, we do not stream textures at runtime and therefore can not take advantage of the reduced texture memory size.

2.1.2 Phong shading

Phong shading is a well known shading model for calculating a light source's impact on a surface with parameters for ambient, diffuse and specular lightning effects [22]. Phong shading is generally calculated through

$$S = k_a + k_d \texttt{max}(n \cdot l,\ 0) + k_s \texttt{max}(r \cdot v,\ 0)^{\alpha} \tag{2.1}$$

where

S is the color of the surface.
k_a is the ambient light parameter.
k_d is the diffuse light parameter.
k_s is the specular highlight parameter.
n is the surface normal.
l is the normalized direction vector to the light source.
r is the normalized direction vector of the light reflected off the surface.
v is the normalized direction vector from the surface point to the camera.
α is the shininess of the material.

In equation 2.1 above, k_a, k_d, k_s represent colors for the respective lighting effect. In practice we often obtain the diffuse color parameter, k_d from a texture and in doing so allow the diffuse color of an object to freely change across the surface. Also notably, the $\texttt{max}(n \cdot l,\ 0)$

factor from the diffuse term will grow until the surface normal is pointing straight at the light source meaning that diffuse light contributions are predictably strongest where the surface faces the light. On the other hand, the $\texttt{max}(r \cdot v, 0)$ factor from the specular term will grow as the reflected light around the surface normal gets closer to pointing straight at the camera, giving the surface some shiny spots mimicking how shiny objects get their specular highlights in reality. The end results look something like in figure 2.1, and as you can see the resulting surface often looks somewhat like plastic and this model is not enough to truly represent a more diverse range of materials. However, it is still good enough for our purposes since we are not focusing on any specific shading model.

2.2 Foveated rendering

As mentioned earlier, foveated rendering refers to the process of concentrating computational resources on areas of the screen that are close to the point where the eyes are focused. This can be done in a variety of ways, some of which are summarized by Mohanto et al. [20] and include:

- Reducing geometric complexity towards the periphery.
- Reducing the quality and bandwidth requirements of colors.
- Varying the shader complexity and cost.
- Rendering at variable resolutions at different areas of the screen.
- Reducing the rate at which peripheral areas are updated.

These can further be categorized based on whether they are static or dynamic. Respectively that means either assuming the eyes are focused at some point or regions of the screen, or adapting to the shifting focus of the eyes in real time as it changes by using something like an eye tracker. The type of foveated rendering we chose for this project essentially renders at variable resolutions and for our tests we assume the user is looking at the center of the screen since we lack eye tracking hardware. However, our focus point for the eyes is freely movable in real time without any issues. This means that we could integrate our implementation with eye tracking hardware to make it adapt to the user without any obvious problems. For the remainder of this chapter we go into more detail about why foveated rendering is promising and what specific method we chose to implement.

2.2.1 Human eye acuity

The human eye is not uniform in how well it performs at various tasks across the visual field [26]. This is related to the density of the photosensitive rods, cones and ganglion cells being unequally distributed in the eye. The distribution is visualized in figure 2.2, in which can be seen that the concentration of cone and ganglion cells are greater towards the center of the view and decline with increased eccentricity. Part of that central area is called the fovea and roughly encompasses the central 5 degrees of the human visual field [23]. However, the density of ganglion cells drops drastically as eccentricity increases which is the primary

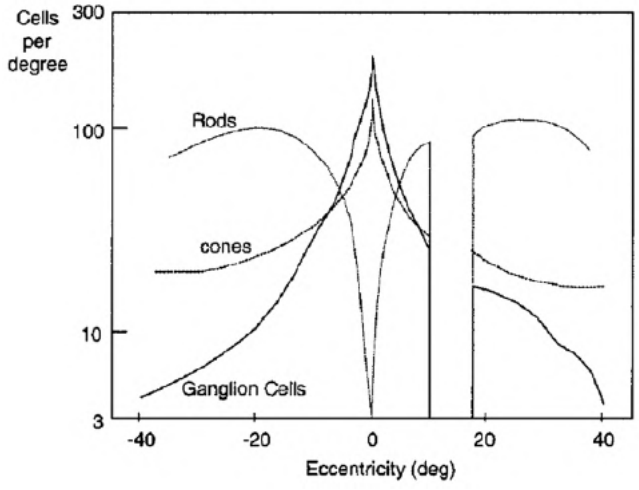


Figure 2.2: Visualized here is the distribution of cones, rods and ganglion cells in the eye as a function of eccentricity [27]. The absence of these cells at 10°–20°can be attributed to the blind spot of the human eye.

reason that the peripheral vision suffers from reduced detail perception [7]. This is what we attempt to leverage with foveated rendering in order to obtain better performance.

This falloff in visual acuity has the potential to save large amounts of rendering resources with little impact to the perceived image quality, assuming that the rendering takes into account the current fixation point of the eyes and the periphery does not contain noticeable visual artifacts. In fact, the foveated rendering approach presented by Meng et al. [19] managed to obtain a 2.8 – 3.2 times speedup without losing much perceived detail according to their user study. Koskela et al. [12] managed to achieve roughly a 2 times minimum speedup with ray tracing on a head mounted display.

Application of foveated rendering to head mounted displays such as VR headsets is a natural optimization to make since only one user is supported per display and they tend to have higher field of views and resolutions than traditional displays. Additionally, if displays start reaching resolutions that rival the capabilities of the human eye it might become necessary to apply some form of foveated rendering to these systems due to the sheer workloads involved.

2.2.2 Kernel log-polar space

A natural way of generating a foveated rendering is to utilize the log-polar transformation since that is very similar to how our own eyes project the world onto their retinas [5]. This transformation maps circles around a central point (the foveal point for our purposes) to vertical lines in log-polar space and maps lines extending radially outwards from that central point to horizontal lines in log polar space. For applications in rendering, this means that the sample density decreases linearly with distance from the central point and is therefore useful in reducing the rendering cost. However, since this type of transformation subsamples the regions that are far from the central point we tend to get visual artifacts in said regions. Ideally, one would want for each log-polar sample to represent the average of the corresponding area that is covered in rectilinear space but that is not trivial to achieve with log-polar trans-

formations not aligning well with the pixels. One way to combat this effect is to increasingly blur the result as the sample position gets further from the central point, which is what we do. Another way is to not use log-polar transforms entirely and instead use something like the log-rectilinear transformations suggested by Li et al. [16], wherein they use summed area tables to efficiently obtain the average of the area covered in normal rectilinear space.

For this project, as we have mentioned before, we chose to use a kernel log-polar transformation that is an extension on the log-polar transform provided by Meng et al. [19]. Given the central point $C = (\mathring{x}, \mathring{y})$ and rectilinear pixel coordinates (x, y), the transformation to and from kernel log-polar coordinates (u, v) is described by equations 2.2 and 2.3 as follows.

$$\begin{aligned} u &= \mathbf{K}^{-1}\left(\frac{log||x', y'||_2}{L}\right) \cdot w \\ v &= \left(arctan\left(\frac{y'}{x'}\right) + \mathbf{1}[y' < 0] \cdot 2\pi\right) \cdot \frac{h}{2\pi} \end{aligned} \tag{2.2}$$

$$\begin{aligned} x &= e^{A \cdot \mathbf{K}(u)} \cdot cos(Bv) + \mathring{x} \\ y &= e^{A \cdot \mathbf{K}(u)} \cdot sin(Bv) + \mathring{y} \end{aligned} \tag{2.3}$$

Where,

$$\begin{aligned} x' &= x - \mathring{x} \\ y' &= y - \mathring{y} \\ \mathbf{1}[y' < 0] &= \begin{cases} 1 & y' < 0 \\ 0 & y' \geq 0 \end{cases} \\ A &= \frac{L}{w} \\ B &= \frac{2\pi}{h} \end{aligned} \tag{2.4}$$

and, w denotes the width of the kernel log-polar buffer, h denotes the height of the kernel log-polar buffer, L is the logarithm of the distance to the furthest corner from C and finally $\mathbf{K}$ denotes the kernel function with $\mathbf{K}^{-1}$ meaning the inverse of the kernel function.

From these equations we can see that the kernel function K changes how densely the rectilinear coordinates map to the u-axis of kernel log-polar space. When used in foveated rendering this has the intended effect of allowing flexibility in how the foveated samples are distributed radially outwards from the foveal point allowing easy adjustment and the potential to better mimic the human visual system's characteristics. We go into more detail about which kernel function we chose and what parameters we use for this transformation in Section 4.2.

2.3 Real time ray tracing

In contrast to rasterization which does a very good job of approximating the first recursion level of the rendering equation, ray tracing allows for a simple method of further refining the

rendered results by simulating the rays of light as they travel through the scene. However, the cost of this is a significant performance impact as the rays can essentially go anywhere in the scene therefore making optimizations harder to do. In order to make ray tracing feasible to do in real time applications (i.e. interactive applications such as games, 3D modeling software etc.) there has been a push towards producing hardware that specializes in calculating ray intersections with the scene geometry. For the remainder of this section we will discuss the concept and benefits of ray tracing in more detail and how ray tracing is done in practice, using the work of Haines et al. [10] as reference. We also cover what DXR is as well as some important technical details at the end.

2.3.1 Tracing rays

The basic idea of ray tracing is to shoot rays in random directions from each light source in the scene and then let these rays interact with the scene geometry until they either don't intersect with anything or hit the camera. Each time a ray intersects a piece of geometry during this process it can spawn zero, one or several other rays. For example, a ray intersecting a glass-like material can spawn a ray for the reflected portion of the light and one for the refracted light. The consequence of this is that the number of rays can quickly explode, resulting in degraded performance. In practice it is therefore usually required to put a limit on the recursion depth of the rays, with each newly spawned ray being one recursion level deeper. Depending on how the ray tracing is done, there can also be a significant amount of noise in the final rendered image due to the stochastic nature of how the rays can bounce around the environment and how they are launched from the light sources. To reduce the prevalence of this noise there are generally three approaches as follows; one could make all rays deterministic and in the process reduce the quality of the lighting; one could employ a denoising algorithm as a post-processing step; or finally you could also just increase the number of rays that are traced until the rendered image converges.

One of the big benefits of ray tracing as compared to rasterization is that ray tracing inherently provides an easy mean to produce effects that otherwise would be much harder to produce. Shadows for example, are built into the rendered results, at least when using the ray tracing strategy outlined above. For a rasterized renderer the process of producing realistic shadows often involves additional calculations and still often only simulates shadows convincingly within certain limitations. Another example would be reflections, which only require launching a reflection ray when using ray tracing, but oftentimes rely on capturing additional views or inaccurate screen space techniques when using rasterization.

Despite the relative simplicity and utility of ray tracing it has not been widely adapted for use in real time rendering because of its low performance. When tracing rays, the ray needs to be aware of all geometry in the scene in order to remain flexible in how and where rays can be launched. This means that many ray intersections could end up having to be tested, depending on the geometric complexity of the scene. Compounded with the rapidly ray multiplying nature of effects such as refraction and global illumination, rendering times become prohibitively long. It is only in more recent years as hardware has improved that ray tracing has started seeing usage in real time applications.

So far we have described ray tracing as an algorithm simulating light rays as they leave light sources, interact with the environment and eventually either hit or miss the camera. This is usually not how ray tracing is done in practice however, as a significant portion of the

launched rays might end up not contributing to the final image. To reduce wasted computations on rays that miss the camera, rays are usually launched from the camera instead. This essentially means the rays are traced backwards into the scene. The color imparted onto the ray by the lights in the scene has to be checked explicitly in this case, requiring rays to be cast in the direction of the light sources and the contributions of each light source gets added up. Shadows can also be calculated with those same rays.

2.3.2 The DXR pipeline

We built the ray tracer for this project using Microsoft's DXR API which is their DirectX 12 ray tracing API[1]. In order to give some context for the rest of this thesis and to explain some recurring terms we provide background information on the DXR API below. We cover five areas of DXR, namely miss shaders, closest hit shaders, any-hit shaders, ray generation shaders, and an overview of the pipeline structure.

Ray generation shader

The ray tracing starts by setting up the pipeline state and resources, and then dispatching the rays into the scene. When that happens DXR will trigger a special shader type called a ray generation shader which runs once for each pixel. The job of this shader is to create and launch one or more rays into the scene and it is also usually desired to write the results of the launched rays to a buffer for later use.

Closest hit shader

Each ray that is launched during ray generation has to find where the closest intersection with the scene is, if any at all. If the closest intersection is found then DXR will invoke the closest hit shader which can take into account where the intersection happened to change the ray information payload. For example, this could be assigning the ray a color. The closest hit shader can also launch other rays if need be, for example to calculate shadows or reflections.

Any-hit shader

While attempting to find the closest intersection DXR will search through an acceleration structure in some order, the order is not guaranteed but is deterministic. Thus we can use something called the any-hit shader to take advantage of otherwise wasted intermediate intersections. The any-hit shader is similar to the closest hit shader but instead of only being invoked for the closest intersection, it will be invoked whenever any intersection is found. It also has the ability to change how the pipeline flows, by for example accepting the current intersection as the closest one and stopping any further searching. This kind of shader can be useful for transparency and shadow for example.

[1] `https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html`

Miss shader

In case of no intersections being found then the ray is handled by the miss shader. Its job is to provide the ray payload with some information in case of a miss, such as rendering a sky background or just a solid color. It can not launch new rays and has no intersection information available since none were found for its respective ray.

Pipeline structure

Of the shaders covered in this section all but the any-hit shader are required for DXR to function. There is also another shader type called the intersection shader which we do not use since we assume all geometry will be supplied as triangles. The flow of the pipeline once rays have been launched looks as follows

1. DXR finds a new intersection candidate and determines the intersection point, if any.
2. If the intersection is on transparent geometry then the any-hit shader needs to determine what to do with the intersection. Otherwise, DXR will update the closest hit to be this new intersection and narrow the search.
3. Once all possible candidates are exhausted or if the search is ended prematurely, then there are two possibilities. Either the closest hit was updated at some point and the closest hit shader will run on that intersection or no hits were found in which case the miss shader is run.
4. Finally, DXR returns control to the shader that launched the ray and if that happens to be the ray generation shader then DXR is done with that pixel.

2.4 DLSS and TAA

We make use of both DLSS and TAA as integral parts of this project and thus we go into some more detail about what they are and how they can be used in this section. It is also noteworthy that DLSS can fulfill a similar role to TAA and seems to be intended as a replacement for it. However, we found it useful (if not necessary) to use both for our application to provide better results.

2.4.1 DLSS

DLSS stands for deep learning super sampling, and is intended to perform anti-aliasing by approximating a super sampled result from relatively few input samples. Additionally, it can perform image super resolution as part of its process, meaning that an input image can be upscaled to a higher resolution and sharpened to hopefully produce a good high resolution version of the input [28]. The way DLSS works is at least partially by utilizing a convolutional neural network in what is called a recurrent autoencoder. The autoencoder works by taking in a variety of input signals which it then encodes internally into a condensed signal form that is thereafter decoded to produce the full image. The fact that the autoencoder is described as recurrent is because it takes into account previous internal states when evaluating new

input [6]. The input required in the case of DLSS is a depth, color and motion vector input. It also requires the input to have subpixel jittering applied to it in order to vary the input signal over time for improved results.

Depth refers to the same depth values mentioned earlier, that is a value that describes how far away from the camera the geometry is. Color is simply the color value calculated through rendering the scene, and when using DLSS often one might often want to render at a lower resolution than intended for display to allow for upscaling. Subpixel jitter simply refers to slight smaller-than-a-pixel perturbations applied to the sample positions for rendering. This causes variation within the pixel from frame to frame and can be used for a cheap yet effective anti-aliasing technique called temporal anti-aliasing, which is probably very akin to what DLSS uses it for. Finally, motion vectors refer to vectors that describe how a sample position has moved in screen space between frames. That is to say, how the geometry that was sampled for the previous frame changed its position on the screen, either by the camera moving, the geometry itself moving or both.

The ultimate goal of DLSS is to allow for increased rendering speed due to reducing the render resolution and upscaling the result, as well as improving the rendered result by reducing aliasing.

2.4.2 TAA

Temporal anti-aliasing is a method of performing anti-aliasing that amortizes the computational cost over several frames, essentially applying the effect over time, hence the reason it has temporal in the name. The basic idea is that one introduces a slight randomization to the samples every frame, a technique usually referred to as jittering, and then one can average the contributions of each such history frame over time to attain a higher effective sample rate per pixel as explained by Yang et al. [31]. To achieve good results one ideally wants to consider the contributions of as many frames as possible, but since high resolution frames contain a lot of data it is often infeasible to store all but a fraction of them in memory. However, a common method to average the contributions which only requires storing one additional history frame is to use an iterative accumulation function as seen in equation 2.5 below.

$$f_n(p) = \alpha \cdot s_n(p) + (1 - \alpha) \cdot f_{n-1}(\pi(p)) \tag{2.5}$$

where $f_n(p)$ is the current frames color at pixel p for frame number n, α is the retention bias which decides the rate of accumulation, and $\pi(p)$ is the reprojection of pixel p to the history frame. In this case, reprojection refers to calculating the equivalent history frame pixel position of the current pixel which is necessary since any motion between frames would cause the history and current samples to become misaligned. To perform this reprojection we can calculate the screen space motion vectors of the motion between the frames and use that to offset our current pixel position. As explained by Yang et al. [31], it is common to set $\alpha = 0.1$ in equation 2.5 which results in a steady state effective sample rate of about 19 samples per pixel. However, when α is set too low it can introduce resampling errors that appear as ghosting and blurring of the image when in motion. For this project we reduce this using a heuristic based on the motion vectors and average frame rate which is discussed in greater detail in section 4.4. Other stability optimizations such as history rejection for reprojected pixels that have a depth difference to current depth values greater than some error tolerance, and using a box filter on the color buffers to determine a suitable α, can be used as well.

Chapter 3
Approach

In this following chapter we go over how we attempt to answer the research questions stated in section 1.2. We split the approach up into two parts, one concerning the ray tracer modes we use for our data gathering and one regarding how we evaluate the data once we have it.

3.1 Ray tracer modes

It is important we define exactly what we mean with each ray tracer mode to avoid confusion as to what they each do. We have already briefly mentioned the four modes we use in section 1.1 but we now describe them in much greater detail to hopefully minimize any ambiguity when presenting and discussing results in the later chapters.

All modes have in common that they use a default recursion depth of 1, meaning that besides the first rays generated, each ray can launch another level of rays. Furthermore, in all modes but the ground truth mode we use a default sample rate of 1 sample per pixel. Finally, all ray tracer modes across all attempts at data gathering are targeting a resolution of 1920×1080 which is a common resolution for desktop monitors.

3.1.1 The foveated modes

To start with we use a foveated rendering mode as the baseline for all experiments to provide a reference point to compare with. This mode uses the kernel log-polar foveated rendering technique discussed earlier and we set the kernel function to be the same as the one used by Meng et al. [19]. We specifically set $\alpha = 4$ as was found to be optimal in the user study they did. In this mode we also replace a disk around the foveal area with a uniformly sampled image to reduce the number of necessary rays which also means that even though we attempt to sample once per pixel in practice we use fewer samples in the foveated rendering mode.

The DLSS mode is then just an extension on this by simply enabling DLSS. In our case we have DLSS initialized with the `NVSDK_NGX_PerfQuality_Value_MaxQuality` quality

setting, and we use a custom downscale ratio of $\delta = 0.555$ which we also derive from the optimal parameters from the user study mentioned earlier [19]. We obtain specifically 0.555 from the fact that they determined from the study that a downscale ratio of $\frac{1}{1.8} \approx 0.555$ provides a good balance. This means that if we intend to target a resolution of 1920×1080 we actually render at $0.555 \cdot 1920 \times 1080 = 1065.6 \times 599.4$ which is then truncated to 1065×599 as the final render resolution, which we then let DLSS upscale to the target resolution. We use this DLSS mode to determine the impact of DLSS on the baseline foveated mode.

TAA can then finally be added on top of DLSS and that gives us the TAA mode and we use the same TAA settings independent of other factors. However, our TAA implementation uses a retention bias that varies across the screen, the details of which can be found in section 4.4. The impact of enabling our TAA implementation along with DLSS as compared to the foveated baseline, can be ascertained by using this TAA mode.

3.1.2 The ground truth

To provide a ground truth image to compare all test images with, we use a uniformly sampled ray tracer mode with a high sample rate that we are calling the ground truth mode. When using the ground truth mode we set the ray tracer to take 36 samples per pixel and depending on the scene we can also increase the recursion depth although by default it is still set to 1. All other effects such as DLSS and TAA are disabled and the uniformly sampled foveal disk is also disabled. To stabilize the image we also set any jitter offset to 0 such that each ground truth image samples the same screen space positions every time. Care is also taken to ensure that any stochastic behavior such as area light sampling, randomizes per sample such that an increased number of samples converges the result. All of this combines to provide an image that is fairly close to what is achievable with the ray tracing techniques we use without the impact of DLSS or TAA factored in, which is ideal for comparisons to our test images.

3.2 Evaluation strategy

We evaluate the ray tracer modes on three primary metrics in order to determine the suitability of applying DLSS to a foveated rendering scheme, and to determine how our proposed improvement to the results actually affect the render. The first such metric is the image quality for which we use an image difference evaluation, secondly we attempt to determine temporal stability by examining the flicker over time, and lastly we measure the render times to ascertain the impact on rendering cost.

3.2.1 Image difference metric

As mentioned, a part of our work involves determining the image quality and in order to achieve this we employ an image difference evaluation tool called ꟻLIP [4]. The focus of ꟻLIP is to provide an evaluation of the difference between a test image and a ground truth reference image. More specifically, ꟻLIP attempts to evaluate the difference a human would perceive if they were to alternate the test and reference images back and forth.

Using this evaluation method we can attempt to determine the impact on perceived difference when using the ray tracer in the foveated, DLSS, and TAA modes as compared to the ground truth mode. The change in the perceivable difference between ꟻLIP evaluations performed on each mode can give us an idea of the effect, but since ꟻLIP was never intended for use on foveated rendering there will always be an inherent difference from the uniformly sampled ground truth. That is why we use the foveated rendering mode as a baseline reference point to compare the DLSS and TAA mode with. It is not a perfect measure since it does not account for foveated rendering, but luckily ꟻLIP provides both a set of numeric condensations and a heatmap image of the per pixel evaluated difference. Thus, we try to use both of these information sources to better evaluate the actual perceivable impact of the ray tracer modes. However, this means that it is hard to say exactly what perceivable difference can be expected without an evaluation method geared towards foveated rendering specifically or as is the more common approach, performing a user study. However, ꟻLIP should at least provide an upper bound of the effect on image quality that can still be useful.

3.2.2 Flicker evaluation

An important aspect of any real-time rendering, but especially foveated real-time rendering, is the temporal stability of the rendered images. It is usually not desirable to have flicker, ghosting, or other temporal artifacts present in real-time graphics applications but when using foveated rendering these effects can be exaggerated in the peripheral areas due to the inherent subsampling that occurs in those areas. On top of that, temporal artifacts in the peripheral region can often be very distracting and are often even more noticeable than those in focus, likely due to the fact that the temporal resolution of the human eye tends to be highest at around $30^{o} - 60^{o}$ of eccentricity from the foveal point [24].

In order to determine the temporal stability of the various modes we simply examine the noticeable artifacts in video recordings of the application running. This will not provide a precise numeric value for the temporal stability but with how big the temporal stability changes are in our application it is still possible to gain useful insights from this method. Ideally, we would want a temporal stability evaluation tool similar to ꟻLIP but accounting for foveated rendering, however it was only recently that a unified measurement of the temporal resolution and spatial frequency of the human eye was proposed which would be required for such an evaluation tool [13].

3.2.3 Render times

In order to determine the render time savings we can obtain by applying foveated rendering in tandem with upscaling using DLSS we measure the render times of some of the ray tracer rendering stages. The ray tracer has three interesting stages that we need to time for this purpose, and those are the ray dispatching, the kernel log-polar remapping, and the DLSS stages of the rendering pipeline. These together account for the render time that is specifically caused by our implementation of ray tracing, foveated rendering and our DLSS usage. Also, by specifically looking at the ray dispatch time we can determine the savings in just ray computations. Further, by looking at the kernel log-polar remapping we can see the cost of calculating the remapped coordinates, and also the cost of blurring when foveated rendering is used and any cost that might be associated with our TAA implementation. Finally, by

considering DLSS in isolation we can determine what the render time trade off is between DLSS and the rest of the considered pipeline stages.

Chapter 4

Implementation

An overview of the four most important processing stages of the rendering process can be seen in figure 4.1, where we have the ray tracing itself, the remapping and blurring of the foveated buffer, a TAA effect that applies to varying degrees across the screen, and finally the DLSS invocation. The process shown in the figure can be summarized as first dispatching rays to gather information about the viewable scene, then remapping the resulting information to rectilinear coordinates and blurring where appropriate. If we have enabled TAA then that would be applied next and finally after all of that, if we are using DLSS then the final step is to pass everything along to DLSS to produce the final image. You will notice that DLSS mode skips the TAA step and foveated mode skips both DLSS and TAA. Only the TAA mode goes through all four steps.

In this chapter we will go into more detail about the techniques used in this process. We will cover how the ray color payload is calculated, how texture level-of-detail (LOD) is calculated, and how supersampling for the ground truth tracer mode is done. We will also cover the various steps involved in generating rays at kernel log-polar coordinates and how the results are interpreted to produce the final foveated image out of those rays. We also cover how the TAA is implemented in detail and what techniques we use for that. Finally, we will discuss the DLSS post-processing layer and the inputs it receives.

4.1 The core ray tracing

For this project we implement a Whitted style of ray tracing [29] which only accounts for basic reflection, refraction, shadows and diffuse color. The rays also need to be able to handle transparency and variable sampling rates, all of which will be covered in greater detail in this section.

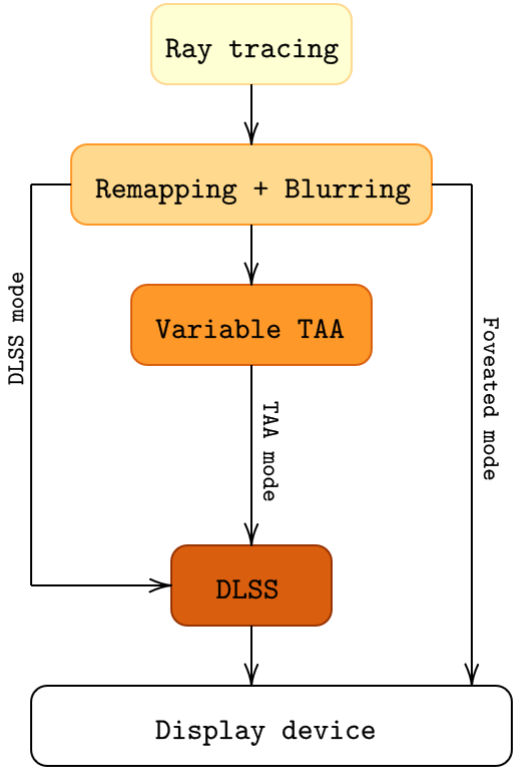


Figure 4.1: An overview of the important steps of the rendering process that we use.

4.1.1 Color rays

The color of a ray is determined by a multiplicity of factors. The first recursion level of a ray is imparted an ambient color term based on what it hits and a constant multiplier. Furthermore, whenever the ray hits a textured surface, a texture sample is acquired and used to determine the baseline diffuse color which is used for lighting the surface with various scene lights. To calculate the lighting we launch three shadow rays at a slight offset from the surface in the normal direction of the surface. Each shadow ray direction is randomly sampled from a cone using blue noise to provide improved sample coverage of the cone. Using blue noise as a method of generating soft shadows with ray tracing is a good approach when the number of samples are limited, as is covered by Wolfe [30]. A shadow ray determines if the light source in question is in direct line of sight or not by considering any hit with opaque geometry to be blocking the light ray. Once all three samples have been taken we can then average the contribution to get shadows with soft edges, but to reduce the noisiness of the shadow we still need to employ something like TAA or DLSS. If a shadow ray does not detect any hits we can allow the light to contribute color to the surface and in doing so we also apply a diffuse and a specular shading factor inspired by the respective terms from classic Phong shading [22]. Therefore, we end up with a mostly Phong shaded look to our materials when reflection and refraction is not present.

All these calculations are done in a closest hit shader meaning that they only apply for whichever surface intersection ends up closest to the camera. However, if a ray misses the

Figure 4.2: Image demonstrating how the transparency of the Sponza [21] plants look.

geometry of the scene, then the miss shader will trigger and the color payload becomes a gray color by default. We will call the color resulting from these calculations, the hit color for the rest of this section.

In order to handle transparency with the DXR pipeline we use an any-hit shader to calculate colors and store them in a K-buffer. The specific method used is the one presented by Krüger et al. [14] in their paper on tracing with transparency. In essence we store colors and their transparency in a buffer with K elements as intersections are found along the ray. The elements are stored in depth order until the buffer is full meaning that intersections further from the camera end up further back in the buffer. So long the buffer is not filled yet we ignore hits in the any-hit shader, meaning that the ray tracing max distance that is used by DXR to narrow the search for the closest hit is never updated. However, once the buffer is filled and further intersections are found, we start narrowing the search by updating the max distance. Further, if new intersections are found when the buffer is full we combine two elements of the buffer before inserting the color of the new intersection. This process ensures that the information stored in the buffer relates to the K closest intersections that were found. Finally, once the closest intersection along the ray has been found, the closest hit shader is run, calculating the hit color and that then gets combined with the transmitted color of each element in the K-buffer.

A note about this system for rendering transparency is that it was really not necessary for our purposes. It is useful when you have complex transparent objects with varying degrees of transparency and you need the colors to blend correctly, but we never render such objects. The only transparent objects present in the scenes we use are the plants from Sponza [21] (see figure 4.2) which all have textures with areas that are either fully opaque of fully transparent. For those plants it would suffice to just ignore ray hits that land on transparent areas, regardless of how transparent they are.

4.1.2 Texure level-of-detail

In this section texture level-of-detail (LOD) refers to the anti-aliasing technique of sampling textures at lower resolutions as more texels (texture pixels) map to a pixel on screen. When using rasterization, it is fairly straightforward to determine what LOD to use. You could, for

example, calculate the partial derivatives of the texture space coordinates with regard to the screen space coordinates and use those values to estimate the mapping from screen space to texture space [25]. Using the depth buffer calculated for rasterization and leveraging the fact that the geometry is made up of planar surfaces makes the partial derivatives mostly trivial to calculate. However, when ray tracing, the problem has to be solved differently since no depth buffer is supplied by default when doing the texture sampling. One such solution is to calculate cones traced through the pixel, along the ray direction as Akenine-Möller et al. suggest [3]. We implemented a similar solution to the one suggested, however we do not take reflection LOD into account and we also do not include the approximated triangle LOD term in the final LOD level. This is because reflections are not very prevalent in our test scenes and we use a different base bias than the approximated triangle LOD term with foveated rendering and DLSS. Essentially we calculate the cone width at the point of intersection and weight it by the intersected surface angle to the ray. This value is then used to calculate the final LOD level as

$$L = b + \log_2(\alpha \|d\| \frac{1}{|d \cdot n|}) \tag{4.1}$$

where d is the distance to the intersection point from the camera, α is the largest angle of a cone that is launched from the camera through a single pixel, and n is the normal of the triangle that was hit by the ray. Here, b is a LOD bias which is calculated as in equation 4.2 below.

$$b = 0.3 + \delta + \phi \tag{4.2}$$

Where δ is a DLSS contingent bias and ϕ is a bias introduced to counter texture aliasing in the periphery when using foveated sampling. In the case of δ we follow the Nvidia recommendation to introduce a negative LOD bias to produce sharper upscaled images. We do not however, use this negative bias if foveated sampling is in use as that introduces severe aliasing into the foveated result before it is upscaled which remains clearly visible after upscaling. The ϕ term in equation 4.2 is calculated as

$$\phi = 3 * S_1(u^{k_\alpha})^3 \tag{4.3}$$

wherein the $S_1(x)$ function is the smoothstep function in HLSL which is calculated as a cubic Hermite interpolation of the input clamped to the range $[0, 1]$. Further, the u is the normalized x screen space coordinate of the pixel being considered and k_α is the kernel functions alpha parameter.

The effect of the ϕ LOD bias is that the texture resolution goes down as u increases, which means that the resolution is lower closer towards the periphery and the bias evaluates to 0 at the foveal point.

4.1.3 Supersampling

The ground truth ray tracer mode relies on supersampling to produce the ground truth image. To achieve this, we have implemented a regular grid pattern supersampling scheme. The idea is simple; sample each pixel in a $N \times N$ grid and average out the color results to produce the final sample. The supersampling happens during ray generation and happens before any

kernel log-polar transformations are applied to the sample positions. That is to say that the grid will be distorted when foveated rendering is used.

4.2 Foveated ray sampling

The foveated ray tracer mode uses kernel log-polar space (see chapter 2.2) to determine the positions which the rays are launched towards. In the following section we will describe how this is implemented in more detail and the steps involved in generating a foveated image result. We will first go over the process of calculating the sampling positions and then the process of mapping the log-polar buffer to Cartesian coordinates, including how we blur the periphery. Finally we will cover how we ensure that an area at the foveal point is sampled at one sample per pixel.

4.2.1 Calculating sample positions

In order to calculate the direction of a ray we have to take the pixel position and camera view into account. To do this we use the ray dispatch coordinates as obtained by the `RayDispatchIndex()` function which provides the coordinate of the pixel that we are currently considering for ray tracing. We can then remap the coordinates to the range $[-1, 1]$ with the point $(0, 0)$ being the center of the screen. The coordinate at this point represents the relative offset for the ray. Then by applying the camera view matrix we can calculate a direction that aligns with the pixel coordinate that is to be traced.

When we wish to calculate ray directions that align with a foveated sampling pattern, we perform a very similar process. In such cases we treat the ray dispatch coordinates as kernel log-polar coordinates and do an inverse transform to Cartesian screen space according to equation 2.3. Once we have the transformed coordinates we can then calculate the ray direction as above and launch the ray towards that direction instead. We are essentially sampling the kernel log-polar space uniformly which means our samples end up warped such that the sample density is higher close to the foveal point and gets progressively lower towards the periphery. The result of the traced ray is then stored in an output buffer at the original, non-transformed, Cartesian screen space coordinates. This means that the foveated rendering output is a warped log-polar like image of the scene as illustrated in figure 4.3.

The kernel log-polar transformation includes a predefined kernel function which in our implementation is $K(x) = x^{\alpha}$, where α is a manually set parameter. This is the same function structure that Meng et al. [19] used in their user study, and therefore lets us decide on parameters that are somewhat experimentally supported. Specifically, using a user study they determine that $\alpha = 4$ when downscaling the kernel log-polar buffer by a factor of 1.8 is a good balance since it provides the best performance boost whilst still providing no perceived loss in visual quality 80% of the time. This does not mean that these are truly optimal parameters but they represent a good starting point with some experimental backing, hence we use these values for our parameters as well.

(a) **(b)**

Figure 4.3: When we apply the kernel log-polar transformation the output color buffer looks like **(a)** when the same scene and camera setup would normally produce a color buffer as in **(b)**.

4.2.2 Transforming back to Cartesian coordiantes

Since the immediate output of foveated rendering is in kernel log-polar space, we need to have a post ray tracing step that remaps the output back to Cartesian coordinates. In this step we first start with a buffer index that we want to write the remapped color value to, and in order to get the remapped value we transform the buffer index to kernel log-polar coordinates using equation 2.2. Using the obtained log-polar index we can then sample the foveated buffer and store the results at the original buffer index, thus mapping the foveated buffer values back to Cartesian coordinates.

However, a problem occurs when remapping the colors close to the foveal point or close to the periphery; each Cartesian space pixel will map to either many log-polar space pixels (when close to the foveal point) or less than one log-polar space pixel (when close to the periphery). The results are that there will be instability and aliasing issues near these areas that have to be dealt with. Therefore, we also apply a blur when sampling the foveated buffer for remapping by applying a radial Gaussian filter with a variable diameter, using a similar method as Meng et al. [19]. The diameter $\eta(x)$ is in our case determined by

$$\eta(x) = max(3 + 0.8\frac{x - 0.1}{0.05},\ 0) \tag{4.4}$$

where x is the normalized distance from the buffer index to the foveal point. This provides a blur diameter that grows quickly as the buffer index gets closer to the peripheral areas. With this, when the resulting remapped color buffer is generated, the results are an image that is anti-aliased at the foveal point (similarly to when supersampling is used) and blurred towards the periphery which reduces distracting visual artifacts.

4.2.3 Foveal area sampling

In order to obtain a consistent performance improvement when using foveated rendering we can ensure that the pixels in the foveal area are rendered at one sample per pixel. With this method we should ideally end up with an image that is indistinguishable from a non-foveated image similarly sampled at one sample per pixel. We implement this by skipping a disk at the fovea when tracing rays with foveation enabled and later using a separate ray dispatch

call to uniformly sample across the disk and trace the rays to fill it in. When we do this we have to also fade out the blur such that no blurring occurs inside the disk which would be detrimental for the disk image. We arbitrarily decide the radius of the disk and set it to be 15% of the distance to the furthest corner from the foveal point. However, having the radius depend on the furthest distance to any of the corners is not ideal as when the foveal point deviates from the center of the screen the radius will grow. This is not an issue in our case since we always set the foveal point to the center of the screen but any future improvements should include fixing this.

A disk at the foveal point in Cartesian screen space is equivalent to a rectangular area in kernel log-polar space. This rectangular area will have opposing corners in $(0, 0)$ and $(l(r), h)$ where h is the height of the kernel log-polar buffer, r is the radius of the disk, and $l(r)$ is given by

$$l(r) = \left\lfloor \mathbf{K}^{-1}\left(\frac{log(r)}{L}\right) \cdot w \right\rfloor \tag{4.5}$$

where $\mathbf{K}^{-1}$ is the inverse kernel function, w is the width of the kernel log-polar buffer, and L is the same one discussed in subsection 2.2.2, as in it is the logarithm of the distance from the foveal point to the furthest corner of the screen. What we are calculating in equation 4.5 is the kernel log-polar u-coordinate where the circle of radius r lies.

When we have the bounds of the rectangle in kernel log-polar space all we have to do is skip tracing the rays that originate from pixels inside those bounds, since we interpret the pixel coordinates as kernel log-polar coordinates when foveation is enabled. Then in the subsequent ray dispatch call we ensure that pixel coordinates are interpreted as they normally are in Cartesian screen space but now only trace rays that fall within r pixels of the foveal point. The colors from each separate ray dispatch call are stored in separate buffers to avoid overwriting colors and then stitched together by the compute shader that also handles remapping of the kernel log-polar buffer to produce the final result.

4.3 DLSS layer

One of the main components of our ray tracer is the DLSS post-processing layer and it runs right after all buffers have been mapped to Cartesian coordinates. Any blurring and TAA that needed to be done would also have happened by the time DLSS is run. As discussed in section 2.4.1, DLSS requires additional inputs other than the color buffer and we cover how those are generated and processed in this section. We also discuss the subpixel jittering, where and how it is applied and how the offsets are generated.

4.3.1 Depth and motion vectors

The simplest of the additional inputs to generate is the depth buffer. When using DXR, rays will by default keep track of the distance along the ray they have traveled which is what we use to determine the depth. DLSS expects the depth in the range $[0, 1]$ with 0 being nearest to the camera and 1 being the furthest distance. By remapping the distance along the ray to a value between 0 and 1 with a moderate choice of furthest distance, we can determine how far the ray traveled into the scene. This is not exactly the same as the rasterization based depth

buffer which relies on the near and far plane when determining the values of the depth but it still seems to allow for DLSS to operate correctly.

The motion vectors require a more involved process to calculate and are an active research area in and of themselves. The problem with accurate motion vectors is that colors on the screen often do not move in the same way as objects in the scene. An example highlighted by Hanika et al. [11] is that when a glass surface moves the highlights and backdrop of the glass do not move in the same direction. This sometimes means it is very hard or even impossible to obtain accurate motion vectors. Another issue is with motion vectors is handling reflections as those also move differently to the objects in the scene and have to be tracked separately as is done by Zheng et al. [33]. We do not handle these issues in our motion vector calculations as they are beyond the scope of this project and would take too much time. Instead our solution for this is a simple and limited motion vector calculation which relies on a world position buffer from the previous frame. The way we do it is by recording the world position of the ray intersection for each pixel and then we project that position information the next frame to determine which pixel each recorded world position maps to. Thus by determining how the pixel position changed between frames, we can derive a delta in screen space which tells us how objects have moved. The projection of the world position only uses the view-projection matrix derived from the camera properties. In more concrete terms, this means that if we let p be the current pixel being considered for a frame, W be the world position 2D buffer with the same dimensions as the render resolution, and f be the view-projection transform function, we have

$$m_p = p - f(W[p]) \tag{4.6}$$

where m_p refers to the motion vector for the current pixel p and $W[p]$ is the world position at the 2D index p from the world position buffer. From equation 4.6 we can see that the motion vectors describe the opposite direction of movement in screen space which is correct in terms of what DLSS expects. Additionally, it should be noted that this method does not detect any motion vectors for moving objects in the scene since it only takes the camera's view-projection transform into account thus only detecting motion due to camera movement. However, this is not an issue for our test scenes since we do not make use of moving objects.

Both the depth and motion vectors are calculated during ray tracing in the ray generation shader. They are then processed by the compute shader responsible for remapping from kernel log-polar coordinates to Cartesian coordinates, during which neither depth nor motion vectors get blurred (as opposed to the color buffer) in order to preserve the information better for use by DLSS. However, tests with and without blurring these additional buffers have shown no easily distinguishable differences.

4.3.2 Subpixel jitter

When we use DLSS for rendering we have to apply subpixel jitter to the color buffer we supply DLSS with. We apply the jitter offsets according to a Halton sequence with a base of 2 and 3 for the x and y axis respectively. We use a phase of $16\left\lceil \frac{W_t^2}{W_d^2} \right\rceil$, where W_t is the target display resolution width and W_d is the render resolution width. This means that the number of elements in the section of the Halton sequence that we repeat will grow proportionally

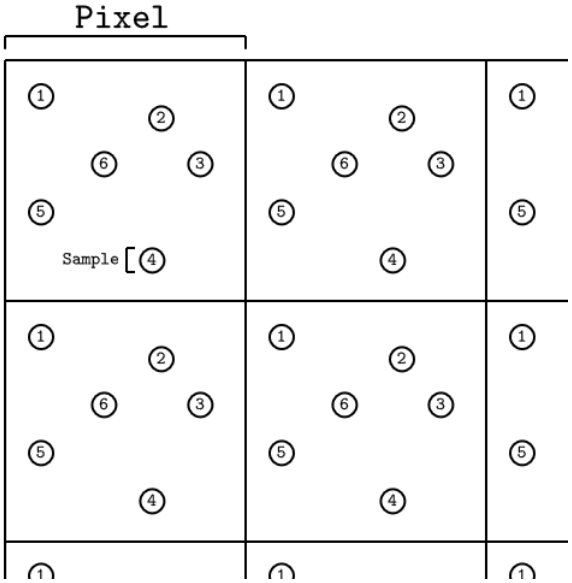


Figure 4.4: An example sequence of subpixel jitter applied to samples over 6 frames. Each numbered circle here corresponds to a sample position and the number indicates for which frame this sample position is used.

to the square of this factor. The Halton offset is then remapped from the range $[0, 1]$ to the range $[-0.5, 0.5]$ and used as an offset when rendering. An illustration of a possible sequence of sample positions when using subpixel jitter is displayed in figure 4.4; note that each pixel has the same offset each frame. This offset is applied during the ray generation by simply offsetting the dispatch index of the ray, and that produces a stable image when not using foveated rendering in combination with DLSS.

However, when we do use foveated rendering and DLSS simultaneously we get heavy flickering towards the periphery for unknown reasons. Initially we thought it might be caused by the kernel log-polar transforms distorting the jitter offset to be larger than it should. But when we use our TAA implementation which also utilizes subpixel jitter and apply that instead of DLSS, in combination with foveated rendering we do not experience such flickering. This seems to indicate that this problem does not stem from the subpixel jitters themselves but instead comes from how DLSS makes use of the jitter when presented with a foveated input image. This poses a big problem for using DLSS with foveated rendering, but to circumvent this and allow DLSS to be used with minimal flickering we apply TAA to the color input before it is passed along to DLSS. This has the effect of eliminating the jitter offset from the color buffer as the offset color sample is worked into the TAA results. The benefit of this is that flicker from DLSS is reduced but it also means that DLSS will perform worse since it no longer has a variety of sample offsets to draw from. As this would reduce the visual quality of the whole image, we decided to leverage the fact that we only perform the kernel log-polar transformations in the peripheral regions. This means that we only need to apply TAA as a stability measure in the periphery. Additionally, since the periphery is blurred it is hard to make out any loss in visual quality due to DLSS lacking jittered samples. Thus when we are using what we have previously called the TAA ray tracer mode we only actually apply TAA and DLSS outside the foveal area disk, and the disk itself only has DLSS applied.

4.3.3 DLSS optimal resolution

Built in to DLSS are helper functions to query the optimal render resolution for a specific targeted display resolution and quality. We mainly make use of these settings by querying DLSS for the optimal render resolution whenever the display resolution changes, setting the render resolution accordingly. However, DLSS does not put any restrictions on render resolution and if so desired one could use the target display resolution to render which essentially disables the super resolution feature and thus provides no render time speedup. That could be suitable if DLSS is only to be used for anti-aliasing, but generally we want to be able to vary the render resolution as needed. Therefore we additionally implement a setting that allows free scaling of the render resolution when using DLSS. We can then set the downscale factor to be 1.8 which should provide balanced results as mentioned in subsection 3.1.1. The downscale factor we use is actually the same that is applied when you query DLSS for an optimal resolution using the balanced quality settings.

4.4 Temporal anti-aliasing

We implement a version of TAA for this project that is primarily based on information gained from the survey performed by Yang et al. [31], in which they outline a set of problems and proposed solutions that come with TAA implementations. We decided to go with the common approach of accumulating an average color according to equation 2.5, as it is simple to implement and provides good performance. However, as we mentioned in section 2.4.2, there are issues that come with having α from equation 2.5 set close to 0 in some cases. To remedy this we use a heuristic to increase the α when needed thus reducing any potential ghosting artifacts we may get from having it set too low. For the remainder of this section we will be referring to this α value as the *retention bias*, referring to the fact that α decides how much of the history buffer information is retained between frames. Our heuristic is based on the motion vector magnitude $\|v_m(p)\|_2$ per pixel p and the average frame rate R_n for frame number n calculated as a moving average as in equation 4.7 below.

$$R_n = 0.1 \cdot \frac{1}{\Delta t_n} + 0.9 \cdot \frac{1}{\Delta t_{n-1}} \tag{4.7}$$

where Δt_n is the time it took to process frame n. Using the values of R_n and $v_m(p)$ we obtain our heuristic value $H(v_m(p), R_n)$ from equation 4.8 as follows.

$$H(v_m(p), R_n) = \left(\frac{\|v_m(p)\|_2}{\|(w,h)\|_2} \right)^{\frac{1}{4}} \cdot \frac{60}{R_n} \tag{4.8}$$

Where w is the width and h is the height of the render resolution. Using $H(v_m(p), R_n)$ we get a value that monotonously grows as motion increases and shrinks as the frame rate increases, and ends up taking on values in a reasonable range close to $[0, 1]$ for motion and frame rates that we can expect to see. Next, we have to apply the heuristic and in order to do so we first calculate a minimum retention bias α_{min} for our current frame n by applying a box filter to the error estimate for each pixel p between the current frame color c_n and the history buffer color c_{n-1}. This method of finding a minimum starting retention bias is the same as the one proposed by Yang et al. [32] and is intended to find a retention bias based on

how different the current frame is from the last. A higher difference means that we should retain less information from previous frames since we want the final image to represent the current frame accurately. The box filter $B_{3\times3}$ is a 3×3 filter that computes the average value in the neighborhood of p to obtain a stabilized error estimate $\varepsilon(p)$ according to

$$\varepsilon(p) = B_{3\times3} * (c_n - c_{n-1})(p) \tag{4.9}$$

and then we set the minimum $\alpha_{min} = max(0.1, min(\varepsilon(p), 1))$, or in other words $\varepsilon(p)$ clamped to the range $[0.1, 1]$. With both α_{min} and $H(v_m, R_n)$ computed we can finally determine the retention bias α as

$$\alpha = \alpha_{min} + max(min((1 - \alpha_{min}) \cdot H(v_m(p), R_n),\ (1 - \alpha_{min})),\ 0) \tag{4.10}$$

which means that we have $alpha \in [0.1, 1]$ with higher values when temporal artifacts are expected to be more prominent.

The TAA techniques described so far are all implemented in the remapping compute shader that is invoked as soon as all ray tracing has finished. The results from this algorithm is an image with reduced visual artifacts while remaining anti-aliased. This image can then be sent on as input to DLSS assuming the algorithm is constrained to run outside the foveal disk as described in the previous section.

Finally, as for how we arrived at this specific heuristic, we started with our knowledge of when you can expect to see artifacts due to a low retention bias. We then tried to factor those things in to the final heuristic value while keeping the values from getting too large. We then adjusted the constants in equation 4.8 to provide a balance between reduced artifacts and anti-aliasing performance. After a few iterations of adjusting and checking the results we arrived at the specific heuristic that was explained in this section.

Chapter 5
Results

In this chapter we present and discuss the results of our ꟻLIP image evaluation, render time measurements, and the temporal stability evaluation, in that order. We evaluated the ray tracer in general by the methods described in 3.2, but in some cases there are slight deviations or other details that are important which we make sure to highlight. The graphics hardware we used to obtain all the results below is an Nvidia RTX 2080 Ti GPU which we had available. This graphics card has the prerequisite dedicated ray tracing hardware that we require to use DXR to its fullest extents. Following this chapter we conclude with remarks on these results and other future work that could be done.

5.1 ꟻLIP still image comparisons

In order to evaluate the image quality of the ray tracer modes, we chose to use ꟻLIP [4] as an evaluation tool. We present the results separated based on which mode and further, which scene the results pertain to; starting with the baseline foveated mode, then moving on to the DLSS mode, and finally we present the TAA mode results. The results are showcased through images produced for and by the ꟻLIP evaluation, and we also use the weighted median that ꟻLIP provides as a condensed measure of the evaluated difference. The weighted median in this case is calculated by using the median of the weighted histogram representation of the ꟻLIP errors, which itself is produced by distributing the pixels into buckets based on their error value to form a classic histogram representation and then multiplying each bucket by its own median ꟻLIP error. This means that larger errors count for more than smaller ones and should provide a more intuitive measurement of the overall error value. Also, for reference we get a ꟻLIP weighted median difference in the range 0.03-0.05 when comparing a uniformly sampled, temporally anti-aliased image with ground truth. So we will consider this as a low difference since it represents the minimum error we could achieve without foveated rendering. Finally, for the heatmaps that ꟻLIP produces we have that brighter colors indicate larger differences and black indicates no perceivable difference.

In the following subsections we look at the results for the baseline foveated mode, the DLSS mode, and the TAA mode; looking at the scenes Sponza [21], the Cornell box [18], and the Sun temple [8] for each mode. The scenes are commonly used in graphics research, but there are significantly more complex scenes available. However, we do not use those since we did not have the time resources to spend implementing support for the more complex scenes. We also discuss the results themselves, any important details, and what we are trying to test with the different camera and scene setups.

All figures referred to in this section can be found in Appendix A.

5.1.1 Baseline foveated mode

Here we present the baseline foveated ray tracer mode results when compared to the ground truth mode with the ꟻLIP tool. We go over Sponza, then the Cornell box, and finally the Sun temple in that order.

Table 5.1: The weighted median ꟻLIP error for each view of the Sponza and Sun temple scenes when using the baseline foveated mode.

View	ꟻLIP error
Sponza Central	0.228
Sponza Overview	0.187
Sponza Plant	0.189
Sun Temple Statue	0.239
Sun Temple Chamber	0.272

Sponza

For Sponza we chose three camera and scene setups to try and cover a variety of conditions. We use a central position to showcase a balanced view at medium distance from most geometry, we then use a overview position to capture the effect of distant geometry and this view also includes a high contrast area which lights up on the ꟻLIP error heatmap. Finally, we use a close up of one of the plants in Sponza to investigate the effects of high frequency input. The resulting images can be seen in figure A.1, in which we use the foveated ray tracer mode with the foveal point set to the center of the screen. The ꟻLIP error is quite high across the board for these images as indicated by the brighter colors. For a uniformly sampled image most areas would have been colored closer to black in the resulting ꟻLIP heatmap. This high ꟻLIP error is expected since the foveated render is both subsampled and blurred in the periphery which creates an error offset that is not accounted for by ꟻLIP . However, a circular area in the center of the ꟻLIP error heatmap is much darker than most of the rest due to the uniform foveal area disk sampling that takes place there instead.

As for the weighted median values that we mentioned earlier, they can be found in table 5.1 for this baseline measurement.

Cornell box

The Cornell box is a simple scene that we mainly use to showcase refractions and reflections clearly. Some reflections are present in the Sponza scene as well but could be harder to make out than in the Cornell box. For this scene we use only a single view that looks straight into the box with two material spheres at a medium distance. One of the spheres is quite reflective and the other is less reflective but is also refractive. For this scene specifically we use a recursion depth of 3 for the test image and 4 for the ground truth, since that better approximates the interactions of light with the spheres and lets us more clearly see the effects of reflection and refraction. The images of the results can be seen in figure A.2, which show that the difference between a recursion depth of 3 and 4 is quite noticeable for the refractive sphere. We see this because the refractive sphere on the right side of the image lights up bright in the ꟻLIP heatmap. We can also see that the stochastic soft edges around the shadows become a hotspot for errors as the ground truth converges closer to the true shadow edge by taking many more random samples. This ꟻLIP comparison produced a weighted median of **0.133** which is lower than the rest of the other values observed but can be attributed to how simple the Cornell box is.

Sun temple

For the Sun temple we use two views for our tests, each intended to investigate a different aspect of the lighting. One is of the main statue to display the stretched out shadows of a low hanging directional light that shines in from outside. This will cause the soft edges of the shadows to spread out, introducing significant noise to the image. The second view is of the inner chamber statue, which is lit by two local spherical lights, one that is relatively dim and orange in color, and one that is brighter and blueish white in color. Additionally there is some minor contributions from the directional light that shines in from the hallway that leads to the main statue. This is intended to test many different light interactions at various scales, with one view showing shadows at a low angle and the other showing multiple lights in a single view. The exact positions and angles from which we capture the views are somewhat arbitrary, we just try to ensure that most of the central objects and lights are visible. In figure A.3 we can see, same as with the Sponza scene, that a darker region in the center of each ꟻLIP heatmap corresponds to the foveal area disk although it is not as clear, likely due to the increased complexity of the lighting, geometry, and some textures introducing more aliasing and shadow sampling errors. We also clearly see a higher weighted median for this scene in table 5.1, indicating that these images indeed contain more discernible errors than images from the other two scenes.

5.1.2 DLSS mode

Here we show the results in each scene similarly as in the previous subsection but instead using the DLSS ray tracer mode. We will focus on evaluating the perceivable loss in visual quality for these results by comparing them to the results of the baseline foveated mode. All images that where taken in the DLSS mode have a small rectangle in the lower right corner that covers up the watermark DLSS applies to the images. This way we avoid additional ꟻLIP errors when comparing the watermarked DLSS test images and the watermark free

ground truth images.

Table 5.2: The weighted median ꟻLIP error for each view of the Sponza and Sun temple scenes when using the DLSS mode. As expected, we seem to observe a higher error across all the viewpoints.

View	ꟻLIP error
Sponza Central	0.263
Sponza Overview	0.221
Sponza Plant	0.200
Sun Temple Statue	0.281
Sun Temple Chamber	0.304

Sponza

Performing the same ꟻLIP comparison as with the baseline foveated mode but now using the DLSS mode instead we see a reduction in visual quality. The loss in visual quality can be seen in figure A.4 when compared with the images from figure A.1, especially looking at the ꟻLIP heatmap we see a noticeable increase in brightness in many areas. There is also a loss in quality in the foveal area disk due to the fact that it is now upscaled by DLSS from almost half the render resolution. The loss in the foveal disk is fairly small however and not very noticeable, at least from the viewpoint we observed. We also predictably see an increase in the weighted medians listed in table 5.2, but the increase differs significantly per viewpoint with the largest difference of 0.035 and 0.034 in the central and overview viewpoints and the smallest difference of 0.011 in the plant viewpoint. This might indicate that we can expect less loss in visual quality when viewing the scene close up, likely due to the reduced aliasing from subpixel features.

Cornell box

Similarly to Sponza we see an increased loss of visual quality with the Cornell box. In figure A.5 we can see the resulting images and while it is hard to spot the difference, we can see that areas that were relatively bright in the ꟻLIP heatmap from the baseline tests are now brighter. The weighted median for this test is 0.142 which is an increase over the baseline by 0.009, similar to the plant viewpoint of Sponza. The reason we see a fairly small difference might be attributable to the simplicity of the scene which is largely free of high frequency features and therefore do not cause perceptible differences when supersampling. To support that idea, we point out that in the baseline we only saw hotspots in ꟻLIP heatmap where there were edge transitions, shadow noise or due to an additional layer of refracted color contribution from the higher ground truth recursion depth. Those same hotspots are now brighter in the heatmap while everything else remains dark, indicating that there is not much room for the image to become distinctly more different than ground truth.

Sun temple

Following the same pattern we also see an increase in loss of visual quality for the Sun temple scene. From table 5.2 we see that the difference in the weighted median ꟻLIP error is 0.042 for the statue view and 0.032 for the inner chamber view. The differences seem to roughly match the difference observed in the Sponza central and overview viewpoints, which makes sense as these views involve a fair amount of detailed, relatively high frequency features. Figure A.6 showcases the resulting images from these tests and we see yet again, an increase in brightness across multiple areas in the heatmap while the foveal area disk remains relatively unaffected. Interestingly, we also see ꟻLIP error hotspots in areas with bright lights, likely due to the increased contrast in these areas which might lead to an increase in perceived spatial frequency.

5.1.3 TAA mode

This final part of the ꟻLIP comparisons cover the TAA mode results. For this part we compare the ꟻLIP errors to the previous two modes and attempt to show that there is little additional perceived visual quality lost when using the TAA mode as compared to the DLSS mode alone.

Table 5.3: The weighted median ꟻLIP error for each view of the Sponza and Sun temple scenes when using the TAA mode. In this case we see very little difference from the DLSS mode results.

View	ꟻLIP error
Sponza Central	0.262
Sponza Overview	0.220
Sponza Plant	0.200
Sun Temple Statue	0.280
Sun Temple Chamber	0.304

Sponza

The Sponza results in the TAA mode show very little difference in perceptible visual quality loss as compared to the DLSS mode. In fact, manually examining the ꟻLIP heatmaps in figure A.7 and comparing them to the heatmaps from figure A.4, we see almost no difference. The weighted median error values listed in table 5.3 confirms that there indeed seems to be little change in the visual error compared to ground truth. This indicates that the visual quality of each frame in isolation is similar between the TAA mode and DLSS mode.

Cornell box

Using the TAA mode we observe a weighted median ꟻLIP error of 0.139 for the Cornell box which is comparable to previous results in the baseline and DLSS modes. In figure A.8 we see that the ꟻLIP error heatmap has lost a pattern of hotspots on the sidewalls and on the roof when comparing to the DLSS mode. Indeed, by comparing the foveated test images we

can also clearly see some form of visual artifact at those same areas in the DLSS mode image, which is not present in the TAA mode image. This could be a result of the fact that in TAA mode we essentially eliminate the subpixel jitter in the peripheral regions of the DLSS color input. DLSS might be interpreting the peripheral jitter incorrectly when used with foveated rendering on its own.

Sun temple

Finally, the FLIP comparisons run in Sun temple with the TAA mode show the same pattern of results as in the other scenes, with very litte change from the DLSS mode results. In figure A.9 we see that the FLIP error heatmap is very similar to the ones produced with DLSS mode and again in table 5.3 we see that the weighted median error is almost the same as observed for the DLSS mode, further confirming the similarity in visual quality of the isolated frames. In all, we propose that DLSS has some effect on the visual quality depending on the view as we observed increases in weighted median FLIP error ranging from 0.009 to 0.042. We further propose that adding TAA with varying parameters across screen space before invoking DLSS results in very similar visual quality to using only DLSS. This is supported by the lack of any significant change in weighted median error between the DLSS and TAA modes. This should mean that when using DLSS with foveated rendering we can also use our TAA method without incurring any additional visual cost.

5.2 Render time measurements

In this section we present the render times we measured across the baseline foveated mode, DLSS mode, and TAA mode. We measured how long the GPU took to compute each of the ray tracing stage, the remapping compute shader stage, and the DLSS stage. As discussed in section 3.2, we define each stage as follows; the ray tracing stage consists of the ray dispatch calls which performs the ray intersection calculations and outputs all necessary data for the subsequent stages; the remapping compute shader stage includes the dispatch call to the compute shader which handles remapping the kernel log polar buffer, applies a blur to the result, and handles TAA; the DLSS stage simply consists of the call to invoke DLSS on the results of all previous stages or if DLSS is disabled, it will only copy the results so far to the expected output buffer. As an additional note, we include the time to copy the color buffer to the history buffer in our compute shader stage, as this is to be considered a part of the TAA algorithm.

We chose to record the time measurements in the Sponza scene from what has been previously described as the overview position. For us, this proved to be the most taxing scene and viewpoint to render and thus we chose it to make the effects of the various modes as clear as possible. All rendering was done with a target resolution of 1920×1080.

Figure 5.1 displays the measurements taken for each stage wherein we recorded each stage for 1000 frames, as we iterate over the three real-time modes; that is the foveated mode, DLSS mode, and TAA mode. In this case with a target resolution of 1920×1080 it looks like the time DLSS takes to compute is almost equal to the time saved in the compute shader stage. From the figure it also looks like the effect of DLSS is a drop in ray trace and compute shader render times when switching to DLSS mode. Indeed we see that the time spent ray tracing and

the time spent in the compute shader are both roughly halved when DLSS mode is enabled. Further, we unsurprisingly see that the TAA mode has little overall impact on performance. To evaluate the magnitude of the render time difference before and after DLSS we use the mean total render time for each mode. The means are listed in table 5.4, where we see that the render time ratio between foveated mode and DLSS mode $\frac{5.193}{9.158} = 0.567$ and since TAA mode performs similarly to DLSS in terms of render time we have that TAA mode takes 43.3% less time than the foveated mode.

The time measurements for each stage used a foveation threshold of 15%, meaning that we replace the foveal area with a uniformly sampled disk at the foveal point with a radius of $0.15 \cdot \lambda$ where λ is the furthest distance to a corner from the foveal point. The 15% value was chosen somewhat arbitrarily based on what we determined would provide a decent performance improvement. We arrived at this value after testing a range of values between 0% and 100%. However, to provide a bit more of a well founded value for the percentage that could be used, we measured the render time with a varying foveation threshold and determined the foveation threshold that produced the minimum measured time. Figure 5.2 shows the results of these measurements and seems to indicate that an optimal foveation threshold, at least for the kernel function $K(x) = x^{\alpha}$ where $\alpha = 4$, should be somewhere in the range of $10\% - 20\%$ with our observed optimal value being 12.443%. The estimated normalized render time that can be seen in figure 5.2 is calculated from equation 5.1. This function evaluates the number of samples we should expect to be rendering by considering the area of the screen that is covered by the foveal disk.

$$f(x) = \frac{A_p(x) + A_c(x)}{W \cdot H} \tag{5.1}$$

where,

$$A_c(x) = \pi r(x)^2 - A_{out}(x) \tag{5.2}$$

and

$$A_p(x) = (w - u(r(x))) \cdot h \tag{5.3}$$

In these equations we have that $x \in [0, 1]$ is the foveation threshold, where $r(x)$ is the radius of the foveal area disk given x. We further have that $u(r(x))$ is the kernel log-polar radial coordinate, i.e. the coordinate that grows as we get further from the foveal point, expressed in log-polar buffer pixels. In this case, we use $u(r(x))$ to get the coordinate that maps to the border of the foveal area disk. Also, w and h are the rendering resolution width and height respectively, W and H is the target resolution width and height respectively. Finally, $A_{out}(x)$ represents the area of the foveal disk that is outside the screen bounds, $A_c(x)$ represents the area of the foveal disk that is left inside the screen bounds, and $A_p(x)$ represents the area of the kernel log-polar buffer that maps to the remaining regions that are not accounted for by the foveal area disk.

The numerator in the estimate in equation 5.1 consists of the amount of samples we trace outside the foveal area disk in addition to the number of samples we can expect the foveal area disk to contain. In figure 5.2 we show the observed and estimated render times normalized to the range $[0, 1]$. The normalization is done separately for the observed and estimated render times, and is calculated for each set of render times $\mathbf{R}$ by mapping each value $r \in \mathbf{R}$ to the normalized value $r_n = \frac{r - min(\mathbf{R})}{max(\mathbf{R}) - min(\mathbf{R})}$. In this case we see that the minimum render

Table 5.4: Table of the mean overall render time for each mode, derived from the data displayed in figure 5.1.

Mode	Render time (ms)
Foveated mode	9.158
DLSS mode	5.193
TAA mode	5.190

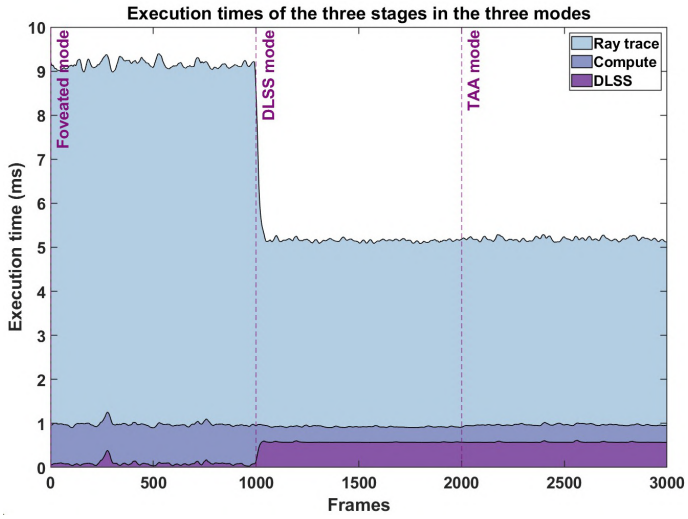


Figure 5.1: This graph shows the rendering time spent in the different stages we measured.

time was achieved at a foveation threshold set to 12.443% of the distance to the furthest corner from the foveal point. From the figure it seems that the observed render times and the estimation derived above follow a similar pattern of change which could indicate that equation 5.1 provides a good estimate of the behaviour with varying foveation threshold. However, more testing would be required to provide a more statistically robust answer, but in future render time measurements we would most likely use a value closer 12.443%.

5.3 Temporal stability evaluation

The temporal stability of the DLSS mode and the baseline foveated mode is low since we see large amounts of flickering and rapid color changes in the periphery. The foveated rendering causes the periphery to be subsampled which leads to instability especially when coupled with stochastic shadow samples. In fact, in our implementation the degree of subsampling that occurs towards the peripheral areas increase with a higher α value in our kernel function $\mathbf{K}(x) = x^{\alpha}$ which means that pushing up the α to increase performance, will on the other hand reduce the stability of the image. Further enabling DLSS on top of the already unstable foveated mode as we do in the DLSS mode, has shown to either keep the temporal stability roughly similar or even worsen it. The main reason we found for this result was that DLSS, which relies on subpixel jitter, does not smooth out the accumulation of samples over time enough for the jittering to be imperceptible in the periphery. This is most evident by

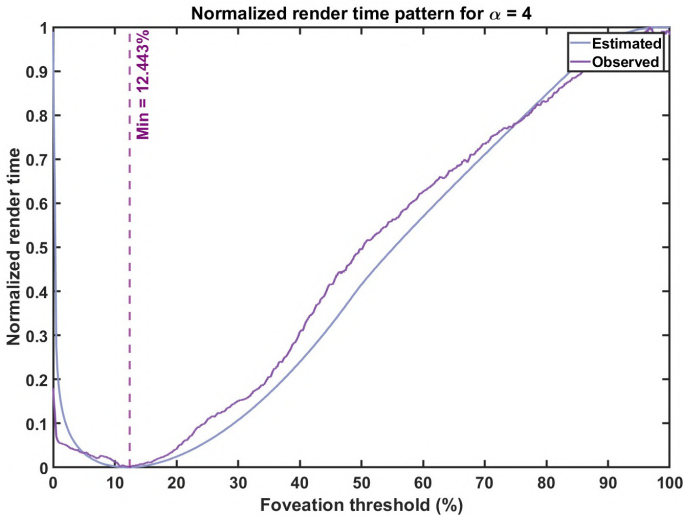


Figure 5.2: Here we see the observed and estimated render times, normalized to the range **[0, 1]**.

changing the magnitude of the subpixel jitter offsets when using DLSS, and as the magnitude approaches 0 we see an increase in temporal stability and the image starts to resemble a upscaled version of the foveated mode image.

However, we saw big improvements to the temporal stability when using the TAA mode, but we unfortunately could not provide a quantitative measurement of the improvement due to lack of temporal stability metrics that we could implement in time. So instead we rely on our visual perception and have taken a series of three videos to showcase the effect. The videos depict the inner chamber of the Sun temple as the camera orbits around the statue in the room. This scene and trajectory was chosen as it has several properties that make it especially likely to produce unstable frame sequences. The area has several lights at different scales that combine to produce both high contrast regions, and shadow with soft edges that are spread out over large surfaces which increases shadow noise. It also contains various high frequency geometry information that produce aliasing artifacts even when taking texture LOD into account. Finally, the orbital trajectory brings both distant and close up objects into view and puts the camera under constant predictable motion which puts our TAA implementation under a fair amount of stress.

The baseline foveated mode video[1] shows that the baseline foveated mode includes noticeable temporal artifacts such as flickering in the periphery. There are some temporal artifacts in the foveal disk too in this case since no anti-aliasing is being done.

We then have the video for the DLSS[2] mode, which shows that DLSS mode made the temporal artifacts more intense. It almost looks like the periphery is shaking on top of the already present temporal artifacts from the foveated rendering.

We then further show in the video for the TAA[3] mode that TAA mode reduces the temporal artifacts greatly. There are still some slight distortions and flickers that happen, especially in areas close to the foveal disk boundary. This might be due to the fact that TAA is phased out before reaching the foveal disk, instead of after reaching the disk. In future

[1] `https://youtu.be/cbpXxbaU-yY`
[2] `https://youtu.be/EVWUk1oZkqU`
[3] `https://youtu.be/FoKbaS0_4hk`

implementations we would recommend to phase the TAA out after reaching the foveal disk since an impact to the visual quality at the border of the disk is probably less noticeable than flickering in this area.

Finally, we have a video for reference of the ground truth mode[4] which shows how the scene looks without foveation and at a high sample rate. These videos can all be found on YouTube.

[4] https://youtu.be/FAiDEBwQB6c

Chapter 6
Conclusion

This is the final chapter where we first give some final remarks on the results presented as we try to evaluate the trade off between visual quality and render time that DLSS provides a foveated renderer. We then also write about some of the limitations of the results we got and discuss some future work that could be done on this and what we might have done differently. Finally, we also summarize the answers we arrived at for the research questions stated in section 1.2.

6.1 Final remarks

The results we obtained confirm that DLSS and foveated rendering indeed provides a reduction in render times and that DLSS also impacts the visual quality of the render both temporally and for each frame in isolation too. Foveated rendering with log-polar transformations is well known to produce visual artifacts, and is never really used without TAA or other methods to reduce these artifacts. However, we would argue that DLSS compounds this problem as it does not work well with TAA and is not adjustable enough on its own to be used with foveated rendering. This is what led us to use our TAA mode to selectively apply TAA where it should not affect perceived visual quality and provide a practical way to be able to use DLSS at all.

From the TAA mode results in sections 5.1.3 and 5.2 we can see that the worst case for the reduction of visual quality to be an increase in ꟻLIP weighted median error of **0.042**. Further, we see that the worst case overall TAA mode render time we found is **5.190** ms which is a considerable reduction of about **43.3%** over the foveated mode. This suggests that the visual quality loss due the DLSS with TAA is not high overall considering we took a ꟻLIP difference in the range **0.03-0.05** to be low when discussing flip in section 5.1. Further this also suggests that the performance increase is substantial since we see a **43.3%** reduction in render time. Since DLSS mode on its own produces a similar error as TAA mode we can also likely conclude that the visual quality impact of DLSS alone is also not that high.

It is hard however, to evaluate what the increase in the ꟻLIP error means in terms of user experience as we use foveated rendering to produce our images and ꟻLIP does not account for this.

From this we think that using the settings of the TAA mode has the potential to prove beneficial due to the performance boost and low quality loss. But we also think that more work is needed to more precisely determine the trade off between visual quality and performance that we see in these results.

6.2 Limitations

In this section we write about some limitations with our implementation and observations, as well as some potential sources of error for the results we obtained.

The fact that foveated rendering samples screen space in a non-uniform manner, and that ꟻLIP does not account for this means that the ꟻLIP results are inconclusive. For example, all the error differences we measured between the modes could have mostly been induced by visual changes in the peripheral regions alone, which would mean that the actual perceived quality was not impacted nearly as much as we observed. However, since we are uniformly sampling the foveal area we can at least assume that the perceived quality should be the same or similar to when using DLSS with uniformly sampled images.

We also need to mention that each ꟻLIP error is based on the difference evaluation of a single pair of test and ground truth images, we do not account for variations in the ꟻLIP error but it is likely small due to the dimensions of each image. Similarly, we only use 1000 data points for each render time measurement but we also do not expect to see big variation between frames, as the scenes are static and there is no camera motion in between the frames.

A note about the temporal instability results from section 5.3 is that whilst the foveated mode produced significant noise in the periphery, it would likely never be used without TAA or something similar applied to the results. This means that the temporal stability of the foveated mode video is not indicative of what one might expect in real applications of foveated rendering. However, the fact remains that the temporal stability is not helped by DLSS and that DLSS doesn't combine well with TAA, again why we apply the TAA selectively in the blurred peripheral areas and avoid the high detail foveal area.

As for the implementation, we should mention that the TAA implentation we use does not attempt to reject reprojected samples that are mismatched. This causes some noticeable ghosting around edges in the direction of movement as geometry hidden behind said edges suddenly becomes visible. There are ways to reduce this kind of error discussed by Yang et al. [31], and we made an attempt at implementing a depth based rejection method but it produced issues with the sampling of foveal area disk, and we dropped the idea due to resource constraints.

6.3 Future work

We have briefly mentioned various areas for improvement through out this thesis, but here in this section we describe these and some other potential future work in more detail.

To begin with, we mentioned earlier this chapter that more work is needed to better understand the impact of the reduction in visual quality. One way one might investigate this is by performing a user study, preferably using eye-tracking hardware and a head-mounted display, to determine perceived quality better. This would take the foveated rendering into account as opposed to ꟻLIP and could better account for variations from one person to the next, as we do not all have identical eyes.

Another potential area for further improvement is to implement an evaluation tool, similar to ꟻLIP , but which takes foveated rendering into account. This could possibly be done by weighting the perceived error that ꟻLIP produces by known data about the spatial resolution of the human eye. Yet another tool which could be useful for this kind of work is to attempt to apply the results produced by Krajancich et al. [13] to ꟻLIP results. In their work they present a unified measurement of the temporal resolution and spatial resolution of the human eye with regards to eccentricity from the fovea. This could be used to produce a temporal stability evaluation tool for use in providing numerical stability values for foveated rendering. To do this one might save a sequence of color buffers and corresponding motion vector buffers, and then use ꟻLIP to evaluate the difference of the overlapping areas from one frame to the next. Once the ꟻLIP errors are calculated one could use the temporal resolution measurements to weight the differences and produce something that is a closer representation of the perceivable change over time. Better yet, we could combine the temporal and spatial ꟻLIP based tools to produce something more generally applicable.

Another option instead of implementing a new image quality evaluation tool for foveated images would be to try using FovVideoVDP developed by Mantiuk et al. [17] which could work for evaluating our foveated images instead. FovVideoVDP is a video difference metric similar to ꟻLIP except it is meant for videos and takes into account the spatial, temporal, and peripheral aspects of human vision.

More direct uses for the results presented by Krajancich et al. [13] might be to attempt to use the human eye temporal resolution that they measure to affect the TAA retention bias so that areas with less temporal resolution can have a higher retention of history buffer color data, thus reducing any remaining perceivable flicker even more without majorly increasing the amount of visual artifacts. However, to apply the temporal resolution measurements one would also need to somehow measure the spatial frequency of the image itself which may not be easy.

On another note, as we showed in the results, foveated rendering based on kernel log-polar transforms definitely produces visual artifacts towards the periphery. We might have been able to avoid some of those artifacts by using something like the log-rectilinear transformations proposed by Li et al. [16]. This could have produced better results, and is probably worth an attempt in the future.

We also briefly discussed the optimal foveation threshold to maximize performance, but we did not have enough information to provide a conclusive answer. So that is another area worth investigating, by taking more measurements, in a variety of conditions, and possibly using a variety of foveation parameters, maybe even different kernel functions. Providing a good set of optimal foveation threshold values to choose from for different scenarios and kernel functions would give better tools for applying this kind of technique more generally whilst still maximizing performance.

Finally, since it is hard to extract what the ꟻLIP error differences mean for the quality of the foveal area disk. We think that better results in that regard can be obtained by ad-

ditionally only comparing the foveal areas themselves to the equivalent ground truth areas. We expect that there will not be any difference between DLSS mode and TAA mode since TAA is never applied in the foveal area disk, but that is not necessarily true. DLSS might produce different results in the foveal area by the fact that the peripheral areas are less noisy, not jittered, and just generally containing different information. Since it is unknown how exactly DLSS currently processes images, we can not say what the results would be.

6.4 Summary

We now go over the three research questions from section 1.2 and provide a summarized answer to each.

With the first question we set out to evaluate the render time reduction that we could get with DLSS. To answer that we can look at the render time results we got in section 5.2 where we see that DLSS provides a **43.3%** reduction in the render time.

The second question involved us asking what visual quality loss we would get from using DLSS with foveated rendering when comparing to ground truth. For this we used the FLIP evaluations and arrived at a maximum visual quality loss of **0.042** which we classify as a small loss since in section 5.1 we considered the FLIP error between a uniformly sampled, anti-aliased image and ground truth to be small.

Finally, with the third question we asked about how temporally stable the foveated render result is with DLSS applied. Unfortunately, we could not provide a quantitative measure for this due to time resource constraints. However, by observing video recordings we saw that the temporal stability with DLSS was probably worse or at best similar to the stability without DLSS. Since the temporal stability of foveated rendering alone is poor we then conclude that the combination with DLSS is also of poor temporal stability. But we also found that selectively applied TAA helped with the stability issues.

References

[1] Tomas Akenine-Möller, Eric Haines, Naty Hoffman, Angelo Pesce, Michał Iwanicki, and Sébastien Hillaire. *Real-Time Rendering 4th Edition*, chapter 2, pages 11–27. A K Peters/CRC Press, Boca Raton, FL, USA, 2018.

[2] Tomas Akenine-Möller, Eric Haines, Naty Hoffman, Angelo Pesce, Michał Iwanicki, and Sébastien Hillaire. *Real-Time Rendering 4th Edition*, chapter 6, pages 167–222. A K Peters/CRC Press, Boca Raton, FL, USA, 2018.

[3] Tomas Akenine-Möller, Jim Nilsson, Magnus Andersson, Colin Barré-Brisebois, Robert Toth, and Tero Karras. *Texture Level of Detail Strategies for Real-Time Ray Tracing*, pages 321–345. Apress, Berkeley, CA, 2019.

[4] Pontus Andersson, Jim Nilsson, Tomas Akenine-Möller, Magnus Oskarsson, Kalle Åström, and Mark D Fairchild. Flip: A difference evaluator for alternating images. *Proc. ACM Comput. Graph. Interact. Tech.*, 3(2):15–1, 2020.

[5] Helder Araujo and Jorge M Dias. An introduction to the log-polar mapping [image sampling]. In *Proceedings II Workshop on Cybernetic Vision*, pages 139–144. IEEE, 1996.

[6] Chakravarty R Alla Chaitanya, Anton S Kaplanyan, Christoph Schied, Marco Salvi, Aaron Lefohn, Derek Nowrouzezahrai, and Timo Aila. Interactive reconstruction of monte carlo image sequences using a recurrent denoising autoencoder. *ACM Transactions on Graphics (TOG)*, 36(4):1–12, 2017.

[7] Christine A Curcio and Kimberly A Allen. Topography of ganglion cells in human retina. *Journal of comparative Neurology*, 300(1):5–25, 1990.

[8] Epic Games. Unreal engine sun temple, open research content archive (orca), October 2017. `http://developer.nvidia.com/orca/epic-games-sun-temple`, Accessed: 2022-04-26.

[9] Jason Gregory. *Game Engine Architecture, Third Edition*, chapter 11, pages 667–697. CRC Press, 2018.

[10] Eric Haines and Tomas Akenine-Möller. *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*, chapter 1-3, pages 5–47. Springer, 2019.

[11] Johannes Hanika, Lorenzo Tessari, and Carsten Dachsbacher. Fast temporal reprojection without motion vectors. *Journal of Computer Graphics Techniques Vol*, 10(3), 2021.

[12] Matias Koskela, Timo Viitanen, Pekka Jääskeläinen, and Jarmo Takala. Foveated path tracing. In *International Symposium on Visual Computing*, pages 723–732. Springer, Cham, 2016.

[13] Brooke Krajancich, Petr Kellnhofer, and Gordon Wetzstein. A perceptual model for eccentricity-dependent spatio-temporal flicker fusion and its applications to foveated graphics. *ACM Transactions on Graphics (TOG)*, 40(4):1–11, 2021.

[14] Jens Krüger, Matthias Niessner, and Jörg Stückler. Multi-layer alpha tracing. 2020.

[15] Eric Lengyel. *Foundations of Game Engine Development: rendering*, chapter 5, pages 32–51. Number v. 2 in Foundations of Game Engine Development. Terathon Software LLC, 2019.

[16] David Li, Ruofei Du, Adharsh Babu, Camelia D Brumar, and Amitabh Varshney. A log-rectilinear transformation for foveated 360-degree video streaming. *IEEE Transactions on Visualization and Computer Graphics*, 27(5):2638–2647, 2021.

[17] Rafał K Mantiuk, Gyorgy Denes, Alexandre Chapiro, Anton Kaplanyan, Gizem Rufo, Romain Bachy, Trisha Lian, and Anjul Patney. Fovvideovdp: A visible difference predictor for wide field-of-view video. *ACM Transactions on Graphics (TOG)*, 40(4):1–19, 2021.

[18] Morgan McGuire and Guedis Cardenas. Cornell box, originally created at cornell university, 2011. `https://casual-effects.com/data/`, Accessed: 2022-04-22.

[19] Xiaoxu Meng, Ruofei Du, Matthias Zwicker, and Amitabh Varshney. Kernel foveated rendering. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 1(1):1–20, 2018.

[20] Bipul Mohanto, ABM Tariqul Islam, Enrico Gobbetti, and Oliver Staadt. An integrative view of foveated rendering. *Computers & Graphics*, 102:474–501, 2022.

[21] Pierre Moreau, Morgan McGuire, Frank Meinl, and Marko Dabrovic. Sponza, edited by pierre moreau for cg_labs, August 2020. `https://github.com/LUGGPublic/CG_Labs`, Accessed: 2022-02-11.

[22] Bui Tuong Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, 1975.

[23] Jan M Provis, Adam M Dubis, Ted Maddess, and Joseph Carroll. Adaptation of the central retina for high acuity vision: cones, the fovea and the avascular zone. *Progress in retinal and eye research*, 35:63–81, 2013.

[24] Jyrki Rovamo and Antti Raninen. Critical flicker frequency and m-scaling of stimulus size and retinal illuminance. *Vision research*, 24(10):1127–1131, 1984.

[25] Leon Shirman and Yakov Kamen. A new look at mipmap level estimation techniques. *Computers & Graphics*, 23(2):223–231, 1999.

[26] Hans Strasburger, Ingo Rentschler, and Martin Jüttner. Peripheral vision and pattern recognition: A review. *Journal of vision*, 11(5):13–13, 2011.

[27] Zhou Wang and Alan C Bovik. Embedded foveation image coding. *IEEE Transactions on image processing*, 10(10):1397–1410, 2001.

[28] Alexander Watson. Deep learning techniques for super-resolution in video games. *arXiv preprint arXiv:2012.09810*, 2020.

[29] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, jun 1980.

[30] Alan Wolfe. *Using Blue Noise for Ray Traced Soft Shadows*, pages 367–394. Apress, Berkeley, CA, 2021.

[31] Lei Yang, Shiqiu Liu, and Marco Salvi. A survey of temporal antialiasing techniques. In *Computer Graphics Forum*, volume 39, pages 607–621. Wiley Online Library, 2020.

[32] Lei Yang, Diego Nehab, Pedro V Sander, Pitchaya Sitthi-Amorn, Jason Lawrence, and Hugues Hoppe. Amortized supersampling. *ACM Transactions on Graphics (TOG)*, 28(5):1–12, 2009.

[33] Zheng Zeng, Shiqiu Liu, Jinglei Yang, Lu Wang, and Ling-Qi Yan. Temporally reliable motion vectors for real-time ray tracing. *Computer Graphics Forum*, 40(2):79–90, 2021.

Appendices

Appendix A
Figures

(a) Central position

(b) Overview position

(c) Close up plant view

Figure A.1: The ꟻLIP evaluation images for Sponza using the baseline foveated mode, each row representing a different view of the scene. From left to right within each row we have, the foveated test image, the ground truth reference image, and the ꟻLIP error heatmap. In these images the foveal point is set to the center of the screen, and notice how the central area is quite dark in the ꟻLIP heatmap due to the foveal area disk.

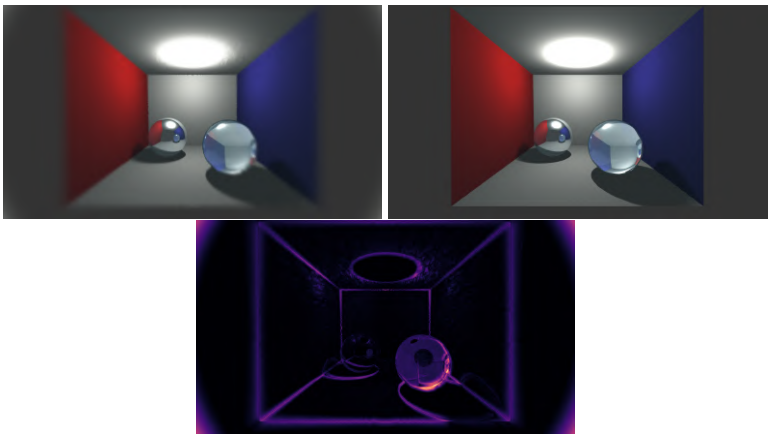


Figure A.2: The ꟻLIP evaluation images for the Cornell box with a reflective and a refractive sphere, using the foveated ray tracer mode. The top left image is the foveated test image, the top right image is the ground truth reference image, and the bottom image is the ꟻLIP error heatmap.

(a) Statue view

(b) Chamber view

Figure A.3: The ꟻLIP evaluation images for the Sun temple using the baseline foveated mode, each row representing a different view of the scene. From left to right within each row we have, the foveated test image, the ground truth reference image, and the ꟻLIP error heatmap. Similar to the Sponza images in figure A.1, we see a darker center disk in the ꟻLIP heatmap.

(a) Central position

(b) Overview position

(c) Close up plant view

Figure A.4: The ꟻLIP evaluation images for Sponza using the DLSS mode. Each row of images are taken from a different viewpoint, and for each row we have from left to right, the foveated test image, the ground truth reference image, and the ꟻLIP heatmap. The images show a slight reduction in visual quality compared to baseline foveated mode, especially in the periphery. This can be seen in the brighter ꟻLIP heatmap images and is confirmed by the weighted median in table **??**.

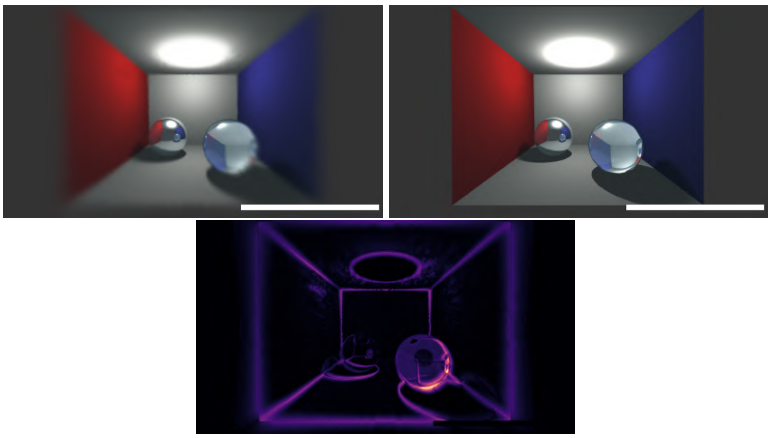


Figure A.5: The ꟻLIP evaluation images for the Cornell box with a reflective and a refractive sphere, using the DLSS mode. The top left image is the foveated test image, the top right image is the ground truth reference image, and the bottom image is the ꟻLIP error heatmap.

(a) Statue view

(b) Chamber view

Figure A.6: The ꟻLIP evaluation images for the Sun temple using DLSS mode, each row representing a different view of the scene. From left to right within each row we have, the foveated test image, the ground truth reference image, and the ꟻLIP error heatmap.

(a) Central position

(b) Overview position

(c) Close up plant view

Figure A.7: The ꟻLIP evaluation images for Sponza using the TAA mode. Each row of images are taken from a different viewpoint, and for each row we have from left to right, the foveated test image, the ground truth reference image, and the ꟻLIP heatmap.

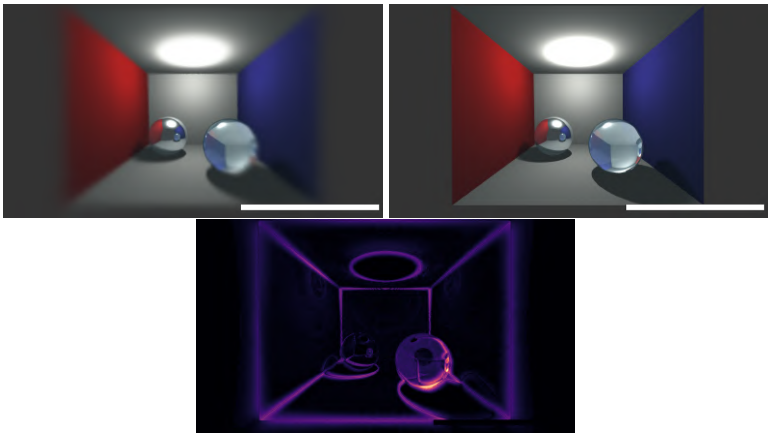


Figure A.8: The ꟻLIP evaluation images for the Cornell box with a reflective and a refractive sphere, using the TAA mode. The top left image is the foveated test image, the top right image is the ground truth reference image, and the bottom image is the ꟻLIP error heatmap.

(a) Statue view

(b) Chamber view

Figure A.9: The ꟻLIP evaluation images for the Sun temple using TAA mode, each row representing a different view of the scene. From left to right within each row we have, the foveated test image, the ground truth reference image, and the ꟻLIP error heatmap.

EXAMENSARBETE Image Upscaling for Ray Traced Foveated Rendering
STUDENT Charlie Mrad
HANDLEDARE Michael Doggett (LTH)
EXAMINATOR Flavius Gruian (LTH)

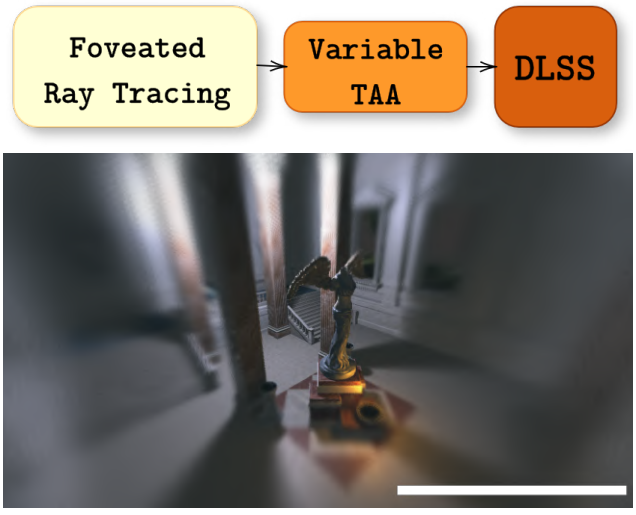
Snabb rendering av VR-anpassad strålspårning

POPULÄRVETENSKAPLIG SAMMANFATTNING **Charlie Mrad**

Virtuell realitet (VR) och strålspårning har blivit allt mer populära, och dessutom investeras många resurser i dessa teknologier. I detta arbete använder vi Nvidias DLSS för att skala upp strålspårade bilder som är anpassade för VR och uppnår en 1.75X ökning i prestanda.

I dagsläget har strålspårning ökat i popularitet i samband med att ny hårdvara lanserats och stora framgångar inom datorgrafiksforskning. Dessa framgångar består delvis av nya lösningar för bildförstoring. Det är speciellt i applikationer som använder virtuell realitet (VR) där strålspårning kan vara effektiv eftersom man kan lätt rendera realistiska bilder med strålar som efterliknar riktiga ljusstrålar, men VR är också extra kostsamt att rendera för. Detta på grund av att VR oftast använder två högupplösta skärmar, en för vardera öga. Mycket av det en människas ögon ser har en låg upplösning och därmed en dålig känslighet för detaljer. Denna kunskap har använts förr i datorgrafikrendering för att minska mängden arbete som krävs för att generera bilder och benämns på svenska som *fovea-styrd rendering*.

I det här examensarbetet utforskar vi möjligheten av att kombinera en ny bildförstornings teknologi kallad DLSS som utvecklats av Nvidia, med fovea-styrd rendering. Kombinationen gör det möjligt att generera bilder snabbare än om endast fovea-styrd rendering används. Resultatet visade att vi kan använda denna kombination för att nå ungefär 1.75X snabbare renderingstider, dock introducerar DLSS ökad instabilitet i bilden över tid. Instabiliteten uppenbarar sig som flimmer i periferiområdet av seendet.



För att åtgärda instabiliteten har vi använt en teknik som bygger på en kantutjämningsmetod vars engelska namn är *temporal anti-aliasing*. Vi testade tekniken i flera sammanhang, med en variation av olika 3D scener. I samtliga fall visade sig vår teknik att inte minska bildkvaliteten och samtidigt öka stabiliteten. I slutändan kunde vi i stort sett eliminera all ostabilitet och slutresultatet var en kraftig minskning av renderingstiden med minimal förminskning av bildkvalitet.