TU WIEN Informatics

Implementierung eines Hybrid Renderers in Rust unter Verwendung einer Forward+ Pipeline

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software & Information Engineering

eingereicht von

Munir Yousif Elagabani
Matrikelnummer 12022518

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Dipl.-Ing. Lukas Lipp, Bsc

Wien, 3. Februar 2025

Munir Yousif Elagabani	Michael Wimmer

TU WIEN Informatics

Implementation of a Hybrid Renderer in Rust using a Forward+ Pipeline

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Munir Yousif Elagabani
Registration Number 12022518

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Dipl.-Ing. Lukas Lipp, Bsc

Vienna, February 3, 2025

Munir Yousif Elagabani	Michael Wimmer

Erklärung zur Verfassung der Arbeit

Munir Yousif Elagabani

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 3. Februar 2025

Munir Yousif Elagabani

Danksagung

Ich möchte mich herzlich bei meiner Familie und meinen Freunden bedanken, die mir immer beigestanden sind und mich bei der Anfertigung dieser Bachelorarbeit unterstützt und motiviert haben.

Meinem Assistenten Dipl.-Ing. Lukas Lipp möchte ich auch einen großen Dank aussprechen. Für die hilfreichen Anregungen und die entspannten Meetings bedanke ich mich herzlich.

Ebenfalls möchte ich mich bei meinen Freunden Sina Bayatmoghadam, Arian Sabri und Stefan Brandmair für das Korrekturlesen dieser Arbeit bedanken.

Zu guter Letzt möchte ich mich nochmals bei meinen Eltern bedanken, die mir es ermöglicht haben, dieses Studium ohne Sorgen abzuschließen.

Acknowledgements

I would like to thank my family and friends who have always stood by me and supported and motivated me during the development of this Bachelor's thesis.

Special thanks to my assistant Dipl.-Ing. Lukas Lipp for his helpful suggestions and the relaxed environment during the meetings.

I also want to thank my friends Sina Bayatmoghadam, Arian Sabri, and Stefan Brandmair for proofreading this thesis.

Last but not least, I would like to thank my parents once again for making it possible for me to complete this bachelor's program without any worries.

Kurzfassung

In dieser Arbeit untersuchen wir das Thema des hybriden Renderings, welches Rasterisierung und Raytracing kombiniert, um realitätsnahe Bilder zu erzeugen. Die Rasterisierung ist zwar seit Jahrzehnten die primäre Rendering-Technik, diese hat aber Schwierigkeiten mit der Erzeugung akkurater Lichteffekte, da dabei jedes Primitiv isoliert verarbeitet wird. Raytracing hat in den letzten Jahren an Popularität gewonnen, da mit dieser Methode es möglich ist, akkurate Beleuchtungseffekte zu erzeugen, welche einer der primären Faktoren für die wahrgenommene Realitätsnähe sind. Es ist jedoch durch die hohe Anzahl an benötigten Strahlen eine Herausforderung, mit einem vollständig auf Raytracing basierenden Renderer eine Echtzeitleistung zu erzielen. Das hat die Entwicklung von Renderern motiviert, die die Rasterisierung als primäre Rendering-Technik und Raytracing für die Berechnung von Lichteffekten verwenden.

Das Ziel dieser Arbeit ist es, die Verwendung einer Forward+ Pipeline für das Bauen eines hybriden Renderers zu zeigen, der die Hardwarebeschleunigung der Grafikkarte für Raytracing-Aufgaben verwendet. Dazu bauen wir eine Rendering-Engine, die "raytraced hard shadows" implementiert und Methoden untersucht, die die Anzahl der Schattenstrahle reduzieren. Das Culling von Strahlen auf der Grundlage des Winkels der Oberflächennormalen mit der Strahlrichtung und das kachelbasierte Light-Culling sind die zwei Methoden, die in dieser Arbeit implementiert worden sind. Die Ergebnisse zeigen eine Leistungssteigerung von 39 % mit den in dieser Arbeit vorgestellten Verbesserungen.

Abstract

In this thesis, we explore the topic of hybrid rendering, which combines rasterization and ray tracing to produce high-fidelity images. While rasterization has been the primary rendering technique for real-time applications for decades, it struggles with producing accurate lighting effects because of its nature of processing each primitive in isolation. Ray tracing has gained in popularity in recent years due to its ability to produce accurate lighting effects, which is one of the primary contributors to perceived visual fidelity. Achieving real-time performance with an entirely ray tracing-based renderer is challenging because of the high number of rays required to produce images. This has motivated the development of renderers that use rasterization as the primary rendering technique and ray tracing for producing lighting effects.

The goal of this thesis is to showcase the usage of a forward+ pipeline to build a hybrid renderer utilizing the hardware acceleration of the graphics card for ray tracing tasks. To this end, we build a rendering engine that implements ray-traced hard shadows and explore methods that reduce the number of shadow rays. Shadow ray culling based on the angle of the surface normal with the ray direction and tile-based light culling are the two methods implemented in this work. The results have shown a performance uplift of 39% with the improvements presented in this thesis.

Contents

CHAPTER 1

Introduction

Light is a primary factor contributing to the perceived photorealism of computer graphics. The demand for photorealistic graphics has been growing in recent years. This can be observed in the gaming industry, where more and more new games aim to deliver high visual fidelity.

Traditionally, the primary rendering technique for real-time applications has been rasterization. Rasterization's strength is that it can render 3D meshes to the screen efficiently. However, this rendering technique struggles with accurate light simulations. The reason is that light calculations require the ability to query information about the whole scene, which is usually enabled in rasterization by utilizing an approximated scene representation.

The rendering technique that does not suffer from this limitation is ray tracing. This method allows information to be queried from any point to any direction in the scene without resorting to approximated scenes.

Ray tracing is notorious for being computationally expensive to calculate. Even with recent advancements in both hardware acceleration and software support, achieving real-time performance by utilizing ray tracing remains challenging.

This has motivated the creation of hybrid renderers that take advantage of both rendering techniques. These renderers use ray tracing for a selection of lighting effects and rasterization as the primary rendering technique. Utilizing both techniques yields high performance and visual fidelity.

One technique to integrate ray tracing into a rasterization-based rendering engine is to use the rasterized world positions as a ray tracing starting point to determine the shading for each fragment. The problem one quickly runs into while doing that is that triangles might overlap, resulting in already shaded fragments getting overdrawn, rendering all shading computations for the overdrawn fragments obsolete. This can be addressed by utilizing the deferred rendering pipeline. It solves this so-called overdraw problem by

storing all the visible geometry in a collection of textures, often called the geometry buffer (G-Buffer), which are then utilized by a shading pass.

Forward+ is an alternative rendering technique aiming to reduce overdraw by utilizing a depth prepass to determine the closest fragment to the camera. Compared to deferred rendering, it requires less memory bandwidth, enables more complex materials, and allows for easier transparency and hardware anti-aliasing.

Another benefit of the Forward+ pipeline is that it supports many-light rendering via light culling. Light culling organizes lights into a data structure to enable the retrieval of potential illuminators for each point in the scene. This is relevant for the ray tracer, as the information from light culling also allows the culling of shadow rays, which we will explore in this thesis.

1.1 Method

In this thesis, a hybrid renderer is built using the Rust programming language and Vulkan Graphics API. The goal is to build an example hybrid renderer that illustrates the usage of a Forward+ pipeline in a hybrid renderer.

This hybrid renderer implements hard shadows. Although hard shadows are rarely found in nature, all the concepts used to reduce the number of shadow rays can be applied to a soft shadow implementation. We will utilize the light culling from the Forward+ pipeline to minimize the number of shadow rays required. Although Forward+ does have other positive properties that have already been mentioned, this thesis only focuses on the points relevant to light.

We chose Rust as the programming language due to personal preference and its convenient package manager called Cargo. The Vulkan graphics API is used to interface with the dedicated hardware, which accelerates ray-tracing tasks.

1.2 Contributions

We will discuss how to design a hybrid rendering engine that uses both rasterization and ray tracing. We will show how shadow rays can be culled based on light culling and surface normals. We will highlight an improved method of accurately converting depth values to world positions necessary for ray tracing. The surface normals for shadow ray culling are reconstructed from depth values. We highlight a method that produces more accurate reconstructed normals.

1.3 Structure of this Work

In Chapter 2 we dive into the theory of hybrid rendering explaining the concept of both rasterization and ray tracing. Additionally, the limitations of rasterization are highlighted.

Chapter 3 discusses the related work that made it possible to integrate ray tracing into real-time rendering applications.

Chapter 4 presents the design of the hybrid rendering engine. We go over all the necessary rendering stages to produce a rendered image with ray-traced shadows.

The results and testing setup of the evaluation of this rendering engine and the methods used to cull rays are presented in Chapter 5. In Chapter 6 we discuss the limitations and viability of hybrid rendering methods for real-time applications.

CHAPTER 2

Background

This chapter gives an overview of the theory of rasterization and its limitations. We introduce ray tracing and explain how rasterization and ray tracing can be combined in a hybrid renderer. In order to improve the accuracy of depth values, the reverse-Z projection is discussed. Finally, the theory of the light model used in the hybrid renderer developed in this thesis is explained.

2.1 Rasterization

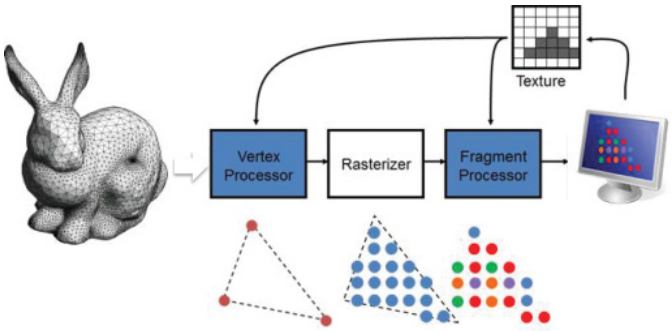


Figure 2.1: Overview of rasterization-based rendering [DNL+17]

Rasterization-based rendering is enabled by the graphics pipeline. The graphics pipeline, as illustrated in Figure 2.1, consists of several stages, some programmable and others fixed-function, meaning they are implemented on a hardware level and can therefore only be configured. We will not cover geometry and tessellation shaders, as they are not used in this project. In this context, scenes are represented as a set of meshes, where each mesh is a set of primitives, usually triangles [AMHH19].

The first stage of the graphics pipeline is Input Assembly, where vertex attributes, such as position, texture coordinates, and normals are fetched from memory. Next comes the programmable Vertex Shader stage, where vertices are transformed into clip space, and additional vertex attributes for later shading are computed. In the following stage Primitive Assembly groups vertices into primitives and culls primitives that face away from the camera and those outside the viewport.

After those stages, often also referred to as the geometry processing stage, comes the rasterization stage. This stage generates fragments through spatial sampling on a regular grid for the remaining primitives and interpolates vertex attributes across the surface of each primitive. Fragments are candidate pixels of the final image.

The following stages are usually referred to as the fragment processing stage. It starts with the stage where early per-fragment operations are performed, which includes discarding fragments through early depth and stencil testing. Next comes the programmable fragment shader stage, where the fragment's color is determined using the interpolated vertex attributes generated in the rasterization stage. Finally, if early tests are disabled, stencil and depth tests are performed, blending is applied and the final fragments are written into the frame buffer.

2.2 Limitations of Rasterization

The fact that primitives in rasterization-based rendering are handled separately introduces a limitation. This limitation is the ability to accurately simulate light transport. Light effects such as shadows, reflections, refractions, and global illumination require that multiple primitives can be considered at once. This is because those effects need to query the surroundings, which rasterization can not provide without resorting to workarounds [WS19].

Rasterization-based rendering techniques that work around this limitation include shadow mapping, screen-space reflections, screen-space ambient occlusion, and baked light maps. What all of those techniques have in common is that they build data structures that contain approximations of the scene that are used for lookups during shading. Precomputing and storing this simplified scene in a resolution suitable for accurate light simulations is generally infeasible, resulting in rendering artifacts or missing effects [WS19]. In contrast to rasterization, ray tracing considers all primitives while tracing rays. This makes it possible to query the surroundings and enable accurate light transport simulations.

2.3 Ray Tracing

Ray tracing works similarly to how humans see. We see objects because they reflect light from a light source into our eyes. Not all reflected light rays from a light source arrive at our eyes. In ray tracing-based rendering, only rays that finally hit the camera are

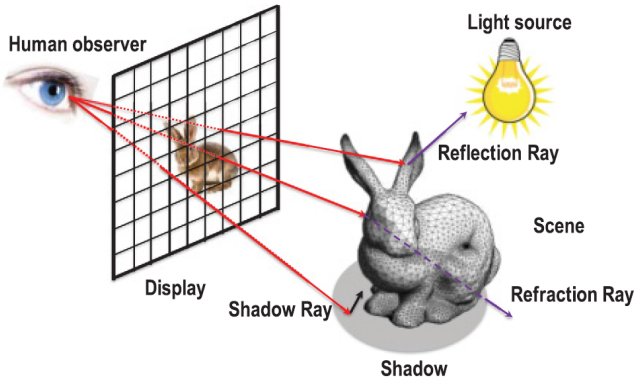


Figure 2.2: Ray tracing-based rendering [DNL+17]

relevant. For this reason, the process is performed in reverse, meaning the rays start from the camera rather than from the light sources. This is valid because it is assumed that light travels in straight lines. The inverse path of a line is again a line. The physical properties of light hold regardless of the direction in which it travels.

As illustrated in Figure 2.2, for every pixel on the screen a ray is traced from the camera's position through the screen's pixel into the scene. Those initial rays are called primary rays. When those rays intersect with an object, secondary rays are cast from the intersection point to determine the shading of the point. Shadow rays are used to determine whether any light source illuminates the intersection point of the primary ray with the scene's geometry. Other secondary rays include the reflection ray for reflective materials and the refraction ray for translucent materials.

There are different types of ray tracing that allow the calculation of certain lighting effects. With Whitted ray tracing, a single deterministic sample per pixel is sufficient to create effects such as hard shadows or specular reflections [Whi80]. Distributed ray tracing uses several random samples per pixel from different directions, making soft shadows or diffuse reflections possible [CPC84]. Path tracing builds on this and enables global illumination using several stochastic samples per pixel and multiple bounces [Kaj86]. As it is necessary to limit the number of samples per pixel for real-time performance, the stochastic nature of distributed ray tracing and path tracing leads to noise in the image. This thesis focuses on Whitted ray tracing to create hard shadows.

2.4 Acceleration Structure

A naive implementation of ray tracing tests each ray for intersection with every primitive of the scene. The number of operations of this approach is in $\Omega(n \cdot m)$, where n is the number of rays and m is the number of primitives. With a scene that consists of several thousands to millions of triangles and a ray for each pixel of a 1080p screen, the number

of operations hinders real-time performance. To address this, acceleration structures have been developed to reduce the number of intersection tests.

The Bounding Volume Hierarchy (BVH) is the de facto standard acceleration structure. Bounding volumes are tightly enclosing geometries that are cheap to test for intersections with a ray. Common volumes are Axis-Aligned Bounding Box (AABB), Oriented Bounding Box, or Bounding Spheres. In a BVH, the bounding volumes are organized into a tree-structured hierarchy, where each inner node represents a bounding volume and the leaf nodes are primitives [MOB+21].

The traversal of the acceleration structure starts by testing for intersection with the root bounding volume. If an intersection is found, then the ray is recursively tested with the subsequent child nodes until a leaf node is reached, where only a handful of primitives need to be tested for intersection. Using this acceleration structure, all ray-triangle intersection tests of triangles contained in a bounding box can be skipped if the ray does not intersect the respective bounding volume.

2.5 Hybrid Rendering

Ray tracing can be integrated into traditional renderers that use rasterization. Rasterization can be used to find the intersection point of the primary ray with the scene's geometry. This is possible because a depth buffer is used for rasterization-based rendering to ensure that only the closest fragment to the camera is written into the frame buffer. The depth buffer stores the depth of each fragment, which is used during depth testing in the graphics pipeline. Those depth values can be converted back into world positions that can be used as the starting point for secondary rays such as shadow rays.

2.6 Reverse-Z Projection

A reverse-Z projection matrix can be used to enhance the precision of the depth values. As seen in Equation 2.1, the matrix is adapted for use in Vulkan, which has a right-handed coordinate system with the y-axis pointing down. In the equation, a is the aspect ratio of the framebuffer $\frac{w}{h}$, ϕ the field of view angle, f the distance to the far plane, and n the distance to the near plane.

$$P(a, \phi, z_n, z_f) = \begin{pmatrix} \frac{a^{-1}}{\tan\frac{\phi}{2}} & 0 & 0 & 0 \\ 0 & -\frac{1}{\tan\frac{\phi}{2}} & 0 & 0 \\ 0 & 0 & -\frac{z_f}{z_n - z_f} & -\frac{z_f \cdot z_n}{z_n - z_f} \\ 0 & 0 & -1 & 0 \end{pmatrix} \tag{2.1}$$

The traditional projection matrix maps the near plane to a depth value of 0.0 and the far plane to 1.0. Due to the nature of IEEE 754 floating point numbers, most of the precision is near 0.0, where the range from 0.0 to 0.5 contains 99.21% of all representable

values [HH]. This results in most of the depth buffer's precision being concentrated near the camera, while at greater distances the precision is decreased significantly.

The reverse-Z projection matrix maps the near plane to a depth of 1.0 and the far plane to 0.0. This results in a more uniform distribution of precision across the entire depth range [Ree]. Consequently, the depth testing operator has to be changed from $\leq$ (less than or equal) to $\geq$ (greater than or equal) and the clear value to 0.0 instead of 1.0. Additionally, when fetching the maximum depth, the depth at which objects are the furthest, the minimum operator has to be used, and vice versa.

2.7 Cook-Torrance Light Model

The Cook-Torrance light model is based on the microfacet theory where it is said that any surface can be described by tiny imaginary surface patches, which are much smaller than an output pixel. The alignment of said surface patches can vary depending on the roughness parameter [WMLT07].

Using the rendering equation, the outgoing radiance $L_o(x, \omega_0)$ at point x in direction ω_0 can be calculated as follows [BWB19]:

$$L_o(x, \omega_0) = L_e(\omega_0) + \int_\Omega f_r(x, \omega_i, \omega_0) L_i(x, \omega_i)(\omega_i \cdot n) d\omega_i \tag{2.2}$$

In this equation, $L_e(\omega_0)$ represents the self-emitted radiance, $f_r(x, \omega_i, \omega_0)$ is the Bidirectional Reflectance Distribution Function (BRDF), $L_i(x, \omega_i)$ denotes the incoming radiance of the i-th light from direction ω_i and n the normalized surface normal.

The Cook-Torrance BRDF models how much each light ray contributes to the final reflected light from an opaque object. The BRDF is composed of a diffuse and specular term and is combined as follows:

$$f_r = \underbrace{k_d \cdot f_{\text{lambert}}}_{\text{diffuse term}} + \underbrace{k_s \cdot f_{\text{cook-torrance}}}_{\text{specular term}}, \quad \text{where} \quad k_s + k_d = 1 \tag{2.3}$$

In this model, the constants k_s and k_d represent the ratio of incoming light that is reflected and refracted, respectively, and must sum to 1 to ensure energy conservation. The Lambertian reflection model denoted as f_{lambert} is the diffuse part of the BRDF. The k_s constant is calculated through the Fresnel function, which models the proportion of reflected and refracted light [Sch94].

The specular term $f_{\text{cook-torrance}}$ is further refined by the normal distribution function and the geometry function. The normal distribution function approximates the proportion of microfacets that are orthogonal to the halfway vector h between the surface normal n and light direction ω_i. The geometry function approximates the proportion of microfacets that are self-shadowing [WMLT07].

CHAPTER 3

Related Work

In this section, we will discuss the advancements in hybrid rendering. Since rasterization has been a well-established rendering technique in real-time rendering for decades, we will focus on the advancements that made ray tracing possible in real-time.

3.1 Optimizing Bounding Volume Hierarchies

Current research on acceleration structures focuses on optimizing both the construction time and rendering time of the bounding volume hierarchies. The quality of an acceleration structure is determined by the predicted number of required operations to find an intersection with a ray [MOB+21]. For real-time applications with dynamic scenes, we strive for a balance of construction time and traversal time. In the survey by Meister et al. [MOB+21], they present heuristics for determining the probability of traversing a child node given the parent node is hit. One of the highlighted heuristics is the surface area heuristic (SAH). Under the assumption that rays are uniformly distributed, SAH states that the probability of hitting a node is proportional to its surface area. Other heuristics also mentioned in the paper use the ratio of ray hits or the ratio of visible primitives [MOB+21].

3.2 Introduction of Hardware Ray Tracing

Motivated by the growing demand for photorealism in the gaming industry, graphics card manufacturers have started integrating dedicated hardware into their graphics cards that accelerate ray tracing tasks. The two most common ray tracing tasks are the traversal of acceleration structures and ray-triangle intersection tests. The acceleration structure implemented on the graphics card for ray tracing is the Bounding Volume Hierarchy (BVH). Nvidia and Intel accelerate the traversal of the Bounding Volume Hierarchies (BVHs) and ray-triangle intersection tests with their RT Core and Xe Cores, respectively

[NVI18, Int22]. AMD's Ray Accelerator solely accelerates the calculation of ray-triangle intersections [Pom21].

3.3 Graphics API Support for Hardware Accelerated Ray Tracing

With the introduction of specialized hardware for ray tracing, graphics APIs had to add an interface to utilize them. Older graphics APIs such as OpenGL or prior versions of DirectX 12 do not support interfacing with the dedicated hardware. Interfacing with said hardware is possible in Vulkan using the ray tracing extensions or in DX12 using the DirectX Raytracing feature. The two interface implementations provide ways to build acceleration structures and trace rays on the GPU. Rays can be traced inline in any type of shader or in a dedicated ray tracing pipeline to enable dynamic shader execution based on different object hits.

3.4 Light culling

Light culling can not only improve shading performance but also reduce the number of shadow rays. Essentially, light culling determines which lights illuminate a certain region. If light culling determines that a certain light does not affect a given space, then the light does not need to be considered in the shading calculation, and no shadow ray needs to be traced to that light.

A common approach is to divide the screen into tiles. This approach is used in both tiled- [OA11] and clustered [OBA12] shading. The screen-space tiles extend into volumes in 3D. Each tile stores the lights where the light's volume intersects with the tile's volume. In clustered shading, the volumes are also subdivided along the Z axis to reduce the number of false positives due to depth discontinuities such as edges.

A different approach that is especially suited for ray tracing is to use bounding volume hierarchies. In the work of Moreau et al. [MPC19] they use a two-level BVH to enable efficient updates. Instead of recreating the entire hierarchy from scratch, they refit changes and retain the original topology.

CHAPTER 4

Designing a Rendering Engine

This chapter describes the implementation of the hybrid rendering engine. As depicted in Figure 4.1, the rendering engine goes through four passes to render a final frame. Those passes are the depth pre-pass, light culling pass, ray traced shadows pass, and the forward pass for shading. Since the results of each pass are required in subsequent passes, it is necessary to place synchronization barriers between each pass.

The pipeline used for this renderer is based on the Forward+ pipeline presented by Harada et al. [HMY12] and is extended with a ray-traced shadow pass.

4.1 Depth-Prepass

The entire scene is rendered in the depth pre-pass stage, but only depth values are stored. No fragment shading is performed, eliminating the need for a fragment shader in this

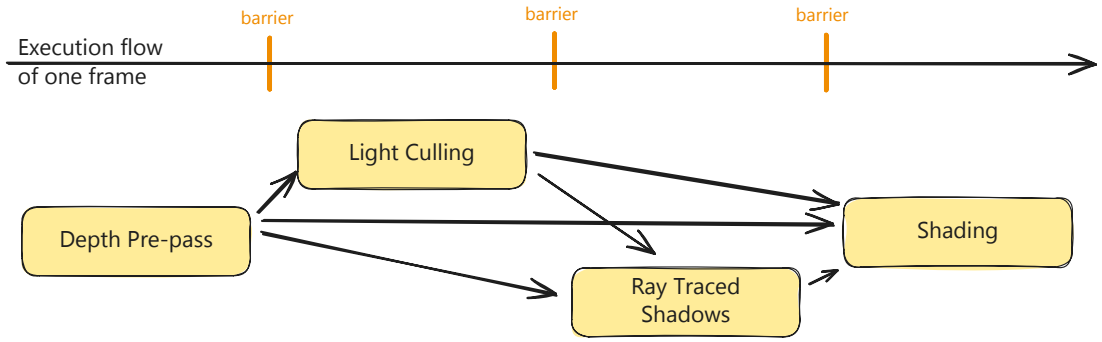


Figure 4.1: Overview of rendering pipeline: The pipeline consists of four passes highlighted in yellow boxes. The arrows illustrate the data flow between different passes, while the barriers ensure proper synchronization necessitated by the data dependencies.

pass. We utilize a reverse Z projection matrix to improve the precision of the depth values as described in Section 2.6. This pass is essential for the other three passes that use the depth values.

4.2 Light Culling

Light culling is a technique for rendering many lights by efficiently determining which lights affect which parts of the scene and therefore minimizing the number of lights that need to be considered for each texel's shading. Performing light culling on a per-pixel basis is not only computationally expensive due to the required number of intersection tests, but it also significantly increases the memory footprint due to the need to store light indices [HMY12].

4.2.1 Tile-based Light Culling

A more practical solution to this problem of per-pixel light culling is to divide the frame into equally sized square tiles and perform light culling for those larger areas. All pixels within a tile share a common light list. This means that even if only a single pixel of a tile is affected by a light source, this light would still be included in the shading calculation of all the tile's pixels. It is therefore crucial to choose a tile size that balances accuracy and efficiency [HMY12].

In 3D space, each tile's volume forms a frustum due to perspective effects. Lights that intersect a tile's frustum are included in its shading calculations. Figure 4.2 illustrates how light culling works on a single tile.

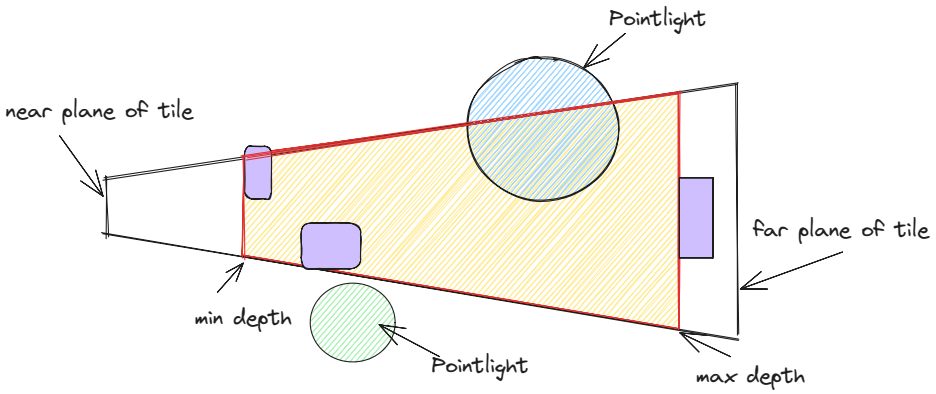


Figure 4.2: Light Culling: Visualized in a top-view is the frustum of a single tile outlined in red, with two point lights (blue and green) and objects (purple). The minimum and maximum depth bound the frustum. Lights volumes such as the blue shaded point light that intersects the tile's frustum, are stored in a list of lights for the corresponding tile. The green shaded point light does not intersect the tile's frustum and can therefore be omitted in the shading calculations.

4.2.2 Light Range Calculation

Light culling requires that light sources have a finite range because lights with an infinite range can never be culled as they illuminate the whole scene. The glTF file format used for scene loading in this renderer, allows light sources to have an undefined range, which should be interpreted as an infinite range. In this case, it is necessary to calculate a range where most of the light is transported. We have chosen a threshold of 0.02 for the minimum light intensity, similar to the approach in [dV].

$$I/D^2 = 0.02 \iff D = 50 \cdot \sqrt{0.02 \cdot I} \tag{4.1}$$

Here, I is the intensity of the light source, and D is the distance at which the light's intensity is measured. This constant 0.02 is a threshold that is close to the $5/256 = 0.01953125$ from [dV]. When solving for D, this results in a function that requires only a few operations to calculate. Only non-negative solutions are relevant for D.

4.2.3 Implementation

We implement light culling using the gather approach, as described by Harada et al. [HMY12]. The idea is to have a thread group for each tile, each calculating the light culling for their respective tile, and then gather light culling results into a common data structure. In this implementation, the light culling step is split into two compute shaders that run in sequence.

Two data structures are used together to store the results of light culling: The Light Index List and the Light Grid. The Light Index List contains sets of light indices for all tiles, stored in a linear list. The Light Grid data structure associates a tile to its corresponding slice in the Light Index List by specifying the offset and length. This is illustrated in Figure 4.3.

The first step in light culling is to calculate the four side planes of the tile's frustum. This involves transforming the screen-space positions of the four tile corners into view space. A plane is defined by its normal and the distance to the origin. For each plane of the frustum, two of the transformed corner positions and the eye position are used to determine the plane parameters. Note that the eye position is at the origin in view space and can therefore be hard-coded to $(0, 0, 0)$. Those steps are performed by the first shader, which stores the resulting frustums in a buffer for use in the next light culling shader.

Next, the remaining two front and back planes are calculated by finding the maximum and minimum depth within the tile. The depth values are fetched from the depth buffer created during the depth pre-pass. When using a reverse Z projection matrix, the minimum operator must be used to determine the maximum depth and vice versa.

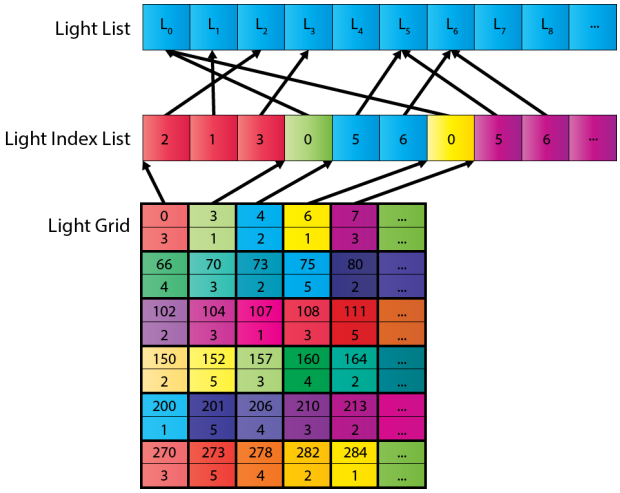


Figure 4.3: Light culling data structures: The Light Index List is a linear list consisting of sets of light indices for all tiles. The Light Grid associates a tile to its corresponding slice in the Light Index List. Each entry consists of the offset (top value) and the length (bottom value). [Jer]

In a round-robin fashion, each thread in the thread group is assigned a light to check for intersection with the tile's frustum. Different algorithms are used for the three punctual light types:

- Point lights intersect the frustum if the sphere is not fully contained in the negative half-space of at least one of the frustum's planes [Eri05].
- Spot lights intersect the frustum if the tip of its cone or the point Q on the base, which is furthest away from the plane, is not inside the negative half-space of any of the frustum's planes [Eri05].
- Directional lights do not have a limited range and are therefore never culled.

There is, however, a problem with those intersection tests: It is possible to have a point light that intersects the planes but not the actual frustum volume. This issue is illustrated in Figure 4.4. Such cases, where a light is incorrectly determined to intersect the frustum, are false positives.

To reduce the number of false positives, we apply an optimization described by Turanszkij [tur18]. The axis-aligned bounding box (AABB) of the tile's frustum is calculated. The AABB is a reasonable approximation when the range between minimum and maximum depth is small, as illustrated in Figure 4.5.

This optimization is used together with the previous intersection testing method for point lights. Combining both approaches makes it possible to leverage the benefits of the

AABB optimization while not introducing new false positives when the depth range is large.

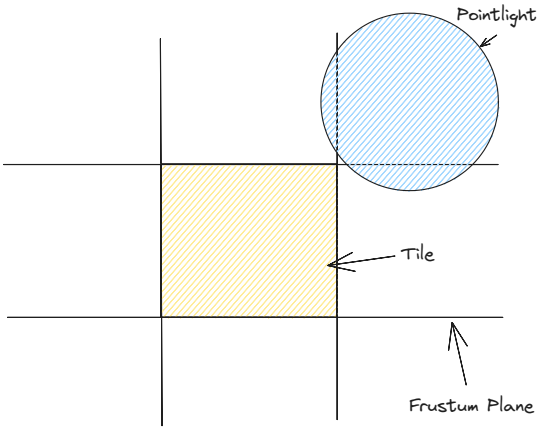


Figure 4.4: False positive case: Front view of a tile and point light. The point light intersects two side planes of the tile's frustum but not the actual frustum.

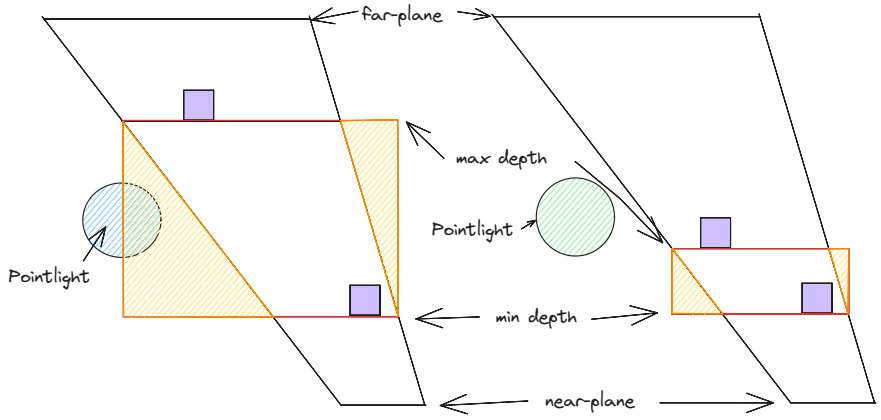


Figure 4.5: AABB Optimization: Two top views of tile's frustums with varying depth ranges. The left frustum has a bigger depth range, while the right one has a smaller depth range. The area highlighted in orange represents the volume where false positives may occur using this method. This volume is smaller for smaller depth ranges.

Finally, the results of each thread group are written into the data structure and to be used in both the subsequent shadow and shading pass.

4.3 Ray Traced Shadows

In this renderer, shadows for punctual lights producing hard shadows are implemented using ray tracing through ray queries in a separate compute shader. Ray queries enable

testing for intersections with geometry within the scene.

4.3.1 Normal Reconstruction from Depth

Since ray tracing is a relatively expensive operation, we aim to minimize the number of traced rays. One way to save shadow rays is to examine the surface normal and only trace when the angle between the surface normal and the light direction is less than 90°. When both vectors are normalized, this can be calculated using the dot product of both vectors. In this case, a negative dot product indicates that the surface is facing away from the light and the corresponding shadow ray does not need to be traced. Given that only depth values from the depth pre-pass are available at this stage, we opted to reconstruct the surface normal through the depth values.

The shadow compute shader is invoked for each texel of the depth buffer with a group size equal to the tile size, used in the light culling step. For the normal reconstruction, at least one extra depth sample per axis has to be fetched. Every sample from the depth buffer is used multiple times, when it is the current texel and as the neighboring sample for normal reconstruction. As reading from shared memory is cheaper than fetching a texture, group shared memory is used to reduce the number of texture look-ups. For this, each group, responsible for a single tile, fetches the relevant depth information for its tile and puts it in the group shared memory. This is implemented in a way where each thread only fetches at most 2 texels.

The naive approach for reconstructing the normal takes two samples in each axis, the current and next one, converts all three depth values to world positions, calculates the vectors from the current world position to the next world positions, and finally computes the cross-product of the resulting two vectors. However, this approach struggles to produce correct normals at the edges. Figure 4.6 (1) illustrates how this naive implementation can lead to incorrect shadow ray culling at edges.

A more sophisticated approach, as described by Yuwen Wu [Wu] and also implemented in this renderer, takes advantage of using five depth samples per axis compared to the two of the naive approach. These samples include the current sample and the neighboring two samples in either direction. To calculate the derivative for both axes, the two extra samples for each direction are extrapolated to the center sample, creating two new points. The point that is closer to the actual center point is taken for the derivative calculation. Each depth value is first transformed into world space and subtracted from each other. Finally, the cross-product of the horizontal and vertical derivatives is calculated to get the normal. This approach promises to provide reconstructed normals with no artifacts on depth discontinuities or edges. There is however the edge-case where artifacts do occur on triangles that are less than 3 pixels across. This method is illustrated in Figure 4.6 (2).

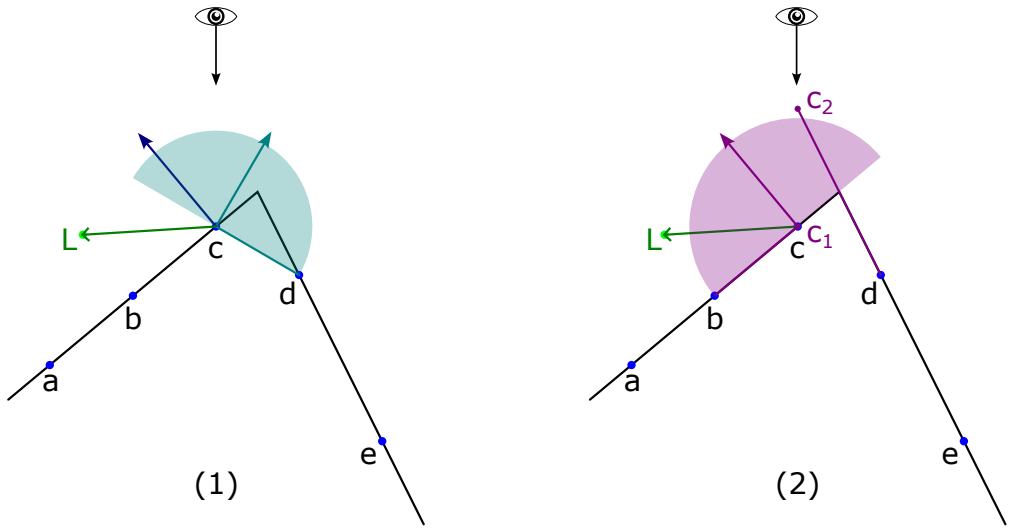


Figure 4.6: Normal Reconstruction: Geometry is sampled at 5 positions (a-e) with a light at L. The semi-circle represents all light directions where ray tracing still needs to be performed for shadow determination. In (1), the naive approach calculates the normal to the segment cd, which leads to c falsely being in shadow, even though the normal at c (blue) faces L. In (2), the improved approach extrapolated segments ab and ed to get the points c_1 and c_2, respectively. The extrapolated point c_1 is closer to the actual point c, so the normal to segment ab is correctly determined to be the surface normal at sample point c.

4.3.2 Optimized Depth-To-World-Position Conversion

Measures have been implemented to ensure that the world positions in both the shadow pass and shading pass are as close as possible to each other by minimizing floating-point errors in the depth-to-world-position conversion. Tracing rays for a different position from what is ultimately shaded could lead to incorrect shadows. Additionally, motivated by the frequent use of the conversion function, the number of operations required for calculating world positions from depth values is reduced to improve performance.

The Herbie web demo is used, which can optimize expressions by finding floating-point problems and suggesting alternative expressions [Gro]. Herbie optimizes the accuracy of the first step of the depth-to-world-space conversion where the screen-space position is transformed into Normalized Device Coordinates (NDC). The initial expression $((u + 0.5) * (1/w)) * 2 - 1$, which yielded a 99.6% accuracy, but Herbie optimized it to $\text{fma}\,(\frac{u+0.5}{w}, 2, -1)$ with a 100% accuracy. $\text{fma}\,(a, b, c)$ performs a fused multiply-add operation $a * b + c$. In the initial expression, u is the texel coordinates in the $[0, w]$ range, and w is the width of the window extent. The same expression is used to calculate the second component of the NDC vector.

Another accuracy optimization is simplifying from a matrix-vector multiplication to a

vector definition where the result of the original multiplication is inlined. The transformation from NDC to view space is done via the inverse of the used projection matrix. The used projection matrix, which can also be seen in Equation 2.1, has the following structure:

$$\begin{pmatrix} A & 0 & 0 & 0 \\ 0 & B & 0 & 0 \\ 0 & 0 & C & D \\ 0 & 0 & -1 & 0 \end{pmatrix} \tag{4.2}$$

The computer algebra system Mathcad [PTC] is used to find the simplification as seen in Equation 4.3. The inverse matrix is calculated using this formula: $A^{-1} = \frac{1}{\det A}\hat{A}^T$. The calculation of the determinant alone requires 28 multiplications, 12 subtractions, and 5 additions, as seen in the implementation of the determinant method for 4x4 matrices [Ols20]. It can be argued that this simplification results in fewer rounding errors since the inverse of the projection matrix is not needed anymore.

$$\begin{pmatrix} A & 0 & 0 & 0 \\ 0 & B & 0 & 0 \\ 0 & 0 & C & D \\ 0 & 0 & -1 & 0 \end{pmatrix}^{-1} \cdot \begin{pmatrix} s_x \\ s_y \\ depth \\ 1.0 \end{pmatrix} = \begin{pmatrix} \frac{s_x}{A} \\ \frac{s_y}{B} \\ -1.0 \\ \frac{depth+C}{D} \end{pmatrix} \tag{4.3}$$

The final version of the depth-to-world-position conversion function can be seen in Algorithm 4.1.

Algorithm 4.1: Conversion from depth value to world positions

```
1  Function world_pos_from_depth(vec2 screen, float depth):
2      vec2 ndc = fma((screen + vec2(0.5)) / vec2(data.extent), vec2(2), vec2(-1));
3      float A = camera.proj[0][0];
4      float B = camera.proj[1][1];
5      float C = camera.proj[2][2];
6      float D = camera.proj[3][2];
7      vec4 view_space = vec4(ndc.x / A, ndc.y / B, -1.0, (depth + C) / D);
8      view_space /= view_space.w;
9      vec4 world_space = camera.view_inv * view_space;
10     return world_space.xyz;
```

4.3.3 Tracing Shadow Rays

Each thread calculates the visibility via ray tracing for its corresponding texel to all lights in its tile where the angle between the surface normal and the light direction does not exceed 90°.

The direction of the ray is the normalized light vector pointing to the light. A constant t_{min} of 0.001 is used for ray tracing, skipping any intersections that are closer than 0.001 units from the origin of the ray to eliminate self-intersection artifacts. t_{max} is in the case of a point or spot light the distance between the light source and the current world position and for directional light a constant large distance of 1000.0 is used. Ray tracing flags `gl_RayFlagsTerminateOnFirstHitEXT` and `gl_RayFlagsOpaqueEXT` are set to stop at the first intersection and indicate that there is no transparency in place.

A 32-bit integer texture is used, with each texel representing a bitmask that stores the visibility information. For every light at index i in the tile's light list, if no intersection from the texel's world position with any geometry in the light direction is found, the i-th least significant bit is set in the bitmask. This bitmask is later used in shading pass to darken areas that are in shadow.

4.4 Shading

In the shading pass, the entire scene is rendered again, taking advantage of the already calculated depth buffer to minimize overshading. The usage of the depth buffer does not eliminate overshading, as shading is performed in 2x2 texel quads for the calculation of derivatives used for mipmapping, and even if only one of the four texels in the quad is shaded, all four have to be shaded [Wim23]. Since we only have opaque objects, the depth tests compare operator is set to `EQUAL`, which translates to only keeping fragments whose depth is the same as the one from the depth buffer.

To use the albedo and roughness-metallic textures in the shading pass, a descriptor set with variable length is used, which can be accessed with an index given in the material struct. This approach is called bindless textures, because it allows for fewer binds of textures, as all textures are bound at once, enabling the use of indirect draw calls.

This renderer employs the Cook-Torrance light model [CT82] to achieve physically based rendering, which is further described in more detail in Section 2.7. For the Fresnel function, the renderer uses Schlick's approximation [Sch94] and implements the GGX-Trowbridge Reitz normal distribution function. Additionally, the geometry function utilizes the Smith method in conjunction with the Schlick-GGX model [WMLT07].

Since we have a finite number of infinitely small light sources, we can replace the integral in Equation 2.2 with a sum. Furthermore, our objects do not emit light so the $L_e(\omega_0)$ term can be omitted. Shadows are introduced by adding a visibility term $v(x_s, \omega_l)$ that equals 1 if the screen-space position s_x of point x can see the light source L_i and 0 otherwise. The visibility term is queried from the shadow bitmask texture generated in the shadow pass. Additionally, only direct illumination is calculated.

With these considerations, the outgoing radiance equation simplifies to:

$$L_o(x, \omega_0) = \sum_l f_r(x, \omega_l, \omega_0) L_i(x, \omega_l) V(x_s, \omega_l)(\omega_l \cdot n) \tag{4.4}$$

CHAPTER 5

Results

We tested various tile sizes, the effect of light culling, and the application of the AABB optimization. Additionally, culling shadow rays using reconstructed normals is tested with both the naive and improved approach.

As the test scene, we used the widely used Sponza scene [DMM10] with ten point-lights spread around the scene with varying intensities, colors, and ranges. A fixed camera is positioned in a central location, to capture most of the scene's complexity.

All tests were performed on a machine running Windows 10 with a Ryzen 7 3700X CPU and Nvidia RTX 2060 SUPER GPU at a resolution of 1080p. The timings of each rendering stage are measured using Vulkan timestamp queries and are averages over the last second.

Enabled Optimisations	FT (ms)	Diff (%)
no optimisations	16	0.00
light culling	11.52	-28.00
light culling + AABB	10.93	-31.69
light culling + AABB + shadow ray culling	9.59	-40.06
light culling + AABB + improved shadow ray culling	9.67	-39.56
improved shadow ray culling	13.55	-15.31

Table 5.1: Frame time (FT) of different optimization configurations along with the difference in frame time in % compared to the reference performance with no optimizations. Measurements are performed using a tile size of 8x8.

Table 5.1 shows our performance measurements for different optimization configurations using a tile size of 8x8. The optimal configuration combines AABB-optimised light culling and shadow ray culling using the improved normal reconstruction. This configuration achieves the shortest frame time with the fewest artifacts, being 39.56 % faster than the

reference configuration with no optimization enabled. The primary bottleneck in the rendering pipeline is the ray-traced shadow pass, which on average takes up 92 % of the frame time.

The optimization that had the greatest impact on performance, after light culling, is the culling of shadow rays using reconstructed normals. Even with light culling disabled, this optimization improves performance by 15.31 %. Enabling the AABB optimization yields a frame time improvement of 5.12 %. The improved normal reconstruction method only takes 0.8 % longer than the naive approach.

Tests have shown that smaller tile sizes improve frame times but only to a small degree. With each halving of the tile size, the memory footprint of the light culling data structures increases by a factor of 4. Going from 16x16 to 8x8 improves the frame time by 1.25 % but requires about four times the memory.

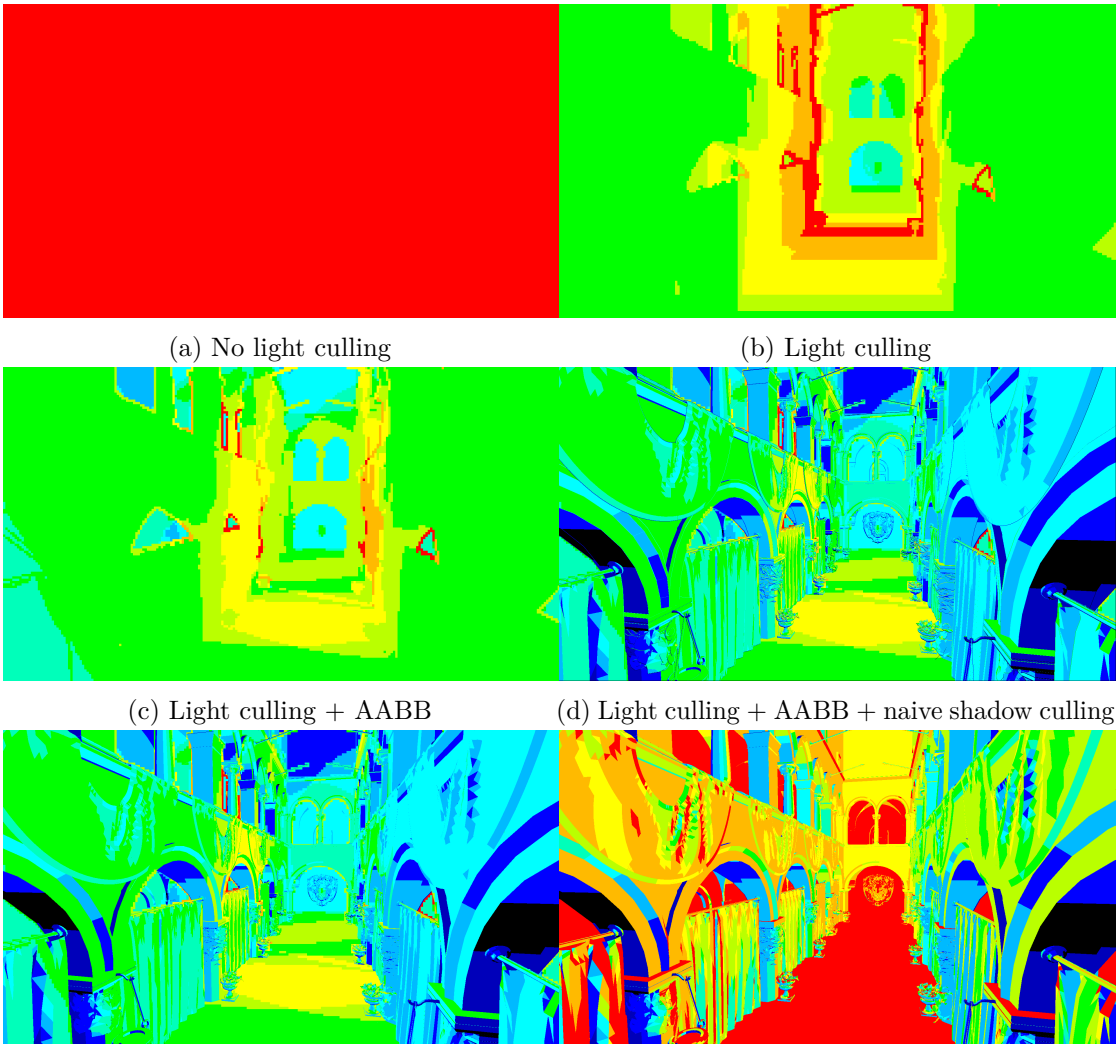
Figure 5.1: Rendered Image

The rendered image is shown in Figure 5.1. In Figure 5.2 several heatmaps visualize the number of traced rays for each pixel. In the heatmaps, black and blue indicate 0-3 rays, cyan and green 4-7 rays, and yellow and red 8-10 rays.

Figure 5.3 illustrates the naive and improved reconstruction along with the reference face normals. The error maps highlight how much the reconstructed normals differ from the reference normals. The naive approach has a mean difference of 0.07975 and the improved approach has a mean difference of 0.02264.

tile size	frame time (ms)
8x8	**9.44**
16x16	9.56
32x32	9.99

Table 5.2: Frame times of different tile sizes having all optimizations enabled.

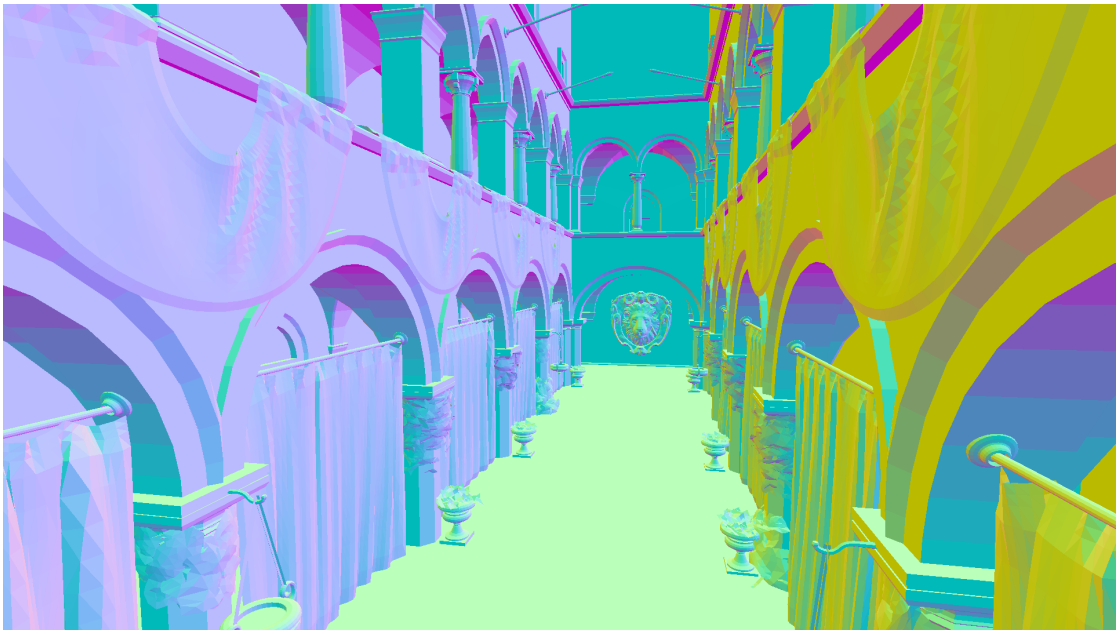


(a) No light culling (b) Light culling

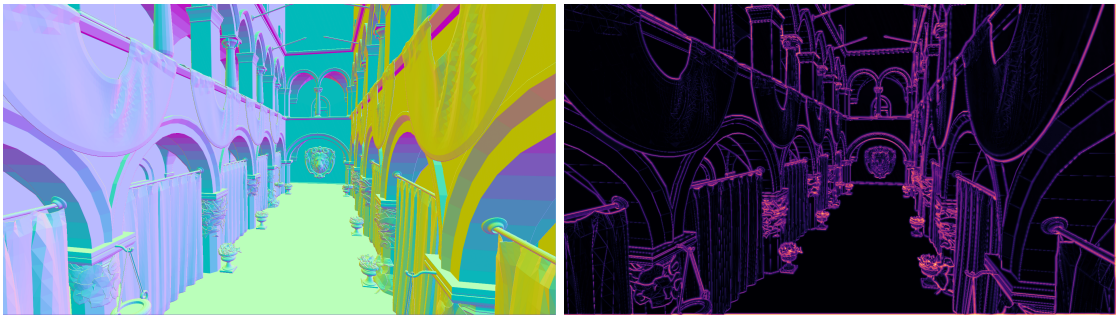
(c) Light culling + AABB (d) Light culling + AABB + naive shadow culling

(e) Light culling + AABB + imp. shadow culling (f) No light culling + imp. shadow culling

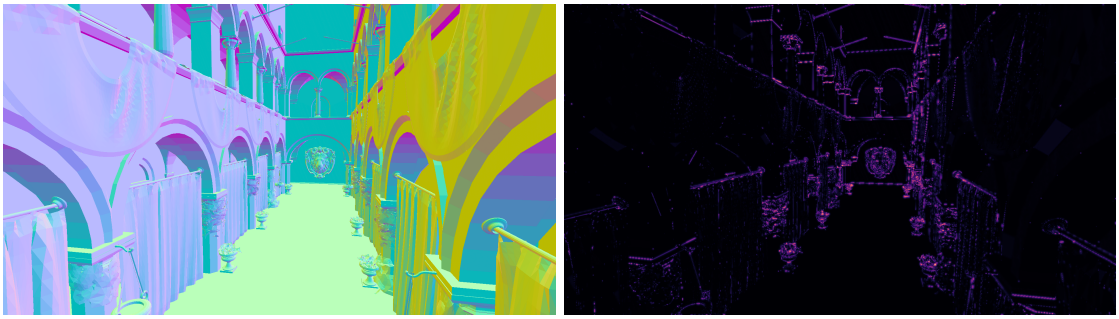
Figure 5.2: Ray Count Heatmaps: The heatmaps are computed in a shader within the rendering engine. The values in the heatmaps range from 0 to 10, with low counts represented in black and blue, medium counts in cyan and green, and high counts in yellow and red. Heatmaps (b-c) appear pixelated because they only use per-tile data, which remains constant across all the pixels in a tile. Note that the missing rays at edges in (d) result from the incorrect normal generated by the naive normal reconstruction approach.



(a) Reference



(b) Naive normal reconstruction with ꟻLIP Error Map to reference [ANA+20]



(c) Improved normal reconstruction with ꟻLIP Error Map to reference [ANA+20]

Figure 5.3: Normal reconstructions: Visualization of reconstructed normals along with their respective error map to the reference normals.

CHAPTER 6

Conclusion and Limitations

This implementation of a hybrid renderer has a few limitations. First, it only supports hard shadows, as all light sources are punctual, resulting in sharp shadow edges. This pixel-perfect nature, without any filtering, leads to aliasing artifacts. Additionally, the renderer limits the number of lights per tile to 32 to fit the binary visibility information of a pixel within a tile into a single 32-bit integer. Tracing more than 32 lights per pixel in a tile would result in less than real-time rendering performance.

Another limitation is that the renderer only supports opaque objects and cannot handle transparent or translucent materials. Supporting those would require more logic in both the light culling step and shadow calculation, which is beyond the scope of this project.

Despite these limitations, this thesis demonstrates the viability of using a Forward+ pipeline for hybrid rendering. We showed that rather than using a G-Buffer that includes world positions, it is possible to convert the already existing depth values from the Forward+ depth-prepass to world positions for ray tracing purposes.

Moreover, light culling has proven to be a significant performance improvement, as it reduces the number of lights to consider and, consequently, the number of rays that need to be traced. As the shadow pass takes the most time to calculate, reducing the number of rays improves performance. It is therefore worth optimizing the light culling stage to minimize the number of false positives. Even optimizations such as using AABBs to better bound the tile's volume, which slow down the light culling stage, have a net positive impact, as ray tracing is the primary bottleneck.

Another ray reduction method, resulting in a substantial performance gain, is culling shadow rays for surfaces that face away from the light. We showed that the required surface normal can be accurately reconstructed using the depth values from the depth pre-pass, as long as the triangle size is greater than 3 pixels across.

In conclusion, the result of this work demonstrates that Forward+ is a viable option for real-time hybrid rendering applications.

Overview of Generative AI Tools Used

During the writing of this thesis, ChatGPT 4o and Grammarly were used to find better formulations and for proofreading.

List of Figures

List of Tables

List of Algorithms

Bibliography

[AMHH19] Tomas Akenine-Moller, Eric Haines, and Naty Hoffman. *Real-time rendering*. AK Peters/crc Press, 2019.

[ANA+20] Pontus Andersson, Jim Nilsson, Tomas Akenine-Möller, Magnus Oskarsson, Kalle Åström, and Mark D. Fairchild. ꟻLIP: A Difference Evaluator for Alternating Images. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 3(2):15:1–15:23, 2020.

[BWB19] Jakub Boksansky, Michael Wimmer, and Jiri Bittner. *Ray traced shadows: maintaining real-time frame rates*, pages 159–182. Springer, 2019.

[CPC84] Robert L Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. *ACM SIGGRAPH Computer Graphics*, 18(3):137–145, 1984.

[CT82] Robert Cook and Kenneth Torrance. A reflectance model for computer graphics. *ACM Trans. Graph.*, 1:7–24, 01 1982.

[DMM10] Marko Dabrovic, Frank Meinl, and Morgan McGuire. Sponza model. `https://github.com/KhronosGroup/glTF-Sample-Assets/tree/main/Models/Sponza`, 2010. Accessed: 2024-09-22.

[DNL+17] Yangdong Deng, Yufei Ni, Zonghui Li, Shuai Mu, and Wenjun Zhang. Toward real-time ray tracing: A survey on hardware acceleration and microarchitecture techniques. *ACM Comput. Surv.*, 50(4), August 2017.

[dV] Joey de Vries. LearnOpenGL - Deferred Shading. `https://learnopengl.com/Advanced-Lighting/Deferred-Shading`. Accessed: 2024-07-16.

[Eri05] Christer Ericson. Chapter 5 - basic primitive tests. In Christer Ericson, editor, *Real-Time Collision Detection*, The Morgan Kaufmann Series in Interactive 3D Technology, pages 125–233. Morgan Kaufmann, San Francisco, 2005.

[Gro] UW PLSE Group. Herbie: Automatically improving floating point accuracy. `https://herbie.uwplse.org/`. Accessed: 2024-08-08.

[HH] Tom Hulton-Harrop. Reverse z (and why it's so awesome). https://tomhultonharrop.com/mathematics/graphics/2023/08/06/reverse-z.html. Accessed: 2024-07-25.

[HMY12] Takahiro Harada, Jay McKee, and Jason C Yang. Forward+: Bringing deferred lighting to the next level. In *Eurographics (Short Papers)*, pages 5–8, 2012.

[Int22] Intel. Introduction to the xe-hpg architecture. https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-the-xe-hpg-architecture.html, 2022. Accessed: 2024-09-08.

[Jer] Jeremiah. Forward+ rendering. https://www.3dgep.com/forward-plus/. Accessed: 2024-06-08.

[Kaj86] James T Kajiya. The rendering equation. *ACM SIGGRAPH Computer Graphics*, 20(4):143–150, 1986.

[MOB+21] Daniel Meister, Shinji Ogaki, Carsten Benthin, Michael J. Doyle, Michael Guthe, and Jiří Bittner. A survey on bounding volume hierarchies for ray tracing. *Computer Graphics Forum*, 40(2):683–712, 2021.

[MPC19] Pierre Moreau, Matt Pharr, and Petrik Clarberg. Dynamic many-light sampling for real-time ray tracing. In *ACM/EG Symposium on High Performance Graphics (HPG)*, June 2019.

[NVI18] NVIDIA. Nvidia turing gpu architecture: Graphics reinvented. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf, 2018. Accessed: 2024-09-08.

[OA11] Ola Olsson and Ulf Assarsson. Tiled shading. *Journal of Graphics*, GPU:235–251, 11 2011.

[OBA12] Ola Olsson, Markus Billeter, and Ulf Assarsson. Clustered Deferred and Forward Shading. In Carsten Dachsbacher, Jacob Munkberg, and Jacopo Pantaleoni, editors, *Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics*. The Eurographics Association, 2012.

[Ols20] Gray Olson. ultraviolet. https://github.com/fu5ha/ultraviolet/blob/377b0016211ddebc521a0ee784774d85118de2b0/src/mat.rs#L1353-L1371, 2020. Accessed: 2024-08-08.

[Pom21] Andrew Pomianowski. Rdna™ 2 gaming architecture. In *2021 IEEE Hot Chips 33 Symposium (HCS)*, pages 1–18, 2021.

[PTC] PTC Inc. Mathcad: Engineering Math Software. https://www.mathcad.com/en/. Accessed: 2024-08-08.

[Ree] Nathan Reed. Depth precision visualized. https://developer.nvidia.com/content/depth-precision-visualized. Accessed: 2024-07-22.

[Sch94] Christophe Schlick. An inexpensive brdf model for physically-based rendering. In *Computer graphics forum*, volume 13, pages 233–246. Wiley Online Library, 1994.

[tur18] turanszkij. Optimizing tile-based light culling. https://wickedengine.net/2018/01/optimizing-tile-based-light-culling/, January 2018. Accessed: 2024-07-24.

[Whi80] Turner Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, 1980.

[Wim23] Michael Wimmer. Real-time rendering - rendering pipeline lecture. Lecture, Vienna University of Technology, 11 December 2023, 2023. Accessed: 2024-08-08.

[WMLT07] Bruce Walter, Stephen R Marschner, Hongsong Li, and Kenneth E Torrance. Microfacet models for refraction through rough surfaces. *Rendering techniques*, 2007:18th, 2007.

[WS19] Turner Whitted and Martin Stich. Foreword. *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*, pages 15–19, 2019.

[Wu] Yuwen Wu. Accurate normal reconstruction from depth buffer. https://atyuwen.github.io/posts/normal-reconstruction. Accessed: 2024-07-27.