Hybrid and Non-Hybrid Rendering for Diffuse Global Illumination in Real-Time Applications

Diploma Thesis

Nikolaidis Dimitrios
dnikolaidis2@tuc.gr, 2015030100

Committee

Supervisor:	Professor Mania Aikaterini
Committee Member:	Professor Deligiannakis Antonios
Committee Member:	Associate Professor Samoladas Vasilis



Technical University of Crete
School of Electrical and Computer Engineering

Chania February, 2025

Αλγόριθμοι Φωτισμού σε Πραγματικό Χρόνο αξιοποιώντας Τεχνικές Υβριδικής και μη Υβριδικής Απεικόνισης

Διπλωματική Εργασία

Νικολαΐδης Δημήτριος
dnikolaidis2@tuc.gr, 2015030100

Επιτροπή

Επιβλέπουσα: Καθηγήτρια Μανιά Αικατερίνη
Μέλος Επιτροπής: Καθηγητής Δεληγιαννάκης Αντώνιος
Μέλος Επιτροπής: Αναπληρωτής Καθηγητής Σαμολαδάς Βασίλης



Πολυτεχνείο Κρήτης
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Χανιά Φεβρουάριος, 2025

Acknowledgements

I would like to take this opportunity to thank the people without whose support this thesis would not have been possible. First of all, I would like to thank my thesis supervisor Katerina Mania, for her time, help, trust, and support throughout this process. I am also grateful for this opportunity to work and expand my graphics knowledge. Working in graphics has been a goal of mine even before I joined the Technical University of Crete, and this opportunity has helped propel me to where I am today. Second, I would like to thank Andreas Polychronakis for his guidance in formulating the initial proposal and supporting me in the early research and implementation stages of the thesis. In addition, I would like to thank my family for their immeasurable support throughout all these years. Finally, I would like to thank all my friends who have been there for me throughout all stages of this thesis.

Περίληψη

Τα γραφικά και οι τεχνικές απεικόνισης σε πραγματικό χρόνο έχουν εξελιχθεί πιέζοντας τα όρια της ρεαλιστικής απεικόνισης. Ένα από τα πιο κρίσιμα στοιχεία για την επίτευξη ρεαλιστικών εικόνων είναι ο φωτισμός και συγκεκριμένα ο διάχυτος φωτισμός. Η προσομοίωση του διάχυτου φωτισμού μίας σκηνής με ακρίβεια συμβάλλει σημαντικά στην ποιότητα της απεικόνισης της και προσδίδει φωτορεαλισμό στο τελικό αποτέλεσμα. Η χρήση τεχνικών ray και path tracing συνηθίζεται για την ακριβή προσομοίωση του φωτός και επίτευξη εφέ φωτισμού. Οι εξελίξεις στην αρχιτεκτονική καρτών γραφικών έχουν εισάγει την δυνατότητα επιτάχυνσης ray-tracing και την χρήση της σε εφαρμογές πραγματικού χρόνου. Από τις εξελίξεις αυτές προέκυψε μια καινούργια κατηγορία τεχνικών απεικόνισης που αποκαλούνται υβριδικές τεχνικές. Αυτές οι τεχνικές συνδυάζουν κλασσικές τεχνικές resterization και ray-tracing για την ταχεία απεικόνιση μιας σκηνής με υψηλά επίπεδα ρεαλισμού.

Μέσω αυτής της διπλωματικής σκοπό έχουμε να μελετήσουμε την υβριδική και μη-υβριδική απεικόνιση για παραγωγή ολικού φωτισμού. Αυτό επιτεύχθηκε με την υλοποίηση ενός υβριδικού renderer με την χρήση του Vulkan API. Για την απόδοση του διάχυτου φωτισμού της σκηνής έγινε χρήση της τεχνικής Dynamic Diffuse Global Illumination (DDGI). Αυτή είναι μια τεχνική που διαιρεί τον χώρο σε τμήματα και χρησιμοποιεί σφαίρες σε κάθε τμήμα του χώρου για την συλλογή ορατής ακτινοβολίας και την ορατότητα σημείων στον χώρο. Με αυτό τον τρόπο μπορεί να παράγει διάχυτο φωτισμό σε περίπλοκες σκηνές αντιδρώντας σε αλλαγές και διατηρώντας υψηλές επιδόσεις. Παράλληλα με αυτή την τεχνική υλοποιήθηκαν τεχνικές όπως deferred rendering, variance shadow mapping, και screen space ambient occlusion (SSAO) για την απόδοση σκιών, και σκίαση από τον περιβάλλοντα χώρο.

Για την ορθή αξιολόγηση των αποτελεσμάτων του υβριδικού renderer υλοποιήθηκε ένας path-tracer, μη-υβριδικός renderer αξιοποιώντας την ίδια αναπαράσταση για την σκηνή και το Vulkan API. Αυτό μας επιτρέπει να συγκρίνουμε απευθείας τα οπτικά αποτελέσματα των δύο renderer. Συγκρίνουμε τα αποτελέσματα, τόσο τα οπτικά όσο και τον χρόνο εκτέλεσης, με εκείνα του path-tracer και αναλύουμε μετρικές όπως τον χρόνο ανά frame, και τον δείκτη structured similarity index (SSIM). Η ανάλυση αυτών των αποτελεσμάτων δείχνει ότι αυτή η υλοποίηση ενός υβριδικού renderer παράγει διάχυτο φωτισμό σε πραγματικό χρόνο, τα οπτικά αποτελέσματα του οποίου είναι συγκρίσιμα αυτά του path tracer, έχοντας ταυτόχρονα περιορισμένα οπτικά λάθη. Τα αποτελέσματα μας δείχνουν πως αυτή η υλοποίηση υβριδικού renderer έχει την δυνατότητα να ανταποκριθεί στις απαιτήσεις των εφαρμογών πραγματικού χρόνου στην παραγωγή διάχυτου φωτισμού.

Abstract

In the past decades, advances in real-time graphics have pushed the limit of realistic rendering. One of the most critical elements in achieving visual realism is lighting, specifically global illumination (GI). When correctly modeled, indirect light in a scene significantly contributes to enhancing rendering fidelity and realism. Ray and path tracing techniques are a standard way for accurately modeling light transport and in producing lighting effects. Recent GPU architectures have introduced hardware-accelerated ray-tracing features, making ray-tracing in real-time applications viable. This has given rise to the popularity of a new class of rendering approach known as hybrid rendering. These techniques combine traditional rasterization and hardware-accelerated ray tracing to balance speed and realism.

In this thesis, we aim to study hybrid and non-hybrid rendering in global illumination. This was achieved by first implementing a hybrid renderer using the Vulkan API. The renderer leverages the Dynamic Diffuse Global Illumination (DDGI) technique to generate global illumination. This hybrid technique discretizes a scene with robes and collects radiance and visibility using ray tracing and rasterization. In this way, it can handle complex scenes and lighting changes while maintaining interactive performance. Deferred rendering, variance shadow mapping, and screen space ambient occlusion (SSAO) were implemented for this renderer to produce effects like shadows, ambient occlusion, and direct illumination.

A path-traced non-hybrid renderer was also implemented to provide a baseline for evaluation using the same scene representation and the Vulkan API. This allows for a direct visual comparison of the hybrid renderer's results. We compared the performance and visual fidelity of the hybrid renderer with the path tracer by analyzing visual results and metrics such as frame time (ms) and structural similarity index (SSIM). Our analysis of these results shows that this implementation of a hybrid renderer can produce global illumination in real-time. These results are comparable to those of the path tracers, while producing minimal visual artifacts. The findings underscore this hybrid renderer's potential to generate global illumination in interactive applications and to produce photorealistic results.

Contents

Acknowledgements i

Περίληψη ii

Abstract iii

List of Figures vi

1 Introduction 1

2 Theory 5
2.1 Mathematical Background . . . 6
2.1.1 Triangles . . . 6
2.1.2 Coordinate Systems . . . 7
2.1.3 Spherical Fibonacci . . . 8
2.1.4 Uniform Random Rotations . . . 8
2.1.5 Uniform Unit Hemisphere . . . 9
2.1.6 Octahedral Mapping . . . 9
2.2 Scene . . . 9
2.2.1 Geometry . . . 10
2.2.2 Textures . . . 10
2.2.3 Materials . . . 11
2.2.4 Cameras . . . 11
2.2.5 Light Sources . . . 11
2.3 Shading . . . 13
2.3.1 Illumination Models . . . 13
2.3.2 Rasterization Pipeline . . . 15
2.3.3 Lighting . . . 16
2.4 Ray Casting . . . 19
2.4.1 Acceleration Structures . . . 19
2.4.2 Ray Tracing . . . 20
2.4.3 Path Tracing . . . 20
2.5 Post-Processing . . . 22
2.5.1 Anti-aliasing . . . 22
2.5.2 Gaussian Blur . . . 22
2.5.3 High Dynamic Range . . . 23
2.5.4 Tone Mapping . . . 23
2.5.5 Gamma correction . . . 23

3 Software Architecture and Development 25
3.1 Vulkan . . . 26
3.1.1 Overview . . . 26
3.1.2 Execution Model . . . 27
3.1.3 Resources . . . 27
3.1.4 Pipelines . . . 29
3.1.5 Synchronization . . . 30
3.1.6 Vulkan Ray Tracing Extension . . . 31
3.2 glTF 2.0 . . . 33
3.3 auto-vk-toolkit . . . 33
3.3.1 assimp . . . 34

4 Implementation **36**
4.1 Overview . . . 37
4.2 Scene . . . 37
4.2.1 Model . . . 37
4.2.2 Material . . . 37
4.2.3 Textures . . . 38
4.2.4 Camera . . . 38
4.2.5 Light Sources . . . 39
4.2.6 Acceleration Structure . . . 40
4.3 Hybrid Renderer . . . 40
4.3.1 Overview . . . 40
4.3.2 Shadow Map . . . 41
4.3.3 G-Buffer . . . 45
4.3.4 Global Illumination . . . 45
4.3.5 Ambient Occlusion . . . 50
4.3.6 Lighting . . . 51
4.4 Path Tracer . . . 52
4.4.1 Ray Generation . . . 52
4.4.2 Closest Hit . . . 52
4.4.3 Direct Light . . . 53
4.4.4 Miss . . . 54
4.5 Post Processing . . . 54

5 Analysis **57**
5.1 Overview . . . 58
5.2 Methodology . . . 58
5.3 Results . . . 59

6 Conclusion and Future Work **65**
6.1 Conclusion . . . 66
6.2 Future Work . . . 66

A Detailed Results **72**
A.1 Full Lighting . . . 73
A.2 Indirect Illumination . . . 77
A.3 Direct Illumination . . . 81
A.4 Ray per Probe Comparison . . . 82

List of Figures

1.1 Overview of the hybrid renderer's render passes and techniques. 2
1.2 A frame generated from the path tracer (left) and the hybrid renderer (right). . . . 3

2.1 A triangle $\triangle \mathbf{V_0 V_1 V_2}$, its plane P and normal $\vec{n}$, tangent $\vec{t}$, and bitangent $\vec{b}$ vectors. 6
2.2 A triangle defined by vertices $\mathbf{V_0}, \mathbf{V_1}, \mathbf{V_2}$ with a point $\mathbf{P}$ on the triangle and its barycentric weights λ_0, λ_1, λ_2. 7
2.3 The transformation chain between local space to the screen [Vri15]. 8
2.4 Spherical Fibonacci points and directions on a unit sphere for various n values. . . 8
2.5 Mapping of an octahedron to a 2D square. 9
2.6 Stanford bunny. From left to right the same object represented by a low resolution mesh to a high resolution mesh. 10
2.7 A cube in Blender 3D. Right is a 3D representation of the Cube and its vertices. Left are those vertices unwrapped into the UV plane. 10
2.8 On the left, the Stanford bunny drawn by a perspective camera. On the right, a diagram of a perspective view frustum [Vri15]. 11
2.9 On the left, the Stanford bunny drawn using an orthographic camera. On the right, a diagram of an orthographic view frustum [Vri15]. 12
2.10 Directional light source illustrated by a sun with its light rays shining on a surface. 12
2.11 A point light of radius r. Light is attenuated as it leaves the source. 13
2.12 A spot light with inner cone of angle ϕ and inner angle of θ. 13
2.13 An example square defined in screen space with its Vertex buffer and Vertex + Index buffer alternative. 14
2.14 Graphics pipeline stage diagram with a small example output of each stage [Ove15]. 15
2.15 A single quad rendered with a brick texture. Lit from above. Left has only a diffuse texture. Right has diffuse and normal texture. 17
2.16 Diagram explaining shadow mapping. The left side shows how light rays are emulated when drawing the scene from the light source's point of view. The right side shows how the fragment P's visibility can be sampled by transforming its position to the light source's space. [Vri15] 18
2.17 A quick demonstration on the effect ambient occlusion can have on a scene. [Vri15]. 19
2.18 A diagram demonstrating bounding volume hierarchies (BVH). The scene on the left with its objects and bounding boxes A, B, and C is described and traversed as the tree seen on the right [Sch11]. 20
2.19 Visualization of the rendering equation for point $\mathbf{x}$ and incidence $\vec{\omega_i}$ and outgoing direction $\vec{\omega_o}$. 21
2.20 Three triangle edges being rasterized with various anti-aliasing filtering settings applied. Left 0 pixels, middle .75 pixels, right 1.5 pixels. 22
2.21 A set of blurred images and the kernels that were used to blur them. 23

3.1 Vulkan's foundational objects in relation to GPU hardware or device and the host or operating system. 27
3.2 Vulkan execution model. 28
3.3 Vulkan command buffer execution order as submitted by the CPU and as run on the GPU. 28
3.4 A multi-threaded example of command submission to the GPU. 29
3.5 Graphics pipeline stages. The solid blocks are programmable stage and the dashed blocks are not. 29
3.6 Work group and invocations of a compute dispatch. [Ove15] 30
3.7 Diagram showing the Host/Device relationship when VkFence is used. 30

3.8 Diagram showing the relationship between submissions when a VkSemaphore is used. 31
3.9 Diagram showing the relationship between two draw calls when using a pipeline barrier. . . . 31
3.10 Diagram showing the structure of a Vulkan acceleration structure. . . . 32
3.11 Diagram of Vulkan's ray tracing pipeline. Solid blocks are programmable. Dashed and gray blocks are optimal and user-programmable. The black dashed line blocks are fixed. . . . 32
3.12 Example of auto-vk graphics pipeline definition for a simple debug line renderer. . 34

4.1 Vertex data that is used for this project. . . . 37
4.2 Material struct as definition for GPU usage. . . . 38
4.3 Light struct definition as used on the GPU. . . . 39
4.4 Overview of the hybrid renderer's render passes and techniques. . . . 40
4.5 Shadow Mapping flow chart. . . . 41
4.6 Visualization of variance shadow map blurring process. From left to right: None, Horizontal Blur, Vertical + Horizontal Blur. Images are normalized for visualization. 42
4.7 Pseudo code excerpt from the shadow_map_visibility function that is used to sample the shadow map and return a visibility value. . . . 42
4.8 Pseudo code excerpt from the direct_illumination function calculating direct illumination for a given surfel. . . . 43
4.9 Pseudo code excerpt from the variance_shadow_map_visibility function. . . . 44
4.10 G-Buffer stage flow chart. . . . 45
4.11 G-Buffer render targets . . . 45
4.12 DDGI Probe states . . . 46
4.13 DDGI render targets. . . . 47
4.14 A point $\mathbf{x}$ with normal $\vec{n}$ on surface S that is being shaded by probe P of the probe cage. . . . 47
4.15 Pseudo code for sampling irradiance from the probe volume. . . . 48
4.16 This is the flow chart of the DDGI algorithm for every frame. Every block indicates each shader that is being run. Every dashed line indicates the binding of a pipeline. 48
4.17 Diagram showing the flow of the ray tracing pipeline. . . . 48
4.18 This is a flow chart showing all the stages that run when calculating probe offsets. Dashed lines show when pipelines are bound. . . . 49
4.19 Pseudo code excerpt from the code that is used to generate the SSAO kernel. . . . 50
4.20 Pseudo code excerpt from the SSAO shader. . . . 50
4.21 Ambient Occulusion stage flow chart. . . . 51
4.22 Comparison each step output render target. First no bluring. Second horizontal bluring. Third horizontal + vertical blur. . . . 51
4.23 This is the resulting HDR image of the Lighting stage. . . . 51
4.24 Overview of the Path Tracer reference renderer. These are the layers active during path tracing. . . . 52
4.25 Path tracing step flow chart. . . . 53
4.26 Pseudo code excerpt from the ray generation shader of the path tracer layer . . . 54
4.27 Pseudo code excerpt from the closest hit shader of the path tracer layer. . . . 55
4.28 Pseudo code from the direct illumination function. . . . 55
4.29 Pseudo code form the miss shader. . . . 55
4.30 This is the result of the tone mapping and gamma correcting step. . . . 56
4.31 This is the result of the AA step applying FXAA to the final frame. . . . 56
4.32 This is the result of the FXAA step. . . . 56

5.1 An area that is not covered by probes and is not lit in View 1. From left to right path-traced reference, 8x8x8, 16x16x16, and 32x32x32 probe count. . . . 59
5.2 Indirect illumination differences based on probe positioning/density in View 6. From left to right path-traced reference, 8x8x8, 16x16x16, and 32x32x32 probe count. . 60
5.3 Similarities in indirect light form roof in View 2. From left to right path-traced reference, 8x8x8, 16x16x16, and 32x32x32 probe count. . . . 60
5.4 Light bleed artifacts in hybrid renderer results based on probe position in View 4. From left to right path-traced reference, 8x8x8, 16x16x16, and 32x32x32 probe count. . . . 61
5.5 Differences in direct illumination in View 6. From left to right path-traced reference, 8x8x8, 16x16x16, and 32x32x32 probe count. . . . 61

5.6 Direct illumination differences in shadows in View 1. From left to right path-traced reference, 8x8x8, 16x16x16, and 32x32x32 probe count. 61
5.7 Differences in direct illumination in View 2. From left to right path-traced reference, 8x8x8, 16x16x16, and 32x32x32 probe count. 61
5.8 Float precision issues with path-traced reference in View 3. From left to right path-traced reference, 8x8x8, 16x16x16, and 32x32x32 probe count. 62
5.9 Indirect lighting on brick surface in View 4. From left to right path-traced reference, 8x8x8, 16x16x16, and 32x32x32 probe count. 62
5.10 Only indirect light differences in View 1. From left to right path-traced reference, 8x8x8, 16x16x16, and 32x32x32 probe count. 62
5.11 Quantitative analysis results for various configurations and probe counts 63
5.12 Performance results per technique. 64
5.13 DDGI performance results. 64
5.14 Overall performance average. 64

A.1 Direct and indirect illumination from path traced reference compared with hybrid renderer with 8 by 8 by 8 probes and 256 rays per probe. First column is the path traced reference. Second column is the hybrid renderer. Final column is the SSIM of the two previous images in the row. Every row is for each view from 1 to 6. . . . 74
A.2 Direct and indirect illumination from path traced reference compared with hybrid renderer with 16 by 16 by 16 probes and 256 rays per probe. First column is the path traced reference. Second column is the hybrid renderer. Final column is the SSIM of the two previous images in the row. Every row is for each view from 1 to 6. 75
A.3 Direct and indirect illumination from path traced reference compared with hybrid renderer with 32 by 32 by 32 probes and 256 rays per probe. First column is the path traced reference. Second column is the hybrid renderer. Final column is the SSIM of the two previous images in the row. Every row is for each view from 1 to 6. 76
A.4 Only indirect illumination from path traced reference compared with hybrid renderer with 8 by 8 by 8 probes and 256 rays per probe. First column is the path traced reference. Second column is the hybrid renderer. Final column is the SSIM of the two previous images in the row. Every row is for each view from 1 to 6. 78
A.5 Only indirect illumination from path traced reference compared with hybrid renderer with 16 by 16 by 16 probes and 256 rays per probe. First column is the path traced reference. Second column is the hybrid renderer. Final column is the SSIM of the two previous images in the row. Every row is for each view from 1 to 6. 79
A.6 Only indirect illumination from path traced reference compared with hybrid renderer with 32 by 32 by 32 probes and 256 rays per probe. First column is the path traced reference. Second column is the hybrid renderer. Final column is the SSIM of the two previous images in the row. Every row is for each view from 1 to 6. 80
A.7 Only direct illumination from path traced reference compared with hybrid renderer. First column is the path traced reference. Second column is the hybrid renderer. Final column is the SSIM of the two previous images in the row. Every row is for each view from 1 to 6 except 2 since no direct illumination is visible from that camera perspective. 81
A.8 Direct and indirect illumination from hybrid renderer with 256 rays per probe compared with hybrid renderer 512 rays per probe. Both renderers with 8 by 8 by 8 probes. First column is the hybrid renderer with 256 rays per prob. Second column is the hybrid renderer with 512 rays per probe. Final column is the SSIM of the two previous images in the row. Every row is for each view from 1 to 6. 83

Chapter 1

Introduction

Over the past few decades, 3D graphics have been used in various industries such as the film industry, video games, product design, architectural, and scientific visualization. In most applications, generating realistic images is important for immersion, storytelling, design validation, and more. Lighting, in particular, is a crucial element in the creation of compelling and realistic 3D environments. Not only does it serve to convey shape, texture, and depth, it also serves as an important cue for emotional and narrative elements. For example, in video games, light or the lack of can evoke tension, or mystery. In general, it can set the mood or tone of a given scene. Dynamic changes in light can also help to communicate the passage of time. In architectural visualization, however, lighting enables clients and designers to visualize how interior and exterior spaces will look under different lighting conditions.

Early research aimed at solving scene lighting led to the development of techniques such as ray tracing [Whi80] and path tracing [Kaj86]. These methods simulate lighting by tracing the path of light rays through a scene. In path tracing specifically, rays are traced from the viewer's perspective and followed as they bounce through the environment. As each ray interacts with objects and light sources, radiance information is collected, and the final radiance is then used to construct the rendered image. This approach accurately simulates light transport and can produce a wide range of effects, including direct and indirect illumination as well as shadows. However, it is computationally expensive, particularly in scenes with high geometric complexity.

Historically, this led to a split between real-time rasterization techniques and offline ray or path tracing. Real-time applications such as video games, extended reality, and CAD/CAE software must render frames very quickly (e.g., within 33.3 ms, 16.6 ms, or even 11.1 ms), making full light transport via path tracing impractical. In contrast, offline path tracing is typically used when photorealism is critical, such as in films and other non-interactive media, where rendering a single frame can take hours rather than milliseconds.

NVIDIA has been working to accelerate ray tracing on GPUs since 2009 with the OptiX API. These efforts have significantly reduced rendering times in ray and path-traced renderers and culminated with the introduction of the Turing GPU architecture [NVI18], which features dedicated ray tracing cores. This hardware innovation further accelerates ray tracing, while the introduction of DirectX Ray tracing (DXR) and Vulkan Ray Tracing have made hardware-accelerated ray tracing widely accessible. As a result, new hybrid rendering approaches have emerged that combine rasterization and ray tracing to leverage the advantages of both methods for improved visual quality and performance.

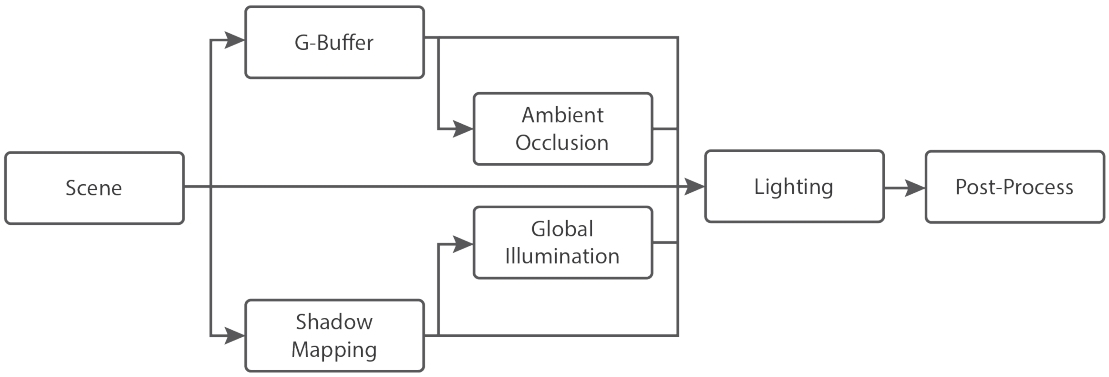


Figure 1.1: Overview of the hybrid renderer's render passes and techniques.

In this thesis, we studied hybrid rendering—and rendering in general—through the lens of diffuse global illumination. To this end, we implemented a hybrid renderer using the Vulkan API and C++. Our implementation is based on the Dynamic Diffuse Global Illumination (DDGI) technique introduced by Majercik et al. in [Maj+19], which builds on the work of [McG17]. These methods are called probe-based techniques because they discretize the scene by placing probes (spheres) that collect lighting information for the surrounding geometry. These probes form a probe volume that can later be queried for indirect lighting of the affected geometry. The key innovation of DDGI is the inclusion of additional visibility information—based on variance shadow mapping—alongside the irradiance data for each probe. Combined with GPU-accelerated ray tracing, this enables more accurate lighting, while avoiding common artifacts found in probe-based techniques.

Alongside DDGI, we implemented other techniques. The complete hybrid renderer architecture is presented in Figure 1.1. Deferred rendering is used to generate a G-Buffer, which is used when rendering the final image and in producing ambient occlusion via Screen Space Ambient Occlusion (SSAO). Variance shadow mapping is also used to generate shadows and handle direct illumination calculations during the final lighting stage. Separately, we developed a path tracer to create reference images, enabling us to compare and evaluate the results of the hybrid renderer. Finally, both the scene and post-processing implementations are shared between the hybrid renderer and the path tracer, ensuring visually consistent and comparable outcomes.

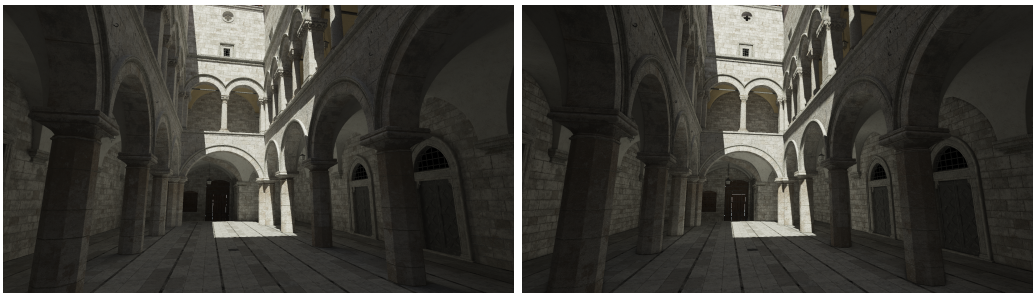


Figure 1.2: A frame generated from the path tracer (left) and the hybrid renderer (right).

We evaluated the hybrid renderer by collecting performance metrics and comparing its final frames to those produced by the path-tracer. A manual visual comparison was performed, and any notable differences or artifacts were documented. In addition, we used the structured similarity index (SSIM) and the peak signal-to-noise ratio (PSNR) to provide objective measures of image fidelity. The visual comparisons between the hybrid renderer and path tracer showed that their results are very similar, as seen in Figure 1.2. This was also confirmed by the SSIM results. Although several artifacts, like light bleeding, were identified and examined in detail, they did not deduct significantly from the overall quality of the hybrid renderer's results. Our renderer can generate a full frame in 12 ms to 20 ms depending on the configuration. The visual result of this hybrid renderer is directly comparable to the path tracer achieving anywhere between 70% and 90% SSIM index depending on the current view. These results can meet the performance requirements of many interactive applications while presenting results comparable to our path tracers.

Thesis Structure

This thesis is divided into six main sections with an additional appendix:

1. **Introduction.**

2. **Background Research.** This chapter introduces the theoretical foundations and techniques used throughout this thesis. First, mathematical concepts such as coordinate spaces, spherical Fibonacci sampling, and octahedral mapping are presented. Next, the definition of the scene and its components, as applied in this thesis, are described. Subsequently, rendering concepts and techniques—including Lambertian shading, deferred rendering, shadow mapping, dynamic diffuse global illumination, and path tracing—are introduced. Finally, post-processing concepts are discussed at a theoretical level, focusing primarily on Gaussian blurring.

3. **Software Architecture and Development.** This chapter covers the low-level aspects of the project, beginning with an overview of the Vulkan API. Key Vulkan concepts such as synchronization, graphics and compute pipelines, and the Vulkan Ray Tracing extension are explained. The chapter concludes with a discussion of the utilities and libraries used, including glTF 2.0 and the *auto-vk-toolkit*.

4. **Implementation.** In this chapter, the implementation details of both the hybrid renderer and the path tracer are presented in detail. The algorithmic aspects of each technique are described, starting with the scene definition and continuing with variance shadow mapping, deferred rendering, screen space ambient occlusion, and, most importantly, dynamic diffuse global illumination. The path tracer implementation is then detailed. The chapter closes with an explanation of the common post-processing steps applied to both renderers. Code excerpts and diagrams are provided throughout to illustrate key techniques.

5. **Analysis.** This chapter outlines the methodology used to evaluate the results of the hybrid renderer. Then it presents a thorough analysis of the outcomes produced by the hybrid renderer.

6. **Conclusion and Future Work.**

A. **Results (Appendix).** All resulting frames from both the path tracer and the hybrid renderer are compiled and presented here for reference.

Chapter 2

Theory

2.1 Mathematical Background

The world of 3 dimensional (3D) rendering and ray tracing offers a rich tapestry of visual realism and immersive experiences, transforming digital concepts into lifelike images. At the heart of this transformation lies a robust framework of mathematical principles. These principles enable us to simulate the behavior of light, shape, and perspective with high fidelity, resulting in stunningly realistic visual representations. In this section, we will go over the mathematical concepts used as part of this thesis.

2.1.1 Triangles

Triangles are defined by 3 points in space called vertices $\mathbf{V_0}, \mathbf{V_1}, \mathbf{V_2}$. Each pair of vertices defines an edge of the triangle. If a triangle's vertices are aligned in a straight line, the triangle is called a degenerate triangle. A nondegenerate triangle defines a plane P. The order in which the triangle's vertices are defined is called the winding order and is used to define which face is considered to be the front face or the back face. Using the right-hand rule the direction which is considered to be front-facing can be found. A triangle also has a normal vector $\vec{n}$ which is a unit vector and is perpendicular to the plane of the triangle. The normal vector is calculated using the cross product of the triangle's two edges $\overrightarrow{V_1 - V_0} \times \overrightarrow{V_2 - V_0}$. A triangle also has a tangent vector $\vec{t}$. This is a vector that is tangent to plane P of a triangle and orthogonal to the normal vector. Also a bitangent or binormal vector $\vec{b}$ can be defined. It is orthogonal to the normal vector and tangent vector. All of the above are visually represented in 2.1.

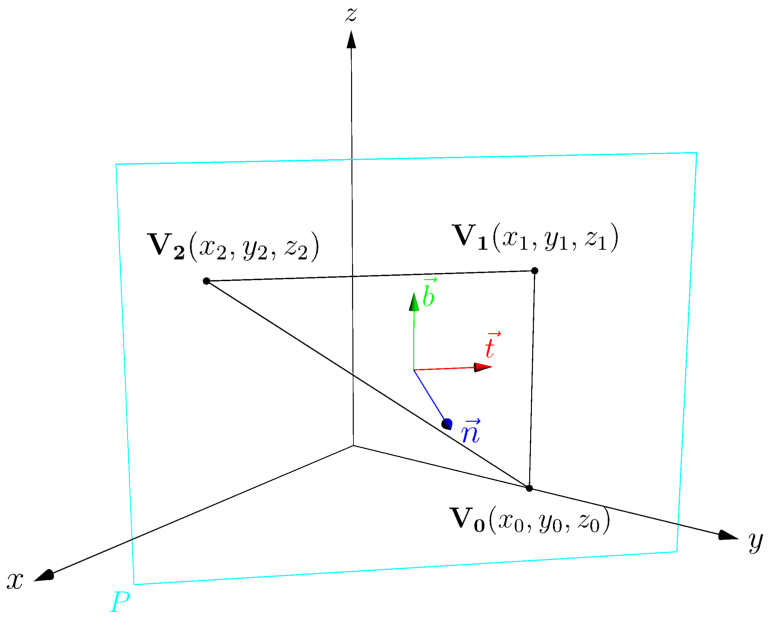


Figure 2.1: A triangle $\triangle \mathbf{V_0 V_1 V_2}$, its plane P and normal $\vec{n}$, tangent $\vec{t}$, and bitangent $\vec{b}$ vectors.

Triangle Tangent Space

In some cases it can be useful to define some data on another basis. This is sometimes used to define vectors relative to a triangle's normal vector. This is done using a change of basis transformation. The basis in this case is defined by the triangle's tangent, bitangent and normal.

$$TBN = \begin{bmatrix} t_x & b_x & n_x \\ t_y & b_y & n_y \\ t_z & b_z & n_z \end{bmatrix}$$

In this thesis, TBN matrices are used in various situations where it is needed to transform between a triangle's local coordinate space and tangent coordinate space. Some examples include normal mapping described in 2.3.3 where normals for each triangle are stored in tangent space. In addition to normal mapping, it is used when performing Ambient Occlusion calculations described in 4.3.5.

Barycentric Coordinates

Barycentric coordinates [AHH18] provide a way to express the position of a point within a triangle as a linear combination of its vertices. These allow us to interpolate between per-vertex variables to calculate the values inside the triangle. They are very important for 3D graphics as they are used in both rasterization and ray tracing.

For a given point P in the triangle $\triangle \mathbf{V_0 V_1 V_2}$ the barycentric coordinates $(\lambda_0, \lambda_1, \lambda_2)$ of P relative to the triangle's vertices are the weights that satisfy:

$$P = \lambda_0 \mathbf{V_0} + \lambda_1 \mathbf{V_1} + \lambda_2 \mathbf{V_2}$$

With $\lambda_0 + \lambda_1 + \lambda_2 = 1$. How these are calculated is beyond the scope of this thesis. Visual representation of barycentric coordinates can be seen in 2.2.

Barycentric coordinates and interpolation are used in this thesis when performing interpolation between a triangle's vertices for generating a point inside the triangle. This is performed automatically by the graphics pipeline described in 2.3.2 and is manually performed when performing path tracing as described in 4.4.

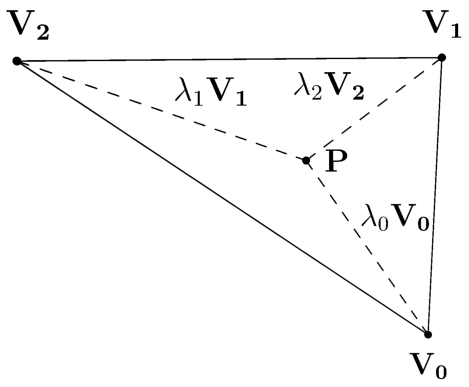


Figure 2.2: A triangle defined by vertices $\mathbf{V_0}, \mathbf{V_1}, \mathbf{V_2}$ with a point $\mathbf{P}$ on the triangle and its barycentric weights λ_0, λ_1, λ_2.

2.1.2 Coordinate Systems

When rendering many objects it is useful to be able to transform data through multiple spaces in order to perform certain calculations efficiently. These transformations are performed using 4x4 matrices. Object geometry is usually defined in **local space**. When an object is authored in a 3D modeling software like for example Blender 3D, all geometry is defined in relation to a predetermined origin point. When placing this object in a scene (Section 2.2), the artist needs to define the object's position, rotation, and scale to best fit the scene they are creating. The object's final position in the scene is called **world space** position and the matrix with which it is transformed is called the **model matrix**. Every object usually has one such matrix associated with it for a specific scene. The scene's active camera (Section 2.2.4) also has a matrix and space associated with it. The **view matrix**, as it is called, transforms every object so that it is oriented and positioned based on the camera's perspective. This space is called **view space**. Next, the graphics APIs expect the final coordinates sent to the GPU to be in **normalized device coordinates** or NDC. These coordinates are normalized from -1.0 to 1.0 on each axis and sometimes or in some axes between 0.0 and 1.0. Transformation between view space and **clip space** is achieved using the **projection matrix**. When rendering, everything that is not contained in the clip space will be discarded. The projection matrix also has another purpose. It skews the rendered geometry according to the projection type as described in 2.2.4. When rendering, we are outputting to either the screen or a framebuffer (described in 2.3.2) that have a set resolution. Thus, the final transformation that takes place transforms geometry to **screen space** using the **viewport matrix**. A full visual representation of the following process can be seen in 2.3.

In this thesis and in graphics programming in general, it is important to perform transformations between various coordinate systems. Many graphics effects, especially shadow mapping as described in 2.3.3, rely on this.

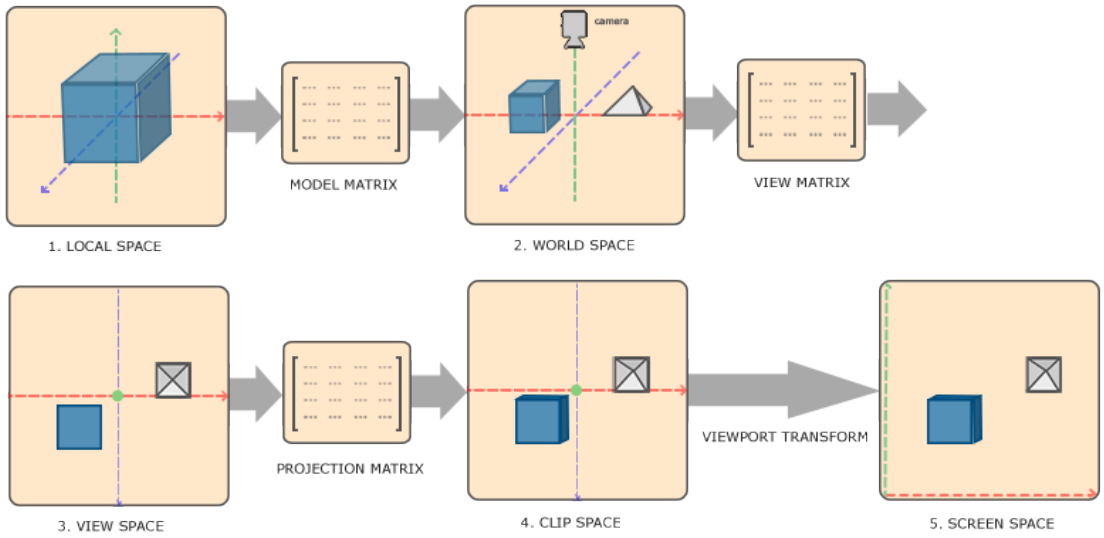


Figure 2.3: The transformation chain between local space to the screen [Vri15].

2.1.3 Spherical Fibonacci

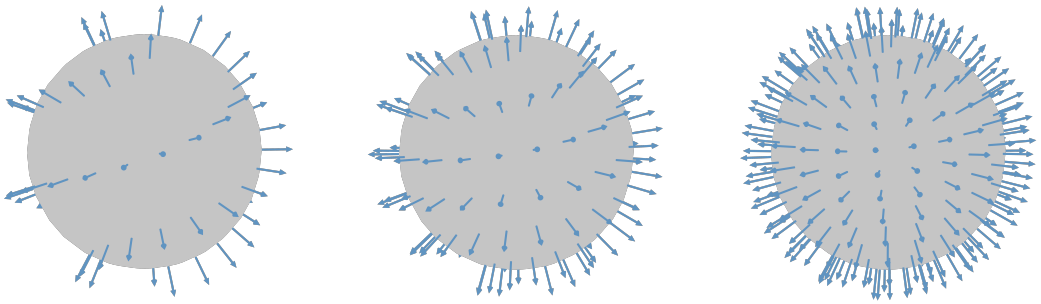
Spherical Fibonacci point sets yield nearly uniform point distribution on the unit sphere and are easy to implement on the graphics processing unit (GPU) as described by Keinert et al. in [Kei+15]. More precisely A point with index i of an **SF** point set with n samples is given as:

$$\mathbf{SF}_i^n = \mathbf{P}(\phi_i, \arccos(z_i)),$$

$$\phi_i = 2\pi \left[\frac{i}{\Phi} \right], z_i = 1 - \frac{2i+1}{n}, i \in \{0, \ldots, n-1\},$$

$$\mathbf{P}(\phi, \theta) = (\cos(\phi)\sin(\theta), \sin(\phi)\sin(\theta), \cos\theta)$$

and $\Phi = \frac{\sqrt{5}+1}{2}$ being the golden ratio. A visual representation of the points on a sphere and their directions in the spherical Fibonacci distribution can be seen in 2.4.



(a) $n = 64$ (b) $n = 128$ (c) $n = 256$

Figure 2.4: Spherical Fibonacci points and directions on a unit sphere for various n values.

In this thesis, when we trace rays to update the DDGI's probe volume as described in 4.3.4 it is important to sample a sphere uniformly. Since the technique uses spheres to sample the scene for irradiance and visibility, sampling the sphere uniformly is important.

2.1.4 Uniform Random Rotations

Random sampling of rotation quaternions in a uniform can be achieved by a method described by Shoemake in [Sho92]. To be precise a random uniform quaternion q can be generated by using the following equation:

$$q = (\sqrt{1-x}\sin(2\pi y), \sqrt{1-x}\cos(2\pi y), \sqrt{x}\sin(2\pi z), \sqrt{x}\cos(2\pi z))$$

where $x, y, z \in [0, 1]$ are uniform random values.

In this thesis, when updating the DDGI's probe volume in every frame, rays are rotated to ensure more uniformity in the probe's data. Thus, unbiased random rotation generation is important as not to bias the result towards a single direction.

2.1.5 Uniform Unit Hemisphere

Obtaining a random direction vector from a hemisphere can be achieved uniformly by using the following equation as described in [PJH16] by Pharr, Jakob, and Humphreys.

$$d = (\cos(2\pi\xi_2)\sqrt{1-\xi_1^2}, \sin(2\pi\xi_2)\sqrt{1-\xi_1^2}, \xi_1)$$

where $\xi_1, \xi_2 \in [0, 1]$ are uniform random values.

In this thesis, and in Monte Carlo path tracing in general, a hemisphere is sampled uniformly, when generating the next ray to be traced from a given surface.

2.1.6 Octahedral Mapping

Octahedral mapping or octahedral normal vectors is the method by which a unit direction vector can be encoded to a 2D square in the range of $[-1, 1]$. This can be used to encode direction information into an image and store it on the GPU. As defined by Meyer et al. [Mey+10] a 3D vector $\vec{n} = (x, y, z)$ can be encoded into (u, v) using the L1 norm $|x|_1 = \sum_{r=1}^{n} |x_r|$ as such.

$$(u, v) = \begin{cases} \left(\frac{x}{|\vec{n}|_1}, \frac{y}{|\vec{n}|_1}\right), & \frac{z}{|\vec{n}|_1} \geq 0 \\ \left(\left(1 - \frac{y}{|\vec{n}|_1}\right)\sigma(x), \left(1 - \frac{x}{|\vec{n}|_1}\right)\sigma(x)\right), & \frac{z}{|\vec{n}|_1} < 0 \end{cases}$$

Where $\sigma(t)$ is the sign function of t and $\sigma(t) = 1$ for $t \geq 0$ and $\sigma(t) = -1$ for $t < 0$. The inverse can be achieved by.

$$(x, y, z) = \begin{cases} (u, v, 1 - |u| - |v|), & 1 - |u| - |v| \geq 0 \\ (\sigma(v) - v, \sigma(u) - u, 1 - |u| - |v|), & 1 - |u| - |v| < 0 \end{cases}$$

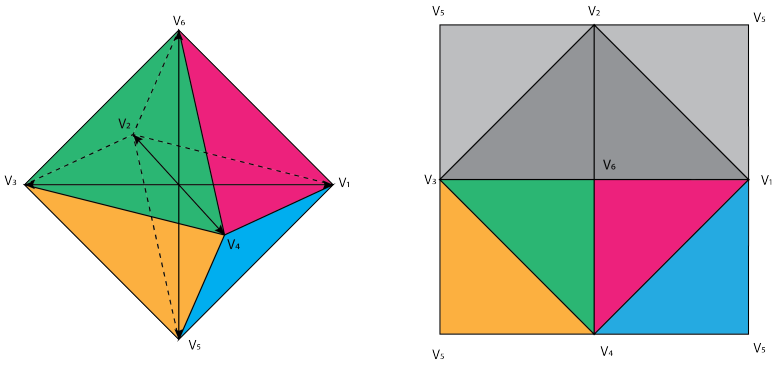


Figure 2.5: Mapping of an octahedron to a 2D square.

In this thesis, when implementing the DDGI technique, octahedral mapping is used to map a sphere into a 2D texture.

2.2 Scene

In graphics, everything starts with the scene. It is the core of every imaging project. For virtual environments, 3D or 2D, we also begin with a scene. A scene is the collection of objects, lights, cameras, and any other element that can then be used to create a visual representation of a virtual environment.

In this thesis, the goal is to render high-quality, ideally photorealistic images that represent a physical scene. Thus, how that scene is defined is an important factor. Each decision about how the physical equivalents of digital objects are modeled in the virtual world impacts the final result. In this section, we will go over the elements that are typical in a scene. The definitions presented will be used from now on as needed in this work.

2.2.1 Geometry

When representing the physical world in the virtual world, the objects are represented by geometric primitives. A collection of geometric primitives composes a mesh. The mesh can be 3D or 2D and defines the surface on which every other effect will be produced. A scene can contain many meshes and the mesh refers only to the geometry and not any other element like textures and materials, which will be discussed later. The underlying geometric primitives can have many forms. To name a few, geometric primitives can be defined by a point, line, quad, n-gon, voxel, and signed distance function. While these are very useful geometric primitives, the most important one for GPU-based graphics is the triangle. Triangles are the foundational shape used in graphics. Many triangles can be used to mimic physical objects, the use of more triangles will result in a more detailed representation of an object as seen in Figure 2.6.

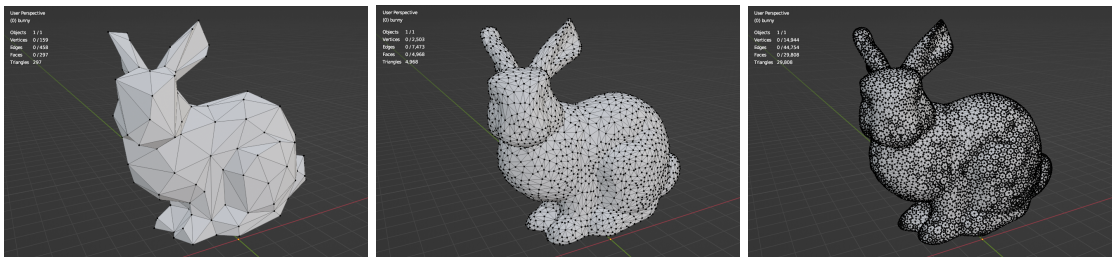


Figure 2.6: Stanford bunny. From left to right the same object represented by a low resolution mesh to a high resolution mesh.

2.2.2 Textures

The more detail in a 3D mesh the more geometry needs to be processed by the GPU. While we can add more geometry with more vertex data for color, transparency etc, the amount of information would eventually be self-defeating. That is why textures are used to add more detail to a mesh. Textures are 2D images, usually square, that are mapped to 3D surfaces using the surface's UV coordinates. UV coordinates are a set of 2D coordinates mapping each triangle's vertices from a 3D space onto a 2D plane. The process by which a mesh's triangles are mapped to 2D plane is called uv unwrapping (Example in Figure 2.7). Textures are used to provide more information for the 3D surface like color, fine detail, transparency, and much more, giving an artist more control of the 3D mesh appearance.

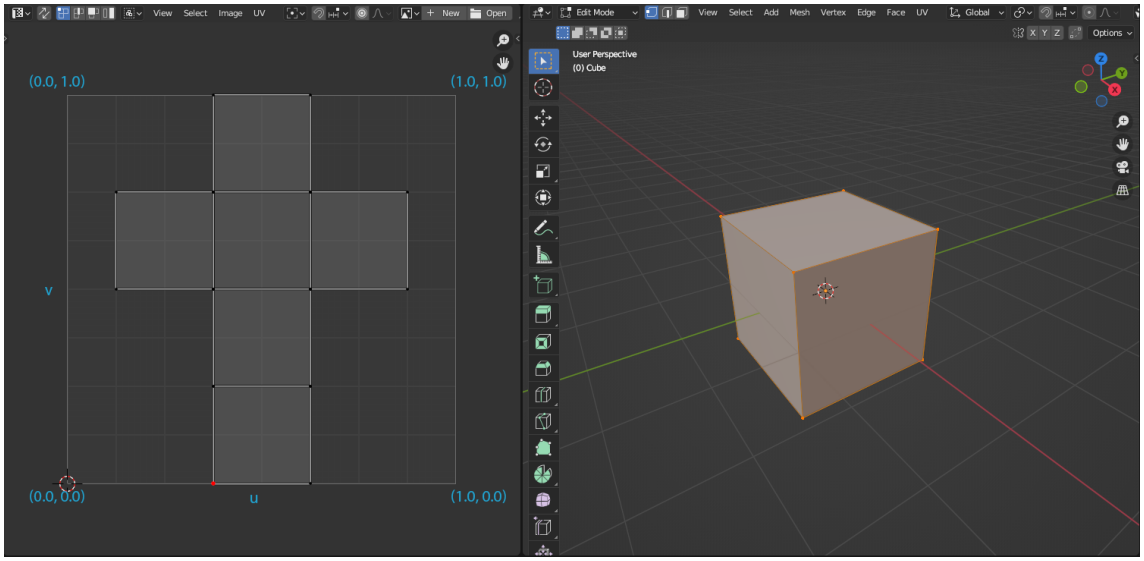


Figure 2.7: A cube in Blender 3D. Right is a 3D representation of the Cube and its vertices. Left are those vertices unwrapped into the UV plane.

2.2.3 Materials

All surface optical properties are defined by a material that contains properties such as color, texture, transparency, and reflectivity. A material defines how a surface appears when rendered. An object's material holds all information that is needed to render it and thus is dependent on the material and shading model used.

2.2.4 Cameras

To be able to view a scene, we must define the observer's point of view. This is defined by a camera, and this camera is simply defined by two matrices: a view matrix and a projection matrix as seen in Section 2.1.2. The view matrix is the inverse of the camera's transformation matrix (translation and rotation). The projection matrix is used to transform the geometry into the device's NDC with the appropriate projection. A camera is also defined by a near and far plane that are used as cutoff points. Only geometry that is between the two planes is rendered. The volume inside the camera field of view is called the **view frustum**. The two most common projection types used are perspective and orthographic projections.

Perspective

Perspective projection is a projection method that represents 3D geometry on a flat surface as seen by the eye. As seen by the observer, objects that are further away from the camera appear smaller whereas objects closer to the center of the line of sight appear taller than those to the edges.

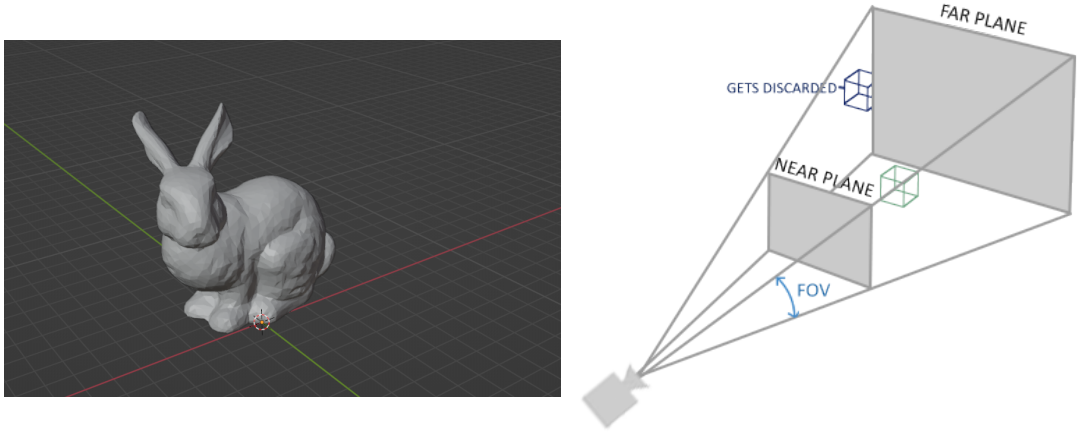


Figure 2.8: On the left, the Stanford bunny drawn by a perspective camera. On the right, a diagram of a perspective view frustum [Vri15].

Orthographic

The orthographic, or sometimes called parallel camera, has a different effect than the perspective camera. It does not skew the objects and preserves proportions and parallelism of lines inside the view frustum. This type of camera is preferred in CAD software where the geometric properties of the object are more important than simulating human vision.

2.2.5 Light Sources

When trying to recreate the real world in digital form, light is one of the most important aspects. In this thesis the main goal is to rendering light realistically. Thus, the way a light source or emitter is defined in a scene is very important. There is a plethora of models that attempt to emulate real-world light sources. All light sources are defined by a color and intensity of light. This can be defined in a physically based manner by a metric like Watts (W) but can also be a unit-less. This is sometimes multiplied by the color, thus producing a combined color-intensity value. In this section, we will go over the most important ones as seen in the literature.

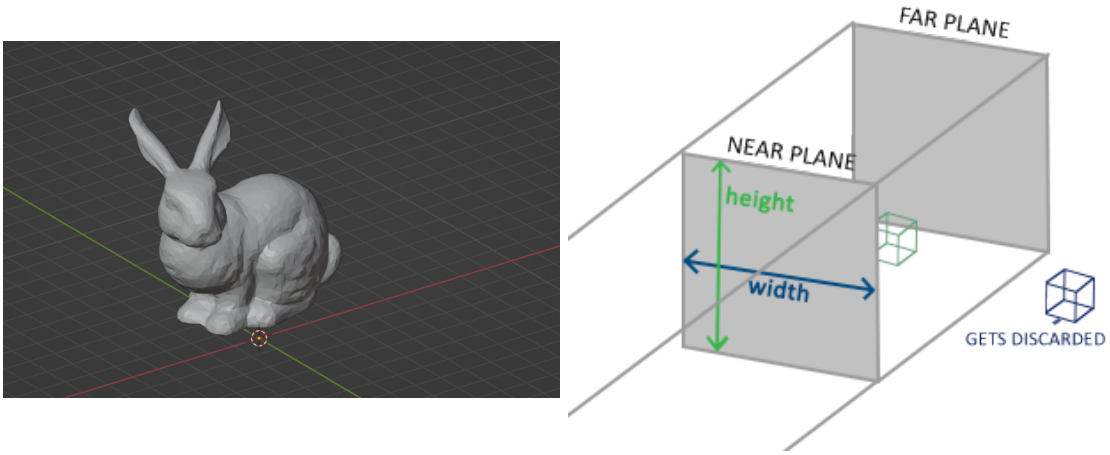


Figure 2.9: On the left, the Stanford bunny drawn using an orthographic camera. On the right, a diagram of an orthographic view frustum [Vri15].

Directional Light

The Directional light or distant light, as it is sometimes referred to, emulates a very distant and powerful light source such as the Sun. All light rays that emanate from the light source are parallel and close together. While in theory a directional light does not have a location, in practice in 3D rendering they are simply placed far away from the scene they are lighting. This light source is defined by the angle or direction of its light rays.

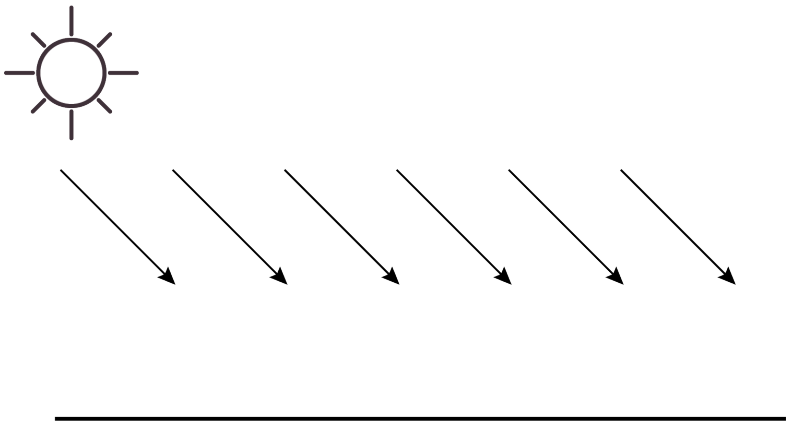


Figure 2.10: Directional light source illustrated by a sun with its light rays shining on a surface.

Point Light

Point lights or omni-directional light sources are light sources that emit light in all directions from their center. Light emitted from a point light has a radius of effect r. Light is strongest closer to the center of the point light. Light is attenuated and as it moves away from the light source its intensity is reduced. Thus surfaces closer to the light sources are brighter to the ones further away. All objects outside the radius r are not lit from this source at all.

Spot Light

Spot lights are light sources that emit light in a cone. This light is usually defined by a position and direction of the cone. There can be two cones, an inner cone with angle θ and an outer cone with angle ϕ. Objects inside the inner cone are lit uniformly. Between the inner and outer cones light begins to fall-off. Outside both cones the spot light does not affect any object.

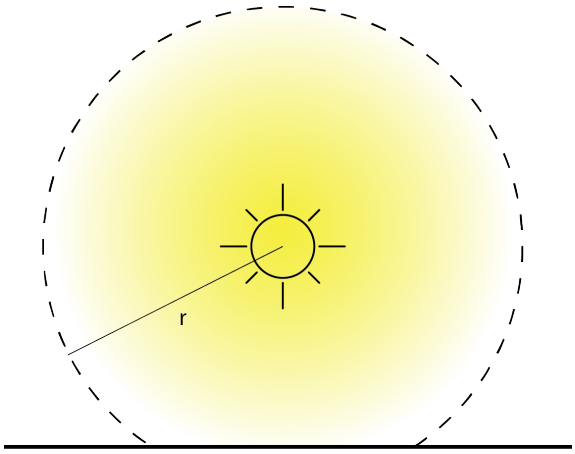


Figure 2.11: A point light of radius r. Light is attenuated as it leaves the source.

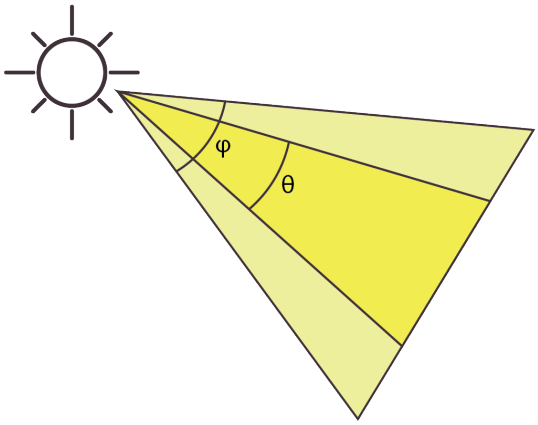


Figure 2.12: A spot light with inner cone of angle ϕ and inner angle of θ.

2.3 Shading

Shading is the process by which a scene is drawn in a visually appealing, physically accurate, or visually distinct way. It aims to simulate light as it travels in a scene and as it interacts with objects. Capturing the color produced by this interaction. There are many aspects and techniques to consider when shading a scene.

2.3.1 Illumination Models

Illumination models or shading models, as they are sometimes referred to, describe the intensity of light that is reflected on a point of a surface. As illumination is one of the most important parts when rendering a scene, these models have been the subject of intense study. Many shading models exist, without one being superior over another. Shading is a subjective result. Some algorithms produce realistic results while others produce artistic results. Some focus on performance over visual quality and others on the opposite. There is no definitive shading model, just many options. As in this thesis the Lambertian model is used, we will quickly go over it here.

Lambertian Shading Model

The Lambertian shading model [Kop14] is a diffuse shading model. It simulates diffuse or matte surfaces or surfaces that reflect light in all directions. For example, an unfinished wood surface could be considered as a diffuse surface. Since it is a diffuse model the apparent brightness of the surface does not change with the observer's point of view. So according to the lambertian shading model the brightness of the reflected color of a diffuse surface is:

$$B_D = \vec{\omega_i}\vec{n}\rho I_l$$

Where ρ is the color, I_l the intensity of the light source at the point of the surface, $\vec{n}$ is the surface normal and $\vec{\omega_i}$ the incident light ray direction.

In this thesis, the Lambertian shading model was used as we focus on generating global illumination for diffuse surfaces and do not aim to produce more complex effects.

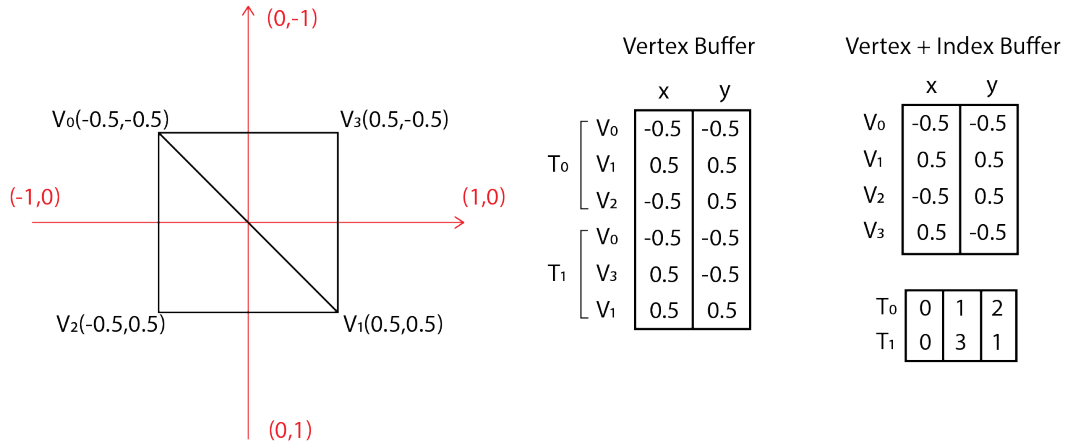


Figure 2.13: An example square defined in screen space with its Vertex buffer and Vertex + Index buffer alternative.

2.3.2 Rasterization Pipeline

One way of approaching rendering a scene is rasterization. This is the process by which we can convert geometric vector-based primitives such as triangles to pixels on the screen. GPUs began as processors for accelerating this process, allowing for real-time rendering of scenes. To achieve real-time performance, many trade-offs, need to be made. Thus techniques for rendering on the GPU for real-time performance usually focus on performing "good enough" approximations of effects like lighting and shadows. With each passing hardware generation, many improvements on the hardware have been made allowing for improved visual quality and have allowed for more and more realistic results.

The GPU accelerates the rasterization of geometric primitives and exposes that functionality in the form of the graphics pipeline. The main purpose of this pipeline is to process geometric primitives and output pixels. The graphics pipeline has various stages as seen in Figure 2.14. More information on each stage is provided in Section 3.1.4. In this section, the vertex buffer, index buffer, vertex shader, rasterization, and fragment shader will be presented.

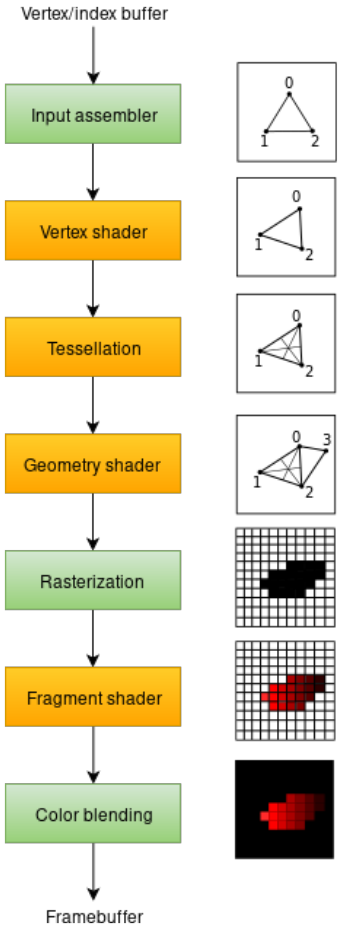


Figure 2.14: Graphics pipeline stage diagram with a small example output of each stage [Ove15].

Vertex/Index buffer

To be able to process primitives, we need to pass them to the GPU. This is done through the Vertex buffer with or without an Index buffer. Let us say for example, that we want to render the square seen in Figure 2.13. This square has 4 vertices $\mathbf{V_0}, \mathbf{V_1}, \mathbf{V_2}, \mathbf{V_3}$. These are defined directly in screen space for this example, but can be defined in any space required by the application (Section 2.1.2). The red coordinate space is the screen coordinate system. This quad, as it is also called, is made up of two triangles. When drawn on the GPU it is broken down to two triangles and drawn.

When rendering, the GPU expects the two triangles' vertices in a buffer (area of memory) to be able to draw from. This buffer is called the **Vertex buffer**. The vertex buffer holds a number of vertices for each primitive. Since we are drawing triangles, the GPU consumes 3 vertices per primitive from the vertex buffer. To draw this quad, we thus need the buffer depicted in Figure 2.13. Since the two triangles share vertices, this can waste a lot of memory if more triangles need to be rendered. Thus, the use of an index buffer is allowed to save on memory. In this case, the vertex buffer holds only unique vertices. The GPU now reads from an **Index buffer** to construct the primitive. The index buffer holds the indices to the vertex buffer and can repeat an index many times depending on the object drawn. An example on the quad can be seen in Figure 2.13.

Vertex shader

Shaders are programs that run on the GPU. The Vertex shader is a shader that runs for every vertex for the primitive that is being drawn. It is used to position the vertices from which a primitive will be drawn in the correct spot. To achieve this, the transformations mentioned in Section 2.1.2 are applied. Apart from calculating the final position of the vertex, the user can also store variables in the vertex shader for usage at later stages of the graphics pipeline.

Rasterizer

The rasterizer converts the primitives produced in the vertex shader into pixels. It is a hardware stage of the graphics pipeline that cannot be fully programmed by the user. Despite that, some options are exposed to the user. One such option is the ability to depth test primitives. Depth

testing allows the usage of a depth buffer to discard non-visible primitives. Geometry that is outside the view frustum can also be discarded.

Fragment shader

The fragment shader is a program that is run for every fragment or pixel that is generated. This shader is used to calculate each pixel's color and perform shading. This shader is sometimes called a pixel shader. We can use information passed from the previous shader stages to perform more complicated lighting calculations. For example, we can use the fragment shader to perform lighting calculations as described in 2.3. Per-vertex values stored in the vertex shader to be passed are interpolated with using the current fragments barycentric coordinates and optionally can choose not to be interpolated.

Framebuffer

After the Fragment shader processes and colors all pixels, the result is stored in something known as a framebuffer. A framebuffer is a collection of images, also called textures, or when attached to a framebuffer, framebuffer attachments. Each attachment has the same resolution but can have a different format (for example RGB, RG, etc).

Depth testing

When rendering to a framebuffer, a special texture attachment called a depth buffer can be used. For each pixel that is being rendered the Z value in camera space (see 2.1.2) is used to determine if the pixel is visible or is hidden by other geometry. The up-to-date Z value for a given pixel is then stored in normalized 0.0 to 1.0 range to the depth buffer for later use. In depth testing, a fragment's z value is compared to the value in the depth buffer. If there is a fragment with a value closer to the value of the fragment that is being rendered, the current fragment is discarded. Which value is considered close and which far is set by the user. The range of 0.0 and 1.0 can also be interpreted as -1.0 and 1.0.

Forward rendering

The straight forward way to render a scene is to have a Vertex and Index buffer for each object and draw each object with a single draw call to the screen. This is called Forward rendering. In cases where the Fragment shader is expensive, we can reduce Fragments that are being shaded by reducing overdraw. Overdraw is when the multiple objects are drawn one in front of the other. In the case of Forward rendering this will call the fragment shader for each pixel of each object and then discard the ones that are not visible using the depth buffer. However, this can be wasteful.

Deferred rendering

An alternative to Forward rendering is Deferred rendering [Tha11]. This is a technique in which all information required for lighting calculations for an entire frame are written into a large buffer called the G-Buffer. Using said G-Buffer the final shading can be performed for each pixel and output to the final frame. This has the distinct advantage that lighting calculations are performed once for each pixel on the screen. Thus, lighting complexity scales only with resolution.

In summary, the concepts presented here are used in the hybrid renderer that is implemented. The rasterization pipeline is part of almost all real-time applications, and its characteristics are important when implementing a renderer. The most important concept here is deferred rendering and the G-Buffer.

2.3.3 Lighting

Lighting a scene is one of the most important aspects in rendering a scene in a visually appealing manner. However, rasterization as a rendering method cannot be used to produce physically accurate lighting, especially indirect lighting on its own. Some techniques do exist that approximate lighting effects, such as shadows, direct illumination, and indirect illumination. When using the rasterization pipeline for rendering, these effects need to be emulated in various ways. In this section, we will go over some techniques that approximate lighting effects.

Direct Illumination

Direct illumination [AHH18] describes the way light from a light source 2.2.5 hits an object. This effect is relatively simple to produce. Using information from the vertex shader like the fragment's position, normal, and object color as well as the scene's list of lights we can calculate the light hitting the fragment and shade it. The fragment can be shaded according to the lighting model that fits the intended final result.

One way to add more detail to illumination is to use diffuse color textures and normal maps. A triangle has only 3 vertices. For these vertices color and normal values can be defined. This way to increase detail on the mesh, means that many more triangles would need to be rendered. This can lead to performance issues. Thus textures for diffuse color and normals are used. Color values are read using the fragments (u, v) coordinates 2.2.2 and are read from the textures. Normal values are read in the same way from the normal texture, otherwise called a **normal map**, and are transformed to the triangle's local space using 2.1.1. This transformation may or may not need to be applied if the normal map is in tangent space. The addition of the normal map can greatly increase perceived detail as it adds perceived dents and bumps to surfaces that may be visually appealing without requiring extra performance. More textures for glossy and other aspects for more complicated illumination models can be used.

The main difficulty with calculating direct illumination is determining whether the fragment that is being drawn is visible or not by the light source.



Figure 2.15: A single quad rendered with a brick texture. Lit from above. Left has only a diffuse texture. Right has diffuse and normal texture.

Shadows

While calculating direct illumination on a fragment can be easily determined, whether a light ray from a fragment can reach a light source is not. When a fragment is not reachable from any light source it is in shadow. Only ambient and indirect light hits it. This effect can be difficult to produce without casting rays 2.4 to check for visibility of a fragment.

One solution formulated early on is **Shadow Mapping** [AHH18]. In this family of techniques, the scene is rendered from the light source's point of view. For this draw, only the z values of the geometry visible to the light source are kept. The resulting depth buffer from this draw is called a shadow map. Thus the shadow map encodes the depth values from the point of view of all geometry that is visible to the light source. All geometry that is not in the shadow map should be in shadow from that light source's point of view. When rendering we can convert a fragment's position to the space of the light's shadow map and check whether the fragment is visible from the light source or not.

Although shadow mapping is a widely used technique, its main drawback is resolution. The resolution of the shadow map causes aliasing artifacts in the shadows drawn on the screen. Simply increasing the resolution can be impractical and cause performance issues. While other techniques have been proposed to solve this problem the main one we will present here is **Variance Shadow Mapping** [DL06]. In this technique, on the shadow map the z and z^2 values are stored in the shadow map. Using z, z^2 as mean μ and standard variance σ^2 respectively, the chebyshev inequality

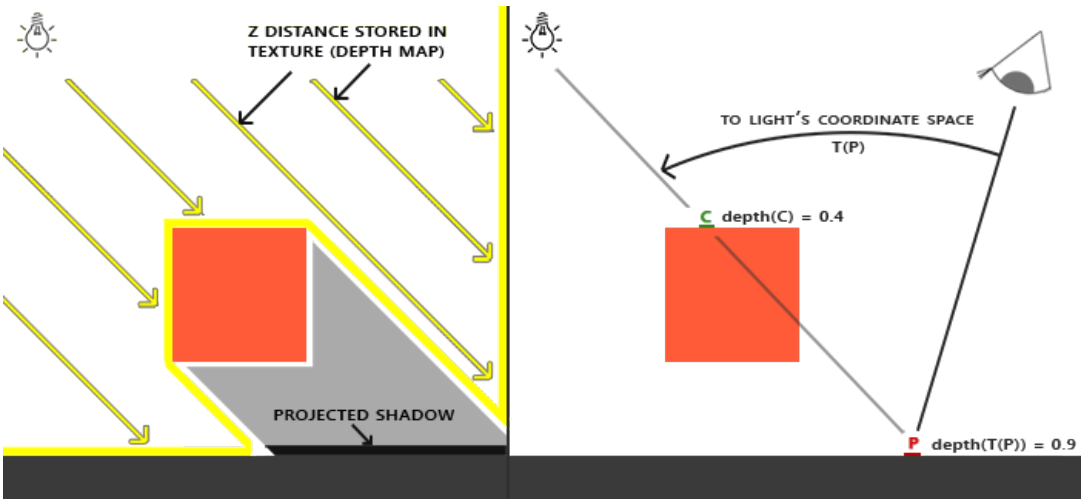


Figure 2.16: Diagram explaining shadow mapping. The left side shows how light rays are emulated when drawing the scene from the light source's point of view. The right side shows how the fragment P's visibility can be sampled by transforming its position to the light source's space. [Vri15]

can be used to check if a fragment with z_f distance is in the space of light source's.

$$\mu \leq \frac{\sigma^2}{\sigma^2 + (z_f - \mu)^2} = p_{max} \tag{2.1}$$

If the current fragment has $z_f < \mu$ then it is in shadow. Otherwise, scale the light intensity by p_{max}. This technique has the advantage that of produces a more smooth shadow effect when combined with Gaussian blur on the variance shadow map.

Global Illumination

Global illumination or indirect illumination [AHH18] is light that bounces inside the scene and reaches a surface. Adding indirect illumination to a render adds a lot of realism to a scene. Many techniques exist to calculate global illumination. Many techniques focus on offline computation of light data and storing them to be used in real-time. Techniques such as surface prelight described in [AHH18] calculate lighting information for a scene statically. The result can then be stored and used when running the scene in, for example, a game. These algorithms produce fast end results, but are more restrictive to the developer. Scene lighting conditions need to be calculated when authoring the scene and cannot change easily. They also add complexity when making the scene because they require baking. Baking is the lighting recomputation step that is required before the user can interact with the scene. Algorithms that calculate indirect illumination in real-time are called dynamic and have advantages over precomputations for the above reasons. These algorithms, however, can be expensive and demanding on the user's hardware. Thus, fast algorithms and techniques are valuable.

Hybrid approaches are techniques that rely on GPU hardware-accelerated ray tracing to produce GI in real time. These methods represent the cutting edge of real-time computer graphics research. Despite hardware requirements, some industry players are ready to embrace them.

Dynamic diffuse global illumination (DDGI) is a hybrid rendering technique, proposed by Majercik et al. [Maj+19] and further in [Maj+21], that extends the basic light field probe presented in [McG+17]. It efficiently encodes the irradiance, calculated with hardware path tracing, and the visibility of the geometry in relation to the probe. Based on this, a visibility-aware moment-based interpolation irradiance query can be performed to produce excellent GI in a scene. This is achieved without light leaking, noise, or requiring meticulous probe placement, while using the existing irradiance robe code path of a game engine.

Ambient Occlusion

Ambient occlusion is a class of indirect lighting algorithm. It aims to approximate the darkening effect from creases, holes and surfaces close to each other. Areas like that are occluded from nearby geometry and thus fewer light rays are able to reach them. In practice, ambient occlusion algorithms are used to supplement global illumination algorithms. Some global illumination algorithms are

not able to simulate the high frequency detail produced by ambient occlusion techniques. Thus they are used to simulate that aspect of indirect light.



Figure 2.17: A quick demonstration on the effect ambient occlusion can have on a scene. [Vri15].

Screen Space Ambient Occlusion is one such technique [Mit07]. This technique works in screen space of G-Buffer. N random samples in a hemisphere around the fragment are picked. These samples remain constant on each frame to reduce flickering. The depth value is read for each sample. If the rendered depth value of the sample is less than the sample's depth it is considered occluded. Otherwise, it is not. All occluded samples are summed up with a factor 1 and normalized by the count of samples. This produces an ambient occlusion factor. Many small processing differences exist between implementations. In some of the samples, the z-value is biased by a small amount. Most blur the result to achieve smoother results without as many samples. This technique is very easy to implement and scales only with screen resolution.

In summary, as lighting is a very important aspect in rendering scenes, there are many ways to generating its various components. In this thesis, a small subset of lighting techniques are used for our hybrid renderer, and their function is important for understanding the final renderer. The main technique is the global illumination technique, Dynamic Diffuse Global Illumination, around which the entire thesis is built.

2.4 Ray Casting

Ray casting is the a class of techniques for rendering images and determining object visibility. Algorithms based on ray casting work differently from rasterization. Instead of focusing on putting geometry on screen, they focus on tracing rays from a specific point in the scene and finding intersection with geometry. The way rays interact with the scene is by intersecting with geometry. To render images with this technique the radiance transported along their path can be calculated. A lot of research has taken place over decades on this class of techniques. Many renderers use these techniques for rendering especially in offline renderers and photorealistic rendering applications. Ray casting can be used to solve the rendering equation 2.3 and thus has a lot of potential for producing photorealistic results.

2.4.1 Acceleration Structures

Since ray casting algorithms work by mainly casting rays onto a scene and determining where they hit, ways to accelerate this process are needed to allow for better performance and more efficient rendering. Enter acceleration structures. They are a category of data structure used to accelerate ray-scene intersection. They are designed around efficiently managing and organizing geometric data and partitioning the scene in ways that allow the least number of intersection tests to be performed to determine an intersection. Fast and efficient implementations of acceleration structures are indispensable for real-time and offline rendering software.

The concept of accelerator structures has been explored since the beginning of computer graphics research. The high cost of rendering, along with the slow hardware at the time, drove research

towards techniques to accelerate rendering. Early methods such as uniform grids provided an initial approach by dividing the scene into a grid where each cell contained a list of objects. These methods were soon found to be inefficient for scenes without a uniform distribution of geometry and large empty spaces. The introduction of techniques like Bounding Volume Hierarchies (BVH) as described in [RW80], KD-Trees as described in [Ben75], and Octrees since at least [Sam89] marked a leap in the field. These structures leverage hierarchical and spatial partitioning techniques to optimize the search process for the intersection of rays with objects. Finally, the latest GPU hardware integrates hardware accelerated structures for ray-object intersections. These are based on proprietary BVHs that are very fast [SFG20] especially in the case of ray-triangle traversal.

These advancements in the field have allowed for many exciting new developments, including faster, real-time ray-traced images, and hybrid approaches as seen in this thesis.

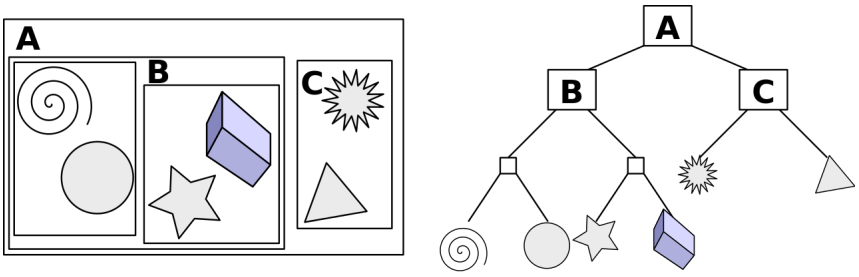


Figure 2.18: A diagram demonstrating bounding volume hierarchies (BVH). The scene on the left with its objects and bounding boxes A, B, and C is described and traversed as the tree seen on the right [Sch11].

Although acceleration structures are not directly studied in this thesis, understanding the concept is important as GPU accelerated ray tracing uses acceleration structures to perform ray tracing efficiently. An acceleration structure is part of the Vulkan extension for ray tracing and thus one is created and used by the hybrid renderer and path tracer implementation.

2.4.2 Ray Tracing

Ray tracing, as introduced by Whitted in [Whi80], is a ray casting technique that is used to create highly realistic images by simulating the behavior of light. It traces the path of the rays in the scene as it travels and interacts with objects. It can accurately capture effects like reflection, refraction, shadows, and global illumination. This technique can be used to solve the rendering equation 2.3 and produce physically accurate results.

The basic idea behind it is that for each pixel in the image, rays, called primary rays, are traced into the scene. These rays are intersected with the scene's geometry, and the closest intersection point is found. The color of the intersection point is calculated based on the surface properties such as color, texture, and material. Apart from surface information lighting information must be factored into shading. To calculate direct lighting, shadow rays are cast towards the light sources to determine visibility, or if the point is in shadow. More rays are traced from the point, called secondary rays, recursively to gather indirect lighting information. This process can continue until the termination criteria are met. This can be a max recursion depth or it can be based on the contribution by the ray to the final result. Other types of rays can also be traced depending on the surface. Reflection rays are traced for reflective surfaces to calculate mirror reflections. Refraction rays are traced for refractive surfaces and are bent according to Snell's law.

2.4.3 Path Tracing

Path tracing, as introduced by Kajiya in [Kaj86], is a rendering algorithm used in computer graphics to simulate realistic lighting by tracing the paths of light as they interact with surfaces in a scene. It is an extension of ray tracing, focusing on more accurately representing the behavior of light, including effects like global illumination, caustics, and soft shadows. The theoretical foundation of path tracing is rooted in physics and mathematical principles, particularly the rendering equation 2.3, Monte Carlo integration, and stochastic sampling techniques.

Rendering equation

The rendering equation is a recursive integral equation for calculating the radiance leaving a point x. This equation is the basis for most realistic rendering techniques. Algorithms like path tracing

and radiosity aim to solve this equation and produce realistic images.

$$L_o(\mathbf{x}, \vec{\omega_o}, \lambda, t) = L_e(\mathbf{x}, \vec{\omega_o}, \lambda, t) + L_r(\mathbf{x}, \vec{\omega_o}, \lambda, t) \tag{2.2}$$

$$L_r(\mathbf{x}, \vec{\omega_o}, \lambda, t) = \int_\Omega f_r(\mathbf{x}, \vec{\omega_i}, \vec{\omega_o}, \lambda, t) L_i(\mathbf{x}, \vec{\omega_i}, \lambda, t)(\vec{\omega_i} \cdot \vec{n})\, d\vec{\omega_i} \tag{2.3}$$

Where:

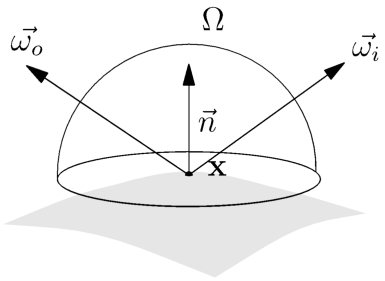


Figure 2.19: Visualization of the rendering equation for point $\mathbf{x}$ and incidence $\vec{\omega_i}$ and outgoing direction $\vec{\omega_o}$.

- $\mathbf{x}$ is the location in space.
- $\vec{\omega_o}$ is the direction of the outgoing light.
- $\vec{\omega_i}$ is the negative direction of the incoming light.
- λ is a particular wavelength of light.
- t is time.
- $\vec{n}$ is the surface normal.
- Ω is the unit hemisphere centered around $\vec{n}$ containing all possible values of $\vec{\omega_i}$.
- $L_o(\mathbf{x}, \vec{\omega_o}, \lambda, t)$ is the total radiance for position $\mathbf{x}$ along direction $\vec{\omega_o}$ of wavelength λ at time t.
- $L_e(\mathbf{x}, \vec{\omega_o}, \lambda, t)$ emitted radiance.
- $L_r(\mathbf{x}, \vec{\omega_o}, \lambda, t)$ reflected radiance.
- $\int_\Omega \dots d\vec{\omega_i}$ is an integral over all possible incidence directions.
- $L_i(\mathbf{x}, \vec{\omega_i}, \lambda, t)$ is the radiance of wavelength λ coming toward $\mathbf{x}$ from the direction $\vec{\omega_i}$ at time t. This is the recursive part of the equation.
- $\vec{\omega_i} \cdot \vec{n}$ is the weakening factor of outward irradiance due to incident angle. This is often written as $\cos(\theta_i)$.
- $f_r(\mathbf{x}, \vec{\omega_i}, \vec{\omega_o}, \lambda, t)$ is the bidirectional reflectance distribution function, the proportion of light reflected from $\vec{\omega_i}$ to $\vec{\omega_o}$.

The bidirectional reflectance distribution function or BRDF is a very important component of the rendering equation. It defines how light from a source is reflected off a surface. The visual result of a solution of the rendering equation is highly dependent on the BRDF and many such functions exist, producing various results.

This algorithm produces realistic images by solving the rendering equation. It solved the integral of the rendering equation using Monte Carlo integration with importance sampling. This way, the rendering equation is formulated as such.

$$L_o(\mathbf{x}, \vec{\omega_o}, \lambda, t) = L_d(\mathbf{x}, \vec{\omega_o}, \lambda, t) + \sum^{N} \frac{f_r(\mathbf{x}, \vec{\omega_i}, \vec{\omega_o}, \lambda, t)(\vec{\omega_i} \cdot \vec{n})}{p(\vec{\omega_i})} L_i(\mathbf{x}, \vec{\omega_i}, \lambda, t) \tag{2.4}$$

Since the integral is being estimated by Monte Carlo integration it converted to a sum. The term $p(\vec{\omega_i})$ is the probability density of sampling direction ω_i. This way the integral can be aproximated resulting in an image.

In this thesis, the rendering equation and Monte Carlo path tracing are the basis for the implementation of the reference path tracer. A careful study of the BRDF and its use in performing path tracing was also important in order to get visually comparable results from both the hybrid renderer and path tracer. Both renderers should use and interpret the materials in the same way, so that their results could be compared.

2.5 Post-Processing

Post-processing is a set of techniques applied to images to improve visual quality and produce certain desired effects. It is widely used in 3D rendering to produce effects to emulate reality more closely or to produce effects for artistic effect.

2.5.1 Anti-aliasing

Anti-aliasing are a group of techniques used to eliminate the visual artifact of aliasing or "jaggies" [AHH18]. When trying to map any shape to a grip of pixels, as is done in 3D graphics, some edges are always not aligned with the grid, and since a pixel cannot contain multiple values, the resulting image is jagged. See an example of aliasing in 2.20.

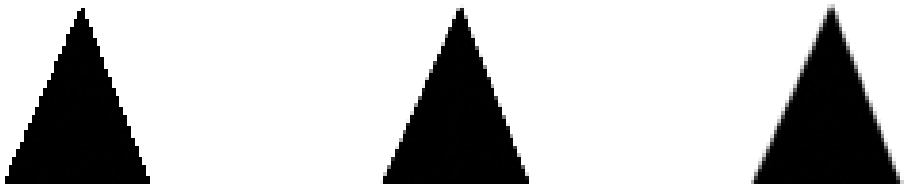


Figure 2.20: Three triangle edges being rasterized with various anti-aliasing filtering settings applied. Left 0 pixels, middle .75 pixels, right 1.5 pixels.

2.5.2 Gaussian Blur

Gaussian blur is the image processing effect that results when an image is filtered by a Gaussian kernel [AHH18]. This processing effect results in a blurred image, an example seen in 2.21. Gaussian kernel is generated using the following function.

$$G(x) = e^{-\frac{x^2}{2\sigma^2}} \tag{2.5}$$

where x is the distance from the center of the kernel and σ is the standard deviation. This being the un-normalized version of the Gaussian kernel. For a two-dimensional kernel, the function is transformed to:

$$G(x, y) = e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{2.6}$$

where y is the vertical distance from the center of the kernel. As seen in 2.21 the various sizes of the kernel change the degree to which the image is blurred. In signal processing, the kernel is square. The radius is denoted by N. Also, when performing image processing kernels are typically normalized.

In GPU programming, the Gaussian kernel also presents an important attribute. It is separable which means that calculations can be split into two steps [Nik23]. This is the result of 2.6 since it can be transformed as such.

$$\begin{aligned} G(x, y) &= e^{-\frac{x^2+y^2}{2\sigma^2}} \\ &= e^{-\frac{x^2}{2\sigma^2}} e^{-\frac{y^2}{2\sigma^2}} \\ &= G(x)G(y) \end{aligned}$$

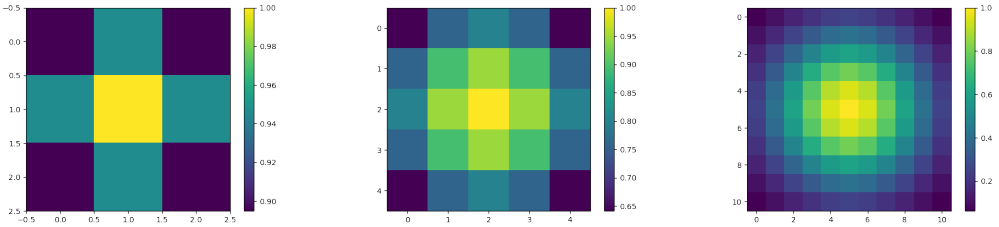
Sampling an image to apply a kernel of $(2N+1) \times (2N+1)$ dimension square with around the pixel in the following ranges $[-N \dots N] \times [-N \dots N]$ requires $(2N+1)^2$ texture sample operations. Due to the fact that the Gaussian kernel is separable applying the filter can be broken into two steps. A horizontal blur step around $[-N \dots N]$ taking $2N+1$ sample operations and a vertical step of similar cost totaling $4N+2$ samples. Model GPU hardware allows neighboring pixels to be sampled and linearly interpolated in a single texture sample operation. This is called linear



(a) Blurred image. $N = 3$, $\sigma = 3$

(b) Blurred image. $N = 5$, $\sigma = 3$

(c) Blurred image. $N = 11$, $\sigma = 3$



(d) Gaussian Kernel. $N = 3$, $\sigma = 3$

(e) Gaussian Kernel. $N = 5$, $\sigma = 3$

(f) Gaussian Kernel. $N = 11$, $\sigma = 3$

Figure 2.21: A set of blurred images and the kernels that were used to blur them.

sampling. This can be used to further optimize blurring by allowing for the following conversion to take place.

$$w0p_i + w1p_{i+1} = (w0 + w1)\text{lerp}(p_i, p_{i+1}, \frac{w1}{w0 + w1}) \tag{2.7}$$

Using 2.7 the number of samples needed can be reduced down to 2N + 2 for the entire blur. These optimizations are very important for blurring large images on the GPU. Sampling a texture has a cost in GPU time and the fewer needed, the better. This is especially the case for large kernels that are performed on large images, for example a full screen texture.

In this thesis, many techniques that were implemented required the use of blur filter. A Gaussian is used because it is a common one and because it is separable allowing for fewer texture samples and increased performance.

2.5.3 High Dynamic Range

When capturing a scene with 3D rendering or a camera in real life, lighting is one of the most important aspects to achieve realism or any other artistic effect. High Dynamic Range or HDR allows us to capture a scene with a wide range of brightness levels while also retaining detail on all those levels. In the context of 3D graphics and GPU programming rendering used to be performed in R8G8B8A8 framebuffers when GPU hardware and memory were limited. This only allowed for 8 bits per channel for the final image resulting in less accuracy for floating point values used to store lighting information. In newer hardware, more formats like R16G16B16A16 and R32G32B32A32 were introduced, allowing for more dynamic range when rendering.

2.5.4 Tone Mapping

While rendering can be done in HDR, most consumer monitors are standard dynamic range or SDR and thus the result must be converted back to a format that can be displayed in monitors. This is done using the process of tone mapping. A common and simple tone mapping method is Reinhard tone mapping [Rei+02] where:

$$V_{out} = \frac{V_{in}}{V_{in} + 1} \tag{2.8}$$

2.5.5 Gamma correction

Ever since the introduction of CRT displays most display devices exhibit power law relationship between input and radiance emitted. While most calculations when rendering are linear when outputting to a display an inverse power is applied called gamma correction. This relationship also

matched the inverse light sensitivity of the human eye, which makes adjustments to the brightness more intuitive.

$$y = x^{\frac{1}{\gamma}} \tag{2.9}$$

The standard gamma value for personal computer displays is defined as a color-space transformation called sRGB.

$$y = \begin{cases} 1.055x^{\frac{1}{2.4}} - 0.055, & where\ x > 0.0031308, \\ 12.92x, & where\ x \leq 0.0031308, \end{cases} \tag{2.10}$$

This transformation is provided by GPU hardware and is handled automatically with sRGB texture formats such as R8G8B8A8SRGB.

Chapter 3

Software Architecture and Development

3.1 Vulkan

With the introduction of GPUs also came the introduction of graphics APIs. Such an API is designed to allow software developers to utilize GPU hardware capabilities in their software. Until a couple of years ago the main graphics APIs were OpenGL a cross-platform open source API developed by the Khronos Group and DirectX a Windows specific closed source graphics API developed by Microsoft. OpenGL and DirectX were introduced in 1992 and 1995, respectively, and have played a pivotal role in graphics development and making millions of games and software applications possible. While these APIs were going strong, GPU hardware has been evolving in the background. Initially, they were only used to render graphics. Over the years, the demands for graphics applications have increased. Additional capabilities such as general purpose computing was introduced on GPU hardware. Software design has evolved to use the multiple CPU cores available in modern processors. Also, the high-level nature of these graphics APIs with the need to high performance put a lot of emphasis on driver developers to optimize the GPU driver over software side optimization on the software developer side. Thus, over the past decades DirectX and OpenGL have been trying to add features while keeping compatibility with previous versions while trying to evolve. These conditions eventually created the need for new and modern graphics APIs.

Enter Vulkan and the new generation graphics APIs like DirectX 12 and Metal. They are the next step in communication with graphics hardware. They are more verbose and low-level than previous generation of APIs. Making the API more low-level allows the software developer to more closely control the graphics hardware and achieve higher performance. It also makes the driver lighter, adding less CPU overhead. The new APIs add the ability to use GPU hardware in a way that also allows for more multi-threaded software to use hardware in a simpler and easier way.

For our case, Vulkan is an open source, low-level, cross-platform 3D graphics and GPU compute API. It is written in the C programming language, developed by the Khronos Group and first released in 2016 [Gro24]. An important feature of Vulkan is the multitude of extensions to the core specification. These extensions are developed by software and hardware vendors and added to the Vulkan specification in a way similar to the OpenGL extensions. These extensions add support for various features not present in the core API itself, such as ray tracing 3.1.6. For the above reasons and due to its open nature and support for ray tracing Vulkan was used for this thesis.

3.1.1 Overview

Let us go over the basics of Vulkan as described in the Vulkan specification [Gro24]. All Vulkan objects are given to the user as a handle or an unsigned 64-bit integer identifier for that object. Vulkan holds as little global state and as little per-application state as possible. The state held is stored in an instance object called **VkInstance**. This is the main object with which the user communicates with the API. It is used in the creation of most other Vulkan objects. It also encapsulates most application-specific extensions and layers.

Vulkan allows the user to communicate and query one or more GPUs at the same time. A GPU is referred to as a device and is represented by two objects. First, the physical device object called **VkPhysicalDevice**. This represents the physical hardware of the GPU as made available by Vulkan. There is also the logical device or **VkDevice** object. This represents an instance of a physical device with its own state and resources independent of other logical devices. The device object is where all application-specific work will run like creating images etc.

Finally, Vulkan also exposes another foundational object, the surface or **VkSurfaceKHR**. This object is a representation of a window or any other surface on which Vulkan can render images. This is OS specific and is not required by the specification as Vulkan can run on hardware that is only designed for computation. This feature is provided as a Vulkan extension.

GPUs provide some programmable operations for various stages. The way we program these operations is through programs called shaders. These shaders are defined using the GLSL language, a C-like language that is then compiled in SPIR-V bytecode format. GLSL and shaders, in general, are very limited in what they can do. They are designed mainly for mathematical operations on data read from buffers and images. However, they are essential for producing visually appealing images and performing computations. There are various types of shaders, such as compute, vertex, and fragment depending on the context that the shader will be running in. Each type has some differences from the other. More will be discussed about the differences in later sections.

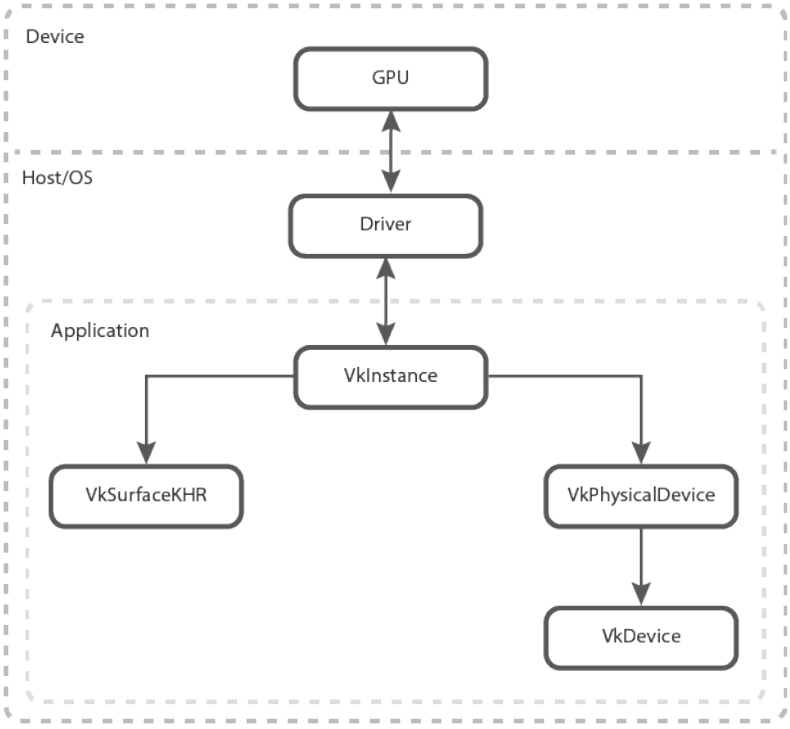


Figure 3.1: Vulkan's foundational objects in relation to GPU hardware or device and the host or operating system.

3.1.2 Execution Model

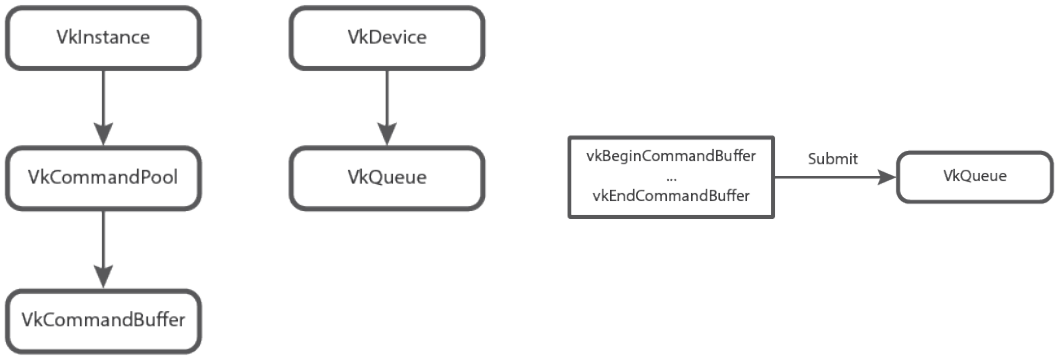
Vulkan's execution model is multi-threading friendly and represents the GPU hardware in a more accurate way. All work to be sent to the GPU is recorded in a command buffer or **VkCommandBuffer**. For example, draw calls, resource bindings, and compute operations. Command buffers are allocated from a **VkCommandPool**. The command pool is used to manage the memory that is used to store command buffers. A command buffer can have multiple commands recorded into it.

Command buffers are submitted to a **VkQueue**. This represents an execution queue on the GPU used to run commands. The submitted command buffers are not run in sequential order. There are some ordering guarantees [Gro24] the full extent of which is beyond the scope of this thesis. Commands start running in the order that they were submitted to the queue. Execution of each command is not guaranteed to complete in any kind of order. As a general rule, the user must synchronize the execution of the command buffers. Queues are created for each device and are also described by a queue family. This basically describes what kind of work can be executed on a specific queue. Such work can be graphics, compute, transfer, video work, etc. Devices may or may not support some queue families and usually provide more than one queue. The queue families provided by the device indicate the work that that device is able to perform.

Since command execution is synchronized by the user, it allows multiple threads to record and submit command buffers to a single queue without synchronization on the host side. This is one of Vulkan's great strengths compared to previous generation graphics APIs. Applications used to be CPU bound since only one thread was allowed to send commands to the GPU. Also, the model of recording commands to a command buffer is more efficient.

3.1.3 Resources

A very big part of the Vulkan API and any other graphics API are the resources used for rendering and computations. Starting with the **VkBuffer** object. This object represents a linear array of data. A buffer can be used for various purposes such as vertex data, index data and used in graphics and compute pipelines. Buffers can be allocated in various memory types like device local or host visible. Every device has multiple types of memory that are made available to the user. Each memory type has its uses. Device local memory is faster for GPU access, but cannot be used



(a) Execution model objects relations. (b) Command buffer recording and submission.

Figure 3.2: Vulkan execution model.

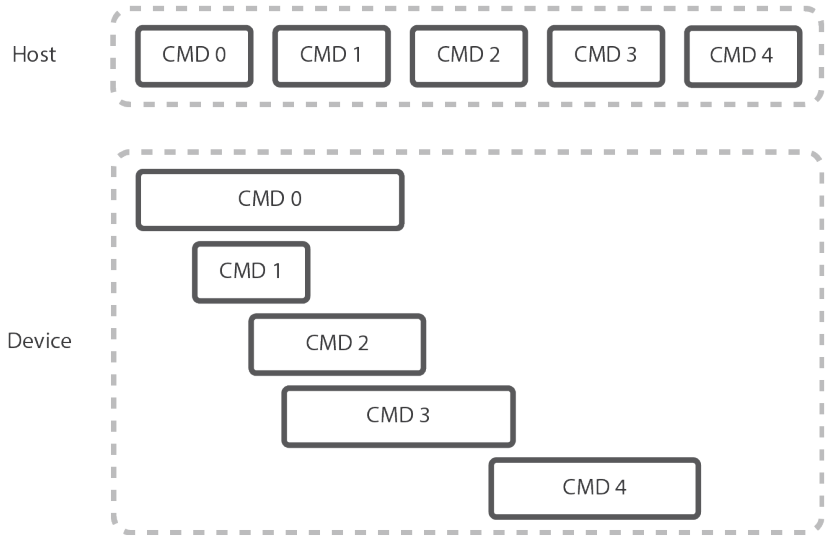


Figure 3.3: Vulkan command buffer execution order as submitted by the CPU and as run on the GPU.

to upload data to the GPU since the host cannot write to it. Host visible memory is used to write or read by the CPU. Buffers are also used to transfer data back and forth from the device to the host.

A **VkBufferView** is view into a contiguous range of a buffer along with a specific format used to interpret the data. Buffer views are used to interpret buffer data from a shader.

Vulkan also exposes **VkImage**. Images are used as containers of pixel data. They represent a multidimensional array (up to 3) of data. They are used as textures and attachments. They can store color, depth or stencil data. They can be accessed by shaders.

Similarly with buffer views, a **VkImageView** is an object that describes a view of a VkImage. It represents a range of an image and the format with which the underlying data should be interpreted by the shader. It also allows for the image to be used with different types of samplers.

Samplers (**VkSampler**) are used to define how textures are filtered when read by the shader. Various options are exposed as part of the sampler, with the most important being the minification and magnification filter, which can either be nearest-neighbor or linear filtering.

A descriptor is not strictly speaking a Vulkan object, but still an important concept. A descriptor represents shader resource such as buffers, buffer views, image views, and samplers. Descriptors are organized in descriptor sets. These are bound during command recording and allow the shader in the following drawing commands to use the resources.

Another important resource is the **VkFramebuffer**. This resource describes a set of images called attachments that will be written on when rendering. The framebuffer collects the attachment

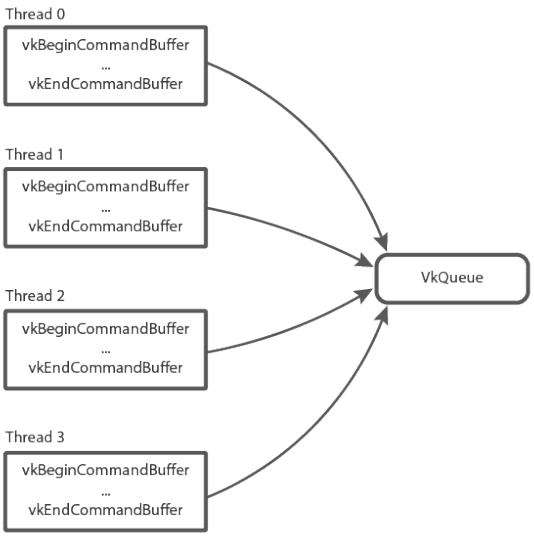


Figure 3.4: A multi-threaded example of command submission to the GPU.

types and some configuration options for these attachments. To output rendering results to an image, it must be attached to a framebuffer.

In this thesis, the Vulkan API was used to implement both a hybrid renderer and a path tracer. The basis of computation on the GPU is its resources. The data model used when rendering is important, as well as how these resources are used on a low-level when the implementation is using the API directly. Although we do not go into detail about the exact resources used for our implementation in Chapter 4, some context is important especially when it comes to reading the source code provided with this thesis.

3.1.4 Pipelines

Vulkan pipelines are the set of rules with which work will be performed on the GPU. There are various types of pipelines, each controlling part of the GPU for performing a kind of work.

The most fundamental pipeline type in Vulkan is the graphics pipeline **VkGraphicsPipeline**. The graphics pipeline defines the way in which we transform geometric data into pixels in an image. This pipeline is comprised of programmable and fixed-function stages. Common functionality between shaders are descriptor sets defined per pipeline indicating the resources each shader has access to. Also, a small amount of data, 128 bytes guaranteed by the specification, can be passed to the shaders with a mechanism called push constants. Basically, a small buffer can be read by the CPU and GPU very fast. The input assembler assembles vertices into geometric primitives such as

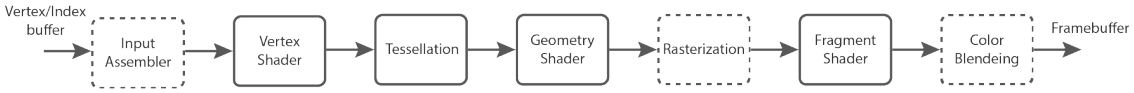


Figure 3.5: Graphics pipeline stages. The solid blocks are programmable stage and the dashed blocks are not.

points, lines, triangles. In the vertex shader the user can define how vertices are transformed and other attributes are calculated from the vertices. In the tesselation stage more geometry can be generated from a tesselation shader. The geometry shader stage operates on groups of vertices and emits zero or more output vertices. The reasterization stage converts primitives into fragments or pixels associated with a region of the framebuffer. The fragment shader then runs on the generated fragments from the rasterizer. This shader outputs a color value per fragment. This color value is then blended with the existing framebuffer value if specified. In this stage, depth and other values are output to the framebuffer.

Compute pipelines are relatively simpler. They are used to run a compute shader to process data. GPUs being massively parallel execution engines, compute shaders can run multiple times when the compute pipeline is called. This is controlled by how many work groups and invocations the compute shader is going to run. Work group size is set by the user when dispatching a

compute command. They define how compute workloads are formed and processed by the compute hardware. The invocation number are set by the shader and can be thought of as threads that will run in parallel per workgroup. Invocations within the same work group have access to shared memory.

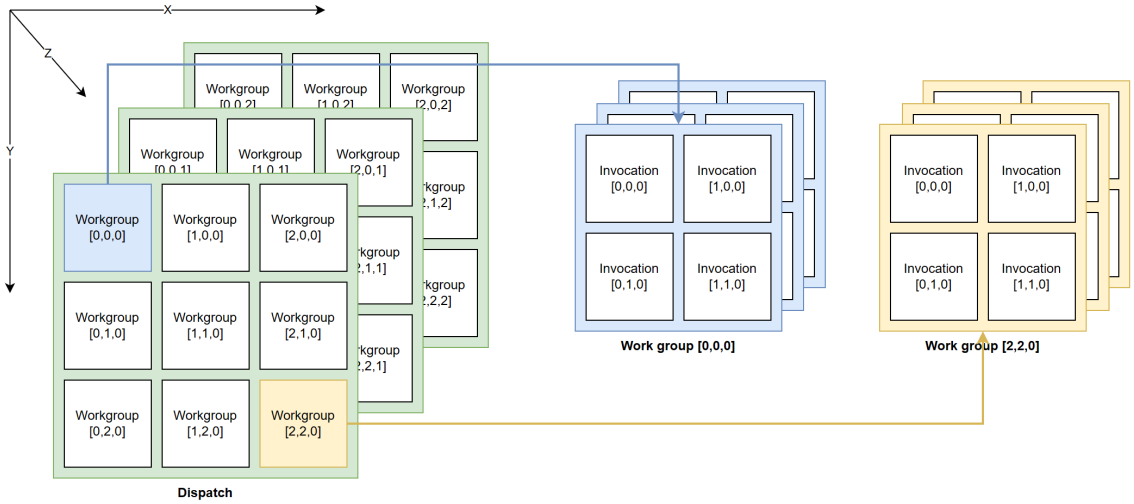


Figure 3.6: Work group and invocations of a compute dispatch. [Ove15]

In this thesis, the graphics and compute pipelines were used extensively. These pipelines are the basis for computation on the GPU. They are the the model by which the GPU carry out large parallel computations in fast manner. The exact implementation details are not presented but can be found in the source code provided.

3.1.5 Synchronization

As described earlier Vulkan command buffers are not synchronized by the API. This mean that the user must ensure the correct synchronization between commands. This is done using Vulkan's synchronization primitives.

One such primitive is the **VkFence**. A fence is created on the host and submitted with any submission to a queue. The host can then check if the fence is signaled or wait for the fence to be signaled. The fence will be signaled when the queue has finished executing the work associated with the submission. This primitive is useful for injecting dependency between the host and the device, e.g., when uploading data to the device and need to wait before continuing to give commands to the device.

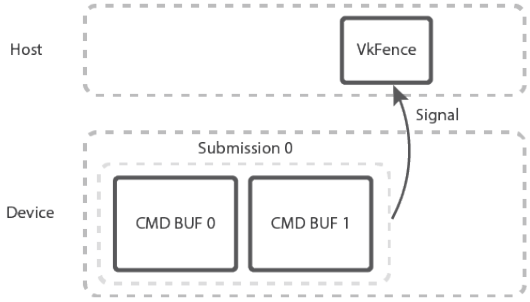


Figure 3.7: Diagram showing the Host/Device relationship when VkFence is used.

Another primitive for synchronization is the **VkSemaphore**. This primitive is used to insert a dependency between queue submissions and other queue submissions to the device. It is used to signal that one submission batch has been completed. Another batch submitted to the device can wait on the semaphore and begin execution when the previous work is completed. This is useful when defining GPU dependencies for the resources and other work has been completed.

Vulkan also provides pipeline barriers. This is a synchronization primitive that allows for fine-grained control and dependency injection between commands. Pipeline barriers are recorded

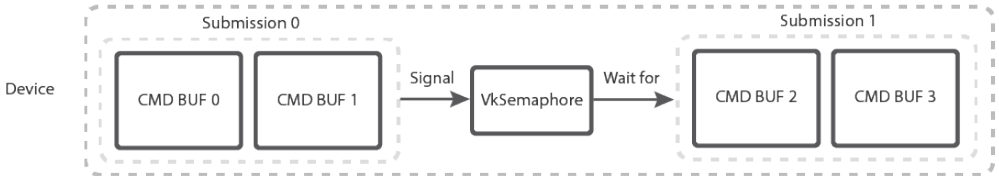


Figure 3.8: Diagram showing the relationship between submissions when a VkSemaphore is used.

directly into the command buffer and cannot be interacted with by the host. A pipeline barrier can be an execution barrier, meaning that the dependency is applied between all previous commands before the barrier and the next command. A pipeline barrier can be applied per resource. In that case, the dependency is injected between the last command that uses the resource and the next command using the resource. This is a very powerfull tool as it defines the minimum wait required for a resource to be available for the next command to execute or similarly for the command that must be completed before the next command can execute. Barriers can be applied to images and buffers.

Through pipeline barriers, image memory layout transitions are also handled. On a GPU there are multiple types of memory used when processing images depending on the work that is performed on it called memory layout. The user must ensure that the appropriate memory layout is used for the image according to the command that will be executed on it using pipeline barriers to transition its layout.

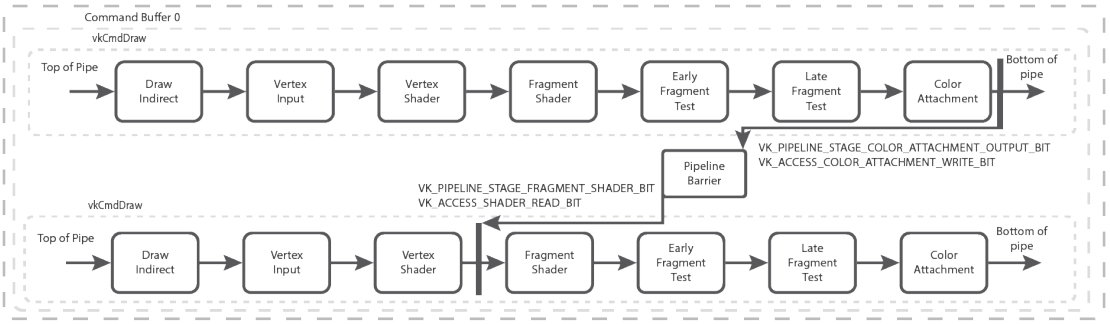


Figure 3.9: Diagram showing the relationship between two draw calls when using a pipeline barrier.

Finally **VkEvent** is the most flexible synchronization primitive. An event is created by the host, and the user can either choose to signal the event either from the host or the device and can also wait for the event from the host or the device. This allows for a complete user-driven synchronization to occur.

Vulkan also provides timeline semaphores and GPU idle operations. These are outside the scope of this thesis.

In this thesis, correct synchronization is very important as it ensures accurate and fast results. It is also important to ensure that data and execution dependencies are defined correctly, when implementing anything that uses the Vulkan API.

3.1.6 Vulkan Ray Tracing Extension

Hardware based ray tracing has been part of the Vulkan specification since 2018 [Gro24] with the VK_NV_ray_tracing extension. This was later supplanted by VK_KHR_acceleration_structure and VK_KHR_ray_tracing_pipeline extensions. This marked the official support of ray tracing on Vulkan. This feature has allowed many new possibilities when rendering. Hardware-based ray tracing is fast, allowing for ray tracing to be used for real-time applications. Many applications have used this extension to add ray-traced effects like shadows and reflections.

These extensions define an acceleration structure, the ray tracing pipeline, and the shader binding table. To achieve good ray tracing performance, an optimized data structure is built over the scene called an acceleration structure (AS). As seen in 3.10 the acceleration structure is built from multiple components. Geometry like vertex and index data or axis-aligned bounding boxes is used to build a bottom-level acceleration structure. These hold geometric information and are the smallest components of the structure. Instance information is then created from the

bottom-level acceleration structure. An instance holds transform and shading information. One bottom-level acceleration structure can be referenced by many instances. Instance and bottom-level acceleration structure information is used to create the top-level acceleration structure. The method by which the acceleration structure is created is proprietary and is implemented by the hardware manufacturer. The application can use the created acceleration structure of the scene to perform query operation or trace ray against it.

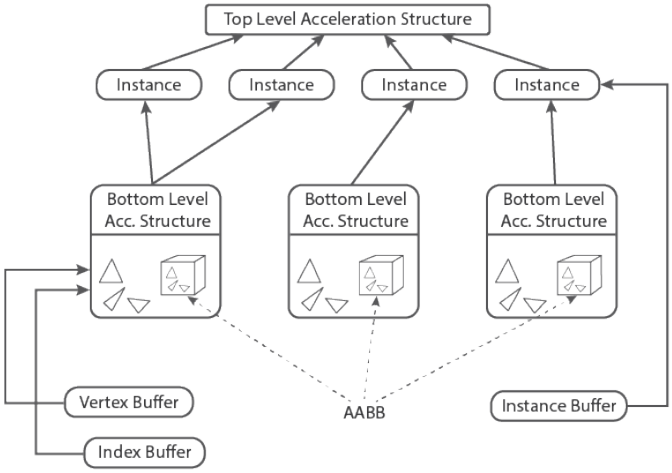


Figure 3.10: Diagram showing the structure of a Vulkan acceleration structure.

Ray Tracing Pipeline

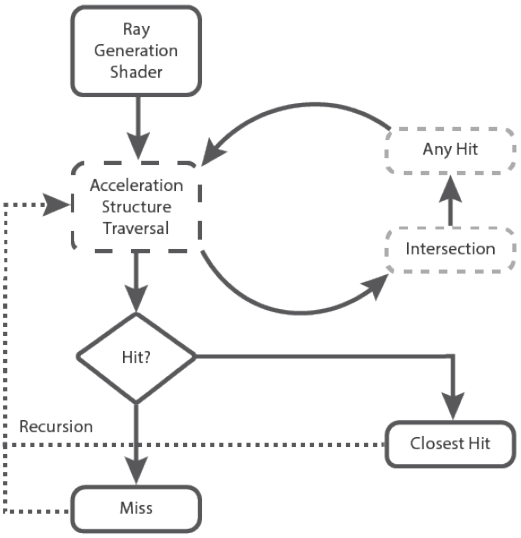


Figure 3.11: Diagram of Vulkan's ray tracing pipeline. Solid blocks are programmable. Dashed and gray blocks are optimal and user-programmable. The black dashed line blocks are fixed.

The ray tracing pipeline allows for tracing rays against the acceleratrion structure of the scene. The ray tracing pipeline has multiple shader stages 3.11. Everything starts with the ray generation shader. In this shader, the user has to define the rays that will be traced as part of this ray tracing execution. It also has access to the result of the ray tracing to write to buffers or images. During ray traversal on the acceleration structure, two optional shaders can be invoked. The intersection shader is used when the bottom-level acceleration structure is defined as an AABB. This allows for intersecting with custom shapes. The any hit shader is used for cases where the geometry is transparent. Since a triangle intersection does not necessarily indicate a hit, we have to check the intersection with the any hit shader. Finally, the hit and miss shaders define what happens if the ray traversal results in a hit or a miss respectively. These shaders can trace more rays recursively.

Shader Binding Table

When ray tracing we trace rays throughout the scene and can hit any object. Thus, the GPU must know what shader to use for each object. The shader binding table is the mechanism by which we can do that. The shader binding table has entries for ray generation, hit, miss any hit, and intersection shaders. The table can hold multiple entries. When tracing rays the appropriate offset can be used either directly in the traceRayEXT call inside the ray generation shader, in the vkCmdTraceRays call, or the instance offset defined when creating the acceleration structure.

In summary, the basis of this thesis is the use of the hardware accelerated ray tracing. Thus, access to it through the Vulkan ray tracing extension and the concepts around it are important. The architecture for the path tracer and DDGI implementations are directly based on the way this part of the API functions.

3.2 glTF 2.0

A fundamental aspect of this thesis is how the data should be stored and read by each renderer. The file format chosen is the glTF 2.0 standard [Gro21]. This file format was developed by the Khronos Group and is designed to efficiently store scene data such as node hierarchy, cameras, materials, geometry, animations, and other data. It stores these data using a JSON-based format as well as a binary format for more efficient storing of blocks of binary data. The glTF standard also defines a Physically-Based Rendering (PBR) material model. Materials are defined by a base color, metallic-roughness factor, normal map, occlusion, and emissive textures. Scene structure is also encoded in a hierarchical node structure. The scene is broken down into nodes representing every mesh, light, and camera. A tree that represents the hierarchical relationship of each node is also stored. Another important feature of glTF is extensibility. It supports extensions that can be used to extend the supported types of data. An important extension is KHR_lights_punctual [Gro18] as it defines directional, point, and spot light sources. Light sources are defined by color, type, intensity, and range.

In this thesis, there was a need for a 3D scene description format that could be used to define the scene in a program like Blender 3D instead of building tools around the renderer for that. glTF 2.0 is a very powerful format with a wide variety of support both in 3D modeling software and other engines. It allows for a common format to define any scene making software and renderer development more streamlined. Thus, it was chosen for this thesis.

3.3 auto-vk-toolkit

While Vulkan is a very robust and powerful API this power is traded for a verbose syntax. This results in slower and more error-prone development. To address this issue several frameworks have been developed to abstract some of this detail and allow for faster development with less errors. auto-vk-toolkit and auto-vk [UKW23] are two such frameworks for the Vulkan API. The first one, auto-vk-toolkit, is an application framework allowing for development of software applications around Vulkan. The second, auto-vk, is the Vulkan abstraction layer. Together, they aim to hit the sweet spot between programmer convenience and efficiency while still supporting full Vulkan functionality. Both are written in model C++ using some libraries such as GLFW for windowing, assimp for model loading, and stb image for image loading.

An application written with auto-vk-toolkit works by defining a set of invokee objects. These objects define an initiaslize (), update(), and render() function as seen bellow.

```
class invokee
{
public:
    virtual void initialize();
    virtual void update();
    virtual void render();
};
```

An invokee is the foundation block with which the user will develop the renderer. It can be used for various tasks but should be used to hold data and define rendering operations for a single stage of the renderer. The initialize () function is used to initialize all resources that the stage will hold such as images, buffers, pipelines etc. The update() function is used to perform all the CPU tasks that need to be done for rendering. Finally, render() records and submits all the GPU work.

```
m_line_pipeline = avk::context().create_graphics_pipeline_for(
    "shaders/debug.vert",
    "shaders/debug.frag",
    avk::cfg::polygon_drawing::config_for_lines(),
    avk::cfg::primitive_topology::lines,
    avk::cfg::depth_test::disabled(),
    avk::cfg::viewport_depth_scissors_config::from_framebuffer(
        avk::context().main_window()->backbuffer_reference_at_index(0)),
    avk::context().create_renderpass({
        avk::attachment::declare(
            avk::format_from_window_color_buffer(avk::context().main_window()),
            avk::on_load::load, avk::usage::color(0), avk::on_store::store
        ),
        avk::attachment::declare(
            avk::format_from_window_depth_buffer(
                avk::context().main_window()),
                    avk::on_load::load.from_previous_layout(
                        avk::layout::depth_stencil_attachment_optimal),
                    avk::usage::depth_stencil, avk::on_store::store),
    },
    avk::context().main_window()->renderpass_reference().subpass_dependencies()),
    avk::from_buffer_binding(0)
        ->stream_per_vertex(&line_vertex::position)->to_location(0),
    avk::from_buffer_binding(0)
        ->stream_per_vertex(&line_vertex::color)->to_location(1),
    avk::push_constant_binding_data{
        avk::shader_type::vertex | avk::shader_type::fragment,
        0, sizeof(push_constants) }
);
```

Figure 3.12: Example of auto-vk graphics pipeline definition for a simple debug line renderer.

When the application starts it creates a window and Vulkan instance based on options provided by the user. The user also creates the invokees that will be used for rendering and passes them to auto-vk. After the application is initialized auto-vk will start to initialize all the ivokees that were given. After this, it enters a loop until otherwise stopped first calling all update() functions and then render().

The Vulkan abstraction layer provided is very powerful. It allows for high-level expression of Vulkan operations while allowing for control when needed. It integrates the popular memory management library Vulkan Memory Allocator (VMA) for GPU allocation operations. An example can be seen in 3.12. In this example, a Vulkan Graphics Pipeline is created based for lines. Everything is defined in the block of code from pipelines drawing mode, the primitive mode to be used by the rasterizer, disabling depth testing, a renderpass based on the swapchain attachments, the input buffer bindings, and finally the push constant data. The same information is not shown here in pure Vulkan as it would not fit a single page. Needless to say, this allows for faster development while still giving great control.

3.3.1 assimp

The assimp loader loads a 3D object file into a scene. The scene contains arrays of meshes, materials, animations, textures, lights, lights and cameras as well as the root node. The root node is the root of a tree-like structure that denotes the hierarchy of meshes in the scene. Each node in the structure contains a list of its child and parent nodes as well as a list of indices to the scene mesh list. Each mesh contains a list of faces and a list of lists for all vertex data in the mesh. It contains data for vertex positions, normals, texture coordinates, tangents and bitangents. Each face contains a list of indices back to the vertex data of the mesh. Each face contains 3 indices in the case that the mesh is triangulated. An interface for assimp and for faster model loading is provided by the auto-vk-framework. Assimp also supports loading for glTF 2.0 file formats.

In summary, there was a need to accelerate development. Writing everything from scratch

was not practical for this thesis. Thus auto-vk-toolkit and assimp were chosen to help accelerate development and make it more simple to express Vulkan concepts in C++.

Chapter 4

Implementation

4.1 Overview

In this chapter, we discuss the work implemented as part of this thesis. Both a Hybrid Renderer and a Path Tracer, as a point of comparison, were implemented. The described implementations are based on common implementations of Scene and Post Processing. These are discussed in separate sections. This chapter describes GPU and CPU related work that was implemented for this thesis. A note on terminology for GPU execution, discrete shader invocations will be referred to as steps. A collection of steps producing a certain effect are referred to as stages. This is not standard in literature but helps distinguish the more common passes terminology with the RenderPass and Subpass objects that are part of the Vulkan specification.

4.2 Scene

The implementation starts with the scene object. The scene contains all assets used to render the final result, such as models, materials, textures, lights, and an acceleration structure. It holds both the CPU memory for those objects as well as the handle to the GPU objects holding this data. The scene is an avk::invokee as described in 3.3. Assets are compiled into a scene in Blender 3D and stored using the gltf 2.0 file format, see more 3.2. This format allows us to save geometry, materials, light, and cameras. Extra information is stored in a scene .json file to allow for further parameterization not possible by the gltf 2.0 standard.

4.2.1 Model

Every 3D model used by a scene is stored as a single gltf file and loaded using the auto-vk-toolkit wrapper to the assimp library as seen in 3.3.1. After loading the model, we group each mesh by its materials loading the grouped index and vertex data together so that they can be rendered as one draw call per material. This was done because the material complexity is relatively low and some materials can require different shaders to be rendered. Binding different pipelines and thus shaders is a costly operation [Kap19] thus the reuse of the the currently bound pipeline whenever possible is important.

After consolidating the meshes we create the appropriate vertex and index buffers. For our vertex buffer we store the vertex position, normal, texture coordinate, tangent, bitangent of each triangle vertex.

```
struct vertex
{
    vec3 position;
    vec3 normal;
    vec2 texcoord;
    vec3 tangent;
    vec3 bitangent;
};
```

Figure 4.1: Vertex data that is used for this project.

The indices are stored in an array of unsigned integers. All data is uploaded to the GPU in separate buffers. This is done to allow this data to be read in ray tracing shaders that are not specialized to read interleaved data like the vertex shader. Buffer views are also for the aforementioned buffers to allow binding the buffers to the ray tracing pipeline. All positions are pre-transformed to world space. Thus there is no need for them to be transformed when drawn.

A cube is also created and loaded in a similar way to pose as a dummy object when needed.

For every mesh and model loaded we calculate the minimum and maximum vertex in world space. The min and max vertex positions are used to calculate a bounding box for the entire scene.

4.2.2 Material

Materials are loaded by the avk framework in the form of avk::MaterialGpuData form the assimp model. All materials are stored in a single buffer as an array of materials. Again this is done to reduce the number of binds of buffers. The following material struct members are the most important for our case. We mainly need the reflectivity for each lighting component Ambient,

```
struct MaterialGpuData
{
    vec4 mDiffuseReflectivity;
    vec4 mAmbientReflectivity;
    vec4 mSpecularReflectivity;
    vec4 mEmissiveColor;
    ...

    int mDiffuseTexIndex;
    int mSpecularTexIndex;
    int mAmbientTexIndex;
    int mEmissiveTexIndex;
    int mNormalsTexIndex;
    ...
};
```

Figure 4.2: Material struct as definition for GPU usage.

Diffuse, Specular, Emissive. We also store the an index to the texture array.

Sampling

For this material model, we use the Lambertian BRDF model. The probability density function is defined as

$$f_{BRDF}(\theta) = \frac{\cos(\theta)}{\pi}$$
$$f_{BRDF}(\vec{\omega_i}, \vec{n}) = \frac{\vec{\omega_i} \cdot \vec{n}}{\pi}$$

Where ρ is the diffuse reflectivity of the surface and θ the angle between the normal vector $\vec{n}$ and $\vec{\omega_i}$ being the incident light ray direction. As for the evaluation function, it is defined as.

$$eval_{BRDF}(\rho) = \frac{\rho}{\pi}$$

The reflected ray direction is sampled by uniform hemisphere as described in 2.1.5.

4.2.3 Textures

Every texture referenced by a material is loaded onto the CPU. The appropriate Vulkan Image are created for each material texture as well as each texture's Sampler. The appropriate format, image layout, and filter mode are selected for the Image and Sampler and texture data is flipped if necessary and uploaded to the GPU. We store all samplers in one consecutive array of samplers. This allows us to be able to bind all textures needed for rendering at once and simply index the array of sampler with indices given by each material.

Except for the material textures, we also read a skybox if specified by the scene .json file. This is loaded as a cubemap in a similar manner to the material textures and a cubemap Sampler is also created.

4.2.4 Camera

When loading the scene with assimp we also load all the scenes' cameras. Both Orthographic and Perspective cameras are loaded. Cameras can be scene cameras or light cameras. Light cameras are used to draw the shadow map of for the respective light. We create the perspective cameras projection matrix using the loaded values for field of view, aspect ratio, near and far plane distance. For the orthographic camera the width is used for the top, bottom, left, right distances and the near and far plane to define the view frustum and the projection matrix.

For each camera we also retrieve its rotation and position with great care to transform them to world space.

4.2.5 Light Sources

Light sources are defined in the scene gltf model file with blender. We support Direct and Spot lights. Each light is loaded from the assimp library. Each light source also has a child camera. This is used to generate the shadow map of the light source. Each camera is loaded as mentioned before. The following information is loaded for the light sources. All information is loaded, and

```
struct Light                              //l
{
    vec3 color;                           //rho
    vec3 position;                        //pl
    vec3 direction;                       //dl
    float angle_outer_cone;               //Coc
    float angle_inner_cone;               //Cic
    float falloff;                        //Cof

    float attenuation_constant;           //Cc
    float attenuation_linear;             //Cl
    float attenuation_quadratic;          //Cq
};
```

Figure 4.3: Light struct definition as used on the GPU.

stored in one continuous buffer that is then uploaded to the GPU.

Sampling

Directional lights light an entire area and have the following probability density function.

$$f() = 1$$

Spot lights on the other hand fade over distance thus the probability density functions is described as follows.

$$f(p) = d^2$$

Where d is distance from a point being lit to the light source. Note that in the end, the inverse of the probability density function is applied to the rendering equation. Thus, the above definition of the function is consistent with the inverse square law.

Attenuation

Light attenuation expresses how light intensity changes as the light travels through a scene. The light attenuation function can be used to calculate the intensity of light at a given position p with incidence direction $\vec{\omega_i}$ for a given light source ℓ. This does not take into account whether the point at position p is visible by the light source. Directional lights are again very simple to model and their attenuation is calculated as follows.

$$\mathcal{I}() = 1$$

Spot lights are more complicated and are calculated as described in [Gro18]. This breaks down the data loaded by glTF2.0 into two factors. The linear factor, presented below.

$$\mathcal{I}_l() = \frac{1}{4\pi(C_c + C_l d + C_q d^2)}$$

The angular component of the attenuation is calculated in two parts. The angular scale described below.

$$\mathcal{I}_{as}() = \frac{1}{\max\left(0.0001, \cos\left(C_{ic}\right) - \cos\left(C_{oc}\right)\right)}$$

and the angular offset.

$$\mathcal{I}_{ao}() = -\cos\left(C_{oc}\right)\mathcal{T}_{as}$$

The final angular component is calculated.

$$\mathcal{I}_a(\phi) = \left(\cos\left(\phi\right)\mathcal{T}_{as} + \mathcal{T}_{ao}\right)^2$$

Where ϕ is the angle direction of the spotlight and the light vector. Combining the angular and linear components as follows we get the final attenuation of a spot light.

$$\mathcal{I}(\phi) = \mathcal{T}_l() * \mathcal{T}_a(\phi)$$

4.2.6 Acceleration Structure

All the vertex data loaded from the model is used to build bottom-level acceleration structures for each mesh. Every light source also gets a small cube mesh to be able to check visibility during path tracing. We create geometry instances for each object and light source BLAS. We set a difference mask and instance offset to the geometry instance. After all base level data is ready we create the top level acceleration structure.

4.3 Hybrid Renderer

4.3.1 Overview

This is our hybrid renderer. Using both traditional vertex/fragment pipelines and compute pipelines as well as ray tracing pipelines to render scenes. In this section with the help of 4.4 we will go over the general steps that are taken to render a scene.

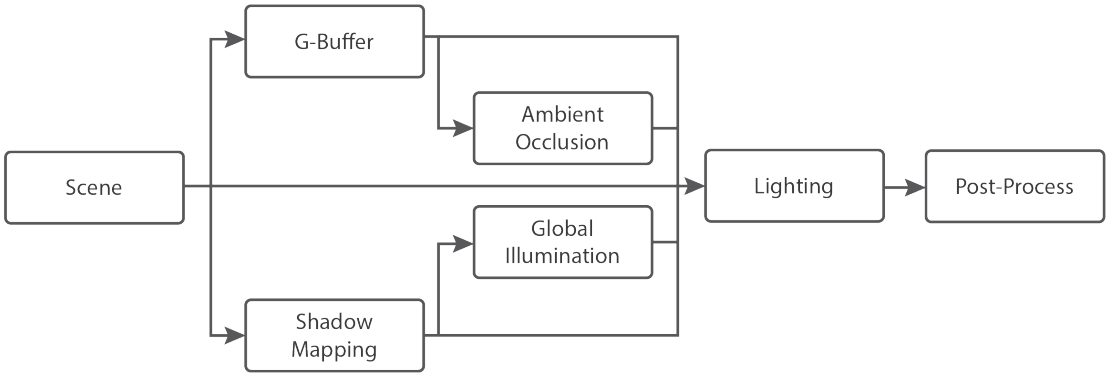


Figure 4.4: Overview of the hybrid renderer's render passes and techniques.

Scene

The scene stage is responsible for loading and storing all assets and ensuring that their respective GPU resources are up to date.

G-Buffer

The G-Buffer stage processes geometry to produce the G-Buffer for the main camera. It depends on the Scene stage.

Ambient Occlusion

This stage processes the G-Buffer to produce Ambient Occlusion lighting information using the SSAO technique. It depends on the G-Buffer and Scene stages.

Shadow Mapping

In the Shadow Mapping stage, we produce shadow maps and variance shadow maps for each light source. This stage depends on the Scene stage.

Global Illumination

This is the most complicated stage. It uses a mix of ray-tracing, vertex/fragment and compute shaders to produce indirect illumination information for the scene. It uses the DDGI technique. It depends on the Shadow Mapping and Scene stages.

Lighting

This stage combines all the above information using deferred shading technique to draw the frame. It depends on the Scene, G-Buffer, Ambient Oclusion, Shadow Mapping and Global Illumination stages.

Post-Process

The Lighting stage does not output the final frame. Some post processing is needed to present the final frame. This is done in this stage. Tone mapping and anti aliasing is applied to the final frame.

4.3.2 Shadow Map

In this stage we calculate a shadow map and a variance shadow map as described in 2.3.3, and [DL06] for each light source in a scene. Light sources and their respective cameras are defined in the glTF scene file produced with Blender 3D. In the first vertex/fragment pipeline we render all scene geometry from the perspective of the light source. This pipeline outputs a depth texture $d[0, 1]$ in D32SFloat format and a variance shadow map composed of a pair of values (z, z^2) where z being the respective coordinate of the geometry being rendered. This is done according to and is stored in R32G32SFloat format. These render targets are cleared to 1.0 and $(0.0, 0.0)$ respectively.

The following vertex/fragment pipelines are both blur passes. For the full effect of soft shadow, the variance shadow map needs to be blurred. We apply a Gaussian blur. This is done in two steps to employ separable filtering for better computational complexity as described in 2.5.2. First, a horizontal, then a vertical blur with a filter radius of 5. In order not to introduce artifacts, great care is taken to exclude infinite (close to the camera's far plane). Filter weights that would be multiplied with infinity are stored and multiplied with the z value of the far plane once. If the resulting value is close to the z value of the far plane the entire result is overwritten with a large negative value.

Each step is repeated for stage as demonstrated in 4.5. This is done to reduce the number of costly pipeline binding operations [Kap19]. Every step employs appropriate semaphores and memory barriers to ensure memory safety.

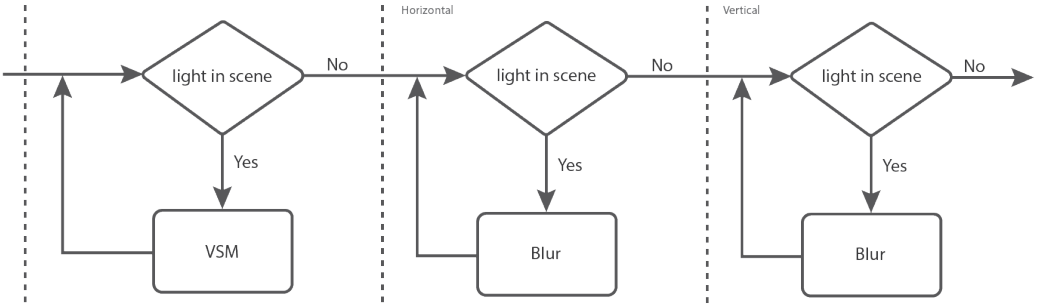


Figure 4.5: Shadow Mapping flow chart.

Sampling

Since both traditional and variance shadow maps are used in this implementation, both textures generated from the previous stage need to be sampled and combined to produce the optimal result. Sampling occurs at a certain position with a given normal, color, and w_o being the outgoing light direction.



Figure 4.6: Visualization of variance shadow map blurring process. From left to right: None, Horizontal Blur, Vertical + Horizontal Blur. Images are normalized for visualization.

```
float shadow_map_visibility(vec4 coord,
    sampler2DShadow sm, vec2 inv_size) {
    vec4 s = vec4(inv_size, -inv_size.x, 0.0);
    vec4 d = s * 1.4;
    return ((shadow_fetch(shadow_map, shadow_coord, inv_size) +
        shadow_fetch(sm, coord + s.xww, inv_size) +
        shadow_fetch(sm, coord - s.xww, inv_size) +
        shadow_fetch(sm, coord + s.wyw, inv_size) +
        shadow_fetch(sm, coord - s.wyw, inv_size) +
        shadow_fetch(sm, coord + d.xyw, inv_size) +
        shadow_fetch(sm, coord - d.zyw, inv_size) +
        shadow_fetch(sm, coord + d.zyw, inv_size) +
        shadow_fetch(sm, coord - d.xyw, inv_size)) / 9.0);
}
```

Figure 4.7: Pseudo code excerpt from the shadow_map_visibility function that is used to sample the shadow map and return a visibility value.

For every light source attenuation, and w_i being the incoming light direction is calculated for these sampling parameters according to 4.2.5 and 4.2.2. The coordinate of the given position in the shadow map space is calculated using the model view projection matrix for the light source, more in 2.1.2. An additional matrix is inserted into the usual multiplication chain. This matrix is an offset matrix to help with a visual artifact called shadow acne. It transforms the position by a small offset amount, called bias. In this case, the bias is set to 0.05. Also the distance between the position being sampled from the light source is calculated. These coordinates and distance are used to calculate the visibility term for each shadow mapping technique. That is, traditional shadow maps and variance shadow maps. From the two terms, the smallest is chosen. The final term is multiplied by the attenuation. If the attenuation meets a threshold and thus affects the lighting of the point being lit direct illumination for this given light is calculated. The final light for each light source is calculated according to the rendering equation as described in 2.4.3. All the direct illumination values for each light are summed together and returned. The process described here can also be seen in the form of pseudocode in 4.8.

When sampling the traditional shadow mapping, the map is sampled at the coordinates for the point being shaded. Also it is sampled in a square around the point being sampled and the resulting value is averaged. This technique is called percentage closer filter and results in smoother shadows [Fer04]. When the shadow map is sampled, it uses a specific sampler provided by Vulkan/GLSL that compares the resulting value of the texture with the z value provided and returns 1 if the the point is visible or 0 if it is not. More options for how this comparison is performed is provided to the sampler at creation time 3.1.3. The pseudocode can be seen here 4.7 for this process.

To sample the variance shadow map the coordinate calculated earlier is also used. The distance of the point being sampled from the light source is also used. The mean and variance values are reconstructed. A Chebychev term is calculated according to 2.3.3 and [DL06]. If the distance or light_space_z is less than equal to the mean, the point is fully shaded and 1 is returned. Otherwise,

```
vec3 direct_illumination(position, normal, color, w_o) {
    vec3 direct_illumination = vec3(0.0);
    for (l : light_array) {
        auto [attenuation, w_i] = compute_attenuation(normal,
            l, position);
        if (attenuation > attenaution_threshold) {
            vec4 coord = l.mvp * vec4(position, 1.0);
            coord = coord / coord.w;
            float visibility = shadow_map_visibility(
                coord, light_shadow_map_sampler, l.map_inv_size);

            vec4 light_space_z = -(l.view * vec4(position, 1.0)).z;
            float vsm_visibitliy = vsm_visibility(shadow_coord,
                light_space_z, light_vsm_sampler,
                l.light_bleed_reduction);

            visibility = min (visibility, vsm_visibility);
            attenuation *= visibility;

            if (attenuation > attenuation_threshold) {
                direct_illumination += attenuation * abs(dot(normal, w_i))
                    * l.color.rgb * material_pdf(color)
                    / (light_pdf(l, position) + EPSILON);
            }
        }
    }
    return direct_illumination;
}
```

Figure 4.8: Pseudo code excerpt from the direct_illumination function calculating direct illumination for a given surfel.

the Chebychev term is returned and used to modulate the attenuation value.

```
float vsm_visibility(vec2 coord, float light_space_z,
    sampler2D vsm) {
    vec2 M = textureLod(vsm, coord, 0).xy;

    float mean = M.x;
    float variance = M.y - M.x * M.x;

    float tSubMean = light_space_z - mean;
    float chebyshev = variance / (variance + (tSubMean * tSubMean));

    if (light_space_z <= mean) {
        return 1.0f;
    }
    else {
        return chebyshev;
    }
}
```

Figure 4.9: Pseudo code excerpt from the variance_shadow_map_visibility function.

4.3.3 G-Buffer

The G-Buffer stage is the stage where the G-Buffer structure is generated. This is a core structure for the deferred shading technique described in 2.3 and [Tha11]. Each mesh and the scenes skybox are all rendered into a collection of textures with the following format.

```
struct vertex {
    vec4 position;   //R16G16B16A16Sfloat
    vec4 normal;     //R8G8B8A8Unorm
    vec4 diffuse;    //R8G8B8A8Unorm
    vec3 glossy;     //B10G11R11UfloatPack32
    vec4 emissive;   //R8G8B8A8Unorm
    float depth;     //D32Sfloat
};
```

Position and normal are transformed to screen space to allow for screen space effects in Ambient Oclusion stage. The rest are calculated from the meshes material and textures. After all meshes are rendered into the GBuffer the skybox gets rendered into it. The skyboxes diffuse color is output to the emissive texture to be used later in lighting calculations. Even though zero is output to the glossy texture it is used for later calculations to simulate using glossy texture in the rendered for accurate performance. The output resolution is the same as the internal renderer resolution. This is either the final frame resolution or a greater number when SSAA is running. Normals are calculated using each mesh material using normal mapping technique discussed in 2.3.

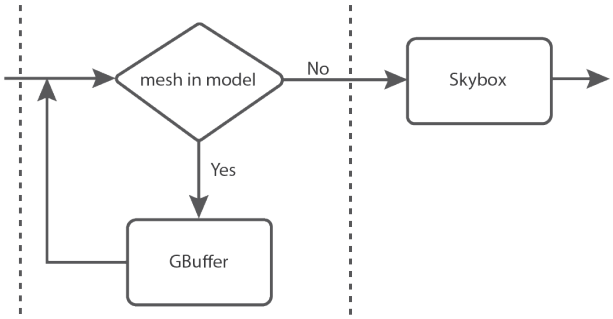
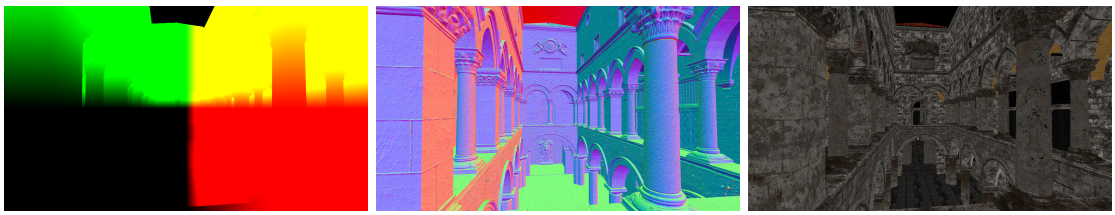
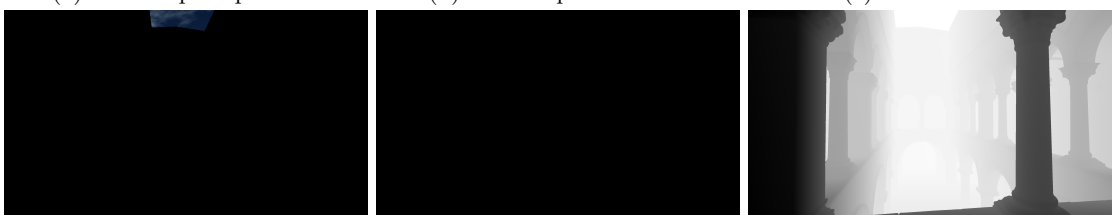


Figure 4.10: G-Buffer stage flow chart.



(a) Screen space position (b) Screen space normal (c) Diffuse color



(d) Emissive (e) Glossy (f) Depth buffer

Figure 4.11: G-Buffer render targets

4.3.4 Global Illumination

Global Illumination is the most important effect implemented in this thesis. This is an implementation of the hybrid technique called "Dynamic Diffuse Global Illumination" 2.3.3 or DDGI. It uses

both ray tracing and rasterization to produce Global Illumination. It is a probe-based technique. This means that the scene is split into sections. Each section holds a probe in its center that collects irradiance and visibility information for the surrounding geometry. The probe information is updated and every frame can be used to sample the indirect light in the area when rendering. This implementation is based on the information and resources generously provided by [Maj+19] [Maj+21].

Probe Volume

As mentioned above, this algorithm is probe-based. This means that the scene is split into equal sections that all have a probe associated with them. The collection of probes in a scene is called a probe volume. The probe volume has a probe count per axis. Splitting the scene in such a way allows for efficient calculation and storage of indirect illumination of the scene. The irradiance and visibility information that is calculated and stored in the volume is used for both rendering the final frame and calculating the next frames volume state. If no geometry or lighting changes, this information does not need to be updated. It would be highly inefficient to update the volume for every frame without caching the results and reusing them. Thus the more dense the volume is, the more information needs to be stored and the more calculations need to be performed per frame. Every scene needs to be tuned for the most realistic results with the least amount of probes in the scene.

For each probe irradiance, visibility, offset and state information are stored. The probe irradiance is encoded in a texture of format A2B10G10R10UnormPack32 with each probe encoding irradiance per direction using octahedral mapping 2.1.6 with a resolution of 8x8 pixels. Visibility per direction information is encoded in a R16G16SFloat texture in a similar way to the Variance Shadow Mapping technique. Distance and distance squared from the probe to geometry per direction is stored in 16x16 pixel resolution using octahedral mapping. The probe grid is uniformly distributed in the volume. This can result in some probes being placed inside geometry. This can cause visual artifacts when sampling the volume. To solve this issue, this technique allows for probes to be moved to a better position, outside of enclosing geometry, so that it can contribute to the final result. If this is not possible, the probe is disabled as it would be inefficient to calculate information for probes that do not contribute, or even contribute negatively to the final result. More about this method is discussed in 4.3.4. The offset information as well as the state of the probe is encoded into a single texture. The offset is stored as the RGB component of the texture and the state as the Alpha component. The offset is a simple 3 component vector from the center of the probe. The state is defined as seen in 4.12. All probes start in state UNINITIALIZED

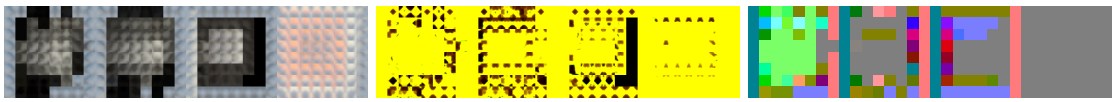
```
enum probe_state {
    OFF             = 0,
    JUST_VIGILANT   = 4,
    VIGILANT        = 5,
    UNINITIALIZED   = 6,
};
```

Figure 4.12: DDGI Probe states

before their offsets and final state have been calculated. After offsets have been calculated probes that have visible geometry are set to JUST_VIGILANT. More rays are traced for these probes to try and get them to converge faster. After rays are traced for JUST_VIGILANT probes they are set to VIGILANT. For these probes, rays will always be traced and their results will be updated over a number of frames. Probes that are inside walls or outside the scene are set to OFF and are not being traced. The original paper also has more probe states for frustum culling probes and handling dynamic geometry but in this implementation since all geometry is static no such states were implemented. Offset and probe state are encoded in a R8G8B8A8SInt texture.

Sampling

To calculate the indirect light of a point $\mathbf{x}$ on a surface S the 8 probes around it are considered (Figure 4.14). These 8 probes form what is called a probe cage. The process of calculating indirect light from the volume is referred to as sampling the volume. The process that is followed can be seen in Figure 4.15. Even if a probe has been offset in space for the purposes of sampling it is considered to fall on a regular grid. For each probe P of the cage the following terms are calculated:



(a) Example probe volume irradiance texture.

(b) Example probe volume visibility texture.

(c) Example probe volume offset and state texture.

Figure 4.13: DDGI render targets.

a) The direction from the probe to the point that is being shaded. b) The distance between the two. The probe's visibility term is loaded from the volume's texture based on the direction of the probe to the point. From this visibility term, the mean and variance are calculated to perform the Chebyshev test and calculate the Chebyshev weight as described in Section 2.3.3. This term is used as a weight for the final irradiance that is sampled. This accounts for occlusion between the probe and the point that is being shaded. The weight reduces the irradiance reaching the point proportionally to how much it is occluded. Trilinear interpolation [Bus03] is also performed, based on the position of the probe and the point. Probes that are closer to the point that is being shaded are factored more than probes that are far away. Finally, a weight is calculated based on the dot product of the normal of the point being shaded and the direction to the probe. This accounts for probes that are not facing the probe. Irradiance should not reach the point if the probe is behind it. At the end, a total weight is calculated from all of the above terms. This weight is multiplied by the irradiance sampled at the probe with the direction of the normal vector. All probe irradiance contributions are added and normalized by the sum of weights for each probe.

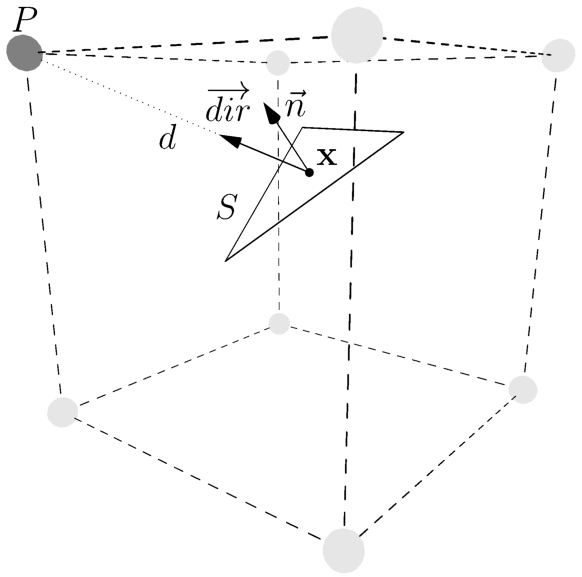


Figure 4.14: A point $\mathbf{x}$ with normal $\vec{n}$ on surface S that is being shaded by probe P of the probe cage.

Ray Casting

This technique uses ray casting in the scene to compute a G-Buffer like structure to then be used to update the probe volume. Rays are traced from each probe in uniform directions calculated by Spherical Fibonacci method described in Section 2.1.3. This allows for a uniform sampling of a sphere in all directions for any number of rays that are being traced. These rays are then randomized by rotating them by a random rotation matrix generated by the technique described in Section 2.1.4. This matrix is updated on every frame, thus ensuring sampling of more directions from the probe in a uniform way. Rays are traced using the a ray tracing pipeline. This pipeline only defines a ray generation shader for each probe as described above, a closest hit shader, and a miss shader. The closest hit shader reads the hit geometry's information and performs barycentric interpolation (Section 2.1.1). For hits the shader stores information on the DDGI G-Buffer. It stores the hits diffuse color, glossy reflectivity, and emissive value combining textures and material information. It also stores the world space normal, world space position, distance of hit, and

```
vec4 irradiance = vec4(0.0);
for (int i = 0; i < 8; ++i) {
    vec3 dir = normalize(P - X);
    float d = length(dir);
    float weight = (dot(dir, n) + 1.0) *.5;
    vec2 temp = texture(visibility_sampler, ...).xy;
    float mean = temp.x;
    float variance = abs(square(mean) - temp.y);
    float chebyshevWeight = 1.0;
    if (d > mean) {
        chebyshevWeight = variance / (variance + square(d - mean));
    }
    weight *= chebyshevWeight;
    weight *= trilinear(P, X);
    vec3 probeIrradiance = texture(irradiance_sampler, ...).rgb;
    rradiance += vec4(weight * probeIrradiance, weight);
}
irradiance.xyz *= 1.0 / irradiance.a;
return irradiance;
```

Figure 4.15: Pseudo code for sampling irradiance from the probe volume.

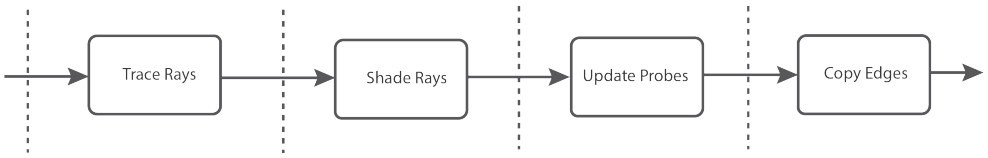


Figure 4.16: This is the flow chart of the DDGI algorithm for every frame. Every block indicates each shader that is being run. Every dashed line indicates the binding of a pipeline.

whether the hit was a front face hit or a back face hit. Position, distance, front face information are packed in one 4 vector value. If the ray hits no geometry, it is marked as a miss.

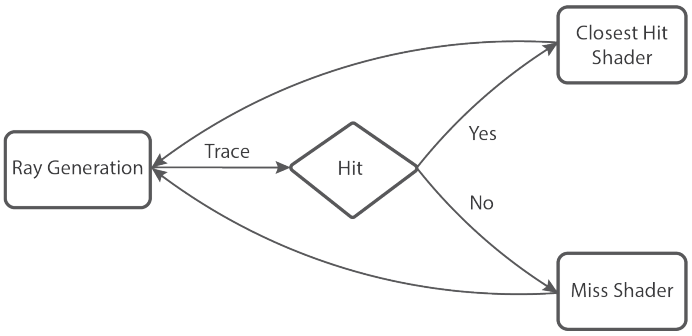


Figure 4.17: Diagram showing the flow of the ray tracing pipeline.

Probe Offsets

Information gathered at the ray casting stage can be used for moving the probes automatically to better position to collect irradiance information of the scene. This process is performed at the start when the probes are considered uninitialized. A small amount of rays are cast and then a compute shader is run to compute probe offsets. This utilizes the front/back face information gathered for the rays. If the probe's rays does not hit any back face triangles, it is located in a good location to collect lighting information and shade triangles and it is set to NEWLY_VIGILANT. If the probe has a certain percentage of rays that hit back face triangles, it is moved by a certain amount toward the closest back face hit location. This process is repeated a few times trying to

find optimal positions for all probes. If after that a probe sees only back face ray hits it can be turned off. It is probably still inside the geometry or only visible to the sky and will not contribute. The probes final state and offset are recorded in the offset texture.

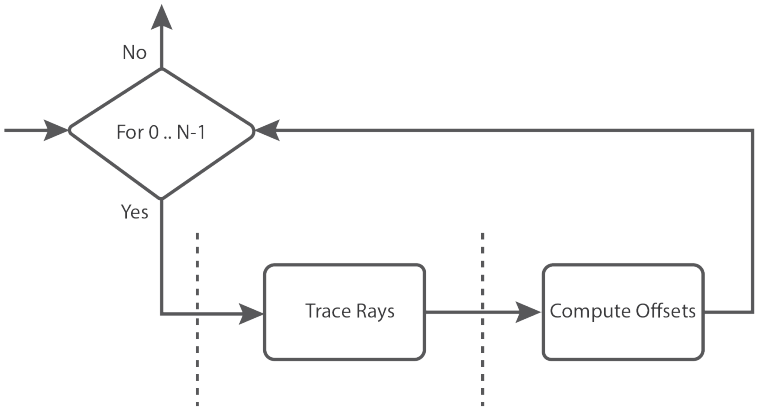


Figure 4.18: This is a flow chart showing all the stages that run when calculating probe offsets. Dashed lines show when pipelines are bound.

Ray Shading

Rays are shaded using a full screen fragment shader pass. For each pixel of the G-Buffer the fragment shader collects information for the point being shaded. Using the same technique described in Section 4.3.6, the rays are shaded as if they were pixels for the final frame. This means that this stage needs up-to-date shadow maps. Indirect lighting from the probe volume also contributes to fragment shading. Since the final shaded rays will be used to update the probes with irradiance information injecting the previous frames, probe irradiance information in this stage allows DDGI to simulate multi-bounce lighting.

Update Probes

Probe information is updated by a set of two shaders using information from the Ray Casting and Ray Shading stages. For each probe and for each irradiance texel (texture element or pixel) the texel's direction $\overrightarrow{dir_p}$ is calculated using octahedral mapping (Section 2.1.6). For each ray that was traced for this probe a weight is calculated as such $W_r = \max(0, \overrightarrow{dir_r} \cdot \overrightarrow{dir_p})$. This weight allows for rays with direction similar to the direction of the texel that is being updated to contribute more. For the N rays that were traced for each probe the new Irradiance value for a given texel is given from the following:

$$I_p = \frac{\sum^N I_r W_r}{\sum^N W_r} \tag{4.1}$$

Similarly the visibility term of the probe on a given texel is given as:

$$V_p = \frac{\sum^N (d_r, d_r^2) W_r}{\sum^N W_r} \tag{4.2}$$

These terms are not stored directly on the texture they are interpolated with the old values using the $mix()$ function according to:

$$newValue = result \times (1 - h) + oldValue \times h \tag{4.3}$$

The value h is a hysteris values. It makes the update of the final value slower, thus allowing for more consistent visual result between frames. This process is performed in the probe update shader. Afterwards another shader is called copying the edges of the texture on each probe's border. This is done to allow for sampling of the probe's area without reading values from other probes.

4.3.5 Ambient Occlusion

This Ambient Occlusion technique is a screen space technique based on [Vri15] and [Mit07]. For a given point in the G-Buffer it samples the scene in screen space in a hemisphere around the point. The points on the hemisphere are generated on the CPU for a unit hemisphere and saved to a buffer used as the sampling kernel. A texture with random rotations is also used to add some randomness to the hemisphere. Kernel samples for this implementation are set to 64 and the random direction texture is 4 by 4 pixels.

```
glm::vec3 sample(
    randomFloat(generator) * 2.0f - 1.0f,
    randomFloat(generator) * 2.0f - 1.0f,
    randomFloat(generator)
);
sample = glm::normalize(sample);
sample *= randomFloat(generator);
const float scale = static_cast<float>(i) / static_cast<float>(m_kernel_size);
sample *= glm::lerp(0.01f, 1.0f, scale * scale);
m_kernel_values[i] = glm::vec4(sample, 0.0f);
```

Figure 4.19: Pseudo code excerpt from the code that is used to generate the SSAO kernel.

Each sample in the kernel is a random vector on the hemisphere's surface that is scaled by a random amount to move it somewhere within the hemisphere. The sample is further scaled by an amount proportional to the inverse of the number of the sample. This biases the samples toward the center of the hemisphere. This process can be seen in more detail in 4.19.

```
mat3 TBN = mat3(tangent, biTangent, screenSpaceNormal);
float occlusion = 0.0;
for (uint i = 0; i < c_kernel_size; i++) {
    vec3 samplePos = TBN * u_kernel[i].xyz;
    samplePos = screenSpacePosition + samplePos * p_radius;

    vec4 offset = vec4(samplePos, 1.0);
    offset = p_projection_mat * offset;
    offset.xyz /= offset.w;
    offset.xy = offset.xy * 0.5 + 0.5;
    float sampleDepth = texture(u_position, offset.xy).z;
    float rangeCheck = smoothstep(0.0, 1.0,
        p_radius / abs(screenSpacePosition.z - sampleDepth));
    occlusion += (sampleDepth > samplePos.z + p_bias ? 1.0 : 0.0)
        * rangeCheck;
}

occlusion = 1.0 - (occlusion / float(c_kernel_size));
o_color = pow(occlusion, p_power);
```

Figure 4.20: Pseudo code excerpt from the SSAO shader.

This information is used to calculate ambient occlusion for a given G-Buffer in the SSAO shader. A TBN (2.1.1) matrix is generated based on the screen space position, screen space normal, and random vector. The TBN matrix is used to transform the hemisphere position read from the kernel to the correct position scaled using a radius constant. The transformed hemisphere position is the position that is used to sample from the G-Buffer position texture. The sample position is projected and transformed further into uv coordinates (2.2.2) and used to sample the position texture for the Z value or depth at these coordinates. If the sampled depth value is greater then the sampled position depth plus a bias term 1 is added to an occlusion accumulator. Otherwise it is 0. This value that is stored is also augmented by a range check. In case the difference of the sample depth and the depth of the position for which ambient occlusion is calculated is greater than the radius, the sample's factor is reduced so as to not affect the final result. The occlusion

samples are summed and normalized and finally raised to a user defined power to increase the visual effect. This process can be seen in more detail in the pseudocode in 4.20

After generating the SSAO result is blured in a similar way with vsm 4.3.2 with separable filters as described in 2.5.2 with a radius of 1 and $\sigma = 2.4$.

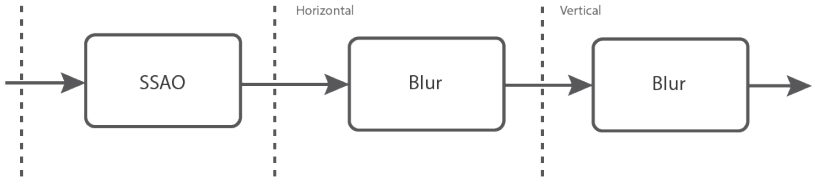


Figure 4.21: Ambient Occulusion stage flow chart.

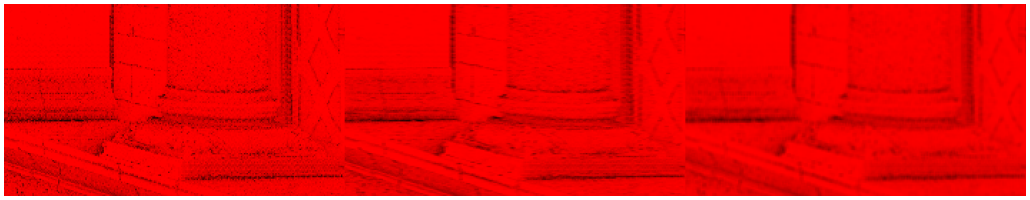


Figure 4.22: Comparison each step output render target. First no bluring. Second horizontal bluring. Third horizontal + vertical blur.

4.3.6 Lighting

Having calculated all of the above, performing lighting calculations is fairly simple. A full screen shader runs for every pixel in the framebuffer. Position, normal, Lambertian reflectivity, and emissive value are read from the GBuffer to a surfel structure. Using the information from the VSM stage, the surfel, and the incidence direction, direct illumination is calculated as seen in 4.3.2. Having calculated direct illumination, indirect illumination is calculated using the same information by sampling the global illumination volume as seen in 4.3.4. Reading the Ambient Occlusion value for the pixel being rendered, the final color is calculated as seen bellow:

```
o_color = vec4(s.emissive + direct_illumination * AO
          + indirect_illumination * AO, 1.0);
```

Where directAO is a smaller value for AO since direct illumination is less affected by ambient occlusion.

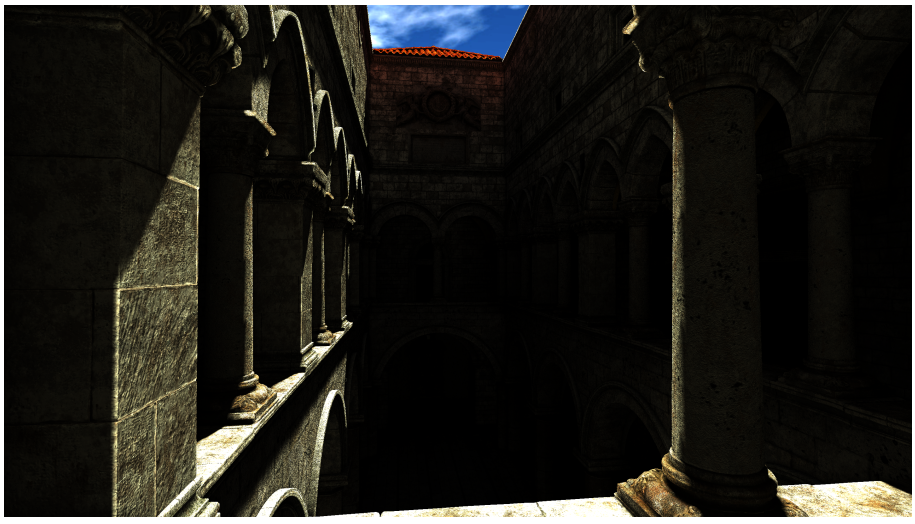


Figure 4.23: This is the resulting HDR image of the Lighting stage.

4.4 Path Tracer

In this section, we describe the path tracer implementation used in this thesis for reference image generation. As discussed in 2.4.3 this path tracer is a Monte Carlo path tracer with Russian roulette ray termination. This path tracer shares the scene and post-process layers with the Hybrid renderer for visual parity.



Figure 4.24: Overview of the Path Tracer reference renderer. These are the layers active during path tracing.

The layers seen in 4.24 are run sequentially in every frame. The path tracing layer contains the following resources that the initialize () function generates: a) An avk:: ray_tracing_pipeline 3.1.6 for tracing rays that will be described later on. b) Two R32G32B32A32Sfloat avk::image_sampler to store the resulting image and the accumulated radiance. c) Finally, an avk::semaphore that is singled when all GPU commands emitted from this layer are executed. The update() function has little work to do, as the acceleration structure and all other resources are updated in the **Scene** layer. The render() function executed a command that binds all resources that are going to be used in the ray tracing pipeline, it binds the ray tracing pipeline, and it traces rays with dimensions screen_width, screen_height. In essence, every frame the path tracer layer traces 1 ray per pixel and captures one sample of the rendering equation 2.3. Since this path tracer incorporates virtual lights that do not have physical dimensions, shadow rays are incorporated in the rendering equation to calculate the direct light hitting the surface. Thus, the formulation of the rendering equations that is used is the following:

$$L_o(\mathbf{x}, \vec{\omega_o}, \lambda, t) = \sum^{N} L_e(\mathbf{x}, \vec{\omega_o}, \lambda, t) + L_d(\mathbf{x}, \vec{\omega_o}, \lambda, t) + \frac{f_r(\mathbf{x}, \vec{\omega_i}, \vec{\omega_o}, \lambda, t)(\vec{\omega_i} \cdot \vec{n})}{p(\vec{\omega_i})} L_i(\mathbf{x}, \vec{\omega_i}, \lambda, t) \quad (4.4)$$

Where the new terms $L_d(\mathbf{x}, \vec{\omega_o}, \lambda, t)$ is the direct illumination reaching point $\mathbf{x}$ based on 4.2.5. The term $\frac{f_r(\mathbf{x}, \vec{\omega_i}, \vec{\omega_o}, \lambda, t)(\vec{\omega_i} \cdot \vec{n})}{p(\vec{\omega_i})}$ is referred to as the multiplier.

The ray tracing pipeline follows the following flow chart 4.25 on the GPU.

4.4.1 Ray Generation

The recursive part of the rendering equation described in 4.4 is implemented as follows. The ray generation shader runs for every ray that is traced from the ray tracing pipeline. It initially calculates the primary rays that will be traced from the camera for every pixel. While the max bounces limit has not been reached, continue tracing rays. For every ray the hit_info struct is filled with the information necessary to trace the rays. The cullMask is set to cull all intersections that hit light sources. Since we are handling direct illumination with shadow rays, these intersections are not needed. The rays are traced and the flow continues to other shaders. When the trace is complete, the returning radiance and multiplier are saved. If the termination criterion is reached, the ray may terminate early without reaching max_bounce through Russian roulette termination. This stops further secondary ray generation. Otherwise, the secondary ray that was generated by the Closest Hit shader is setup for tracing. When this process is completed, the radiance calculated is combined with the accumulated radiance from previous traces. The result is scaled according to the number of samples and is stored in the image.

4.4.2 Closest Hit

If rays traced in the ray generation shader hit geometry, the closest hit is run on the closest intersection point. This shader is responsible for calculating the rest of the rendering equation and the hits' radiance. Initially vertex and index data along with barycentric coordinates are used to calculate the hit position, normal, uv coordinate, tangent, and bitangent. Using the material and texture array, the normal and diffuse color of the point are further specified. The view direction, scatter direction, BRDF, and PDF are also calculated 4.2.2. The first two components of 4.4 are calculated initially and added to the radiance sum, as seen in 4.27, multiplied by the

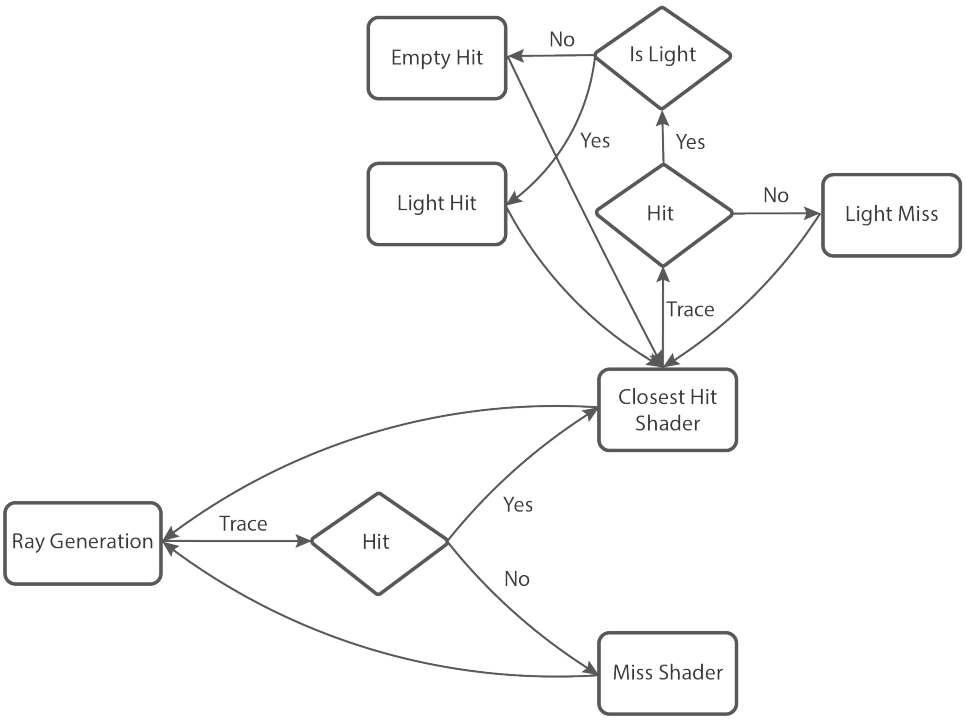


Figure 4.25: Path tracing step flow chart.

multiplier of previous rays. The two components being the materials emissive component and the direct illumination calculated by diret_illumination as described in 4.4.3. Afterwards the first termination criterion is considered. If the PDF at this point is less then equal to zero the ray is terminated. Then, the multiplier is updated with the new value according to 4.4. Finally, Russian roulette calculations take place. These are only active if the current ray depth is greater than 2 or if every component of multiplier is too small, less then 0.01 in this case. Even then, that does not guarantee that the ray will terminate here. A probability q is calculated according to which the ray will be terminated. If the ray does not terminate, the multiplier is reduced by a fraction of itself. This ensures that the probability q will be higher on the next ray and as a result the ray will probably terminate. The scatter direction and position of the hit is returned to continue tracing secondary rays or the terminated flag is set to true to terminate the ray. The ray's radiance is also returned to the ray generation shader.

4.4.3 Direct Light

This path racer uses shadow rays to calculate direct illumination from virtual light sources. Since virtual lights do not have physical dimensions in space, shadow rays are traced to really small bounding boxes that are used to check for visibility with the source. One of the benefits of sampling the approach of path tracing is that direct illumination does not need to be calculated for each light source. It can be calculated for a random light source for every sample. Since samples are averaged, the final result will have every light's contribution in the final image. First the random light source, and the direction to the light source is calculated. As seen in 4.28, shadow rays are traced from the position and direct illumination is calculated toward the light source. The cull mask is set to 0xFF. This means that no hits will be culled. So if the shadow ray hits a light source, the is_shadowed variable will be set to false. If the sky (miss shader) or any other geometry (empty hit) is hit, the variable is set to true. If not in shadow, the light's contribution is calculated. The lights attenuation 4.2.5, multiple importance weight [PJH16], material BRDF 4.2.2, and light PDF 4.2.5 are calculated. These are all factored into the light's radiance that is directed towards the point as seen in the pseudo code 4.28.

```
for (uint i = 0; i < max_bounces; i++) {
    // Hit everything but light sources
    uint cullMask = 0x7f;
    // This is call is shortened for brevity
    traceRayEXT(cullMask, rayOrigin, rayDirection);
    radiance = hit_info.radiance;
    if (hit_info.terminate) {
        break;
    }
    rayOrigin = hit_info.hit_position;
    rayDirection = hit_info.scatter_direction;
}
accumulatedRadiance += radiance;
// Final radiance of pixel
// Scaled based on the number of samples
radiance = accumulatedRadiance * scale;
```

Figure 4.26: Pseudo code excerpt from the ray generation shader of the path tracer layer

4.4.4 Miss

Finally, the primary and secondary rays miss shader is very straight forward. It is triggered when the ray hits the sky box. In this case, the sky boxes diffuse color at the direction that is treated as the emissive component of 4.4. The ray cannot scatter any further and is terminated.

4.5 Post Processing

The post-processing stage is the final step before presenting the finished render to the screen. It is shared by both the hybrid renderer and the path tracer to ensure visual consistency. This stage consists of two steps. First, tone mapping and gamma correction are applied to the lit HDR result from the lighting stage (see 4.3.6). Second, anti-aliasing is performed. After these steps, the finished frame is presented to the screen.

Tone Mapping and Gamma correction

This step uses a fullscreen fragment shader that performs standard gamma correction (see 2.5.5) and Reinhard tone mapping (see 2.5.4). The result can be adjusted by changing the γ term (see 2.9) and by modifying a sensitivity term that is premultiplied before tone mapping is applied. Tone mapping is performed first, followed by gamma correction. An example of the output from this step is shown in Figure 4.30.

Anti-aliasing

In this step, anti-aliasing is performed to ensure that the final result has crisp edges without jagged edges, as described in 2.5.1. Two methods are used:

1. Super-Sample Anti-Aliasing (SSAA). In this technique, the renderer operates at two resolutions: an internal (higher) resolution and the final presentation resolution. When it is time to display the final image, the higher-resolution frame is downsampled (using bilinear filtering) to the presentation resolution. Although rendering at a higher resolution increases the cost for each draw call tied to this resolution, it was necessary in our case to ensure that both renderers produce comparable visual results.

2. Fast Approximate Anti-Aliasing (FXAA). This is a screen-space technique that analyzes the final image and applies smoothing to edges [Tim09]. The details of this implementation are out of the scope of this thesis, as it is taken directly from NVIDIA. Examples of FXAA in action can be seen in Figures 4.31 and 4.32.

```
// View direction
vec3 V = -gl_WorldRayDirectionEXT;
// Scatter direction
vec3 L = Sample(material, N, V, hit_info.seed);
hit_info.radiance += (material.emissive_color.rgb
    + direct_light(N, position, material, hit_info.seed))
    * hit_info.multiplier;
if (pdf <= 0.0) {
    hit_info.terminate = true;
    return;
}
hit_info.multiplier *= (cosTheta * brdf) / (pdf + EPSILON);
// Russian roulette
if (max_component(hit_info.multiplier) < 0.01 && hit_info.depth > 2) {
    float q = max(float(.05), 1. - max_component(hit_info.multiplier));
    if (random_float(hit_info.seed) < q) {
        hit_info.terminate = true;
    }
    hit_info.multiplier /= (1. - q);
}
```

Figure 4.27: Pseudo code excerpt from the closest hit shader of the path tracer layer.

```
vec3 direct_light(normal, position, material) {
    int index = int(random_float(seed) * float(lightsource_count));
    l = light_array[index];
    if (l.lightsource.info[0] == lightsource_type_directional) {
        lightDirection = -l.lightsource.direction.xyz;
    }
    else {
        lightDirection = l.lightsource.position.xyz - position;
    }
    traceRayEXT(0xFF, position, lightDirection);
    if (!is_shadowed) {
        L = attenuation * misWeight
            * cosTheta * l.lightsource.color.rgb
            * brdf / (lightPdf + EPSILON);
    }
    return L;
}
```

Figure 4.28: Pseudo code from the direct illumination function.

```
hit_info.radiance += texture(skybox_texture,
    hit_info.scatter_direction).rgb
    * hit_info.multiplier;
hit_info.terminate = true;
```

Figure 4.29: Pseudo code form the miss shader.



Figure 4.30: This is the result of the tone mapping and gamma correcting step.



Figure 4.31: This is the result of the AA step applying FXAA to the final frame.



Figure 4.32: This is the result of the FXAA step.

Chapter 5

Analysis

5.1 Overview

Testing and evaluating a global illumination algorithm is inherently challenging. Modern 3D renderers involve a wide range of visual and performance characteristics that can be quantified, such as frame time (ms), memory usage, and image fidelity metrics. In addition to these objective measures, subjective factors such as perceived realism or aesthetic quality play an equally important role in determining whether a particular technique is suitable for a given application.

In this chapter, a thorough analysis of the hybrid renderer described in 4.3 will be presented. The hybrid renderer is compared with a path-traced renderer described in 4.4. The two renderers are compared on visual quality. Finally, performance metrics are presented for both renderers but only as a reference.

5.2 Methodology

Both renderers are tested using the new Intel Sponza [Mei+22] as a scene. This scene is a remastering on an old classic graphics research scene. The new one, created by Intel, has 2027489 vertices that are used to compose 3738442 triangles. This scene also uses 41 number of materials that in turn use 71 4096 by 4096 pixel textures. Note that only diffuse and normal textures are used in this thesis. The scene also defines 6 cameras. This provides us with standardized viewports to use when rendering between engines. The scene is further processed with Blender 3D by adding the directional light and defining the directional light's camera for generating shadow maps. The scene is output from Blender 3D to a gltf file (3.2) and a json file is generated with extra information that is needed for both renderers.

The two image quality metrics used for comparing the generated images to the reference result are the peak signal-to-noise ratio, PSNR, and structured similarity index measure or SSIM. PSNR is an error summation technique, which means that the absolute error is calculated and presented in this metric. To be more specific PSNR is defined as follows:

$$PSNR(f,g) = 10\log\left(\frac{255^2}{MSE(f,g)}\right)$$

where

$$MSE(f,g) = \frac{1}{NM}\sum_{i=1}^{M}\sum_{j=1}^{N}(f_{ij} - g_{ij})$$

where f is a reference picture and g is the test image. Both of dimensions MxN. PSRN is logarithmic metric measured in dB. SSIM, as introduced in [WSB03], on the other hand is a perception-based model trying to measure what the human eye can perceive instead of the absolute error. It is defined as follows:

$$SSIM(f,g) = l(f,g)c(f,g)s(f,g)$$

The three comparison measurements l contrast, s structure, c contrast are defined as such:

$$l(f,g) = \frac{2\mu_f\mu_g + c_1}{\mu_f^2 + \mu_y^2 + c_1}$$
$$c(f,g) = \frac{2\sigma_f\sigma_g + c_2}{\sigma_f^2 + \sigma_g^2 + c_2}$$
$$s(f,g) = \frac{\sigma_{fg} + c_3}{\sigma_f\sigma_g + c_3}$$

Finally the constants used in the above formula are defined here:

- μ_f the pixel sample mean of f.
- μ_g the pixel sample mean of g.
- σ_f^2 the variance of f.
- σ_g^2 the variance of g.
- $\sigma_f g$ the covariance of f and g.
- $c_1 = (k_1 L)^2$, $c_2 = (k_2 L)^2$, $c_3 = \frac{c_2}{2}$ variables used to stabilize division with weak denominators.

- $k_1 = 0.01$, $k_2 = 0.03$ by default.
- L is the dynamic range of the pixel-values.

Although SSIM is a more useful metric for this thesis as it measures similarity as perceived by humans, and the final result produced is something destined for the human eye. Both metrics are used for a more complete analysis.

When comparing renderers for visual quality the path tracer is run for 10000 samples. Both renderers use SSAA and render at 2880 by 1620 that is then scaled down to 1920 by 1080 image. While the normal AA method for path tracers is multisampling, since using this technique with the hybrid renderer would be more difficult, this approach was chosen. The output frames are then converted to grayscale compared using PSNR and SSIM. The resolution of the Shadow map for the hybrid renderer is 4096 by 4096.

When it comes to performance, general performance metrics per-stage are gathered for each renderer. Comparing milliseconds per frame between the path-tracer and hybrid renderer would not provide any useful information, as one is inherently designed to run in real time and the other one is not. It is useful though to frame the results with some performance metrics. To capture performance metrics Vulkan Query Pool's are used. More in [Gro24].

The machine used to generate both visual and performance results is running an AMD Ryzen 2700 CPU with 24 GB RAM. The GPU used is the NVIDIA GeForce RTX 3060 with 12 GB of VRAM and all this runs on Windows 11.

5.3 Results

In this section, we discuss the visual and quantitative results of this thesis. The results generated are mainly visual, and could not all be include in this section. The entire collection of captured frames can be found in Appendix A.

Overall, the visual results generated by the Hybrid renderer are visually comparable to the results of the reference path-tracer. This is supported by the SSIM metrics presented in Figure 5.11. A player exploring a world renderered from the Hybrid renderer would be presented with a visually appealing result. Despite that, some differences and visual artifacts can be seen on closer inspection. PSNR metrics do not indicate great visual similarity. This is expected as PSNR is value-based metrics and cannot accurately capture visual similarities like SSIM.

One category of visual errors is caused by probe placement. The probes are initially placed on a grid and automatically moved to best illuminate the scene as described in Section 4.3.4. This can sometimes cause poor probe placement or not cover some cases at all. Two examples of this artifact can be seen in Figures 5.1 and 5.2. In the case seen in Figure 5.1 the area not lit is a small opening in the wall. It is probable that the probe position algorithm detected it as a point inside geometry and moved or disabled the probe in question. In the case of Figure 5.2, indirect light is not captured fully on the column again because of missing probe information in the area. Only when the probe density is at 32 by 32 by 32 is this alleviated. This is not a realistic scenario for this scene as the amount of probes take a significant amount of time to update as seen in Figure 5.13.



Figure 5.1: An area that is not covered by probes and is not lit in View 1. From left to right path-traced reference, 8x8x8, 16x16x16, and 32x32x32 probe count.

Another category of visual error present is light bleeding. This appears when extra light is drawn in areas where it shouldn't. This can usually be seen in geometry with corners. One texbook example of this artifact can be seen in Figure 5.4. The Dynamic Diffuse Global Illumination technique covers most cases and in general as seen in the results performs well. In this case depending again on probe position it can fail. Another case where extra light information is present is in Figure 5.3. The red tiles of the roof are correctly and realistically scattered on the brick of the walls, but in some cases and especially around the ornate wall decoration it causes



Figure 5.2: Indirect illumination differences based on probe positioning/density in View 6. From left to right path-traced reference, 8x8x8, 16x16x16, and 32x32x32 probe count.



Figure 5.3: Similarities in indirect light form roof in View 2. From left to right path-traced reference, 8x8x8, 16x16x16, and 32x32x32 probe count.

light bleeding. In general, this case is very close to the reference and can also be considered a good result of this case.

Visual errors can also appear in direct illumination not just indirect. The variance shadow map technique does create visually appealing shadows by bluring the shadow map, as described in Section 4.3.2, but differences can be seen with the shadow generated with the reference path-tracer. These can be seen in the Figures 5.5, 5.6, and 5.7.

A final class of errors is present in the path-tracer. This is a floating-point accuracy that causes rays to bounce outside of geometry. This causes bright areas where they should not be. When the next ray to be traced is calculated, small errors caused by floating-point precision can cause the ray to spawn behind geometry. That ray is then traced and can cause an intersection with geometry that is not in the correct path for that ray that is lit by other sources of light. The extra light is factored in the final ray radiance causing the visual artifact. This phenomenon is covered in more detail in the Book *Physically Based Rendering: From Theory to Implementation* [PJH16]. In this thesis, these artifacts can be seen in the following Figures: 5.8, and 5.9.

In general, through the SSIM and PSNR metrics seen in Figure 5.11 we can see that the increase in probe density does not correlate with an increase in visual quality and can reduce the SSIM metric in some cases. Also when controlling for direct illumination the Dynamic Diffuse Global Illumination technique is proven even more capable for producing global illumination. In view 4 in particular which is not lit by any direct illumination a high comparison metric is observed. When observing only indirect illumination frames the differences mainly causes by DDGIs' inability to capture first bounce light. For example, light that bounces off the edge of the arch after the first hit to the wall is not captured with DDGI, as seen in 5.10.

Another aspect we evaluated is performance. Starting with the performance numbers per stage, as seen in Figure 5.12, we can see that the longest time is taken by the GBuffer and Shadow Mapping stages. This is expected as they are the most geometry-heavy operations and scale with the geometry in the scene. Another outlier is the SSAO technique. This technique costs 7.6 ms when rendered at 2880 by 1620. This is expected as it is a screen space technique followed by a two-pass blur of the entire framebuffer. The performance of this technique is directly tied to screen resolution; thus in higher resolutions, we observe reduced performance. The final lighting pass that combines and shades the GBuffer is also resolution-dependent but scales better than SSAO. SSAA was used in this thesis to ensure visual parity between path-traced and hybrid renderer results. Thus, it is not the expected use case to use SSAA in an actual engine, although it is not uncommon. Although SSAA might not be common on most renderers, higher than 1080p resolution is, making the evaluation of the renderer in higher resolutions important. We present both results for completeness.

In addition, the performance results of the DDGI technique can be seen in Figure 5.13. These performance metrics are impressive. Even with just the 8x8x8 technique with only 0.5 ms cost, the global illumination produced is easily comparable to that of our path tracing. Even with a higher density with 16x16x16 probes, the 2.9 ms time needed to generate global illumination is an acceptable cost for most engines. The 32x32x32 is a very costly scenario, but not realistic. It is mainly shown to illustrate how the technique scales with probe density.

Finally, the overall results are presented in Figure 5.14. We can see the overall GPU time taken



Figure 5.4: Light bleed artifacts in hybrid renderer results based on probe position in View 4. From left to right path-traced reference, 8x8x8, 16x16x16, and 32x32x32 probe count.



Figure 5.5: Differences in direct illumination in View 6. From left to right path-traced reference, 8x8x8, 16x16x16, and 32x32x32 probe count.



Figure 5.6: Direct illumination differences in shadows in View 1. From left to right path-traced reference, 8x8x8, 16x16x16, and 32x32x32 probe count.



Figure 5.7: Differences in direct illumination in View 2. From left to right path-traced reference, 8x8x8, 16x16x16, and 32x32x32 probe count.

to render a single frame in various configurations. In general, depending on the configuration somewhere between 12 ms and 20 ms, realistic results can be generated with minimal visual artifacts. Finally, the path tracer implementation took 52 ms for each sample per pixel, which is still very impressive considering that rays are path-traced through the scene. For the final frames used for the comparison, we used 10000 samples per pixel. This meant that each frame took around 9 minutes to generate. Although the frame-time results are not comparable, it helps illustrate the time difference versus the small differences in quality between the hybrid and path-traced results.

The use of more rays per probe was evaluated as seen in Section A.4. By comparing results for 256 rays per probe, which is the default against 512 rays, no significant visual difference was observed. This is because rays are randomly assigned in every frame. When the probe volume has converged, most angles around the probe have been covered. More rays help the probe volume converge faster at the cost of performance. However, this scenario is not covered in this thesis.



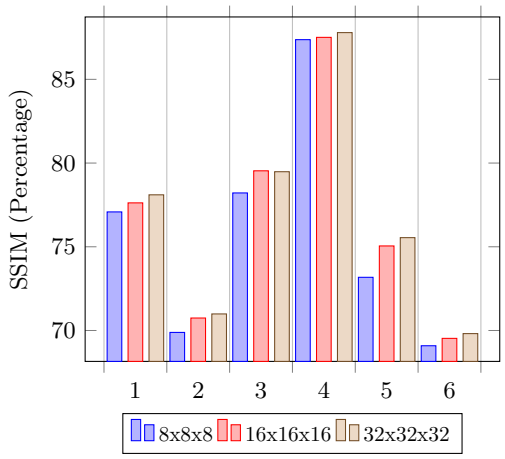
Figure 5.8: Float precision issues with path-traced reference in View 3. From left to right path-traced reference, 8x8x8, 16x16x16, and 32x32x32 probe count.



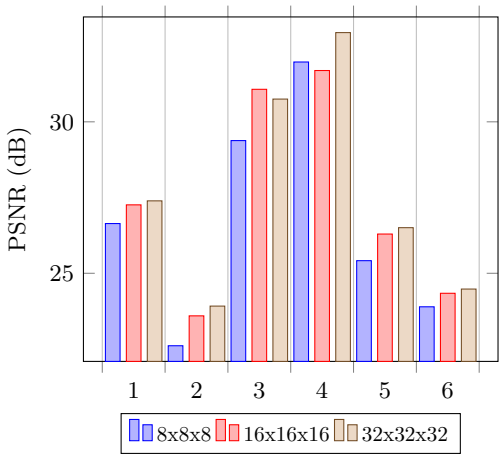
Figure 5.9: Indirect lighting on brick surface in View 4. From left to right path-traced reference, 8x8x8, 16x16x16, and 32x32x32 probe count.



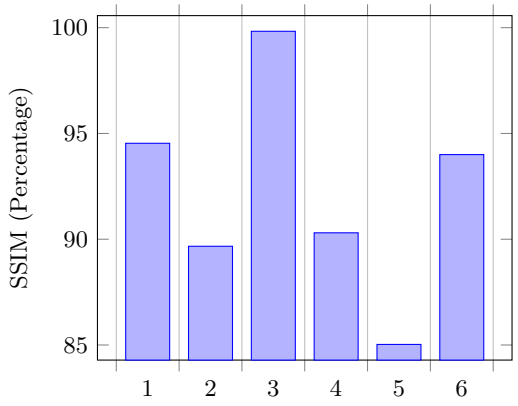
Figure 5.10: Only indirect light differences in View 1. From left to right path-traced reference, 8x8x8, 16x16x16, and 32x32x32 probe count.



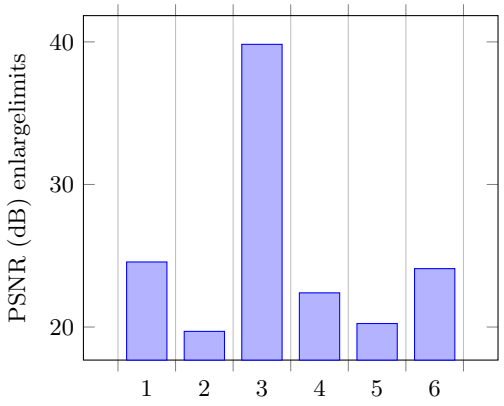
(a) SSIM comparison of full illumination between path traced and hybrid renderer results



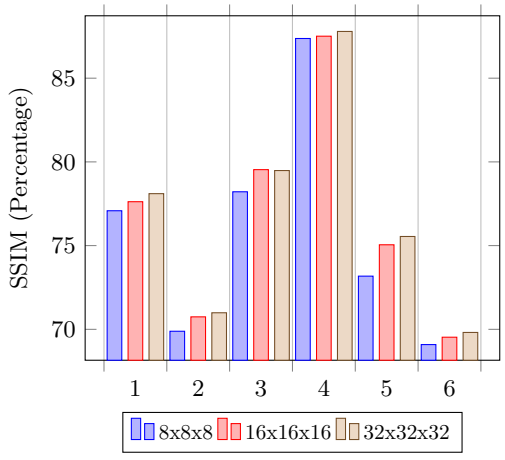
(b) PSNR comparison of full illumination between path traced and hybrid renderer results



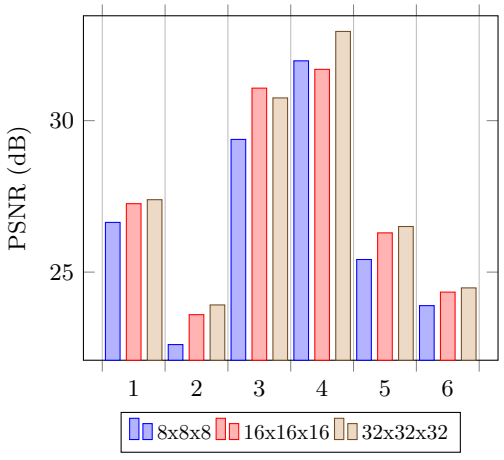
(c) SSIM comparison of direct illumination between path traced and hybrid renderer results



(d) PSNR comparison of direct illumination between path traced and hybrid renderer results



(e) SSIM comparison of indirect illumination between path traced and hybrid renderer results



(f) PSNR comparison of indirect illumination between path traced and hybrid renderer results

Figure 5.11: Quantitative analysis results for various configurations and probe counts

Stage	Average (ms) with SSAA	Average (ms) with FXAA
GBuffer	4.94	3.02
Mesh	4.92	3.02
Skybox	0.006	0.005
Post-Process	0.13	0.18
AA (only FXAA)	0.0	0.11
Tone Mapping	0.13	0.065
Shadow Mapping	3.25	3.17
Depth Pass	1.72	1.74
Blur	1.53	1.43
SSAO	7.6	2.39
Ambient Occlusion	7.22	2.02
Blur	0.38	0.19
Light Pass	2.39	1.08

Figure 5.12: Performance results per technique.

	Average (ms)		
Probe count	8x8x8	16x16x16	32x32x32
Total	0.49	2.86	20.8
Trace Rays	0.19	1.07	5.55
Shade Rays	0.14	0.98	10.77
Update Probes	0.15	0.75	4.03
Copy Probe Edges	0.008	0.04	0.41

Figure 5.13: DDGI performance results.

	Average (ms)		
Probe count	8x8x8	16x16x16	32x32x32
FXAA	11.8	15.83	40.46
SSAA	20.2	23.95	56.66
Parh-Tracer	51.14		

Figure 5.14: Overall performance average.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This thesis presents our research on and the implementation of a hybrid renderer. Our work demonstrates an example of how a hybrid approach can be integrated into a conventional 3D renderer, highlighting its potential to produce realistic high-quality images in real-time. Our renderer can generate a full frame in 12 ms to 20 ms, depending on the configuration. The visual result of this hybrid renderer is directly comparable to the path tracer, achieving anywhere between 70% and 90% SSIM index, depending on the current view. The key visual differences between this hybrid renderer and the path tracer are some visual artifacts like light leaking and the difficulty in producing second bounce indirect illumination. Although these visual artifacts are present —as discussed in detail in Section 5.3— the renderers output remained broadly comparable to that of a path-traced reference. Other drawbacks observed in this implementation include the poor scaling with resolution. Notably, the SSAO implementation shows the poorest performance and has room for improvement in higher resolutions. Overall, these results are satisfactory and align well with the visual and performance requirements of real-time applications like modern video games and extended reality applications.

With the ever-expanding ray-tracing capabilities of modern GPU hardware, even present in some mobile devices, hybrid rendering offers exciting possibilities. Ongoing research continues to refine techniques that combine rasterization and ray tracing, leveraging both to create more realistic and performant visual effects. Our results underscore the potential of hybrid rendering to bridge the gap between the high-fidelity visuals typically associated with ray tracing and the efficiency of rasterization. In particular, the global illumination technique showcased here is notable for delivering convincing indirect lighting without incurring excessive rendering costs.

Hybrid techniques can have a broad impact and not just affect video game renderers. Realistic rendering is in high demand for many applications and industries. Example uses include architectural visualization, CAD/CAE, training simulations, and virtual reality environments. By offering a fast, yet accurate approximation of light behavior, hybrid rendering can help inform design decisions and create immersion in virtual spaces. This demand, coupled with the availability of hardware accelerated ray tracing hardware poses an immense opportunity for further research and innovation.

6.2 Future Work

This work is intrinsically constrained by time, as are all projects and thus various extensions were considered but not implemented in the end. It was originally envisioned as a comparison between hybrid, path-traced, and classic non-hybrid rendering techniques for global illumination. A third technique for global illumination could be implemented, such as voxel-based global illumination as described in [Cra+11]. This technique is based on hierarchical voxel octree representation of the scene that is generated and updated for the scene on-the-fly. This acceleration structure can be used to perform cone tracing inside the scene and approximate low-frequency indirect illumination. This technique was further expanded by NVIDIA in [NVI] and presents good results in various scenarios. This comparison would give a more complete picture of the current landscape for global illumination techniques. As mentioned before, not all GPUs support hardware-accelerated, and thus not all users have access to the hardware to be able to run hybrid techniques. When choosing a global illumination technique it is also important to compare these factors.

Another avenue for expansion is by implementing improvements to the base DDGI technique. The technique implemented here is based only on the [Maj+19] and some aspects of [Maj+21] and does not provide a full picture of its possible extensions of the technique. Notable omissions from this implementation of DDGI is responded to changes in scene lighting and geometry. This can be done with engines more advanced than the one that was implemented in this thesis. Changes in light and geometry that affect a probe can cause that probe to be set to a state that traces more rays and has a lower hysteresis value to update illumination information faster. While working on this thesis, various expansions to the original technique have also been proposed. Possible expansions to evaluate include importance-based ray allocation techniques, such as those proposed in [Liu+23]. In addition, adaptive importance sampling of probes using heuristic methods to allow more dense probe placement without affecting performance, as described in [DGZ23].

It would also be valuable to compare DDGI with other hybrid techniques for global illumination. One such technique could be Surfel-Based Global Illumination described in [Hal+21]. This technique discretizes the scene to calculate global illumination with surfels on the surface being rendered. Another technique that uses radiance caching is the one described in [Wri21]. The field

of hybrid techniques is new and fast evolving; thus comparing techniques would be valuable to those selecting a hybrid technique for a renderer.

Finally, the hybrid renderer can also be further improved. The main area that needs improvement is the generation of ambient occlusion. The current implementation of SSAO is very costly and does not scale well with resolution. Also, SSAO based techniques fail to capture occlusion for objects that are not visible in the current frame due to its screen space nature. Other ray-traced techniques such as the one described by Gautron in [Gau20] could be applied for better results. This technique uses spatial hashing to store and cache ambient occlusion samples calculated with ray tracing. This scheme can then be used to efficiently sample the data structure to produce ambient occlusion. In addition to ambient occlusion, other improvements can also be made. Additional support for more light types like area light sources and omni-directional light sources is missing form this renderer implementation.

Bibliography

[AHH18] Tomas Akenine-Mller, Eric Haines, and Naty Hoffman.
Real-Time Rendering, Fourth Edition. 4th. USA: A. K. Peters, Ltd., 2018.
ISBN: 0134997832.

[Ben75] Jon Louis Bentley.
"Multidimensional binary search trees used for associative searching".
In: *Commun. ACM* 18.9 (Sept. 1975), pp. 509–517. ISSN: 0001-0782.
DOI: 10.1145/361002.361007. URL: https://doi.org/10.1145/361002.361007.

[Bus03] Samuel R. Buss.
3D Computer Graphics: A Mathematical Introduction with OpenGL.
USA: Cambridge University Press, 2003. ISBN: 0521821037.

[Cra+11] Cyril Crassin et al. "Interactive Indirect Illumination Using Voxel Cone Tracing".
In: *Computer Graphics Forum* 30.7 (2011), pp. 1921–1930.
DOI: https://doi.org/10.1111/j.1467-8659.2011.02063.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2011.02063.x.
URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2011.02063.x.

[DGZ23] Sayantan Datta, Negar Goli, and Jerry Zhang.
"Adaptive Dynamic Global Illumination".
In: *arXiv preprint arXiv:2301.05125* (2023).

[DL06] William Donnelly and Andrew Lauritzen. "Variance Shadow Maps".
In: *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*.
I3D '06. Redwood City, California: Association for Computing Machinery, 2006, pp. 161–165. ISBN: 159593295X. DOI: 10.1145/1111411.1111440.
URL: https://doi.org/10.1145/1111411.1111440.

[Fer04] Randima Fernando.
GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics.
Pearson Higher Education, 2004. ISBN: 0321228324.

[Gau20] Pascal Gautron. "Real-Time Ray-Traced Ambient Occlusion of Complex Scenes using Spatial Hashing". In: *ACM SIGGRAPH 2020 Talks*. SIGGRAPH '20.
Virtual Event, USA: Association for Computing Machinery, 2020.
ISBN: 9781450379717. DOI: 10.1145/3388767.3407375.
URL: https://doi.org/10.1145/3388767.3407375.

[Gro18] Kronos Group. *KHR_lights_punctual*.
The Khronos® 3D Formats Working Group. June 2018.
URL: https://github.com/KhronosGroup/glTF/blob/main/extensions/2.0/Khronos/KHR_lights_punctual/README.md.

[Gro21] Kronos Group. *glTF™ 2.0 Specification*. Version 2.0.1.
The Khronos® 3D Formats Working Group. Oct. 2021.
URL: https://registry.khronos.org/glTF/specs/2.0/glTF-2.0.html.

[Gro24] Kronos Group. *Vulkan® 1.3.288 - A Specification*. Version 1.3.288.
The Khronos® Vulkan Working Group. June 2024.
URL: https://registry.khronos.org/vulkan/specs/1.3/html/vkspec.html.

[Hal+21] Henrik Halen et al. "Global Illumination Based on Surfels".
SIGGRAPH: Advances in Real-Time Rendering in Games. 2021. URL: https://advances.realtimerendering.com/s2021/SIGGRAPH%20Advances%202021%20-%20Surfel%20GI.pdf.

[Kaj86] James T. Kajiya. "The Rendering Equation". In: *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '86. New York, NY, USA: Association for Computing Machinery, 1986, pp. 143–150. ISBN: 0897911962. DOI: 10.1145/15922.15902. URL: https://doi.org/10.1145/15922.15902.

[Kap19] Arseny Kapoulkine. "Writing an Efficient Vulkan Renderer". In: *GPU Zen 2: Advanced Rendering Techniques*. Ed. by W. Engel. GPU Zen Series. Independently Published, 2019. Chap. 4, pp. 215–248. ISBN: 9781797583143.

[Kei+15] Benjamin Keinert et al. "Spherical Fibonacci Mapping". In: *ACM Trans. Graph.* 34.6 (Nov. 2015). ISSN: 0730-0301. DOI: 10.1145/2816795.2818131. URL: https://doi.org/10.1145/2816795.2818131.

[Kop14] Sanjeev J. Koppal. "Lambertian Reflectance". In: *Computer Vision: A Reference Guide*. Ed. by Katsushi Ikeuchi. Boston, MA: Springer US, 2014, pp. 441–443. ISBN: 978-0-387-31439-6. DOI: 10.1007/978-0-387-31439-6_534. URL: https://doi.org/10.1007/978-0-387-31439-6_534.

[Liu+23] Zihao Liu et al. "Importance-Based Ray Strategies for Dynamic Diffuse Global Illumination". In: *Proc. ACM Comput. Graph. Interact. Tech.* 6.1 (May 2023). DOI: 10.1145/3585500. URL: https://doi.org/10.1145/3585500.

[Maj+19] Zander Majercik et al. "Dynamic Diffuse Global Illumination with Ray-Traced Irradiance Fields". In: *Journal of Computer Graphics Techniques (JCGT)* 8.2 (June 2019), pp. 1–30. ISSN: 2331-7418. URL: http://jcgt.org/published/0008/02/01/.

[Maj+21] Zander Majercik et al. "Scaling Probe-Based Real-Time Dynamic Global Illumination for Production". In: *Journal of Computer Graphics Techniques (JCGT)* 10.2 (May 2021), pp. 1–29. ISSN: 2331-7418. URL: http://jcgt.org/published/0010/02/01/.

[McG+17] Morgan McGuire et al. "Real-Time Global Illumination Using Precomputed Light Field Probes". In: *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D '17. San Francisco, California: Association for Computing Machinery, 2017. ISBN: 9781450348867. DOI: 10.1145/3023368.3023378. URL: https://doi.org/10.1145/3023368.3023378.

[McG17] Morgan McGuire. *Computer Graphics Archive*. https://casual-effects.com/data. July 2017. URL: https://casual-effects.com/data.

[Mei+22] Frank Meinl et al. *Intel Sample Library*. https://www.intel.com/content/www/us/en/developer/topic-technology/graphics-processing-research/samples.html. 2022.

[Mey+10] Quirin Meyer et al. "On Floating-Point Normal Vectors". In: *Computer Graphics Forum* 29.4 (2010), pp. 1405–1409. DOI: https://doi.org/10.1111/j.1467-8659.2010.01737.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2010.01737.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2010.01737.x.

[Mit07] Martin Mittring. "Finding next Gen: CryEngine 2". In: *ACM SIGGRAPH 2007 Courses*. SIGGRAPH '07. San Diego, California: Association for Computing Machinery, 2007, pp. 97–121. ISBN: 9781450318235. DOI: 10.1145/1281500.1281671. URL: https://doi.org/10.1145/1281500.1281671.

[Nik23] Lisitsa Nikita. *Two-pass Gaussian blur coeffifients generator*. 2023. URL: https://lisyarus.github.io/blog/graphics/2023/02/24/blur-coefficients-generator.html (visited on 02/24/2023).

[NVI] NVIDIA. *NVIDIA VXGI*. URL: https://developer.nvidia.com/vxgi.

[NVI18] NVIDIA. *Nvidia Turing GPU Architecture*. Tech. rep. 2018.
URL: https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[Ove15] Alexander Overvoorde. *Vulkan Tutorial*. 2015.
URL: https://vulkan-tutorial.com/ (visited on 05/06/2023).

[PJH16] Matt Pharr, Wenzel Jakob, and Greg Humphreys.
Physically Based Rendering: From Theory to Implementation. 3rd.
San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2016.
ISBN: 0128006455.

[Rei+02] Erik Reinhard et al. "Photographic tone reproduction for digital images".
In: *ACM Trans. Graph.* 21.3 (July 2002), pp. 267–276. ISSN: 0730-0301.
DOI: 10.1145/566654.566575. URL: https://doi.org/10.1145/566654.566575.

[RW80] Steven M. Rubin and Turner Whitted.
"A 3-dimensional representation for fast rendering of complex scenes".
In: *SIGGRAPH Comput. Graph.* 14.3 (July 1980), pp. 110–116. ISSN: 0097-8930.
DOI: 10.1145/965105.807479. URL: https://doi.org/10.1145/965105.807479.

[Sam89] Hanan Samet. "Implementing ray tracing with octrees and neighbor finding".
In: *Computers & Graphics* 13.4 (1989), pp. 445–460. ISSN: 0097-8493.
DOI: https://doi.org/10.1016/0097-8493(89)90006-X. URL:
https://www.sciencedirect.com/science/article/pii/009784938990006X.

[Sch11] Schreiberx. *Example of bounding volume hierarchy*. Dec. 2011.
URL: https://commons.wikimedia.org/wiki/File:Example_of_bounding_volume_hierarchy.svg.

[SFG20] V. V. Sanzharov, V. A. Frolov, and V. A. Galaktionov.
"Survey of Nvidia RTX Technology".
In: *Programming and Computer Software* 46.4 (July 2020), pp. 297–304.
ISSN: 1608-3261. DOI: 10.1134/S0361768820030068.
URL: https://doi.org/10.1134/S0361768820030068.

[Sho92] Ken Shoemake. "III.6 - UNIFORM RANDOM ROTATIONS".
In: *Graphics Gems III (IBM Version)*. Ed. by DAVID KIRK.
San Francisco: Morgan Kaufmann, 1992, pp. 124–132. ISBN: 978-0-12-409673-8.
DOI: https://doi.org/10.1016/B978-0-08-050755-2.50036-1. URL:
https://www.sciencedirect.com/science/article/pii/B9780080507552500361.

[Tha11] Jonathan Thaler. *Deferred Rendering*. Tech. rep. Feb. 2011.
URL: https://www.researchgate.net/profile/Jonathan_Thaler2/publication/323357208_Deferred_Rendering/links/5a8fce31aca272140560aaad/Deferred-Rendering.pdf.

[Tim09] Lottes Timothy. *FXAA*. Tech. rep. NVIDIA Corporation, 2009.
URL: https://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf.

[UKW23] Johannes Unterguggenberger, Bernhard Kerbl, and Michael Wimmer.
"Vulkan all the way: Transitioning to a modern low-level graphics API in academia".
In: *Computers & Graphics* 111 (2023), pp. 155–165. ISSN: 0097-8493.
DOI: https://doi.org/10.1016/j.cag.2023.02.001. URL:
https://www.sciencedirect.com/science/article/pii/S0097849323000249.

[Vri15] Joey de Vries. *Learn OpenGL*. 2015.
URL: https://learnopengl.com/ (visited on 05/11/2023).

[Whi80] Turner Whitted. "An improved illumination model for shaded display".
In: *Commun. ACM* 23.6 (June 1980), pp. 343–349. ISSN: 0001-0782.
DOI: 10.1145/358876.358882. URL: https://doi.org/10.1145/358876.358882.

[Wri21] Daniel Wright. "Radiance Caching for Real-Time Global Illumination".
SIGGRAPH: Advances in Real-Time Rendering in Games. 2021.
URL: https://advances.realtimerendering.com/s2021/Radiance%20Caching%20for%20real-time%20Global%20Illumination%20(SIGGRAPH%202021).pptx.

[WSB03] Z. Wang, E.P. Simoncelli, and A.C. Bovik.
"Multiscale structural similarity for image quality assessment". In: *The Thrity-Seventh Asilomar Conference on Signals, Systems & Computers, 2003*. Vol. 2. 2003, 1398–1402 Vol.2. DOI: 10.1109/ACSSC.2003.1292216.

Appendix A

Detailed Results

A.1 Full Lighting

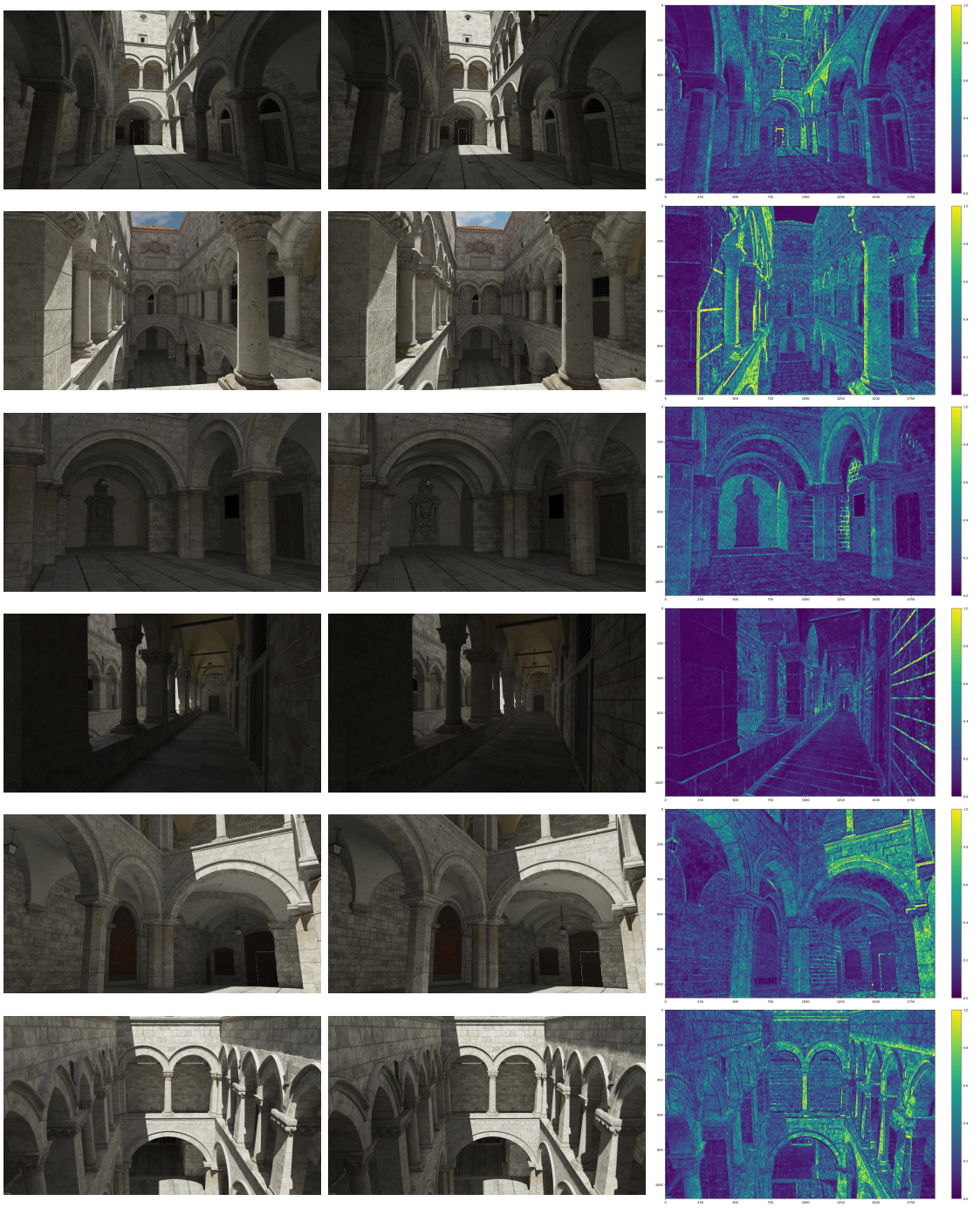


Figure A.1: Direct and indirect illumination from path traced reference compared with hybrid renderer with 8 by 8 by 8 probes and 256 rays per probe. First column is the path traced reference. Second column is the hybrid renderer. Final column is the SSIM of the two previous images in the row. Every row is for each view from 1 to 6.

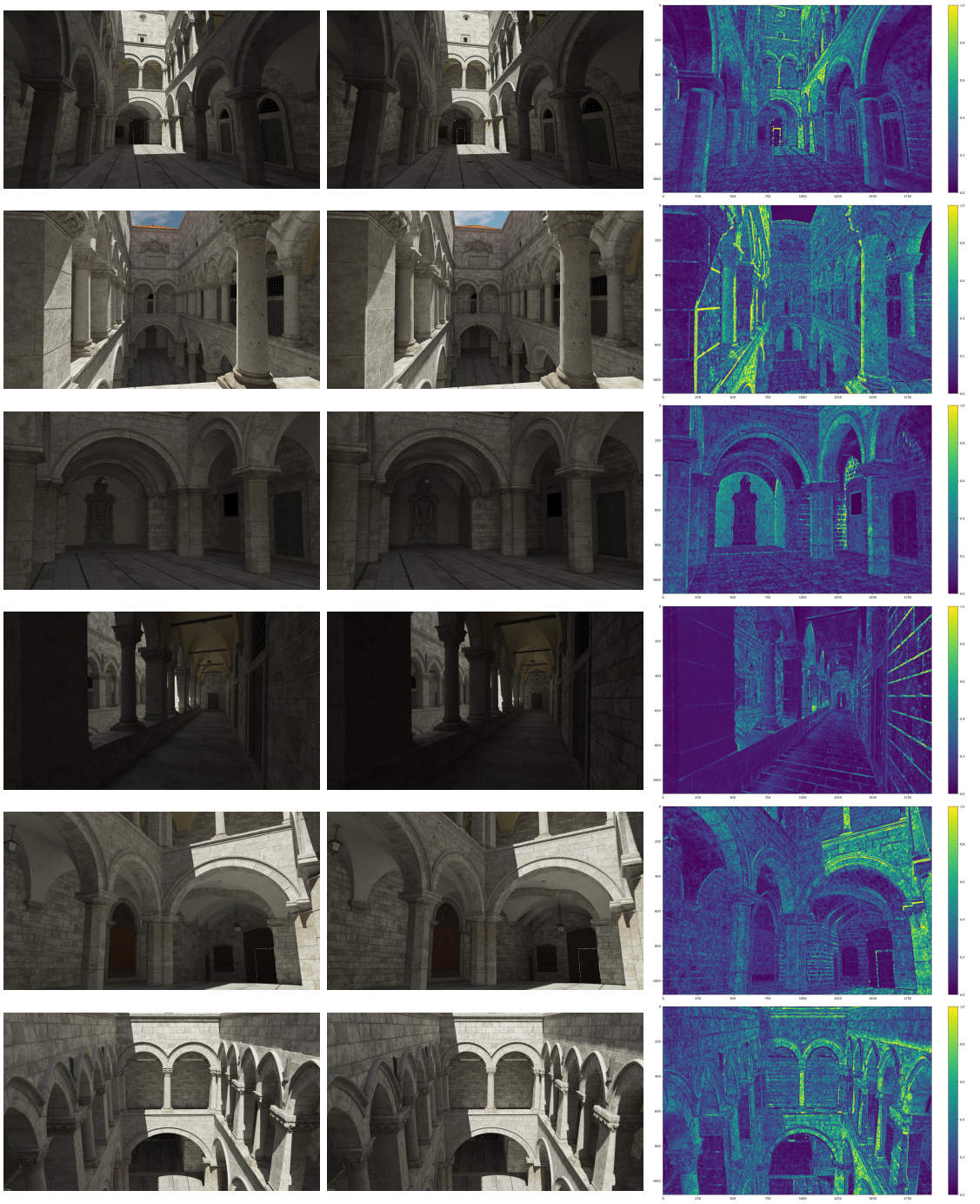


Figure A.2: Direct and indirect illumination from path traced reference compared with hybrid renderer with 16 by 16 by 16 probes and 256 rays per probe. First column is the path traced reference. Second column is the hybrid renderer. Final column is the SSIM of the two previous images in the row. Every row is for each view from 1 to 6.

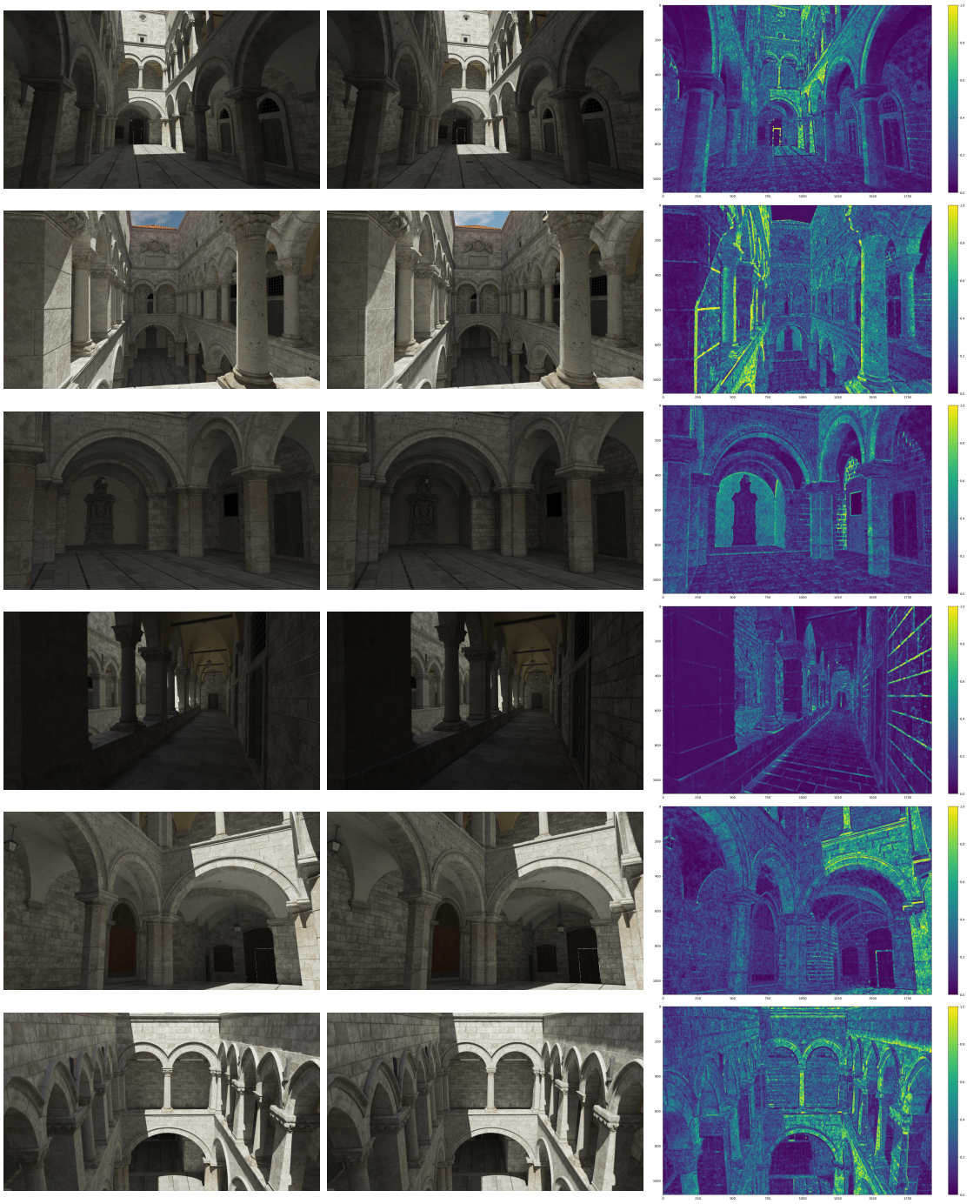


Figure A.3: Direct and indirect illumination from path traced reference compared with hybrid renderer with 32 by 32 by 32 probes and 256 rays per probe. First column is the path traced reference. Second column is the hybrid renderer. Final column is the SSIM of the two previous images in the row. Every row is for each view from 1 to 6.

A.2 Indirect Illumination

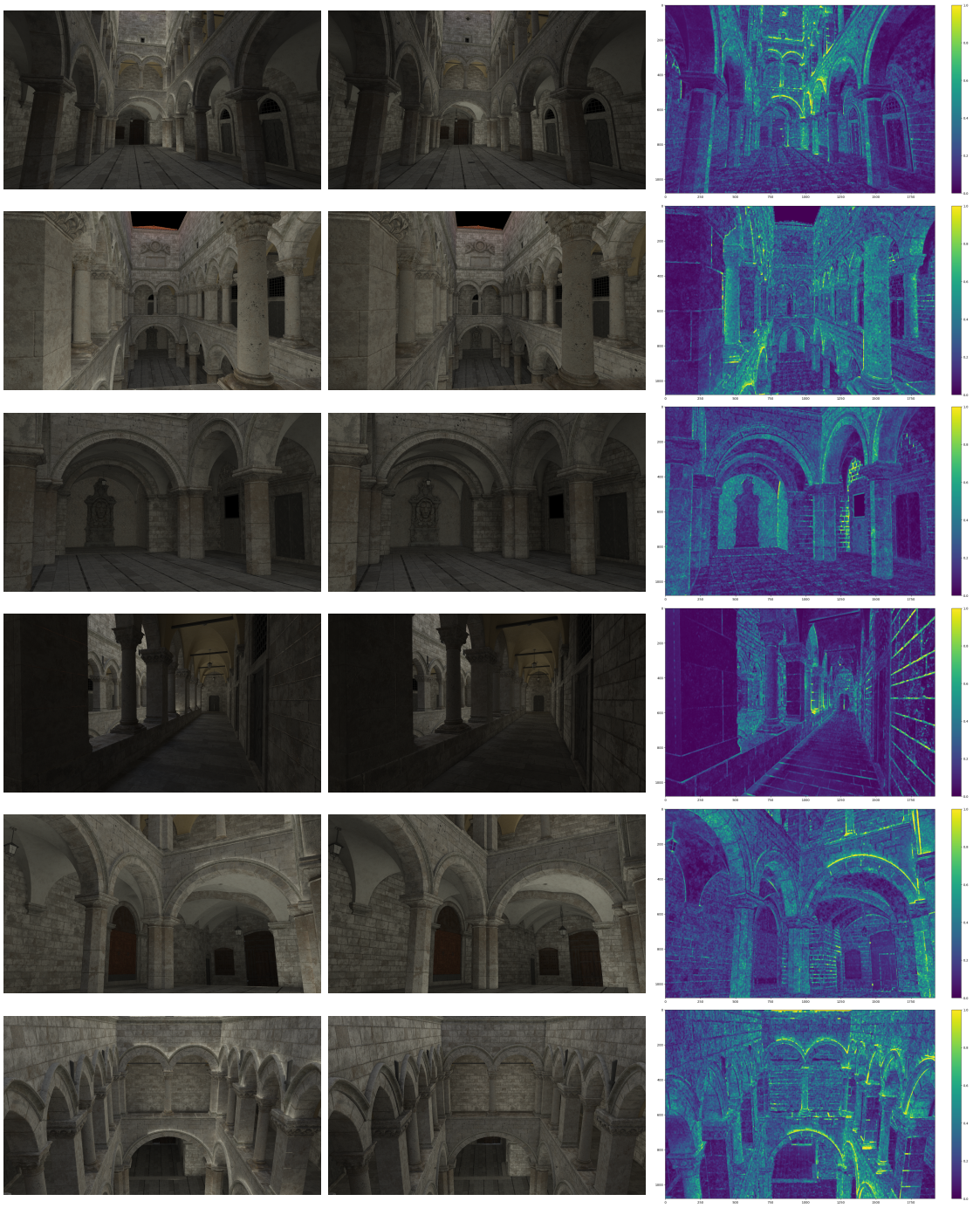


Figure A.4: Only indirect illumination from path traced reference compared with hybrid renderer with 8 by 8 by 8 probes and 256 rays per probe. First column is the path traced reference. Second column is the hybrid renderer. Final column is the SSIM of the two previous images in the row. Every row is for each view from 1 to 6.

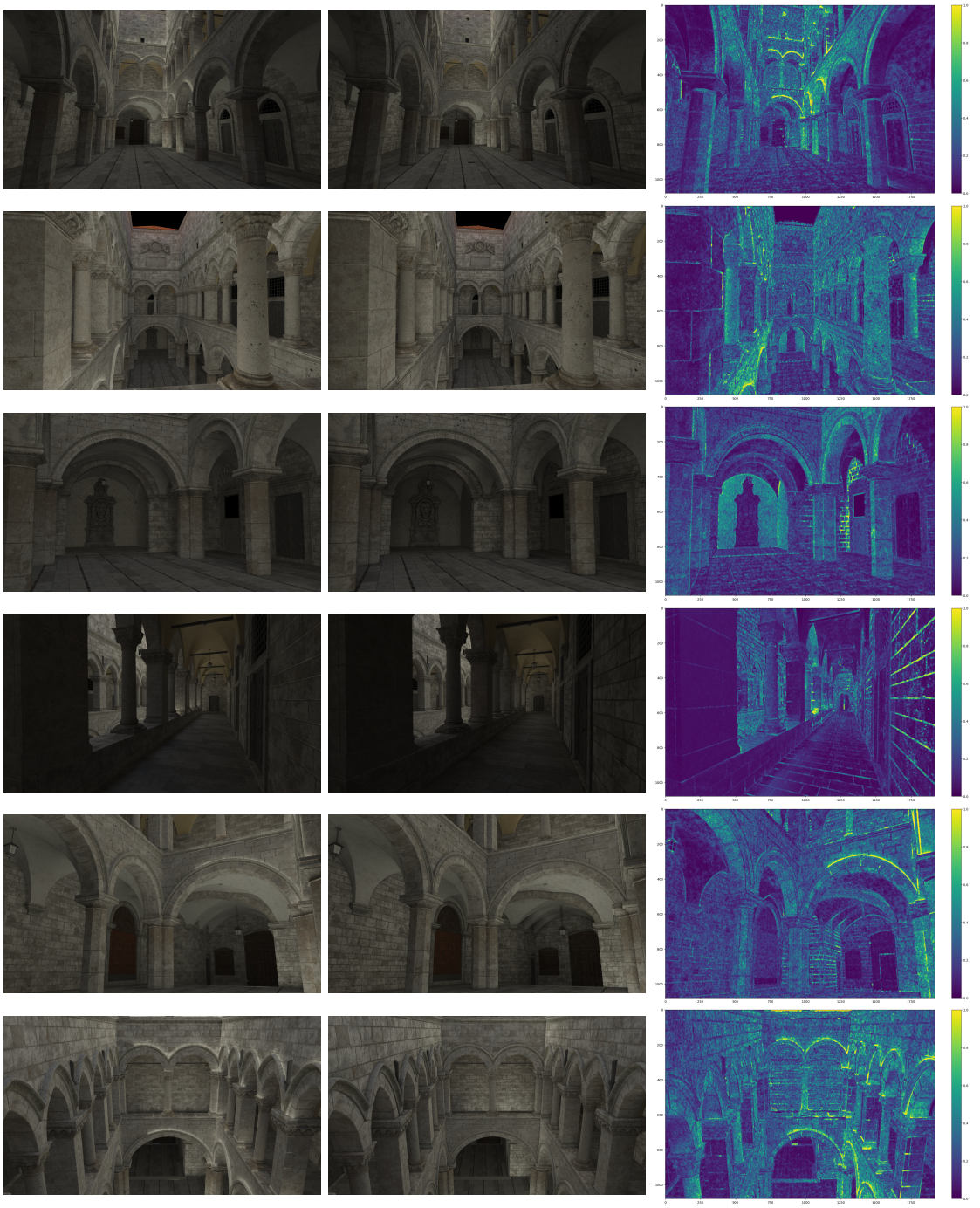


Figure A.5: Only indirect illumination from path traced reference compared with hybrid renderer with 16 by 16 by 16 probes and 256 rays per probe. First column is the path traced reference. Second column is the hybrid renderer. Final column is the SSIM of the two previous images in the row. Every row is for each view from 1 to 6.

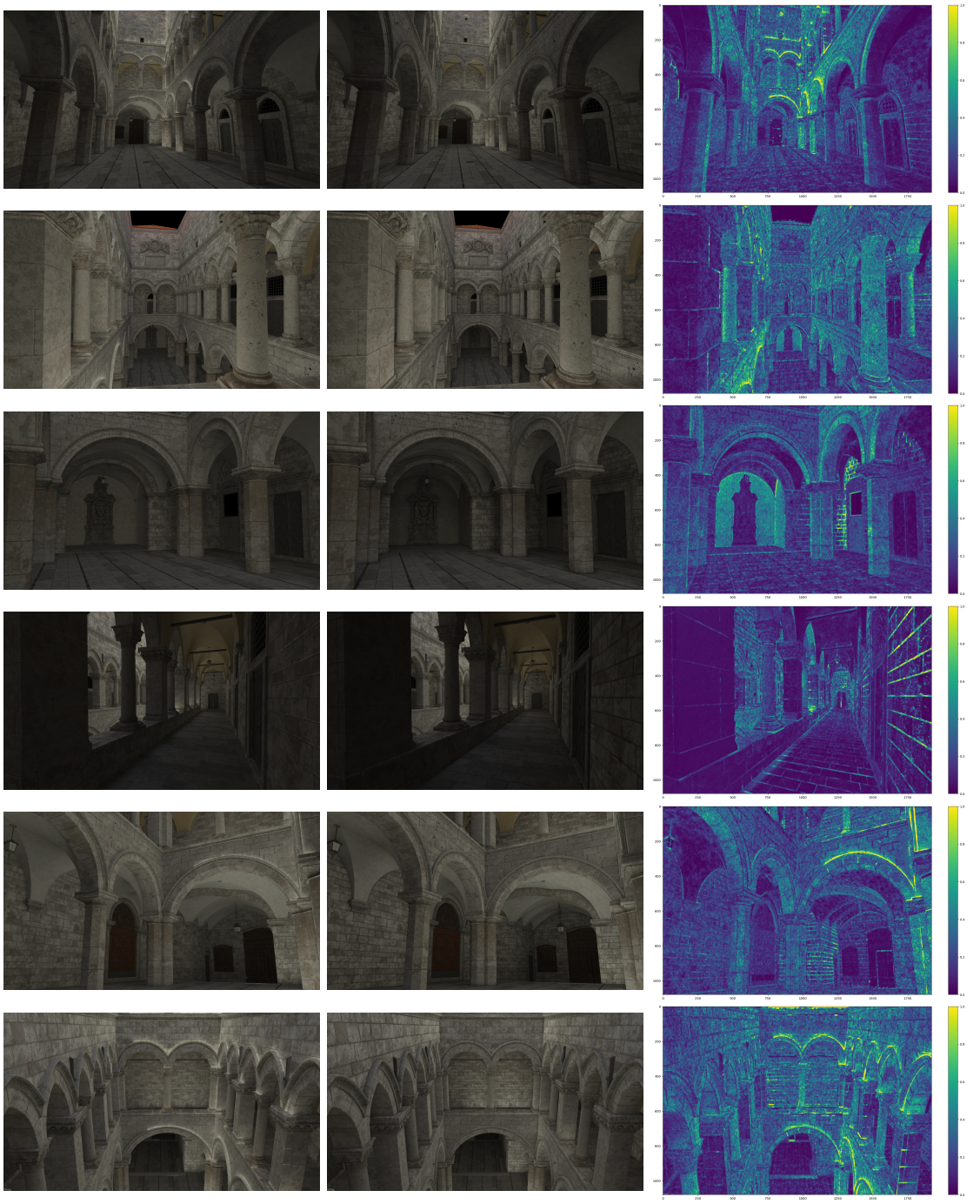


Figure A.6: Only indirect illumination from path traced reference compared with hybrid renderer with 32 by 32 by 32 probes and 256 rays per probe. First column is the path traced reference. Second column is the hybrid renderer. Final column is the SSIM of the two previous images in the row. Every row is for each view from 1 to 6.

A.3 Direct Illumination

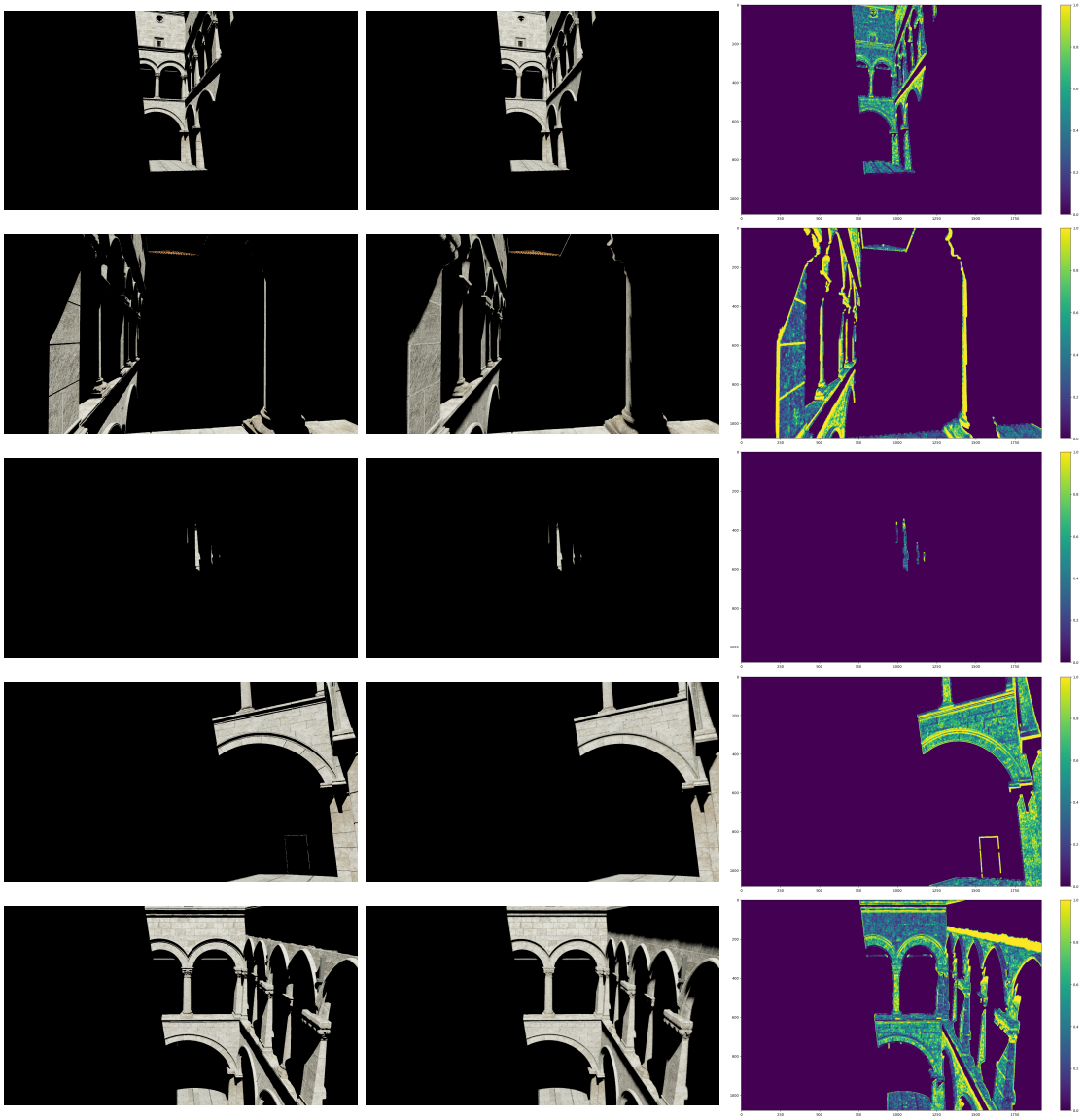


Figure A.7: Only direct illumination from path traced reference compared with hybrid renderer. First column is the path traced reference. Second column is the hybrid renderer. Final column is the SSIM of the two previous images in the row. Every row is for each view from 1 to 6 except 2 since no direct illumination is visible from that camera perspective.

A.4 Ray per Probe Comparison

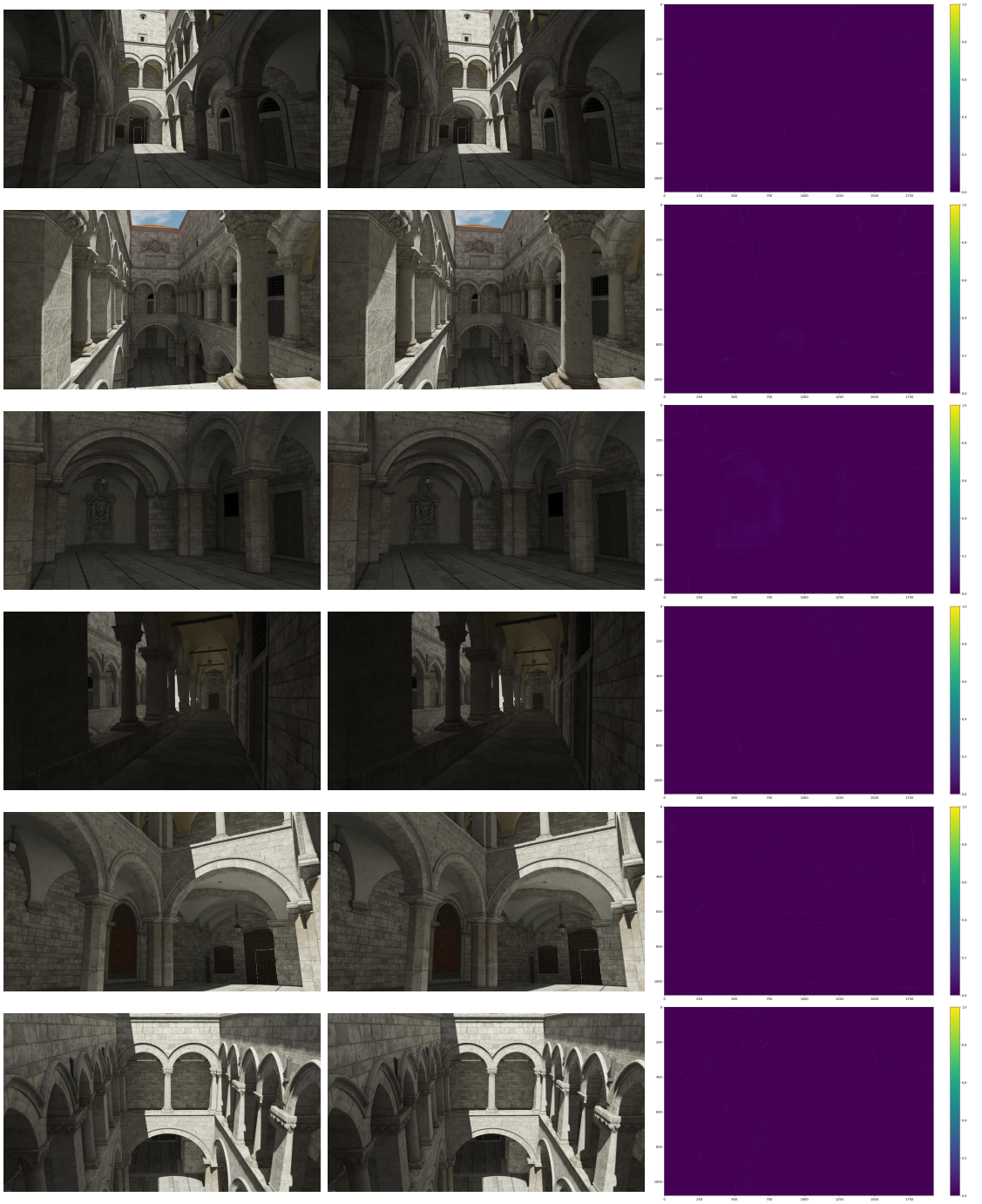


Figure A.8: Direct and indirect illumination from hybrid renderer with 256 rays per probe compared with hybrid renderer 512 rays per probe. Both renderers with 8 by 8 by 8 probes. First column is the hybrid renderer with 256 rays per prob. Second column is the hybrid renderer with 512 rays per probe. Final column is the SSIM of the two previous images in the row. Every row is for each view from 1 to 6.