Implicit and Foveated Techniques for Real-time Computer Graphics Rendering

by

Andreas Polychronakis

A thesis submitted in partial fulfillment for
the degree of Doctor of Philosophy

School of Electrical and Computer Engineering
Technical University of Crete
Greece

Chania, February 2025

Declaration of Authorship

I, Andreas Polychronakis, declare that this thesis titled, "Implicit and Foveated Rendering Techniques for Real-time Computer Graphics Rendering" and the work presented in it are my own. I confirm that:

- This work was done wholly while in candidature for a research degree at this University.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.

Jury

3-member Committee

Prof. Katerina Mania	Technical University of Crete
Prof. Antonios Deligiannakis	Technical University of Crete
Assoc. Prof. George Alex Koulieris	Durham University

Examiners

Prof. Ioannis Fudos	University of Ioannina
Assoc. Prof. Vangelis Kalogerakis	Technical University of Crete
Prof. Michail G. Lagoudakis	Technical University of Crete
Prof. Costas Balas	Technical University of Crete

Abstract

Real-time rendering is a cornerstone of modern interactive applications, such as virtual reality, gaming, and simulations, where high visual fidelity and low latency are critical to user immersion. However, achieving these goals remains a significant challenge due to the computational complexity of rendering techniques and the limitations of existing methods. Traditional approaches often struggle to balance performance and quality, particularly in scenarios requiring dynamic adaptation to user attention or the rendering of complex geometries. These challenges are especially pronounced in foveated rendering, which aims to optimize resource allocation by prioritizing high-quality rendering in the user's focal region, and in sphere tracing of Signed Distance Functions (SDFs), a technique used to render intricate implicit surfaces. Current methods frequently suffer from visual artifacts, inefficient resource utilization, and an inability to adapt to real-time demands, highlighting the need for innovative solutions that bridge the gap between computational efficiency and perceptual quality.

This doctoral dissertation presents innovative approaches designed to enhance the efficiency of real-time rendering techniques, focusing on foveated rendering and sphere tracing of Signed Distance Functions (SDFs). The research addresses the limitations of existing rendering methods that rely heavily on low-level image features and often suffer from artifacts, when optimizing computational resource allocation.

First, we present an emulated foveated path tracing framework that establishes thresholds for imperceptible image manipulations based on user gaze. Our perceptual studies yield varying eccentricity thresholds for foveated performance, highlighting the impact of experimental methodologies on user sensitivity to visual changes. The results indicate potential computational complexity reductions of at least 2× - 3× in path tracing performance through foveated rendering methods.

Second, we introduce the Foveated Inverted Pyramid Rendering (FIPR) system, which optimizes rendering quality based on user gaze. By employing a multi-scale inverted pyramid structure, we leverage low-resolution renderings to incrementally refine ray distances, dramatically reducing the overall ray step count and enabling

efficient rendering of complex scenes in virtual reality. Our method supports 16× super-sample anti-aliasing, while maintaining imperceptible image quality transitions, even in peripheral vision.

Lastly, we develop a novel rapid rendering pipeline for sphere tracing SDFs that significantly reduces the overall ray step count using ultra-low resolution renderings, while minimizing artifacts. By employing a single low-resolution buffer and scaling SDFs within this buffer, our method ensures the visibility of small features while enabling earlier ray termination for high-cost surface edges. This approach yields a substantial performance improvement, achieving speedups exceeding 3× compared to traditional methods.

Collectively, these contributions advance the field of real-time graphics rendering, providing valuable insights into optimizing rendering techniques, while enhancing the user experience in interactive applications, such as gaming and simulations.

Περίληψη

Η σχεδίαση γραφικών σε πραγματικό χρόνο (real-time rendering) αποτελεί έναν από τους βασικούς πυλώνες των σύγχρονων διαδραστικών εφαρμογών, όπως η εικονική πραγματικότητα (virtual reality), τα παιχνίδια (gaming) και οι προσομοιώσεις (simulations), όπου η υψηλή οπτική πιστότητα (visual fidelity) και η χαμηλή καθυστέρηση (low latency) είναι κρίσιμες για την εμβύθιση του χρήστη (user immersion). Ωστόσο, η επίτευξη αυτών των στόχων παραμένει μια σημαντική πρόκληση λόγω της υπολογιστικής πολυπλοκότητας των τεχνικών rendering και των περιορισμών των υφιστάμενων μεθόδων. Οι παραδοσιακές προσεγγίσεις συχνά δυσκολεύονται να ισορροπήσουν την απόδοση (performance) και την ποιότητα, ειδικά σε σενάρια που απαιτούν δυναμική προσαρμογή στην προσοχή του χρήστη (user attention) ή την απόδοση πολύπλοκων γεωμετριών (complex geometries).

Αυτές οι προκλήσεις είναι ιδιαίτερα έντονες στο foveated rendering, η οποία στοχεύει να βελτιστοποιήσει την κατανομή των πόρων (resource allocation) δίνοντας προτεραιότητα στην υψηλή ποιότητα σχεδίασης στην περιοχή εστίασης του χρήστη (focal region), καθώς και στον αλγοριθμό sphere tracing των Signed Distance Functions - SDFs, μια μέθοδο που χρησιμοποιείται για την απόδοση περίπλοκων υπονοούμενων επιφανειών (implicit surfaces). Οι υπάρχουσες μέθοδοι συχνά υποφέρουν από οπτικά artifacts, αναποτελεσματική χρήση πόρων (inefficient resource utilization) και αδυναμία προσαρμογής στις απαιτήσεις του πραγματικού χρόνου (real-time demands), υπογραμμίζοντας την ανάγκη για καινοτόμες λύσεις που γεφυρώνουν το χάσμα μεταξύ υπολογιστικής αποδοτικότητας (computational efficiency) και αντίληψης ποιότητας (perceptual quality).

Αυτή η διατριβή παρουσιάζει καινοτόμες προσεγγίσεις που έχουν σχεδιαστεί για να ενισχύσουν την αποδοτικότητα των αλγορίθμων γραφικών σε πραγματικό χρόνο, λαμβάνοντας υπόψη την ανθρώπινη όραση στην παραγωγή εικόνων (foveated rendering) και τον αλγόριθμο ιχνηλάτησης σφαιρών (sphere tracing), για επιφάνειες που περιγράφονται από Συναρτήσεις Προσημασμένης Απόστασης (Signed Distance Functions - SDFs). Η έρευνα αντιμετωπίζει τους περιορισμούς των υφιστάμενων μεθόδων απόδοσης που βασίζονται σε μεγάλο βαθμό σε χαρακτηριστικά από εικόνες

χαμηλή ανάλυσης και συχνά υποφέρουν από artifacts όταν γίνεται βελτιστοποίηση της κατανομής των υπολογιστικών πόρων με βάση την ανθρώπινη όραση.

Αρχικά, παρουσιάζουμε ένα εξομοιωτή απόδοσης για τον αλγόριθμό path-tracing με βάση την ανθρώπινη όραση (emulated foveated path tracing) που καθορίζει όρια στα οποία τα artifacts που παράγονται είναι μη αντιληπτά από τον χρήστη. Οι μετρήσεις που πραγματοποιήθηκαν στους χρήστες αποφέρουν διαφορετικά όρια εκκεντρότητας για την απόδοση με βάση την όραση, υπογραμμίζοντας την επίδραση των πειραματικών μεθοδολογιών στην ευαισθησία των χρηστών στις οπτικές αλλαγές. Τα αποτελέσματα υποδεικνύουν πιθανές μειώσεις της υπολογιστικής πολυπλοκότητας τουλάχιστον 2x-3x στον αλγόριθμό path-tracing με βάση την ανθρώπινη όραση.

Στην συνέχεια, προτείνουμε ένα σύστημα απόδοσης γραφικών Foveated Inverted Pyramid Rendering, το οποίο διαφοροποιεί την ποιότητα της παραγόμενης εικόνας με βάση πού εστιάζει ο χρήστης στην οθόνη. Χρησιμοποιώντας μια δομή ανεστραμμένης πυραμίδας πολλαπλής κλίμακας, αξιοποιούμε αποδόσεις (renderings) χαμηλής ανάλυσης για να βελτιώσουμε σταδιακά τις αποστάσεις των ακτινών, μειώνοντας δραματικά τον συνολικό αριθμό βημάτων που πραγματοποιούν οι ακτίνες και επιτρέποντας την αποτελεσματική απόδοση σύνθετων σκηνών στην εικονική πραγματικότητα. Η μέθοδός μας έχει την δυνατότητα για την χρήση anti-aliasing έως 16 δείγματα ανά pixel διατηρώντας υψηλό ρυθμό παραγωγής εικόνας ενεργοποιώντας το anti-aliasing σε συγκείμενες περιοχές της εικόνας, ενώ διατηρεί ανεπαίσθητες μεταβάσεις ποιότητας εικόνας ακόμη και στην περιφερειακή όραση.

Τέλος, αναπτύσσουμε μια νέα γρήγορη διαδικασία για την ιχνηλάτηση σφαιρών για SDFs που μειώνει σημαντικά τον συνολικό αριθμό βημάτων των ακτίνων χρησιμοποιώντας εικόνες εξαιρετικά χαμηλής ανάλυσης που παράγονται με βάση την ιχνηλάτηση σφαιρών, ενώ ελαχιστοποιεί τα artifacts. Χρησιμοποιώντας ένα μόνο buffer χαμηλής ανάλυσης και κλιμακώνοντας τις SDFs μέσα σε αυτό το buffer, η μέθοδός μας διασφαλίζει την ορατότητα μικρών χαρακτηριστικών ενώ επιτρέπει τον τερματισμό της διαδικασίας ανίχνευσης σφαιρών όταν αυτές περνάνε κοντά από τις γωνίες των επιφανειών. Αυτή η προσέγγιση αποφέρει μια σημαντική βελτίωση στην απόδοση, επιτυγχάνοντας επιταχύνσεις που ξεπερνούν το 3x σε σύγκριση με τις παραδοσιακές μεθόδους.

Συλλογικά, αυτές οι συνεισφορές προωθούν τον τομέα της απόδοσης γραφικών σε πραγματικό χρόνο, παρέχοντας πολύτιμες γνώσεις για τη βελτιστοποίηση των τεχνικών απόδοσης ενώ βελτιώνουν την εμπειρία του χρήστη σε διαδραστικές εφαρμογές όπως τα παιχνίδια και οι προσομοιώσεις.

Acknowledgment

I would like to express my deepest gratitude to my supervisor, Katerina Mania. Her unwavering support and insightful guidance throughout my PhD journey have been instrumental in my research on rendering algorithms. Katerina's constructive feedback and encouragement helped me navigate the challenges I faced and kept me motivated. I deeply appreciate her mentorship and the opportunity to learn from her expertise.

I would also like to extend my heartfelt thanks to my co-supervisor, George Alex Koulieris, for his trust and support. I am particularly thankful to George for his ingenious suggestions and innovative approaches to the challenges I faced. His remote support and availability have made navigating this path much smoother.

My appreciation also extends to Gregory D. and Minas K., whose assistance and camaraderie in the lab created an inspiring environment. Their friendship, patience, and understanding, especially during my most challenging moments, have been invaluable.

I am also grateful to my friends—Magda A., Manos A., Zisis C., Aneza D., Evaggelos G., Ardit N., Andreas S., and Konstantinos S.—for their constant encouragement, collaboration on numerous projects, and the joyous moments we shared throughout my Bachelor's and PhD studies. These experiences have enriched my life and provided me with cherished memories.

I owe a profound debt of gratitude to my family for their unwavering support and patience during the stressful periods of my thesis. Their encouragement has served as a steady foundation, enabling me to persevere and succeed.

Lastly, I would like to extend my heartfelt thanks to my life partner, Vasilia Dionysia Rapsomaniki. Her constant encouragement, love, and belief in me have been a source of strength, helping me to overcome the challenges of my studies with renewed motivation and hope.

This doctoral thesis forms part of the project 3D4DEPLHI, co-financed by the European Union and Greek funds through the Operational Program Competitiveness, Entrepreneurship and Innovation, under the call 'Specific Actions, Open Innovation for Culture' (project code: Τ6ΥΒΠ-00190)

This doctoral thesis forms part of the project SUN, funded by the European Union through the Horizon Europe research and innovation program under grant agreement No 101092612.

Table of Contents

List of Figures

List of Tables

Publications

- Polychronakis, A., Koulieris, G. A., & Mania, K. (2024). Skipping Spheres: SDF Scaling & Early Ray Termination for Fast Sphere Tracing. In Proceedings of the 42th Eurographics Conference on Computer Graphics & Visual Computing (CGVC 2024), the Eurographics Association.
- Polychronakis, A., Koulieris, G. A., & Mania, K. (2023). An Inverted Pyramid Acceleration Structure Guiding Foveated Sphere Tracing for Implicit Surfaces in VR. In Proceedings of the 34th Eurographics Symposium on Rendering (EGSR 2023), the Eurographics Association.
- Polychronakis, A., Koulieris, G. A., & Mania, K. (2021, November). Emulating foveated path tracing. In Proceedings of the 14th ACM SIGGRAPH Conference on Motion, Interaction and Games (pp. 1-9).
- Mania, K., McNamara, A., & Polychronakis, A. (2021). Gaze-aware displays and interaction. In ACM SIGGRAPH 2021 Courses (pp. 1-67).

Chapter 1

Introduction

The challenges in real-time rendering arise from the limitations of existing rendering techniques, particularly in the context of foveated rendering [4] and sphere tracing [5] of Signed Distance Fields (SDFs). Current methods often fall short in efficiently predicting user gaze and managing computational resources effectively, primarily because they rely heavily on low-level image features such as luminance and contrast [6] without adequately considering the high-level contextual factors that influence visual attention like edges. This oversight results in a failure to optimize rendering in non-attended scene regions, consequently diminishing the potential for enhanced performance in applications such as gaming and simulations.

This dissertation addresses these issues by presenting three innovative approaches aimed at refining rendering methodologies. The first approach introduces a perceptual sandbox that emulates foveated path tracing techniques to establish thresholds for imperceptible quality reductions based on eccentricity angles. The second establishes an Inverted Pyramid Rendering pipeline for sphere tracing to dynamically adjust rendering quality according to user gaze, effectively balancing performance and visual fidelity. Lastly, the third approach employs SDF scaling and early ray termination techniques to significantly reduce computational steps while preserving detail. Through these contributions, we aim to enhance user experience in interactive environments by developing advanced, context-aware rendering solutions.

This chapter provides detailed motivation in addition to a description of our novel contributions. We also describe the structure of this thesis.

1.1 Problem Statement

Despite the advancements in rendering technology, current methods that utilize foveated rendering [4, 7] or sphere tracing techniques [8, 9, 10] still struggle with significant limitations. Traditional foveated rendering methods aim to optimize performance by delivering higher quality in regions of interest while lowering the quality of peripheral areas. However, the effectiveness of these methods is undermined by several prevalent issues. In particular, visual artifacts such as aliasing [4], blurriness [11], and distortion may arise in regions where the level of detail is decreased. These artifacts can severely detract from the immersive experience users expect, particularly in interactive environments where precision and realism are paramount. Moreover, existing foveated rendering techniques often require complex calibration processes that can complicate implementation in dynamic scenarios [12].

The core challenge with foveated rendering lies in rendering the areas of the screen that correspond to the periphery of our vision at the lowest possible quality while avoiding a break in user immersion due to this reduced rendering quality. Most existing foveation models rely on linear approximations [13] of visual acuity to describe the decline in visual acuity as we move from the fovea to peripheral vision. Additionally, while some foveated methods utilize higher-level salience factors, such as luminance [6], to develop their models, they often fail to account for scene geometries where peripheral vision remains sensitive [14], resulting in suboptimal rendering and subsequent inaccuracies in quality.

On the other hand, sphere tracing—a technique often employed for rendering complex geometries defined by SDFs—faces its own significant challenges related to computational efficiency. While sphere tracing offers a systematic method for rendering implicit surfaces, it inherently demands high computational resources, especially when approaching sharp boundaries in complex scenes [5, 15]. The computational cost skyrockets as the rays must adaptively reduce their step sizes for accurate convergence near edges, resulting in extended processing times that can exceed the requirements for real-time rendering applications. Unlike popular rendering paradigms such as ray tracing, which benefits from specialized hardware acceleration, sphere tracing remains under-optimized within most GPU architectures, leading to persistent inefficiencies during rendering operations [16].

The existing sphere tracing techniques often introduce a trade-off between rendering quality and performance, with many methods implementing dynamic step size adjustments [8, 10] or approximate bounds for surfaces [9] approaches to alleviate the computational burden. However, these methods frequently rely on heuristics that may not be well suited to all rendering scenarios, resulting in inconsistent per-

formance and an inability to handle dynamic scene changes effectively. Additionally, the reliance on predetermined scene parameters can limit the flexibility necessary for interactive applications where real-time decisions about rendering depth and quality must be made.

There is an increasing demand for rendering systems that can operate within the constraints of real-time performance [17] while still delivering visually compelling images capable of engaging users in immersive experiences. Many current rendering techniques either compromise quality in peripheral areas in a way that is visually jarring or require extensive manual configuration that detracts from their usability in dynamic applications. As user expectations for graphic fidelity and interactivity rise, the necessity for novel approaches that efficiently utilize the perceptual characteristics of human vision alongside advanced computational techniques becomes increasingly critical.

In this context, the challenges of implementing effective foveated rendering techniques and optimizing sphere tracing performance present a significant barrier to achieving high-quality, immersive, and interactive graphics. This dissertation aims to address these critical challenges through innovative solutions that integrate perceptually informed strategies with advanced rendering methodologies, providing significant advancements in rendering capabilities while meeting the demands of modern applications.

1.2 Contributions

This thesis presents novel contributions to the field of foveated rendering, specifically through the development of advanced techniques that leverage human visual perception to enhance rendering efficiency and quality. The specific contributions of this research include:

- **Foveated Path Tracing Framework:** We introduce an emulated foveated path tracer that adjusts rendering parameters based on eccentricity, determining thresholds for imperceptible foveated path tracing. By employing gaze tracking, we can selectively blend pre-rendered content, while maintaining visual fidelity in the critical foveal zone.
- **Inverted Pyramid Rendering System:** We develop a multi-resolution sphere tracing pipeline that utilizes an inverted pyramid framework to optimize ray tracing efficiency. This method intelligently masks edges based on depth information and eye tracking, ensuring that salient details remain perceptually accurate without incurring unnecessary computational costs for the

non-salient regions of the scene.

- **Efficient SDF Scaling and Early Ray Termination:** We introduces a rapid rendering pipeline for sphere tracing SDFs that incorporates a scaling technique to enhance surface visibility. By terminating ray tracing prematurely when nearing high-cost edges, we achieve substantial reductions in computational steps and rendering artifacts, providing a performance improvement over existing state-of-the-art methods.

Through these contributions, this thesis aims to advance the field of real-time rendering, addressing critical challenges and enhancing the feasibility of rendering complex scenes with high visual fidelity.

1.3 Thesis Structure

This thesis is organized into seven chapters, each focusing on a different aspect of the research conducted. The structure is as follows:

- Chapter 1: Introduction
  This chapter introduces the research topic, outlines the motivation behind the study, and presents the objectives, research questions, and significance of the work.

- Chapter 2: Related Work
  In this chapter, we review the relevant literature that informs our study. We begin with an overview of the human eye and its functionality. Further sections delve into concepts such as Signed Distance Functions (SDF), various ray-casting techniques (including ray tracing and sphere tracing), and the fundamentals of foveated rendering. This chapter establishes the context for our work and identifies gaps that our research aims to address.

- Chapter 3: Overview of Unity Game Engine
  This chapter provides an overview of the Unity Game Engine, which is utilized for implementing the rendering techniques explored in this study. Topics include scripting with C#, the integration of eye and head tracking systems, and the use of compute shaders for advanced rendering tasks.

- Chapter 4: Emulated Path Tracing
  Here, we present the development of an emulated foveated path tracer. This chapter details the implementation process, discusses the perceptual study conducted to evaluate the effectiveness of the approach, and presents the results and discussions related to visual fidelity and computational complexity.

- Chapter 5: An Inverted Pyramid Acceleration Structure Guiding Foveated Sphere Tracing for Implicit Surfaces in VR
  This chapter introduces a novel inverted pyramid rendering system designed to optimize sphere tracing for signed distance functions. We describe the implementation details and present a perceptual study to evaluate the performance and visual quality of our approach, along with the results and discussions on its implications for VR environments.

- Chapter 6: Skipping Spheres: SDF Scaling & Early Ray Termination for Fast Sphere Tracing
  In this chapter, we explore rapid rendering techniques for sphere tracing by implementing SDF scaling and early ray termination strategies. We detail our evaluation methodology, present visual fidelity assessments, performance evaluations, and an ablation study, highlighting the strengths and advances of our proposed techniques.

- Chapter 7: Conclusion, Limitations, and Future Work
  The final chapter summarizes the key findings of the research, discusses its limitations, and proposes directions for future work that can build on the contributions made in this thesis.

By structuring the thesis in this manner, we aim to guide the reader through the complexities of foveated rendering techniques and their application in real-time rendering systems, providing a comprehensive understanding of both the theoretical and practical aspects of the research.

Chapter 2

Related Work

In this section, we include an overview of eye physiology, ray-casting algorithms and analyze past research in foveated rendering for rasterization, ray-tracing video streaming and optimizations for sphere-tracing.

2.1 Human Eye

The visual system is a crucial human factor in designing VR hardware and software. The eye, one of the body's most complex organs, adapts to changing lighting conditions and focuses light rays from various distances. Light is converted into impulses and transmitted to the brain, where an image is perceived.

2.1.1 Anatomy of the Eye

Understanding eye tracking requires knowledge of the eye's anatomy, components, and operation. The eye has three layers, enclosing three transparent structures [18]. The outermost layer, the fibrous tunic, consists of the cornea and sclera. The middle layer, the vascular tunic or uvea, includes the choroid, ciliary body, and iris. The innermost layer is the retina, receiving circulation from the choroid and retinal vessels, visible through an ophthalmoscope.

Within these layers are the aqueous humor, the vitreous body, and the flexible lens. The aqueous humor is a clear fluid contained in the anterior chamber between the cornea and iris, and the posterior chamber between the iris and lens. The lens is suspended by the suspensory ligament (zonule), made of fine transparent fibers. The vitreous body, a clear jelly larger than the aqueous humor, is present behind the lens and bordered by the sclera, zonule, and lens, connected via the pupil.

The cornea is the transparent, outer "window" and primary focusing element of

the eye. Its outer layer, the epithelium, protects the eye and consists of transparent cells capable of quick regeneration. The inner layers of the cornea, also transparent, allow light to pass through. The pupil, the dark opening in the center of the colored iris, controls light entry. The iris functions like a camera's iris, adjusting the amount of light entering through the pupil. The lens, located immediately behind the iris, finely focuses light rays on the retina. In people under 40, the lens is soft and pliable, allowing fine focusing at various distances. For those over 40, the lens becomes less pliable, making near focus difficult, a condition known as presbyopia. The retina, lining the back of the eye, contains photoreceptor cells. These cells react to light and send impulses to the brain via the optic nerve, where the brain processes the signals into an image.

2.1.2 Foveal and Peripheral Vision

The photosensitive retina contains two types of photoreceptors: approximately 6 million cones, which enable color and high-resolution vision, and about 20 times as many rods, which support monochromatic peripheral vision. The fovea, located in the center of the retina's posterior portion, measures 1.5 mm in diameter and corresponds to 5.2° of the visual field, consisting entirely of cones.

Foveal vision is defined as the central 1.5–2° of the visual field [19]. Eccentricity refers to the angular distance from the center of the visual field, and cone density decreases with increasing eccentricities. Surrounding the fovea are the parafovea and the perifovea. The parafovea has an outer diameter of 2.5 mm, covering around 5.2° to 9° of the visual field, while the perifovea extends from 9° to 17°. Vision outside the fovea is called peripheral or indirect vision [20]. The highest density of rods is approximately 15–20° around the fovea. The fovea, the central part of the retina, is responsible for the clearest vision and sharpest color and detail.

Visual acuity, the clarity of vision, depends not only on optical factors but also on neural factors such as the ability to form a sharp retinal image, the health of the retinal function, and the brain's interpretation of visual stimuli. Visual acuity drops significantly outside the small foveal region that generates a high-resolution image. Acuity decreases rapidly with increasing eccentricity, following an inverse-linear decline, similar to a hyperbola [19]. At an eccentricity of 6°, visual acuity is already reduced by 75.

Furthermore, specific neurons sensitive to edges reside in the visual cortex [14]. Visual contrast sensitivity drives visual edge detection and provides insight into multi-channel spatial frequency selection of the visual system. We propose a foveated model that exploits the degraded visual acuity of the HVS in the periphery, while

retaining the high-cost but perceptually significant edge information, when rendering SDFs.

2.1.3 Vision Types

There are two types of vision, binocular and monocular vision [21]. Binocular is the vision in which both eyes are used together and on the other hand, monocular vision, is the vision in which each eye is used separately.

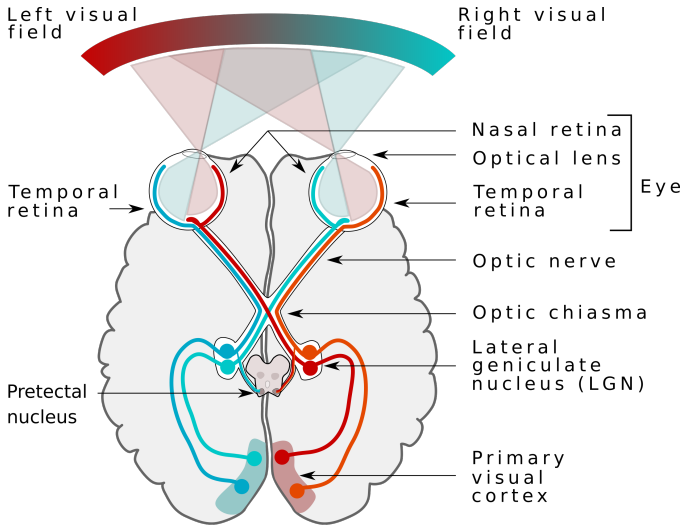


Figure 2.1: Binocular vs. Monocular Vision [98]

Stereopsis is used to refer to the perception of depth and 3-dimensional structure obtained on the basis of visual information deriving from two eyes by individuals with normally developed binocular vision [22]. Binocular vision results in two slightly different images projected to the retinas of the eyes because of the different lateral positions the eyes are located on the head. These positional differences are referred as binocular disparities. These disparities are processed by the brain to yield depth perception. While binocular disparities are naturally present when viewing a real 3-dimensional scene with two eyes, they can also be simulated by artificially presenting two different images separately to each eye using the method of stereoscopy.

On the other hand using the eyes separately, monocular vision, the FOV is increased while depth perception is limited. Monocular vision implies that only one eye is receiving optical information, the other one is closed. The perception of depth and 3-dimensional structure is, however, possible with information visible from one eye alone, such as differences in object size and motion parallax (differences in the object over time with observer movement), though the impression of depth in these cases is often not as vivid as the obtained from binocular disparities. Therefore, the term stereopsis can refer specifically to the unique impression of depth associated with binocular vision; what is colloquially referred to as “seeing in 3D”.

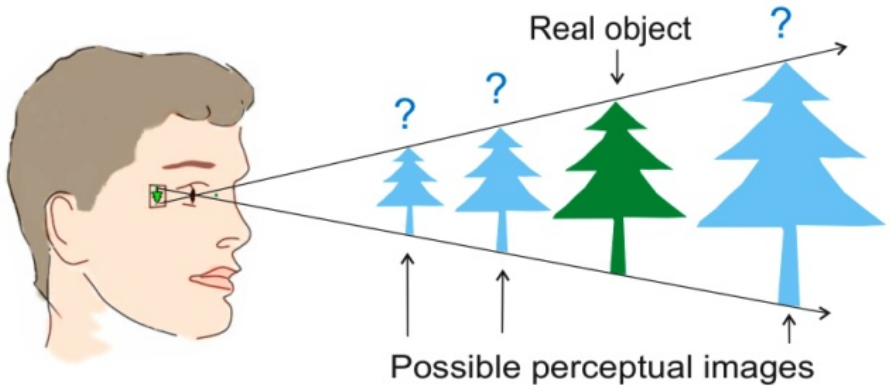


Figure 2.2: Perception of a plane object in monocular vision

In this thesis, the two displays of our Head mount display (HMD) must render the same image but with minor differences in order to create the stereopsis phenomenon and provide the user with the illusion that there is depth in our 3D environment. To create the stereopsis phenomenon, we placed two virtual cameras, one for each eye, in the virtual environment with a short distance between them based on the average IPD. Also, our Foveated sphere tracing system creates two inverse pyramid structure one for each eye to create the stereopsis phenomenon which enable to speed up the rendering process.

2.1.4 Eye Movement and Control

As eye movement, both voluntary and involuntary, refers to the movements of the eyes that help in acquiring, fixating, and tracking visual stimuli [23]. The human eye has numerous parts that must be controlled. Below, there will be a simple reference of the most major types of eye movements, and only the most commonly used in this thesis will be described. The main types of eye movements are saccades, pursuit, smooth, blinking, and compensatory. The main measurements used in eye-tracking research are fixations and saccades. In this thesis, we are interested in the eye fixations observed on the head-mounted display to render images for each eye based on the fixation point. We also investigate whether saccadic movements alert the user to the drop in quality produced by our foveated rendering system.

A saccade is a rapid eye movement, a jump, which is usually conjugate and under voluntary control but ballistic; once they are initiated, the path of motion and destination cannot be changed . It takes about 100 to 300 milliseconds to initiate a saccade, from the time a stimulus is presented until the eye starts moving, and another 30 to 120 milliseconds to complete the saccade. The purpose of these movements is to bring images of particular areas of the visual world onto the fovea. Saccades are therefore a major instrument of selective visual attention . It is often

convenient to consider that a saccadic eye movement always occurs in a straight line and that we do not 'see' during these movements.

Fixation is the moment, averaging 218 milliseconds, when the eyes are relatively stationary between saccades, taking in or encoding information. Fixation duration provides an index of the speed with which information is processed. Increasing fixation duration is associated with tasks requiring more detailed visual analysis. The frequency of fixations often serves as a measure of sampling quantity. They can reveal the amount of processing being applied to objects; therefore, studying them can tell us about the complexity or salience of an object in an interface.

In this thesis, we present a foveated sphere tracing rendering technique that utilizes the eye tracker of the HTC Vive to monitor the user's gaze on the head-mounted display. We render images according to the user's fixation point, providing the highest possible quality at that point and in the surrounding area. As we move away from the fixation point, the rendering quality decreases. Our perceptual study demonstrates that our system can produce images with lower quality in areas away from the fixation point without the user perceiving a noticeable drop in quality, even during saccadic eye movements.

2.1.5 Perception in an Immersive Virtual Environment

Perception of our immediate environment is not based on what we actually see or what is there. It is based upon little actual sensory information and is for the most part illusory. Theoretically everything we 'see' around us exist as a model in our minds. We rely upon our perceptive system so much that it enables us to be fooled.

When immersed in a Virtual Environment (VE) our compelling senses are presented with an alternative view of our local environment whilst the real world is shut out [24]. Our perceptual system is trained over many years to recognize our everyday reality, thus has no experience to distinguish it from the VR. An immersive virtual environment (IVE) simply provides cues that are a sufficient match for our inner conceptual models of what it is to be in an environment. For instance, stage magicians rely upon this fact by providing basic cues that purposely misinform our perceptual system and leave us wondering how we have apparently jumped from one world-state to another. Just as when we see an illusion and are able to accept the perhaps 'odd' perspective that is implied, when we view an IVE we can accept the virtual world perspective implied over the real world.

In this thesis, we aim to render a 3D environment with parameters that align

with the human eye's visual model, maintaining high immersion while maximizing performance by lowering quality in regions that correspond to peripheral vision. By combining path tracing for realistic rendering with foveated rendering, we optimize the process by reducing image resolution in areas where the eye cannot discern fine details. Additionally, we propose foveated methods for sphere tracing, where quality of rendering is dynamically adjusted based on user gaze fixation within the HMD, further enhancing efficiency in less perceptually critical regions and overall improving rendering performance.

2.2 Singed Distance Functions

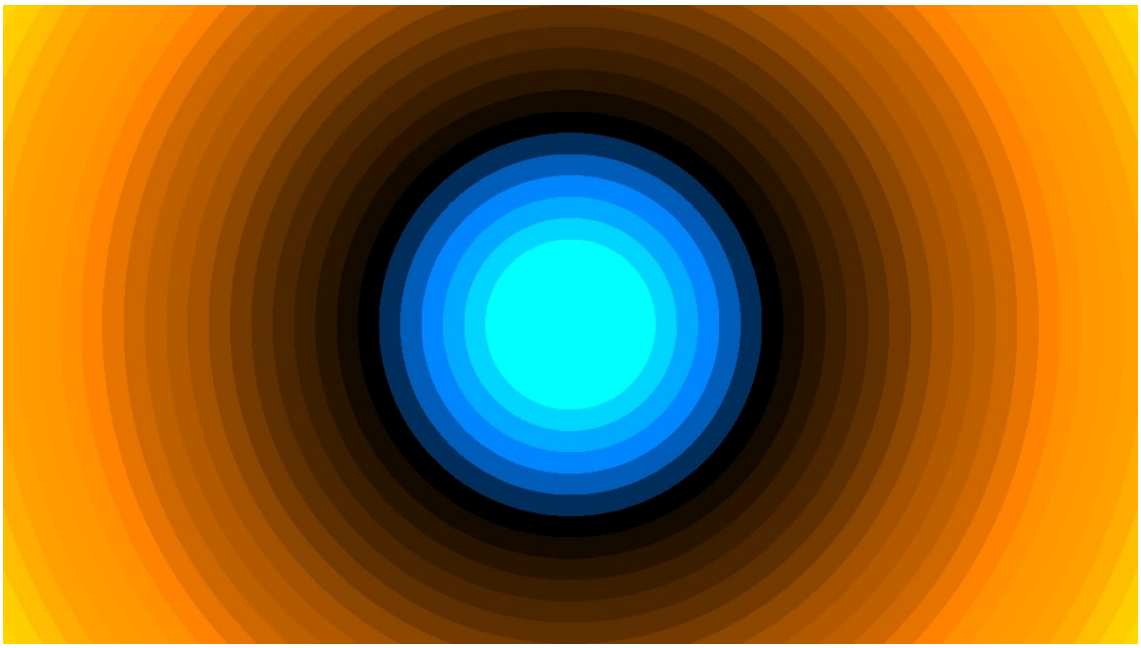


Figure 2.3: Visualization of a Circle Signed Distance Function (SDF) 2D [1] where the blue region represents the interior space, the orange region signifies the exterior space, and the black boundary illustrates the surface of the circle.

In computer graphics, scene representation serves as the foundational framework for defining and organizing the elements that constitute a virtual environment. It encompasses the spatial, geometric, and semantic structures required to model objects, lighting, textures, and interactions within a digital scene. Effective scene representation is critical for rendering realistic images, enabling efficient computational processing, and supporting advanced techniques such as ray tracing, global illumination, and real-time simulations. This subsection explores the key methodologies and data structures employed in scene representation, highlighting their roles in bridging the gap between abstract mathematical models and visually compelling graphical outputs. By understanding these principles, we can better appreciate the

underlying mechanisms that drive the creation of immersive and interactive virtual worlds.

Signed Distance Functions (SDFs) are mathematical representations primarily used in computer graphics to describe geometric shapes. They determine the distance from a point in space to the closest point on the surface of the shape, with the sign indicating whether the point is inside or outside the shape. Mathematically, for a point $\mathbf{p}$ in space and a shape with surface defined by $S(\mathbf{x}) = 0$, where $\mathbf{x}$ is a point on the surface, the signed distance function is given by $d(\mathbf{p}) = \text{sign}(S(\mathbf{p}))\|S(\mathbf{p})\|$, where $\|\cdot\|$ denotes the Euclidean norm [25]. Functions of this type enable shape manipulation and rendering, resulting in efficient algorithms for tasks such as ray tracing, sphere tracing, and collision detection in computer graphics [26, 27]. SDFs are also used to manually or procedurally generate content [28] due to their efficient memory utilization. Examples include procedural displacement mapping tools [29], artistic rendering [1], or fully connected neural networks learning from multiple images [30, 31, 32] to estimate surfaces in those images.

Among the various techniques for surface representation, SDFs offer a powerful and versatile approach for defining analytic implicit surfaces. SDFs represent a surface implicitly by defining a continuous scalar field that specifies the shortest distance from any point in space to the surface, with the sign indicating whether the point lies inside (negative) or outside (positive) the volume. This representation provides several advantages. First, SDFs enable precise and smooth surface definitions, making them particularly well-suited for modeling complex geometries with high fidelity. Second, they facilitate efficient geometric operations, such as Boolean combinations, blending, and deformations, due to their mathematical properties. Third, SDFs are inherently resolution-independent, allowing for scalable representations that avoid the artifacts often associated with discrete mesh-based models. Additionally, SDFs are computationally efficient for ray marching and collision detection, making them highly suitable for real-time applications. By leveraging these strengths, SDF-based scene representations have become a cornerstone in modern computer graphics, enabling advanced techniques in rendering, animation, and physical simulation.

In addition to SDFs, other techniques for surface representation include polygonal meshes, parametric surfaces, and voxel-based representations. Polygonal meshes, composed of vertices, edges, and faces, are widely used due to their simplicity and compatibility with rendering pipelines. However, they often struggle to represent smooth surfaces accurately without excessive polygon counts, leading to increased memory and computational costs. Parametric surfaces, such as NURBS (Non-Uniform Rational B-Splines), provide smooth and precise representations but can be computationally intensive to evaluate and manipulate. Voxel-based representa-

tions discretize space into a grid of volumetric elements, which are useful for certain applications like medical imaging or fluid simulations but can suffer from resolution limitations and memory inefficiencies when representing fine details.

Given these trade-offs, SDFs stand out as a compelling choice for scene representation due to their unique combination of precision, flexibility, and computational efficiency. Their ability to define smooth, resolution-independent surfaces makes them ideal for applications requiring high-quality geometry, such as procedural modeling, real-time rendering, and physical simulations. Furthermore, SDFs support efficient geometric operations and are well-suited for modern techniques like ray marching, making them a natural fit for both offline and real-time graphics pipelines. By choosing SDFs, we leverage their mathematical elegance and practical advantages to create robust and scalable representations that bridge the gap between abstract modeling and visually compelling virtual environments.

2.2.1 3D Shapes - Signed Distance Function

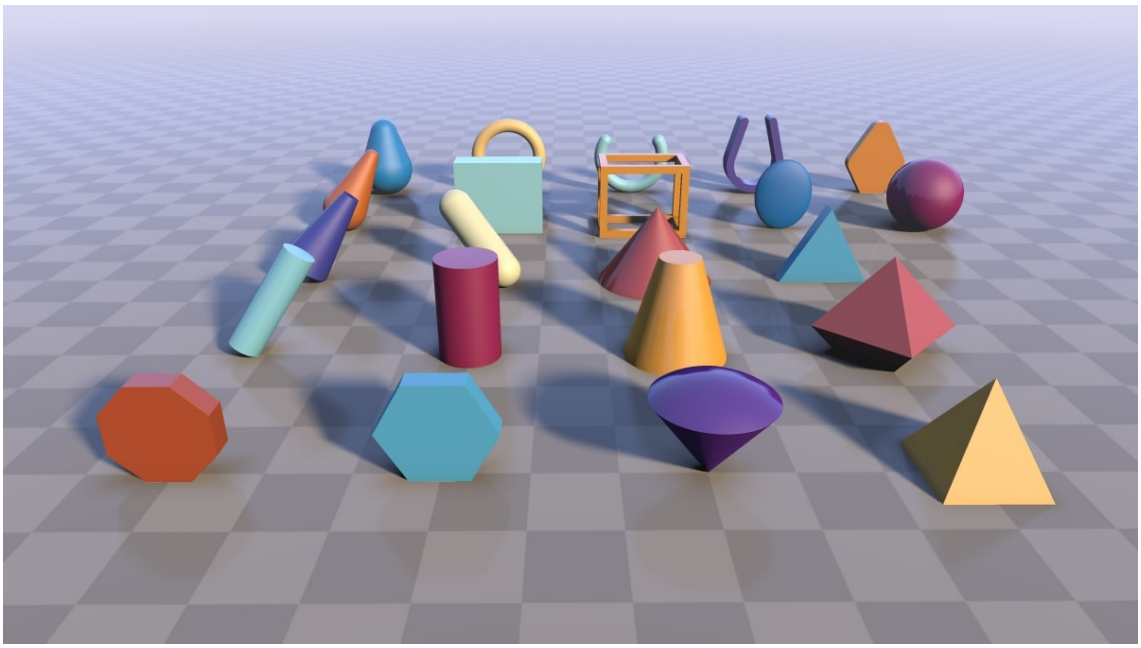


Figure 2.4: Visualization of singed distance function primitives [1].

Signed Distance Functions (SDFs) as mention above are mathematical descriptions that provide the shortest distance from a point to a surface. Here, we describe various 3D shapes using SDFs, as seen in figure 2.4. Each shape has its unique SDF, which can be used in rendering and ray-marching algorithms to create complex scenes. Below is the mathematical representation of signed distance functions of various primitives shapes.

The function Plane$(\hat{p}, \hat{n}, h)$ calculates the signed distance from a point $\hat{p}$ to the

surface of a plane defined by its normal vector $\hat{n}$ and distance from the origin h in 3D space. The main parts of the equation are:

$$\text{Primitive}_{\text{Plane}}(\hat{p}, \hat{n}, h) = \hat{p} \cdot \hat{n} + h$$

1. $\hat{p} \cdot \hat{n}$: This part calculates the dot product of the point $\hat{p}$ and the normal vector $\hat{n}$, which show if the ray at point p is parallel or perpendicular to the plane.
2. $\hat{p} \cdot \hat{n} + h$: This part adds the distance h to the dot product result and control the height of the plane. It represents the signed distance from the point $\hat{p}$ to the plane.

The equation Sphere$(\hat{p}, s)$ calculates the signed distance from a point $\hat{p}$ to the surface of a sphere with radius s centered at the origin in 3D space by calculating the euclidean distance between the center of the world and the current position of the ray and then subtract from the calculate distance the radius of the sphere.

$$\text{Primitive}_{\text{Sphere}}(\hat{p}, s) = ||\hat{p}||_2 - s$$

The function Box$(\hat{q})$ calculates the signed distance from a point $\hat{p}$ to the surface of a box defined by its half extents $\hat{b}$ in 3D space.The main parts of the equation are:

$$\hat{q}(\hat{p}, \hat{b}) = |\hat{p}| - \hat{b};$$

$$\text{Primitive}_{\text{Box}}(\hat{q}) = ||\max(\hat{q}, 0)||_2 + \min(\max(q_x, \max(q_y, q_z)), 0)$$

1. $(\hat{q})$ is a vector that generally comprises the distances in the x, y, and z dimensions. This means we are subtracting the half extent of the box along each axis from the absolute position of the point.
2. $||\max(\hat{q}, 0)||_2$: This part takes the maximum of each component of $(\hat{q})$ with zero, effectively ignoring any negative distances (those indicating that the point is inside the box). It then computes the Euclidean norm (or length) of this resultant vector.
3. $\min(\max(q_x, \max(q_y, q_z)), 0)$: This takes the maximum value among (q_x), (q_y), and (q_z) and then compares it to zero. The minimum function ensures that if the maximum value is positive (the point is outside the box), this will contribute a non-positive value (zero or negative) to the overall distance. Therefore, if the point extends beyond the box surface, it adds a negative value

indicating how far it goes outside the box.

The function RoundBox($\hat{p}, \hat{b}, r$) calculates the signed distance from a point $\hat{p}$ to the surface of a rounded box defined by its half extents $\hat{b}$ and the radius r of the rounded corners in 3D space. It is similar to the Box function but with rounded corners.

$$\hat{p}_{new}(\hat{p}, \hat{b}) = |\hat{p}| - \hat{b};$$

$$\hat{q}(\hat{p}, \hat{b}, r) = |\hat{p}_{new}| - \hat{b} + r;$$

$$\text{Primitive}_{\text{RoundBox}}(\hat{q}, r) = ||max(\hat{q}, 0)||_2 + min(max(q_x, max(q_y, q_z)), 0.0) - r$$

The function BoxFrame($\cdot$) calculates the signed distance from a point $\hat{p}$ to the surface of a box frame defined by its half extents $\hat{b}$ and the frame thickness e in 3D space. It is similar to the Box function but with a frame around the box.

$$\hat{q}(p, b, r) = |\hat{p} + e| - e;$$

After we calculate the three different boxes, as shown below, to build the BoxFrame shape.

$$d_1(\hat{q}, \hat{p}, \hat{b}, r) = ||max(\begin{bmatrix} p_x \\ q_y \\ q_z \end{bmatrix}, 0) + \min(\max(p_x, \max(q_y, q_z)), 0.0)||_2;$$

$$d_2(\hat{q}, \hat{p}, \hat{b}, r) = ||\max(\begin{bmatrix} q_x \\ p_y \\ q_z \end{bmatrix}, 0) + \min(\max(q_x, \max(p_y, q_z)), 0.0)||_2;$$

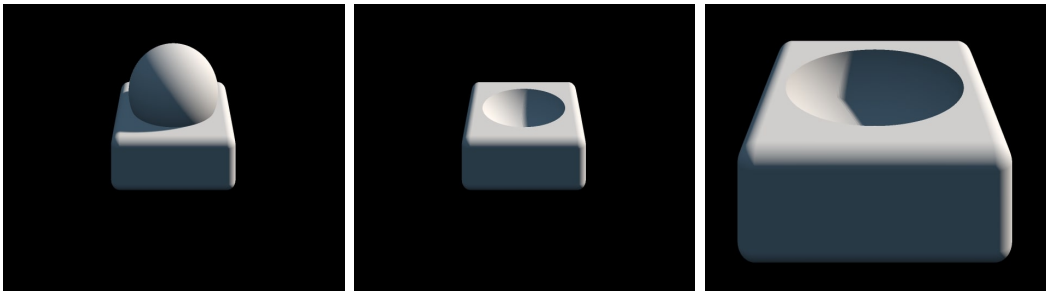
$$d_3(\hat{q}, \hat{p}, \hat{b}, r) = ||\max(\begin{bmatrix} q_x \\ p_y \\ p_z \end{bmatrix}, 0) + \min(\max(q_x, \max(q_y, p_z)), 0.0)||_2;$$

The BoxFrame signed distance function then is calculated by combining three boxes using the union operator to create the BoxFrame shape.

$$\text{Primitive}_{\text{BoxFrame}}(d_1, d_2, d_3) = \min(d_1, d_2, d_3);$$

In this thesis, we present a brief introduction to 3D shapes represented by signed distance functions (SDFs); for a more comprehensive exploration of shapes, please refer to [1]. SDFs offer precise distance calculations for a variety of 3D shapes, making them indispensable tools in computer graphics for crafting detailed and accurate 3D models and scenes. By combining these functions, we can efficiently model complex shapes and objects, allowing us to evaluate the performance of our methods under various conditions.

2.2.2 Operations



(a) Union (b) Subtraction (c) Scaling of (b)

Figure 2.5: Useful operations for handling / combining SDFs ([1])

To create a 3D scene, each object is represented by a single or more SDFs or by combining SDFs using operators. For example, equation 2.2.1 describes a sphere. By combining primitives defined by a Signed Distance Function (SDF) (primitive(p)), such as mention on section 2.2.1, using the union or subtraction operators (Figure 2.5), we can create more complex shapes.

The union operation takes the calculated distance of two SDFs and chooses the minimum distance as seen in the following equation, to combine two objects into one (Figure 2.5a):

$$Union(p, \cdots) = \min(primitives_1(p, \cdots), primitives_2(p, \cdots))$$

where p is the current ray position in the world. Similarly, the subtraction operation takes the two distances but instead inverts the sign of the distance (i.e., object) we want to remove from the other SDF as seen in the following equation:

$$Subtraction(p, \cdots) = \max(primitives_1(p, \cdots), -primitives_2(p, \cdots))$$

Using the subtraction operation, we can remove parts of the implicit surface (Figure 2.5b). Another common operation when creating a 3D environment is surface scaling to adjust the size of objects. To scale an SDF, we can compress the space where the surface exists. Distance estimation is then performed in the *scaled space* and finally we multiply the result with the scale value to rescale the space as seen in equation 2.2.2:

$$Scale(p) = primitives(p/s) \times s$$

where s is the scaling factor applied in the signed distance functions (Figure 2.5c).

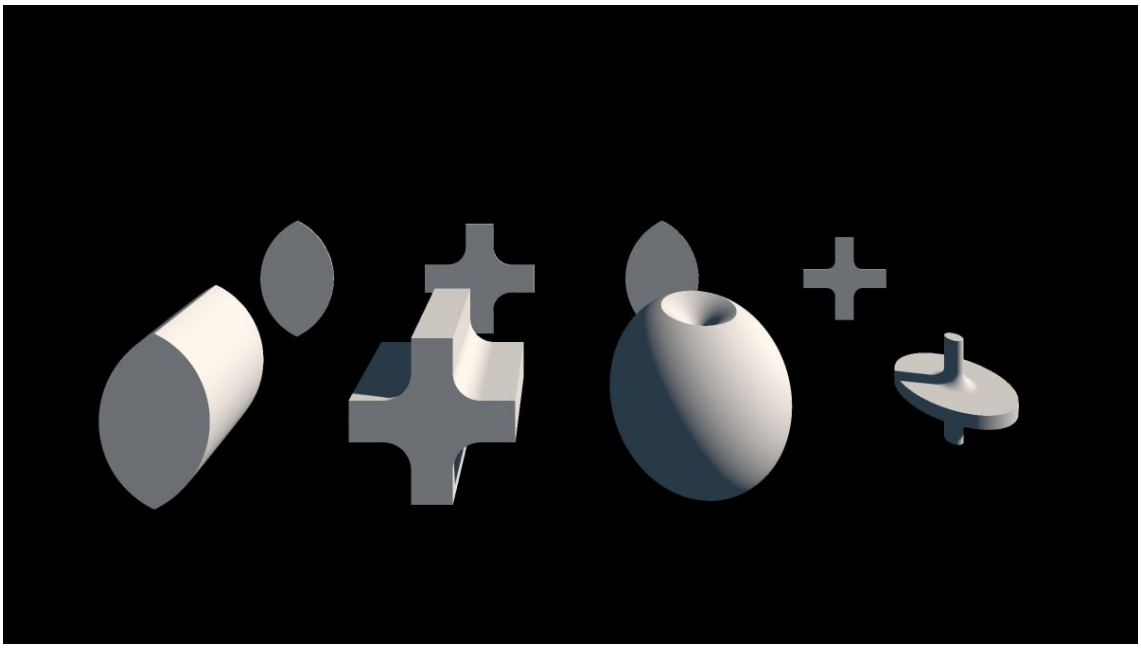


Figure 2.6: Shapes produced using revolve or extrude operation ([1]).

The list above is not exhaustive; a simple SDF can easily describe many other shapes. One effective method is to take any 2D SDF and either revolve or extrude it, as seen in figure 2.6. This method is straightforward and has the advantage that if the 2D SDF we start with is exact, the resulting 3D volume will also be exact. This is beneficial for a couple of reasons: first, creating a shape through 3D boolean operations of basic forms does not produce an exact SDF, whereas revolving or extruding a 2D shape does. Secondly, 3D boolean operations generate suboptimal code since they do not reuse common expressions across primitives. Thus, using

revolution or extrusion of 2D shapes produces the correct SDF and is also faster to compute.

Revolution is expressed as follow:

$$\hat{q}(\hat{p}, \hat{o}) = \sqrt{p_x^2 + p_z^2} - \hat{o}$$

$$\hat{p}_{\text{revolution}}(\hat{q}) = \begin{bmatrix} q_x \\ q_y \\ p_y \end{bmatrix}$$

$$\text{Revolution}(\hat{p}_{\text{revolution}}, \hat{o}) = \text{primitive}(\hat{p}_{\text{revolution}})$$

Extrusion is expressed as follow:

$$\hat{q}(\hat{p}) = \begin{bmatrix} p.x \\ p.z \end{bmatrix}$$

$$d = \text{primitive}_{\text{2D}}(\hat{q})$$

$$\hat{w}(d, \hat{p}, h) = \begin{bmatrix} d \\ |p_z| - h \end{bmatrix}$$

$$l(\hat{w}) = ||\max(\hat{w}, 0)||_2$$

$$Extrusion(\hat{w}, l) = \min(\max(w_x, w_y), 0.0) + l$$

It's also possible to create new types of 3D primitives from existing 3D primitives. Here are some examples:

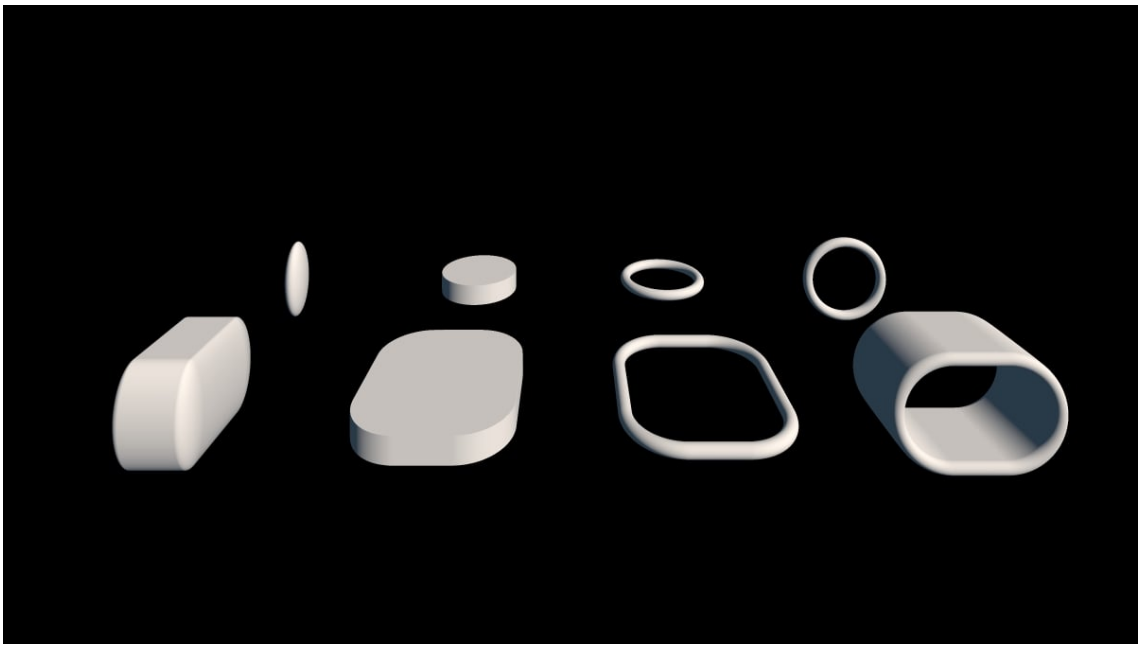


Figure 2.7: Shapes produced using Elongating operation ([1])

Elongating is a useful technique to construct new shapes. It essentially splits a primitive into parts, moves them apart, and connects them, as seen in figure 2.7. This operation preserves distances perfectly, without introducing any artifacts in the SDF. Some basic primitives, like the Capsule primitive, use this technique.

$$\hat{q}(\hat{p}, h) = \begin{bmatrix} p.x - \max(min(p.z, -h), h) \\ p.z - \max(min(p.z, -h), h) \\ p.z - \max(min(p.z, -h), h) \end{bmatrix}$$

$$Elongate(\hat{q}) = primitive(\hat{q})$$

The second function ensures exact interior distances even for 2D and 3D elongations.

Rounding a shape involves subtracting a distance, as seen in figure 2.8, effectively changing the isosurface. This technique can be applied to various shapes such as cones or hexagons. If preserving the overall volume is necessary, the source primitive can be shrunk by the same amount being rounded.

$$Round(\hat{p}, r) = primitives(\hat{p}) - r$$

To carve interiors or give thickness to primitives without performing expensive boolean operations or distorting the distance field, you can use "onioning". This

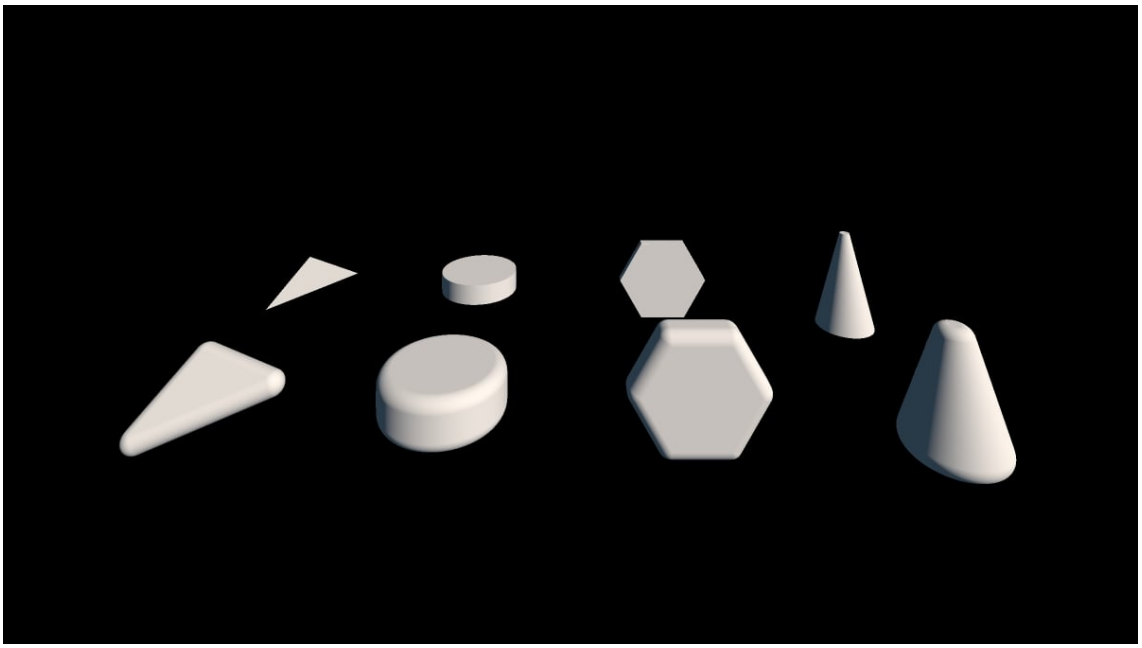


Figure 2.8: Primitive shapes where rounding has been applied ([1]).

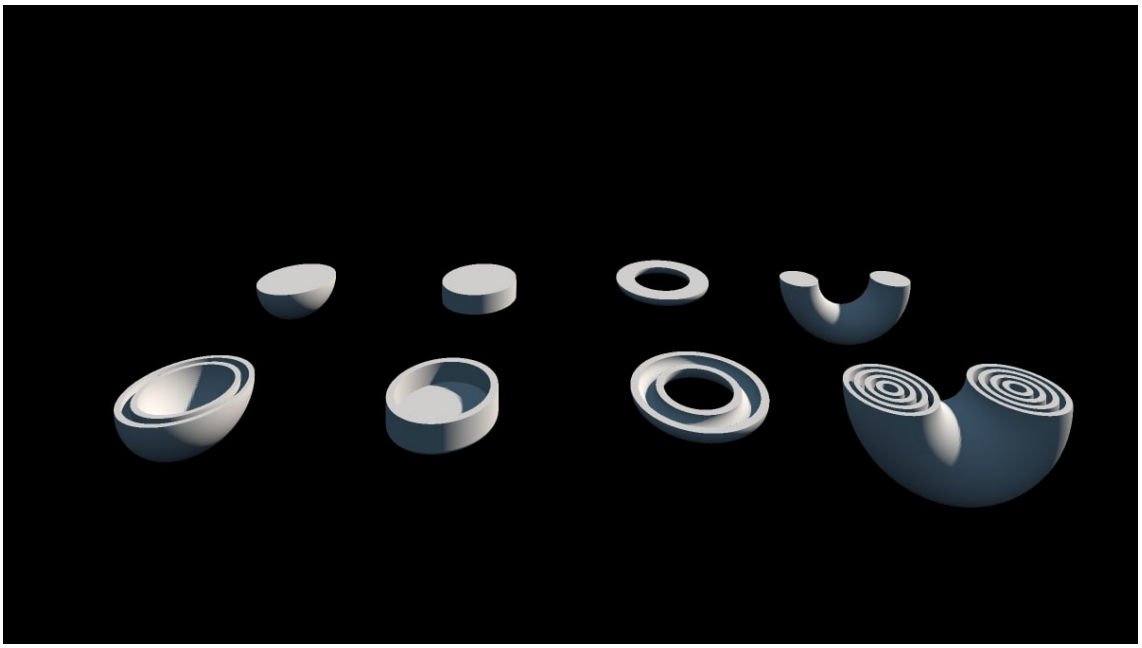


Figure 2.9: Carve interiors to shapes using the "onioning" operation ([1]).

method can create concentric layers in your SDF, as seen in figure 2.9.

$$\hat{q}(\hat{p}) = \begin{bmatrix} |p_x| \\ |p_y| \\ |p_z| \end{bmatrix}$$

$$opOnion(\hat{q}, thickness) = primitives(\hat{q}) - thickness$$

Blending primitives is a powerful tool that allows for constructing complex and organic shapes without the geometric seams produced by normal boolean operations. The basic smooth operations replace the 'min()' and 'max()' functions used in 'opUnion', 'opSubtraction', and 'opIntersection' with smooth versions. The parameter k defines the size of the smooth transition between the two primitives (d_1 is the first SDF and d_2 is second SDF).

The smooth Union operator is calculated as followed:

$$h(d_1, d_2, k) = \max(\min(0.5 - 0.5 \times \frac{(d_2 + d_1)}{k}, 1), 0)$$

$$SmUn(d_1, d_2, k, h) = d_2 \times h + (1 - h) \times d_1 - k \times h \times (1.0 - h)$$

The smooth Subtraction operator is calculated as followed:

$$h(d_1, d_2, k) = \max(\min(0.5 - 0.5 \times \frac{(d_2 + d_1)}{k}, 1), 0)$$

$$SmSub(d_1, d_2, k, h) = d_2 \times h + (1 + d1) + k \times h \times (1.0 - h)$$

The smooth Intersection operator is calculated as followed:

$$h(d_1, d_2, k) = \max(\min(0.5 - 0.5 \times \frac{(d_2 - d_1)}{k}, 1), 0)$$

$$SmIn(d_1, d_2, k, h) = d_2 \times h + (1 - h) \times d_1 + k \times h \times (1.0 - h)$$

Placing primitives in different locations and orientations in space is fundamental in designing SDFs. While rotations, uniform scaling, and translations are exact operations, non-uniform scaling distorts the Euclidean space and can only be bound. Therefore, it is not included here.

Since rotations and translations do not compress or dilate space, all we need to do is transform the point being sampled with the inverse of the transformation matrix used to place an object in the scene (position and orientation of object):

$$opTx = primitive(\boldsymbol{Tranformation}^T \times \hat{p});$$

Creating multiple copies of the same object can be done easily at no memory or

performance cost by making the SDF function itself symmetric or periodic, resulting in automatic instancing in constant time.

Symmetry is useful because many things around us are symmetric. By using the absolute value of the domain coordinates before evaluation, one can model only part of the desired shape and duplicate it automatically.

$$\hat{q}_x(\hat{p}) = \begin{bmatrix} |p_x| \\ p_y \\ p_z \end{bmatrix}$$

$$SymX(\hat{q}_x) = primitive(\hat{q}_x)$$

$$\hat{q}_{xz}(\hat{p}) = \begin{bmatrix} |p_x| \\ p_y \\ |p_z| \end{bmatrix}$$

$$SymXZ(\hat{q}_{xz}) = primitive(\hat{q}_{xz})$$

Domain repetition allows creating unlimited copies of a primitives with a single singed distance function evaluation:

$$Repetition(\hat{p}) = primitive(\hat{p} - s \times (\frac{\hat{p}}{s} + 0.5))$$

2.2.3 Deformations and Distortions

Deformations and distortions modify the distance function (d) of a shape defined by a Signed Distance Function (SDF) (primitive(p)).

Given a displacement function (displacement(p)), the new distance can be expressed as:

$$\text{Displaced}(p) = d_1 + d_2$$

Where:

(d_1 = primitive(p)) is the original SDF and (d_2 = displacement(p)) is the displacement applied to each point (p).

The twist operation can be represented as transforming the coordinates of a

point using rotation defined by angle (k) around the y-axis. The transformation is expressed as:

$$c(\hat{p}) = \cos(k \cdot p_y)$$

$$s(\hat{p}) = \sin(k \cdot p_y)$$

$$\boldsymbol{m}(c, s) = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}$$

$$\hat{q}(\hat{p}, \boldsymbol{m}) = \begin{bmatrix} m_{1,1} \times p_x + m_{1,2} \times p_z \\ m_{2,1} \times p_x + m_{2,2} \times p_z \\ p_y \end{bmatrix}$$

$$\text{opTwist}(\hat{q}) = \text{primitive}(\hat{q})$$

In this thesis, our scenes use these operations to create complex worlds by repeating specific SDFs or combining two or multiple primitive SDFs to one. By mastering these operations, you can build a wide variety of complex and organic shapes in an efficient and mathematically sound manner something that would being time consuming using a triangle mesh representation for the virtual environment and would require millions of triangle which will increase the rendering time.

2.3 Ray-Tracing

Direct imaging algorithms, such as rasterization, work by projecting a 3D model onto a 2D screen and filling the image register with colors. These algorithms synthesize images by mapping data from the 3D environment into 2D space, utilizing the z-buffer algorithm to manage hidden surface removal directly on the screen.

In contrast, ray tracing is a versatile and comprehensive method that generates images by tracing rays from the camera's viewpoint through each pixel, interacting with the 3D scene as they do so. When a ray intersects an object's surface, it may absorb, reflect, refract, or diffuse light based on the material properties of that surface. Hidden surface removal occurs in 3D space, as the rays encounter the closest surfaces to the camera while traversing the scene.

The concept of tracing the path of light and calculating its interactions with

various materials dates back long before computer graphics, rooted in the studies of electromagnetic wave transmission, optical geometry, and the laws of reflection and refraction. These foundational principles laid the groundwork for the development of ray tracing algorithms in the late 1970s [33].

While direct imaging techniques often calculate colors and shading in isolation, without considering interactions with other objects in the scene, they also require separate calculations for shadows, reflections, and refractions. These effects are integrated into local lighting models during scene scanning. In contrast, ray tracing elegantly combines these calculations into a unified recursive algorithm known as recursive ray tracing. Though ray tracing can produce perfect reflections, it does not inherently yield photorealistic images. To achieve global illumination, path tracing uses stochastic sampling to randomly sample light paths throughout the scene, effectively capturing interactions between light and surfaces. This method introduces variability in the rays' paths, allowing path tracing to simulate complex lighting effects such as soft shadows and indirect lighting; however, it can also introduce noise, necessitating a high number of samples to produce smooth, realistic images.

Moreover, ray tracing algorithms simulate light interactions with the environment based on material properties. Whether employing polygonal surfaces or signed distance functions (SDF) for representation, the differences lie primarily in the intersection tracing methods, which enable a variety of rendering techniques. This flexibility ensures accurate light transfer and realistic visualization across diverse surface types.

2.3.1 Recursion Ray-Tracing Algorithm

The idea for casting rays from the camera to the scene for the construction of an image with the correct depth without the use of the Z-buffer algorithm was first introduced by Appel [34], Goldstein and Nagel [35]. A complete method for the recursive ray-tracing, where rays are reflected and refracted in the scene was proposed later by Whitted [33]. This method combine the previous algorithm, where they send primitives rays form the camera to the scene until they find an intersection with a surface. When they find an intersection they shade the intersection points base on local model of light with they recursive born of new rays from these points.

The Ray tracing algorithm is simple: For each pixel a ray is generated (primitive ray) with starting point the camera where pass through the center of the pixel. The ray is looking for intersection with the scene geometry in order to find the closest intersection of ray with scene based on the starting point, as seen in Figure 2.10. When a valid intersection point has been detected, a local lighting model is applied

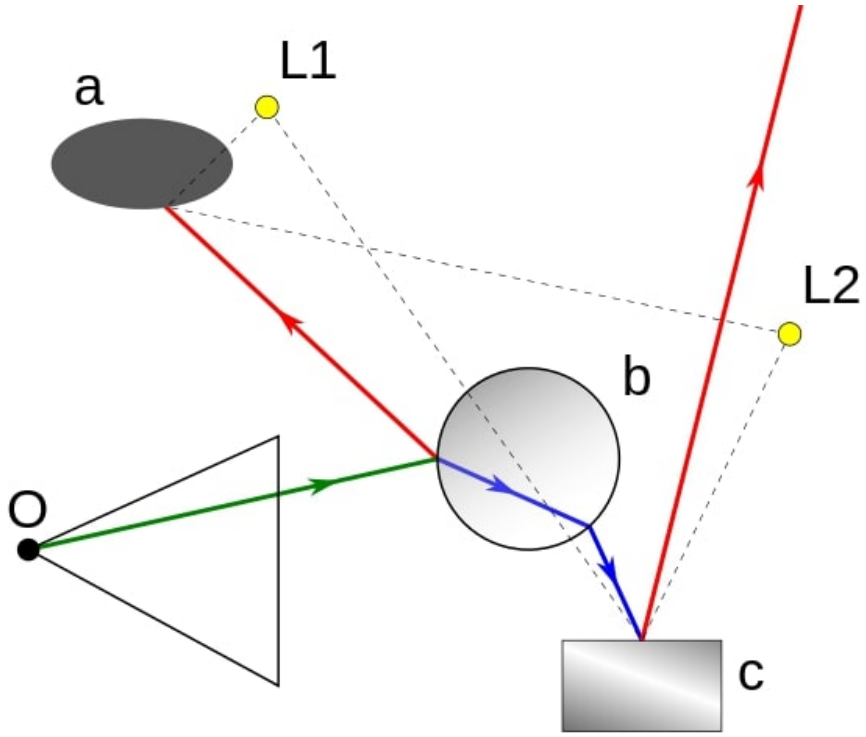


Figure 2.10: Recursion Ray-Tracing Algorithm in a scene. Green ray is the first ray shoot form point O. Red, blue and dashed lines are rays generated after intersect with a surface. Red rays are reflect from geometry b and c . Blue Rays are refracted form geometry b. Dashed Rays are rays cast toward the light source L1 and L2.

on to be shaded base on the light sources that are visual contact with the intersection point. If an intersection is not detected, the ray takes the color of the background. If the material of the surface where the ray stops is transparent, a second transparent ray is generated from the intersection point. If the surface is reflective, a second ray is generated towards the direction of perfect reflection. Both the secondary rays are treated as the primitive ray meaning from the point of intersection we look for new intersection with the scene geometry. In case of hitting a surface, shading is calculated again with a local illumination model and new rays are generated based on the material of the surface.

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

pixel.

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

In this thesis, we propose improvements to streamline the sphere-tracing process for implicit surfaces by approximating the travel distance of rays during the tracing process for both non-foveated and foveated renderings. To evaluate our method, we utilize recursive ray tracing, focusing specifically on shadow tracing to enhance realism. By selecting shadows for evaluation, we can leverage signed distance functions to easily control the shadow tracing and achieve soft, visually appealing shadows, validating that our approach can efficiently trace both primary and secondary rays.

2.3.2 Path-Tracing

Figure 2.11: The Sponza palace scene, 16th century, reconstruction, with embedded transparent Stanford dragons enabling experimentation with many complex effects such as diffuse inter-reflection, shadows, transparency and more.

Path-tracing (PT) [36] is a rendering technique producing photo-realistic images by simulating light transport. By simulating the full global illumination of a scene,

including indirect lighting, color bleeding, and soft shadows, path tracing produces highly realistic images. It extends ray tracing by using Monte Carlo integration to approximate the rendering equation, tracing the full path of light as it interacts with surfaces multiple times. PT has a high computational cost as a large number of rays shot form each pixel have to be tracked along their path through a virtual scene to calculate the light reaching a virtual camera.

Path tracing works by casting rays from the camera into the scene, simulating how light interacts with surfaces. When a ray intersects a surface, the algorithm randomly chooses a new direction for the ray to follow, mimicking light scattering. This process continues until a user-defined maximum number of bounces is reached. For effects like reflections and refractions, higher bounce limits are necessary for more accurate results. However, each time a new path is traced for a pixel sample, a slightly different image is generated. To reduce noise and achieve a clearer image, many samples are averaged together since each ray starts from a slightly different point inside the pixel. The Bidirectional Reflectance Distribution Function (BRDF) further introduces randomness in how rays reflect from surfaces, which adds to the complexity of simulating light interactions.

While path tracing excels at producing photorealistic images by modeling global illumination and complex light effects like color bleeding compared to recursive ray tracing, it is also computationally demanding. A single sample per pixel can produce a noisy result, as the random scattering of light paths causes variations in the image. To overcome this, more samples must be taken and averaged, which reduces the noise and leads to a more refined image. This requirement for multiple samples increases the computational cost, making path tracing less suitable for real-time applications. Although some advancements have enabled its use in real-time graphics, path tracing remains primarily an offline rendering method due to the heavy processing needed to reduce noise and reach convergence. However, hardware advancements like NVIDIA's RTX GPUs have enabled the use of path tracing with just one sample per pixel, resulting in noisy output. Additionally, the current state-of-the-art focuses on improving denoisers using AI or image filtering algorithms to effectively remove noise, or employing AI to render at lower resolutions and upscale the images for enhanced detail.

In this thesis, we present an emulated foveated path-tracing system designed to investigate the perceptibility of our approach with increased samples per pixel and the integration of a denoiser. Our method enhances rendering quality by providing high fidelity in the regions where the gaze is focused, while effectively reducing quality in the peripheral areas, all without users noticing a decline in visual performance in virtual reality environments. This technique can be applied to both

polygonal and implicit surfaces, enabling efficient rendering that optimizes resource usage while preserving an immersive experience.

Monte Carlo

(a) 8 spp (b) 1024 spp

Figure 2.12: (a) Image with noise by averaging 8 samples per pixel. (b) Image with noise reduced by averaging 1024 samples per pixel([2]).

Monte Carlo is a mathematical method used to solve problems that involve randomness or uncertainty by relying on random sampling to approximate results. In the context of computer graphics, particularly path tracing, Monte Carlo methods are employed to estimate the behavior of light as it bounces around a scene. Instead of tracing every possible light path, which would be computationally infeasible, Monte Carlo randomly samples a subset of paths and then averages their contributions to approximate the final color of each pixel.

Mathematically, Monte Carlo integration can be expressed as:

$$I \approx \frac{1}{N} \sum_{i=1}^{N} f(x_i) \tag{2.1}$$

Where:

- I represents the estimated integral (in this case, the approximated lighting contribution for a pixel).
- N is the number of samples taken.
- $f(x_i)$ is the function being evaluated (the light interaction at a given point x_i).

In this formula, N random samples are generated from camera origin towards the direction the camera is looking, and the function $f(x_i)$, which represents the behavior of the ray when intersecting with the surface based on surface properties along these paths, is computed for each sample. The results are then averaged to produce an estimate of the total ray contribution. The accuracy of this approximation improves as the number of samples, N, increases, but taking fewer samples results in noisier outputs, as seen in figure 2.12. This balance between computational cost and image quality is at the core of path tracing, where Monte Carlo methods enable the simulation of complex lighting effects, such as global illumination and soft shadows, with high realism.

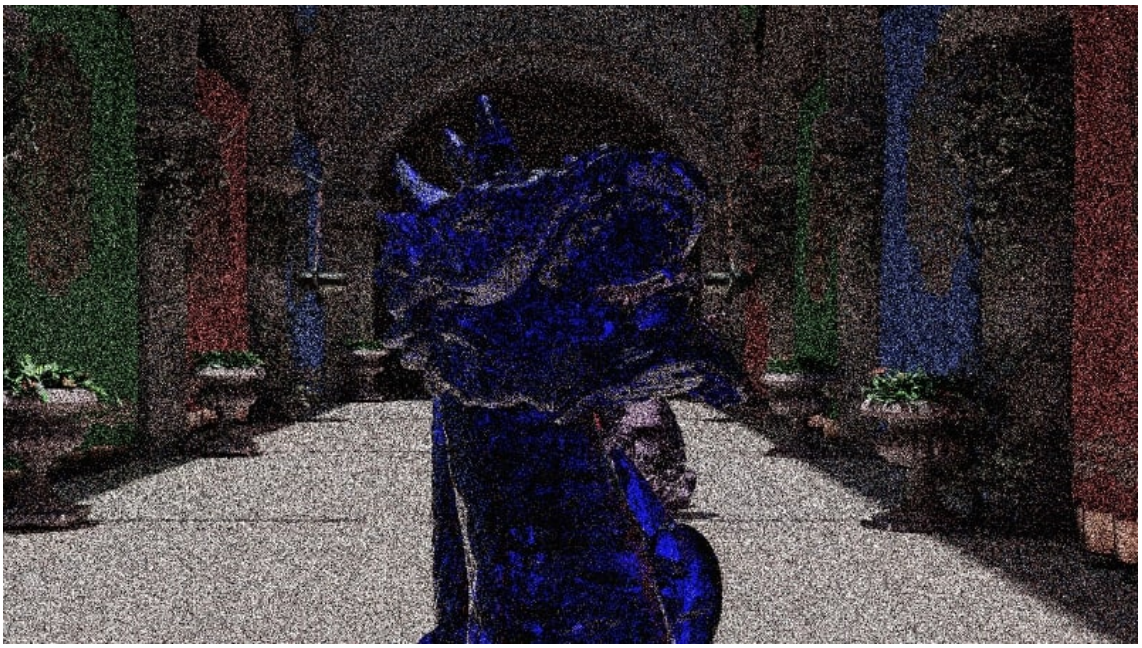
In this thesis, we employ Monte Carlo methods to produce pre-rendered images using path tracing for our emulated foveated framework. This approach enables us to determine the perceptibility thresholds of our foveated model by using a sufficient number of samples per pixel while leveraging a denoiser to eliminate any remaining noise. Additionally, because we utilize offline rendering, we can generate these buffers without concerns about performance penalties.

Denoisers

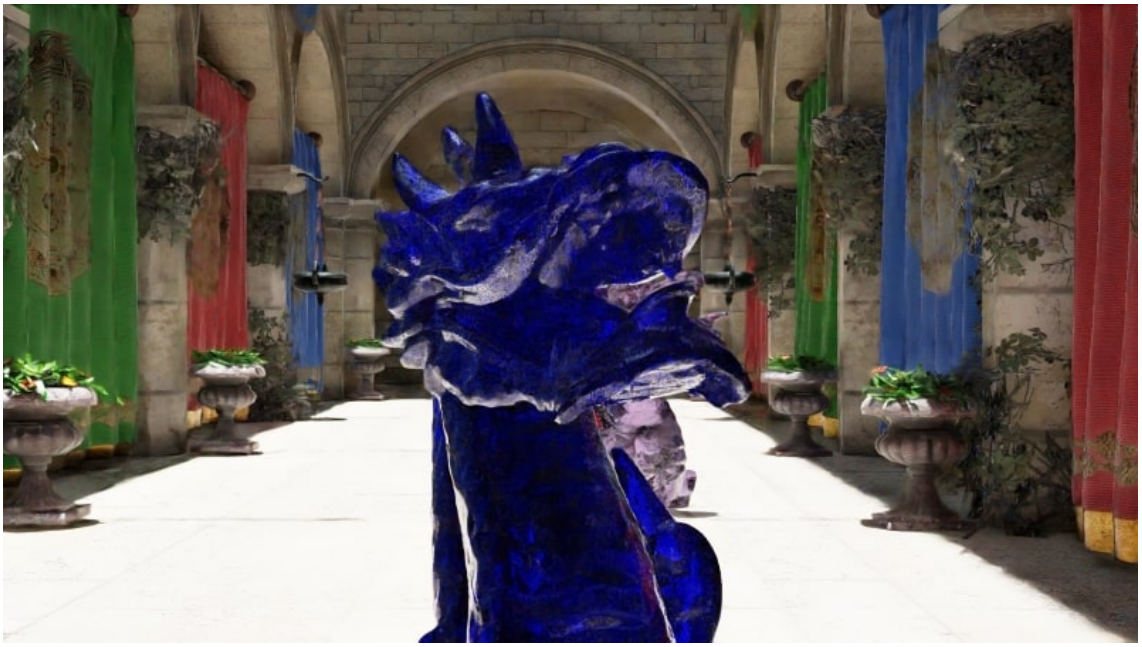
In path-tracing algorithms, rendering each pixel involves calculating numerous light rays to simulate global illumination, reflections, refractions, and other complex light interactions. However, to generate high-quality images with path-tracing, a vast number of rays must be traced per pixel, which increases the computational cost and time. When fewer samples per pixel (SPP) are used to reduce computation time, the result is a noisy image due to insufficient light samples. This is where denoisers play a critical role.

The primary purpose of a denoiser is to reduce the noise in the output image while preserving important details, such as edges, textures, and lighting nuances, without needing to increase the sample count. By applying denoising techniques, it becomes possible to generate visually pleasing images at much lower sample counts, significantly improving rendering efficiency and performance. Denoisers are especially crucial in real-time applications and in scenarios where rendering time is a critical factor, such as in video games or real-time simulations.

Denoisers typically work by analyzing the noisy rendered image and using statistical or machine learning models to predict and reconstruct the final image. Many modern denoisers leverage information from auxiliary buffers, such as normal maps, depth maps, and albedo maps, which provide geometric and material information of the scene. This additional data helps the denoiser differentiate between true de-



(a) Noise image



(b) Denoise image

Figure 2.13: (a) Sponza palace image rendered with 1SPP. (b) Image with noise removed using the Optix Denoiser.

tails and noise, enabling it to selectively smooth out noisy regions while preserving sharpness in the areas that contain important visual details.

One of the key advantages of using denoisers is the significant improvement in rendering performance. By allowing for lower sample counts without compromising visual quality, denoisers enable faster rendering times. This is particularly beneficial in real-time applications like video games and simulations, where maintaining high

frame rates is essential. Denoisers also make path-tracing more resource-efficient by reducing the computational load on hardware, particularly GPUs, allowing for better utilization of system resources. This balance between quality and performance makes denoisers a crucial component in modern rendering pipelines.

However, denoisers are not without their drawbacks. In some cases, they can cause a loss of fine detail, particularly when applied too aggressively. This can lead to images appearing overly smooth or lacking subtle textures and lighting variations that contribute to realism. Additionally, denoisers can sometimes introduce visual artifacts, such as ghosting or smearing, especially in scenes with complex lighting or high-frequency details. Moreover, many denoisers depend on auxiliary data like depth maps or normal information, and inaccuracies in this data can result in suboptimal denoising outcomes. These challenges mean that denoisers must be carefully tuned to balance noise reduction with the preservation of important visual details.

Common denoising techniques include:

- Spatial Filtering: This technique applies filtering operations over the pixel neighborhoods to smooth out noise while maintaining important features like edges.
- Temporal Filtering: Used in animations or real-time applications, temporal filtering takes into account information from previous frames to stabilize noise across time and prevent flickering artifacts.
- Machine Learning-Based Denoisers: These denoisers are trained on large datasets of noisy and clean images, learning how to distinguish between noise and valid details, and can deliver highly accurate results.

To eliminate such noise, many denoisers [37, 38, 39] have been proposed. Particularly effective in reducing noise, is Nvidia's Spatiotemporal Variance-Guided Filtering (SVGF) [37]. SVGF reprojects previous frames to increase the number of samples in the filter input. This both improves temporal stability and is an indicator of the per pixel variance. The size of the filter is determined based on the per-pixel variance. SVGF's main limitation is that it only works on the primary rays and as a result, noise from e.g., reflections remains, inducing temporal blur in the denoised frame.

An extension of the SVGF, attempting to eliminate temporal blur, is the Adaptive-Spatiotemporal Variance-Guided Filtering (A-SVGF) [38]. Temporal blur appears when, for example, a light source or light from a reflection, that has ceased to exist, appears still in subsequent frames, usually as a *trail of light*. To eliminate temporal blur the A-SVGF filter adapts the per-pixel temporal accumulation factor based on

an estimation and reconstruction of the sparse temporal gradients.

A deep learning-based approach for denoising PT by Nvidia is based on recurrent autoencoders (RA) to improve temporal stability [39]. The RA considers information from the normal & depth channels of neighboring pixels to both increase the performance and the image quality of the output (Figure 2.13). Another approach to speed-up path tracing is to render at a much lower resolution and then use Nvidia's deep learning super-sampling technique (DLSS) [40]. DLSS uses machine learning to upscale a low resolution frame to a higher resolution one.

In this thesis, we employ a denoiser during the pre-rendering of images for our emulated foveated path tracing. This approach allows us to determine the perceptibility thresholds of our foveated model when the samples per pixel are set to 16 sample per pixel and denoiser enable to eliminate the remaining noise.

2.3.3 Sphere-Tracing

Sphere tracing was originally proposed to render implicit surfaces [5] described by signed distance functions (SDFs). SDFs can outline 3D objects or procedural 3D worlds, fractals [28] as well as generate complex visual effects [41]. SDFs, in traditional rasterization, improve displacement mapping [29] as well as rendering small-scale details with complex topology in a fragment shader with a few GPU instructions and little memory. Implicit neural representation uses learned SDFs or occupancy fields to represent scene geometry by training a neural network [30, 31, 32, 42]. The networks learn from sparse images or point clouds to reconstruct objects, finding the SDF that matches the target object. Manually designing 3D objects and scenes is also possible, by combining simple SDFs for basic primitives (circles, boxes, cones, etc.) using union, difference & intersection operators to create intricate 3D scenes [1]. Distortion or noise can be additionally applied as well as combining primitives. Although complex, dedicated tools can automatically do this [43].

Contrary to ray tracing which finds the closest triangle that intersects with a ray, in sphere tracing rays traverse the world propagated based on their geometric distance from the closest *implicit* surface, as described by a signed distance function (SDF): $R^3 \Rightarrow R$ representing the distance from the boundary of an implicit surface $f^-1(0)$. The SDF is a scalar field mapping each point in 3D space to a scalar value, representing the distance of that point to the implicit surface. The points *on* the implicit surface have zero distance, while the points *inside* or *outside* the surface has a negative or positive distance respectively:

Algorithm 1 Hart's Sphere Tracing algorithm [15]

Input: $\boldsymbol{f}$: $R^3 \rightarrow R$, ϵ, t_{max}, s_{max}
Output: $t \geq 0$ surface properties
for $iteration = 1, 2, .., s_{max}$ **do**
    $p = r_o + r_d * t$
    $h = \boldsymbol{f}(p)$
    if $h.x \leq \epsilon \parallel t \geq t_{max}$ **then**
        break
    end if
    $t+ = h.x$
end for

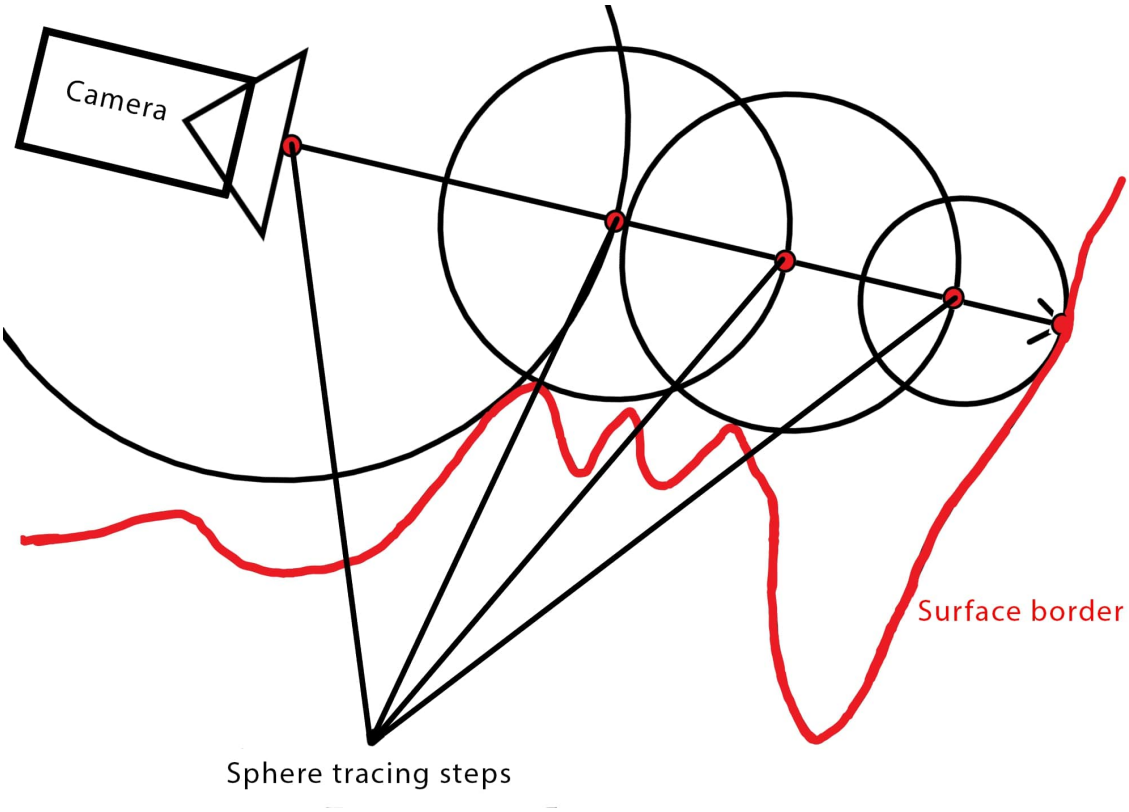


Figure 2.14: The rays traverse the world by propagating based on their geometric distance from the closest implicit surface, as described by a signed distance function (SDF).

$$SDF(\boldsymbol{x}) = \begin{cases} -dist(\boldsymbol{x}, f^{-}1(0)), \text{ray inside of } f^{-}1(0) \\ +dist(\boldsymbol{x}, f^{-}1(0)), \text{ray outside of } f^{-}1(0) \end{cases}$$

The number of steps required for a ray to intersect with an implicit surface varies depending on scene complexity. The step size is adjusted based on the distance returned by the SDF.

Ray casting algorithms simulate how light interacts with the environment by

taking into account material properties. Depending on the surface representation, whether using a polygonal surface or a signed distance function (SDF) approach, the method for tracing intersections between the ray and the surface varies. This distinction allows for flexibility in rendering techniques, ensuring accurate light transfer and realistic visualization across different surface types. To create realistic effects such as reflections and shadows for implicit surfaces, we can implement a recursive ray-tracing algorithm. For achieving full global illumination, the path-tracing approach is highly effective.

Previous research attempted to reduce the step count [10, 44, 8] by over-relaxing the step size by a fixed amount and thus reducing the total step count of the ray based on the error at full resolution during the rendering pass, leading to increased performance. In this thesis, we improved Sphere tracing utilizing an approximated distance calculated from an inverted pyramid resolution multi-pass rendering approach increasing performance without introducing artefacts and further improved it by utilizing operators to provide a relaxed but accurate approximated distance using only a lower resolution buffer.

Sphere Tracing for Soft Shadows

Soft shadows are an essential component in achieving realistic lighting effects in computer graphics. One effective method for calculating soft shadows is through the use of sphere tracing, a technique that approximates the distance to the nearest surfaces in a scene to determine light attenuation. The following algorithm [45] offers a robust approach for rendering soft shadows efficiently.

Algorithm 2 Sphere tracing soft shadows [45]

Input: $\boldsymbol{f}$: $R^3 \rightarrow R$, r_o, r_d, ϵ, t_{max}, S_{max}, k
Output: Shadow attenuation
res = 1.0
for $iteration = 1, 2, .., S_{max}$ **do**
    $p = r_o + r_d * t$
    $h = \boldsymbol{f}(p)$
    if $res \leq \epsilon$ **then**
        return 0.0
    end if
    $res = min(res, k * h/t)$
    $t+ = h.x$
    if $t \geq t_{max}$ **then**
        break
    end if
end for
return res

The sphere tracing algorithm for soft shadows operates using several key input parameters. The first is the distance function ($\boldsymbol{f}$), which maps points in 3D space to a scalar value representing the signed distance to the nearest surface. A negative value from this function indicates that the point lies inside a surface, whereas a positive value denotes an exterior point. The ray origin (r_o) defines the starting point of the ray, typically positioned at the light source, while the ray direction (r_d) is a vector that specifies the direction in which the ray is cast.

In addition, there is a threshold (ϵ) that determines when to terminate the ray tracing early if further calculations would not significantly affect the shadow result. The algorithm also incorporates a maximum trace distance (t_{max}) to prevent the ray from extending indefinitely, thereby ensuring computational efficiency. Another important parameter is the maximum number of iterations (S_{max}), which restricts the iterations in the ray tracing process to maintain control over the computational resources spent. Finally, a scaling factor (k) is utilized to scale the shadow attenuation based on the distance to the nearest object.

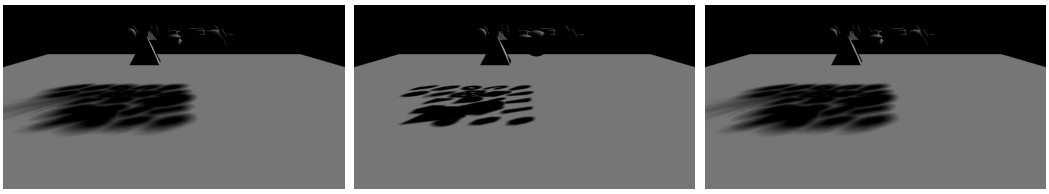
The output of the algorithm is the computed shadow attenuation, which signifies the degree to which the light is obscured due to occlusions.

The algorithm begins with the initialization of the shadow attenuation result (res) to 1.0, indicating full illumination. It then enters a loop that can iterate up to (S_{max}). Within each iteration, the algorithm calculates the current position (p) along the ray using the equation ($p = r_o + r_d \cdot t$). The distance function is evaluated at this current point to determine the distance to the nearest surface, denoted by ($h = \boldsymbol{f}(p)$).

To enhance computational efficiency, the algorithm checks if the result (res) is less than or equal to (ϵ). If this condition is met, the algorithm terminates early and returns a shadow value of 0.0, indicating full shadowing. If early termination does not occur, the algorithm updates the shadow attenuation value by calculating ($res = \min(res, k \cdot h/t)$). This calculation reflects the minimum shadow contribution based on the distance to the nearest surface.

Subsequently, the ray's distance (t) is incremented by (h.x), effectively moving it forward in space according to the nearest surface distance. The algorithm also checks if (t) exceeds (t_{max}); if so, it breaks out of the iteration loop. Finally, after completing the iterations or terminating early, the final shadow attenuation value (res) is returned.

In conclusion, this algorithm effectively captures the nuances of soft shadows by tracing rays in a scene, evaluating distances to surfaces, and providing a flexible



(a) k = 8 (b) k = 32 (c) k = 128

Figure 2.15: Soft shadows "hardness" can be controlled using different k value.

mechanism to account for occlusions. By incorporating soft shadows into rendering systems, we enhance the visual fidelity and realism of synthetic images, thus contributing to advancements in computer graphics.

In this thesis, our Signed Distance Function (SDF) scaling and early termination strategy for improving sphere tracing can be adapted to shadow tracing. This approach reduces rendering time by approximating the traveled distance of a ray based on low-resolution buffers, thus optimizing the efficiency of the shadow computation process.

Sphere Tracing Acceleration Techniques

As scene complexity increases (*more* SDFs describing the scene), the sphere tracing step size generally *decreases*, as there are more surfaces *closer* to the ray's endpoint increasing the step size, but also, generally, more SDF distance evaluations are also required. As a result, a ray will need to be traced for significantly more steps until it converges to a surface, increasing computations overall and resulting in a drop in performance. To achieve real-time rendering, several strategies aimed at accelerating sphere tracing have been presented:

The first family of strategies aims at reducing the required number of steps. Keinert et al. [8] controlled step advancement by multiplying the step size using a scene-specific fixed multiplier ranging between 1 and 2, which enhances tracing speed but the multiplier has to be manually selected. This scene-specific configuration limitation prompted subsequent research by Bán et al. [46], to propose a dynamic step adjustment based on a function including the rate of change of the previous step size and the current step. Bálint et al. [10] further refined [46] by constructing linear distance approximations of rays to SDFs; the method is particularly effective for planar surfaces. Cone tracing by Bán et al. [47] and super sample anti-aliasing by Chubarau et al. [3] have further improved rendering quality and efficiency. A method utilising local Lipschitz bounds [48], has demonstrated promising results in reducing computational overhead by avoiding calculating distances that don't intersect with a ray. All aforementioned approaches work well for older GPUs, that

had large batch sizes, meaning that pixels that need to perform more steps would force pixels that needed less steps to stall. For modern GPUs, these methods provide limited performance improvements while still generating artifacts.

The second family of strategies leverages acceleration structures to accelerate ray traversal towards scene points that contain surfaces. Söderlund et al. [49] and Balint et al. [9] propose the use of a grid density field where each voxel of the grid has a pre-calculated signed distance to the nearest SDF at its eight corners. They then compute the distance between a ray and the surface defined by trilinear interpolation of signed distances at corners of a voxel. The major drawback of these approaches is that the rendering quality and increased memory usage of the produced volumetric grid relies on a step size for the grid that depends on the scene's structure and complexity.

In this thesis, we employs low resolution render targets [50] to approximate ray distances in the 3D environment. Our first approach [51], combine several buffers in a pyramidal render target hierarchy and filtering to spawn rays closer to surfaces resulting in fewer steps overall. This method offers a balance between computational resources (less memory than grids) and similar rendering quality while being faster than the other methods. However, some artifacts still remain and requires a complex hierarchy of render targets (high memory cost) and filtering (high computational cost). Our second approach [52] avoid the pyramidal render target by applying a two pass pipeline. In the first pass, the sphere tracing is applied at a low resolution by adjusting the SDFs size and reducing the accuracy of the sphere tracing to generate a single low-resolution buffer in order to spawn rays closer to surfaces in the second pass resulting in fewer steps overall by avoiding the need for filters.

2.4 Foveated rendering

Foveated rendering is a technique in computer graphics that optimizes the rendering process by taking advantage of the human eye's varying sensitivity to detail across its field of view [1]. The human visual system perceives high detail in the central vision (the fovea) and lower detail in the peripheral vision. Foveated rendering leverages this by rendering the highest level of detail only in the foveal region, while progressively decreasing the level of detail and resolution in the peripheral regions. This selective allocation of computational resources significantly reduces the overall processing power required, allowing for more efficient rendering. The technique is particularly beneficial in applications like virtual reality and gaming, where high

[1] Mania, K., McNamara, A., & Polychronakis, A. (2021). Gaze-aware displays and interaction. In ACM SIGGRAPH 2021 Courses.

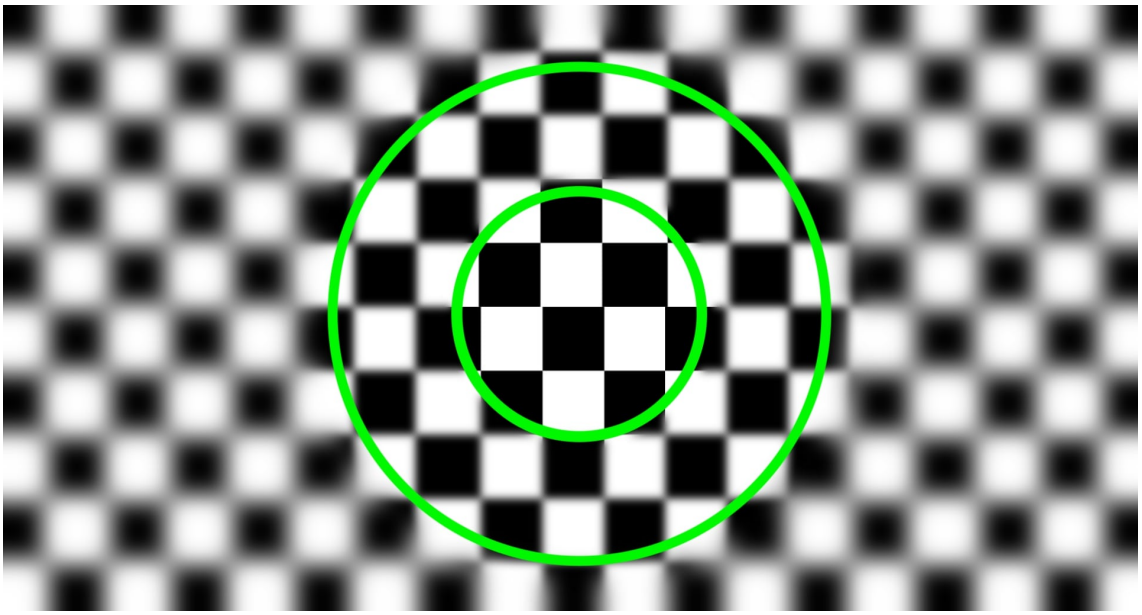


Figure 2.16: Visualisation of the foveal (inner circle), middle and outer peripheral zones. The placement of the boundaries of the zones (two green circles) are parameters of our experiment. Foveation exaggerated here, for visualisation.

frame rates and real-time performance are crucial. By integrating real-time gaze tracking, foveated rendering can dynamically adjust the focus area based on where the user is looking, ensuring that the perceived image quality remains high without unnecessary computational overhead.

2.4.1 Foveated Streaming

An early foveated compression mode proposed by [53] was a foveated video streaming multi-resolution pyramid video code/decoder. Each pyramid level is motion compensated, multi-resolution pyramid coded, and thresholded / quantized based upon human contrast sensitivity as a function of spatial frequency and retinal eccentricity. The final lossless coding includes zero-tree coding. [54] instead propose a foveal visual quality metric, the signal-to-noise ratio, to find the best compression and rate control parameters for a given target bit rate. [55] to test foveated compressing algorithms, they create a framework to reconstruct video of various video coding formats (H.263, MPEG-4). [56] propose more foveation filtering (local bandwidth reduction), motion estimation, motion compensation, video rate control, and video post-processing for foveated compression. These studies on Foveated video streaming quality assessment consider only static foveation mechanisms. [57] propose a model that integrates visual attention into the foveation mechanism. They build a wavelet-based distortion visibility measure to develop an attention-driven foveated video quality (AFViQ) metric. AFViQ exploits adequately perceptual visual mechanisms in video quality assessment.

[58] propose a technique for video streaming service over the internet using an ordinary webcam by informing the server of the user's real-time gaze. Based on that, they develop a multi-resolution video coding to pre-code a video in a small number of copies for a given set of resolutions designed to match the error performance of an eye tracker built using commodity webcams. [59] similar create a real-time 360^o Video foveated stitching framework. The scene is rendered in different levels of detail. The algorithm takes videos from multiple cameras as input, combined with measurements of human visual attention (i.e., the acuity map and the saliency map), which results in fewer pixels calculated. The growth in gaming streaming requires low latency due to bandwidth limits, especially in competitive online games where the latency must be low. [60, 61] proposed a foveated video streaming system for cloud gaming that adapts video stream quality by adjusting the encoding parameters on the fly to match the player's gaze position. The reconstruction happens by selecting the Video that closely matches this sparse input stream of pixels based on learning from natural videos and fetching them to the neural network. [62] instead uses neural networks to create a foveated video streaming (DeepFovea) that recreates the frame by a small fraction of pixels provided every frame.

The peripheral vision of the human visual system can see a pair of physically different images as same. These pairs are usually called metamers. [63] exploit these metamers for foveated rendering for near-eye displays. Using metamers instead of blurring in the periphery, foveated quality is improved compared to state-of-the-art foveated methods. Calculating metamers is highly costly. [63] propose a type of statistics suited for realistic real-time rendering using a GPU to compute these statistics, and they can compress these statistics and transmit at low bandwidth. In addition, the decompression can happen in real-time. This method suffers from fewer artifacts compared to common blurring.

In this thesis, we focus on improving real-time rendering performance by reducing the rendering time while the previous approach focus on reducing the latency from streaming render images thought the network.

2.4.2 Foveated Rasterization

Early foveated rendering pipelines achieved performance gains by dynamically manipulating geometric level-of-detail (LOD) based on eccentricity, often resulting in aliasing and system lag issues [64, 65]. Based on real-time gaze tracking, LOD has been reduced in unattended areas while the users performed simple tasks [66, 67]. The most common drawback of these systems was the appearance of visual artifacts due to the inability to maintain stable frame rates. [68] instead of using an eye tracker, use saliency models to predict the gaze position. [68] develop these saliency

models from low-resolution images by extracting features such as luminance and contrast, or from task maps, [69, 70], or high-level context [71]. Using low-level saliency models without any other information usually fails to predict the gaze direction because of the context [72]. Details on early work on gaze-contingent displays can be found in [73, 74, 17]. These techniques did not manipulate spatial resolution based on eccentricity, ignoring the potential speed up for scenes with high per-pixel shading cost.

Later work attempted to reduce also the resolution in the periphery to achieve better performance. [75] create three layers (inner, middle, and outer) where resolution, LOD, and refresh rate drop when moving to the middle and outer layers. This method increases the performance, but artifacts appear in the periphery due to the low point-sampling density and the system latency from the half refresh rate. [11] to solve this issue use a post-process Gaussian blur with a progressively increasing filter-width based on eccentricity. They discovered that a large radius of blur induces a sense of tunnel vision. To solve the problem, they enhance the constant in the periphery, which allows them to increase the amount of blurring in the periphery before any noticeable tunnel vision appears. [7] has created a foveated render pipeline of two passes. The first pass transforms the Cartesian coordinates of the render frame to kernel log-polar coordinates. This transformation leads to a lower resolution buffer. In the second pass, they invert the log-polar transform and perform anti-aliasing to map the reduced resolution to the target resolution. By applying this method, the sample stays more coherent in the gaze direction without having to perform multiplying render passes like [75]. Their method improves performance, but the major drawback is aliasing issues appearing in the periphery. [76] propose a new foveated method for the dominant eye. [76] render the scene with higher eccentricity compared to the non-dominant, based on the observation that the human visual system prioritizes scene perception of one -dominant- eye over the other.

Previous work has shown that temporally stable approaches, both blurred and aliased, were less noticeable than temporally volatile approaches when down-sampling an image [77]. The boundaries between zones in the periphery region were found to have low visibility. The replacement from a low to a high-resolution image was not detectable after testing, if presented for less that 40 ms. The processing time of our foveated method does not surpass the time limit of 40 ms, in order for the drop in quality in the periphery region not to be perceptible after the rendering process. In our work, the rendering zones, e.g. fovea, periphery and outer, are blended.

Most of the above foveated rendering methods model the sensitivity as a function of eccentricity by ignoring the content of the 3D environment. [6] explored the im-

portance of luminance. They propose a luminance-contrast-aware foveated rendering technique. They develop a predictor capable of finding the foveated parameters using a low-resolution frame and calculate the luminance based on that. [78] used pixels from previous frames by spatiotemporally re-projecting. The artifacts and disocclusions due to the re-projection are reevaluated based on a new proposed confidence value. The confidence value is calculated based on the eccentricity, contrast, and holes-size that appears in the image. So areas with low confidence values are redrawn, which lowers the primitive processing and shading costs. [79] propose a perceptual technique based on the observation that the human eye can detect high frequencies for specific eccentricities when these are absent due to the production of low frequencies images by the foveated rendering. Their technique does not require reproducing these frequencies; instead can replace them with noise generated in a less expensive post-processing step.

2.4.3 Foveated Ray-Casting Rendering Techniques

Ray-tracing pipelines, unlike rasterization pipelines, allow for samples for specific parts of the screen [80], [13], [12] without rendering multiple passes to produce layers of lower quality and to interpolate the produce layers as presented to [75]. [80] propose a gaze-contingent rendering based on ray tracing where near object silhouettes and high contrast areas increase the samples per pixel. Similarly, [81] creates a system that results in visual artifacts appearing in the periphery and not yet evaluated by users. [82] propose a foveated ray tracing that uses a sampling mask to generate a non-uniformly distributed set of pixels. They use the triangulation method with barycentric interpolation to reconstruct the regular Cartesian image. Also, to avoid the appearance of artifacts, temporal anti-aliasing is applied. [13, 12] achieve significant cost reductions by generating rays based on a linear model that does not match the pattern of the foveal receptor density. For the pixels that do not generate rays, they use a support image and re-projected frames to reconstruct the frame. Temporal artifacts appear due to re-projected frames, and some errors appear due to the supported image. [83] extends [13] his foveated pipeline by integrating it with a gaze-contingent depth-of-field (DoF) filter. The DoF is used to conceal the rendering artifacts produced from their foveated method.

A theoretical study has shown that foveated path-tracing rendering in VR is potentially feasible as total ray counts could be reduced down to 94% [84]. To demonstrate that the potential ray reduction could be significant, the authors, based on a human visual acuity approximation function, estimate how many rays need to be generated. The study does not describe how the non-calculated (empty) pixels will be re-constructed but only focuses on the potential performance gains from this

reduction, without evaluating their results in a user study. In our work, we first develop an emulator to measure detectability thresholds for foveated path tracing, then provide a method to reconstruct missing pixels due to down-sampled images, and, finally, we calculate potential performance gains in desktop displays.

[85] create a foveated method for the path-tracing algorithm. They generate rays for path-tracing [36] based on polar coordinates similar to [7] and a probability model similar to the foveal receptor density. Also, the probability model was used to reduce the generated rays as eccentricity increased. Also, they apply the denoiser filter to polar coordinates and achieve the same quality results as if they were denoising in the target resolution in Cartesian coordinates. Then, they follow the same process to remap to Cartesian coordinates like [7]. This process increases path-tracing performance due to the lower resolution when rendering in polar space. We [50] instead create an emulated foveated path-tracing to study the visual perception of foveated path-tracing when the path-tracing settings are high (more than one sample per pixel, a high number of bounces). In addition, our method shows that the potential performance gains are to be above 2x - 3x.

Foveated rendering has been applied in the volume rendering algorithm also. [86] propose a model that adapts samples based on Linde-Buzo-Gray sampling and natural neighbor interpolation. Due to peripheral vision being sensitive to contrast changes and movement, they apply a temporal filter to attenuate undersampling artifacts. Their method focuses on the fovea without considering the peripheral vision properties except for the visual acuity. While Volume rending is similiar to sphere-tracing the step-size is prefixed and the intersections are caclulated based on density of topology compare to sphere tracing where are calculated when a ray reach too close to the surface.

In this thesis, we developed two foveated models one for path-tracing and one for sphere tracing. For path tracing, we generate an extra ray for a set of pixels for the periphery and outer zones. Then, we reconstruct the frame using spatial data of the rendering frame and apply a post-processing effect to create an imperceptible frame based on the user's gaze. Our method does not suffer from temporal artifacts. The aliasing issues in the middle periphery and outer periphery are eliminated employing a blur effect. For sphere tracing our foveated model despite using edge detection similarly to previous studies (e.g., [87, 6]), has some crucial differences: (i) previous approaches focus either on rasterization or ray tracing meshes, whereas our methodology focuses on implicit surfaces described by SDFs; (ii) rasterization and ray tracing can benefit from specific hardware acceleration capabilities of GPUs whereas implicit rendering in its current form, cannot; (iii) we focus specifically on reducing the step count during sphere tracing by positioning the ray closer to the

implicit surface thus reducing the number of steps required to accurately render the surface using an innovative inverted pyramid multi-resolution technique; (iv) we do not rely on the use of reprojected frames [13], which can introduce errors. We focus on accelerating SDF rendering using foveation, enabling their rendering in VR headsets, that is otherwise severely constrained due to performance limitations.

Chapter 3

Overview of Unity Game Engine

The Unity engine stands as a cornerstone in contemporary game development and interactive media creation, offering a robust suite of tools and functionalities that empower developers to bring their creative visions to life with unprecedented flexibility and efficiency. Founded on principles of accessibility and versatility, Unity caters to a diverse spectrum of users, from seasoned professionals to aspiring enthusiasts, by providing a user-friendly interface coupled with powerful capabilities for 2D, 3D, Virtual, and augmented reality (AR) content creation.

At its core, Unity streamlines the game development pipeline through its intuitive interface and comprehensive feature set. Developers harness the engine's capabilities to construct immersive worlds, intricate gameplay mechanics, and captivating visual effects, all within a unified environment that encourages rapid iteration and experimentation. The engine's asset pipeline facilitates seamless integration of multimedia elements, including 3D models, textures, animations, audio files, and more, empowering creators to realize their artistic visions with precision and fidelity.

Moreover, Unity's cross-platform compatibility ensures widespread accessibility across various devices and platforms, ranging from desktop computers and consoles to mobile devices and virtual reality (VR) headsets. This versatility extends to its deployment capabilities, allowing developers to publish their creations across multiple platforms with minimal additional effort, thereby maximizing reach and potential audience engagement. With its extensive ecosystem of plugins, asset store resources, and community-driven support, Unity fosters a collaborative and dynamic environment where developers can leverage shared knowledge and resources to overcome challenges and achieve their goals. As such, Unity stands as a pillar of innovation and creativity in the realm of interactive media, empowering creators worldwide to realize their imaginative visions and shape the future of digital entertainment.

In addition to its widespread adoption in game development and interactive media production, Unity serves as a valuable tool in the realm of research, facilitating the exploration and analysis of various phenomena across diverse fields and disciplines. Researchers harness Unity's versatile framework and extensive feature set to design and implement interactive simulations, experimental environments, data visualization tools, and more, enabling them to conduct studies and investigations with unprecedented depth and precision.

Unity's adaptability and customizability make it well-suited for a wide range of research applications, including psychology, neuroscience, education, healthcare, architecture, engineering, and beyond. Through the creation of immersive virtual environments, researchers can simulate real-world scenarios, manipulate variables, and observe participant responses in controlled settings, providing invaluable insights into human behavior, cognition, and decision-making processes. Moreover, Unity's support for data integration and analysis enables researchers to collect, store, and analyze vast amounts of experimental data with ease, facilitating quantitative analysis and statistical inference.

Furthermore, Unity's cross-disciplinary appeal and accessibility democratize research by lowering barriers to entry and fostering collaboration among researchers from disparate fields. Its intuitive interface, extensive documentation, and vibrant community ecosystem empower researchers with varying levels of technical expertise to leverage the engine's capabilities for their research endeavors. Whether conducting experiments, prototyping solutions, or visualizing complex data sets, Unity offers researchers a powerful and versatile platform for advancing knowledge, driving innovation, and addressing pressing challenges across diverse domains.

3.1 Scripting using C#

Scripting in Unity is a fundamental aspect of game development, enabling developers to create interactive and dynamic content by writing custom behaviors in C#. Unity's scripting API provides extensive access to the engine's capabilities, allowing control over game objects, physics, animation, user input, and more. Scripts are typically attached to game objects as components, and they interact with other components and objects in the scene. Developers write scripts in an integrated development environment (IDE) such as Visual Studio, leveraging the power of C# to create robust and maintainable code. Unity's MonoBehaviour class serves as the base class from which all scripts derive, providing essential methods like 'Start()', 'Update()', and 'OnTriggerEnter()' for managing object lifecycles and handling events. Through scripting, developers can implement game logic, control

character behaviors, respond to player actions, and manage game states, making it a versatile and powerful tool for bringing interactive experiences to life. The combination of Unity's user-friendly interface and the flexibility of C# scripting empowers developers to create complex and engaging games efficiently.

3.2 Eye-tracker and Head-tracker Integration in Unity Engine

In the context of a thesis, detailing the calibration procedure for the Tobii Eye Tracker 4C is crucial to ensuring the accuracy and reliability of gaze tracking data. The calibration process begins by initiating the software interface of the Tobii Eye Tracker within the Unity environment or the designated application platform. Through the software, users are guided through a series of steps to perform calibration. These steps typically involve the presentation of visual stimuli, such as calibration points or targets, displayed on the screen.

During calibration, participants are instructed to fixate their gaze on each presented point sequentially. As participants focus on each calibration point, the Tobii Eye Tracker captures and analyzes their gaze data, mapping the relationship between the displayed points and the user's eye movements. It's imperative to emphasize to participants the importance of maintaining a stable head position throughout the calibration process to obtain accurate calibration results.

Upon completion of the calibration procedure, the Tobii Eye Tracker adjusts its internal parameters and algorithms based on the collected data. This calibration data is then utilized to enhance the accuracy and precision of gaze tracking during subsequent interactions with the application. By meticulously documenting and implementing the calibration process within the thesis framework, researchers can ensure the validity and reliability of gaze tracking data, thereby contributing to the overall robustness of the study's findings and conclusions.

3.2.1 Eye Tracking Data

In the foundational integration of eye tracking data acquisition within the Unity environment, our methodology extends to utilizing this gaze data in real-time rendering processes. Specifically, the eye tracking data captured from the Tobii Eye Tracker 4C is sent to a compute shader, enabling adaptive rendering techniques based on user attention within the virtual environment.

As participants interact with the scene, the gaze coordinates, fixation points, and

other relevant metrics are continuously processed and transmitted to the compute shader. This integration allows the rendering pipeline to adjust the visual fidelity of different screen areas depending on where users are focusing their attention. For instance, regions of the screen that receive prolonged gaze can be rendered with higher detail, while areas that are not actively attended to may be rendered with lower fidelity, optimizing computational resources without sacrificing the overall user experience.

By employing this technique, our application not only enhances visual performance but also creates a more immersive and responsive interaction for participants. The real-time adaptability of the rendering process ensures that resources are allocated efficiently, providing clear visual cues and maintaining engagement in the virtual environment. This synergy between eye tracking and compute shaders exemplifies the potential for leveraging user attention to augment and refine interactive experiences in Unity.

3.3 Compute Shaders

Compute shaders in Unity provide a powerful tool for performing complex calculations and parallel processing tasks directly on the GPU, significantly enhancing performance for certain operations. Unlike traditional shaders, which are primarily used for rendering graphics, compute shaders are designed to handle general-purpose computation. This allows developers to leverage the massive parallel processing power of modern GPUs for tasks such as physics simulations, image processing, and large-scale data manipulation. By offloading these tasks to the GPU, compute shaders can dramatically reduce the CPU workload and improve overall application performance.

In Unity, compute shaders are written in HLSL (High-Level Shading Language) and integrated into your project through a ‘.compute‘ file. They are executed through a ‘ComputeShader‘ object, which provides methods to set parameters and dispatch the shader for execution. The flexibility of compute shaders in Unity allows for creative and efficient solutions to problems that require high computational power. For example, compute shaders can be used to implement advanced graphics techniques such as real-time global illumination, fluid dynamics, or complex particle systems. The ability to handle large-scale computations on the GPU opens up new possibilities for creating more realistic and interactive virtual environments in Unity.

3.3.1 Dispatch Compute Shader

Dispatching a compute shader in Unity involves instructing the GPU to execute the shader code across a specified number of threads organized into thread groups. This process begins by defining the compute shader in HLSL, specifying the number of threads per group using the 'numthreads' attribute.

Here's an example HLSL compute shader for a basic raymarcher:

```
// Raymarcher.compute
#pragma kernel CSMain

// Number of threads in each group
[numthreads(8, 8, 1)]
void CSMain (uint3 id : SV_DispatchThreadID)
{
    // Get pixel coordinates
    float2 uv = (id.xy / float2(1024, 1024));
    // Normalize UV coordinates

    // Implement raymarching logic here...

}
```

In the Unity C# script, the compute shader is prepared by loading the shader file, setting any required parameters, and configuring buffers or textures. The 'Dispatch' method is then called with parameters corresponding to the number of thread groups to execute, calculated based on the total workload and the size of the thread groups. For example, if a texture of size 1024x1024 needs processing with 8x8 threads per group, the dispatch call would be 'computeShader.Dispatch(kernelHandle, 128, 128, 1)'. This instructs the GPU to run the shader code over 128x128 thread groups, effectively covering the entire texture.

```
// RaymarcherController.cs
using UnityEngine;

public class RaymarcherController : MonoBehaviour {
    public ComputeShader raymarcherShader;
    private RenderTexture renderTexture;

    void Start() {
        // Create a RenderTexture to store results
        renderTexture = new RenderTexture(1024, 1024, 0);
        renderTexture.enableRandomWrite = true;
        renderTexture.Create();
```

```
        // Set texture to the compute shader
        int kernelHandle = raymarcherShader.FindKernel("CSMain");
        raymarcherShader.SetTexture(kernelHandle, "Result", renderTexture);

        // Dispatch the shader
        raymarcherShader.Dispatch(kernelHandle, 128, 128, 1);
    }

    void OnRenderObject() {
        // Display the final output or use it in further processing
        Graphics.Blit(renderTexture, null as RenderTexture);

        // Example visualization
    }

    void OnDestroy() {
        renderTexture.Release();
    }
}
```

The dispatch process enables parallel computation, leveraging the GPU's ability to perform many calculations simultaneously, which is crucial for tasks requiring high performance, such as image processing, physics simulations, complex mathematical operations and rendering algorithms. Proper synchronization and resource management ensures that data dependencies are handled correctly and performance is optimized.

3.3.2 Passing Data in Compute Shader

Passing data to and from the GPU when using compute shaders is a crucial step that allows for efficient computation and rendering. This section discusses the various types of data that can be passed, the mechanisms used for data transfer, and provides example code that illustrates these concepts effectively.

Data Structures and Types

In Unity, data can be passed to compute shaders using several mechanisms, including:

- Compute Buffers: Used to store structured data, such as arrays or custom structs.
- Textures: Leveraging textures for 2D or 3D data, especially useful for image processing.

- Primitive Types: Simple data types like integers, floats, and vectors can also be passed directly.

The following example demonstrates how to create a compute buffer that holds structured data, set data values from the CPU, and then access this data within the compute shader. We begin by defining a simple struct to hold our data in C#:

```
// DataStructure.cs
[System.Serializable]
public struct VertexData {
    public Vector3 position;
    public Vector3 normal;
    public Vector4 color;
}
```

The C# script encompasses the setup of the compute shader, creation of the compute buffer, and passing of the structured vertex data.

```
// ComputeBufferExample.cs
using UnityEngine;

public class ComputeBufferExample : MonoBehaviour {
    public ComputeShader computeShader;
    private ComputeBuffer vertexBuffer;
    private int bufferSize = 100; // Number of data elements
    private VertexData[] vertexDataArray;

    void Start() {
        // Initialize vertex data array
        vertexDataArray = new VertexData[bufferSize];
        for (int i = 0; i < bufferSize; i++) {
            vertexDataArray[i] = new VertexData {
                position = new Vector3(Random.Range(-1.0f, 1.0f), Random.Range(-1.
                normal = new Vector3(0.0f, 0.0f, 1.0f),
                color = new Vector4(Random.value, Random.value, Random.value, 1.0f
            };
        }

        // Creation of ComputeBuffer
        vertexBuffer = new ComputeBuffer(bufferSize, System.Runtime.InteropService
        vertexBuffer.SetData(vertexDataArray);

        // Setting the buffer in Compute Shader
        int kernelHandle = computeShader.FindKernel("CSMain");
        computeShader.SetBuffer(kernelHandle, "vertices", vertexBuffer);

        // Dispatch the compute shader
        computeShader.Dispatch(kernelHandle, bufferSize / 10, 1, 1);
```

```
    }

    void OnDestroy() {
        // Release the compute buffer
        if (vertexBuffer != null) {
            vertexBuffer.Release();
        }
    }
}
```

Next, we define the compute shader that processes this data. In the shader, we will read the vertex data from the buffer and perform a simple operation (for example, transforming positions).

```
// ComputeBufferShader.compute
#pragma kernel CSMain

// Define the structured buffer
StructuredBuffer<VertexData> vertices;
RWStructuredBuffer<VertexData> outputVertices;

[numthreads(10, 1, 1)]
void CSMain (uint3 id : SV_DispatchThreadID) {
    VertexData data = vertices[id.x]; // Read from input buffer

    // Simple operation: Transform the position
    data.position += float3(0.1, 0.0, 0.0); // Move positions to the right

    outputVertices[id.x] = data; // Write result to the output buffer
}
```

Accessing Textures

In addition to buffers, textures offer a powerful way to represent and manipulate image data. Below is a simplified example illustrating how to set up a RenderTexture to be used in the compute shader for image processing.

```
// TextureExample.cs
using UnityEngine;

public class TextureExample : MonoBehaviour {
    public ComputeShader computeShader;
    private RenderTexture renderTexture;

    void Start() {
        // Create a RenderTexture for processing
        renderTexture = new RenderTexture(1024, 768, 0);
```

```
        renderTexture.enableRandomWrite = true;
        // Allow the compute shader to write to this texture
        renderTexture.Create();

        // Find the kernel in the compute shader
        int kernelHandle = computeShader.FindKernel("CSMain");

        // Set the RenderTexture as an output target for the compute shader
        computeShader.SetTexture(kernelHandle, "Result", renderTexture);

        // Optionally set any other parameters needed for the compute shader
        // computeShader.SetFloat("SomeParameter", value);

        // Dispatch the compute shader (e.g., 128 groups of 8x8 threads)
        computeShader.Dispatch(kernelHandle, 128, 96, 1);
    }

    void OnRenderObject() {
        // Blit the contents of the RenderTexture to the camera's screen
        Graphics.Blit(renderTexture, null as RenderTexture);
    }

    void OnDestroy() {
        // Release the RenderTexture when done
        if (renderTexture != null) {
            renderTexture.Release();
        }
    }
}
```

Compute Shader Code Using Textures

Now we can define a compute shader that will manipulate the texture data. In this example, we'll modify the colors of the texture based on some simple procedural functions.

```
// TextureShader.compute
#pragma kernel CSMain

// The output texture to be written
RWTexture2D<float4> Result;

// Main compute kernel
[numthreads(8, 8, 1)]
void CSMain (uint3 id : SV_DispatchThreadID) {
    float2 uv = id.xy / float2(1024, 768); // Normalized coordinates

    // Create a color gradient based on UV position
```

```
    float4 color = float4(uv.x, uv.y, 0.5, 1.0); // Simple gradient color
    Result[id.xy] = color; // Write to the output texture
}
```

This section has demonstrated various methods for passing data between the CPU and GPU when using compute shaders in Unity. By leveraging compute buffers, textures, and primitive data types, developers can efficiently manage and manipulate data during GPU computations, enabling complex rendering workflows and interactive graphics applications.

3.4 Sphere tracing with Compute Shaders

Sphere tracing is a powerful rendering technique used in computer graphics to visualize complex scenes defined by signed distance fields (SDFs). This technique allows for the rendering of intricate geometric shapes and scenes through the iterative evaluation of distances, providing high levels of detail and flexibility compared to traditional rasterization methods. In this section, we present the implementation of a raymarcher using compute shaders in Unity, highlighting the underlying principles, architectural decisions, and performance considerations.

3.4.1 Implementation

Sphere tracing differs from traditional ray tracing by successively evaluating the signed distance to the nearest surface along a ray's path. The core of the raymarching algorithm involves a loop that iteratively updates the position along the ray based on the computed distance until a surface is reached or a maximum ray distance is exceeded. This approach is particularly suited for rendering complex implicit surfaces defined by mathematical functions.

To implement the Sphere tracing, a compute shader was developed in HLSL (High-Level Shader Language). The shader defines a kernel that processes multiple threads in parallel, utilizing the capabilities of modern GPUs for high-efficiency computation. The following code snippet outlines the structure of the compute shader, Raymarcher.compute:

```
// Raymarcher.compute
#pragma kernel CSMain

// Texture for writing output
RWTexture2D<float4> Result;

// Signed distance function for a sphere
float SphereSDF(float3 p, float radius) {
```

```
    return length(p) - radius;
}

// Raymarching function
float RayMarch(float3 ro, float3 rd) {
    float d = 0.0; // Accumulated distance
    for (int i = 0; i < 100; i++) { // Maximum iterations
        float3 p = ro + rd * d; // Calculate current position
        float dist = SphereSDF(p, 1.0); // Evaluate SDF
        if (dist < 0.001) { // Surface hit threshold
            return 1.0; // Hit
        }
        d += dist; // Accumulate distance
        if (d > 50.0) break; // Exceed maximum distance
    }
    return 0.0; // No hit
}

[numthreads(8, 8, 1)]
void CSMain(uint3 id : SV_DispatchThreadID) {
    // Normalized device coordinates
    float2 uv = (id.xy / float2(1024, 768)) * 2.0 - 1.0;
    uv.x *= 1024.0 / 768.0; // Aspect ratio correction

    // Define ray origin and direction
    float3 ro = float3(0.0, 0.0, -3.0); // Camera position
    float3 rd = normalize(float3(uv, 1.0)); // Ray direction

    // Execute raymarching
    float hit = RayMarch(ro, rd);

    // Assign color based on hit result
    Result[id.xy] = hit > 0.0 ? float4(1.0, 0.0, 0.0, 1.0) : float4(0.0, 0.0, 0.0,
}
```

3.4.2 C# Script Integration

Integrating the compute shader within the Unity environment involves setting up a corresponding C# script to manage its execution and render results. The following C# code snippet demonstrates how to set up the compute shader and utilize a RenderTexture for output:

```
// RaymarcherController.cs
using UnityEngine;

public class RaymarcherController : MonoBehaviour {
    public ComputeShader raymarcherShader; // Assign the compute shader in the ins
```

```
    private RenderTexture renderTexture;

    void Start() {
        // Initialize Render Texture
        renderTexture = new RenderTexture(1024, 768, 0);
        renderTexture.enableRandomWrite = true;
        renderTexture.Create();

        // Find the compute shader kernel
        int kernelHandle = raymarcherShader.FindKernel("CSMain");

        // Set the Render Texture as an output target
        raymarcherShader.SetTexture(kernelHandle, "Result", renderTexture);

        // Dispatch the compute shader (16 groups of 16x16 threads)
        raymarcherShader.Dispatch(kernelHandle, 128, 96, 1);
 // Dispatch the compute shader (16 groups of 16x16 threads)
        raymarcherShader.Dispatch(kernelHandle, 128, 96, 1);
    }

    void OnRenderObject() {
        // Visualize the output by blitting the Render Texture to the screen
        Graphics.Blit(renderTexture, null as RenderTexture);
    }

    void OnDestroy() {
        // Release the Render Texture to prevent memory leaks
        renderTexture.Release();
    }
}
```

Explanation of the Integration Process:

- Initialization: In the Start() method, a RenderTexture is instantiated, which will store the output of the compute shader. The texture is set to enable random write access, necessary for the compute shader to modify its contents.
- Kernel Handle Retrieval: The kernel handle corresponding to the raymarching function CSMain is obtained using FindKernel(). This identifier is utilized for dispatching the shader.
- Setting the Texture: The SetTexture method links the RenderTexture to the compute shader, allowing pixel data to be output during shader execution.
- Dispatching the Shader: The Dispatch() method is called with defined thread group sizes. In this case, 128 groups of 8x8 threads are utilized, effectively processing the output resolution of 1024x768 pixels.

- Rendering: The OnRenderObject() method is invoked during Unity's rendering cycle, where the results from the compute shader are blit to the screen by using Graphics.Blit(). This operation renders the final image based on the data stored in renderTexture.
- Cleanup: In the OnDestroy() method, the RenderTexture resources are released to prevent memory leaks when the associated GameObject is destroyed.

3.4.3 Performance Consideration

Efficient utilization of compute shaders in a raymarcher significantly enhances rendering performance by leveraging the parallel processing capabilities of modern GPUs. The use of iterative distance evaluation requires careful balancing of the maximum number of iterations and the distance threshold to optimize for both performance and visual fidelity. The potential for further enhancements includes the integration of more complex SDFs, the implementation of acceleration structures, and the adaptation for real-time user interactions.

3.5 Chapter Summary

This chapter provides a comprehensive overview of the Unity engine, highlighting its significant role in contemporary game development and interactive media creation. It emphasizes Unity's robust tools and features that enable developers to create 2D, 3D, virtual, and augmented reality content with flexibility and efficiency. The user-friendly interface allows both professionals and beginners to navigate the game development pipeline seamlessly, integrating multimedia elements easily.

The chapter further explores compute shaders as a powerful tool for parallel processing on the GPU. It outlines the creation and dispatching of compute shaders for tasks like physics simulations and image processing, ultimately discussing the performance benefits and flexibility they provide for rendering techniques, including sphere tracing.

Finally, it illustrates the implementation of sphere tracing within compute shaders, explaining the algorithm's iterative nature and how it efficiently evaluates signed distance fields to render complex scenes. The chapter concludes with considerations for performance optimization and future enhancements in raymarching applications.

Chapter 4

Emulated Path tracing

4.1 Overview

In this chapter, we introduce our approach to emulating foveated path tracing, a method designed to optimize computational resources by reducing the number of rays fired based on fixation points and visual acuity [1]. We begin by discussing the concept of foveated rendering and how variations in ray processing can affect the quality and efficiency of path tracing systems. Building on existing research, we present our perceptual sandbox and articulate the methodology for reducing ray counts while maintaining visual fidelity. We delineate the model we developed to predict visual acuity based on eccentricity and discuss the dynamic zoning of output resolutions. Our methodology is validated through a series of experiments aimed at establishing the thresholds of foveation zones wherein visual quality remains acceptable. Finally, we analyze the computational gains that can be achieved through this method compared to traditional rendering techniques.

4.2 Implementation

In an ideal foveated path tracing system, the sum total of rays can be significantly reduced, based on the distance from the fixated point. Special care has to be taken to eliminate popping artifacts due to e.g., reduced ray bounces. Such a system does not currently exist.

In our work, inspired by [11], we develop a perceptual sandbox, by emulating a foveated path tracer. In a foveated path tracer, the aim is to reduce the total number of rays fired, and this, in principle, can be done in three ways: (i) reducing

[1] Polychronakis, A., Koulieris, G. A., & Mania, K. (2021). Emulating foveated path tracing. In Proceedings of the 14th ACM SIGGRAPH Conference on Motion, Interaction and Games.

the total number of samples-per-pixel, (ii) reducing the total number of ray bounces, and, (iii) reducing the total pixel count of the output buffer.

Reducing the total number of SPP was not a subject of this study, as modern denoisers perform exceptionally well with 1SPP, and as a result the main limiting factor for real time path tracing remains the enormous amount of rays required for high resolution buffers (even at 1SPP) and the secondary rays instantiated at geometry intersections to track ray bounces.

Figure 4.1: FPT with less secondary ray bounces in the middle and outer peripheral zones. Strong artifacts can be noticed, for example, the half transparent - half black dragon.

Ray bounce reduction enables significant performance gains. However, by reducing the bounces, quality drops remarkably especially for phenomena such as reflection and refraction (Figure 4.1). In a pilot study (see Sec. 4.3.2) we found that reducing the number of bounces for secondary rays is not a viable option for foveated path tracing as such ray elimination generates popping artifacts when, for example, transparent objects change colour as they pass through zones corresponding to different eccentricities.

In this study we investigated the perceptual repercussions and the potential to reduce computational complexity, by reducing the total rays fired by zoning the output buffer to different resolutions, similar to [75]. We develop a model for ray generation in PT based on the acuity of the HVS. Lanczos resampling [88, 89] is then applied to smoothly interpolate between different zones.

Due to current hardware limitations prohibiting real time path-tracing, we pre-

render the three buffers at different resolutions and emulate foveated rendering as a post-process based on eye tracking. We then perform three experiments to estimate conservative thresholds of eccentricity where the image zoning is detectable. Finally we perform an analysis of the expected performance gains for when such a system exists and can run in real time.

4.2.1 Emulating Foveated Path Tracing

The fovea has the highest acuity at determining visual detail. In the periphery and away from the fovea, the acuity of the HVS falls rapidly. We devise a probability model based on eccentricity (ω), from the fixated x, y location. Our model is based on the work of Mandelbaum et. al. [90] that measured the visual acuity for different values of eccentricity of the human eye. We fit to Mandelbaum's data an exponential function closely matching the measured dataset with an accuracy of over 95%:

$$V(\omega) = 0.90964e^{\frac{\omega}{2.9661}} + 0.00792 \tag{4.1}$$

Based on equation 4.1 we then construct a probabilistic method:

$$P(\omega) = \begin{cases} 1 & ,\omega <= \omega_o \\ 0.90964e^{\frac{\omega}{2.9661}} + 0.00792 & ,\omega > \omega_o \end{cases} \tag{4.2}$$

where ω_o is the eccentricity value set to represent the size of the fovea of the human eye in degrees.

We visualise in Figure 4.2, this visual probability function. The function showcases how the lessening visual acuity of the human eye, results in spawning a significantly smaller number of rays as we move away from the fovea to the periphery.

When ω is less or equal than ω_o the probability is equal to one. Otherwise, a probability is estimated for ω, as seen in Figure 4.2. This model can be used to decide whether or not to spawn a new ray at a certain pixel based on the visual angle that pixel spans from the central fovea (ω is the eccentricity angle of each pixel from central fovea). To calculate ω for all pixels we determine the Euclidean distance (eq. 4.3) between the pixel being fixated (p) and all other pixels (q). Having calculated the Euclidean distance, we can then determine visual angle (eq. 4.4) to find the ω.

$$d(p, q) = \sqrt{(q.x - p.x)^2 + (q.y - p.y)^2} \tag{4.3}$$

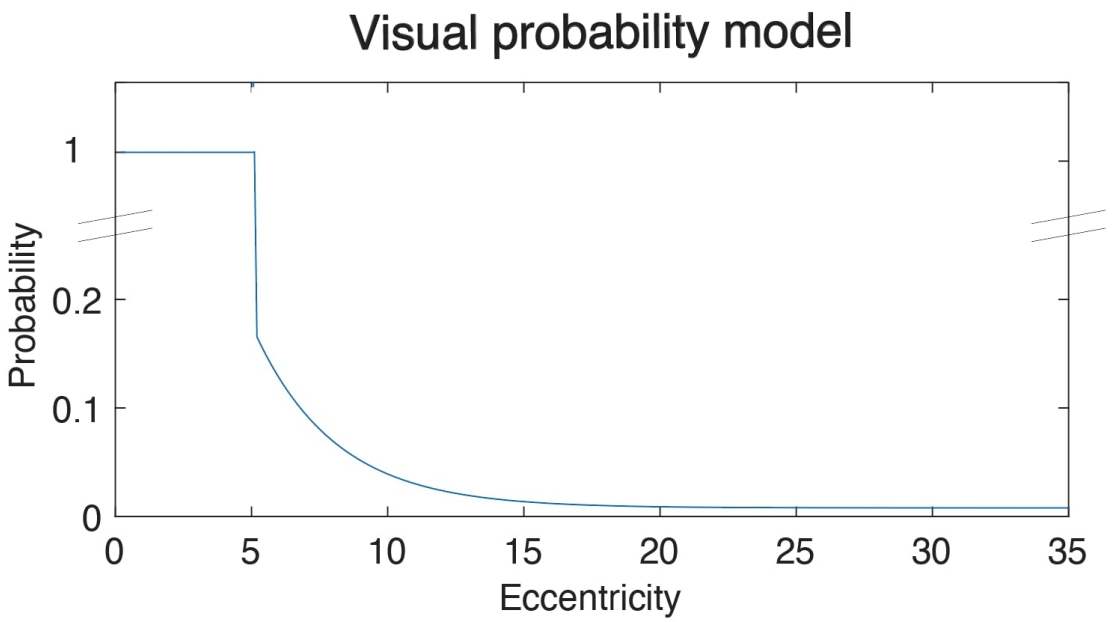


Figure 4.2: Our visual probability model, demonstrated for a foveal angle of 5 degrees. The function showcases how the lessening visual acuity of the human eye, would require a significantly smaller number of traced rays as we move from the fovea to the periphery.

$$V = 2 \arctan\left(\frac{d}{2D_m}\right) \tag{4.4}$$

where D_m is the distance from the display.

Due to hardware limitations prohibiting real-time path-tracing we cannot use our model in real-time. Similarly to [75] & [11], we discretise visual acuity in three distinct zones (Figure 2.16). Inner (foveal), middle (peripheral) and outer (extra-peripheral), with progressively lower resolution. Three zones were deemed to be enough both during our pilot experiments and according to previous work [75, 11]. Having more than three zones approximates the human visual system more closely, but increases the rendering overhead. Based on our discretisation, in the inner region, we path trace at full resolution. In the middle zone we trace a single ray for every 4 pixels. In the outer region we PT a single ray for every 16 pixels. To classify the pixels into regions we use our probability model (eq. 4.2).

To generate the frame buffers for the three zones we start by pre-rendering full-frame high resolution path-traced images (inner) and then down-sampling them to a quarter (middle) and one-sixteenth of the resolution (outer) using nearest neighbor interpolation to drop extra pixels. We then use Lanczos re-sampling to upscale the middle and outer images back to full-frame buffers enabling easy blending between them. We found that Lanczos filtering produces high quality outputs for its com-

plexity. Lanczos re-sampling is also very fast in modern GPUs using a compute shader.

At runtime, centered around the pixel that the eyes are fixating as detected by an eye tracker, we dynamically blend pixels from all three buffers based on our probability model (eq. 4.2) that describes the acuity fall-off in the HVS. The central/inner region samples from the high resolution frame buffer, the middle from the medium resolution buffer and the outer from the low resolution buffer. To determine the boundaries of the zones we sample from the visual probability model, rather than, for example, doubling the foveal region size to obtain the middle zone size as in [75]. We linearly interpolate a short overlapping boundary to avoid sharp transitions between zones.

4.3 Perceptual Study

We conduct a series of experiments to evaluate the potential of foveated PT (FPT). In all experiments the highest visual quality setting (reference) is a full-resolution frame buffer where every pixel has been estimated using path tracing with 128 samples and 32 bounces. Because FPT produces a lower quality image compared to full-blown PT we are interested in determining conservative visual angle thresholds for the three foveation zones, where the drop in visual quality is *undetectable*. The thresholds we estimate for zoned FPT could then be employed in an actual FPT, once such a PT can actually run in real time.

Hypothesis. We hypothesize that there will be a single, average threshold, where the reduced ray-count buffers will be indistinguishable from non-foveated rendering.

Scene. For all experiments we use the Sponza scene [91], as seen in Figure 2.11, an iconic archaeological monument. To study the effect of FPT on complex reflection and refraction, we embedded in the virtual environment three Stanford dragons [92] which are made of glass of different colours.

Basing our approach on [75], we conducted three experiments: a pair test, a ramp test, and a slider test. Each subject participated in all three tests. In all experiments, the dependent variable is the size of the foveal region bordering with the middle region. The border of the middle zone with the outer zone is set based on our probability model. We use eye tracking to dynamically re-construct simulated images of the FPT according to gaze as explained in section 4.2.1. This enabled us to accurately measure conservative thresholds without relying on instructing our subjects to fixate on specific areas of the display; they were rather allowed to look anywhere on the screen.

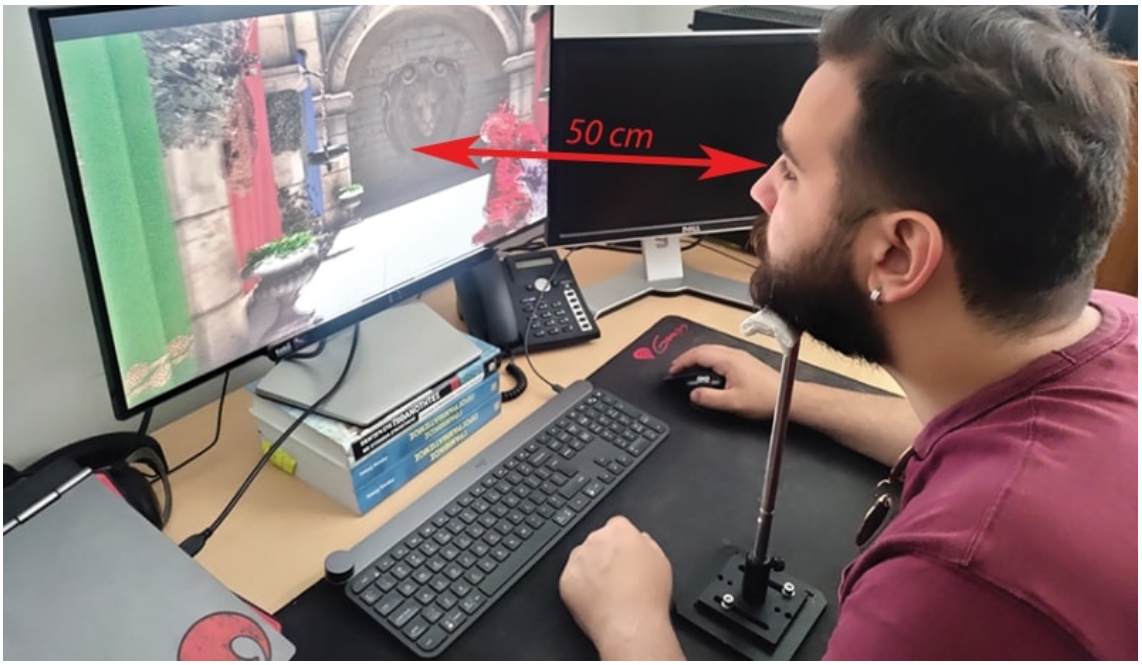


Figure 4.3: The experimental setup and a participant, set at a fixed distance from the monitor.

4.3.1 Experiments: Hardware Setup

We use the 27" DELL S2719H monitor, with a resolution of 1920 x 1080 and a refresh rate of 60 Hz. We employed the Tobii eye tracker 4c to track user gaze. The eye tracker has a sampling rate of 90Hz, however, our latency bottleneck was the limited refresh rate of the monitor (>16.6ms). The computer running the FPT emulator was a Lenovo Legion y740 Laptop with a CPU i7-9750H, 16 GB RAM, and a single Nvidia GPU RTX-2060 with 6GB VRAM.

4.3.2 Pilot experiment: Ray Bounces

During the design phase of our study, we ran pilot experiments serving as a sensitivity analysis for the independent variables of the main experiments. One such experiment, regarding the effect of ray bounces, eliminated any ray bounce modification for our main experiments. We run a pair-test comparing non-foveated rendering to foveated images with reduced secondary ray bounces in the middle and peripheral zones. These secondary bounces generate soft shadows, reflections, refraction, diffuse inter-reflections and more. Reducing the secondary bounces to values low enough to measurably increase performance, generated strong motion signals in the periphery as transparency effects appeared or disappeared when objects transitioned between zones, and shadows & light leaks were flickering among other artifacts (see Fig. 4.1). We decided to exclude any variation in ray bounces for the main experiments, as such variation was perceived by all pilot subjects, unanimously.

4.3.3 Participants

20 subjects participated (8 female, average age 25.45, SD 2.11), only with normal or corrected-to-nornal vision. Participants were placed at 50cm from the screen as seen in Figure 4.3. Each participant performed the standard calibration procedure for the Tobii eye tracker 4c. Following calibration, the experiment sequence was initiated. The average time it took for each participant to finish the three experiments and the calibration process, was around 25.2 minutes.

4.3.4 Experiment 1: Pair

During the pair test, participants were presented with pairs of dynamic (moving) sequences of identical camera trajectories separated by a short (0.5s) black interval. At random, one member of the pair was the non-foveated path-traced rendering and the other part used foveated path-traced rendering at foveal eccentricity levels spanning from 5° to 30°, sampled using 5 discrete steps. Corresponding middle peripheral eccentricity levels spanned 57.5° to 82.5°. Pairs of all levels were presented three times with the foveated rendering first and three times with the non-foveated rendering first, at random. After showing each pair to participants, they were asked to report whether the first or the second rendering was better, or the two were of the same quality. The experiment was designed to investigate the foveation angles where the quality level was deemed similar to non-foveated rendering.

Procedure & Data recording:

For each pair of images, we record the answers provided. To collect the data, we use a user interface that pops up following the presentation of the image pair. The user is then asked to report with an on-screen cursor which sequence was better or if they appeared to be the same.

4.3.5 Experiment 2: Ramp

During the ramp test, each participant was presented with a set of dynamic sequences of identical camera trajectories. An increasing ramp started with a non-foveated sequence and then the foveal and peripheral zones ramped down from foveal eccentricities of 30° down to 5°. A decreasing ramp started with a foveated sequence at 5° and then the foveal and peripheral zones ramped up from a foveal eccentricity of 5°, up to 30°, to non-foveated. Participants were then asked whether the quality had increased, decreased, or remained the same across each sequence step. Each ramp was sampled using 6 discrete steps. Each sequence was 7 seconds long and separated by a short interval of black, aiming to find the lowest foveal eccentricity

perceived to be indistinguishable to the non-foveated reference.

Procedure & Data recording:

While the foveation eccentricity level ramps up/down, we record the subject's response in each change of eccentricity level. We use a similar user interface to the previous experiment to collect user data. The user interface pops up after each sequence ends, asking the participant if the quality has increased, decreased, or remained the same.

4.3.6 Experiment 3: Slider

The slider test allows participants to change the foveal and peripheral eccentricity levels. They are first presented with a non-foveated animation as a reference. Starting at a low level of foveation eccentricity (foveal = 5° and periphery = 57.5°), they can increase the level, see the non-foveated reference again or decrease the level, reporting a quality level equivalent to the non-foveated reference.

Procedure & Data recording:

Users first received instructions that they could change the foveation eccentricity using the keyboard. Having determined the quality level where the foveated sequence quality is indistinguishable from the non-foveated reference, users could save their preference, again using the keyboard.

4.4 Results and Discussion

4.4.1 Visual Fidelity

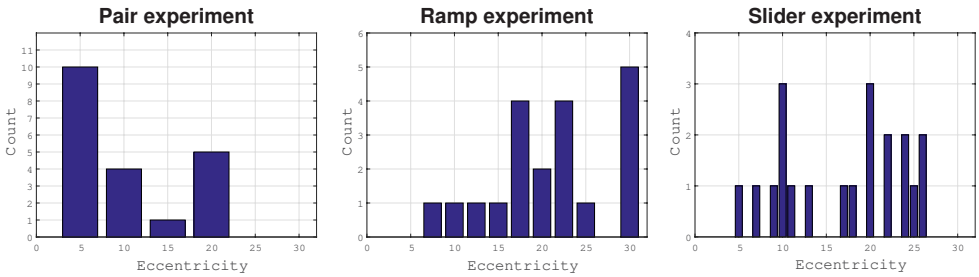


Figure 4.4: The histograms of the raw data for each experiment, as a function of foveal eccentricity.

Raw data recorded for all subjects and experiments are shown in Figure 4.4. In the pair experiment, we have identified the lowest average foveation level for each subject where users reported that FPT looked better or similar to the non-foveated

Foveal Zone (deg)	Middle Zone (deg)					
	50	**55**	**60**	**65**	**70**	**75**
5	4.98x	4.56x	4.19x	3.95x	3.91x	3.91x
10	4.6x	4.23x	3.91x	3.7x	3.67x	3.67x
15	4.07x	3.78x	3.52x	3.35x	3.32x	3.32x
20	3.49x	3.28x	3.08x	2.95x	2.93x	2.93x
25	2.95x	2.79x	2.65x	2.55x	2.53x	2.53x
30	2.47x	2.36x	2.25x	2.18x	2.17x	2.17x

Table 4.1: Performance boost expected from ray reduction. Even with conservative thresholds of 15° foveal and 65° middle, a performance boost of over 3x is expected.

reference. The foveation eccentricity threshold for the pair experiment had a mean of 10.25° (SD 6.38°). In the ramp experiment, we identified the lowest average foveation level, when the users incorrectly report that the quality has changed based on the ramp direction or that the quality was the same across the entire ramp. For the ramp test, the foveation level had a mean of 21° (SD 6.85°). In the slider experiment, we estimate an average preferred value. For the slider experiment, the foveation level had a mean of 16.95° (SD 6.9166°).

We observed significantly different foveation thresholds between the three experiments. For the pair test, we obtain the lowest mean threshold (foveal $\approx 10°$) where the foveated sequence appears to be of equivalent quality to the non-foveated sequence. For the slider test, we obtain a medium mean threshold (foveal $\approx 15°$). Finally, for the ramp test, we obtain a high mean threshold (foveal $\approx 20°$). This was a surprise. Contrary to our hypothesis of a single foveation threshold, we hypothesize that the varying thresholds estimated, arose for a few potential reasons. First, the methodologies of the three experiments (pair, ramp, slider) may have affected the subjects' sensitivity to peripheral blur, as the way that comparisons were made between images were inherently different, i.e., A-B testing vs sequential presentation vs custom adjustment of eccentricities. In addition, we hypothesize that operating on a 60 Hz monitor led us estimate very conservative eccentricity thresholds. If our subjects performed quite rapid saccades, the alterations due to foveated rendering were visible at times; we hypothesize that subjects selected a much higher angle threshold than they would have otherwise. Eccentricity thresholds were shown to be based on the user's ability to detect sudden visual changes in their periphery rather than a specific value.

4.4.2 Computational Complexity Gains

Our method enables an extreme reduction of spawned rays for each frame. We can calculate the reduction of primary rays, and thus performance gains. However,

the reduction of secondary rays is unpredictable. The amount of such rays spawned for shadows, reflections, etc., depends also on scene geometry and the current viewport. We can calculate the reduction of primary rays based on the *Gauss circle problem* [93], i.e., determine how many pixels (lattice points) there exist in the maximum resolution foveal circle, the reduced resolution middle zone, and the further reduced resolution outer periphery (Figure 2.16). The two boundaries of the three zones parametrise the problem (green circles in Figure 2.16).

For our calculations we assume a 27" full-HD monitor at a distance of half a meter. Without foveation, 1920 * 1080 = 2 073 600 rays are normally required per frame. In Figure 4.5 we can observe, as expected, that the number of total rays fired is a function of both the foveal and middle zone boundaries. The maximum value in this plot corresponds to a foveal region that spans the entire field of view ($\sim 70°$), i.e., non-foveated rendering. For any other combination of foveal and middle zone angles we obtain significant ray count reductions and a corresponding performance boost as shown in Table 4.1. We note that to avoid sharp transitions between the three rendering zones, the circles must overlap a few pixels. This has a negligible effect in the total speedup. The same applies for the cost of Lanczos upscaling that has a negligible cost when running in a compute shader.

In Table 4.1, we note that even with a very large foveal region of 30° and a very large middle region of 75° (practically the middle zone spans the rest of the screen) we can still expect at least a 2x speedup. Based on a more conservative threshold of 15° degrees foveal and 65° middle zone (the rest of the screen would be rendered using the outer zone quality), the expected speed-up is over 3x. Even for exterior scenes with daylight, containing pre-rendered skyboxes, where as much as half of the image would not require multiple inter-reflections, we expect that even though the potential savings would be less, they will still be significant, due to the pruning of primary rays for the rest of the scene.

4.4.3 Chapter Summary

This chapter outlines our pioneering efforts to implement a foveated path tracing (FPT) system aimed at enhancing rendering performance while preserving visual quality. We explain the theoretical foundation of FPT and how it aims to minimize the number of rays fired in accordance with viewer fixation points, based on the human visual system's (HVS) acuity. By emulating this system, we propose a probabilistic model based on eccentricity, informed by previous research on visual acuity.

Our experiments reveal that significant visual artifacts arise when ray bounces are

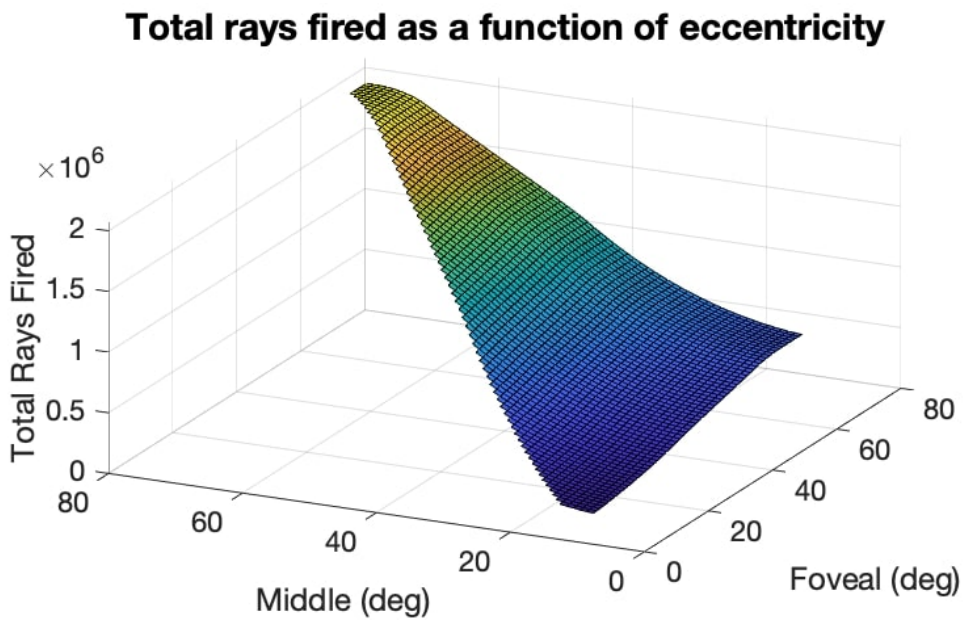


Figure 4.5: Total rays fired as a function of foveal and middle zone eccentricities. The peak of the plot corresponds to a foveal eccentricity spanning the entire field of view, which equates to non-foveated rendering.

reduced, confirming that maintaining adequate interactions is critical for preserving quality. Subsequently, we conducted three experiments to ascertain the visual fidelity thresholds at which FPT remains indistinguishable from standard path tracing. The results showed varying thresholds across experimental methodologies, with an average foveation angle of 10.25° identified in a pair test and a maximum of 21° in a ramp test.

Our findings suggest that integrating foveated rendering could lead to substantial computational efficiencies, with expected performance boosts of over three times when utilizing conservative thresholds for foveal and peripheral zones. This work not only demonstrates the feasibility of a foveated path tracing system but also paves the way for future optimization in real-time rendering applications, outlining the avenues for further development in gaze-aware technologies and real-time path tracing systems.

Chapter 5

An Inverted Pyramid Acceleration Structure Guiding Foveated Sphere Tracing for Implicit Surfaces in VR

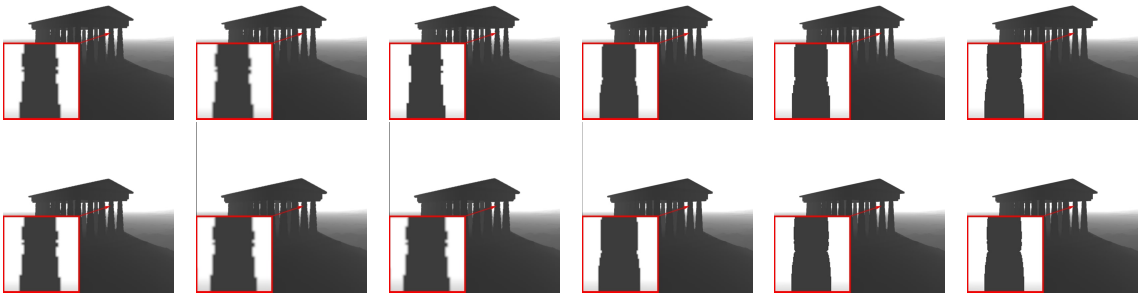
5.1 Overview

In this chapter, we present an innovative method for accelerating the rendering of signed distance fields (SDFs) called Foveated Inverted Pyramid Rendering (FIPR) [1]. Our approach addresses the slow rendering times and performance issues often encountered with traditional sphere tracing, especially in virtual reality (VR) environments. We introduce the concept of an Inverted Image Pyramid (IPR) to facilitate reduced resolution passes, which allow us to approximate ray distances without fully tracing at high resolutions .

By implementing a hierarchical structure of rendering, we progressively refine the distance information from lower resolutions to the target resolution, significantly reducing the total number of steps required for ray marching. To further enhance the quality of rendered images, we integrate foveated rendering, which prioritizes higher detail rendering in areas where the user is fixated, while optimizing peripheral areas based on visual acuity. The chapter culminates in a discussion of our FIPR pipeline—a five-pass rendering process—complete with perceptual studies that confirm the efficacy and performance improvements of our method compared to ground

[1] Polychronakis, A., Koulieris, G. A., & Mania, K. (2023). An Inverted Pyramid Acceleration Structure Guiding Foveated Sphere Tracing for Implicit Surfaces in VR. In Proceedings of the 34th Eurographics Association Symposium on Rendering.

truth rendering.



(a) Level 3 output (b) Level 2 input (c) Level 2 output (d) Level 1 input (e) Level 1 output (f) Ground truth

Figure 5.1: The top row is the inputs and outputs for each level of our inverse pyramid rendering (IPR) without applying the minimum filter. The bottom row shows the inputs and outputs when we apply the minimum filter. The use of the min filter improves the quality of IPR.

5.2 Implementation

SDF rendering employs tracing rays that progressively approach the closest SDF from the ray origin until a threshold is met and then a color is assigned to the rendered pixel (Fig. 2.14). When ray marching SDFs, computation increases proportionally to SDF complexity leading to slow rendering times and, subsequently, to lower frame rates, which is particularly problematic for VR. Edges in the field-of-view are particularly salient and at the same time exceptionally costly, as a high number of steps is required to converge to an edge on screen. In the following implementation, we propose a novel approach to accelerate sphere tracing by approximating the distance that a ray will travel in a 3D scene without fully tracing at the full resolution of the frame buffer, while taking into account salient edges. We do this by first utilizing an inverted image pyramid (IPR), enabling reduced resolution passes to successively refine an approximate distance a ray has to travel and subsequently trace from that approximated position in the target resolution, reducing the total step count.

The highest resolution level of the pyramid is the target resolution frame buffer where all shading computation is performed. To find the final implicit surface that requires shading we progressively reconstruct that final image starting from the low resolution tip of the pyramid, which renders at high frame rates. At each level of the pyramid, each ray's distance that we calculate, corresponds to a group of 4 rays/pixels at the next level. We can thus advance 4 rays closer to a surface, rather than starting from the camera origin. This allows each ray to begin tracing (at

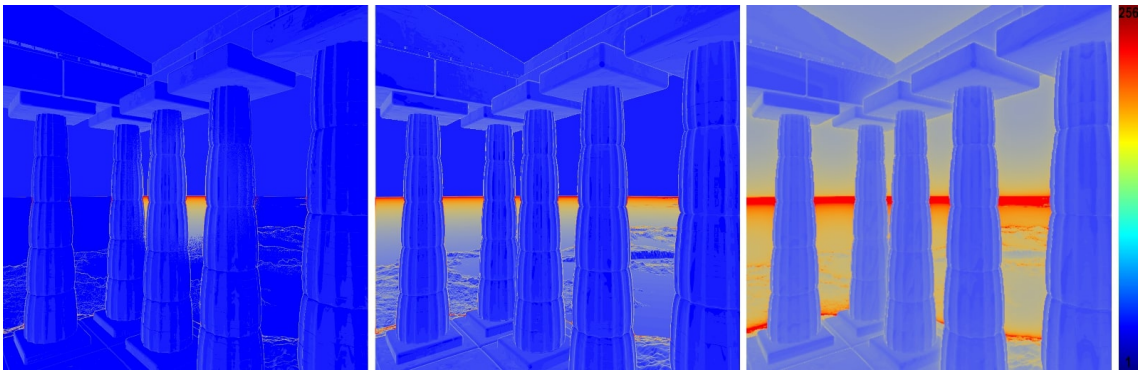


Figure 5.2: *Left to right, total step count FIPR vs IPR vs ground truth. For IPR & FIPR tracing steps are significantly reduced due to the inverted pyramid acceleration structure and foveated rendering respectively. Blue: 1 step, red: 256 steps.*

the subsequent level) from the approximate distance rather than from the camera origin. We repeat this process for subsequent levels of the pyramid, until we reach the top level of the pyramid. As a result the total step count is significantly reduced compared to fully sphere tracing at the target resolution (Fig. 5.2).

However, since each lower resolution approximation does not accurately contain/represent what the higher resolution does, artifacts appear, because pixels may travel beyond the actual surface and might miss a surface that does not appear at a lower resolution altogether. This can result in low quality rendering, as depicted in Fig. 5.1. We eliminate those artefacts using a neighbor *min* filter when propagating ray distances within the pyramid. This filter ensures that rays close to surfaces begin tracing before passing the surface, resulting in correct shapes and a higher quality output, as shown in Fig. 5.1.

Initial testing demonstrated that three levels are sufficient to improve performance, i.e., the bottom level is 1/16th of the target resolution, the middle level is 1/4th of the target resolution. Increasing the number of levels to more than 3, did not increase performance due to an elevated number of rendering calls. IPR demonstrably improves computational efficiency (see Sec. 5.4.2) for sphere tracing without compromising the rendering quality which was evaluated in a small pilot study (see Sec. 5.3.2). In particular, IPR achieves 4.28× performance improvement.

To achieve the high frame rates required for VR applications, we subsequently integrate foveated rendering in IPR. Foveated rendering renders at the highest resolution where people fixate and gradually reduces the resolution towards the periphery of the fixation. Previous foveated rendering approaches for ray/sphere tracing strive to reduce the number of generated rays with the aim of improving performance. IPR can accommodate foveated rendering depending on where a pixel is in the visual field as it can be directly sampled from a different level in the pyramid. We take this one

step further: We want to ascertain that salient edges (which are also particularly costly) are rendered properly. Instead of simply *decreasing* the probability of a pixel to be fully traced as eccentricity increases, which is typical in most foveated methods, we *increase* the probability of pixels in the periphery in areas with prominent edges, ensuring that salient edges in the periphery are properly reconstructed. For such edges, we trace based on the higher resolution level of IPR in the foveal region, where the human eye is most sensitive to detail. We trace rays using the mid-level of IPR for the rest of the frame buffer and then interpolate to the full resolution at a lower refresh rate for those pixels. By doing so, we can maintain high image quality in the foveal region while significantly reducing the computational cost of sphere tracing in the periphery, ultimately allowing us to balance the trade-off between image quality and computational efficiency, leading to more efficient, higher fidelity rendering. We call our full model Foveated Inverted Pyramid Rendering (FIPR).

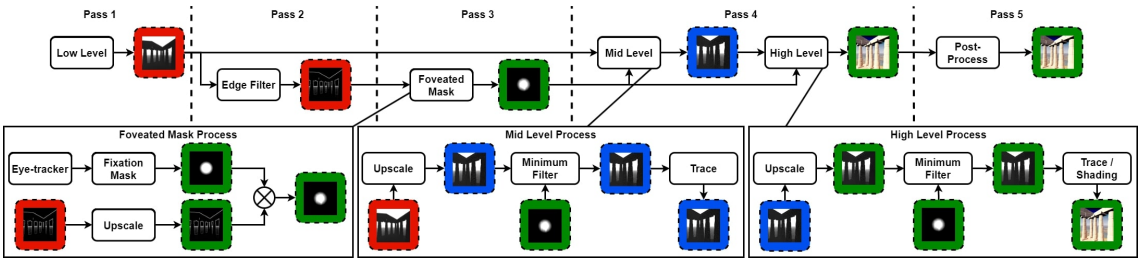


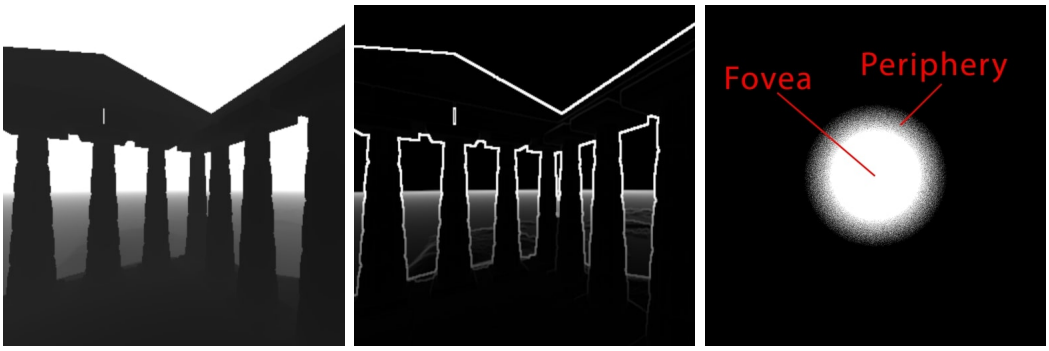
Figure 5.3: *We accelerate rendering of SDFs by accounting for the foveation angle and edges in the periphery in a five-pass pipeline. The red frame indicates buffers at 1/16th of the screen resolution, blue 1/4th of the screen resolution, green denotes full (output) resolution.*

5.2.1 Foveated Inverted Pyramid Rendering (FIPR)

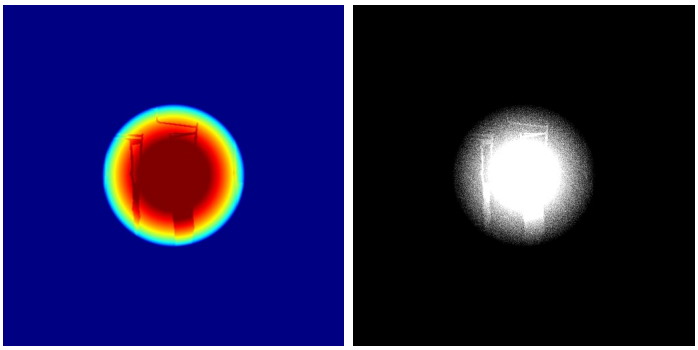
FIPR is a five pass pipeline (Fig. 5.3). Our method was implemented in the Unity 3D Engine using compute shaders and applied as a full-screen quad post-processing effect to the camera. Each pass corresponds to one kernel, implemented using compute shaders. Specifically, the low, mid, and high levels of our inverted pyramid structure, the edge filter, the foveated mask, and the post-processing Gaussian blur effects are all implemented as separate compute shader kernels. The minimum filter is applied to each level (mid / high) of the inverse structure before sphere tracing, thus it is not a separate kernel. We now describe each pass:

Pass 1: Low-res Depth Image

The first sphere tracing pass starts at the base of the inverted pyramid (1/16th of the screen resolution), generating a low resolution depth map (Fig. 5.4a) storing the marched distance from the camera origin to the closest implicit surface. The



(a) Depth map (b) Edge map (c) Visual acuity



(d) Probability (e) Foveated mask

Figure 5.4: *We generate a binary foveated mask used to decide which pixels are re-marched in the final high quality rendering, as a product of a hyperbolic acuity model and an edge detection map over a computationally cheap depth map.*

distances to surfaces are only accurate for this low resolution, and will act as a rough approximation for subsequent tracing in higher levels. This low resolution approximation becomes the input to a probability model which decides which rays will be spawned (depth image forwarded to pass 2) taking into account the scene's rough edge information, but also guides the reconstruction and used for sampling at the target resolution (forwarded to pass 4).

Pass 2: Salient Edge Detection

In the second pass, we apply Sobel edge detection on the low resolution depth map to gather scene edge information, which is computationally cheap as it is applied at $1/16^{th}$ of the target resolution (Fig. 5.4b). Texture pixels can take one of two values, 1 (edge) or 0 (not an edge). We then upscale that edge texture to the target rendering resolution and forward it to pass 3.

Pass 3: The Foveation Mask

In the third pass, we create the foveation mask. Eye tracking data from the VR headset provide the fixation point. We generate a hyperbolic eccentricity-based probability map (Eq. 5.1) corresponding to the fall-off of visual acuity from the

$$p(\omega) = 0.90964e^{\frac{\omega}{2.9661}} + 0.00792 \tag{5.1}$$

5.1: p(ω) is the probability of the pixel, and ω is the eccentricity angle of the pixel.

fixation point to the periphery; measured visual acuity for various human eye eccentricities from [90] as fitted by [50]. This hyperbolic fall-off probability map has three distinct zones: fovea, periphery, and outer periphery as in [13]. In the fovea, probability is set to 1; in the periphery there is a hyperbolic fall-off; in the outer periphery, pixel probability is set to 0.

We subsequently multiply this map with the calculated buffer containing edges from the second pass (Fig. 5.4e). Multiplying the probability map with the edge detected depth map ensures that even in the periphery, salient edges will be traced at full resolution, consistent with the human visual system's high sensitivity to peripheral edges. The resulting texture indicates the final probability assigned for each pixel to be sphere traced in the high level of our IPR, as seen in Fig. 5.4d.

We then construct a binary foveated mask indicating pixels of high importance to be queued for sphere tracing at the highest level only if $\xi_q < p(x, y)$ with $\xi_q \in [0,1]$ being a uniformly distributed random number. This converts the probability into a Boolean True or False value indicating whether a pixel needs to be traced at full resolution, as seen in Fig. 5.4e. Non-foveal regions and peripheral regions without edges are rendered at a lower resolution, significantly accelerating throughput. The foveated mask is forwarded to both passes 4 and 5.

Pass 4: Rendering & Reconstruction

In the fourth pass, we assemble the complete frame using the high fidelity (foveal/edge regions) pixels and the (lower refresh rate) sampled peripheral pixels. The depth texture from pass 1 and the binary foveated mask from pass 3 are inputs for pass 4. We first sphere trace in the mid-level of our acceleration structure where the resolution is now equal to $1/4^{th}$ of the target frame buffer. We first check whether a pixel is of high importance in the binary mask; if it is, we check its neighbours and apply a minimum filter to the depth buffer - this is the distance from which the ray will start tracing. The minimum filter simply replaces the pixel's depth value with the minimum value (i.e., distance/depth) of that pixel and its neighbors (3x3 filter).

Rationale: The minimum filter alleviates a possible artefact of IPR. When rendering at lower resolutions, some surfaces might be completely missed, while they can be visible at the higher target resolution. The minimum filter pushes a ray's tracing origin a bit further back towards the camera origin, ensuring that when the

ray is marched it will not be placed behind a surface and thus miss it altogether. Using a minimum filter, pixels that had traversed almost the same distance as their neighboring pixels in the 3D environment will remain close to the surface based on the approximate distance rather than resetting further back to the camera origin, regardless of the scene configuration. This eliminates artefacts and improves performance due to each pixel repositioning itself based on information gathered from neighboring pixels, compared to using a fixed offset as in [10, 44, 8]. When the minimum filter is not applied, this leads to faster convergence to the surface but a surface could be missed. As the size of the minimum filter increases, quality is further improved, but performance drops.

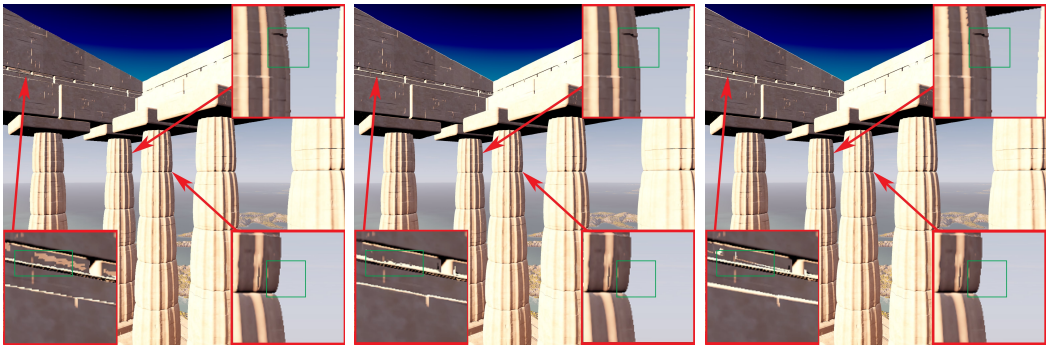
Sphere tracing starts from the generated approximate distance recorded in the depth texture. This reduces the steps that a ray will have to march as compared to a ray being spawned from the camera origin. At the mid-level of the acceleration structure, a new depth texture is produced based on the new approximated distance which is then forwarded to the highest level of the acceleration structure. At the highest level, rendered at the target resolution, after having calculated the final distance to the closest surface for each pixel, we apply shading calculations to high importance pixels. For low importance pixels, we reduce the sampling refresh rate based on which approximate distance is calculated, to increase overall rendering performance, since those pixels mostly go unnoticed [75, 94]. The final texture is a composite of both high-fidelity and low-fidelity pixels, as guided by the foveation mask. This texture is forwarded to pass 5 for post-processing.

Pass 5: Post-Processing

Most foveated rendering pipelines use post-processing effects to hide, reduce, or eliminate visual artefacts in the periphery due to the lower fidelity rendering; usually a Gaussian blur filter [11, 50, 82]. In the fifth (and last) pass, we apply a Gaussian blur guided by the foveated mask. This process smooths any visible transitions between the high-fidelity and low-fidelity areas. The foveated mask from pass 3 and the texture inclusive of shading from pass 4 are inputs to this pass. Based on the foveated mask, we apply a Gaussian blur to the transition zones between low and high fidelity pixels. The final image is a composite of both high-fidelity and low-fidelity pixels, as indicated by the foveation mask. In our study, we investigated if a Gaussian blur filter is helpful when we apply foveated rendering to hide peripheral artefacts.

(a) Greek Temple (b) Column & Lights (c) Primitives

Figure 5.5: *Scenes used in the evaluation, by Inigo Quilez [1], with permission.*



(a) Foveated IPR (FIPR) (b) Non-foveated IPR (c) Ground truth

Figure 5.6: *Quality comparison between foveated inverse pyramid rendering (FIPR), inverse pyramid (IPR), and ground truth rendering. Notice how IPR and ground truth treat edges identically. FIPR distorts insignificant edges in the periphery (eye fixation at the center of the image).*

5.3 Perceptual Study

We conducted two two-alternative forced choice (2AFC) studies. The first (pilot) study aims to investigate whether the perceived quality of our non-foveated acceleration structure (IPR) is similar to ground truth, i.e., it does not introduce any artefacts. The second study aims to parametrize our foveated method by establishing parameters for which the drop in quality in the periphery of vision remains unnoticeable.

We used simplified versions of high-quality scenes by Inigo Quilez (with permission [1], Fig. 5.5), to maintain acceptable frame rates, i.e., by removing soft shadows and ambient occlusion. The first scene named "Primitives", represents an open environment that renders primitive shapes on a plane with a checkerboard pattern for the ground. The second scene named "Column & Lights", features a more complex environment in a closed setting. This scene includes a repeating pattern of columns and five point-lights that move around the columns. Finally, the third scene named

"A Greek Temple", represents an open cultural heritage environment including a procedurally generated cliff terrain and a temple located on a cliff. These scenes depict complex indoor/outdoor environments without explicit geometry and are described using SDFs.

5.3.1 Experiments: Hardware Setup

Figure 5.7: Head mount display - HTC Vive Pro eye.

We used an HTC Vive Pro Eye Head-mounted Display (HMD) (Figure 5.7), with a resolution of 1440 × 1600 per eye, i.e., combined resolution 2880 × 1600 and a refresh rate of 90Hz. The embedded eye tracker on our HMD has a sampling rate of 120Hz and an accuracy of 0.5°0.5 - 1.1°in the central field of view. The computer used had an Intel i9-12900F CPU, 32GB RAM, and a single Nvidia GPU RTX-3070ti with 8GB RAM. Eye tracking is not integrated into most desktop setups as, recently, in VR headsets. Therefore, foveated rendering in VR is meaningful also due to the much higher rendering cost in HMDs compared to desktop displays and the readily available eye-tracking. Desktop setups can of course employ our method using an external desktop eye tracker.

5.3.2 First (Pilot) Study: Inverse Pyramid Rendering (IPR)

The first pilot study compared the quality of the Inverse Pyramid Rendering with ground truth rendering, where all pixels are traced from the camera origin in a single pass. In this study, we wanted to establish that the acceleration structure does not introduce any visible artefacts. At the same time, we benchmarked our structure to demonstrate that it can perform faster than standard sphere tracing. Pilot testing indicated that the optimal value for the filter size is 3x3. If we increase

the filter size, the performance drops without any noticeable quality improvement. If the filter size is small the quality is worse than the ground truth as surfaces might be missed.

We conducted a 2AFC pair experiment with participants wearing the HTC Vive Pro Eye HMD and using an Xbox controller to compare the quality for a series of trials. Each sequence pair consisted of the ground truth rendering and our non-foveated IPR, in random order. Participants were asked "Was the quality the same in both sequences?". For this pilot study we recruited 5 participants from our campus, with an average age of 28.5y (SD 1.5y) and a normal or corrected-to-normal vision. For each scene, there were 6 trials. It took approximately 8 minutes for each participant to complete the experiment. Users uniformly found the quality produced by our non-foveated proposed IPR to be the identical to the ground truth. This established that the IPR structure *does not* introduce artefacts and we could proceed with introducing and parametrising foveated rendering in the main study.

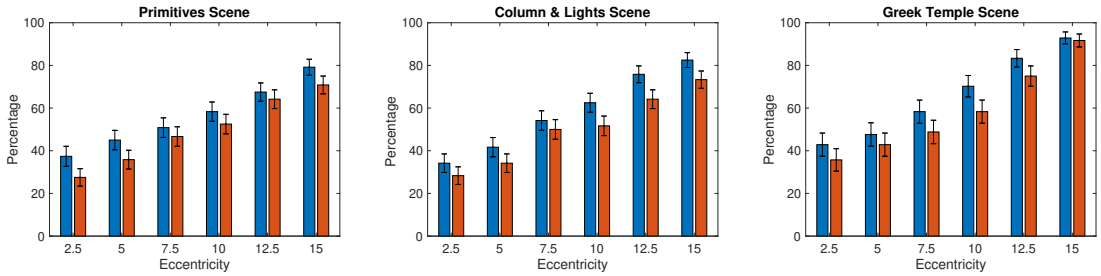


Figure 5.8: *User study results. The blue bars correspond to the percentage of trials indicating that foveated inverse pyramid rendering (FIPR) without Gaussian blur is identical to ground truth, and the orange bar corresponds to the percentage of trials indicating that foveated inverse pyramid rendering (FIPR) with Gaussian blur is identical to ground truth.*

5.3.3 Second (Main) Study: Foveated Inverse Pyramid Rendering (FIPR)

The second study aimed to evaluate whether the drop of quality in the periphery of vision is noticeable, and determine the optimal parametrisation values (dependent variables), i.e., the eccentricity angle of the fovea and the presence or not of Gaussian blur, for our foveated inverse pyramid rendering (FIPR), so that the quality reduction remains imperceptible. Please see the limitations sections as to why additional variables (refresh rate, super-sampling) were not included in the study. We employed the same two-alternative forced choice (2AFC) experimental methodology as in the pilot study. Fovea eccentricity varied from 2.5 to 15 degrees in steps of 2.5. Each pair to be compared consisted of the ground truth rendering and the FIPR

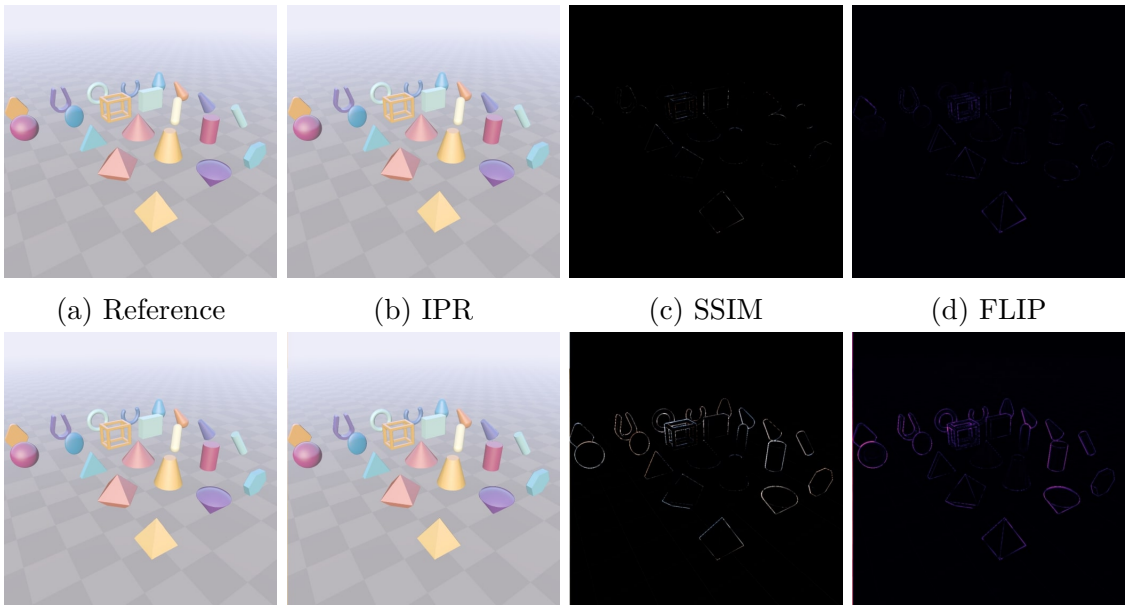
method. The study aimed to determine conservative thresholds for the foveation parameters of our method (eccentricity, peripheral Gaussian blur).

We evaluated the performance gains when rendering in VR for the three scenes. We calculated the time to render a frame with our method. We compared it with the time it takes to render the ground truth (full sphere tracing). The target resolution expected by the HTC Vive Pro Eye due to pinchusion pre-distortion correction, even though the specifications report a resolution of 1440 $\times$ 1600 (a full resolution of 2880 $\times$ 1600) is equal to 2468 $\times$ 2740 per eye (full resolution of 4936 $\times$ 2740). We could not perform any direct performance comparison with previous foveated rendering techniques as they did not involve sphere tracing of SDFs sequences, making the comparison invalid, since we are not using geometry as in [87, 6, 84, 13]. Previous work also took advantage of hardware acceleration for mesh rendering (e.g., geometry shaders and ray tracing acceleration structures) to increase performance. Readily available hardware acceleration structures are not available for sphere tracing. Implementing an apples-to-apples comparison was therefore impossible. Instead, we thoroughly gauged the performance of our technique both in terms of visual fidelity and computational performance against ground truth rendering, measuring substantial performance gains.

Participants A total of 20 participants (6 female) with normal or corrected-to-normal vision took part in the study, with an average age of 28.5y (SD 2.5y). 72 trials were presented for each scene (totaling 216 trials) and the average time it took for each participant to complete each scene session was approximately 15 minutes (45 minutes in total). A 10-minute resting session between sessions was allocated to reduce fatigue. Before the experiment started, each user performed an eye tracker calibration.

5.3.4 Objective SSIM and FLIP Metrics

We utilized the SSIM [95] and FLIP [96] metrics to assess the image quality of our IPR and FIPR methods compared to the ground truth obtained through full sphere tracing. Fig. 5.9 presents a visual representation of the local SSIM values and an image with the differences produced by FLIP and table 5.1 shows the global SSIM value and the mean error of the FLIP for each scene.



(a) Reference (b) IPR (c) SSIM (d) FLIP

(e) Reference (f) FIPR (g) SSIM (h) FLIP

Figure 5.9: *Visualization of the local SSIM and FLIP difference/error map for the primitives scene where the top row shows the results for ground truth vs IPR and the bottom row for ground truth vs FIPR. Practically no difference for ground truth vs IPR. Zooming-in can reveal details for ground truth vs FIPR; the maps were not processed/optimised otherwise to not introduce artificial* error *pixels.*

5.4 Results and Discussion

5.4.1 Visual Fidelity

In Fig. 5.6, we present a visual comparison of our inverse pyramid rendering (IPR) with and without foveated rendering as compared to the ground truth.

We present the results of the user study in Fig. 5.8. For eccentricity values over 7.5° users could not identify any difference in quality between our FIPR method and the ground truth. Users tended to prefer the ground truth rendering over our method when the eccentricity was lower, and that preference towards the ground truth surprisingly increased when blur was applied in the periphery, i.e., blur made the manipulation more apparent.

There were slight variations in the results across the three scenes (Fig. 5.5). Users commented that in the "Primitives" scenes which featured primitive shapes described by SDFs, their attention was primarily focused on the sharp shape edges. Consequently, they were able to identify minor artefacts when low eccentricities were used and concluded that the drop in quality was significant compared to the ground truth.

Similarly, in the "Column & Lights" scene, users reported that when their gaze

	G. Temple	Col. & Lig.	Primit.
GT vs IPR SSIM	0.99579	0.99192	0.99889
GT vs IPR FLIP	0.011108	0.046423	0.050560
GT vs FIPR SSIM	0.9777	0.94939	0.9902
GT vs FIPR FLIP	0.030102	0.077929	0.109450

Table 5.1: Evaluating the visual fidelity of IPR and FIPR with respect to ground truth. The global SSIM value indicates high structural similarity and the mean error measured by the FLIP metric approaches zero for all scenes in IPR, indicating a very high similarity to ground truth. In FIPR, global SSIM decreases and FLIP mean error increases for all scenes, as expected, due to the foveated rendering's quality drop in the periphery. These results were obtained with an eccentricity foveal setting of 7.5°.

was focused on a column, they were able to detect a decrease in quality in the surrounding columns for low eccentricities. This can be attributed to the fact that both scenes have geometries with closely positioned edges.

In contrast, in the "Greek temple" scene almost all users found the quality to be the same as the ground truth for higher eccentricity values to a percentage above 80% compared to the other scenes. This was possibly due to the scene featuring fewer, larger shapes producing less noticeable quality transitions. It is evident from these results that the users found the quality to be the same when the eccentricity is equal or higher than 7.5°and that blur was not required, as blur potentially signaled that quality had dropped.

We hypothesize that this might occur due to the increased complexity of the "Greek Temple" compared to simpler scenes such as "Primitives" which drove users to fixate on a single point of the scene. Users were more sensitive to details such as the primitive's shape and observed drops in quality that may have occurred by focusing on the primitives' shapes. We hypothesize that with faster rendering and a more responsive eye tracker in a VR headset, the drop in quality will not be noticeable due to the faster update of gaze fixation on the screen. Our findings underscore the need for further research to explore these variations and determine the optimal approach to use for various scenes.

5.4.2 Performance Evaluation

In Fig. 5.10, we present the measured performance improvements for IPR without foveated rendering. The speed-up varies from 1.22× to 4.28× as the samples per pixel increase to enable super-sample anti-aliasing (SSAA). In Fig. 5.11, we present the render time for each scene for various eccentricity levels, samples per pixel, and refresh rates for the low importance pixels for IPR when foveated rendering is

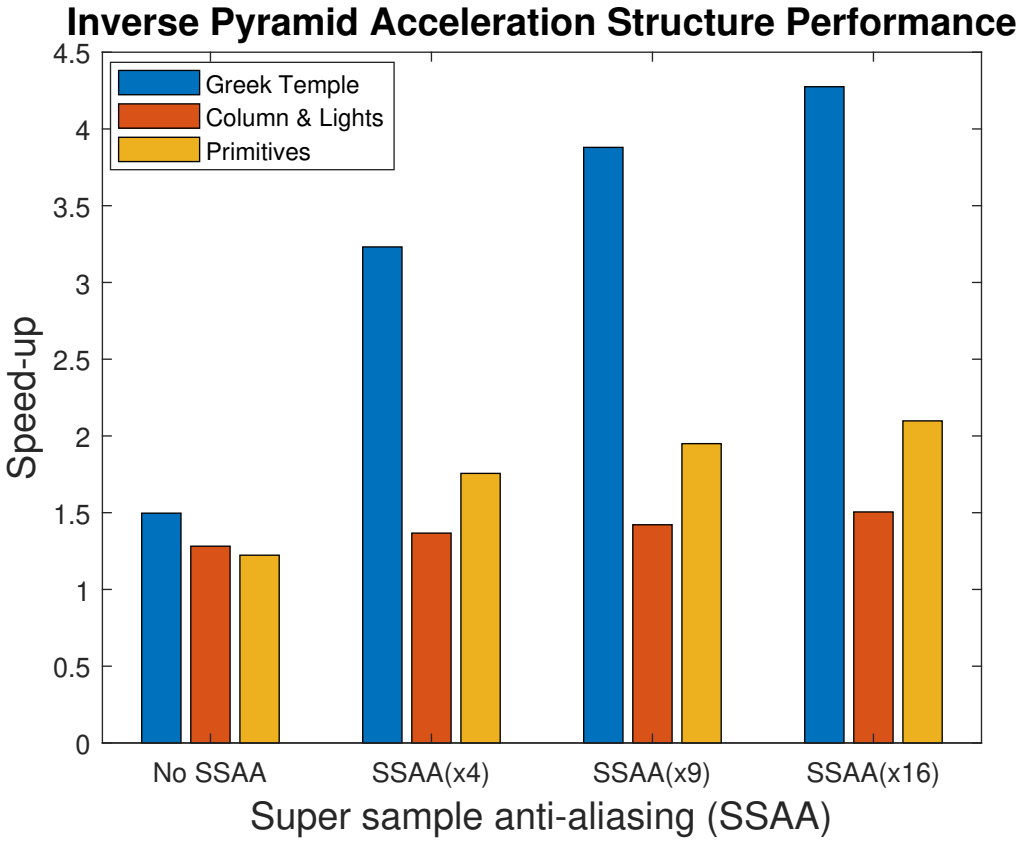


Figure 5.10: *Speed-up of inverse pyramid rendering (IPR) without foveation, with super-sampling varying from 1 to 16.*

applied.

Our results showcase a significant improvement in rendering performance with a speed-up that varies from 1.22× to 20.04× as the eccentricity level drops from 15°to 2.5°, the samples per pixel increase from 1 to 16 and the refresh rate of the low significance pixels is reduced to 1/2 or 1/4 of the refresh rate of the pixels in the fovea when using IPR.

FIPR showed significant improvements in performance for the highly complex 'Greek Temple' scene, even when the samples per pixel were set to their lowest value, and the refresh rate for the periphery was lower. This was because of the higher complexity of the scene compared to the other two scenes, and our foveated inverse pyramid rendering achieved a speedup ranging from 1.23× to 12.65×. In the 'Column & Lights' scene, because of the repetitive nature of the geometry and high saliency of edges, our method achieved a speedup ranging from 1.08× to 10.01× when the samples per pixel were more than 1 or the refresh rate in the periphery was different from the fovea. Similarly, in the 'Primitives' scene, our method achieved a speedup in performance that ranged from 1.37× to 20.04×, except when no super-sample anti-aliasing was applied, and the refresh rate at the periphery was the same

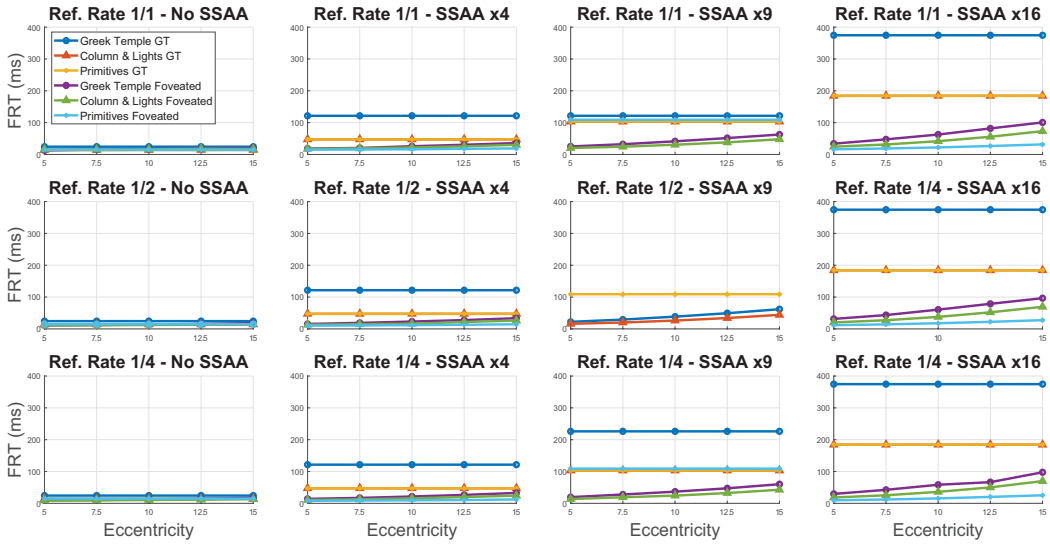


Figure 5.11: *Rendering time for stereo rendering (two viewports) for each scene, foveated and non-foveated (ground truth) in milliseconds. From left to right, supersampling varies from 1 to 16. From top to bottom, the refresh rate outside the fovea varies from 1/1 to 1/4 the rate of the fovea. Our method enables interactive rendering for scenes that would otherwise be impossible to run in VR.*

as the fovea (Fig. 5.10).

Based on the lowest acceptable eccentricity value (Eccentricity = 7.5°) where quality is perceived to be identical to the ground truth, we achieve a speedup ranging from 1.1× to 10.29× for the "Column & Lights" scene, 1.1× to 16.03× for the "Primitives" scene and 1.58× x to 8.83× for the "Greek Temple" scene as the samples per pixel increased from 1 to 16 and the refresh rate for periphery was reduced from 1 to 1/4 of the refresh rate of the fovea area.

Our inverse pyramid rendering is demonstrably highly effective in rendering complex scenes that render at very low frame rates or may not even run at all in VR. Our proposed inverse pyramid rendering demonstrated better performance when the 3D environments comprised of open areas such as mountains or terrains compared to enclosed areas such as caves or dungeons. This is probably due to the lower frequency content in open spaces containing less sharp edges in the open environments. Said another way, in open environments, more rays are not passing close to surfaces compared to enclosed spaces which results in fewer steps and a faster approximation of surface distance for most rays.

Our inverse pyramid rendering even without foveated rendering can significantly improve performance both for one sample-per-pixel or multiple samples-per-pixel. In both cases significantly fewer rays are spawned and marched due to the pyramid and many rays are terminated much faster due to sampling from different levels of the pyramid.

Overall, our inverse pyramid rendering with or without foveation has demonstrated significant improvements in rendering complex VR scenes, achieving faster performance and more accurate approximations of surface distance, even in complex environments.

5.5 Chapter Summary

This chapter discusses the implementation of Foveated Inverted Pyramid Rendering (FIPR) to improve the efficiency of rendering signed distance fields (SDFs). The proposed method begins with the introduction of an Inverted Image Pyramid (IPR), which allows for lower resolution approximations to estimate ray distances more effectively. By utilizing a hierarchical structure, we demonstrate that rays can begin their tracing from an approximate distance determined at lower resolutions instead of from the camera origin. This technique drastically decreases the number of rendering steps required, thus improving the overall frame rate, especially critical in VR environments.

We emphasize the challenges associated with traditional ray marching, notably the computational costs tied to tracing rays near salient edges. To tackle potential artifacts arising from lower resolution approximations, we apply a neighbor minimum filter, enhancing the accuracy of renderings. The integration of foveated rendering within the IPR framework allows us to render salient edges in peripheral vision more accurately, balancing quality and computational load.

Performance evaluations show that FIPR achieves up to 4.28× speed improvement over conventional sphere tracing methods without foveated rendering and up to 20.04× when foveated rendering is applied. Our perceptual studies confirm that with eccentricity thresholds set above 7.5°, users found no noticeable difference in quality between FIPR and traditional rendering, therefore establishing FIPR as a viable solution for high-fidelity, real-time rendering in VR applications. Overall, this chapter establishes the groundwork for further optimization of SDF rendering, encouraging continued research into foveation-aware techniques.

Chapter 6

Skipping Spheres: SDF Scaling and Early Ray Termination for Fast Sphere Tracing

6.1 Overview

In this chapter, we introduce a novel method for enhancing the efficiency of ray tracing by improving step reduction strategies in the rendering of signed distance fields (SDFs) [1]. Our approach builds on existing techniques that lower ray steps and generate low-resolution renders, but seeks to minimize the artifacts often associated with such methods. By employing a single low-resolution buffer and implement-

[1]Polychronakis, A., Koulieris, G. A., & Mania, K. (2024). Skipping Spheres: SDF Scaling & Early Ray Termination for Fast Sphere Tracing. In Proceedings of the 42th Eurographics Association Conference on Computer Graphics & Visualize Computting.

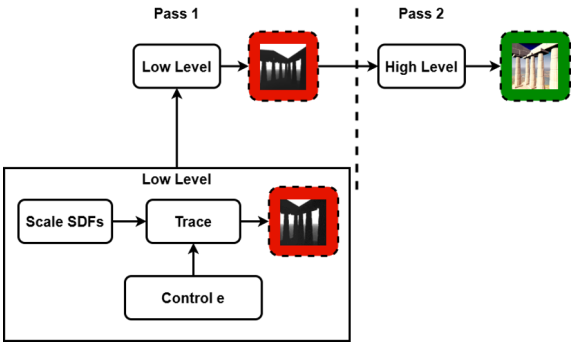
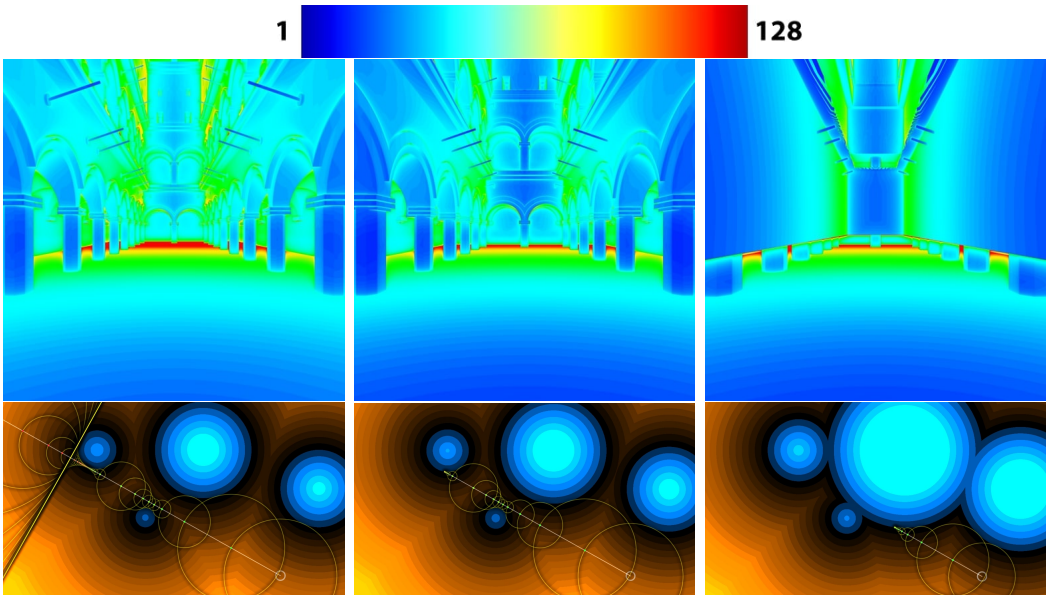


Figure 6.1: *We accelerate rendering of SDFs by enlarging the footprint of the SDFs and by reducing accuracy of sphere tracing by increase the minimum accepted distance between ray and an SDF in a two-pass pipeline. The red frame indicates buffers at 1/16th of the screen resolution and green denotes full (output) resolution.*

ing specific strategies for SDF scaling and ray convergence detection, our system effectively reduces computation without sacrificing visual quality.

We perform the entire rendering process in two distinct passes (Figure 6.1: a pre-processing step that optimizes the low-resolution buffer and a main pass that executes high-resolution sphere tracing. This two-step process minimizes the number of rays required to properly render small features, thereby ensuring that high-cost surface edges are accurately captured. The chapter concludes with a comprehensive evaluation of the method's performance and visual fidelity, comparing it against state-of-the-art techniques to demonstrate its effectiveness.

6.2 Implementation

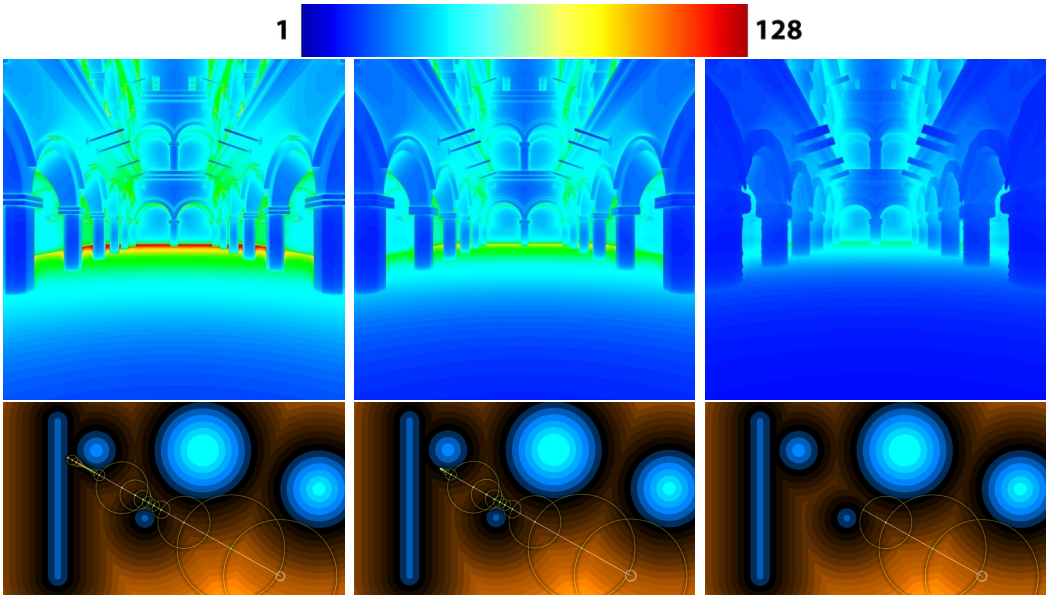


(a) no scaling applied (b) Low scaling (c) High scaling

Figure 6.2: SDF scaling steps (1-128, visualised) for different parameters in the Sponza atrium (top) and a schematic of the phenomenon (bottom). Left: No scaling applied in the low resolution buffer and small surfaces are missed. Middle: SDF size was increased leading to small surfaces been correctly "hit". Right: Extreme SDF scaling leads to artifacts.

Our method draws inspiration from prior techniques aimed at reducing ray steps by adjusting step size and generating lower resolution renders [51]. Yet, the speed gained from low-resolution buffers can lead to noticeable artifacts, particularly in small features, necessitating filtering. Our approach minimizes artifacts while significantly cutting down steps compared to previous methods. This is achieved by using a single low-resolution buffer and a novel tracing optimisation instead of a filtered pyramid. This provides a *head start* origin for bundles of rays when rendering at the full resolution, by positioning rays closer to the surfaces using the information stored

in this low resolution buffer, significantly improving performance. First, we perform SDF scaling in the low-resolution buffer, effectively enlarging the footprint of the implicit surfaces when rendered in low resolution, ensuring visibility of all SDFs; Secondly, we identify when a ray converges to high-cost surface edges and can terminate sphere tracing earlier than usual, further reducing step count significantly. Our method works in two passes described below.



(a) Low ϵ (b) Medium ϵ (c) High ϵ

Figure 6.3: Step visualisation for different ϵ values in the Sponza atrium (top) and a schematic of the phenomenon (bottom). Step count is reduced from left to right as the ϵ value is increasing.

6.2.1 Pre-processing: Low Resolution

In the pre-prossessing pass, we render primary and shadow rays in a low resolution buffer (for a 4K output buffer we reduce the low resolution buffer to a quarter of the resolution in each axis, i.e., from 3840x2160px to 960x540px) while applying scaling operations in the implicit surfaces and use a larger *epsilon*, i.e., terminate the ray further back from the surface, storing a smaller distance in the low-resolution buffer (Figure 6.3). We scale SDFs uniformly because we have no prior knowledge of the volume surface and we increase the volume of the SDFs using the scale operator.

Before scaling the SDFs, we establish whether the camera distance is positive or negative (meaning it is in front or behind an implicit surface, or inside or outside an implicit volume if it is a closed shape). If the distance is positive we scale up, i.e., inflate the volume surface of the SDF. If the distance is negative, we scale down the volume surface, i.e., deflate the SDF. This estimation has negligible computational cost.

This effectively expands the reach of implicit surfaces during rendering in low resolution, guaranteeing the visibility of even the smallest SDFs/objects that might have be missed otherwise. We store the traversed distance in the low-resolution buffer providing a head start origin for bundles of rays at the full resolution (for our target resolution each pixel will be reused by groups of 4x4 pixels (16) in the full resolution) by positioning rays closer to the surfaces using the information stored in the low-resolution buffer. This significantly enhances performance by reducing the distance rays need to travel before reaching surfaces.

For objects that have holes (generated using the subtraction operation) instead of increasing the volume size of the SDF that generates the hole, we *decrease* it , to improve the approximation. This works only when the camera is in front of the implicit surfaces: if the camera is behind the implicit surfaces, we scale up the SDF.

This approach once again, forces the SDFs to cover *more* pixels in the lower resolution to get a better estimation of the distance to be used in the second stage (Figure 6.2). When a larger surface overlaps with a smaller one, this provides a relaxed estimation of the distance between the ray and the surface (Figure 6.2c).

Early Ray Termination. The scaled SDFs may overlap with one another as they now have a different footprint than they originally had. To avoid surface overlaps with a smaller SDF, we terminate the sphere tracing process when the distance between a ray and the surface is smaller than a modified, *larger* epsilon value than normally used. This ensures that the ray doesn't bypass the edges of surfaces and provides a fast distance approximation. The drawback of this approach is that when we set a higher epsilon, shapes with edges start to become more rounded eliminating the edges and providing an inaccurate approximation (Figure 6.3c).

6.2.2 Main pass: Full-resolution

In the second pass, we perform sphere tracing at full resolution (4K in our system), for both primary and shadow rays [15] to produce the final frame. We sample the approximated buffer to get the approximated distance, and we set the origin of each ray in the 3D world by reconstructing its position based on the distance stored in the low resolution buffer - instead of the camera origin. We then perform sphere tracing for each pixel. This results in rays performing significantly fewer steps compared to the ground truth sphere tracing (Figure 6.5). In our method, we take advantage of the additional computational resources by in turn spawning more rays to enable anti-aliasing without increasing the overall frame time cost. For the second, high resolution pass of soft shadow tracing, we use the approximated distance to trace the shadows and set a *shadow hardness* k value according to the

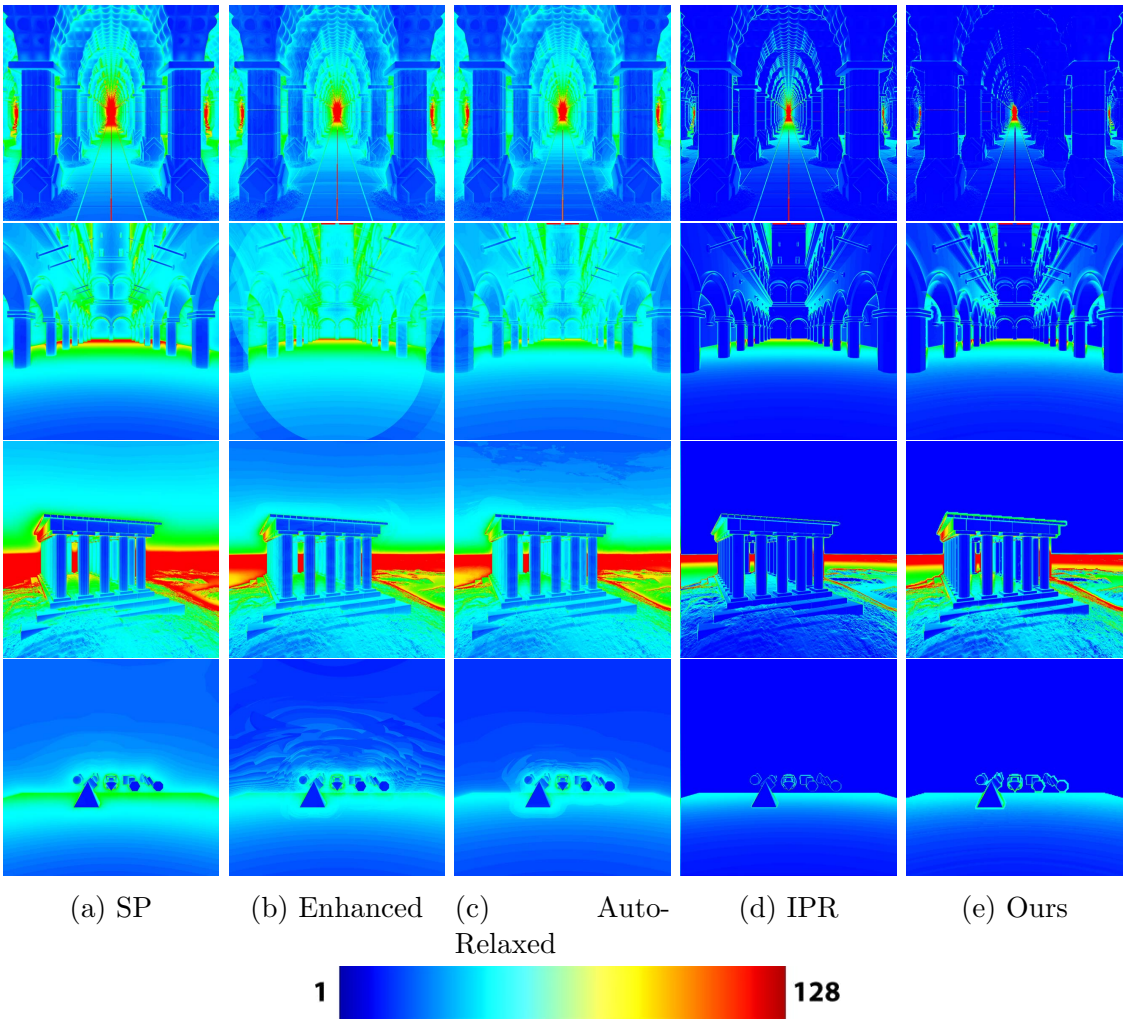
effect we want to achieve (higher k value "harder" shadows) (Figure 6.8).

Figure 6.4: *The scenes used in the evaluation, by Inigo Quilez [1] and [3].*

6.3 Evaluation

We conducted a performance and visual fidelity assessment of our system, our primary objective being twofold: to demonstrate the superior efficiency of our method and to validate that our rendering outputs attain comparable (or better) visual quality to the state-of-the-art and ground truth sphere tracing. We compare our method for primary rays against the Enhanced Sphere tracing [10], Auto-Relaxed Sphere Tracing [46], Inverse Pyramid Rendering (IPR) [51] and classic sphere tracing [15] which we consider the ground truth. We compare our method for shadow rays (soft shadows) against all aforementioned method excluding IPR as it does not support shadows. To ascertain the significance of key components within our rendering pipeline, we also execute an ablation study. The study delves into the impact of early termination and uniform scaling of SDFs with or without soft shadows on rendering performance and quality output. By including/excluding these components and analysing their effects, we gain valuable insights into their respective contributions to the overall rendering process.

We leveraged high quality scenes as depicted in Figure 6.4 (most of them provided by Inigo Quilez, with permission [1]), all represented through procedural SDFs without explicit geometry. The scenes encompass a diverse range of object complexities and environments. The scene "Primitives," serves as an open environment, rendering basic shapes against a plane with a checkerboard pattern. The scene "Column & Lights" presents a more intricate closed setting. The scene dubbed "Greek Temple" showcases a landscape, comprising procedurally generated cliff formations and an ancient temple structure. The last scene is the highly complex Sponza atrium (used in [3]).



(a) SP (b) Enhanced (c) Auto-Relaxed (d) IPR (e) Ours

Figure 6.5: *Left to right, Primary rays total step count Sphere Tracing (SP) vs Enhanced (EN) Sphere tracing vs Auto-Relaxed (AR) Sphere tracing vs Inverse Pyramid Rendering (IPR) vs Ours.*

6.3.1 Visual fidelity

We employed the Structural Similarity Index (SSIM) [95], PSNR [97], and FLIP metric [96] to evaluate the visual quality achieved by our method in comparison to ground truth data obtained via full sphere tracing and state-of-the-art algorithms. Figure 6.6 provides a visual representation illustrating local SSIM values and the error map produced by FLIP for the aforementioned methods without shadows compared against ground truth. Table 6.1 summarises the overall SSIM scores, mean error values resulting from FLIP analysis, and PSNR across different scenes without shadows being rendered for all methods and with shadows for all but IPR (not supported). Figure 6.7 provides a visual representation illustrating local SSIM values and the error map produced by FLIP with shadows.

For all scenes without shadows, our method achieved higher PSNR, and SSIM & FLIP scores compared to the state of the art, demonstrating that our primary

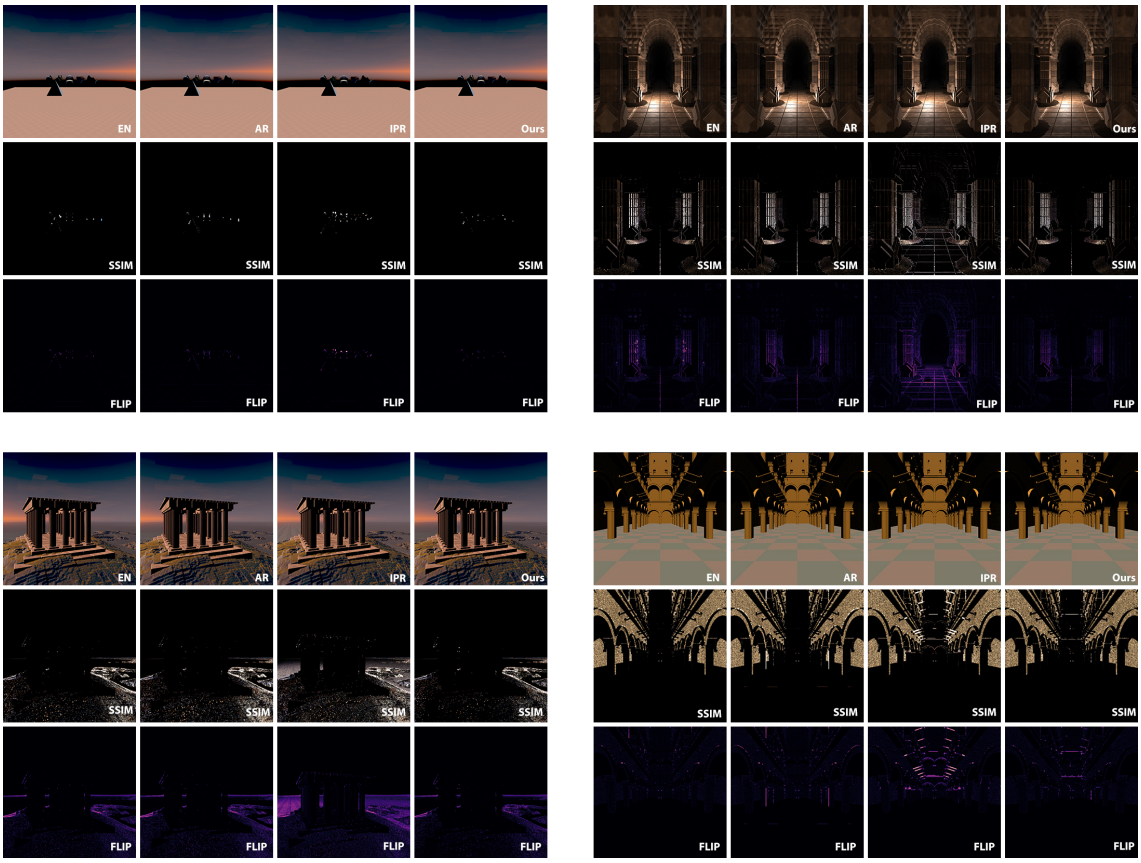


Figure 6.6: Visualization of the local SSIM and FLIP difference/error map for the scenes used in evaluation without shadows, compared against ground truth sphere tracing. SSIM images are normalized for better visualization. The global SSIM value indicates high structural similarity and the mean error measured by the FLIP metric approaches zero for all scenes in IPR method and our method, indicating a very high similarity to ground truth with our method outperforming slightly in SSIM and FLIP the IPR and significant in PSNR.

ray sphere tracing produce images closer to the ground truth, i.e., with less artifacts than the state-of-the-art. With shadows enabled our method achieves better results compared to the state-of-the-art in two of the four scenes. For the other two scenes our method performs only very slightly worse (Table 6.1) while being significantly faster. This is not readily reflected in the actual images (Figure 6.7). Given that we observe a higher error at open environments and only at high-depth values we hypothesize this is due to how we handle depth accuracy in the accelerated low resolution buffer.

6.3.2 Performance Evaluation

For the performance evaluation we calculated the time to render a frame and visualise the spawned rays/second for the aforementioned methods with and without shadows, for 1 sample/pixel or 4 samples/pixel Multisample Anti-Aliasing (MSAA)

		No Shadows			
Scene	Metrics	AR	EN	IPR	OURS
Primitives	PSNR (dB)	57.4427	56.0154	52.4472	**60.3559**
	SSIM	0.99991	0.9998	0.99929	**0.99997**
	FLIP	0.000071	0.000069	0.014882	**0.000038**
Columns	PSNR (dB)	33.6685	33.308	38.9716	**34.1242**
	SSIM	0.9678	0.9676	0.98775	**0.97063**
	FLIP	0.017435	0.015616	0.010609	**0.013130**
Greek Temple	PSNR (dB)	28.3801	29.4527	27.9799	**30.199**
	SSIM	0.94161	0.95423	0.92016	**0.9615**
	FLIP	0.024332	0.020483	0.041158	**0.019113**
Sponza	PSNR (dB)	34.6855	35.5574	39.769	**36.434**
	SSIM	0.90047	0.99364	0.89431	**0.91353**
	FLIP	0.014963	0.007559	0.019894	**0.012358**

		Shadows		
Scene	Metrics	AR	EN	OURS
Primitives	PSNR (dB)	50.1228	51.7603	**54.4241**
	SSIM	0.99963	0.99971	**0.99985**
	FLIP	0.001202	0.001656	**0.000604**
Columns	PSNR (dB)	**34.0404**	27.8927	31.4313
	SSIM	**0.96814**	0.93306	0.96041
	FLIP	**0.020769**	0.049922	0.026841
Greek Temple	PSNR (dB)	32.7802	**38.0089**	34.9144
	SSIM	0.95652	**0.98274**	0.98003
	FLIP	0.031178	**0.011753**	0.026819
Sponza	PSNR (dB)	27.1012	34.8798	**38.572**
	SSIM	0.89368	0.90788	**0.97925**
	FLIP	0.020771	0.446032	**0.011707**

Table 6.1: The global SSIM value indicates high structural similarity and the mean error measured by the FLIP metric approaches zero for all scenes in IPR method and our method, indicating a very high similarity to ground truth with our method outperforming slightly in SSIM and FLIP the IPR and significant in PSNR.

compared to the ground truth (full sphere tracing) which we show in Tables 6.9 and 6.10. The target resolution was Ultra HD (3840 x 2160). The computer used had an Intel i9-12900F CPU, 32GB RAM, and a single Nvidia GPU RTX-3070Ti with 8GB RAM. We present comparison tables of the achieved performance gains against the state-of-the-art and visualize the reduction in steps for all methods.

For the Primitives scene, our method consistently outshines others in terms of frame time (FT) and Million Rays per Second (MR/s), showcasing remarkable reductions in rendering time (speedup 3×) and notably higher efficiency. With anti-aliasing turned off, our method reduces rendering times to half in most scenes compared to the state-of-the-art and with anti-aliasing turned on our method still achieves significant reductions in rendering time.

In the same tables (6.9, 6.10), we also provide a comparison of our method against the state-of-the-art with soft shadows enabled. In the Primitives scene, our method achieved lower rendering time compared to the ground truth with an improvement in performance by 2.1× because most shadow rays finished tracing after 1 step (Figure 6.8). In the Columns scene, we achieved an improvement of 1.52×, whereas in the Greek Temple scene, we achieved an improvement of 1.8×. In the Sponza atrium, the rendering time increases significantly (shadows take up to 20 ms) compared to other scenes due to scene complexity (increased computations to evaluate more SDFs). However, our method can still improve performance by 1.5×. Overall, our approach achieves better rendering times compared to alternative methods across the board.

Compared to the state-of-the-art, our method outperforms previous methods as it skips a significant number of steps (Figure 6.5). For shadow rays, our method improves overall performance by terminating tracing early on the lower resolution buffer based on attenuation resulting in fewer steps overall, and also by reducing the steps in the full resolution rendering by giving an approximate head start to bundles of rays (to 16 rays in the full resolution for just 1 in the low resolution buffer) (Figure 6.8).

6.3.3 Ablation Study

To evaluate the effectiveness of our rendering pipeline's key modules, namely a) early ray termination and b) scaling of SDFs within the approximated buffer, we conduct an ablation study. The results obtained from this study, with uniform scaling set to 1.005 and ϵ set to 0.05 for the generation of the approximated buffer, are shown in Table 6.2.

6.4 Chapter Summary

In this chapter, we detail our innovative rendering method that effectively reduces the computational cost associated with ray tracing through careful adjustments to ray step sizes and the use of low-resolution buffers. Our goal is to enhance the performance of sphere tracing for signed distance fields (SDFs) while maintaining—if not improving—visual fidelity.

Our method begins with a pre-processing pass where we render primary and shadow rays using a low-resolution buffer. By scaling the SDFs—enlarging implicit surfaces for better visibility—we ensure that small features are detected without the generation of excessive artifacts. We also set a larger epsilon value during this pass,

allowing early termination of rays that are likely to converge on high-cost surface edges. This significantly cuts down the overall number of steps required during rendering.

In the main pass, we conduct sphere tracing at full resolution, utilizing the distances approximated in the low-resolution buffer to set ray origins closer to the surfaces. This strategy results in fewer ray steps and allows for the addition of multiple rays for enhanced anti-aliasing without significantly increasing frame time.

Our thorough evaluation process demonstrates that our method attains superior performance metrics against traditional sphere tracing techniques, including Enhanced Sphere Tracing and Inverse Pyramid Rendering. The results reveal modestly improved visual fidelity scores and impressively reduced rendering times across various complex scenes, confirming the efficacy of our dual-phase approach in achieving both high efficiency and quality in ray tracing.

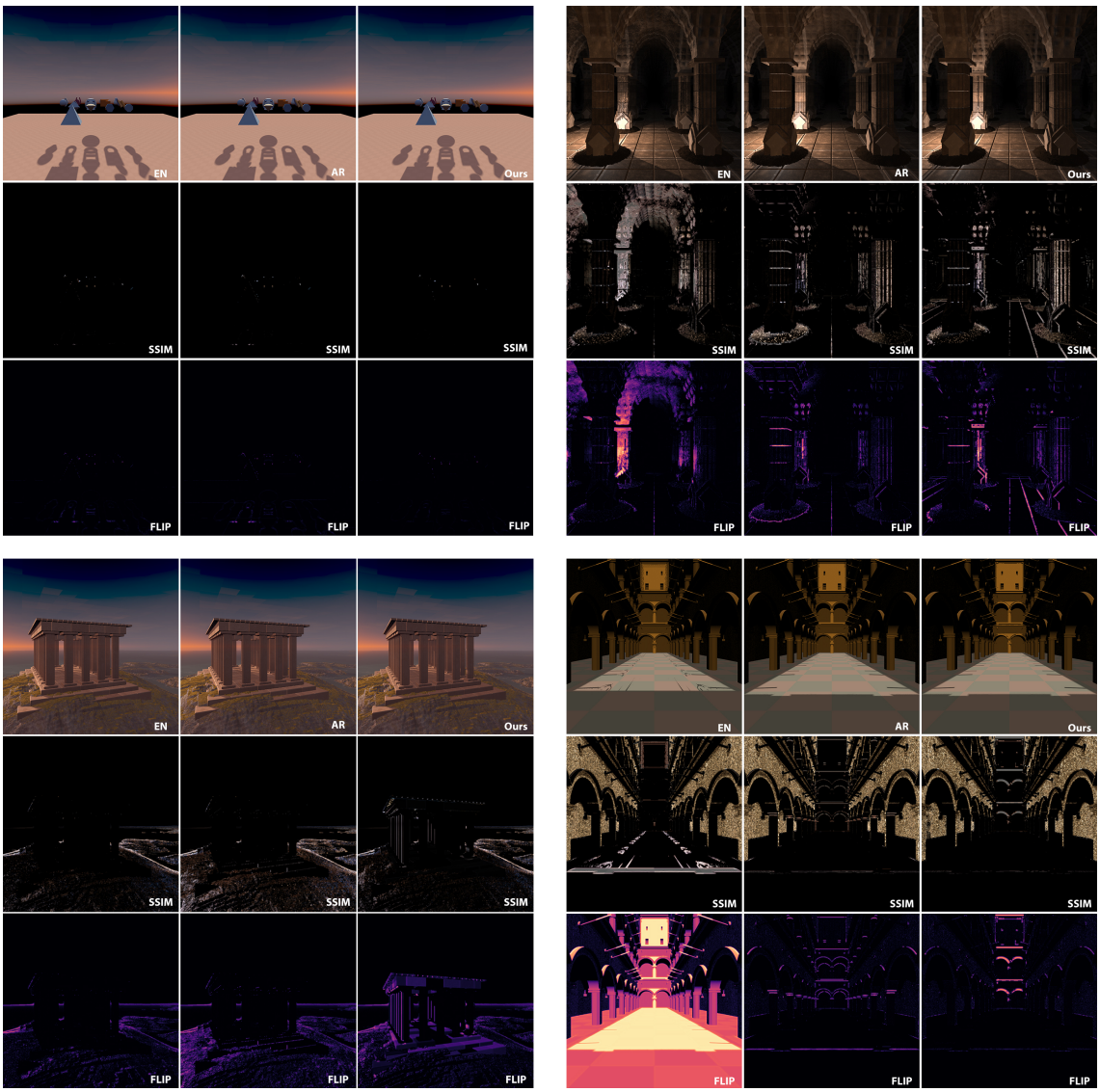
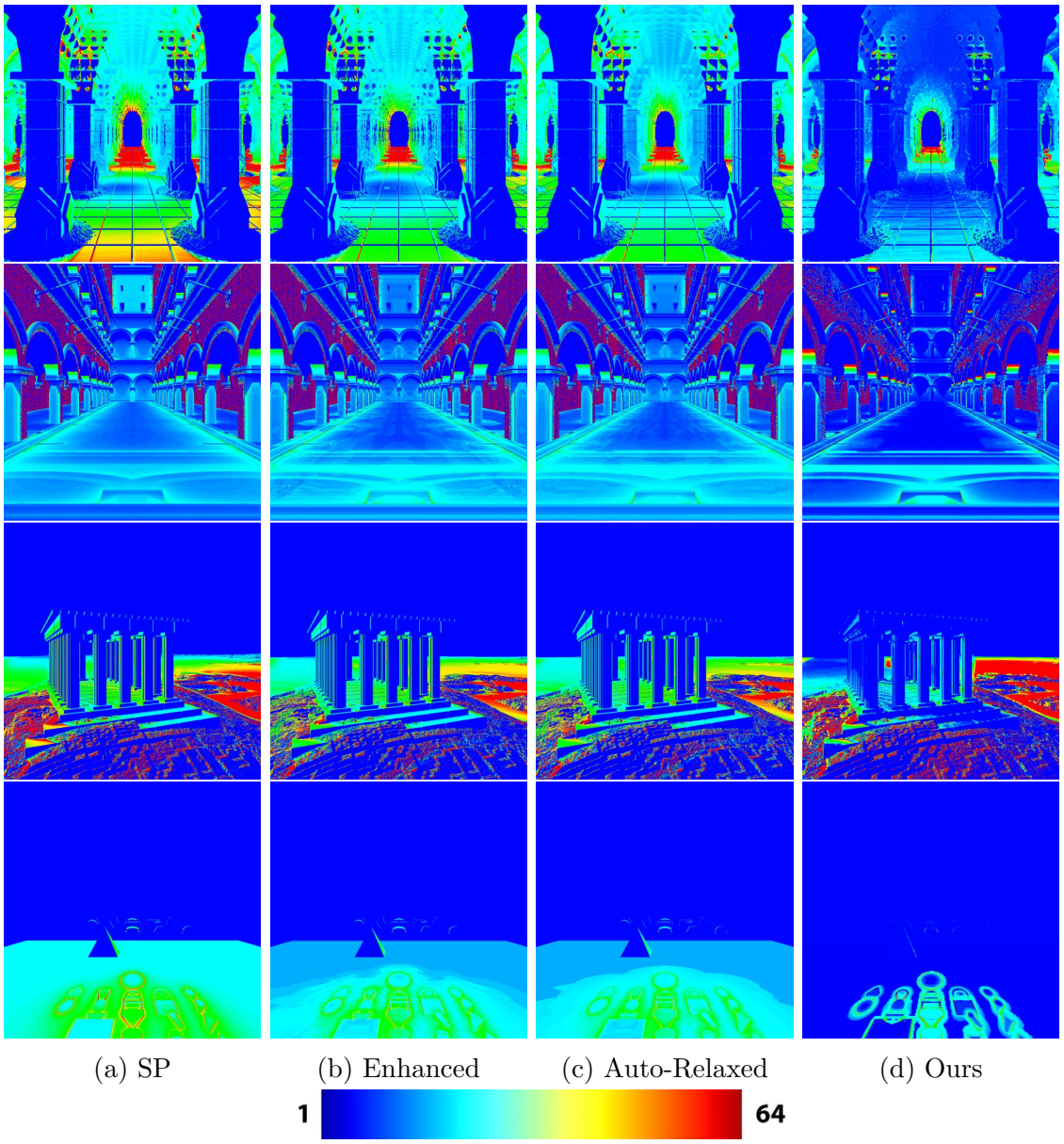
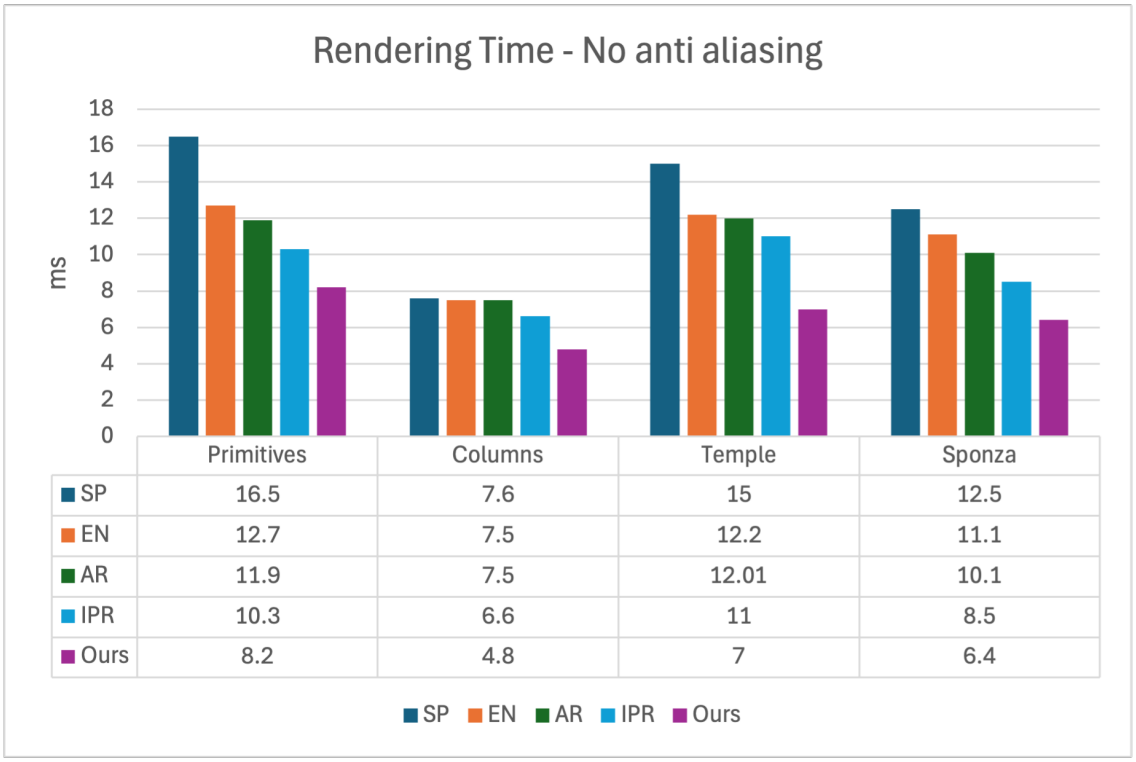


Figure 6.7: Visualization of the local SSIM and FLIP difference/error map for the scenes used in evaluation with shadows turned on, compared against ground truth sphere tracing. IPR is excluded from the comparison due to shadows being unsupported. SSIM images are normalized for better visualization. The global SSIM value indicates high structural similarity and the mean error measured by the FLIP metric approaches zero for all scenes in IPR method and our method, indicating a very high similarity to ground truth with our method outperforming slightly in SSIM and FLIP the IPR and significant in PSNR.

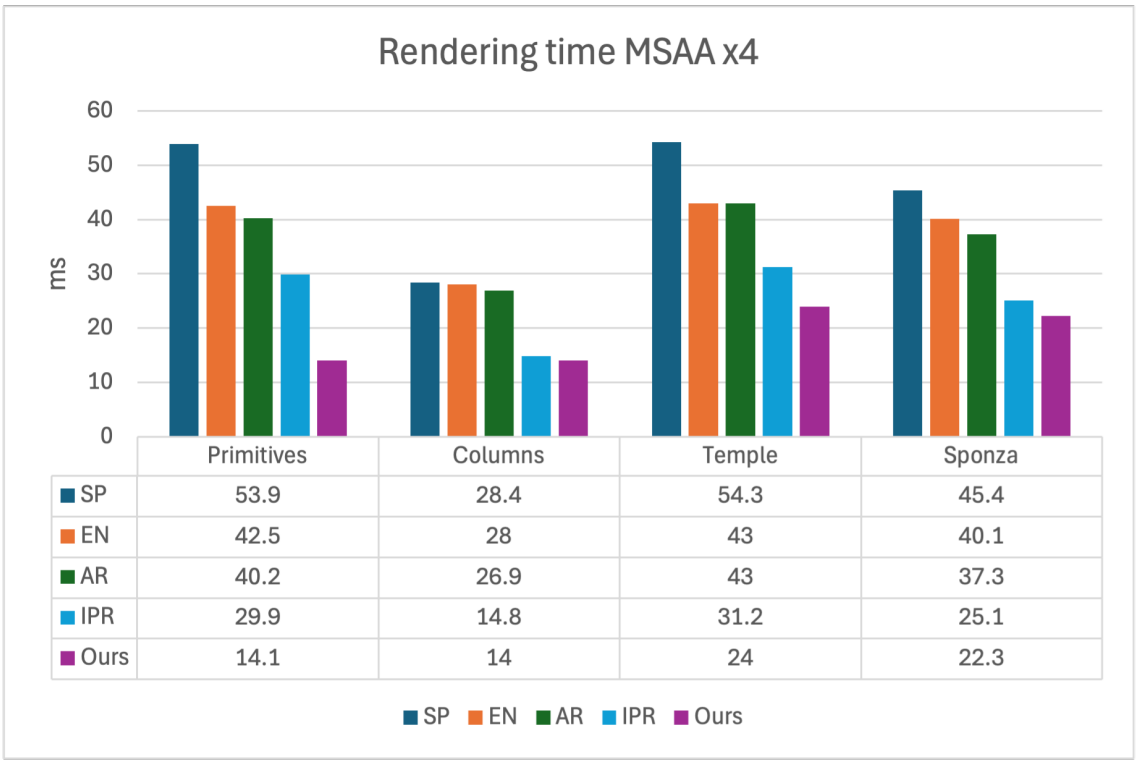


(a) SP (b) Enhanced (c) Auto-Relaxed (d) Ours

Figure 6.8: *Left to right, Shadows rays total step count Sphere Tracing (SP) vs Enhanced (EN) Sphere tracing vs Auto-Relaxed (AR) Sphere tracing vs Ours. IPR is exluded as it does not support shadow rendering.*

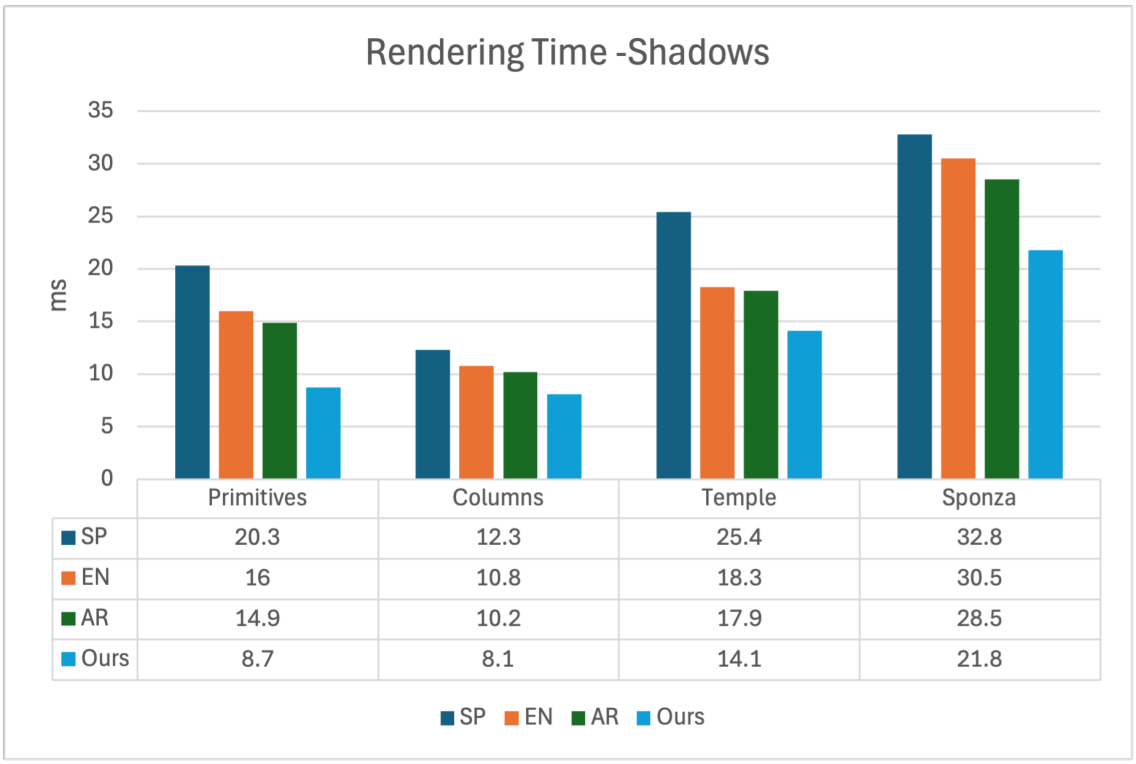


	Primitives	Columns	Temple	Sponza
SP	16.5	7.6	15	12.5
EN	12.7	7.5	12.2	11.1
AR	11.9	7.5	12.01	10.1
IPR	10.3	6.6	11	8.5
Ours	8.2	4.8	7	6.4



	Primitives	Columns	Temple	Sponza
SP	53.9	28.4	54.3	45.4
EN	42.5	28	43	40.1
AR	40.2	26.9	43	37.3
IPR	29.9	14.8	31.2	25.1
Ours	14.1	14	24	22.3

Figure 6.9: Frame rendering time for each scene in milliseconds; metrics for 1 and 4 samples per pixel. Evaluating the performance of Sphere Tracing (SP), Enhanced Sphere Tracing (EN), Auto-Relaxation Sphere Tracing (AR), Inverse Pyramid Rendering (IPR), and our method for primary rays only. Auto-Relaxed utilized a parameter (b) value of 0.3, Enhanced Sphere Tracing employed an omega value of 0.5, and IPR utilized a kernel size of 3.



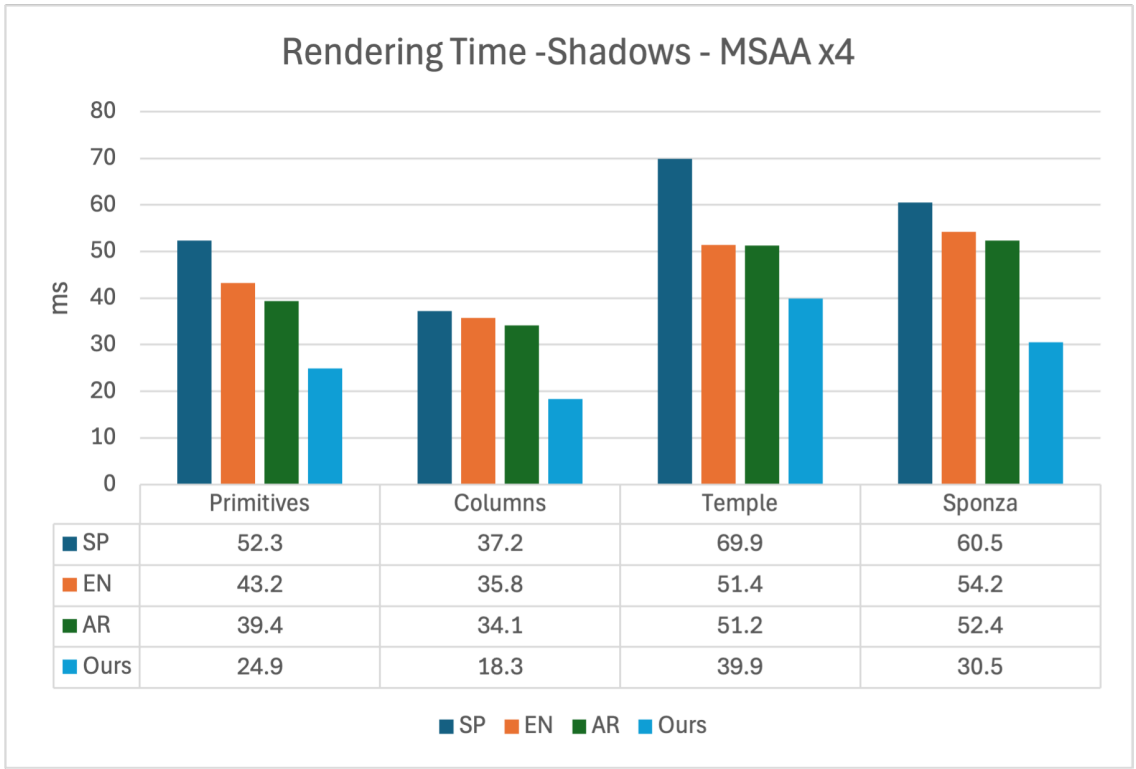


Figure 6.10: Frame rendering time for each scene in milliseconds; metrics for 1 and 4 samples per pixel. Evaluating the performance of Sphere Tracing (SP), Enhanced Sphere Tracing (EN), Auto-Relaxation Sphere Tracing (AR), and our method for primary and shadow rays. Auto-Relaxed utilized a parameter (b) value of 0.3, Enhanced Sphere Tracing employed an omega value of 0.5. Excluded IPR from shadow measurements as shadows are not supported.

		No Shadows		
Scene	Metrics	W/O ET	W/O SC	ET & SC
Primitives	FT (ms)	**7.1**	8.4	8.6
	PSNR (dB)	44.7198	44.7449	**60.3559**
	SSIM	0.99299	0.99402	**0.99997**
	FLIP	0.003921	0.003870	**0.000038**
Columns	FT (ms)	6.2	**6.0**	6.6
	PSNR (dB)	35.4043	37.7992	**38.9716**
	SSIM (dB)	0.97911	0.9852	**0.98775**
	FLIP (dB)	0.015361	0.011609	**0.010609**
Temple	FT (ms)	**7.4**	8.0	8.3
	PSNR	25.3528	31.9126	**31.9215**
	SSIM	0.93149	0.97285	**0.97286**
	FLIP	0.039382	0.015598	**0.012358**
Sponza	FT (ms)	**6.8**	7.7	8.2
	PSNR (dB)	29.0528	35.7455	**39.9769**
	SSIM (dB)	0.97391	0.98955	**0.99364**
	FLIP (dB)	0.024387	0.010603	**0.007359**
		Shadows		
Scene	Metrics	W/O ET	W/O SC	ET & SC
Primitives	FT (ms)	**8.3**	10.3	10.5
	PSNR (dB)	27.9018	35.2168	**54.3279**
	SSIM	0.97347	0.99528	**0.99985**
	FLIP	0.031284	0.003200	**0.000604**
Columns	FT (ms)	9.0	**8.8**	9.9
	PSNR (dB)	24.9847	25.0812	**31.4313**
	SSIM (dB)	0.82202	0.82725	**0.96041**
	FLIP (dB)	0.011528	0.010869	**0.026841**
Temple	FT (ms)	**10.6**	16.4	16.5
	PSNR	27.6492	34.1007	**35.168**
	SSIM	0.93625	0.97896	**0.97925**
	FLIP	0.070749	0.032912	**0.019915**
Sponza	FT (ms)	**18.6**	22.1	22.8
	PSNR (dB)	27.0548	38.4397	**38.172**
	SSIM (dB)	0.89888	0.95915	**0.96332**
	FLIP (dB)	0.046430	0.011139	**0.010707**

Table 6.2: Results of the Ablation study. Early Termination (ET) and Scaling (SC) enabled or disabled for primary and shadow rays. Both components improve both the computational efficiency and the image quality across the board, with minor exceptions.

Chapter 7

Conclusion, Limitations and Future Work

In this work, we present a series of innovative rendering methods developed progressively to address the computational challenges associated with real-time graphics rendering, particularly for path tracing and sphere tracing techniques. Our research journey began with a focus on optimizing rendering performance for both desktop and virtual reality (VR) applications, with the ultimate goal of achieving high-fidelity visuals while maintaining real-time processing capabilities. Each step in our research builds upon the insights and advancements of the previous one, culminating in a comprehensive suite of techniques that significantly enhance rendering efficiency and quality.

Our research began with the development of a perceptual sandbox designed to emulate a foveated path tracer. This initial study aimed to determine eccentricity angle thresholds for imperceptible foveated path tracing based on models of human visual acuity. Despite hardware limitations preventing real-time path tracing at the time, we successfully utilized pre-rendered image buffers and post-process foveation to identify conservative eccentricity thresholds at which image manipulations become detectable. Through a series of perceptual studies—including pair studies, ramp studies, and slider tests—we established mean eccentricity values of 10.25°, 21°, and 16.95°, respectively. These findings highlighted the significant influence of testing methodologies and monitor refresh rates on perceptual sensitivity to blur.

A key outcome of this stage was the computational complexity analysis, which demonstrated the potential for substantial performance improvements in path tracing through foveated rendering. By reducing the number of primary rays required, we estimated a 2x speed-up for a foveal region of 30° and a middle region of 75°, with even greater gains (3x speed-up) for a foveal region of 15° and a middle region

of 65°. This foundational work laid the groundwork for subsequent research into foveated rendering techniques.

Building on the insights from our foveated path tracing research, we next turned our attention to sphere tracing, a technique widely used for rendering implicit surfaces. We introduced Inverted Pyramid Rendering (IPR), a novel method for rendering implicit surfaces using sphere tracing. To further optimize this approach, we integrated foveated rendering principles, resulting in the development of Foveated Inverted Pyramid Rendering for sphere-tracing. This method emphasized the importance of rendering high-fidelity edges in the periphery while reducing computational costs in less critical regions.

Through rigorous perceptual evaluations, we estimated the parameters necessary to ensure that the manipulative effects of foveation remained imperceptible to users. Benchmarking our methods against ground truth data confirmed significant improvements in sphere tracing performance, enabling the rendering of complex Signed Distance Field (SDF)-based scenes in VR that were previously unattainable. This stage of our research demonstrated the practical applicability of foveated rendering in real-time VR environments, marking a significant step forward in rendering efficiency.

The final stage of our research focused on further accelerating sphere tracing by introducing a novel technique that combines early ray termination and SDF scaling on a low-resolution buffer. This approach allowed us to significantly reduce computational overhead while maintaining high visual quality. We extended this method to include high-quality accelerated soft shadows, leveraging the same low-resolution buffer to achieve efficient rendering of complex lighting effects.

Benchmark comparisons with ground truth and state-of-the-art methods demonstrated a remarkable increase in sphere tracing performance, independent of shadow presence. This advancement enabled the rapid rendering of intricate SDF-based scenes without compromising visual fidelity. An ablation study provided deeper insights into the contributions of individual components, confirming the importance of the key innovations we introduced. This stage represents the culmination of our progressive research journey, delivering a robust and efficient solution for real-time rendering of complex scenes.

Our research journey—from foveated path tracing to advanced sphere tracing techniques—demonstrates a clear progression in addressing the challenges of real-time rendering. Each stage built upon the insights and advancements of the previous one, resulting in a comprehensive set of methods that push the boundaries of rendering efficiency and quality. These contributions not only advance the academic

understanding of rendering techniques but also have significant practical implications for applications in VR, gaming, and real-time graphics.

Looking ahead, the integration of emerging technologies such as neural rendering and Gaussian splatting offers exciting opportunities to further enhance our methods. By combining the strengths of these approaches with our foveated and acceleration techniques, we can continue to drive innovation in real-time rendering, paving the way for even more immersive and efficient visual experiences.

7.1 Limitations

Despite our promising findings, there are several essential limitations worth discussing:

Emulated Foveated Path-tracing: A primary limitation arose from the 60Hz refresh rate of our monitor, which introduced a minimum end-to-end latency of up to 16.6ms. Extensive measurements of our pipeline identified the monitor as the primary bottleneck, with the eye tracker and processing pipeline operating smoothly at 90Hz and above 200Hz, respectively, without any delays. Although this total latency of 16.6ms is relatively low, it may have inadvertently biased our measured eccentricity thresholds towards higher values. Rapid eye movements, such as swift saccades, might have skewed the detection thresholds, causing participants to report larger eccentricities. Some subjects also indicated that they noticed image manipulations during instances of multiple blinks, attributable to tracking errors—a challenge similarly documented in prior research. Additionally, the choice of a full-HD monitor for experimentation could limit the applicability of our results to higher resolutions, such as 4K or 5K monitors. While we opted for the most common resolution currently in use, future work in higher-resolution displays may yield different perceptibility thresholds. However, we anticipate that a proportional or even more significant performance boost would be achievable on modern high-resolution monitors.

An Inverted Pyramid Acceleration Structure Guiding Foveated Sphere Tracing for Implicit Surfaces in VR: Although our proposed depth filtering method yielded promising results, it does not guarantee the preservation of all fine details, especially at coarse resolution levels. The comparison of IPR to ground truth reaffirmed that the missing details from high-frequency content were imperceptible, as corroborated by our SSIM and FLIP metrics, which indicated only minimal differences between the images. Our techniques primarily focused on ray marching for geometric edges, which are crucial for rendering performance, but did not specifically address the implications of textural or shading edges, which could detract from

user experience. Although our method supports accounting for these edges in the edge map through low-resolution albedo or shaded maps, implementing this would introduce additional rendering costs without improving performance.

Additionally, the HTC Vive Pro Eye headset introduces a limitation by locking the frame rate to 45 frames per second (fps) whenever the frame rate drops below 60 FPS. This constraint could impede our method from realizing its full potential in VR applications, as the foveated rendering could fail to synchronize adequately with the measured eye fixation rate under lower refresh rate conditions. We also noted that, during implementation, we excluded the refresh rate variable from our experiments, due to instances of low fps in VR, particularly when the headset locked down to 45 fps, which created noticeable latency and visual artifacts, especially during rapid movements.

In our desktop setup, pre-testing indicated that refresh rates could be lowered to one-fourth of the foveal refresh rate without resulting in perceptible artifacts. Further evaluation would be necessary to assess whether this drop in refresh rates in the periphery is detectable and impactful when using a desktop eye tracker.

Skipping Spheres: SDF Scaling & Early Ray Termination for Fast Sphere Tracing: This method currently relies on uniform SDF scaling not informed by depth information while performing sphere tracing. This limitation inhibits the potential to allocate resources more efficiently, particularly in foveated rendering scenarios. Future improvements could explore the incorporation of selective SDF scaling within low-resolution buffers, which would avoid scaling in areas that do not require it, thereby optimizing rendering performance. Moreover, integrating this method with our previously proposed Inverted Pyramid Acceleration Structure could foster a more efficient foveated rendering approach, leading to further enhancements in overall rendering performance.

7.2 Future Work

Looking towards the future, our research opens multiple avenues for continued exploration and advancement:

Emulated Foveated Path Tracing: One significant direction for future work will involve implementing our foveated path tracing method into a functional rendering system, contingent upon hardware capabilities. Real-time evaluations of our technique when applied to complex high-polygon models, large scanned point clouds acquired through drone imaging, or complex implicit surfaces can provide significant insights. We also plan to investigate the integration of cutting-edge denoising

techniques combined with Deep Learning Super Sampling (DLSS) in conjunction with our foveated rendering model. This combination has the potential to substantially improve both efficiency and effectiveness. Additionally, we could aim to explore how latency affects the visual angle at which foveation becomes perceptible in peripheral vision—a critical factor in the continuous evolution of VR technology.

Integration with Neural Rendering: Neural rendering techniques, such as Neural Radiance Fields (NeRF), could be leveraged to enhance the realism of foveated path tracing. By training neural networks to predict high-fidelity details in the foveal region while approximating the periphery with lower computational cost, we could achieve a more efficient and visually compelling rendering pipeline. Furthermore, neural rendering could enable the synthesis of missing or occluded details in complex scenes, reducing the need for exhaustive geometric modeling.

Gaussian Splatting for Efficient Rendering: Gaussian splatting, a technique that represents surfaces as a collection of Gaussian kernels, could be integrated into our foveated path tracing framework. By using Gaussian splats to approximate complex geometry in the peripheral regions, we could significantly reduce computational overhead while maintaining visual quality. This approach would be particularly beneficial for rendering large-scale scenes or dynamic environments where real-time performance is critical.

An Inverted Pyramid Acceleration Structure Guiding Foveated Sphere Tracing for Implicit Surfaces in VR: For the IPR and FIPR techniques, future work may delve into the utilization of neural networks to predict the distance and direction of rays based on low-resolution depth maps. This enhancement could further optimize our inverted pyramid acceleration structure to support a broader range of visual effects, such as ambient occlusion, soft shadows, sub-scattering surfaces, and even semi-transparent materials. While our current structure has been primarily optimized for VR applications, extending its applicability to desktop settings could be an invaluable improvement.

Neural Rendering for Enhanced Effects: Neural rendering could be employed to augment the visual effects supported by the inverted pyramid acceleration structure. For instance, neural networks could be trained to predict complex lighting interactions, such as global illumination or caustics, in real-time. This would allow for more realistic rendering without the computational burden of traditional physically-based methods.

Gaussian Splatting for Dynamic Scene Representation: Gaussian splatting could be used to dynamically represent complex implicit surfaces within the inverted pyramid structure. By adapting the density and scale of Gaussian kernels based on the

foveation region, we could achieve a more efficient representation of detailed surfaces while maintaining high visual fidelity in the foveal area.

Skipping Spheres: SDF Scaling & Early Ray Termination for Fast Sphere Tracing: In advancing this methodology, we plan to address the uniform scaling limitation by developing a solution that incorporates depth information during sphere tracing. Implementing dynamic scaling in regions further from the camera could optimize performance and resource allocation, particularly for complex scenes with varying detail needs. Furthermore, the combination of our selective SDF scaling techniques with the Inverted Pyramid Acceleration Structure could foster an improved foveated rendering approach, driving performance even deeper into real-time rendering capabilities without sacrificing quality.

Neural Rendering for Adaptive SDF Scaling: Neural networks could be trained to predict optimal SDF scaling factors based on scene complexity and viewing distance. This would enable adaptive scaling that dynamically adjusts to the level of detail required in different regions of the scene, further enhancing the efficiency of sphere tracing.

Gaussian Splatting for Early Ray Termination: Gaussian splatting could be integrated into the early ray termination process to approximate distant or low-detail regions of the scene. By replacing computationally expensive SDF evaluations with Gaussian approximations in these regions, we could achieve significant performance gains without noticeable visual degradation.

Through these proposed future works, we anticipate expanding the applicability and efficiency of our rendering methods, potentially redefining the approach to real-time graphics rendering in both virtual reality and traditional computing environments. The integration of neural rendering and Gaussian splatting techniques offers exciting possibilities for enhancing the realism, efficiency, and scalability of our methods. This work not only contributes to the academic understanding of rendering techniques but also has the potential for significant practical applications in graphics software and VR technologies.

Bibliography

[1] Inigo Quilez. https://iquilezles.org/demoscene/, 2022. Accessed: 2022-09-30.

[2] Why Physically-Based Rendering. Physically-based rendering. *Procedia IUTAM*, 13(127-137):3, 2015.

[3] Andrei Chubarau, Yangyang Zhao, Ruby Rao, Derek Nowrouzezahrai, and Paul G. Kry. Cone-traced supersampling with subpixel edge reconstruction. *IEEE Transactions on Visualization and Computer Graphics*, pages 1–12, 2023.

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

[5] J. C. Hart, D. J. Sandin, and L. H. Kauffman. Ray tracing deterministic 3-d fractals. In *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '89, page 289–296, New York, NY, USA, 1989. Association for Computing Machinery.

[6] Okan Tarhan Tursun, Elena Arabadzhiyska-Koleva, Marek Wernikowski, Radosław Mantiuk, Hans-Peter Seidel, Karol Myszkowski, and Piotr Didyk. Luminance-contrast-aware foveated rendering. *ACM Trans. Graph.*, 38(4), jul 2019.

[7] 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.

[8] Benjamin Keinert, Henry Schäfer, Johann Korndörfer, Urs Ganse, and Marc Stamminger. Enhanced Sphere Tracing. In Andrea Giachetti, editor, *Smart Tools and Apps for Graphics - Eurographics Italian Chapter Conference*. The Eurographics Association, 2014.

[9] Csaba Bálint and Mátyás Kiglics. A geometric method for accelerated sphere tracing of implicit surfaces. *Acta Cybernetica*, 25(2):171–185, 2021.

[10] Csaba Bálint and Gábor Valasek. Accelerating Sphere Tracing. In Olga Diamanti and Amir Vaxman, editors, *EG 2018 - Short Papers*. The Eurographics Association, 2018.

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

[12] Thorsten Roth, Martin Weier, André Hinkenjann, Yongmin Li, and Philipp Slusallek. An analysis of eye-tracking data in foveated ray tracing. In *2016 IEEE Second Workshop on Eye Tracking and Visualization (ETVIS)*, pages 69–73. IEEE, 2016.

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

[14] David H Hubel and Torsten N Wiesel. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of physiology*, 160(1):106, 1962.

[15] John C Hart. Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*, 12(10):527–545, 1996.

[16] Steven G Parker, Heiko Friedrich, David Luebke, Keith Morley, James Bigler, Jared Hoberock, David McAllister, Austin Robison, Andreas Dietrich, Greg Humphreys, et al. Gpu ray tracing. *Communications of the ACM*, 56(5):93–101, 2013.

[17] George Alex Koulieris, Kaan Akşit, Michael Stengel, Rafał K Mantiuk, Katerina Mania, and Christian Richardt. Near-eye display and tracking technologies for virtual and augmented reality. In *Computer Graphics Forum*, volume 38, pages 493–519. Wiley Online Library, 2019.

[18] Ibraheem Rehman, Bita Hazhirkarzar, and Bhupendra C. Patel. *Anatomy, Head and Neck, Eye*. StatPearls Publishing, Treasure Island (FL), 2023.

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

[20] Christine A Curcio, Kenneth R Sloan, Robert E Kalina, and Anita E Hendrickson. Human photoreceptor topography. *Journal of comparative neurology*, 292(4):497–523, 1990.

[21] Shojiro Nagata. The binocular fusion of human vision on stereoscopic displays—field of view and environment effects. *Ergonomics*, 39(11):1273–1284, 1996.

[22] Gian F Poggio and Tomaso Poggio. The analysis of stereopsis. *Annual review of neuroscience*, 7(1):379–412, 1984.

[23] Beatriz Luna, Katerina Velanova, and Charles F Geier. Development of eye-movement control. *Brain and cognition*, 68(3):293–308, 2008.

[24] Jose Luis Rubio-Tamayo, Manuel Gertrudix Barrio, and Francisco García García. Immersive environments and virtual reality: Systematic review and advances in communication, interaction and simulation. *Multimodal technologies and interaction*, 1(4):21, 2017.

[25] Olivier Cuisenaire and Benoit Macq. Fast and exact signed euclidean distance transformation with linear complexity. In *1999 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings. ICASSP99 (Cat. No. 99CH36258)*, volume 6, pages 3293–3296. IEEE, 1999.

[26] Pedro López-Adeva Fernández-Layos and Luis FS Merchante. Convex body collision detection using the signed distance function. *Computer-Aided Design*, page 103685, 2024.

[27] Miles Macklin, Kenny Erleben, Matthias Müller, Nuttapong Chentanez, Stefan Jeschke, and Zach Corse. Local optimization for robust signed distance field collision. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 3(1):1–17, 2020.

[28] J. C. Hart, D. J. Sandin, and L. H. Kauffman. Ray tracing deterministic 3-d fractals. *SIGGRAPH Comput. Graph.*, 23(3):289–296, jul 1989.

[29] William Donnelly. Per-pixel displacement mapping with distance functions. *GPU gems*, 2(22):3, 2005.

[30] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. In *ECCV*, 2020.

[31] Michael Oechsle, Songyou Peng, and Andreas Geiger. Unisurf: Unifying neural implicit surfaces and radiance fields for multi-view reconstruction. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 5589–5599, October 2021.

[32] Albert Pumarola, Enric Corona, Gerard Pons-Moll, and Francesc Moreno-Noguer. D-nerf: Neural radiance fields for dynamic scenes. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 10318–10327, June 2021.

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

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

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

[36] James T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150, August 1986.

[37] Christoph Schied, Anton Kaplanyan, Chris Wyman, Anjul Patney, Chakravarty R. Alla Chaitanya, John Burgess, Shiqiu Liu, Carsten Dachsbacher, Aaron Lefohn, and Marco Salvi. Spatiotemporal variance-guided filtering: real-time reconstruction for path-traced global illumination. In *Proceedings of High Performance Graphics*, HPG '17. Association for Computing Machinery, New York, NY, USA, 2017.

[38] Christoph Schied, Christoph Peters, and Carsten Dachsbacher. Gradient estimation for real-time adaptive temporal filtering. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 1(2):1–16, 2018.

[39] 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.

[40] Jon Hasselgren, Jacob Munkberg, Marco Salvi, Anjul Patney, and Aaron Lefohn. Neural temporal adaptive sampling and denoising. In *Computer Graphics Forum*, volume 39, pages 147–155. Wiley Online Library, 2020.

[41] Kun Zhou, Zhong Ren, Stephen Lin, Hujun Bao, Baining Guo, and Heung-Yeung Shum. Real-time smoke rendering using compensated ray marching. In *ACM SIGGRAPH 2008 Papers*, SIGGRAPH '08, New York, NY, USA, 2008. Association for Computing Machinery.

[42] Christian Reiser, Songyou Peng, Yiyi Liao, and Andreas Geiger. Kilonerf: Speeding up neural radiance fields with thousands of tiny mlps. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 14335–14345, 2021.

[43] Matej Vanco. Unity - raymarcher.

[44] Jag Mohan Singh and P. J. Narayanan. Real-time ray tracing of implicit surfaces on the gpu. *IEEE Transactions on Visualization and Computer Graphics*, 16(2):261–272, 2010.

[45] Sebastian Aaltonen. Gpu-based clay simulation and ray-tracing tech in claybook. *San Francisco, CA*, 2, 2018.

[46] Róbert Bán and Gábor Valasek. Automatic Step Size Relaxation in Sphere Tracing. In Vahid Babaei and Melina Skouras, editors, *Eurographics 2023 - Short Papers*. The Eurographics Association, 2023.

[47] Róbert Bán, Csaba Bálint, and Gábor Valasek. Area lights in signed distance function scenes. In *Eurographics (Short Papers)*, pages 85–88, 2019.

[48] Eric Galin, Eric Guérin, Axel Paris, and Adrien Peytavie. Segment Tracing Using Local Lipschitz Bounds. *Computer Graphics Forum*, 2020.

[49] Herman Hansson Söderlund, Alex Evans, and Tomas Akenine-Möller. Ray tracing of signed distance function grids. *Journal of Computer Graphics Techniques Vol*, 11(3), 2022.

[50] Andreas Polychronakis, George Alex Koulieris, and Katerina Mania. Emulating foveated path tracing. In *Motion, Interaction and Games*, MIG '21, New York, NY, USA, 2021. Association for Computing Machinery.

[51] Andreas Polychronakis, George Alex Koulieris, and Katerina Mania. An Inverted Pyramid Acceleration Structure Guiding Foveated Sphere Tracing for Implicit Surfaces in VR. In Tobias Ritschel and Andrea Weidlich, editors, *Eurographics Symposium on Rendering*. The Eurographics Association, 2023.

[52] Andreas Polychronakis, George Alex Koulieris, and Katerina Mania. Skipping Spheres: SDF Scaling & Early Ray Termination for Fast Sphere Tracing.

In David Hunter and Aidan Slingsby, editors, *Computer Graphics and Visual Computing (CGVC)*. The Eurographics Association, 2024.

[53] Wilson S. Geisler and Jeffrey S. Perry. Real-time foveated multiresolution system for low-bandwidth video communication. In Bernice E. Rogowitz and Thrasyvoulos N. Pappas, editors, *Human Vision and Electronic Imaging III*, volume 3299, pages 294 – 305. International Society for Optics and Photonics, SPIE, 1998.

[54] Sanghoon Lee, M.S. Pattichis, and A.C. Bovik. Foveated video compression with optimal rate control. *IEEE Transactions on Image Processing*, 10(7):977–992, 2001.

[55] Sanghoon Lee, M.S. Pattichis, and A.C. Bovik. Foveated video quality assessment. *IEEE Transactions on Multimedia*, 4(1):129–132, 2002.

[56] Sanghoon Lee and A.C. Bovik. Fast algorithms for foveated video processing. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(2):149–162, 2003.

[57] Junyong You, Touradj Ebrahimi, and Andrew Perkis. Attention driven foveated video quality assessment. *IEEE Transactions on Image Processing*, 23(1):200–213, 2014.

[58] Jihoon Ryoo, Kiwon Yun, Dimitris Samaras, Samir R. Das, and Gregory Zelinsky. Design and evaluation of a foveated video streaming service for commodity client devices. In *Proceedings of the 7th International Conference on Multimedia Systems*, MMSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[59] Wei-Tse Lee, Hsin-I Chen, Ming-Shiuan Chen, I-Chao Shen, and Bing-Yu Chen. High-resolution 360 video foveated stitching for real-time vr. *Computer Graphics Forum*, 36(7):115–123, 2017.

[60] Gazi Illahi, Matti Siekkinen, and Enrico Masala. Foveated video streaming for cloud gaming. In *2017 IEEE 19th International Workshop on Multimedia Signal Processing (MMSP)*, pages 1–6, 2017.

[61] Gazi Karam Illahi, Thomas Van Gemert, Matti Siekkinen, Enrico Masala, Antti Oulasvirta, and Antti Ylä-Jääski. Cloud gaming with foveated video encoding. *ACM Trans. Multimedia Comput. Commun. Appl.*, 16(1), feb 2020.

[62] Anton S. Kaplanyan, Anton Sochenov, Thomas Leimkühler, Mikhail Okunev, Todd Goodall, and Gizem Rufo. Deepfovea: Neural reconstruction for foveated

rendering and video compression using learned statistics of natural videos. *ACM Trans. Graph.*, 38(6), November 2019.

[63] David R Walton, Rafael Kuffner Dos Anjos, Sebastian Friston, David Swapp, Kaan Akşit, Anthony Steed, and Tobias Ritschel. Beyond blur: Real-time ventral metamers for foveated rendering. *ACM Transactions on Graphics*, 40(4):1–14, 2021.

[64] Toshikazu Ohshima, Hiroyuki Yamamoto, and Hideyuki Tamura. Gaze-directed adaptive rendering for interacting with virtual space. In *Proceedings of the IEEE 1996 Virtual Reality Annual International Symposium*, pages 103–110. IEEE, 1996.

[65] David Luebke, Benjamin Hallen, Dale Newfield, and Benjamin Watson. Perceptually driven simplification using gaze-directed rendering. Technical report, Tech. Rep. CS-2000-04, Department of Computer Science, University of ..., 2000.

[66] Lester C Loschky and George W McConkie. User performance with gaze contingent multiresolutional displays. In *Proceedings of the 2000 symposium on Eye tracking research & applications*, pages 97–103, 2000.

[67] Kirsten Cater, Alan Chalmers, and Greg Ward. Detail to attention: exploiting visual tasks for selective rendering. In *ACM International Conference Proceeding Series*, volume 44, pages 270–280, 2003.

[68] Peter Longhurst, Kurt Debattista, and Alan Chalmers. A gpu based saliency map for high-fidelity selective rendering. In *Proceedings of the 4th international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*, pages 21–29, 2006.

[69] Sungkil Lee, Gerard Jounghyun Kim, and Seungmoon Choi. Real-time tracking of visually attended objects in virtual environments and its application to lod. *IEEE Transactions on Visualization and Computer Graphics*, 15(1):6–19, 2008.

[70] Sebastien Hillaire, Anatole Lecuyer, Tony Regia-Corte, Remi Cozot, Jérome Royan, and Gaspard Breton. A real-time visual attention model for predicting gaze point during first-person exploration of virtual environments. In *Proceedings of the 17th acm symposium on virtual reality software and technology*, pages 191–198, 2010.

[71] George Alex Koulieris, George Drettakis, Douglas Cunningham, and Katerina Mania. C-lod: Context-aware material level-of-detail applied to mobile graph-

ics. In *Computer Graphics Forum*, volume 33, pages 41–49. Wiley Online Library, 2014.

[72] Veronica Sundstedt, Efstathios Stavrakis, Michael Wimmer, and Erik Reinhard. A psychophysical study of fixation behavior in a computer game. In *Proceedings of the 5th symposium on Applied perception in graphics and visualization*, pages 43–50, 2008.

[73] Andrew T Duchowski and Arzu Çöltekin. Foveated gaze-contingent displays for peripheral lod management, 3d visualization, and stereo imaging. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 3(4):1–18, 2007.

[74] Patrick Baudisch, Doug DeCarlo, Andrew T Duchowski, and Wilson S Geisler. Focusing on the essential: considering attention in display design. *Communications of the ACM*, 46(3):60–66, 2003.

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

[76] Xiaoxu Meng, Ruofei Du, and Amitabh Varshney. Eye-dominance-guided foveated rendering. *IEEE Transactions on Visualization and Computer Graphics*, 26(5):1972–1980, 2020.

[77] David Hoffman, Zoe Meraz, and Eric Turner. Limits of peripheral acuity and implications for vr system design. *Journal of the Society for Information Display*, 26(8):483–495, 2018.

[78] Linus Franke, Laura Fink, Jana Martschinke, Kai Selgrad, and Marc Stamminger. Time-warped foveated rendering for virtual reality headsets. *Computer Graphics Forum*, 40(1):110–123, 2021.

[79] Taimoor Tariq, Cara Tursun, and Piotr Didyk. Noise-based enhancement for foveated rendering. *arXiv preprint arXiv:2204.04455*, 2022.

[80] Hunter A Murphy, Andrew T Duchowski, and Richard A Tyrrell. Hybrid image/model-based gaze-contingent rendering. *ACM Transactions on Applied Perception (TAP)*, 5(4):1–21, 2009.

[81] Masahiro Fujita and Takahiro Harada. Foveated real-time ray tracing for virtual reality headset. *Light Transport Entertainment Research*, 2014.

[82] Adam Siekawa, Michał Chwesiuk, Radosław Mantiuk, and Rafał Piórkowski. Foveated ray tracing for vr headsets. In Ioannis Kompatsiaris, Benoit Huet,

Vasileios Mezaris, Cathal Gurrin, Wen-Huang Cheng, and Stefanos Vrochidis, editors, *MultiMedia Modeling*, pages 106–117, Cham, 2019. Springer International Publishing.

[83] Martin Weier, Thorsten Roth, André Hinkenjann, and Philipp Slusallek. Foveated depth-of-field filtering in head-mounted displays. *ACM Trans. Appl. Percept.*, 15(4), sep 2018.

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

[85] Matias Koskela, Atro Lotvonen, Markku Mäkitalo, Petrus Kivi, Timo Viitanen, and Pekka Jääskeläinen. Foveated Real-Time Path Tracing in Visual-Polar Space. In Tamy Boubekeur and Pradeep Sen, editors, *Eurographics Symposium on Rendering - DL-only and Industry Track*. The Eurographics Association, 2019.

[86] Valentin Bruder, Christoph Schulz, Ruben Bauer, Steffen Frey, Daniel Weiskopf, and Thomas Ertl. Voronoi-Based Foveated Volume Rendering. In Jimmy Johansson, Filip Sadlo, and G. Elisabeta Marai, editors, *EuroVis 2019 - Short Papers*. The Eurographics Association, 2019.

[87] Michael Stengel, Steve Grogorick, Martin Eisemann, and Marcus Magnor. Adaptive image-space sampling for gaze-contingent real-time rendering. *Computer Graphics Forum*, 35(4):129–139, 2016.

[88] Claude E. Duchon. Lanczos filtering in one and two dimensions. *Journal of Applied Meteorology and Climatology*, 18(8):1016–1022, 1979.

[89] Louis Komzsik. *The Lanczos method: evolution and application*. SIAM, 2003.

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

[91] Morgan McGuire. Computer graphics archive, July 2017. `https://casual-effects.com/data`.

[92] Stanford. The stanford 3d scanning repository, 2021.

[93] Richard Guy. *Unsolved problems in number theory*, volume 1. Springer Science & Business Media, 2004.

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

[95] Zhou Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004.

[96] Pontus Andersson, Jim Nilsson, Tomas Akenine-Möller, Magnus Oskarsson, Kalle Åström, and Mark D. Fairchild. FLIP: A Difference Evaluator for Alternating Images. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 3(2):15:1–15:23, 2020.

[97] Alain Hore and Djemel Ziou. Image quality metrics: Psnr vs. ssim. In *2010 20th international conference on pattern recognition*, pages 2366–2369. IEEE, 2010.

[98] By⟨ahref=”//commons.wikimedia.org/wiki/User:Perellonieto”title=”User:Perellonieto”⟩MiquelPerelloNieto⟨/a⟩-⟨spanclass=”int-own-work”lang=”en”⟩Ownwork⟨/span⟩,⟨ahref=”https://creativecommons.org/licenses/by-sa/4.0”title=”CreativeCommonsAttribut0”⟩CCBY-SA4.0⟨/a⟩,⟨ahref=”https://commons.wikimedia.org/w/index.php?curid=37868501”⟩Link⟨/a⟩