UNIVERSIDADE FEDERAL FLUMINENSE

VICTOR MARTINS PERES

# Non-homogeneous denoising for virtual reality in real-time path tracing rendering

NITERÓI 2022

#### VICTOR MARTINS PERES

# Non-homogeneous denoising for virtual reality in real-time path tracing rendering

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Mestre em Computação. Área de concentração: Ciência da Computação.

Orientador: Esteban Walter Gonzalez Clua

> NITERÓI 2022

P434n Peres, Victor Martins Non-homogeneous denoising for virtual reality in real-time path tracing rendering / Victor Martins Peres. - 2022. 57 f.: il.
Orientador: Esteban Walter Gonzalez Clua. Dissertação (mestrado)-Universidade Federal Fluminense, Instituto de Computação, Niterói, 2022.
1. Computação gráfica. 2. Realidade Virtual. 3. Unidade de processamento gráfico. 4. Processamento de imagem. 5. Produção intelectual. I. Clua, Esteban Walter Gonzalez, orientador. II. Universidade Federal Fluminense. Instituto de Computação. III. Título.

Figure 1: Ficha catalográfica

#### VICTOR MARTINS PERES

Non-homogeneous denoising for virtual reality in real-time path tracing rendering

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Mestre em Computação. Área de concentração: Ciência da Computação.

Aprovada em Outubro de 2022.

BANCA EXAMINADORA

Prof. Esteban Walter-Conzalez Clua - Orientador, UFF

Prof. Anselmo Antunes Montenegro, UFF

the fort.

Prof. Antonio Lopes Apolinario Junior, UFBA

Niterói 2022

 $\dot{A}\ minha\ família,\ meus\ amigos\ e\ minha\ companheira\ Nathalie.$ 

# Agradecimentos

Agradeço, em primeiro lugar, aos meus pais, meus irmãos, minha companheira Nathalie pelos apoios de todas as maneiras imagináveis durante toda a minha formação acadêmica e profissional.

Ao meu orientador, Esteban Clua, por todos os ensinamentos durante o mestrado, sejam eles técnicos, científicos ou acadêmicos, e por todos os direcionamentos quando a pesquisa encontrava um bloqueio.

Aos meus amigos do laboratório adquiridos durante o programa de mestrado, por estarem presentes nos momentos de estudo com interações e foco para continuar a pesquisa, e, claro, nos momentos de descontração.

Aos meus amigos de longa data por me ajudar a passar pelos momentos difíceis com as levezas que a vida pede.

E agradeço também à Universidade Federal Fluminense por manter em funcionamento e com excelência o programa de mestrado em computação tão rico quanto este e durante um período tão difícil quanto foram estes últimos anos.

### Resumo

Path-tracing é um método de renderização tradicionalmente usado em computação gráfica, especialmente em cinema, programas de televisão e animações digitais, sendo o estado da arte quando é necessário realismo gráfico. Path-tracing em tempo real está se tornando uma abordagem importante para o futuro dos jogos e aplicações interativas, como ambientes interativos de realidade virtual que necessitam de realismo gráfico para alto grau de imersão. Dado a natureza exponencial, este método de renderização necessita otimizações para ser executado de forma satisfatória (pelo menos 60 quadros por segundo) em tempo real. Entre as diferentes possíveis otimizações, a redução de ruído em imagens renderizadas pelo método de Monte Carlo é importante, devido à baixa densidade de amostras. Lidando com dispositivos de realidade virtual, outras otimizações podem ser consideradas, como técnicas de Foveated Rendering. Este trabalho propõe um novo pipeline de renderização para reconstrução de ruído de imagens renderizadas com Path-tracing em tempo-real para aplicações em um sistema de duas telas como um dispositivo HMD (Head-Mounted Display). Assim, as características da visão foveal são aproveitadas computando os G-Buffers contendo os atributos da cena e um buffer com a distribuição foveal tanto para a tela esquerda quanto para a direita. Em seguida, a renderização em path-tracing é computada conforme o buffer de coordenadas, gerando uma densidade pequena de raios iniciais por píxel selecionado e, logo, uma imagem ruidosa. Por fim, reconstruímos esta imagem com o denoiser não-homogêneo, considerando maior densidade de pixels na região central e menor densidade nas regiões periféricas, conforme as abordagens referentes ao foveated rendering. Os experimentos mostram que o pipeline de renderização proposto tem um fator de speedup de 1,35 em comparação com um pipeline sem as otimizações.

**Palavras-chave**: Foveated rendering, sistemas de duas telas, dispositivos HMD, redução de ruído, path-tracing.

## Abstract

Path-tracing is a rendering method traditionally used in computer graphics, specially in cinema, television programs and digital animation, being the state-of-the-art when graphical realism is required. Real-time Path-tracing is becoming an important approach for the future of games and virtual reality interactive environments, which also requires graphical realism for its immersive factor. Due to its exponential nature, this method of rendering requires optimizations for it to be executed in a satisfactory way (with at least 60 frames per second) in real-time. Among different possible optimizations, denoising Monte Carlo rendered images is necessary, due to low sampling densities. When dealing with Virtual Reality devices, other optimizations can also be considered, such as foreated rendering techniques. This work proposes a novel rendering pipeline for denoising a real-time path-traced application in a dual-screen system such as head-mounted display (HMD) devices. Therefore, characteristics of the foveal vision are leveraged by computing G-Buffers with the features of the scene and a buffer with the foveated distribution for both left and right screens. Later, we path trace the image according to the coordinates buffer generating only a few initial rays per selected pixel, and, thus, a noisy image. Lastly, we reconstruct the noisy image output with a novel non-homogeneous denoiser that accounts the greater pixel density in the central region and the smaller density in the peripherical regions, according to the approaches referenced by foveated rendering. The experiments showed that this proposed rendering pipeline could achieve a speedup factor up to 1.35 compared to one without the optimizations.

**Keywords**: Foveated rendering, dual-screen systems, HMD devices, denoising, real-time path-tracing.

# **List of Figures**

1	Ficha catalográfica	
2	A common representation of the rendering pipeline based on a rasterization method of rendering. Each one of these stages can also be a rendering pipeline into itself, subdividing it into different pipelines and other stages	16
3	Representation of a ray tracing rendering pipeline. The traversal and the intersection steps are computed via hardware and are accelerated by specific core units. Reference from Khronos documentation(KOCH et al., 2020)	17
4	Paths that rays follow starting in the camera, with a bounce over a surface. The colorized rectangles represent objects that have a diffuse surface. In this type of surface, the bounce is in a random direction given the normal hemisphere.	19
5	Left shows a path-tracing rendered image without denoising with 1 spp, middle shows a denoised image, and right shows a ground truth reference with 1024 spp. Reference from Nvidia Research blog (LEFOHN, 2021)	20
6	Distribution of cones and rods in the human vision given the angle of retina, demonstrating the increased concentration of cones in the center of the retina, decreasing as it furthers from it. Reference from SNR analysis in (LIN et al., 2012).	21
7	Our proposed rendering pipeline: the foveated distribution is outside the frame because it does not change between frames and remains the same across the application execution. There are three denoisers being executed, and each one is related to the corresponding region of interest	28
8	Feature buffers rendered in the left screen in order of calculation, from left to right: (a) diffuse, (b) world position and (c) world normal. The green scale at (b) represents the world position not normalized	29

9	The foveated distribution output, represented in a binary mask, rendered with the concentric circles delimiting the range of each of the three layers, with the outer layer being delimited by the borders of the screen	30
10	The path-tracing pass generates a noisy output for both screens using the G-Buffers and the coordinates buffer as an input, accounting for the foveated distribution.	32
11	The reconstructed image after the non-homogeneous denoising pass	33
12	Top illustration from the original method (DAMMERTZ et al., 2010) of the À-Trous filter with 3 levels of iterations with increasing step-width, bottom illustration depicting an example of our adapter À-Trous filter with 3 levels of iterations with decreasing step-width. Ours starts by taking into account the pixels at distance $2^i$ (black dots) from the center pixel in the accumulation process, and decreases for each subsequent iteration. The blue dots are skipped by the À-Trous filter and the gray dots are skipped by the binary mask modification	35
13	Rendering of Sponza scene in the <i>Base</i> experiment	43
14	Rendering of Sponza scene in the <i>NH3LD</i> experiment	44
15	Rendering of Sponza scene in the <i>NH2LD</i> experiment	45
16	Rendering of Sponza scene as a graphical reference using 1024 spp. $\ldots$	46
17	From left to right, top: reference image, image from <i>Base</i> experiment; bottom: image from <i>NH2LD</i> experiment, image from <i>NH3LD</i> experiment.	47
18	Heatmap of the difference between the <i>Base</i> experiment and the reference image	48
19	Forest scene. Left: the image rendered with 1024 spp as in the <i>Base</i> experiment. Right: the image rendered in our non-homogeneous denoising configuration	50
		00

# **List of Tables**

1	Comparison of the related works. NHD is the pipeline presented in this work.	26
2	Region distribution used in the experiment for comparison, with its reso-	~ ~
	lution, respective area size, samples, and spp density	37
3	Scenes tested and its characteristics.	40
4	Performance metrics of Base, NH2LD and NH3LD experiments, in averaged	
	milliseconds per frame for our GPU implementation.	41
5	Objective quality comparison with RMSE, SSIM and PSNR metrics	42

# Contents

1	Intr	oduction	12
	1.1	Motivation	12
	1.2	Proposal	13
	1.3	Hypothesis	13
	1.4	Objective	13
	1.5	Contributions	13
	1.6	Structure	14
2	Con	cepts	15
	2.1	The Rendering Pipeline	15
		2.1.1 Shaders	15
	2.2	Ray Tracing	16
	2.3	Path Tracing	18
	2.4	Denoising	19
	2.5	Foveated Rendering	20
3	Rela	ited Works	22
	3.1	Advancements on Path-tracing	22
	3.2	Advancements on Denoising	23
	3.3	Advancements on Foveated Rendering	24
	3.4	Summary of the related works	25

#### 4 Non-homogeneous denoising for foveated regions

	4.4	Non-homogeneous denoising pass	31
5	Imp	lementation	36
	5.1	Implementation	36
6	Resu	ults	39
6 7	Resu Con	ults clusion	39 49
6 7	<b>Resu</b> <b>Con</b> 7.1	ults clusion Limitations	<b>39</b> <b>49</b> 49
6 7	<b>Resu</b> <b>Con</b> 7.1 7.2	ults clusion Limitations	<ul> <li>39</li> <li>49</li> <li>49</li> <li>50</li> </ul>

## **1** Introduction

Path tracing is a consolidated rendering approach for achieving photo-realistic graphics, but currently not suitable for real-time performance. Shadows, global illumination, and reflection are essential graphical effects that are intrinsically achieved through it. Additionally, since its complexity is directly related to the number of pixels of the screen, using it on high-resolution devices is challenging, requiring non-trivial solutions to be used on head-mounted displays (HMDs).

Furthermore, path tracing is the current state-of-the-art in real-time rendering games, interactive rendering in graphical applications, and pre-rendering films. Recently, Nvidia launched RTX GPUs, embedded with hardware acceleration related to path tracing (KIL-GARIFF et al., 2018), enabling an increasing number of consumers and workplaces to leverage these optimizations for real-time applications, such as games and virtual reality.

#### 1.1 Motivation

Even so, it is still laborious and time-consuming for the graphical processors to render at high resolutions. Therefore, performance optimizations are required, such as hardwarebased Bounding Volume Hierarchy (BVH) acceleration structure, ray-polygon intersections computed in hardware, and better sampling of the traced pixels. Denoising is one of the most important optimization methods. It is used to reduce the variance in a noisy image produced by the path-tracing rendering on a low ray sampling. In the literature, it is possible to find different denoising techniques such as wavelet filters (DAMMERTZ et al., 2010), bilateral filters (PARIS; DURAND, 2006), machine learning algorithms (KALANTARI; SEN, 2013), sampling with spatio-temporal accumulation (SCHIED; KA-PLANYAN, et al., 2017), deep-learning techniques (INTEL®, 2019), to name a few of the most prominent ones.

Foveated rendering is an important topic and is related to the fact that the human eye can only distinguish important details of pigments in a central region of the retina, called fovea (REDDY; REDDY, 1997).

#### **1.2** Proposal

This work presents a novel approach to reconstruct path-traced rendered environment images in real-time by leveraging the perception characteristics at the center of the human vision. The proposed rendering pipeline splits the path-traced noisy image in different concentric layers and apply denoising strategies on each layer differently, with parameter values that are adapted according to the different regions of the visual field. Doing so, we intend to reduce the number of traced rays at the peripheral area of the retina, applying stronger denoising filters, but maintaining acceptable number of rays at the central region of the display.

#### **1.3 Hypothesis**

The hypothesis that this work is attempting to validate is the following one: could a path-traced rendering pipeline, in foveation, with a non-homogeneous method of denoising be more efficient in terms of computer performance, while maintaining or even slightly decreasing in quality, than a path-traced rendering pipeline with a fixed denoising?

#### 1.4 Objective

The objective of this dissertation is divided in two major points and one minor. The first one is to make an improvement in the performance of the path-tracing rendering in foveated systems, so that it can be more feasible to run a realistic scene in a virtual reality environment. The second is to make this into a rendering pipeline in a way that can be applied easily into a game or a graphical application that runs in an HMD. And even a third one can be also discussed, about the effective implementation of the rendering pipeline.

#### **1.5** Contributions

We tested our solution using a collection of scenes with different triangles count, different display resolutions corresponding to the entirety of the dual screens of a regular HMD,

different spatial sampling configurations for the path-traced step, and different reconstruction layer parameters. Our method achieved speedup improvements in all configurations when compared to non-optimized implementations.

The contributions can be summarized in both the introduction of a non-homogeneous denoising rendering pipeline and its implementation, and an improvement in the performance of real-time path-tracing images in a virtual reality environment.

#### **1.6** Structure

In this chapter, we discuss the motivations, objectives, hypothesis, main contributions and main aspects that have led to the development of this work. For the next chapters, this work is organized as follows.

In chapter 2, the main theoretical concepts are explained in order to increase the comprehension of this work and its main aspects. These concepts were explored during the research and implementation of this work.

In chapter 3, the Related Works section brings the groundwork laid down by previous research in the most relevant fields to develop this work, which are Path-tracing, denoising and foveated rendering. Distinguishing the main contributions of these works, their limitations and how the proposed solution implements upon them.

In chapter 4 we introduce our proposed rendering pipeline. It explains each step of the proposed rendering pipeline, from the scene input specification to the generation of the denoised output.

In chapter 5, the implementation details of the pipeline are presented, including the main algorithm that was introduced, modifications to the parameters of the groundwork, the input and output of the rendering pipeline.

In chapter 6, the results obtained through performance benchmarks are presented. It also describes an objective analysis of the graphical quality through commonly used metrics.

At last, the work is concluded in chapter 7, bringing a summarized discussion of how the implementation and the experiments contributed to the field of research. We also discuss how this work could be extended by future researches.

## 2 Concepts

This section aims to present the main concepts involved in the idealization and implementation of the work proposed by this dissertation.

#### 2.1 The Rendering Pipeline

A graphics rendering pipeline consist of different stages to render one or multiple objects in a scene into an image to be shown in the target display (or displays, in the case of virtual reality). These stages are run either in the CPU or the GPU, depending on the task of each step. Figure 2 illustrates a common representation of how the pipeline is split into different stages.

In a pipeline with n steps, the i step with i ranging from 2 to n depends on the results of the step i - 1. Thus, the rendering pipeline is incremental and, excluding the first one, each step either alters or add data on top of the result of the previous steps.

There are other forms of rendering pipelines, like the ones implemented for fully offline rendering. The approach described is basically the canonical of real-time rendering.

Usually, most of the programmer's work resides in coding the shading stages inside the Geometry or the Fragment stages. There are other steps such as render output units that are not fully programmable and only configurable. Additionally, each step, when executed on the GPU, can be described in the form of a shader.

#### 2.1.1 Shaders

The shaders are programs that are executed in a process inside the GPU. The parallel nature of the GPU guarantees that the shaders are the same across an entire streaming processor. Thus, the shaders that are supplied for the GPU are going to be executed in each one of its cores. This guarantees that the same task will be run for a set of data,



Figure 2: A common representation of the rendering pipeline based on a rasterization method of rendering. Each one of these stages can also be a rendering pipeline into itself, subdividing it into different pipelines and other stages.

e.g. a set of vertices representing a model or a set of pixels representing the displays.

Ray-tracing rendering pipelines were defined a little bit different from rasterization and its APIs requires a set of shaders who are also different from rasterization. While the latter follows the steps of vertex processing, followed by geometry shaders and ending in a pixel shader, the former includes 5 different ones, from ray generation to misses and hits.

There's still the use of general compute shaders, which are used for a parallel task to be executed in the GPU, related to graphical rendering or other type of task. In this work, we use the power of compute shaders as a rendering task, in the form of the non-homogeneous denoising step.

#### 2.2 Ray Tracing

Ray-tracing started being massively adopted by the industry and consumers after 2018 with NVIDIA's RTX architecture (KILGARIFF et al., 2018). In contrast with rasterization, it has little hardware optimization over these last few decades. Bounding volume hierarchy (BVH) structure calculations and ray-triangle intersections started being accelerated by a dedicated hardware unit inside the GPU, known as RT cores. Figure 3 represents a pipeline of the ray-tracing method of rendering in the GPU, including the shaders that we use in our work such as ray generation, intersection processing of any or closest hit, and miss processing using the skybox or a clear color.

Ray-tracing is inherently more computationally expensive than rasterization due to the nature of its exponential algorithm. From its starting point being the cameras, it traverses the scene through a viewport pixel until it hits a triangle. From this collision, another set of ray is cast to the lights in the scene to check if the objects are occluded by other objects.

So, from 1 ray generated per pixel other N new rays are created, where N is the



Figure 3: Representation of a ray tracing rendering pipeline. The traversal and the intersection steps are computed via hardware and are accelerated by specific core units. Reference from Khronos documentation(KOCH et al., 2020).

number of lights. If the ray hits a surface of an object, it can generate another N rays at each ray-triangle intersection with the objects in the scene. And it also may generate more rays at this intersections, to cover graphical effects such as reflection, refraction and global illumination.

In this work, we proposed a rendering pipeline of shading using ray-tracing shaders steps run in the GPU and a foveation distribution step in the CPU. One could also implement it using the rasterization shaders for the Pre-pass step.

It is perfectly acceptable to use a hybrid approach instead of a purist full pathtracing rendering pipeline. Specially when this approach results in an increase in the performance and little to no graphical effects are lost. After all, the performance in a realtime rendering is the preference, specially in virtual reality environments which require at least 90 FPS to be acceptable. Rasterization can and should be used in steps such as G-Buffer computations, in the same way that compute shading is suited for post-processing techniques and shadow maps, as it was done in this work in the step of non-homogeneous denoising.

#### 2.3 Path Tracing

Path-tracing is a specific form of ray-tracing that intends to diminish the exponential factor of the original proposition by Whitted (WHITTED, 1979). Kajiya introduced a new, faster form of ray-tracing (KAJIYA, 1986) that computed the rendering equation (2.1) in a novel way.

$$L_o(\mathbf{x},\omega_o) = L_e(\mathbf{x},\omega_o) + \int_{\Omega} L_i(\mathbf{x},\omega_i) f_r(\mathbf{x},\omega_i,\omega_o)(\omega_i \cdot \mathbf{n}) \,\mathrm{d}\,\omega_i$$
(2.1)

In the rendering equation, the term of the outgoing light  $L_o$  represents the final shaded color for a point  $\mathbf{x}$  with the direction outgoing to the eye  $\omega_o$ . The term  $L_e$  represents the emitted light, in case a point is a light source. The integrand over the domain  $\Omega$  have three terms: incoming light  $L_i$  represents the shaded color for a point  $\mathbf{x}$  with the direction incoming from the light  $\omega_i$ , the material term in function  $f_r$  represents the reflected light for the surface in the given directions and point, and the dot product  $\omega_i \cdot \mathbf{n}$  represents the Lambert term of the importance given to the incoming light related to the normal  $\mathbf{n}$  of the surface.

Kajiya's rendering equation describes a ray-tracer that traces several rays per pixel through a path (hence, path-tracing) using only a single bounce in a random direction through the triangle hemisphere. This way, the samples do not grow exponentially, but only in the desired depth of the path. To shade the pixel, one must average the rays sampled in a single pixel. This form of rendering leverages from the Monte Carlo method (VEACH; GUIBAS, 1995) of integrating the rendering equation in a numerical integration, that is based in selecting random samples and averaging these samples to give the resulting light. Figure 4 illustrates how a path tracer works.

The resulting variance in the image is directly related to the number of samples. Thus, the downside from this method of rendering is that, in order to converge to a satisfactory rendered image with a low enough variance between the pixels, thousands of rays must be cast. In any use case that does not have the needs to be rendered in real-time, this is feasible with render programs that leverage from the CPU or GPU multi-threading. But for applications such as VR environments or games, where the rendering in GPU also shares time slices with other tasks, a few samples per pixel are a problem of its own. This problem greatly increases when expanding for some thousands of rays.



Figure 4: Paths that rays follow starting in the camera, with a bounce over a surface. The colorized rectangles represent objects that have a diffuse surface. In this type of surface, the bounce is in a random direction given the normal hemisphere.

#### 2.4 Denoising

Denoising is a strategy for image reconstruction when the image was generated by a Monte Carlo path-tracing rendering method, and it has left a high variance in the shaded pixels of the image. Without it, the rendered image will have only a vague resemblance to the ground truth image (that has no variance) since it will be full of variance between each of the pixels in the same neighborhood. Figure 5 shows a difference between the process of denoising and the ground truth image, with three different rendered images, one being a path-traced image without the denoiser applied (high-variance), one being a path-traced image with the denoiser applied, and one being the ground-truth image.

Denoising takes as an input a noisy image, features over the scene (such as normal and position generated using G-Buffers), and applies filters in the pixels to transform it in an output image with almost no difference to the ground truth image. So, instead of rendering an image with thousands of samples per pixel, an optimized path-tracing rendering pipeline (such as the one from this work) can generate it with few samples per pixels and apply denoising over it to get a comparable result.

Besides the G-Buffers, denoising techniques such as filters (DAMMERTZ et al., 2010)



Figure 5: Left shows a path-tracing rendered image without denoising with 1 spp, middle shows a denoised image, and right shows a ground truth reference with 1024 spp. Reference from Nvidia Research blog (LEFOHN, 2021).

are usually an average over the current pixel with its neighboring pixels, giving weight to each pixel's contribution in according to its position, similarity to the central pixel, and even the normal. There are also denoising techniques such as (SCHIED; KAPLANYAN, et al., 2017) which use color data from the same pixels but over different points in time, i.e. previous frames, adding a temporal accumulation instead of merely spatially. This is also a form of increasing the number of samples, and it is most effective in a non-moving scene, since the pixels are mostly the same over sequential frames.

There are also deep learning techniques that can be fed ground truth images and noisy images as an input and produce a neural network capable of denoising an image with a quality comparable to a "conventional" denoising and its convolution filters (INTEL®, 2019).

#### 2.5 Foveated Rendering

Foveated rendering is a concept that aims to improve the rendering performance by leveraging the characteristics of the human vision, such as the distribution of cones and rods in according to the angle of the eye, as illustrated in Figure 6. It does so by concentrating most of the rendering efforts in an area that the user will focus, the center of the vision, and diminishing the details and effects in other areas, as the periphery of the vision. It usually does either from monitoring the user's gaze using an eye-tracking system via cameras, or from assuming the user is focusing on the center of the screen.



Figure 6: Distribution of cones and rods in the human vision given the angle of retina, demonstrating the increased concentration of cones in the center of the retina, decreasing as it furthers from it. Reference from SNR analysis in (LIN et al., 2012).

There are different forms of leveraging the foveation, and it is not a new concept. (DUCHOWSKI; ÇÖLTEKIN, 2007) adopts the user's gaze to limit the level-of-detail in an object to decrease the number of vertices given the distance to the focused area. By decreasing the LOD of the objects in the periphery, it can improve the performance or choose to give more details to the focused area. For shading, there's the possibility to render in different resolutions according to the areas of the vision, such as dividing the viewport in three different areas with each one with decreasing resolution that was implemented in (GUENTER et al., 2012).

Foveation can also be used to affect other steps of the rendering, such as post-process and denoising. In a path-tracing rendering pipeline, the number of rays can be adapted in according to the user's gaze as well. In the center of the vision, one can diminish the number of rays and in the area correspondent to the periphery of the vision (or the viewport), the denoising can also be adapted. Multiple ray density and denoising factors, according to the fovea is the main proposal of our work presented, combining it with concepts of (GUENTER et al., 2012).

## **3 Related Works**

This work combines different rendering methods and advantages in one single rendering pipeline. We present in this chapter each of the similar rendering methods available in the literature.

Among these, there are different methods available for path-tracing rendering, each one suited for a different type of application. There are interactive rendering, offline rendering and real-time rendering. Since our proposal focus in a real-time application, the discussed methods are all applicable to have a real-time performance with the implementation.

Besides that, this work acknowledges the advancement of other optimization techniques that fall into different categories in the research area that are the path-tracing reconstruction algorithms and techniques known as denoisers. There are denoisers that use machine learning-driven filters or sampling through spatio-temporal accumulation. This work focus on real-time reconstruction techniques that uses a rule-based system, such as bilateral kernel filtering with geometric data in the form of G-Buffers.

In the realm of foreated rendering, there are a few techniques that can be applied or adapted to a path-tracing rendering pipeline.

Our work combines these features to improve path-tracing rendering within dualscreen systems, taking advantage of reduced density of pixel required in peripheral display regions. This section describes the basics and related works associated with its proposal.

#### 3.1 Advancements on Path-tracing

Whitted (WHITTED, 1979) introduced the first ray-traced image with not only shadows directly projected from a light source but also including a series of additional graphical effects, such as specular reflection and refraction, simulating the light transport with optical properties from the Fresnel equations. Later, Cook (COOK; PORTER; CAR- PENTER, 1984) introduced a random distribution of the rays sampled in the hemisphere of a reflected surface.

Kajiya introduced a novel form of ray tracing, named path tracing, which solved the exponential problem that the nature of ray tracing methods introduced (KAJIYA, 1986). It combined concepts of Monte Carlo integration and, instead of shooting a set of rays for each ray bounce, it shot only one ray for each ray bounce. Given this property, it is possible to achieve a high level of graphical realism with all the mentioned visual effects and global illumination features in a much faster way. This is made through several samples per pixel, i.e., the lighting factor of an object that is perceived with the light rays bouncing from different surface sources.

Hybrid approaches such as (BARRE-BRISEBOIS et al., 2019; SABINO et al., 2012) were introduced to further improve performance optimizations by leveraging the advantages of rasterization, ray-tracing, and compute shaders. It uses rasterization in G-Buffer, direct shadowing stages and ray-tracing in other steps such as direct, indirect lighting, and real-time reflections. It also uses compute shaders in the post-processing stages.

To guarantee that the path-tracing algorithm via Monte Carlo integration produces a result that converges into a photo-realistic image, it is necessary hundreds (if not thousands) of samples per pixel to achieve an almost perfect image. Today, the most demanding scenes can handle at most a few sample per pixel (spp) with global illumination. This generates an image with Monte Carlo variance noise and, due to the nature of Monte Carlo integration, the noise is decreased only in a square-root proportion to the number of samples (VEACH; GUIBAS, 1995). Hence, there is a need to reconstruct this non-perfect noisy image using properties from the scene's geometry.

There are several works that are dedicated to improve the state of the art of ray/pathtracing. A recent survey (HUA et al., 2019) of some of these works is a good starting point to understand and to improve the quality/performance of rendering.

#### 3.2 Advancements on Denoising

Several algorithms and techniques are available for reconstructing images in real-time. Methods related to machine learning techniques and neural network training have recently gained attention. Some of these works use the concept of autoencoders, which are being popularized due to their adoption by the graphics industry, such as Intel Open Image Denoise (INTEL®, 2019) and Nvidia Optix Autoencoder (NVIDIA, 2017). There are

also techniques that use concepts of image analysis and processing, such as convolution filters. These are constituted by bilateral filters, which can produce simple results with some graphical artifacts of brightness change or blur. The caveat is that bilateral filters can be computationally expensive as the size of the filters increases, becoming inefficient for the use in path tracing and real-time requirements.

Filters guided by buffers progressively reduce the artifacts. More specifically, the filter introduced by Dammertz et al. (DAMMERTZ et al., 2010) is known as Edge-Avoiding À-Trous. This filter satisfactorily fills the noisy image, being capable of avoiding almost any artifacts, which makes this suited for filters with big kernel size and spaced with smaller sampling size. Further on this, there is another Edge-Aware filter, proposed by Qi et al. (QI; WU; HE, 2012), that improves this approach and is capable of decreasing the atmospheric fog and the haze of an image by filtering with a decreasing step in each iteration of the process. Although the original problem was not related to rendering, these results may also be applied to reducing the same artifacts present in a noisy image.

Other works use similar filtering concepts, such as those produced by Schied et al. (SCHIED; KAPLANYAN, et al., 2017; SCHIED; PETERS; DACHSBACHER, 2018), which uses a guided filter with spatio-temporal variance and temporally accumulates the samples using accumulation buffers from previous frames in a way that moving objects are accounted for the global illumination.

Regression-based techniques with QR factorization are recently showing promising results due to their high performance for real-time denoising in path tracing pipelines (KOSKELA; IMMONEN, et al., 2019). Zwicker et al. (ZWICKER et al., 2015), and Kaplanyan et al. (KAPLANYAN et al., 2019) present more details and discussions related to the topic.

#### 3.3 Advancements on Foveated Rendering

Foveated rendering divides the rendering areas into different regions, using specific and separated rendering parameters for each region. Guenter et al. (GUENTER et al., 2012) split the image into three distinct layers with varying rates of sampling following the user gaze, enabling lower selection in the layer far from the center of the fovea gaze due to its small retina cell cones density. Their work is relevant due to the introduction of the layers based on the user's gaze, even though it uses rasterization as its rendering method. It showed with benchmarking that foveation rendering achieves a speedup with a factor

of 5-6x of a non-foveated rendering on desktop displays (GUENTER et al., 2012).

Weier et al. (WEIER et al., 2016) introduced the idea that a linear falloff is more suitable when dealing with ray tracing for HMD, in comparison with the previous hyperbolic falloff, due to the motion perception in the periphery vision area (WEIER et al., 2016). Since it used only direct lighting with point lights, area lights, or ambient occlusion shading, it missed global illumination and reflection effects. Their proposal achieved different speedups ranging from 1.46 to 4.18 depending on the rendering configuration.

Meng et al. (MENG et al., 2018) developed a work based on a new type of kernel log-polar space mapping, using polynomial functions to map the Cartesian coordinates in an early stage of the rendering pipeline (and inverting the transform afterwards). By leveraging the log-polar, they managed to gain a significant speedup and mitigated the low sampling with a temporal anti-aliasing method with a Gaussian filter with a 3x3 kernel.

Koskela et al. (KOSKELA; LOTVONEN, et al., 2019) introduced a Visual-polar mapping instead, and path-tracing samples in this space that follows the distribution of the visual acuity of human vision, gaining performance in the rendering and denoising phases.

A recent survey (MOHANTO et al., 2022) also provides an integral view of foveated rendering, including these mentioned papers and some others who are tangential to the objects of study of our work as described here.

Our work improves real-time performance optimizations by introducing a novel concept of a non-homogeneous type of path-traced image reconstruction, using denoising filtering algorithms with different levels according to the linear falloff of the center of the user's gaze and rendering it in a path tracing environment with a pre-determined ray sample distribution.

#### **3.4** Summary of the related works

Among these related works, (KOSKELA; LOTVONEN, et al., 2019), (MENG et al., 2018), (WEIER et al., 2016), and (GUENTER et al., 2012) are the ones who developed advancements in the field of foveated rendering together with either ray-tracing/path-tracing and/or denoising further, being the most relevant to the development of our work on a path-traced rendering pipeline with non-homogeneous denoising for virtual reality.

Table 1 brings a comparison between this work and these other works.

Table 1: Comparison of the related works. NHD is the pipeline presented in this work.

Work	Rendering	Post-processing	Foveation
Non-homogeneous Denoising (NHD)	Ray-tracing	Non-homogeneous Edge-avoiding wavelet denoising	Linear falloff
(KOSKELA; LOTVONEN, et al., 2019)	Ray-tracing	Block-wise Multi-order feature regression denoising	Visual-polar space
(MENG et al., 2018)	Rasterization	Anti-aliasing with 3x3 Gaussian Filter	Log-polar space
(WEIER et al., 2016)	Ray-tracing	Temporal resampling	Linear falloff
(GUENTER et al., 2012)	Rasterization	Hardware MSAA and temporal reprojection	Hyperbolic falloff

# 4 Non-homogeneous denoising for foveated regions

This work proposes a novel rendering pipeline suitable for path-tracing techniques running on HMDs. Although the proposed solution is agnostic of the hardware, when designing for a better performance a GPU-based system is ideal due to obvious rendering requirements and to the Non-homogeneous denoising step which is also highly parallelized.

This section describes the rendering pipeline as it is designed to run on both GPU and CPU. The pipeline is illustrated in the Figure 7 and involves four steps: (1) the pre-pass that computes G-Buffers; (2) the foveation distribution step, that computes the pixels that will be sampled in the following passes; (3) the path-tracing pass that computes the lighting for both screens including graphical effects and (4) the non-homogeneous denoising applied onto a noisy path-traced image being rendered in two screens, with the foveation adjustment.

#### 4.1 Pre-pass

The first stage consists of a set of ray-tracing shaders that generates one sample of ray for each pixel (spp) in both sides of the viewport (one for the left and one for the right eye). Once the traced ray hits a surface, the shading data of the point intersected is computed. The calculated shading data for each pixel can be seen in Figure 8, with the diffuse component of the texture (a in the figure); the position in world coordinates (b); the normal of the surface in world coordinates (c). The result of this computation is stored in buffers (G-Buffers), required for the subsequent passes, including both the path-tracing (by feeding the buffers as input of a deferred shading process) and denoising passes.



Figure 7: Our proposed rendering pipeline: the foveated distribution is outside the frame because it does not change between frames and remains the same across the application execution. There are three denoisers being executed, and each one is related to the corresponding region of interest.

#### 4.2 Foveation distribution step

In order to take advantage of the fovea distribution, in this stage we compute a buffer with only the pixels that are being selected to be sampled with rays in the further step of ray generation. This computation is constant for all frames and independent of the time or the scene. For this reason, we choose to compute this step in the CPU and to transfer it to a constant buffer to the GPU.

In that sense, for coordinates distribution, we used only one buffer. Moreover, the buffer is used in both the left and right sides of the screen in the same way, i.e., a pixel with coordinates (x, y) is either going to be sampled or not in both screens. This simplification disregards any possible mismatch of the cones' distribution between one eye and the other.

Using concepts similar to those proposed by Weier et al. (WEIER et al., 2016), this step builds a three-layer distribution that is represented by concentric circles of pixels with different sampling decay, in a linear proportion in each layer. Even though previous studies have already shown that the decay of cones in the view falls in a hyperbolic distribution (GUENTER et al., 2012), there is a major advantage in choosing a linear distribution in foveated rendering: the movement in the scene around the periphery area of the vision affects less the perception when compared to the hyperbolic distribution. Thus, we follow with a linear distribution to mitigate this problem.



Figure 8: Feature buffers rendered in the left screen in order of calculation, from left to right: (a) diffuse, (b) world position and (c) world normal. The green scale at (b) represents the world position not normalized.

The first layer represents the fovea by being located at the center of the screen and has a radius  $r_0$ . It is defined in such a way that the samples are taken at full resolution, i.e., each pixel will receive at least one sample. In case of a configuration of 1 spp, it should be the same spatial sampling that occurred in the G-Buffers computation in the pre-pass step. The inner radius of the second layer is defined with the starting radius right where the first layer ends, i.e.,  $r_0 + 1$  is the inner radius.  $r_1$  is its outer radius. Thus, it is the first layer outside the fovea layer. As in previous works (WEIER et al., 2016), a linear proportion decay is applied in this layer, starting from  $r_0 + 1$  and ending in  $r_1$ , parametrized with the probability p as a constant for this step in order to calculate the random value. This gives the sub-sampling effect of the middle layer, with each pixel having a probability p of being sampled according to the distance of the pixel to the center of the fovea. The third layer starts from radius  $r_1 + 1$  and ends in the edge of the viewport, being the layer outside the second layer. It receives even less sampling than the second layer values, with a constant probability of 1 - p for a pixel being sampled by a ray.

Once the computation of foveation distribution is complete, the screen coordinates of the pixels are stored in the coordinates buffer in the form of a contiguous list of the coordinates to be selected. We also store this computation in a binary mask for the denoising pass to access the coordinates in constant time (with the index to their coordinates in each of the GPU executing thread in the next steps), as shown in Figure 9. Thus, the output of this step are both the coordinates buffer with the size of the number of pixels that are being sampled, and the binary mask, containing the positions of the sampled pixels decided by the foveated distribution with linear decay, through the probability *p*.



Figure 9: The foveated distribution output, represented in a binary mask, rendered with the concentric circles delimiting the range of each of the three layers, with the outer layer being delimited by the borders of the screen.

This step is necessary for the posterior path tracing pass in each one of the screens of the HMD and for the denoising pass.

#### 4.3 Path-tracing pass

The following step computes the direct shadows and the global illumination. Using the previously computed coordinates buffer, the set of path-tracing shaders generates rays only with the coordinates defined by each thread index, according to the buffer. The number of threads is equal to the number of coordinates in the coordinates buffer, and each thread selects only its corresponding coordinate in the buffer to select for sampling. This is

used for generating rays that will pass through the buffer instead of every pixel on the screen. Thus, when taking into account the rendering based on visual acuity advantage, we maintain the sub-sampling property of this render pass, with a density of full-sampling for the inner layers and smaller for the middle and outer layers. Through this foveated rendering technique, it is possible to decrease the number of generated rays through a configurable proportion given by the size of the coordinates buffer in the previous step.

For each pixel, we compute the indirect lighting, direct lighting, and shadow effects, using two different groups of shaders: the shadow group and the indirect lighting group. For the shadow group, when the generated ray hits a point, it shoots a visibility ray to a random light in the scene. For the indirect lighting group, when the generated ray hits a point, it shoots an indirect ray to a random direction in the hemisphere to compute its color, along the standard secondary shadow ray for its bouncing factor. For computing the BRDF, we used the Lambertian material (OREN; NAYAR, 1994).

It is worth noting that instead of two ray-tracing programs or instance of programs running, there is only one program that runs for both the screens. In doing so, we intend that the same program and shader groups are executed for the screen representing the left and right eyes. The difference between the shading of the screens is that, besides the origin and the direction, when generating a ray and defining where it is on screen coordinates, it must account for an offset, given by the width of the leftmost screen.

By designing the path-tracing step in this way, there is a possibility for increasing the ray coherence from both cameras and, thus, an improvement in the memory locality of the BVH computations, discussed further in the chapter 5. With this render pass, the result is a noisy output image representing the path traced algorithm applied in the scene accounting for the foveation distribution, as shown in Figure 10.

#### 4.4 Non-homogeneous denoising pass

The non-homogeneous denoising pass is responsible for the image reconstruction. This stage receives the output from the path-tracing pass with the noisy image, along with the G-Buffers and the binary mask as an input. After the pass finishes, the output is the denoised image with less variance than the path-traced image, as shown in Figure 11.

Since the foveated noisy image has different sampling densities for every layer with varying amount of noise, one single denoising pass will decrease the variance in a different proportion to each of the layers. By adjusting the denoising step to three different layers,



Figure 10: The path-tracing pass generates a noisy output for both screens using the G-Buffers and the coordinates buffer as an input, accounting for the foreated distribution.

we split the denoising step in accordance to this particularity. This way, we differentiate the image in three layers, using the same layers used in the foveation distributed step, as shown in Figure 9. The pass applies the denoising process for each layer using the Edge-Avoiding À-Trous filter (DAMMERTZ et al., 2010) adapted to account for the foveation distribution. Since the filter works by weighting the noisy image, the world normal and the world position G-Buffers input against the neighbors' pixels, we need to filter with the binary mask as input. This mask works like a selection filter that decides which neighbor pixel is going to be accumulated for the final color of the current pixel being iterated. This works in a way that the sub-sampling does not darken and further increases the variance of the shaded pixel. Thus, from the original weight function

$$w(i,j) = w_{rt}(i,j) * w_n(i,j) * w_x(i,j)$$
(4.1)

with pixel positions i and j, where  $w_{rt}$  is the weight of the path-traced color,  $w_n$  is the weight of the world normal,  $w_x$  is the weight of the world position. The modified version is

$$w(i,j) = w_{rt}(i,j) * w_n(i,j) * w_x(i,j) * b(i,j)$$
(4.2)

with the added binary mask b(i,j) at the pixel.



Figure 11: The reconstructed image after the non-homogeneous denoising pass.

As in the original, the weights  $w_{rt}(i,j)$ ,  $w_n(i,j)$ , and  $w_x(i,j)$  are defined as the difference in the value of the neighbor pixel (i, j) and the current pixel (u,v) being denoised. So, e.g.,  $w_{rt}(i,j) = e^{-(|rt(i,j)-rt(u,v)|)}$  is the weight of the path-traced color. The other weights are computed similarly.

By processing the denoise pass only once on both sides of the screen, it is not possible to achieve a satisfactory noise reduction in the inner layer and let alone the middle and outer layers, given that each of the outer layers requires a different number of iterations in proportion to the respective sampling. Among the three layers, by following the foveation distribution and applying to the denoising configuration, the inner layer starts from the center of the screen to the radius  $r_0$  and requires M levels of iteration, the middle layer starts from radius  $r_0 + 1$  to the  $r_1$  and requires N levels, and the outer layer starts from radius  $r_1 + 1$  to the border of the viewport and requires P levels, following the inequality M < N < P.

Each iteration in a layer reconstructs only the area corresponding to that layer. It does so with a 5×5 kernel using a convolution mask based on the same cubic B-spline as described in (DAMMERTZ et al., 2010):  $(\frac{1}{16}, \frac{1}{4}, \frac{3}{8}, \frac{1}{4}, \frac{1}{16})$ , with the difference that it is extended to a two-dimensional kernel. Thus, in the borders of each layer, the neighbor pixels from other layers may be selected for the accumulation of the final shading color. The only exception is on the outer layer, where the neighbor pixels could be out of screen

or in the wrong side of the screen (in this case, these pixels are not selected), and the pixels that were not selected in according to the binary mask. With this, we can optimize the filter to be more efficient by leveraging the sub-sampling configuration from the foveated distribution.

The original Edge-Avoiding Å-Trous filter(DAMMERTZ et al., 2010) works in a way that it uses three values as references of the pixel, including the color of the noisy input, the normal of the surface represented in the pixel, and the world position of the surface. Then it computed the difference between the neighboring pixels (which can vary in according to the desired size of the kernel) and these reference values by using the dot product. These deltas, combined with a chosen constant for each of the values, result in the weight that each neighbor pixel have in the final contribution of the shaded color.

Note that an optional diffuse buffer multiplication is mentioned in the original filter that may improve the details that were blurred during this denoising pass. There are a couple of caveats in choosing to use this diffuse buffer: in the path-tracing pass, the computation of the illumination has to be done without the material evaluation, and it can only be done in case of diffuse surfaces.

To reduce some of the known artifacts originated from the Edge-Avoiding A-Trous filter, we use the modified version with a decreasing step-width (QI; WU; HE, 2012), instead of an increasing step-width in subsequent iterations as illustrated in Figure 12.

The output of this pass is the final image that the user will see in each frame being rendered. This pass is described in the form of a pseudocode by Algorithm 1.



Figure 12: Top illustration from the original method (DAMMERTZ et al., 2010) of the  $\dot{A}$ -Trous filter with 3 levels of iterations with increasing step-width, bottom illustration depicting an example of our adapter  $\dot{A}$ -Trous filter with 3 levels of iterations with decreasing step-width. Ours starts by taking into account the pixels at distance  $2^i$  (black dots) from the center pixel in the accumulation process, and decreases for each subsequent iteration. The blue dots are skipped by the  $\dot{A}$ -Trous filter and the gray dots are skipped by the binary mask modification.

Algorithm 1: Non-homogeneous denoising
<b>Input:</b> $M, N, P$ , binary mask $B$ , current color texture $rt$ , world normal texture $n$ ,
world position texture $x$ .
<b>Output:</b> Denoised texture $c$
Function computeDenoise(step, $r_a$ , $r_b$ ):
<b>foreach</b> paralel pixel $(u,v)$ in the texture $rt$ from $r_a$ to $r_b$ do
$sum \leftarrow 0$
$cum_w \leftarrow 0$
<b>foreach</b> selected neighbour $(i,j)$ in the texture $rt$ at the distance $2^{step}$ <b>do</b> $w(i,j) \leftarrow w_{rt}(i,j) * w_n(i,j) * w_x(i,j) * B(i,j)$ $sum \leftarrow sum + rt(i,j) * w(i,j) * kernel(i,j)$
$cum_{ev} \leftarrow cum_{ev} + w(i \ j) * kernel(i \ j)$
end
$c(u,v) \leftarrow sum/cum_w$
end
Function Main:
for $i \leftarrow M - 1$ to 0 do computeDenoise $(i, 0, r_0)$ ;
for $j \leftarrow N - 1$ to 0 do computeDenoise $(j, r_0, r_1)$ ;
for $k \leftarrow P - 1$ to 0 do computeDenoise $(k, r_1, r_{max})$ ;
return c

## **5** Implementation

This chapter details our implementation, which includes both the foveation distribution details, the path-tracing rendering sampling and details and the denoising configuration.

#### 5.1 Implementation

This work implemented the proposed rendering pipeline and its ray-tracing programs using the Microsoft DirectX APIs for the graphics shaders, the Valve OpenVR APIs (SELAN; LUDWIG; LEIBY, 2015) for the integration with the HMD, and imGui (CORNUT, 2014) for the GUI integration for the experiments. All of these libraries and APIs are inside the Falcor framework, developed by NVIDIA (BENTY et al., 2018).

The Falcor framework used is in a currently deprecated version (3.2.2). The reason for this choice is that, despite its lack of clear documentation, it integrates all the APIs mentioned above easily, enabling rapid prototyping and development for this project. With the support for two-screen rendering of an HMD device and support for DirectX ray tracing shaders, it is suitable for ray tracing applications rendered in an HMD.

The G-Buffers are computed based on (WYMAN et al., 2018) implementation, with modifications in order to support two screens. This modification is done to support each of the screens with a different view matrix representing the left and the right eye. With these two different matrices, the ray-generation shader traces a ray for each side of the screen, with reference to the pixel in each screen and its corresponding view matrix. It enables the computation of the G-Buffers of both screens by the end of a single rendering pass of the ray-tracing shaders, i.e., with a single run of the ray-tracing program instead of running the same program for each side of the screens separately. This improves the memory utilization of the triangle-ray intersection calculations in the RTX GPU, since there is ray coherence in both of the different cameras.

The computation of the foveation distribution, is made in a CPU function before

loading the scene, which is passed to the GPU via a constant buffer. This way, we can guarantee that there is no impact in the total performance of the rendering pipeline. The computation consists of the iterating over every pixel on the left side of the screen, verifying in which region one pixel belongs to, and then selecting the pixel. In the inner layer, every pixel is selected. In the middle and outer layer, a random toss with the probability p set to p = 0.5 define which pixel is going to be selected, following the linear decay mentioned in the previous chapter. In the current implementation, we set this probability for the two resolutions tested in our experiments: Full HD (960×1080 per screen) and the "near-4K" resolution of the Oculus Quest 2 (1832×1920 per screen). This results in the sample distribution cited in Table 2. After this process, we fill both the coordinate buffer and the binary mask.

Table 2: Region distribution used in the experiment for comparison, with its resolution, respective area size, samples, and spp density.

Resolution	Region	Radius (in pixels)	Area (in pixels)	Samples	$\operatorname{spp}$
FHD	Inner Middle Outer	144 288 -	65144 195432 776224	$260576 \\ 586296 \\ 1552448$	$\begin{array}{c} 4\\ 3\\ 2 \end{array}$
Quest 2	Inner Middle Outer	265 530 -	$220618 \\ 661855 \\ 2634967$	882472 1985565 5269934	$4 \\ 3 \\ 2$

The path-tracing pass is a modified version based on an existing implementation (WYMAN et al., 2018) of the diffuse illumination using a Lambertian model of reflection for the objects of the scene and also used for global illumination. In this case, the modification is done in order to generate the initial rays based on the coordinates buffer size and to trace the rays only through the coordinates given by the buffer. With this modification, we were able to generate only a controlled amount of rays in the initial ray generation shader, given the foreated distribution calculated in the previous step.

The implementation was tested with different spp densities, including the configuration that achieved satisfactory performance in the current hardware, fixed in 4 spp density for the inner layer. The middle layer has a sub sampling resulting in 3 spp, and the outer layer has a sub sampling resulting in 2 spp. These path tracing shaders, like as the G-Buffers shaders, were also modified so that they could compute the lighting in both screens in a single pass. Since the G-Buffers already account for both sides of the screens, this modification in the path tracing step was simplified, and it can be resumed to a mapping of the coordinates buffer to the pixel launch coordinates in the ray-generation shader. For the right side of the screen, this mapping also accounts for an offset related to the width of the left screen so that the pixel launch coordinates are correctly shifted to dispatch the rays in the GPU shader.

Our denoising step consists of iterating over the levels of denoising and dispatching a compute task to the GPU. As the compute shader is the same for all three layers, we only pass the required arguments, such as inner radius, outer radius, and the current step width. This implementation also includes an adaptation and translation of the GLSL shader provided by (DAMMERTZ et al., 2010) with some modifications. We changed the loop to a version introduced by Qi et al.(QI; WU; HE, 2012), with a decreasing step-width in each subsequent iteration of the levels of the execution of the shader, in order to reduce the artifacts near the edge of the scene objects. Here, we also included the modification in the path-tracing pass to account for both the left and the right sides of the image (left and right eye). Besides that, since this reconstruction was done in a compute shader, we can dispatch a separate task for every region. The computation of the denoising is done so that each thread calculates the weighted sum of its neighboring pixels in both sides of the image, by implementing an offset of the screen width. We avoid the pixels that should not be selected in either of the regions by selecting through our binary mask from the foveated distribution.

### **6** Results

This chapter details an exploration of our experiments, based on different configurations of input parameters. We give some of the results from our experiments in the form of both objective quality analysis and performance analysis through the pipeline. Finally, a brief discussion of the benchmarks is made, with a speedup analysis.

To present and discuss the results of this work, a series of experiments using different configurations was performed in a single hardware platform. We ran our experiments through an implementation with a machine with the specifications: Intel Core i7-3770S CPU @ 3.10GHz, 8.00 GB RAM, NVIDIA GeForce RTX 2080, Windows 10 OS.

We choose to measure the performance of our rendering pipeline using the GPU frame rendering time as the metric, in milliseconds. In doing so, we can diminish any possible interference from the varying OS system load or other background processes running on the machine (a concern when using the FPS metric) and also compare the performance by using a common metric between the experiments. Since the time of the foveated distribution step finishes before the beginning of the rendering pipeline, we did not measure the time it takes to compute this stage.

We render the scenes with the camera traversing a pre-defined path instead of a static frame of a single image. This is so that we can observe movement through different scene geometry, lighting and texture configurations. We measure the results over 1000 subsequent frames and take the average metric of how long it took to generate a frame for each scene. This amount of frames makes the different configurations varies in terms of how long it took to reach the 1000 frames, and also varies the amount of times the pre-defined path was run. Our target resolutions were set to Full HD (960×1080 per screen, totaling  $1920\times1080$ ) since it is a common resolution in most devices, and "near-4K" from the Quest 2 HMD ( $1832\times1920$  per screen, totaling  $3664\times1920$ ), which is a popular virtual-reality headset.

To have a variation of the features of a possible scene, we selected three different scenes

<u>Table 3:</u>	Scenes tested	and its characteristics.
Scene	Triangle Count	Light Count
Pink Room	n 786056	1 directional; 2 point lights
Sponza	262267	1 directional; 1 point lights
Forest	198541	80 point lights

to be used in our experimentation. Each scene has varying specifications of triangles and light count, as detailed in Table 3.

The first experiment is referred as *Base* for the comparison between our experiments, since it is our baseline for speedup calculations. This configuration was chosen as a reference for the performance analysis because it does not include neither a foveation distribution nor our non-homogeneous denoising. Instead, the pipeline in these Base experiments renders a full-screen denoising pass to a path-tracing with sampling of 4 spp across the entirety of the dual-screen viewport. The amount of denoising iterations is configured with 5 iterations in the full-screen denoising pass, i.e., M = 5 in this configuration. Figure 13 shows the image rendered as a result of this configuration.

In the subsequent experiments referenced as Non-Homogeneous 3-Layer Denoising (NH3LD), we applied the proposed rendering pipeline optimizations. The foveation distribution has three layers as concentric circles, and also coincides with the regions split for the non-homogeneous denoising pass. Following the split in Table 2, for the sake of comparison, we also set the inner layer with a path-tracing sampling density of 4 spp, decreasing the sampling in the middle and outer layers. The amount of denoising iterations is configured with 2 iterations for the inner layer, 3 iterations for the middle layer and 5 iterations for the outer layer, i.e., M = 2, N = 3, P = 5 in this configuration. Figure 14 shows the image rendered as a result of this configuration.

We also consider that the outer layer in the NH3LD experiment has the greater number of denoising levels. Analyzing the averaged millisecond per frame performance in our experiments, we can compare it to the *Base* experiment. In the same level of denoising, the proposed solution with the non-homogeneous denoising optimization applied to three layers shows a speedup of up to 1.35.

A third experiment tested a different split in the non-homogeneous denoising pass, using two layers with different levels of denoising instead of the three-layer design used in the previous experiment. We reference it as Non-Homogeneous 2-Layer Denoising (NH2LD). Note that the foreated distribution still has the three-layer as described in Table 2. For the denoising split, the inner layer coincides with the foveated distribution as well. The outer layer is a join region with the middle and outer layer from the foveated distribution. In other words, the outer layer in the denoising split begins from the circles with radius  $r_0 + 1$  and goes until the edge of the viewport. The amount of denoising iterations is configured with 3 iterations for the inner layer and 5 iterations for the outer layer, i.e., M = 3, N = 5 in this configuration. Figure 15 shows the image rendered as a result of this configuration.

We again consider that the outer layer in the *NH2LD* experiment has the greater number of denoising levels. Analyzing the performance in our experiments with the configuration of the same level of denoising in the first experiment, it is possible to see that the proposed solution with the non-homogeneous denoising optimization applied to two layers achieves a speedup of up to 1.33. The detailed performance measurements with the total averaged milliseconds per frame and the speedup are in Table 4.

Table 4: Performance metrics of Base, NH2LD and NH3LD experiments, in averaged milliseconds per frame for our GPU implementation.

Resolution	Scene	Renderer	Total time (ms)	Speedup factor
		Base	16.26	
	Pink Room	NH2LD	13.99	1.16
		NH3LD	12.81	1.26
		Base	31.01	_
FHD	Sponza	NH2LD	23.28	1.33
		NH3LD	22.95	1.35
		Base	22.84	_
	Forest	NH2LD	18.46	1.23
		NH3LD	17.54	1.30
		Base	45.32	_
	Pink Room	NH2LD	40.58	1.11
		NH3LD	39.19	1.15
		Base	99.27	_
Quest 2	Sponza	NH2LD	75.52	1.31
		NH3LD	74.62	1.33
		Base	73.66	
	Forest	NH2LD	60.24	1.22
		NH3LD	59.03	1.24

For an analysis of objective quality, we rendered the same scenes without the foveation distribution step optimization. We also increased the number of samples for 1024 spp temporally accumulated across several frames, where no denoising was applied to these reference images. This configuration without denoising and with increased sampling makes the three different scenes into our ground truth for the analysis. Figure 16 shows the image rendered as a result of this configuration. In order to measure the error between the reference rendered images and the optimized by our rendering pipeline, we used Root-Mean-Square Error (RMSE), Structural Similarity (SSIM) (WANG et al., 2004), and Peak Signal-To-Noise Ratio (PSNR) metrics.

As show in Table 5, we see a minor variance in the metrics of both our experiments of non-homogeneous denoising in comparison to the full-screen denoise in the *Base* experiment.

To make the comparison between the images easier, the Figure 17 has the four images rendered with our pipeline with the same section being cropped. Three of the images are from our experiments (*Base*, *NH3LD*, and *NH2LD*) and one from the reference image.

Scenes	Rendering Pipeline	RMSE (%)	SSIM	PSNR (dB)
Pink Room	Base NH2LD NH3LD	$3.2862 \\ 3.4374 \\ 3.5997$	0.986438 0.985774 0.987343	28.659183 28.030232 27.619445
Sponza	Base NH2LD NH3LD	5.7633 5.9252 5.9029	$\begin{array}{c} 0.918678 \\ 0.913988 \\ 0.913633 \end{array}$	21.879093 21.420517 21.468767
Forest	Base NH2LD NH3LD	$3.7991 \\ 3.8649 \\ 4.1787$	$\begin{array}{c} 0.813191 \\ 0.795324 \\ 0.778536 \end{array}$	$\begin{array}{c} 22.339938\\ 22.37879\\ 21.69204 \end{array}$

Table 5: Objective quality comparison with RMSE, SSIM and PSNR metrics

Another objective analysis was made in the form of an image, using an algorithm developed by NVIDIA Research (ANDERSSON et al., 2020) and the available executable build of the source code at (LELONG, 2020). This is a difference image evaluator, that given a reference image, it compares to another image and produces a heatmap with an approximated perceived difference between the two images.

To make the comparison of each heatmap easier, the Figure 18 align side by side the image differences between our three experiments (*Base, NH3LD*, and *NH2LD*) and the reference image. The brightest spots in the figure represent the biggest difference between the experiment and the reference image, and, thus, the darkest spot represents the lowest difference.



Figure 13: Rendering of Sponza scene in the Base experiment.



Figure 14: Rendering of Sponza scene in the  $\it NH3LD$  experiment.



Figure 15: Rendering of Sponza scene in the  $\it NH2LD$  experiment.



Figure 16: Rendering of Sponza scene as a graphical reference using 1024 spp.



Figure 17: From left to right, top: reference image, image from Base experiment; bottom: image from NH2LD experiment, image from NH3LD experiment.



Figure 18: Heatmap of the difference between the Base experiment and the reference image.

# 7 Conclusion

This work presented a novel real-time rendering pipeline for virtual reality devices, using non-homogeneous denoising scaled according to foveated regions. Our proposed pipeline was able to present the same effects of visual realism achieved when using regular denoising, but reducing the number of rays and increasing performance. We leveraged the foveation distribution created in the CPU, stored and passed to the GPU in a coordinates buffer with different spatial sampling in each layer of the visual field. For so, it was important to decrease the initial ray generation in the GPU shader and its bounces. Adding non-homogeneous denoising also enabled a decrease in the load of work for the reconstruction steps, with different levels to apply the Edge-Avoiding À Trous (DAMMERTZ et al., 2010) algorithm for the corresponding denoising layers.

Our experiments were executed with this optimized pipeline in several configurations against a non-optimized one. These configurations used different implementation details such as the size of layers, number of layers, and levels of denoising. These experiments were able to show a speedup in rendering time performance of up to 1.35.

#### 7.1 Limitations

Despite our advancements in the denoising path-tracing images in foveated rendering field, there are a couple of limitations in our work and implementation.

Too many lights. In the case of too many lights in a scene, specially emitted lights, the denoising ends up not clearing the image in a way that maintains the details of the geometry and texture. This can be seen in Figure 19.

*Falcor framework.* While this framework has several abstractions that make the development faster, the current version hides a lot of stages for the end user in a way that it makes it very difficult to tune it for performance. As a first improvement on this work, one future implementation could use other frameworks or engines or even the libraries



Figure 19: Forest scene. Left: the image rendered with 1024 spp as in the *Base* experiment. Right: the image rendered in our non-homogeneous denoising configuration.

that Falcor gathers, with the caveat of integrating all of these.

#### 7.2 Future Works

This work can be further explored in a series of different directions. This section lists some of these enhancements that we had a glimpse during our research and implementation.

Other wavelets filters. In future works, we intend to explore how to apply the nonhomogeneous denoising with other known denoisers in the latest step of the rendering pipeline, such as SVGF (SCHIED; KAPLANYAN, et al., 2017) or BMFR (KOSKELA; IMMONEN, et al., 2019).

User study. Another extension is by conducting a user study to test the human perception of the graphical quality. Despite the objective quality analysis in the chapter 6, there is a possibility that we can enhance the performance in order to a perceived quality through different implementation settings, to further optimize our denoising and path-tracing parameters.

Log-polar mapping. Adding the log-polar coordinates mapping to the rendering pipeline brings difficulties for the denoising and post-processing phase, specially because of aliasing and flickering. Combining the works of log-polar mapping (MENG et al., 2018) or Visual-Polar (KOSKELA; LOTVONEN, et al., 2019) space and our denoiser is a possibility.

*Path-tracing improvements.* Rendering can always be improved, specially in the path-tracing phase. While Lambertian is a good approach to diffuse surfaces, we would like to include other types of materials that encompass specular surfaces, such as GGX (WAL-TER et al., 2007; BURLEY; STUDIOS, 2012).

# REFERÊNCIAS

ANDERSSON, Pontus et al. : A Difference Evaluator for Alternating Images.
Proceedings of the ACM on Computer Graphics and Interactive Techniques,
v. 3, n. 2, 15:1–15:23, 2020.

BARRÉ-BRISEBOIS, Colin et al. Hybrid rendering for real-time ray tracing. In: RAY Tracing Gems. [S.l.]: Springer, 2019. p. 437–473.

BENTY, Nir et al. The Falcor Rendering Framework. [S.l.: s.n.], May 2018. https://github.com/NVIDIAGameWorks/Falcor. Available from: <https://github.com/NVIDIAGameWorks/Falcor>.

BURLEY, Brent; STUDIOS, Walt Disney Animation. Physically-based shading at disney. In: VOL. 2012. ACM SIGGRAPH. [S.l.: s.n.], 2012. v. 2012, p. 1–7.

COOK, Robert L; PORTER, Thomas; CARPENTER, Loren. Distributed ray tracing. In: PROCEEDINGS of the 11th annual conference on Computer graphics and interactive techniques. [S.l.: s.n.], 1984. p. 137–145.

CORNUT, Omar. Dear ImGui. [S.l.: s.n.], 2014. https://github.com/ocornut/imgui. Available from: <https://github.com/ocornut/imgui>.

DAMMERTZ, Holger et al. Edge-avoiding a-trous wavelet transform for fast global illumination filtering. In: CITESEER. PROCEEDINGS of the Conference on High Performance Graphics. [S.l.: s.n.], 2010. p. 67–75.

DUCHOWSKI, Andrew T; ÇÖLTEKIN, Arzu. Foveated gaze-contingent displays for peripheral LOD management, 3D visualization, and stereo imaging. **ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)**, ACM New York, NY, USA, v. 3, n. 4, p. 1–18, 2007.

GUENTER, Brian et al. Foveated 3D graphics. ACM Transactions on Graphics (TOG), ACM New York, NY, USA, v. 31, n. 6, p. 1–10, 2012.

HUA, Binh-Son et al. A Survey on Gradient-Domain Rendering. In: WILEY ONLINE LIBRARY, 2. COMPUTER Graphics Forum. [S.l.: s.n.], 2019. v. 38, p. 455–472.

INTEL®. Intel® Open Image Denoise. [S.l.: s.n.], Jan. 2019. Available from: <a href="https://www.openimagedenoise.org/">https://www.openimagedenoise.org/</a>>.

KAJIYA, James T. The rendering equation. In: PROCEEDINGS of the 13th annual conference on Computer graphics and interactive techniques. [S.l.: s.n.], 1986. p. 143–150.

KALANTARI, Nima Khademi; SEN, Pradeep. Removing the noise in Monte Carlo rendering with general image denoising algorithms. In: WILEY ONLINE LIBRARY, 2pt1. COMPUTER Graphics Forum. [S.l.: s.n.], 2013. v. 32, p. 93–102.

KAPLANYAN, Anton S et al. DeepFovea: neural reconstruction for foveated rendering and video compression using learned statistics of natural videos. **ACM Transactions on Graphics (TOG)**, ACM New York, NY, USA, v. 38, n. 6, p. 1–13, 2019.

KILGARIFF, Emmett et al. **NVIDIA Turing Architecture in-depth**. [S.l.: s.n.], Sept. 2018. Available from: <https://developer.nvidia.com/blog/nvidia-turingarchitecture-in-depth/>.

KOCH, Daniel et al. **Ray Tracing In Vulkan** — **khronos.org**. [S.l.: s.n.], 2020. [Accessed 15-Sep-2022]. Available from: <<u>https://www.khronos.org/blog/ray-</u> tracing-in-vulkan#blog\_Ray\_Tracing\_Pipelines>.

KOSKELA, Matias; IMMONEN, Kalle, et al. Blockwise multi-order feature regression for real-time path-tracing reconstruction. **ACM Transactions on Graphics (TOG)**, ACM New York, NY, USA, v. 38, n. 5, p. 1–14, 2019.

KOSKELA, Matias; LOTVONEN, Atro, et al. Foveated real-time path tracing in visual-polar space. In: THE EUROGRAPHICS ASSOCIATION. PROCEEDINGS of 30th Eurographics Symposium on Rendering. [S.l.: s.n.], 2019.

LEFOHN, Aaron. Learning and Rendering Dynamic Global Illumination with One Tiny Neural Network in Real-Time - Nvidia Research. [S.l.: s.n.], 2021. [Accessed 17-Sep-2022]. Available from: <a href="https://www.separativecommunication-style">https://www.separativecommunication-style="text-align: center;">https://www.separativecommunication-style="text-align: center;"/>

//developer.nvidia.com/blog/nvidia-research-learning-and-renderingdynamic-global-illumination-with-one-tiny-neural-network-in-real-time/>.

LELONG, Nicolas. Rotoglup/Nvidia-flip-cpp. [S.l.: s.n.], 2020. [Accessed 07-Nov-2022]. Available from: <a href="https://github.com/rotoglup/nvidia-flip-cpp">https://github.com/rotoglup/nvidia-flip-cpp</a>.

LIN, Fang-Cheng et al. SNR analysis of high-frequency steady-state visual evoked potentials from the foveal and extrafoveal regions of Human Retina. In: v. 2012, p. 1810–4. DOI: 10.1109/EMBC.2012.6346302.

#### REFERÊNCIAS

MENG, Xiaoxu et al. Kernel foveated rendering. **Proceedings of the ACM on Computer Graphics and Interactive Techniques**, ACM New York, NY, USA, v. 1, n. 1, p. 1–20, 2018.

MOHANTO, Bipul et al. An integrative view of foveated rendering. Computers & Graphics, Elsevier, v. 102, p. 474–501, 2022.

NVIDIA. NVIDIA OptiX<sup>™</sup> AI-Accelerated Denoiser. [S.l.: s.n.], July 2017. Available from: <https://developer.nvidia.com/optix-denoiser>.

OREN, Michael; NAYAR, Shree K. Generalization of Lambert's reflectance model. In: PROCEEDINGS of the 21st annual conference on Computer graphics and interactive techniques. [S.l.: s.n.], 1994. p. 239–246.

PARIS, Sylvain; DURAND, Frédo. A fast approximation of the bilateral filter using a signal processing approach. In: SPRINGER. EUROPEAN conference on computer vision. [S.l.: s.n.], 2006. p. 568–580.

QI, Baojun; WU, Tao; HE, Hangen. A novel edge-aware Å-Trous filter for single image dehazing. In: IEEE. 2012 IEEE International Conference on Information Science and Technology. [S.l.: s.n.], 2012. p. 861–865.

REDDY, Martin; REDDY, Martin. The Development and Evaluation of a Model of Visual Acuity for Computer-Generated Imagery. [S.l.], 1997.

SABINO, Thales Luis et al. A hybrid GPU rasterized and ray traced rendering pipeline for real time rendering of per pixel effects. In: SPRINGER. INTERNATIONAL Conference on Entertainment Computing. [S.l.: s.n.], 2012. p. 292–305.

SCHIED, Christoph; KAPLANYAN, Anton, et al. Spatiotemporal variance-guided filtering: real-time reconstruction for path-traced global illumination. In: PROCEEDINGS of High Performance Graphics. [S.l.: s.n.], 2017. p. 1–12.

SCHIED, Christoph; PETERS, Christoph; DACHSBACHER, Carsten. Gradient estimation for real-time adaptive temporal filtering. **Proceedings of the ACM on Computer Graphics and Interactive Techniques**, ACM New York, NY, USA, v. 1,

n. 2, p. 1–16, 2018.

SELAN, Jeremy; LUDWIG, Joe; LEIBY, Aaron. Valvesoftware/openvr: Openvr SDK. [S.l.: s.n.], 2015. https://github.com/ValveSoftware/openvr. Available from: <https://github.com/ValveSoftware/openvr>. VEACH, Eric; GUIBAS, Leonidas J. Optimally combining sampling techniques for Monte Carlo rendering. In: PROCEEDINGS of the 22nd annual conference on Computer graphics and interactive techniques. [S.l.: s.n.], 1995. p. 419–428.

WALTER, Bruce et al. Microfacet Models for Refraction through Rough Surfaces. Rendering techniques, Citeseer, v. 2007, 18th, 2007.

WANG, Zhou et al. Image quality assessment: from error visibility to structural similarity. **IEEE transactions on image processing**, IEEE, v. 13, n. 4, p. 600–612, 2004.

WEIER, Martin et al. Foveated real-time ray tracing for head-mounted displays. In:WILEY ONLINE LIBRARY, 7. COMPUTER Graphics Forum. [S.l.: s.n.], 2016. v. 35,p. 289–298.

WHITTED, Turner. An improved illumination model for shaded display. In: PROCEEDINGS of the 6th annual conference on Computer graphics and interactive techniques. [S.l.: s.n.], 1979. p. 14.

WYMAN, Chris et al. Introduction to DirectX Raytracing. In: ACM SIGGRAPH 2018 Courses. Vancouver, British Columbia: [s.n.], Aug. 2018.

ZWICKER, Matthias et al. Recent advances in adaptive sampling and reconstruction for Monte Carlo rendering. In: WILEY ONLINE LIBRARY, 2. COMPUTER graphics forum. [S.l.: s.n.], 2015. v. 34, p. 667–681.