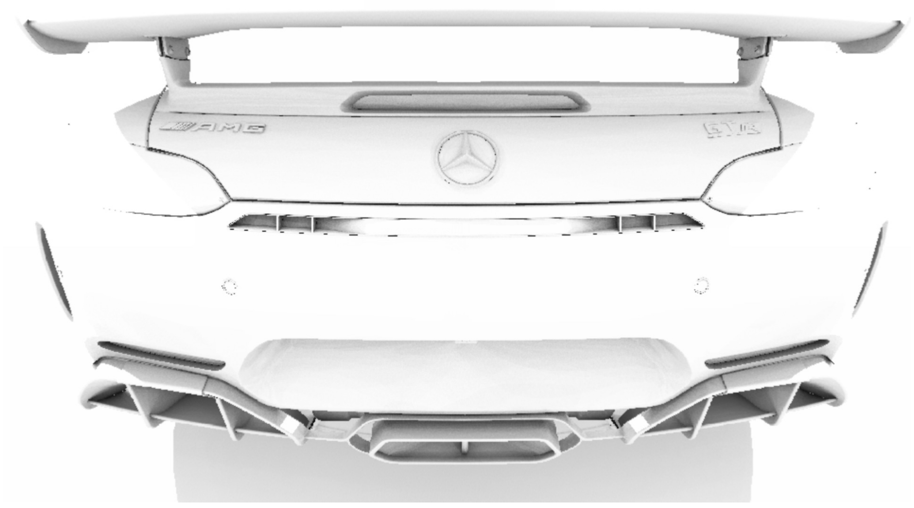
Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Daniel Fedai Larsen

Approximating Global Illumination with Real-Time Ambient Occlusion

Master's thesis in Informatics
Supervisor: Theoharis Theoharis, Bart Iver van Blokland
May 2019



Real-Time Ambient Occlusion (model courtesy of Thomas F)

NTNU
Kunnskap for en bedre verden

Daniel Fedai Larsen

Approximating Global Illumination with Real-Time Ambient Occlusion

Master’s thesis in Informatics
Supervisor: Theoharis Theoharis, Bart Iver van Blokland
May 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

To my parents who always push me to do my best, whilst still giving me the freedom to chose what to pursue.

I would also like to thank

Theoharis Theoharis for being an excellent supervisor and teacher,

Bart Iver van Blokland for (literally) always having his door open,

and Marius Skogen, for whom without the lunches throughout the last academic year would have been a lot more dull.

Abstract

Approximating *global illumination* has long been a problem that real-time applications, such as games, have been trying to solve efficiently. For years, the best and most efficient techniques have been *ambient occlusion* algorithms based in *screen-space*, such as *Horizon-Based Ambient Occlusion*. Although efficient, such algorithms can suffer from artifacts, and a lack of realism due to being screen-space algorithms trying to approximate a *world space* effect.

In 2018, *Nvidia* released the first ever hardware accelerated *ray tracing graphical processing units*. Proper ray traced ambient occlusion could now be a viable alternative to screen-space algorithms. However, to be a viable alternative, the results need to be of a certain quality, and its performance must be in the order of a few milliseconds.

This thesis will show that although real-time ambient occlusion is possible, and its results are of reasonable quality, it is still not quite ready to be used in real-time applications such as games. This is due to some visual patterns in the results that are caused by the low sample count when calculating ambient occlusion, its scene- and viewpoint-dependent performance, and lastly the time required to rebuild acceleration structures.

Sammendrag

Å tilnærme *global illuminasjon* har lenge vært et problem sanntids applikasjoner som spill har prøvd å løse effectivt. I mange år har de beste og mest effektive teknikkene vært *omgivende okklusjon* algortimer basert i *skjerm-rom*. Ett eksempel er *Horizon-Based Ambient Occlusion*. Selv om slike algoritmer er effektive, kan de lide av uønskede effekter, og en mangel på realisme, da de er skjerm-rom algoritmer som prøver å tilnære en *verden-rom* effekt.

I 2018 lanserte *Nvidia* de første maskinvare akselererte "*ray tracing*" *grafikkortene*. Ordentlig "ray traced" omgivende okklusjon kan nå være et alternativ til skjerm-rom algoritmer. Men for å være et realistik alternativ må resultatene være av en viss kvalitet, og utregningstiden må være ikke være mer enn noen millisekunder.

Denne avhandlingen vil vise at selv om sanntids omgivende oklusjon nå er mulig, og resultatene er av relativt god kvalitet, er det fremdeles ikke helt klart til å tas i bruk i spill. Dette er grunnet uønskede visuelle effekter som skapes av det lave antallet prøver som må brukes under utregningen av omgivende okklusjon, dens scene- og synspunkt-avhengige utregningstid, og tiden som kreves for å gjenoppbygge akselerasjonsstrukturene.

Contents

Abstract **iii**

Sammendrag **iv**

Contents **vii**

Abbreviations **viii**

1 Introduction **1**
- 1.1 Background 1
- 1.2 Thesis Goal 4

2 Path Tracing Theory **6**
- 2.1 The Path Tracing Algorithm 6
- 2.2 Calculating Ray Directions for Each Pixel 7
- 2.3 Sub-Pixel Sampling 8
- 2.4 Intersection Testing 9
  - 2.4.1 Acceleration Structures 9
- 2.5 Physically Based Rendering 9
  - 2.5.1 The Rendering Equation 10
  - 2.5.2 Monte Carlo Estimation 11
  - 2.5.3 Bidirectional Reflectance/Transmission Distribution Function 12
  - 2.5.4 Working With BSDFs 15
  - 2.5.5 Fresnel Equations 16
  - 2.5.6 Sampling Directions 17
  - 2.5.7 Sampling Geometrical Shapes 20
  - 2.5.8 Area Lights 22

3 Path Tracing Engine **24**
- 3.1 Scene Creation 25
- 3.2 Main Entry Point 26
- 3.3 The Rendering Loop 26
- 3.4 Calculating Ray Directions 27
- 3.5 Intersection Testing 27
  - 3.5.1 Brute-Force Method 27
  - 3.5.2 Acceleration Structures 28
- 3.6 Materials and BSDFs 29
- 3.7 Choosing a BxDF to Sample 30

3.8 Sampling BxDFs 30
3.9 Account for BxDFs That Were Not Chosen 30
3.10 Fresnel Equations 30
3.11 Parallelization 31
3.12 Rendering Results 34

4 Ray Tracing on GPUs 37
4.1 Graphics Application Programming Interface 37
4.2 Ray Tracing in Vulkan 39
4.2.1 Vulkan at Its Core 39
4.2.2 Shaders and SPIR-V 40
4.2.3 Vulkan Extensions 41
4.2.4 VK_NV_ray_tracing 41

5 Approximating Global Illumination with Real-Time Ambient Occlusion 44
5.1 Ambient Occlusion 44
5.2 Screen-Space Ambient Occlusion 45
5.3 Real-Time Ambient Occlusion 45
5.3.1 Theoretical Details 45
5.3.2 Implementation Details 48

6 Real-Time Ambient Occlusion Results 54
6.1 Visual Quality 54
6.1.1 Real-Time AO vs Physically Based Global Illumination . 54
6.1.2 Real-Time AO vs Offline AO 56
6.1.3 Visibility Function Comparison 58
6.1.4 Box Filter Comparison 60
6.1.5 Temporal Integration Comparison 61
6.2 Performance 62

7 Limitations and Future Work 65
7.1 Current Implementation Limitations 65
7.2 Implementation Improvement and Future Work 66
7.2.1 Sampling 66
7.2.2 Sampling Pattern Mitigation 66
7.2.3 Temporal Integration 66
7.2.4 GPU Performance 66
7.3 Conclusion 67

Appendix I
Source Code I
.brhan File Format I
Comparison Images II
Additional Images IV

Bibliography VI

Abbreviations

AA	=	Anti-Aliasing
AO	=	Ambient Occlusion
API	=	Application Programming Interface
BRDF	=	Bidirectional Reflectance Distribution Function
BRTF	=	Bidirectional Transmission Distribution Function
BSDF	=	Bidirectional Scattering Distribution Function
CG	=	Computer Graphics
CGI	=	Computer-Generated Imagery
CPU	=	Central Processing Unit
FOV	=	Field-of-View
GPU	=	Graphics Processing Unit
HBAO	=	Horizon-Based Ambient Occlusion
L1-3	=	Level 1-3
MIS	=	Multiple Importance Sampling
ms	=	Milliseconds
OS	=	Operating System
PBR	=	Physically Based Rendering
PDF	=	Probability Distribution Function
RGB	=	Red-Green-Blue
SPD	=	Spectral Power Distribution
SPP	=	Samples Per Pixel
SSAO	=	Screen-Space Ambient Occlusion

Chapter 1

Introduction

"The only purpose of any code is to transform data. It has to get it from form *A* to form *B*." [1]

- Mike Acton, CppCon 2014

All of computing essentially boils down to a data problem: given some initial data, how to best transform it to obtain the desired result? The field of computer graphics (CG) is no exception. In CG, there is the sub-field of visualization, where the desired result is some image. This means that the problem visualization systems are trying to solve can be narrowed down from its general form to: how to best *visualize* the data? In addition to this primary problem, there is another one that is almost just as important: how to visualize the data *efficiently*? As the reader will see, these two problems lay the foundation for the question this thesis is trying to answer.

1.1 Background

Although all visualization systems share the two main problems, different visualization systems can have different goals. Usually, a terminal interface is not overly concerned with rendering an image with the highest level of graphical fidelity. Easy to read text and a good overview of the current workflow is the main focus. Meanwhile, a seismic visualization system is likely concerned with rendering an image that is easy for its users to understand and interpret. Game and film rendering systems usually have a common visualization goal: to create the highest fidelity image given the available time. However, the level of fidelity, and the available time, varies greatly between games and film.

In film, the requirements for image fidelity are high. This is as true now as it was in the 70's and 80's, although the overall demands are higher now compared to then. *Futureworld* (1976) was the first film that featured 3D computer-generated imagery (CGI) elements: an artificial hand and face[2]. It took many more years before CGI became the standard for special effects and entire feature films[3]. The reason for this was that the visualization systems of the time were not able to solve the two aforementioned problems to a satisfying degree.

Although continuous improvements were made to the systems of the time, it was not until the introduction of the *path tracing* algorithm that CGI in film really took off. Previously, the *rasterization* algorithm had been used exclusively. Path tracing did not replace rasterization instantly. The reason for this was due to a fundamental challenge with path tracing when compared to rasterization: it is much more computationally expensive. However, with the development of faster and more parallel hardware, both central processing units (CPUs) and graphical processing units (GPUs), the cost of path tracing has decreased. This resulted in the first *fully* path traced, physically based, film, *Monsters University*, being released by Pixar in 2013. A screenshot is shown in Figure 1.1.

Figure 1.1: Monsters University[4]

Games have a different story than film, which is caused by a key difference between rendering in games and film. As games are interactive systems, they are required to create each image in real-time, i.e. less than 33 milliseconds (ms). For film, this constraint is not nearly as hard, with a single image being allowed several hours to be rendered. This difference means that while CGI in film has moved from rasterization to path tracing, the same shift has not been possible in games. Nonetheless, CGI in games have come a long way since its early days. One of the games that pioneered 3D real-time graphics was *Wolfenstein 3D* (1992). A screenshot can be seen in Figure 1.2. Although often viewed as a breakthrough for its time[6], both graphically and from a gameplay perspective, its graphics are almost non-comparable to what is achievable today. A good example of what today's real-time visualization systems are capable of creating is *Battlefield V*. A screenshot is shown in Figure 1.3.

Although it achieves a high level of realism for today's standards in games, the graphics in Battlefield V are seldom mistaken for real images. There are many reasons for this, but one is lighting. One of the most important aspects of creating an approximation of reality, i.e. *physically based rendering* (PBR), is global illumination. As opposed to local illumination, which only considers direct light, global illumination also takes indirect light into account. While path tracing is a global illumination algorithm, rasterization is not. And it is rasterization which is used in games. Thus, the need for an approximation of global illumination is needed: *ambient occlusion* (AO). AO evaluates the space

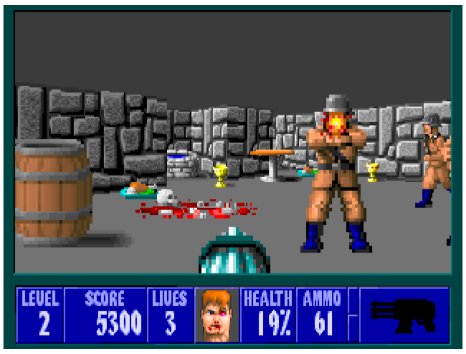


Figure 1.2: Wolfenstein 3D[5]

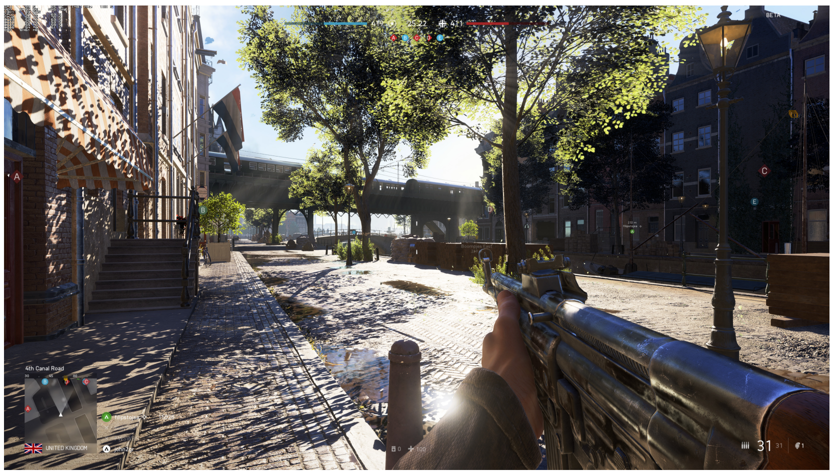


Figure 1.3: Battlefield V[7]

around a point in the scene, and calculates how exposed it is. The more the point is exposed, the more likely it is that indirect light will reach it, and be reflected in a given direction[8]. An example of what AO looks like when visualized can be seen in Figure 1.4.

AO was used heavily in the film industry before global illumination became the standard. This is due to its lower cost. However, although not as expensive as global illumination, it is still not cheap. AO is based on the underlying algorithm of path tracing: *ray tracing*. This has meant that until now, AO has been too expensive for real-time systems such as games. It has been reserved for offline systems, or real-time systems with certain characteristics, where AO could be pre-calculated. This led to an approximation of AO being developed: *screen-space ambient occlusion* (SSAO)[9]. It provided comparable results to AO at a much lower computational cost, and has been the standard in games since the mid 2000's.

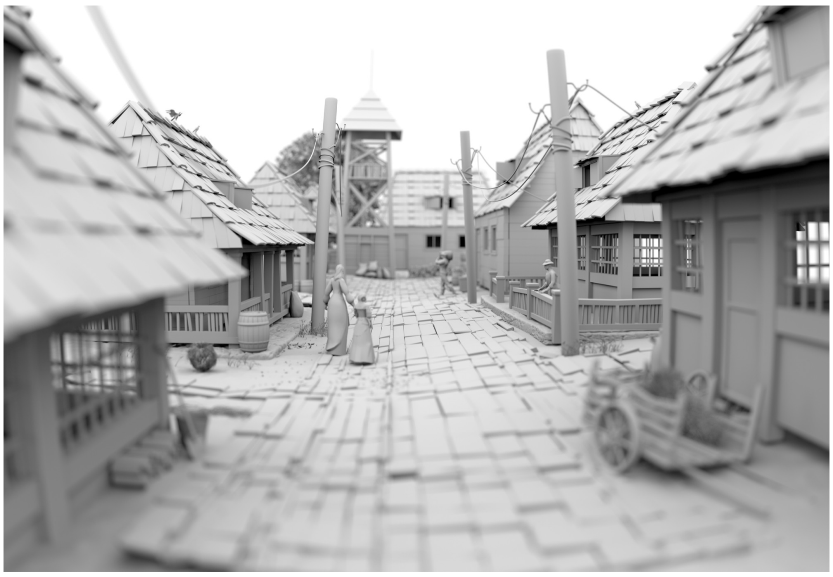


Figure 1.4: Example of ambient occlusion visualization.[8]

1.2 Thesis Goal

Although SSAO, and its leading variation, *horizon-based ambient occlusion* (HBAO+)[10], have helped games improve their graphical fidelity, proper AO would still be an upgrade. Therefore, when Nvidia, one of the major GPU manufacturers, announced and released the first ever GPUs with hardware-accelerated ray tracing capabilities in 2018, the question of whether real-time AO is achievable suddenly became relevant. The work in this thesis has been done to answer that very question:

RQ: Is real-time ambient occlusion a viable option for games?

In order to try and answer this question, the following main tasks were set:

- Get a thorough understanding of the theory behind physically based rendering and global illumination. This would allow for a better understanding of what AO is trying to approximate.
- Create a physically based renderer. Doing this would verify that the understanding obtained in the previous point was indeed correct. This could be accomplished by comparing against other physically based renderers.
- Get a thorough understanding of the theory behind AO. Without this, a real-time AO version could not be implemented.
- Create a real-time AO implementation.
- Compare its visual results against those of a non-real-time AO implementation, and the global illumination result. For real-time AO to be a viable option, its results must be comparable to that of offline AO. Also, by comparing against the global illumination result, insight into how well the real-time version approximates the "overall" target is obtained.

- Evaluate the real-time AO implementation's performance. For real-time AO to be a viable option, its performance must be in the range of real-time, and preferably in the range of HBAO+'s performance.

The remainder of this thesis will go through each of these tasks in order, and explain both theoretical and implementation details necessary to understand the results and conclusion presented at the end. Finally, any shortcomings and possible improvements and future work will be discussed.

Chapter 2

Path Tracing Theory

This chapter will go into the theoretical concepts that are needed to create a physically based renderer using the path tracing algorithm. The book *Physically Based Rendering - From Theory to Implementation, 3rd Edition* authored by Matt Pharr, Wenzel Jakob and Greg Humphreys[11] is the basis upon which this chapter is built.

2.1 The Path Tracing Algorithm

Given the pixels in an image, the goal of any renderer is to fill each pixel with a color in such a way as to most accurately depict the scene that lies "beyond" the image. Renderers that utilize the path tracing algorithm accomplish this by tracing rays through each of the pixels in the image. This process involves tracing multiple rays per pixel, and following a specific path for each ray that is traced.

A path is created by first intersecting the scene from the given pixel, and finding the closest point of intersection. From this point, the next ray direction is calculated, and the next intersection point is found. This process repeats itself until a maximum number of "bounces" is reached, or no intersection is found.

At each intersection point, lighting calculations are performed. This involves determining how much light is arriving at the intersection point, and how much of it that is reflected, or transmitted, in a given outgoing direction (explained in greater detail in Section 2.5). When the path tracing algorithm terminates for a given sample for a given pixel, light has essentially been accumulated along the path according to the properties at the surface points that were intersected.

Due to the nature of how the lighting calculations are performed (see Section 2.5.2), multiple rays are needed for each pixel to approximate the color accurately. After all samples for a pixel have been taken, the accumulated value is averaged and stored in the final image.

When the process is finished for all pixels, the image can be saved for later viewing. It can also be saved in a sequence of images to create a film, or it can be displayed to the screen immediately for an interactive experience.

2.2 Calculating Ray Directions for Each Pixel

When the color of each pixel in the image is to be calculated, at least one ray has to be sent through each pixel. To calculate the direction of each ray given its pixel coordinate (x, y), a few other pieces of information are needed:

- World's up direction: **vec3**(*worldUp*)
- Film plane dimensions: **uint**(*filmWidth*), **uint**(*filmHeight*)
- Camera's position: **vec3**(*camOrigin*)
- Camera's view direction: **vec3**(*camViewDir*)
- Camera's vertical field-of-view (FOV) in degrees: **float**(*camFOV*)

Given this information, the following values can be calculated:

$$\begin{aligned}
\theta &= DegreesToRadians(verticalFOV)\\
aspectRatio &= filmWidth/filmHeight\\
lensHalfHeight &= \frac{tan(\theta)}{2}\\
lensHalfWidth &= lensHalfHeight * aspectRatio\\
camRight &= Normalize(Cross(camViewDir, worldUp))\\
camUp &= Normalize(Cross(camRight, camViewDir))\\
lensTopLeftCorner &= camOrigin + (lensHalfWidth * (-camRight))\\
&\quad + (lensHalfHeight * camUp) + camViewDir\\
lensHorizontalEnd &= lensWidth * 2 * camRight\\
lensVerticalEnd &= lensHalfHeight * 2 * (-camUp)
\end{aligned}$$

These values are showcased in Figure 2.1.

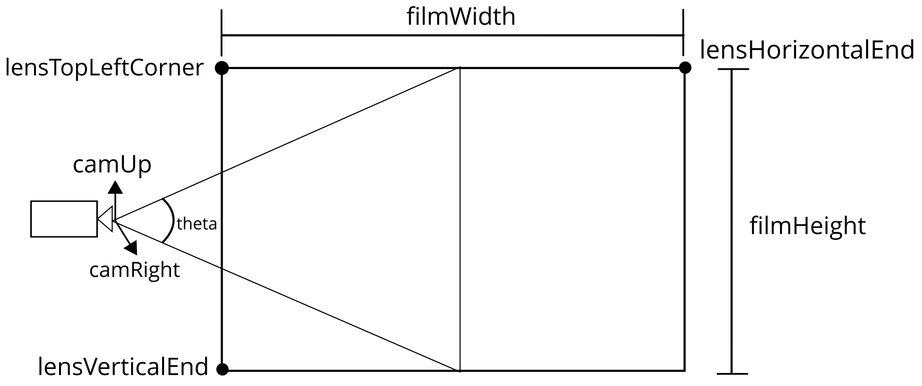


Figure 2.1: Visualization of calculated camera values which are needed for generating rays.

With these values, and the normalized coordinates for a given pixel, (x_n, y_n), the properties of the ray can be calculated as follows:

```
vec3 rayOrigin = camOrigin;
vec3 rayDir = normalize(lensTopLeftCorner +
                        (xn * lensHorizontalEnd) +
                        (yn * lensVerticalEnd) -
                        rayOrigin);
```

The normalized coordinates for a pixel are found as shown below:

```
float xn  = (float(x) + 0.5f) / float(filmWidth);
float yn  = (float(y) + 0.5f) / float(filmHeight);
```

Note that the addition of 0.5 is necessary because the normalized pixel coordinates are at the center of the pixel, and not the top-left corner.

The method shown is a re-implementation of Pete Shirley's code for generating rays from his book series *Ray Tracing in One Weekend*[12]. A visualization of the ray tracing can be seen in Figure 2.2.

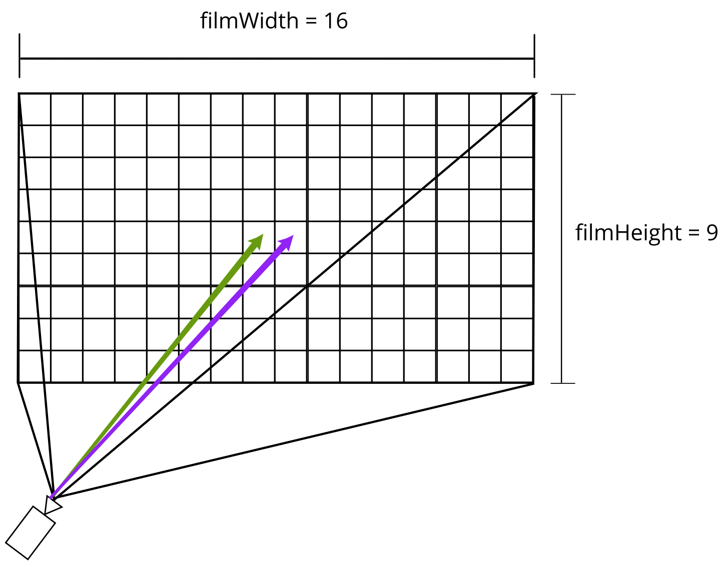


Figure 2.2: Visualization of rays being cast through the film plane and into the scene.

2.3 Sub-Pixel Sampling

As mentioned, it is necessary to send multiple rays through a pixel to get a good approximation of the color for that pixel. One option is to only sample using the normalized pixels coordinates, i.e. the center of the pixel. While this will give good results, only sampling the pixel's center leaves most of the pixel unsampled. This is undesirable, as the small differences in ray directions that sub-sampling a pixel would yield, naturally acts as an anti-aliasing (AA) technique[13]. Thus, performing sub-pixel sampling is desirable as it improves the final result with little to no performance impact. When doing sub-pixel sampling, the goal is to trace rays from a distribution such that as much of the pixel's area has been sampled. The rays could be evenly spaced, or picked from a random distribution, e.g. white noise or blue noise[14].

2.4 Intersection Testing

One of the most important parts of any path tracer is its ability to check whether a ray intersects any of the shapes in the scene. These shapes are usually triangles or spheres. However, there also exist other types of shapes, e.g arbitrary parametric surfaces, that are possible to trace rays against[15]. The mathematical derivations for finding ray-triangle[16] and ray-sphere[17] intersections are well studied, and are not included here.

2.4.1 Acceleration Structures

To reduce the time needed to find the closest intersection point, much research has been done on creating acceleration structures[18]. One of the most common primitive subdivision acceleration structures is a *bounding volume hierarchy* (BVH). A BVH is a data structure that partitions geometric primitives, e.g. triangles, into a hierarchy of disjoint sets. It is a tree structure where primitives are stored in the leaf nodes, while internal nodes store a bounding-box which encapsulates all the primitives in the leaf nodes below it. When a ray is to intersect a BVH, it first checks for intersections with bounding boxes, upon which two scenarios are possible:

1. If the ray intersects a bounding box:
    (a) All internal nodes (bounding boxes) directly below it are added to a queue of nodes that need to be checked for intersections.
    (b) All leaf nodes (primitives) directly below it have to be checked for intersections.
2. If the ray *does not* intersect a bounding box, all internal nodes and leaf nodes below the node containing that bounding box can be skipped.

Another type of primitive subdivision acceleration structure is the *kd-tree*. Although kd-trees generally provide faster intersection testing than BVHs, BVHs are faster to build, and are less prone to missed intersections due to rounding error in floating-point arithmetic[11].

2.5 Physically Based Rendering

In the 70's and 80's, the realism of the rendered images in film was highly limited by the available hardware. Path tracing was not yet viable, and rasterization was the best option. Due to the nature of rasterization, local illumination models, such as the *Phong Illumination Model*[19], and the *Cook-Torrance Reflection Model*[20], were common. Although these models produce believable results, being local illumination models, their approximation of reality is highly limited. However, as hardware got faster, path tracing was no longer a distant dream, and allowed for global illumination models to be used. This then begs the question: how to describe how light, both direct and indirect, interacts between objects in a scene in a general way?

2.5.1 The Rendering Equation

The answer: the *Rendering Equation* introduced by James Kajiya at SIGGRAPH in 1986[21]:

$$L_o(p, \omega_o) = L_e(p, \omega_o) + \int_{\mathcal{H}} f(p, \omega_o, \omega_i) L_i(p, \omega_i) |cos\theta_i| d\omega_i \tag{2.1}$$

The equation has six terms that in words can be explained as follows:

- $L_o(p, \omega_o)$: The total amount of *radiance exitance* from point p in direction ω_o.
- $L_e(p, \omega_o)$: The amount of *emitted* radiance exitance from point p in direction ω_o.
- $\int_{\mathcal{H}} d\omega_i$: An integral over the domain of directions that ω_i can be sampled from. Here the domain is the hemisphere about the normal of the surface at p, hence the notation $\mathcal{H}$.
- $f(p, \omega_o, \omega_i)$: The fraction of *irradiance* arriving along ω_i at p, that is reflected in direction ω_o given the material properties of the surface at p.
- $L_i(p, \omega_i)$: The amount of irradiance at p arriving along ω_i
- $|cos\theta_i|$: The "light attenuation" factor which reduces the amount of incident light as the angle between the incident light and the surface normal increases. The reasoning behind this is that the same amount of light is spread across a larger area when this angle is larger. This is visualized in Figure 2.3.

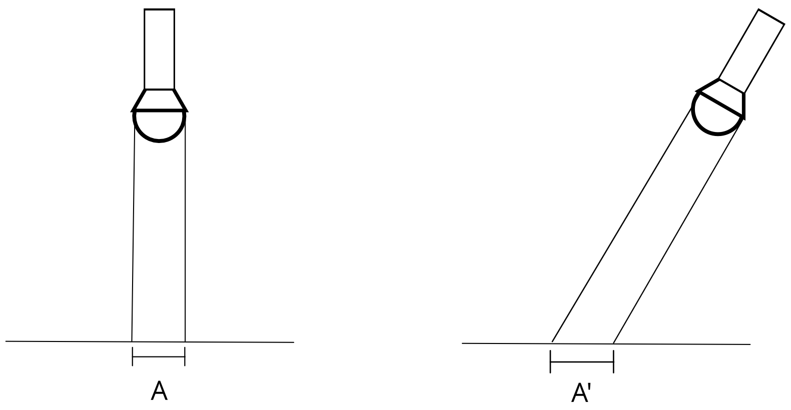


Figure 2.3: Comparison of same amount of light hitting a surface at different angles: notice how A (left) is smaller than A' (right).

Figure 2.4 shows a visualization of what the various notations in the rendering equation represent.

However, the rendering equation, specifically the integral part, has one major issue: it cannot be solved analytically. The reason for this is that any *one* of

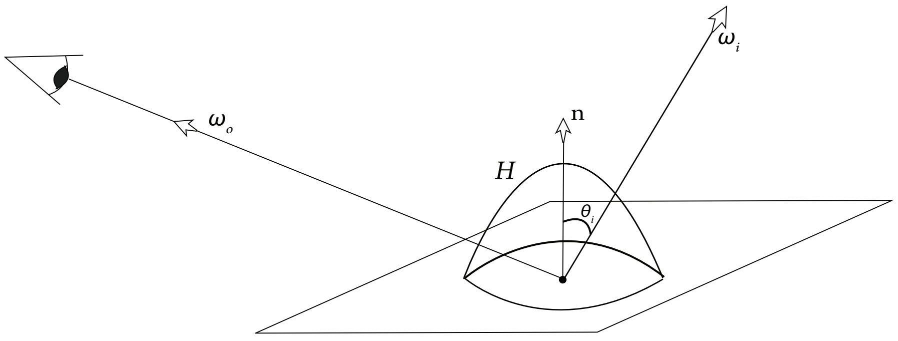


Figure 2.4: The geometric setting and notation used throughout this thesis.

the $L_i(p, \omega_i)$ terms that is needed to solve a given $L_o(p, \omega_o)$, requires that very same $L_o(p, \omega_o)$ term to be solved. This dependence can also be explained as follows:

1. To find $L_o(p_0, \omega_o)$ for a point p_0, integration over its domain, $\mathcal{H}_0$, is performed. This domain includes a direction ω_i which will intersect the scene at point p_1.
2. To find $L_i(p_0, \omega_i) = L_o(p_1, -\omega_i)$, integration over its domain, $\mathcal{H}_1$, is performed. This domain includes a direction ω_j which will intersect the scene at p_0.

Continuing this process results in infinite recursion of rendering equations within rendering equations. Fortunately, the power of numerical integration can be applied here instead. For this particular problem, *Monte Carlo Estimation* is utilized.

2.5.2 Monte Carlo Estimation

A Monte Carlo Estimator can approximate the value of an arbitrary integral[11].

$$\int_a^b f(x)dx$$

Given a sequence of uniform random variables $X_i \in [a, b]$, the Monte Carlo Estimator becomes:

$$\int_a^b f(x)dx = \frac{b-a}{N} \sum_{i=1}^{N \to \infty} f(X_i) \tag{2.2}$$

However, this version can only be used when each $X_i \in [a, b]$ has the same probability, $\frac{1}{b-a}$, of being chosen. Since this is not always desirable when rendering ("intelligent" sampling can help reduce the number of samples needed for the estimated value to converge), it is also useful to have a general version that takes the probability of sampling the value X_i, $p(X_i)$, into account:

$$\int_a^b f(x)dx = \frac{1}{N} \sum_{i=1}^{N \to \infty} \frac{f(X_i)}{p(X_i)} \tag{2.3}$$

The Rendering Equation (Equation 2.1) can now be rewritten using the Monte Carlo Estimator (Equation 2.3):

$$L_o(p, \omega_o) = L_e(p, \omega_o) + \frac{1}{N} \sum_{i=1}^{N \to \infty} \frac{f(p, \omega_o, \omega_i) L_i(p, \omega_i) |cos\theta_i|}{p(\omega_i)} \tag{2.4}$$

Note that although an infinite number of samples are needed for the Monte Carlo Estimator to approximate the integral precisely, this is not needed in practice when rendering (nor is it possible). Depending on the scene, only a few hundred samples per pixel (SPP) might be enough to capture how light interacts with the objects in the scene.

2.5.3 Bidirectional Reflectance/Transmission Distribution Function

The bidirectional reflectance/transmission distribution function is a way of describing how electromagnetic radiation is reflected, or transmitted, from, or through, a surface (the f-term in the Rendering Equation). In the case of rendering, the spectrum of electromagnetic radiation is usually restricted to that of visible light, $380nm$ to $740nm$.

Before continuing, it is necessary to establish an understanding of the abbreviations used in the following sections.

- BRDF: bidirectional reflectance distribution function - describes how a surface reflects incident light.
- BTDF: bidirectional transmission distribution function - describes how a surface transmits incident light through itself.
- BxDF: either a BRDF or a BTDF. This abbreviation is used when either a BRDF or BTDF could be relevant/used.
- BSDF: a composition of one or more BxDFs.

BRDFs

Each instance of a BRDF has its own function which describes what portions of electromagnetic radiation it reflects: a *spectral power distribution function* (SPD). An example is shown in Figure 2.5. If a reflective object appears blue, its BRDF instance absorbs most of the electromagnetic radiation with wavelengths other than those of blue, $430nm$ to $460nm$[22].

A path tracer that aims to capture light with the highest accuracy should work with light in terms of electromagnetic radiation, and not in terms of the more common *red-green-blue* (RGB) color spectrum. This means that instead of $L_o(p, \omega_o)$ being an array of floats with three entries, it should be an array with 360 entries. Although this results in higher accuracy, it also leads to a higher computational cost. A common way to mitigate this is to subdivide the spectrum into discrete bands. By specifying the range of each band, e.g. $6nm$, the array will have 60 entries instead of 360. When the color for a pixel has been calculated, the result must then be converted into a representation that can be viewed by computer monitors, or stored in a image format such as *PNG*. This

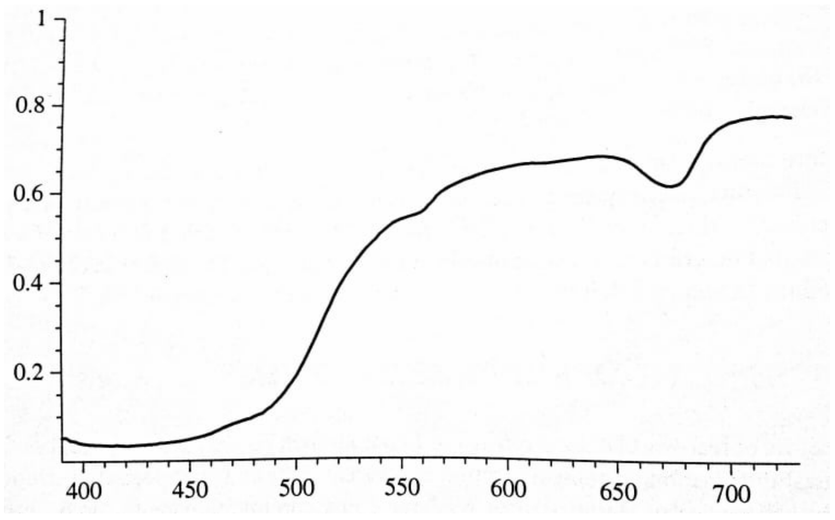


Figure 2.5: The reflection SPD of lemon peel (p.314 in [11]). X-axis: wavelength in nm, Y-axis: proportion of incident electromagnetic radiation that is reflected.

representation is usually RGB. However, for most common rendering use cases, using the RGB color spectrum during rendering is a sufficient approximation.

BRDFs are generally split into four types[11]:

- Diffuse: incident light along $-\omega_i$ is reflected equally in all directions over the hemisphere about the normal n at p. Although this BRDF is not physically realizable, it is often a good representation of materials that appear matte, with little to no highlights when illuminated, e.g. chalk or matte paint[11]. Since all the incident light is reflected equally in all directions, the probability of light being reflected in an arbitrary direction ω_o on the hemisphere is constant. This is shown in Equation 2.5.

$$p(\omega_o) = \frac{1}{2\pi} \tag{2.5}$$

- Specular: incident light along $-\omega_i$ is reflected in only *one* direction. Just as the diffuse BRDF, this BRDF is not physically realizable, but makes for a good approximation to mirror-like materials. Since all the incident light is reflected in *one* direction, the probability of light being reflected in an arbitrary direction ω_o on the hemisphere is 0. The probability of light being reflected in the reflection direction ω_r is 1. This is shown in Equation 2.6 and 2.7, respectively.

$$p(\omega_o) = 0 \tag{2.6}$$

$$p(\omega_r) = 1 \tag{2.7}$$

- Glossy: a common term for most other BRDFs that consists of two or more diffuse and specular BRDFs. A glossy BRDF can also be a unique BRDF, e.g. Disney Animation's BRDF[23]. The probability for light arriving along $-\omega_i$ to be reflected along ω_o is then either a combination of multiple probability density functions (PDFs), or it has its own PDF.

- Retro-Reflective: reflects incident light back predominantly along the incident direction. I.e. if the incident direction is $-\omega_i$, most of the light is reflected back along ω_i.

BTDFs

As with BRDFs, some representation of what wavelengths a BTDF transmits is needed, i.e. a second SPD. The transmittance SPD describes which wavelengths exit on the other side of the surface. In order to trace the outgoing ray, the angle at which it exits is needed.

To find the angle of the transmitted light, *Snell's Law* is used. It states a relationship between the angle θ_i of the incident ray, and the angle θ_t of the transmitted ray. This is shown in Equation 2.8, where η_i and η_t are the indices of refraction for the medium of the incident ray and outgoing ray, respectively.

$$\eta_i sin\theta_i = \eta_t sin\theta_t \tag{2.8}$$

The outgoing angle can then be found through some algebraic reordering. This is shown in Equation 2.9.

$$\theta_t = sin^{-1}(\frac{\eta_i sin\theta_i}{\eta_t}) \tag{2.9}$$

A visualization of Snell's Law can also be seen in Figure 2.6.

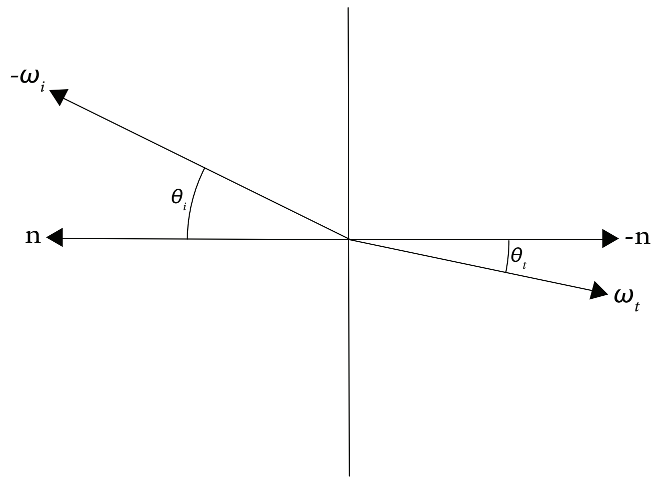


Figure 2.6: Visualization of Snell's Law

The index of refraction for a medium is a measure of how much slower light travels when in that medium compared to when in a vacuum. This is shown in Equation 2.10, where c is the speed of light in a vacuum, and v is the speed of light in the given medium.

$$\eta = \frac{c}{v} \tag{2.10}$$

Material	**Index of Refraction** (η)
Vacuum	1.0
Air at sea level	1.00029
Ice	1.31
Water (20^{o}C)	1.333
Glass	1.5-1.6
Diamond	2.417

Table 2.1: Various indices of refraction for common media[24].

Table 2.1 shows some indices of refraction for various common media.

An important thing to note regarding indices of refraction is that they vary depending on the energy level of the incident electromagnetic radiation. This means that incident light at different wavelengths will be transmitted at different angles. This is what causes the *dispersion* effect often seen in prisms[25]. If working on a wavelength level, this effect should be taken into account. If working with the RGB spectrum, this effect should either be disregarded or emulated.

2.5.4 Working With BSDFs

When an intersection with the scene is found, the piece of geometry will have a set of data associated with it. Within that set is the BSDF for the material at the given surface point. A BSDF can consist of a single BRDF and an RGB SPD. It can also be more complex, consisting of multiple BRDFs, BTDFs and information needed for Fresnel calculations (see Section 2.5.5). Given the BSDF, the task becomes how to calculate the $f(p, \omega_o, \omega_i)$ term in the Rendering Equation (see Equation 2.1). The process consists of three main steps:

1. **Choose a BxDF to sample**: for a given sample in the Monte Carlo version of the Rendering Equation (Equation 2.4), an ω_i is needed. This is done by sampling *one* of the BxDFs that is a part of the BSDF at the given surface point. For a BSDF consisting of a single BxDF, the BxDF which to sample is implied. For a BSDF consisting of multiple BxDFs, e.g. glass, a BxDF has to be chosen. This is done by selecting one at random, giving each BxDF a uniform probability. Note that if the BxDFs do not have a uniform probability of being chosen, this must be taken into account in Equation 2.4.

2. **Sampling the chosen BxDF**: once a BxDF has been chosen, the next step is to sample it. This involves calculating three pieces of information given its properties:

   - ω_i: the incident direction (see Figure 2.4 for a visualization). The procedure to sample the direction depends on the chosen BxDF, and so the appropriate measures must be taken to sample it correctly.
   - pdf_i: the probability of ω_i being chosen. It also depends on the chosen BxDF, and must be calculated according to the BxDF's properties.
   - $f(p, \omega_o, \omega_i)$: the reflected/transmitted color. Again, the chosen BxDF will have its own specific procedure to calculate this value.

3. **Account for BxDFs that were not chosen**: to sample ω_i, only <u>one</u> BxDF is chosen from the BxDFs making up the given BSDF. However, that does not mean that only the chosen BxDF's properties should be considered when calculating $f(p, \omega_o, \omega_i)$; all the BxDFs that could have been chosen must be taken into account. The reason for this is that for a sampled direction, ω_i, it is not only the chosen BxDF that will contribute to the amount of light that is reflected/transmitted along ω_o. Thus, the other BxDFs must be evaluated using the already sampled ω_i. I.e. a new ω_i is not sampled for the other BxDFs. The results, in form of a number of f terms, are accumulated.

   One thing to note here is that the pdf_i value also has to be updated. The other BxDFs, who's f terms are being accumulated, also have a probability of reflecting/transmitting incident light, arriving along $-\omega_i$, out along ω_o. This means that the PDF for each BxDF also has to be evaluated for the sampled ω_i. Now, this accumulation has to be averaged because it is dealing with a probability: it cannot become more than 1.

2.5.5 Fresnel Equations

When a ray of light hits a surface, the fraction of light that is reflected, and the fraction that is transmitted, is not constant. It depends on the incident angle θ_i, and the indices of refraction for the two participating media, η_i and η_t. To find the fraction of light that is reflected, and the fraction that is transmitted, the *Fresnel Equations*[26] are used.

Before discussing the equations, it is important to distinguish between two different classes of materials:

- Dielectrics: materials that do not conduct electricity, and that have real-valued indices of refraction. They do transmit a portion of incident light.
- Conductors: materials that do conduct electricity, and have a complex index of refraction. The fact that they conduct electricity results in the portion of incident light that is transmitted into the material being rapidly absorbed. Only thin conductor materials are able to let some light through itself. Thus, conductors are generally opaque.

When unpolarized light interacts with a dielectric material, the *Fresnel Reflectance* is the average of the square of the parallel and perpendicular polarization terms[25]. This is shown in Equation 2.11.

$$F_r = \frac{1}{2}(r_{\parallel}^2 + r_{\perp}^2) \tag{2.11}$$

$r_{\parallel}$ and $r_{\perp}$ are defined as shown in Equations 2.12 and 2.13, respectively.

$$r_{\parallel} = \frac{\eta_t cos\theta_i - \eta_i cos\theta_t}{\eta_t cos\theta_i + \eta_i cos\theta_t} \tag{2.12}$$

$$r_{\perp} = \frac{\eta_i cos\theta_i - \eta_t cos\theta_t}{\eta_i cos\theta_i + \eta_t cos\theta_t} \tag{2.13}$$

Due to the conservation of energy, the sum of reflected and transmitted radiance must equal the incident radiance. Thus, the fraction of light that is not reflected is transmitted into the material, and is the residual of the reflected light. This is shown in Equation 2.14.

$$F_t = 1 - F_r \tag{2.14}$$

The formulas for conductive materials are not included here, as they are not as relevant for this thesis.

The Fresnel Equations can now be combined with the BSDF at a surface point. To determine the amount of reflected or transmitted light for a specific BxDF, the following steps are needed:

1. Determine η_i and η_t for the two participating media.
2. Calculate $\theta_i = dot(n, \omega_i)$.
3. Use Snell's Law to find θ_t.
4. Find ω_t.
5. Using the dielectric or conductor variant of the Fresnel Equations, to find the fraction of radiance that is reflected and transmitted.
6. Given the particular BxDF, the spectrum of the BxDF is multiplied with that of the correct Fresnel Equation (F_r or F_t, depending on if the BxDF is a BRDF or BTDF).

2.5.6 Sampling Directions

Sampling is one of the core parts of a Monte Carlo Estimator. Remember that for a given sample, X_i, in the Monte Carlo Estimator (Equation 2.3), $X_i \in [a, b]$. So when sampling a direction ω_i, it must uphold this restriction and be in the domain of the integral: $\omega_i \in \mathcal{H}$. The problem thus becomes: how to sample a direction on a hemisphere uniformly?

Spherical Coordinates

So far when discussing directions, a *Cartesian Coordinate* representation $\omega = (x, y, z)$ has been assumed. However, there does exist another representation that becomes useful when sampling a hemisphere uniformly: *Spherical Coordinates*. A spherical coordinate is defined by three components: $\omega = (r, \theta, \phi)$ which are described below:

- r: the *Euclidean distance* from the origin to the point. Since directions are normalized, this will automatically be 1.
- θ: the *inclination angle* $\in [0, \pi]$. Note that since only directions that are in the same hemisphere as the normal n at p are of interest, the domain of θ is further restricted to $\theta \in [0, \frac{\pi}{2}]$.
- ϕ: the *azimuth angle* $\in [0, 2\pi)$.

Furthermore, there exists a relationship between Cartesian and spherical coordinates that allows for converting back and forth. These relationships are shown in Equations 2.15 through 2.17.

$$x = r\sin\theta\cos\phi \Rightarrow x = \sin\theta\cos\phi \tag{2.15}$$

$$y = r\sin\theta\sin\phi \Rightarrow y = \sin\theta\sin\phi \tag{2.16}$$

$$z = r\cos\theta \Rightarrow z = \cos\theta \tag{2.17}$$

Finally, the relationship between differential area of a set of directions and the differential area of a (θ, ϕ) is shown in Equation 2.18.

$$d\omega = \sin\theta d\theta d\phi \tag{2.18}$$

Using the aforementioned equations, the integral in the Rendering Equation (Equation 2.1) can be rewritten as a double integral over differential spherical coordinates, instead of a single integral over differential area of a set of directions:

$$\int_0^{2\pi}\int_0^{\frac{\pi}{2}} f(p,\theta_o,\phi_o,\theta_i,\phi_i)L_i(p,\theta_i,\phi_i)|\cos\theta_i|\sin\theta_i d\theta_i d\phi_i \tag{2.19}$$

Since the integral can be written in this form, a direction on the hemisphere can be sampled using spherical coordinates, and subsequently be converted to Cartesian representation.

Sampling a Direction Uniformly

Variables which are needed to perform the sampling operation are:

- The normal n at p.
- Two uniform random values: u_0 and u_1.

To sample a direction uniformly implies that each direction in the domain should have the same probability of being chosen. It therefore follows directly that the probability density function is constant: $p(\omega) = c$. It also follows that this function must integrate to 1 given its domain. Since the area of a unit hemisphere is 2π, $p(\omega) = \frac{1}{2\pi}$. This is shown in Equations 2.20 through 2.21.

$$\int_{\mathcal{H}} p(\omega)d\omega = c\int_{\mathcal{H}} d\omega = 1 \tag{2.20}$$

$$\int_{\mathcal{H}} d\omega = 2\pi \Rightarrow p(\omega) = c = \frac{1}{2\pi} \tag{2.21}$$

In terms of spherical coordinates, this becomes $p(\theta,\phi) = \frac{\sin\theta}{2\pi}$. This representation contains two variables that need to be sampled. The technique *multi-dimensional sampling*, using the marginal and conditional densities is used to sample the variables. It involves the following steps:

1. Find the marginal density function for θ:

$$p(\theta) = \int_0^{2\pi} p(\theta, \phi) d\phi = \int_0^{2\pi} \frac{\sin\theta}{2\pi} d\phi = \sin\theta \tag{2.22}$$

2. Compute the conditional density function for ϕ:

$$p(\phi|\theta) = \frac{p(\theta, \phi)}{p(\theta)} = \frac{\frac{\sin\theta}{2\pi}}{2\pi} = \frac{1}{2\pi} \tag{2.23}$$

   In probability theory, conditional probability is given by Andrey Kolmogorov's definition[27]:

$$p(A|B) = \frac{p(A \cap B)}{p(B)} \tag{2.24}$$

3. $p(\theta)$ give the probability of any θ being chosen, while $p(\phi|\theta)$ give the probability of any ϕ being chosen given a θ. But to actually sample the variables, the respective distribution functions are needed. These are obtained by integrating over all possible values θ and ϕ can take on. This is shown in Equation 2.25 and 2.26.

$$P(\theta) = \int_0^{\theta} \sin\theta d\theta = 1 - \cos\theta \tag{2.25}$$

$$P(\phi) = \int_0^{\phi} \frac{1}{2\pi} d\phi = \frac{\phi}{2\pi} \tag{2.26}$$

4. Next, these functions must be inverted. Only the case of inverting $f(x) = 1 - \cos x$ is shown:

$$\begin{aligned} P(\theta) &= 1 - \cos(\theta) \\ y &= 1 - \cos(\theta) \\ \theta &= 1 - \cos(y) \\ \cos(y) &= 1 - \theta \\ y &= \cos^{-1}(1 - \theta) \\ P(\theta)^{-1} &= \cos^{-1}(1 - \theta) \end{aligned}$$

   Then, given some random uniform value u_0, $\theta = cos^{-1}(1-u_0) = cos^{-1}(u_0)$. Inverting $P(\phi|\theta)$ gives $\phi = 2\pi u_1$, for some random uniform value u_1.

5. The next step is to convert this back to Cartesian coordinates:

$$\begin{aligned} x &= \sin\theta\cos\phi = \sin(\cos^{-1}(u_0))\cos(2\pi u_1) \\ y &= \sin\theta\sin\phi = \sin(\cos^{-1}(u_0))\sin(2\pi u_1) \\ z &= \cos\theta = \cos(\cos^{-1}(u_0)) = u_0 \end{aligned}$$

6. Finally, the sampled direction must be rotated so that it is sampled around the normal n. From 2.15 through 2.17 it is known that $x \in [-1, 1]$,

$y \in [-1, 1]$ and $z \in [0, 1]$. Thus, ω was sampled around the hemisphere about the positive z-axis $= (0, 0, 1)$. To rotate ω, a 3x3 rotation matrix m that will align the z-axis with n must be calculated. See the function *RotationToAlignAToB* in *GeometricUtilities.cpp* on how to do this. ω is simply multiplied with this matrix, and the direction in Cartesian coordinates is obtained:

$$\omega_n = m * \omega_{z-axis} \tag{2.27}$$

It is also possible to sample directions non-uniformly, as the Monte Carlo Estimator takes the probability of a direction ω_i into account. Non-uniform sampling, if done properly, can improve the results, and is thus widely used[28].

2.5.7 Sampling Geometrical Shapes

There is also the problem of sampling shapes uniformly. There are two types of geometrical shapes that need uniform sampling support: triangles and spheres. Although the method is similar to directional sampling, it is worth going through both here.

Triangles

When working with triangles, it is useful to be aware of *Barycentric Coordinates*. Given a triangle's three vertices v_0, v_1 and v_2, Barycentric coordinates allows for any point within the triangle to be represented as a linear combination of those three points:

$$p = v_0\lambda_0 + v_1\lambda_1 + v_2\lambda_2 \tag{2.28}$$

Barycentric coordinates impose two restrictions that limit their "reach" from points in the triangle's plane, to points within the triangle itself:

1. $\lambda_n \in [0, 1]$
2. $$1 = \lambda_0 + \lambda_1 + \lambda_2 \tag{2.29}$$

   Algebraic reordering then gives:

   $$\lambda_2 = 1 - \lambda_0 - \lambda_1 \tag{2.30}$$

Next, assume that the triangle to be sampled is an isosceles right triangle with an area of $\frac{1}{2}$. As with other types of uniform sampling, the PDF must be constant: $p(q) = c$. Also, it directly follows that the PDF must be 1 over the area of the triangle as it must integrate to 1 over its domain, i.e. its area. Thus $p(q) = \frac{1}{\frac{1}{2}} = 2$. Finally, since λ_2 is given if λ_0 and λ_1 are known (Equation 2.30), only two random uniform values, u_0 and u_1, are needed. The procedure then becomes similar to that of sampling directions:

1. When looking at the triangle to be sampled from a perpendicular angle to the plane it creates, any point on the hypotenuse is described by:

   $$p_h = v_0\lambda_0 + v_1\lambda_1 = \lambda_0 + (1 - \lambda_1) \tag{2.31}$$

   This can be read as the domain of λ_1 being $\in [0, 1 - \lambda_0]$. The domain of λ_0 is simply $\in [0, 1]$

2. We can then find the marginal density for λ_0:

$$p(\lambda_0) = \int_0^{1-\lambda_0} p(\lambda_0, \lambda_1) d\lambda_1 = \int_0^{1-\lambda_0} 2 d\lambda_1 = 2(1-\lambda_0) \tag{2.32}$$

3. The conditional density $p(\lambda_1|\lambda_0)$ can be found as shown in Equation 2.33.

$$p(\lambda_1|\lambda_0) = \frac{p(\lambda_0, \lambda_1)}{p(\lambda_0)} = \frac{2}{2(1-\lambda_0)} = \frac{1}{1-\lambda_0} \tag{2.33}$$

4. Both densities are integrated over their domains to find their conditional density functions:

$$P(\lambda_0) = \int_0^{\lambda_0} 2(1-\lambda_0) d\lambda_0 = 2\lambda_0 - \lambda_0^2 \tag{2.34}$$

$$P(\lambda_1) = \int_0^{\lambda_1} \frac{1}{1-\lambda_0} d\lambda_1 = \frac{\lambda_1}{1-\lambda_0} \tag{2.35}$$

5. Second to last, the functions are inverted (not shown) and the random uniform variables are substituted in. This gives the following:

$$\lambda_0 = 1 - \sqrt{u_0} \tag{2.36}$$

$$\lambda_1 = u_1\sqrt{u_0} \tag{2.37}$$

$$\lambda_2 = 1 - \lambda_0 - \lambda_1 = 1 - (1 - \sqrt{u_0}) - (u_1\sqrt{u_0}) \tag{2.38}$$

6. Finally, to find the point p within the triangle, the formula for Barycentric coordinates (Equation 2.28) is followed.

Spheres

Spheres are just two hemispheres put together, so very little modification is needed to sample a sphere when the procedure for sampling a direction on a hemisphere is known. This is because sampling a point uniformly on a sphere is equivalent to sampling a direction uniformly. This holds because by following the sampled direction r units, where r is the radius of the sphere, a point on the sphere will be reached.

1. First, it is known that the area of a unit sphere is 4π (not 2π as with hemispheres). This means that $p(\omega) = \frac{1}{4\pi} \rightarrow p(\theta, \phi) = \frac{\sin\theta}{4\pi}$

2. The marginal and conditional densities are found as before:

$$p(\theta) = \int_0^{2\pi} \frac{\sin\theta}{4\pi} d\phi = \frac{\sin\theta}{2} \tag{2.39}$$

$$p(\phi|\theta) = \frac{p(\theta, \phi)}{p(\theta)} = \frac{1}{2\pi} \tag{2.40}$$

3. Next, find the conditional density functions by integrating the domains and invert them:

$$P(\theta) = \int_0^{\theta} \frac{\sin\theta}{2} d\theta = \frac{1-\cos\theta}{2} \rightarrow \theta = \cos^{-1}(1-2u_0) \tag{2.41}$$

$$P(\phi) = \int_0^{\phi} \frac{1}{2\pi} d\phi = \frac{\phi}{2\pi} \rightarrow \phi = 2\pi u_1 \tag{2.42}$$

4. The Cartesian coordinates can finally be found the same way as for the hemisphere sampling.

2.5.8 Area Lights

The term $L_i(p, \omega_i)$ in the Rendering Equation (Equation 2.1) takes incident light into account for the current intersection point p. It consists of two "sub-terms":

- Indirect incident light.
- Direct incident light.

The 1^{st} sub-term is dealt with through multiple bounces for a ray. To calculate the 2^{nd} sub-term, it might be necessary to sample the lights in the scene. This depends on whether the BxDF is sampled to get ω_i, or a point, p_a, on a light is sampled, giving $\omega_i = normalize(p_a - p)$.

Regardless of the method used for acquiring ω_i, an area light must be selected to work against. This selection can be done at random, or by taking the lights' properties into account, e.g. area. In the case where the BxDF is sampled, a ray is fired out along ω_i, and checked for an intersection against the selected light. If an intersection is found, the amount of direct irradiance is calculated.

The case where p_a is sampled on a light is a bit different. Given the chosen area light, the amount of direct radiance that is emitted from p_a and reaches the point of intersection, p, must be calculated. This involves a few sub-steps:

1. Sample a point p_a on the area light (see Section 2.5.7).
2. Check for an unobstructed path from p_a to p, i.e. direct light will reach p from p_a.
3. Calculate the amount of light reaching p, which involves checking that the direction from p_a to p, i.e. ω_i, and the normal n at p, are in the same hemisphere. If they are, the radiance emitted from p_a along $-\omega_i$ is calculated.

How to Best Sample the Incident Direction?

In the case where the BxDF at p is a specular BRDF, sampling p_a from a light's distribution is not the best choice. This is because radiance will only be reflected along ω_o if, and only if, the incident radiance arrives along a $-\omega_i$, where ω_i is the reflection of ω_o about n. This will never be the case when ω_i is sampled from the light's distribution. However, if the BRDF at p is sampled, the correct reflection direction ω_i will always be returned, and thus, the Monte Carlo Estimator will converge more quickly. However, in the case of a very rough BxDF, e.g. diffuse or metals, sampling the light source is generally the better alternative. This is because sampling the BRDF would give ω_i's that are uniformly distributed and that might not always intersect the selected light. It is therefore important to choose the sampling method carefully. It is also important to use the correct value for $p(\omega_i)$ depending on method:

- If sampling the BxDF, the probability of sampling ω_i must be used.
- If sampling the light, the probability of sampling the point p_a on the light must be used.

It is also possible to combine the two sampling strategies using multiple importance sampling (MIS)[29].

Finally, regardless of the chosen sampling method, it is very important that the next direction of the ray is sampled using the BxDF at p and not the light's distribution. This can be explained as follows: as light can arrive at p from any direction, only selecting the next direction where there are lights will result in an incorrect result. This is because the selection of the next direction is biased given the distribution of lights in the scene. The only circumstance where this would produce correct results, is if the entire scene is encapsulated by a spherical light source.

Chapter 3

Path Tracing Engine

One of the goals for this thesis was to create a physically based path tracer, who's results could be used a comparison for the real-time AO results. As developing a production-level-renderer like *Arnold*[30] requires years, the scope of the path tracer was set to be a subset of *pbrt-v3*[31]. This included the following features:

- Loading scenes from a specified format.
- A parallel path tracing implementation.
- A pinhole camera model.
- Accelerated intersection testing using a BVH.
- Various BxDFs:
  - Diffuse BRDF
  - Specular BRDF and BTDF
  - Microfacet BRDF using Beckmann's microfacet distribution function.
- Refraction using Snell's Law and the Fresnel Equations.
- Various materials:
  - Matte
  - Mirror
  - Copper and gold
  - Glass

The engine is written in C++, specifically C++11. It also utilizes a C++17 feature for dealing with file systems in an OS agnostic way. C++ was chosen for two main reasons:

1. Its (relatively) low-overhead run-time performance.
2. Its support for inheritance and virtual functions.

The code in this project is written with C in mind, with selected C++ features, e.g. *std::vector*, *std::string*, *std::iterator*, constructors/destructors, inheritance and virtual functions, only appearing where useful.

What follows is a description of the implementation details for some of the features in the engine.

3.1 Scene Creation

Before any rendering can start, the scene has to be created. To make the rendering of various scenes with varying properties easier, a scene file format was created in which the various properties of the scene can be specified. An example is shown below:

```
Camera position[0 5 15] view_direction[0 -0.3 -1.0]
       vertical_fov[45] width[1920] height[1080]
Model file[data/mercedes/Mercedes_AMG_GT-R_OBJ.obj]
      translate[0 0 0] rotate[0 0 0] scale[0.02 0.02 0.02]
      material[matte] diffuse[0.9 0.4 0.25]
SphericalLight center[-5 5 0] radius[0.2] emittance[10 10 10]
SphericalLight center[5 5 0] radius[0.2] emittance[0 10 10]
```

For further details, see Section 7.3 in the Appendix.

The engine supports one file format for loading 3D models: *Wavefront*[32]. To save time and avoid bugs, Syoyo Fujita's single-header loader, *tinyobjloader*[33], is used to load Wavefront files and accompanying material files. A Wavefront file may contain the following data:

- Vertices: the points that make up a 3D model.
- Indices: the order in which the vertices are connected.
- Normals: the plane that a vertex lies in.
- Texture coordinates: where in the accompanying images data is to be sampled from for a given vertex.
- Material data: describe the material properties of a face (three or more connected vertices).

The amount of data that needs to be present for a Wavefront file to be conformant to the format specification is small; all that is needed are three vertices. If normals are missing, they can be calculated on a per-face basis, given that the winding order for faces is consistent throughout the file. Note that per-face normals results in less physically accurate lighting than per-vertex normals. Barycentric coordinates are the reason for this. To find the normal of a point within a face, the normals at its vertices are interpolated. Thus, per-face normals will give the same normal for any point in the triangle, as all vertices will have the same normal. On the other hand, per-vertex normals results in varying normals across the face. This also results in a smoother transition across faces. If texture coordinates are missing, they cannot be calculated. There is also no requirement for material properties to be specified, so a default material must exist in the engine to be used in such cases.

Once an image is rendered, it becomes necessary to save it. For this the *stb* library is used, specifically *stb_image_write*[34].

3.2 Main Entry Point

The main file in the engine is *main.cpp*. It contains the engine's entry point:

```
int main(int argc, char** argv) {/*Things happen*/}
```

In this function, various things happen. The scene is loaded, the camera is initialized, memory for all images to be generated is allocated, a set of random number generators are created etc. Then, depending on the settings described in the *.brhan* file, the appropriate rendering function(s), of which there are three, is selected:

- *Render()*: renders the scene and stores *one* RGB image.
- *RenderInIntervals()*: renders the scene and stores n RGB images at certain intervals, e.g. after 256, 512, 768, 1024 SPP.
- *GenerateDepthImage()*: fires a single ray per pixel and stores a grayscale image with depth values.

3.3 The Rendering Loop

The standard rendering loop, *Render()*, has a the layout which is shown below:

```
for (unsigned int y = 0; y < system.film_height; y++) {
  for (unsigned int x = 0; x < system.film_width; x++) {
        const float u  = (float(x) + 0.5f) /
                            float(system.film_width);
        const float v  = (float(y) + 0.5f) /
                            float(system.film_height);
        glm::vec3 L(0.0f);
        for (unsigned int s = 0; s < system.spp; s++) {
          glm::vec2 sample_offset =
                       pixel_sampler.Sample(s,
                       rngs[omp_get_thread_num()]);
          Ray ray = camera.GenerateRay(u,v,sample_offset);
          L += system.integrator.Li(scene, &ray, rngs,
                       omp_get_thread_num(),
                       0, system.max_depth);
        }
        L /= float(system->spp);
        int idx = (y * system.film_width + x) * 3;
        film[idx+0] = L.r;
        film[idx+1] = L.g;
        film[idx+2] = L.b;
  }
}
WriteImage(film, *system);
```

The other rendering loops for *RenderInIntervals()* and *GenerateDepthImage()* look similar, but vary slightly to accomplish their specific goal.

3.4 Calculating Ray Directions

The technique described in Section 2.2 is implemented directly as is. While it does generate seemingly correct results, it is a simplification when compared to how actual cameras work. However, with the purpose of this thesis in mind, it was decided that implementing realistic camera models was not the best use of the available time.

3.5 Intersection Testing

The engine supports two kinds of shapes: triangles and spheres. Both of these shapes are derived from a parent struct called *Shape*. This design allows for a convenient interaction with objects in the scene. I.e. when intersecting the scene, both *Triangle* and *Sphere* structs have their own implementation of the *virtual* function *Intersect()* defined in *Shape*. This is shown below:

```
struct Shape {
    /*Member data and functions*/
    virtual bool Intersect(Ray* ray,
    SurfaceInteraction* isect, float t_min, float t_max)
    const = 0;
};
struct Triangle : Shape {
    /*Member data and functions*/
    bool Intersect(Ray* ray, SurfaceInteraction* isect,
    float t_min, float t_max) const;
};
struct Sphere : Shape {
    /*Member data and functions*/
    bool Intersect(Ray* ray, SurfaceInteraction* isect,
    float t_min, float t_max) const;
};
```

3.5.1 Brute-Force Method

To find the closest point of intersection given a ray and the scene, the ray needs to be checked against every piece of geometry in the scene for an intersection. For small scenes with a few hundred triangles, a brute-force implementation can work well. For clarity, an example of such an implementation can be seen below:

```
Ray ray = camera.GenerateRay(xn, yn);
float closestHitDist = std::numeric_limits<float>::max();
for (const Mesh& mesh : scene.meshes) {
    for (const Shape& shape : mesh.shapes) {
        if (shape.Intersect(ray) &&
            ray.t < closestHitDist) {
```

```
            closestHitDist = ray.t;
        }
    }
}
if (closestHit == std::numeric_limits<float>::max()) {
    return false;
}
isect->point = ray.origin + (ray.dir * closestHitDist);
return true;
```

However, as the number of shapes grows, the time it takes to compute the closest intersection point grows linearly with it. This is not a desirable scenario.

3.5.2 Acceleration Structures

To speed up the intersection testing, a BVH structure (see Section 2.4.1 for information about BVH), was implemented. The speed up gained from the BVH structure is clear for any scene with more than a few hundred triangles. A comparison of rendering times of the scene described in *scenes/buddha.brhan* with 1 SPP can be seen in Table 3.1. Although not all scenes can be expected

Description	Rendering time (s)
Brute-force	113.6
BVH	0.4

Table 3.1: Comparison of brute-force vs. BVH render times for *buddha.brhan* with 1 SPP.

to have such a drastic reduction in rendering times, this clearly shows how using an acceleration structure can benefit rendering times. In this particular case, the reason for the drastic decrease is that most of the rays never hit any geometry (see Figure 3.1). This means that when a ray fails to intersect the top-most bounding-box in the BVH, all triangles below that bounding-box can be disregarded and the process is then finished. For this scene with its 28 980 triangles, this “early-exit” saves a lot of time.

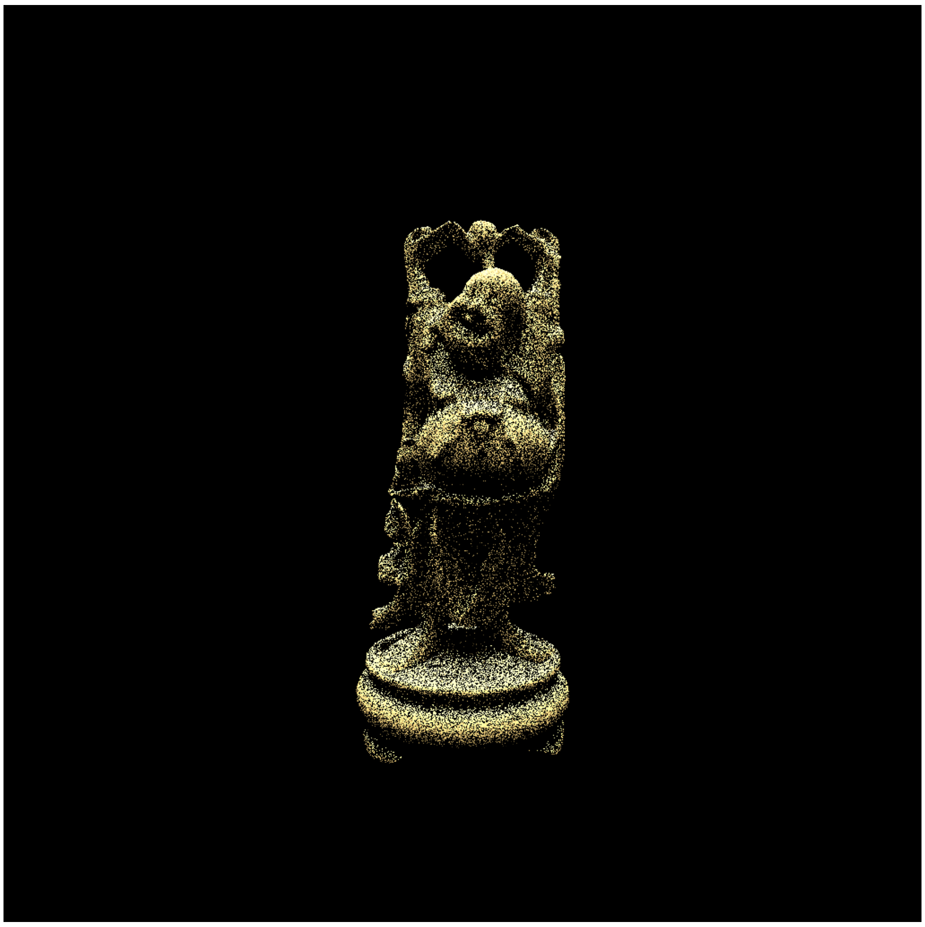


Figure 3.1: *budda.brhan* rendered with 1 SPP (model courtesy of Stanford Computer Graphics Laboratory[35] via PBRT scenes[36]).

3.6 Materials and BSDFs

Once the closest intersection point has been found, the properties at the surface point must be evaluated. This gives the values needed to estimate the Rendering Equation using the Monte Carlo Estimator (see Equation 2.4).

As with different types of shapes, all materials inherit from a common *Material* struct. This allows materials to have their own unique data, but still have a common interface through which they are interacted with. Examples include *MatteMaterial*, *MirrorMaterial* and *MetalMaterial*. They all share one important function which they must implement: *ComputeScatteringFunctions()*. In this function, the BSDF for the given material at the intersected surface point is created. It can then be sampled later on. An example for the *MatteMaterial* can be seen below:

```
void MatteMaterial::ComputeScatteringFunctions(
SurfaceInteraction* isect) const {
  isect->bsdf = new BSDF();
  DiffuseBRDF* l_ptr = new DiffuseBRDF(reflection_spectrum);
  isect->bsdf->Add(l_ptr);
}
```

3.7 Choosing a BxDF to Sample

Once the BSDF for the current intersection point has been created, it can be queried for the necessary information.. The first step in this process is to choose *one* BxDF from the available BxDFs in the BSDF. Given this BxDF, the direction ω_i can be sampled. A code snippet showing a simplified version of this choosing is shown in below:

```
int bxdf_to_sample = std::min(int(uniform_value * num_BxDFs),
                              num_BxDFs - 1);
BxDF* bxdf = bxdfs[bxdf_to_sample];
```

3.8 Sampling BxDFs

When a BxDF is sampled, i.e. its *f()* function is called, the amount of light arriving along $-\omega_i$, hitting point p and being reflected/transmitted out along ω_o, is returned. What this function returns varies between BxDFs. An example for a diffuse BRDF can be seen below:

```
return R * ONE_OVER_PI;
```

For a specular BxDF, the Fresnel Equations have to be evaluated in this function, while for a microfacet BRDF, a new normal must be generated from its microfacet distribution.

3.9 Account for BxDFs That Were Not Chosen

Once a BxDF has been chosen, and ω_i has been sampled, all BxDFs that could have been chosen must be taken into account when calculating pdf_i and $f(p, \omega_o, \omega_i)$. A code snippet showing a simplified version of this process can be seen below:

```
for (int i = 0; i < num_BxDFs; i++) {
    if (bxdfs[i]->MatchesFlags(bxdf->flags)) {
        pdf += bxdfs[i]->Pdf(wo, wi, normal);
        f += bxdfs[i]->f(wo, normal, wi);
    }
}
pdf /= num_BxDFs;
```

3.10 Fresnel Equations

One important implementation detail regarding the Fresnel Equations for dielectric, is that the indices of refraction for a medium are specified in an outside-to-inside order. This means that when a ray interacts with a surface that has a BxDF with a Fresnel property, the indices of refraction must be swapped depending on whether the ray is entering or leaving the medium.

Here is an example: given that $\eta_t = 1$ and $\eta_i = 1.2$, if the ray is entering the medium, the indices are left as they are. However, if the ray is leaving the medium, the indices are swapped, i.e. $\eta_t = 1.2$ and $\eta_i = 1$. Whether or not this swap is necessary can be detected when evaluating the Fresnel Equations, and is shown below:

```
float cos_theta_from = glm::dot(wo, normal);
bool exiting = cos_theta_from <= 0.0f;
if (exiting) {
  std::swap(eta_t, eta_i);
  cos_theta_from = glm::abs(cos_theta_from);
}
```

The dot-product between the outgoing direction ω_o and the normal n at the surface is taken. If this result is negative, it means that the indices must be swapped. This works because the normal for all shapes in the scene have their normal pointing out of the shape. If any normal is facing the wrong way, i.e. into the shape, this technique is invalid.

3.11 Parallelization

An feature of the path tracing algorithm is that it is highly parallelizable. Each ray that is traced from a pixel is independent from any other ray that is traced, both from a different pixel, but also the same pixel. This means that multiple paths can be traced simultaneously.

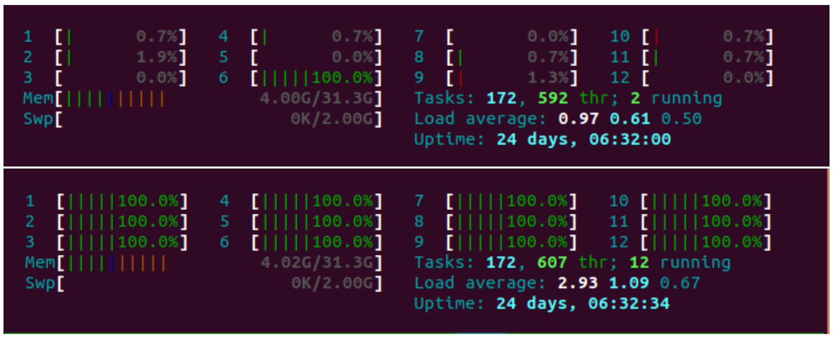
For complex path tracing engines, e.g. production renderers, the parallelization has to be done very carefully. However, for an implementation such as this one, a high level of parallelization is achieved with a single line of code:

```
#pragma omp parallel for // Line of importance
for (unsigned int y = 0; y < system.film_height; y++) {
    for (unsigned int x = 0; x < system.film_width; x++) {
        /*Rendering happens*/
    }
}
```

By utilizing OpenMP's interface, and compiling with *-fopenmp* (in *g++*), the engine is able to utilize all of the CPU's cores close to their max potential, compared to using just one core. This is showcased in Figure 3.2.

To get further insight into how much performance is gained when using 12 cores vs 1 core, the rendering times for both setups, and the speedup gained when using 12 cores compared to 1 core, can be plotted. These plots can be seen in Figure 3.3 and 3.4 respectively.

The first thing to notice is that when the number of samples increases, the rendering time also increases, and it does so in a linear fashion. This is a good thing; the program should not suffer much due to more samples within a pixel. The same holds true when using more cores. Next, Figure 3.4 shows that the program's scalability stays relatively stable when increasing the number of samples. However, notice that the scaling achieved is not close to a factor of 12.



```
1  [|         0.7%]   4  [|         0.7%]   7  [          0.0%]   10 [|         0.7%]
2  [|         1.9%]   5  [          0.0%]   8  [|         0.7%]   11 [|         0.7%]
3  [          0.0%]   6  [|||||100.0%]   9  [|         1.3%]   12 [          0.0%]
Mem[||||||||||          4.00G/31.3G]   Tasks: 172, 592 thr; 2 running
Swp[                       0K/2.00G]   Load average: 0.97 0.61 0.50
                                       Uptime: 24 days, 06:32:00
```

```
1  [|||||100.0%]   4  [|||||100.0%]   7  [|||||100.0%]   10 [|||||100.0%]
2  [|||||100.0%]   5  [|||||100.0%]   8  [|||||100.0%]   11 [|||||100.0%]
3  [|||||100.0%]   6  [|||||100.0%]   9  [|||||100.0%]   12 [|||||100.0%]
Mem[||||||||||          4.02G/31.3G]   Tasks: 172, 607 thr; 12 running
Swp[                       0K/2.00G]   Load average: 2.93 1.09 0.67
                                       Uptime: 24 days, 06:32:34
```

Figure 3.2: Comparison of running without (top) and with (bottom) OpenMP parallelization on an Intel i7-8700K. Note that in the top image, only core six is used for rendering, while in the bottom, all twelve cores are used.

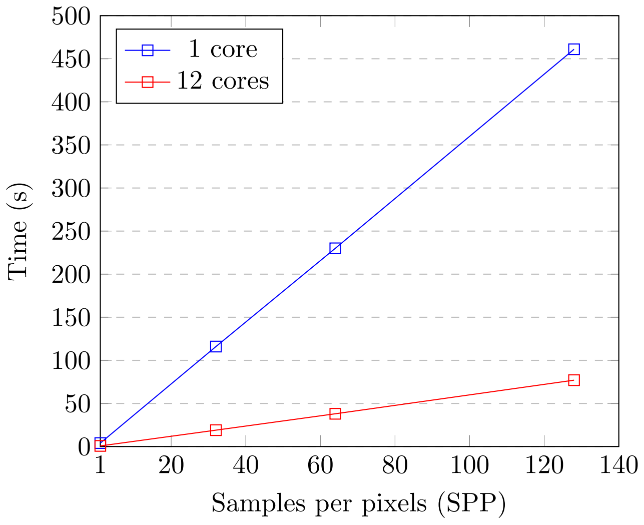


Figure 3.3: Comparison of rendering times

Linear scaling would be an incredible result, and it is very rarely (or never) the case when working with parallel processing.

By testing with different core configurations (all with 128 SPP), it becomes clear that the program scales differently depending on the number of cores. This is shown in Figure 3.5. The program is not close to a linear speedup with any of the configuration. In fact, the $\frac{ActualSpeedup}{PerfectSpeedup}$ ratio decreases as the number of cores increases. There are various reasons that might help explain why this is the case:

- There is some overhead associated with assigning work to threads. This happens each time a thread reaches the end of its double for-loop.
- The Intel i7-8700K (which this entire project is being run on) has 12MB of Intel's SmartCache[37]. Although this design gives each core its own L1 cache, it will share higher levels of cache with one or more of the other

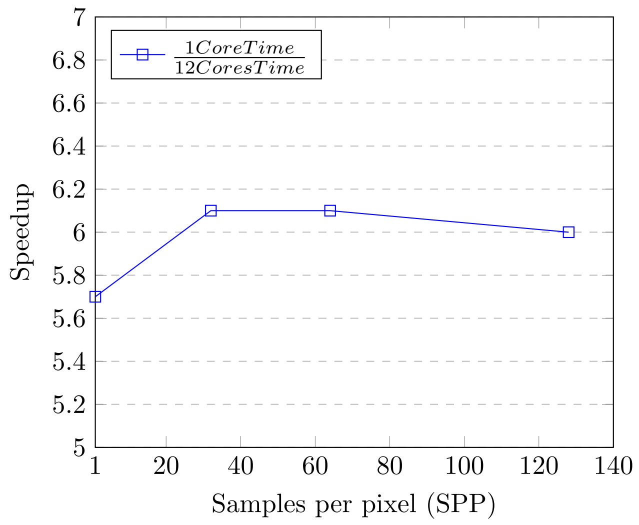


Figure 3.4: Speedup when using 12 cores vs 1 core

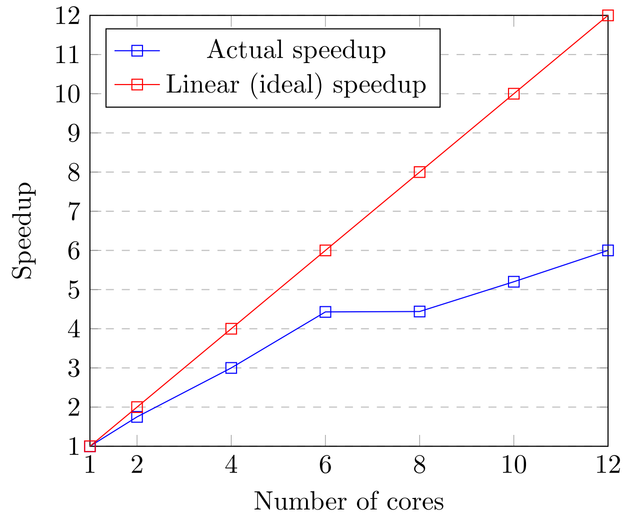


Figure 3.5: Comparison of speedup with different core configurations (all taking 128 SPP)

cores. As the program is running, each path that is being traced will access different data, and will thus weaken the principles of cache locality and temporality in the higher levels of cache.

- The CPU only has six physical cores, but has support for Intel Hyper-Threading[38] which allows for up to twelve logical cores. However, this means that two logical cores will share L1 cache, again weakening the principles of cache locality and temporality on a lower cache level. Additionally, Hyper-Threading does not actually add more execution engines to a core. While the two logical cores will have their own state, they share the *same* execution engine. This means that if both logical cores are running with perfect instruction-level parallelism, very little performance is gained by Hyper-Threading. However, this is usually not the case, and so a speed-up is achieved. However, this does shine light on why the engine is nowhere near a x12 speedup; it is virtually impossible with Hyper-Threading as a technology.

- The CPU also has other tasks, which are scheduled by the operating system (OS), that it must complete. This leads to context switches and accompanying state changes, which are non-desirable when aiming for optimal scalability.

Note that the reasons mentioned above are not specific to this program; they would apply to any parallel program running on this CPU.

3.12 Rendering Results

Below are some images rendered with the path tracer developed for this thesis: *BrhanRenderer*. Note that the first image is a reference image rendered by *pbrt-v3*; the renderer developed for the *Physically Based Rendering* book[11]. These show that the implementation was indeed successful, and the results comparable to that of the reference.

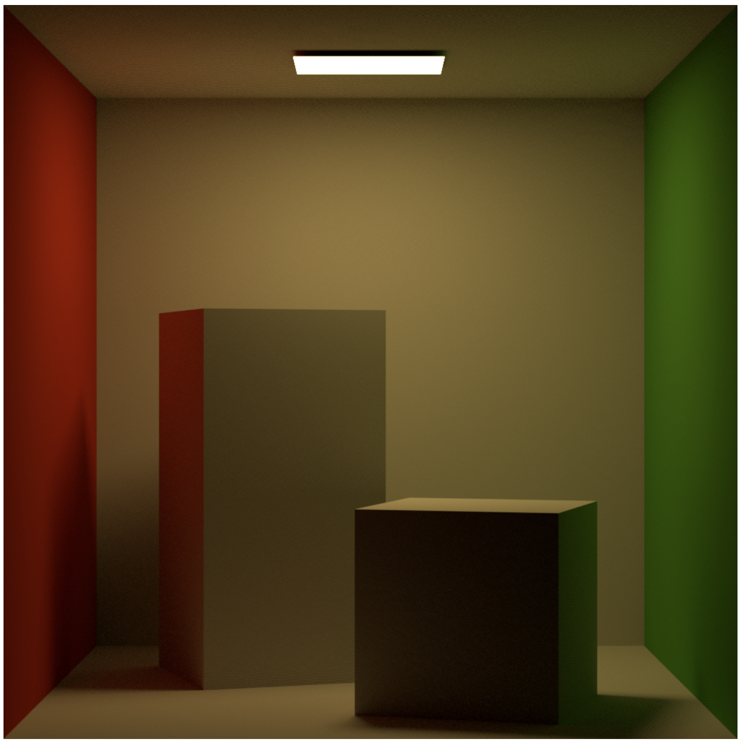


Figure 3.6: Reference *Cornell Box* render by *pbrt* (model courtesy of Morgan McGuire's graphics archive[39]).

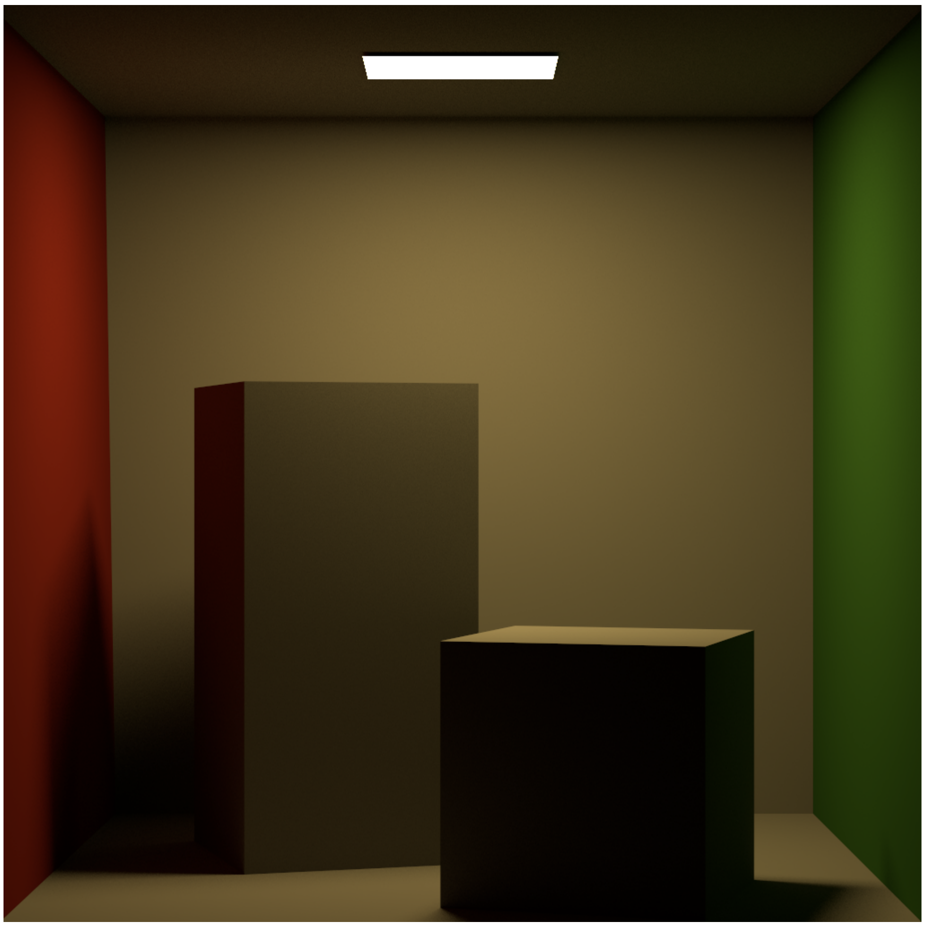


Figure 3.7: *Cornell Box* render by *BrhanRenderer* (model courtesy of Morgan McGuire's graphics archive[39]).

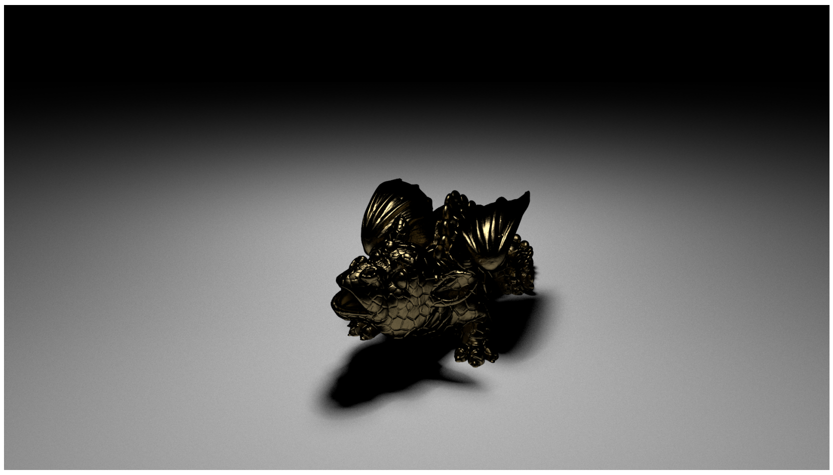


Figure 3.8: *Dragon* render by *BrhanRenderer* (model courtesy of Christian Schüller via PBRT scenes[36]).

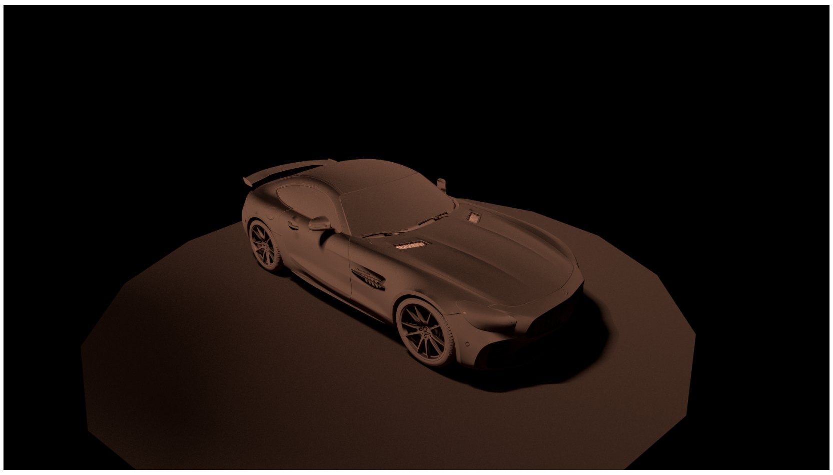


Figure 3.9: *Mercedes* render by *BrhanRenderer* (model courtesy of Thomas F[40]).

Chapter 4

Ray Tracing on GPUs

Although CPUs have multiple cores, which allow them to do parallel work, this is not where they excel. Contrary to CPUs, GPUs are designed specifically for parallel work. Seeing as path tracing, and the underlying ray tracing algorithm, are parallelizable (see Section 3.11), it seems natural that GPUs would be a good platform for running said algorithms. With the release of Nvidia's *RTX* graphics cards in 2018, which have hardware accelerated ray tracing, this statement is more true now than ever before.

Before moving on to the real-time AO implementation in Chapter 5, it is first beneficial to get an overview of how GPUs are programmed, and how the ray tracing functionality is managed.

4.1 Graphics Application Programming Interface

GPUs were originally designed to do a small set of rendering tasks. Thus, making the entire GPU, with all its functionality, available to the programmer would likely not be the most effective solution[1]. Instead, the development of graphics *application programming interfaces* (APIs) became the norm, with one of the first being *OpenGL*[41]. First introduced in 1992, it allowed developers to write code in *C*, which during run-time would call into the GPU's *driver*. The driver then instructed the GPU to perform the required tasks. The driver thus acted as a middleman between the programmer and the GPU. This was beneficial for a number of reasons:

- It allows programmers to write less code.
- It alleviated the task of implementing the crucial core algorithms, e.g. rasterization and texture sampling, from the programmer onto the GPU vendors' driver development teams.
- Most importantly, it allowed for a common programming interface, instead of each GPU vendor having their own libraries etc. for interacting with their GPUs.

[1]This is what is done for CPUs; by writing assembly code, the programmer gets access to virtually all the functionality the CPU has.

Since 1992, OpenGL has evolved a lot, and it is a larger API today than it was back then. Other graphics APIs also exit today, e.g. *Vulkan*[42], *Direct3D*[43] and *Metal*[44]. Some APIs are cross-platform, meaning they can run on multiple OSs, e.g. OpenGL and Vulkan. Others are proprietary, e.g. Direct3D (*Microsoft Windows*) and Metal (*macOS* and *iOS*). With the introduction of graphics APIs, the *graphics pipeline*, the steps GPUs follow to render images, became more standardized. In recent years, the *rendering pipeline* has looked something like what is shown in Figure 4.1.

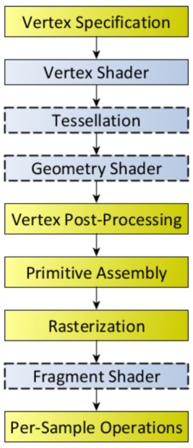


Figure 4.1: The OpenGL graphics pipeline[45].

Today, GPUs do not only do graphics related work as they did in the early days. This is due to how GPUs' design have changed over the years. In the beginning, a GPU was merely a hardware implementation of rasterization and texture sampling. Over time, this has changed, and a GPU now has many cores, each of which is able to execute any instruction it is told to execute. This has led to *compute* becoming a common use case for GPUs. Compute-workload can be just like any other computation performed on the CPU. However, generally, it is a computation that has one or more parts that is parallelizable, which when run on GPUs can result in speed-ups. Because of this, most modern graphics APIs have some compute functionality that usually is a standalone pipeline, i.e. a *compute pipeline*. There also exist compute-only APIs, e.g. *OpenCL*[46] (cross-platform) and *CUDA*[47] (Nvidia). In such APIs, the GPU is not considered to be a graphics processing unit, but rather a *general* processing unit.

Compute APIs have one main advantage over graphics APIs: when using compute in a graphics API such as OpenGL, the compute work is written in a *shader* that has its own language, and thus, its own restrictions. In the case of OpenGL, this language is the *OpenGL Shading Language*[48] (*GLSL*). A compute API like CUDA, however, exposes a lot more specific behaviour and functionality which allows the programmer to utilize the GPU a lot more precisely than what is possible in GLSL. Therefore, if ray tracing was to be parallelized, it would be natural to utilize e.g. CUDA instead of OpenGL's compute pipeline. However, compute APIs like CUDA do not standardize the ray tracing implementation; implementing an efficient ray tracer is left to the programmer. This is time consuming, and not a trivial task. An example of a path tracing implementation using CUDA is Pixar's *RenderMan*[49][2].

However, as ray tracing is generally a graphics technique, another option is to do as was done with rasterization: add a *ray tracing pipeline* to graphics APIs, i.e. Vulkan and Direct3D. This makes the task easier for the programmer, and offloads much of the responsibility of making the ray tracing fast onto the driver. A second step would be to add specialized hardware to the GPU, which is designed to do ray tracing. This is what Nvidia has done with their newest consumer graphics cards line: *RTX*[50].

4.2 Ray Tracing in Vulkan

Vulkan was the API of choice for this thesis, due to it being the only API with ray tracing capabilities that is also cross-platform. Therefore, a brief overview of Vulkan and its ray tracing functionality will be given here.

4.2.1 Vulkan at Its Core

At the core of Vulkan applications are instances (*VkInstance*[51]), and physical and logical devices (*VkPhysicalDevice*[52] and *VkDevice*[53], respectively). When working with Vulkan, the first thing that must happen is an instance of Vulkan being created. The instance can be queried for physical GPUs that have Vulkan support, and one GPU is typically selected. Finally, given the selected physical device, the requested validation layers[54], extensions[55], physical device features[56] and queue indices[57], a logical device can be created. This device is what is supplied to most Vulkan functions when a resource is to be managed, of which an example is shown below (notice the first parameter):

```
VkResult vkCreateImage(VkDevice device,
                       const VkImageCreateInfo* pCreateInfo,
                       const VkAllocationCallbacks* pAllocator,
                       VkImage* pImage);
```

After the initial setup, the programmer is free to create and manage other Vulkan resources that can be used to produce the desired end result. One concept that is crucial to getting any work done in Vulkan are *queues*[57]. They are where *command buffers*[58] are submitted to. Once a queue receives a command buffer, and it is available to do some work, it executes the commands stored in the buffer. There exist various queues for different purposes, but generally the *graphics queue* is used in graphics applications.

As all modern graphics APIs, Vulkan aims to make the task of writing code that is to run on a wide range of GPUs easier. When comparing Vulkan code to what driver-style code would look like, it is at a higher abstraction level, and it requires less code to achieve the same result. However, in the API setting, Vulkan is considered a verbose, low-level API compared to OpenGL. Vulkan's design is based on the idea that programmers should be explicit about their intentions. The reason behind this design is to better allow the driver to make *legal* optimizations according to the specification. An example of this are image layouts (*VkImageLayout*[59]). When creating an image in Vulkan (*VkImage*[60]),

[2]It is important to note that this implementation does use Nvidia's *OptiX* ray tracing *software development kit*.

the image has a layout associated with it. The layout specifies how the data is laid out in memory, and thus how the driver should access it. The different layouts, e.g.

- VK_IMAGE_LAYOUT_UNDEFINED
- VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL
- VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL

are implementation dependent. This means that different Vulkan implementations can use a different data layout for the same *VkImageLayout*. This allows the data layout to be designed with a specific GPU's architecture in mind. However, this shifts some responsibility from the driver onto the programmer. The programmer should ensure that the optimal image layout is set when interacting with an image in a specific way. E.g. when writing to an image from the fragment shader, its layout should be *COLOR_ATTACHMENT_OPTIMAL*.

4.2.2 Shaders and SPIR-V

Vulkan introduced a new way of dealing with shaders compared to other graphics APIs. The general sequence of actions performed when using shader code written in a language such as GLSL in OpenGL, or HLSL in Direct3D, is shown in Figure 4.2. There are a few suboptimal things that this results in:

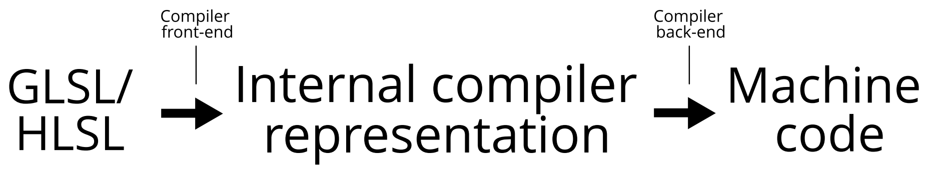


Figure 4.2: General view of shader language compilation in OpenGL/Direct3D.

- The driver's compiler is responsible of translating human-readable code into an internal format. This inserts a possible point of failure should a driver not do this correctly.
- During code optimization, the driver's compiler must be sure to adhere to the specifications, and not make any *illegal* optimizations.
- When the creation of a shader object happens, the entire process shown in Figure 4.2 must be done. This is actually unnecessary. The first step could be done as a preprocessing step if a common intermediate representation was agreed upon. This would result in faster shader creation when an application is running.
- Working with different shader languages becomes more complicated for compiler teams, as each language requires a language → intermediate format compiler front-end.

To address this, the Khronos Group developed SPIR-V[61]: an intermediate format. The Khronos Group is responsible for developing the necessary tools for GPU vendors, and programmers, to use the format[62][63]. This suite of tools includes, but is not limited to:

- SPIR-V Generator: GLSL/HLSL → SPIR-V.
- SPIR-V Optimizer: optimize SPIR-V without breaking its validity.
- SPIR-V Disassembler: SPIR-V → human-readable SPIR-V representation.
- SPIR-V Cross: reflection and SPIR-V → GLSL/HLSL.

When a shader is to be used with Vulkan, its SPIR-V byte-code should first be generated *offline* by the *glslangValidator* tool. Then, during run-time, the byte-code is loaded and passed to the *vkCreateShaderModule* function through a structure. This invokes the driver, which uses its compiler to generate the appropriate machine instructions which can be run by the GPU's shader cores.

4.2.3 Vulkan Extensions

When working with Vulkan, it is important to know if the functionality used is a part of the *core functionality*. The core includes all functionality which a GPU-driver combo that is Vulkan compatible should/must support. The core is ever evolving, with new functionality being added on a regular basis. This new functionality comes from one of two places:

1. It is completely new functionality in all of Vulkan.
2. It has been promoted to the core from an *extension*.

A Vulkan extension is a way for programmers to add new functionality to Vulkan, without it having to be accepted by the *Vulkan Group*. This can be beneficial as adding new functionality to the core is not something that is done quickly. The Vulkan Group consists of many various companies with different interests. Thus, for new functionality to be accepted, more than one partner has to see the benefit of supporting the new functionality in their drivers. If the new functionality is very specific to a particular use case, it might not pass. This is where extensions are useful. GPU vendors can themselves decide if they want to support an extension. And when an extension becomes popular, there is the chance that it will be promoted to be a part of the core functionality. This was the case with *YCbCr* samplers and images.

Thus, when using extension functionality, the Vulkan application must ask for whether or not the functionality is supported by the driver.

4.2.4 VK_NV_ray_tracing

At the time of writing, Nvidia is the only GPU vendor with consumer ray tracing GPUs. The ray tracing functionality is therefore, not unsurprisingly, an extension in Vulkan. Ray tracing in Vulkan can be activated by checking if the physical device supports the *VK_NV_ray_tracing* extension[64], and requesting it upon logical device creation.

The ray tracing extension essentially creates a new pipeline: the *ray tracing pipeline*. The extension contains new data types, flags, functions etc. To avoid rewriting the entire specification, only a few key parts will be discussed here.

VkGeometryTrianglesNV

This new structure is used to specify the properties of the input vertex data. It also has support for indexed vertex data. It is reminiscent of *VkVertexInputBindingDescription* and *VkVertexInputAttributeDescription*, which are used when creating a rasterization pipeline. One important thing is that although the buffer with the vertex data can contain other data as well, only the vertices are of importance when building acceleration structures. Thus, the correct vertex count and stride must be supplied to skip irrelevant data.

vkCreateAccelerationStructureNV()

One of the key parts of the ray tracing GPUs are their ability to do fast hardware-accelerated intersection testing. This requires an acceleration structure, and in this case, multiple structures. The setup is as follows:

1. For each 3D mesh of vertices:
    - A *bottom-level* acceleration structure is created.
    - Memory is allocated for the acceleration structure, and bound to the acceleration structure's handle.
2. A *top-level* acceleration structure is created. Memory is allocated for the acceleration structure, and bound to the acceleration structure's handle.
3. Each bottom-level acceleration structure is built.
4. The top-level acceleration structure is built using all bottom-level acceleration structures.

vkCmdTraceRaysNV()

This is the function that, when put into a command buffer, will actually launch the ray tracing shaders invocations. The shader configuration will not be discussed here, as it is explained well in the specification[64]. Before this function is called, the appropriate ray tracing pipeline, and accompanying descriptor sets, have to be bound.

There are two familiar parameters in this function: *width* and *height*. These specify the dimensions of a grid, usually with the size of the image to be rendered. For each cell in the grid, a ray tracing shader invocation is issued. For any given shader invocation, its *xy*-coordinates in this grid are available in the *gl_LaunchIDNV.xy* decoration in SPIR-V. These coordinates are useful when tracing rays, as they may be used to calculate the ray's direction (see Section 2.2).

traceNV()

A ray tracing shader invocation does not actually perform any ray tracing implicitly. To perform ray tracing, the *traceNV()* function needs to be called in the shader. Its main argument is the acceleration structure which is to be checked for intersections. The acceleration structure has to be made available in the shader through a *uniform buffer*[65].

This design allows for many ray tracing commands to be performed per ray tracing shader invocation. As will become evident in the next chapter, this is useful when multiple rays needs to be fired for the same pixel, but invoking a new ray tracing shader for each ray is unnecessary.

Vertex Data

Other vertex data, e.g. normals, texture coordinates etc. are not available through the acceleration structure. Usually, they are passed through uniform buffers. SPIR-V provides the *gl_InstanceCustomIndexNV* and *gl_PrimitiveID* decorations which are used to index into uniform buffers to access the correct mesh and face respectively, given that the *traceNV()* function intersected some geometry.

Updating the Acceleration Structure(s)

There are two scenarios that can occur which would require the acceleration structures to be updated:

- An entire mesh is transformed: in this case, the internal mesh structure is static, but it is transformed within the scene. This means that *only* the top-level acceleration structure needs to be rebuilt.
- The internal structure of a mesh is changed: in this case, the bottom-level acceleration structure for the given mesh must first be rebuilt, and then the top-level acceleration structure must be rebuilt.

In neither scenario is the required actions performed by Vulkan implicitly, and they have to be done manually by submitting certain command buffers onto the correct queue.

Chapter 5

Approximating Global Illumination with Real-Time Ambient Occlusion

When the restraint to calculate how light interacts in a scene in real-time is lifted, the global illumination techniques discussed in Chapter 2 produce the most realistic results. However, in the setting of interactive applications, e.g. computer games, this is not an option. This is where good approximations become relevant.

5.1 Ambient Occlusion

AO is an approximation of global illumination. It emulates the complex interactions between the diffuse inter-reflections of objects[8]. It also assumes that all light in the scene comes from an infinite uniform environment light, e.g. a distant sun. Equation 5.1 expresses this approximation[66].

$$A(p) = \int_{\mathcal{H}} V(p, \omega_i) \cos\theta_i d\omega_i \tag{5.1}$$

The visibility function, $V(\omega_i)$, checks if the point p is occluded along the direction ω_i. The basic version of the function returns 1 for any occluded direction ω_i, and 0 for any unoccluded direction ω_i.

Just as any other integral, the one in Equation 5.1 can be rewritten in terms of the Monte Carlo Estimator (see Section 2.3). This is shown in Equation 5.2.

$$A(p) = \frac{1}{N} \sum_{i=1}^{N\to\infty} \frac{V(p, \omega_i) \cos\theta_i}{pdf(\omega_i)} \tag{5.2}$$

Although AO, as shown in Figure 1.4, is only an approximation of global illumination, it is still time consuming to calculate. To create good results, it requires many samples of the hemisphere. AO has therefore been reserved for offline rendering that does not require the level of quality which global illumination produces.

5.2 Screen-Space Ambient Occlusion

Another technique that tries to approximate global illumination, is SSAO. Unlike regular AO, SSAO is not a ray tracing algorithm. As the name indicates, SSAO works in screen-space, and it uses a depth buffer from which the depth of neighboring pixels are sampled, and are used to calculate the occlusion value for the current pixel[9]. A benefit of SSAO, besides the fact that it is a fast algorithm, is that its computation time is constant for a given screen size. This makes it a very predictable algorithm.

There are various version of SSAO, and one of the industry-leading is HBAO [67], developed by Louis Bavoil of Nvidia. “Unlike previous SSAO variants, HBAO uses a *physically-based algorithm* that approximates an integral with depth buffer sampling”[10].

Although SSAO implementations produce good results, and are generally fast algorithms, they can suffer from artifacts due to being screen-space algorithms that try to approximate a world space effect. The results are also worse than what regular AO, with a high sample count, is able to produce.

5.3 Real-Time Ambient Occlusion

Given that AO produces better results than SSAO, it would be a step forward if AO was used in real-time applications instead of SSAO. So far this has not been an option. However, with the new RTX GPUs, it might now be possible. What follows are the theoretical and implementation details for the real-time AO implementation that this thesis set out to create.

5.3.1 Theoretical Details

There is one major challenge with doing AO in real-time: sampling the hemisphere. In real-time, the number of samples that can be taken is small compared to what one would ideally want to use to get good results. This essentially reduces the problem of real-time AO to a sampling problem (given that the ray tracing can be done fast). Then, there is also the problem of trying to reduce sampling artifacts that are caused by the low sample count. Relevant background theory for both these areas will be presented below.

Sampling Technique

It seems logical to begin with the same hemispherical sampling technique presented in Section 2.5.6. Given two random uniform values u_0 and u_1, a point on the hemisphere can be generated uniformly, thus giving a direction ω_i. As briefly mentioned in Section 2.5.6, there is also the option to use cosine-weighted sampling instead of uniform sampling. As AO is a way to approximate how light interacts in a scene, cosine-weighted sampling may produce better results than uniform sampling.

The next order of business then becomes how to best generate the random values u_0 and u_1?

Evenly Spaced Samples

The simplest option is to have two for-loops that generate evenly spaced samples in the 2D domain ($u_0, u_1 \in [0, 1]$). An example of this can be seen in Figure 5.1. This sample distribution does, however, result in very visible sampling patterns,

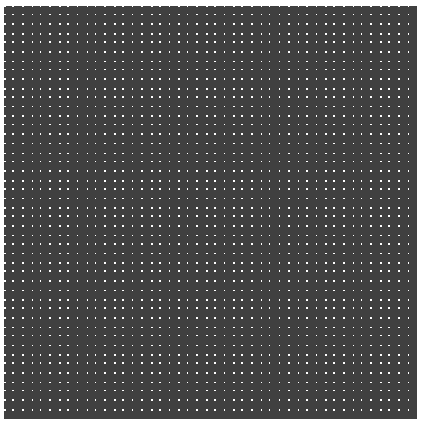


Figure 5.1: Example of evenly spaced samples.

and does not produce good looking results with few samples.

White Noise Samples

Another option is white noise. It is essentially what the random number generator in the path tracing engine does: upon request, it returns a seemingly random value. A visualization of a white noise sampling pattern is shown in Figure 5.2. However, white noise is generally not a good option when the amount of sam-

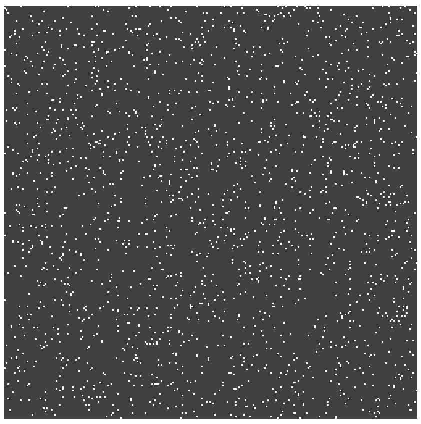


Figure 5.2: Example of white noise samples.

ples is very limited. Due to their completely random nature, they may clump together, resulting in areas being left unsampled.

Blue Noise Samples

Blue noise differs from white noise in a very important aspect: the points are not placed at complete random. The first point is placed at random, but all the following points are placed in such a way as to try and cover as much of the sampling domain as possible. An example is shown in Figure 5.3. A thorough description of *Mitchell's Best Candidate Algorithm*, which was used to generate the samples, can be found in a blog post by Alan Wolfe[14]. As can be seen in

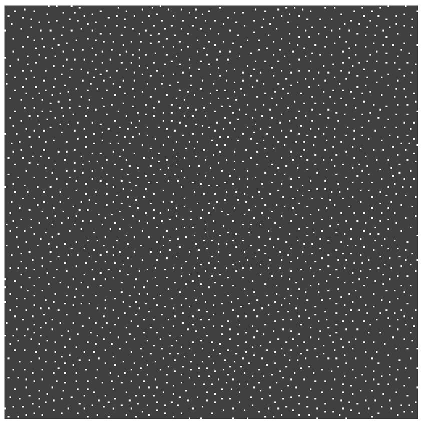


Figure 5.3: Example of blue noise samples.

the figure, there are no visible patterns, and the samples appear random even though they are not. Due to the way blue noise samples are generated, blue noise has the desirable property that it has a lower starting error than many other sampling distributions. This means that although it converges just as fast/slow, a low number of samples can still give a reasonably good representation of the sampling domain[14].

Box Filter

Next comes sampling pattern mitigation. Due to the small amount of available samples, it is still likely that some sampling patterns will persist. A fairly simple way to reduce these patterns' impact on the final image, is to blur the image. There exist various types of blurs that can be applied with varying reach and effect. For the purpose of blurring, a simple box filter is used. A box filter applies a kernel, e.g. 3x3 kernel, onto each texel in the image. At each texel location, it computes the new value of the texel by multiplying the values in the kernel cells with the texel value at the corresponding location in the image. A 3x3 box filter kernel is shown in Figure 5.4.

Temporal Integration

An important thing to take note of is that two frames that follow each other will be very similar in content. Although the camera might change its position and

0.111	0.111	0.111
0.111	0.111	0.111
0.111	0.111	0.111

Figure 5.4: 3x3 box filter kernel.

orientation between the two frames, most of what can be seen in frame $n-1$ will also be present in frame n. This leads to an interesting observation: if a point p is visible in both frames, its color value will have been calculated twice. Also, if AO has been calculated for p, it will be baked into the color values for both frames.

Now, if the same sample values, u_0 and u_1, were used in both frames, the AO values will be identical. However, if different sample values were used, the two AO values will be different. In the latter case, information about the AO at p is available with twice the amount of samples. E.g. if one frame takes 64 AO samples at p, a total of 128 samples are taken over the two frames. If the color values at the texel locations corresponding to p's location in the images are averaged, a *more* accurate estimate of the actual color value is obtained. This procedure is expressed in Equation 5.3,

$$frame_n(u_n, v_n) = \frac{frame_n(u_n, v_n) + frame_{n-1}(u_p, v_p)}{2} \tag{5.3}$$

where u_p and v_p are the texel coordinates of p in the previous $frame_{n-1}$. This results in integration over time, i.e. temporal integration.

To be able to perform temporal integration, u_p and v_p are needed. These values can be calculated given the world coordinates of p and the previous frame's, $frame_{n-1}$, camera properties. Given $frame_{n-1}$'s camera properties, the *view-projection* matrix, m_{vp}, for that frame can be calculated. The texel coordinates for p in $frame_{n-1}$ can then be calculated by projecting p by m_{vp}, performing a manual perspective division, and transforming the xy-coordinates from screen-space to UV-space. This process is shown in Section 5.3.2, and is also described in detail in the presentation titled *Temporal Reprojection Anti-Aliasing in INSIDE* by Lasse Jon Fuglsang Pedersen[68], presented at the *Game Developers Conference* in 2016. Note that the procedure described above is only valid for static scenes. For dynamic scenes, refer to [68] for further implementation details.

5.3.2 Implementation Details

Rendering Setup

Before describing the implementation details of the various techniques used, it is necessary to first describe the rendering setup. Figure 5.5 shows the various render passes and what each render pass reads (R) and writes (W).

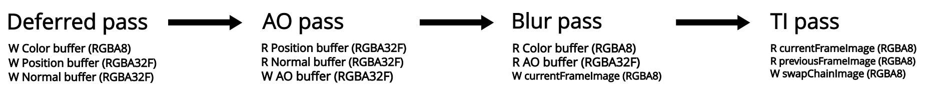

Deferred pass
W Color buffer (RGBA8)
W Position buffer (RGBA32F)
W Normal buffer (RGBA32F)
AO pass
R Position buffer (RGBA32F)
R Normal buffer (RGBA32F)
W AO buffer (RGBA32F)
Blur pass
R Color buffer (RGBA8)
R AO buffer (RGBA32F)
W currentFrameImage (RGBA8)
TI pass
R currentFrameImage (RGBA8)
R previousFrameImage (RGBA8)
W swapChainImage (RGBA8)

Figure 5.5: Visualization of the render passes in the application.

First comes the deferred render pass. For each pixel in the final image, it traces a ray in the correct direction and checks for an intersection. There are then a few possibilities:

- If the closest intersection is with a light, the color of the light is stored in the color buffer, and a value of 0 is stored in the position and normal buffers.
- If no intersection was found, a custom color is stored in the color buffer, and a value of 0 is stored in the position and normal buffers.
- If an intersection with some geometry is found, simple diffuse-only Phong lighting is calculated and stored in the color buffer, the position and normal are stored in the position and normal buffers respectively. Finally, the fraction of point lights that are visible from the intersection point is stored in the w-component in the position buffer.

Then comes the AO render pass. It samples both the position and normal buffer and opts for an early exit:

- If either the position is 0, or the fraction of visible point lights is larger than 0, a value of 0 is stored in the AO buffer.
- Otherwise, AO is calculated and stored in the AO buffer.

Next comes a render pass that consists of two subpasses (see Section 7 in the Vulkan specification[69]), both of which rasterize two triangles that cover the entire screen. The first subpass blurs the contents of the AO buffer, and combines that result with the contents of the color buffer. An important thing to note here is that only the AO buffer is blurred, and it is only blurred if the center texel has a value other than 0. This is to avoid unnecessary texture lookups. The second subpass performs the temporal integration by interpolating between $frame_{n-1}$'s final result and the current result of $frame_n$.

The last thing that happens, which is not shown in Figure 5.5, is that the final result for $frame_n$ is *blitted* into a buffer that always contains the result from the previous frame. This will be used in $frame_{n+1}$'s temporal integration subpass.

Blue Noise

Out of the three sample distributions described, i.e. evenly spaced, white noise, and blue noise, blue noise was chosen. To calculate 2D blue noise upon request in real-time is not a feasible option; it takes too long. Therefore, an array of 2D points is pre-calculated using the algorithm described in [14]. The array is stored explicitly in the shader that performs AO calculations. A small example is shown below:

```
struct SamplePoint { float u0; float u1; };
SamplePoint samplePoints[4] = {
    {0.869141, 0.657227},
    {0.379883, 0.838867},
    {0.349609, 0.327148},
    {0.610352, 0.965820} };
```

In the actual implementation, 64 samples are stored in the array.

However, in order for temporal integration to work well, the *same* samples should not be used between two consecutive frames. This is dealt with by performing a rotation on the indices used to retrieve the blue noise samples. The procedure is as follows:

1. First, an angle of rotation is calculated. This is done by sampling a normalized value from a blue noise texture (downloaded from *http://momentsingraphics.de/?p=127*). This is then added to the result of multiplying the current frame number and the golden ratio. The fraction of this number is then found and multiplied with 2π, giving the angle of rotation in radians. The procedure is shown below:

```
vec2 blueNoiseUV = gl_LaunchIDNV.xy /
            vec2(BLUE_NOISE_IMAGE_SIZE);
float rotationAngle =
            texture(blueNoiseImage, blueNoiseUV).r;
rotationAngle += GOLDEN_RATIO * float(frameNumber);
rotationAngle = fract(rotationAngle) * TWO_PI;
```

   Note that the BLUE_NOISE_IMAGE_SIZE is 64, and so, a sampler with a *repeating* addressing mode must be used.

2. Next, a double for-loop is started, where each inner-most instance represents a single sample AO of the hemisphere. Each such sample will have an x and y index associated with it from the for-loops. These indices are transformed to be centered about $(0, 0)$, instead of the half-way point of the for-loops. This is shown below:

```
int sampleCenterX = numOcclusionSamples / 2;
int sampleCenterY = numOcclusionSamples / 2;
for (int y = 0; y < numOcclusionSamples; y++) {
        for (int x = 0; x < numOcclusionSamples; x++) {
            vec4 sampleRelative =
                    vec4(x - sampleCenterX, 0.0f,
                         y - sampleCenterY, 1.0f);
        }
}
```

3. Then, the actual rotation is performed. Here, the indices are set to be in the *xz*-plane, and so a rotation about the y-axis must be performed to achieve the desired result. The procedure is shown below:

```
// Calculate and apply rotation matrix around y-axis
mat4 xzPlaneRotation = Rotate(blueNoiseRotationAngle,
                              vec3(0.0f, -1.0f, 0.0f));
```

```
vec4 rotatedSampleRelative = xzPlaneRotation *
                             sampleRelative;
int rotatedSampleRelativeX =
        int(round(rotatedSampleRelative.x));
int rotatedSampleRelativeZ =
        int(round(rotatedSampleRelative.z));
// Move back to original coordinate system
int rotatedSampleX = rotatedSampleRelativeX +
                     sampleCenterX;
int rotatedSampleZ = rotatedSampleRelativeY +
                     sampleCenterY;
```

4. The second to last step is to make sure that the new rotated indices are within the range of the blue noise sample space. This is accomplished by wrapping the *rotatedSampleX* and *rotatedSampleZ* indices about their maximum/minimum values, i.e. *numOcclusionSamples*. This is fairly simple, and the code is omitted here.

5. Finally, the index, with which the array of blue noise samples is to be indexed, is calculated. This is shown below:

```
int idx = wrapRotatedSampleZ * numOcclusionSamples +
          wrapRotatedSampleX;
```

The blue noise sample points can now be retrieved from the array, and a direction can be sampled.

Sampling

The two selected sample points are passed as parameters to the given sampling function. Having chosen cosine-weighted hemisphere sampling, the implementation follows directly the one described in Chapter 13.6.3 in [11] and is omitted here. At this point, the ray's origin, p, and direction, ω_i, are available.

Calculating AO

For each AO sample, a ray is traced against the scene. There are two scenarios that can occur at this point:

- The ray intersects with the scene: an occlusion value is calculated.
- The ray does not intersect the scene: no action is taken.

In the case where an intersection is detected, the visibility term $V(p, \omega_i)$ must be calculated. This is shown below:

```
if (hitDist.x >= 0.0f) {
#if SAMPLE_COSINE
  occlusion += VisibilityFunction(hitDist.x);
#elif SAMPLE_UNIFORM
  occlusion += VisibilityFunction(hitDist.x) *
               dot(isectNormal, occlusionRayDir);
#endif
}
```

Note that if cosine-weighted sampling is used, the cosine term in Equation 5.2 disappears. This is because when cosine-weighting samples, $pdf(\omega_i) = \cos\theta_i$, and the cosine terms cancel out.

As mentioned previously, the value returned by the visibility function can be 1 for the most basic implementation, but more complicated implementations can give better results. This is because of the following argument: if p is occluded along ω_i by an occluder that is far away, the amount of indirect light that will reach p is likely to be large. If the occluder instead is very close, the amount of indirect light that will reach p is likely to be much lower. Thus, the visibility function is often implemented in such a way that it takes the distance d to the occluder into account. A way to view the visibility term is in the form of a question: "How subject to indirect light is p along ω_i?" If implemented in its binary form, it answers yes or no to a *non*-yes-no question.

Through testing of different visibility functions that could be used in place of the binary function, Equation 5.4 was found to give good results:

$$y = c^{-d} \tag{5.4}$$

where c is a constant that can be adjusted according to the desired visual result. A comparison of Equation 5.4 with different values of c can be seen in Figure 5.6.

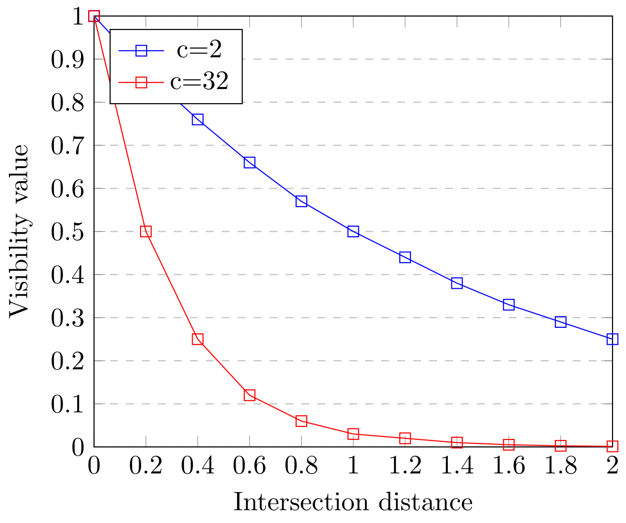


Figure 5.6: Comparison of $y = c^{-d}$ with different values for c

Finally, if the AO is calculated at full resolution, the maximum number of rays that could be traced would be $1920 * 1080 * 64 = 132710400$. Through testing, it was found that reducing the resolution of the AO pass yielded a significant performance boost, without compromising too much on the quality of the results. This means that the AO pass is calculated at a resolution of 960x540, effectively reducing the maximum number of rays traced to $960 * 540 * 64 = 33177600$, $\frac{1}{4}$ of the 1920x1080 case.

Box Filter

The implementation of the 3x3 box filter is straight forward. The 9 texels are sampled individually using GLSL's *textureOffset* function. It allows for the sampling of texels specified by an offset in texel coordinates. An example is shown below:

```
float v = textureOffset(image, uv, vec2(-1, -1)).r;
```

These values are then averaged.

Temporal Integration

To perform temporal integration between $frame_{n-1}$ and $frame_n$ for point p, four pieces of data are needed:

- The 3D coordinates of p.
- The camera's details in the previous frame.
- The previous frame's result.
- The current frame's result.

The two triangles that are rendered in this subpass have UV-coordinates associated with them. These are used to sample p's coordinates from the position buffer. Then, the color value from $frame_n$ is loaded using *subpassLoad()*[70]. Next, the position is checked to see if this is a special case, where it is 0. If it is not, p's UV coordinates in $frame_{n-1}$ are calculated, and its color retrieved. Finally, the two color values from the two frames are mixed. The procedure is shown below:

```
vec3 currentFramePosition = texture(positionImage,fUV).xyz;
vec3 currentFrameColor = subpassLoad(currentFrameImage).rgb;
if (currentFramePosition == vec3(0.0f)) {
  outColor = vec4(currentFrameColor, 1.0f);
  return;
}
vec4 previousFrameProjected = previousViewProjection *
        vec4(currentFramePosition, 1.0f);
vec2 previousFrameUV = previousFrameProjected.xy /
        previousFrameProjected.w;
previousFrameUV.y *= -1.0f; // Vulkan specific
previousFrameUV *= 0.5f;
previousFrameUV += 0.5f;
vec3 previousFrameColor = texture(previousFrameImage,
                                  previousFrameUV).rgb;
outColor = vec4(mix(previousFrameColor,
                    currentFrameColor, 0.5f), 1.0f);
```

Chapter 6

Real-Time Ambient Occlusion Results

When looking at the results, there are two main aspects that are of utmost importance:

- Quality: how good are the results?
- Performance: how fast are the results achieved?

In most of CG, but especially in real-time applications, this is a constant trade-off. This chapter will go through the results achieved by the implementation of real-time AO described in Section 5.3.2. Note that unless otherwise specified, a value of $c = 8$ is used in the visibility function, and 64 AO samples are taken per pixel when necessary.

6.1 Visual Quality

6.1.1 Real-Time AO vs Physically Based Global Illumination

By developing a physically based path tracer, physically correct images can be rendered, and compared against their real-time AO counterpart. Because AO is an approximation of global illumination, this comparison is relevant. Before proceeding, two notes on the approximation need to be mentioned:

- Due to how AO is calculated in this implementation, i.e. only for pixels that represent a point that is not directly illuminated by any point lights, the comparison will not be perfect. For a better approximation, shadows should be calculated and added to the result. This was not part of this thesis, and has therefore not been done.
- The physically based results were rendered with area lights as light sources. However, in the real-time AO implementation, point lights were used for simplicity and efficiency. Point lights are not physically possible, and therefore reduce the quality of the comparison.

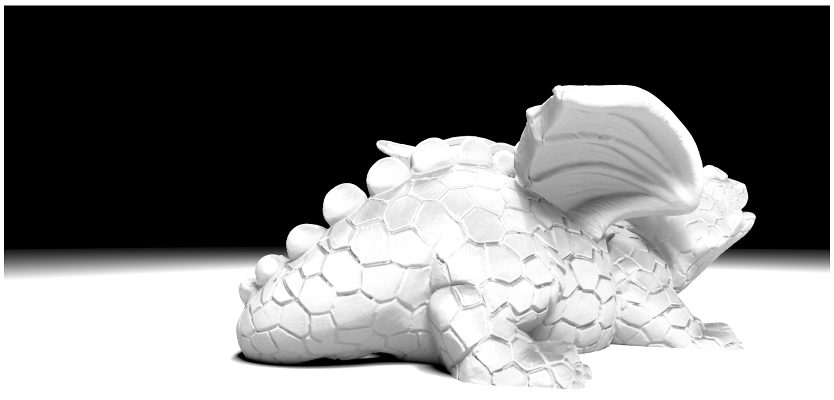


Figure 6.1: Dragon rendered with 8196 SPP from view point 1 (model courtesy of Christian Schüller via PBRT scenes[36]).

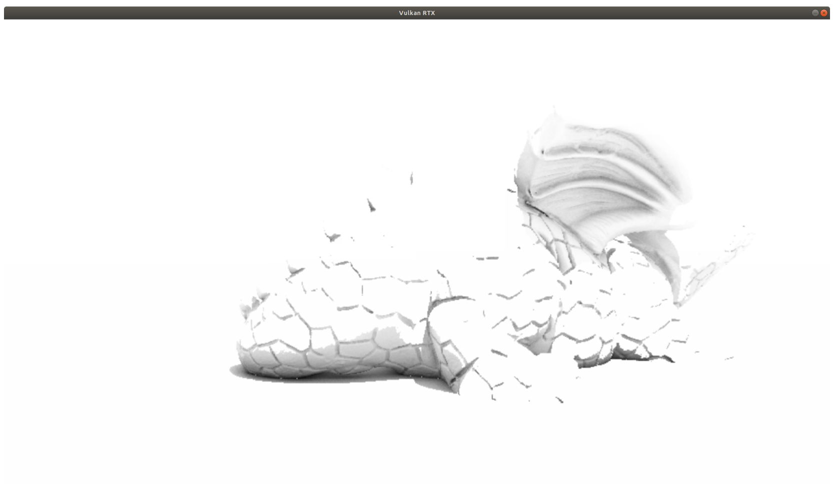


Figure 6.2: Dragon rendered with real-time AO from view point 1 (model courtesy of Christian Schüller via PBRT scenes[36]).

The first comparison is between Figure 6.1 and 6.2. First, note how there is AO in the crevices of the dragon scales. This shows that the technique is able to capture AO in very small areas. Next, there is also visible AO on the down-facing side of the wing, in the crevice on the neck, and finally below the dragon. Also note that at the top of the dragon, there is little to no AO. This is because one or more point lights have a direct effect on these areas. Thus, no AO is calculated there. When compared against the global illumination result, it is fairly similar. The areas below the tail are not as dark, and the wing is not as dark at the back. However, from this point of view, the results are comparable.

Next, Figure 6.3 and Figure 6.4 are compared. First notice how the crevices of the scales have AO, which is similar to that of the global illumination result. Next, see how the inside of the mouth has AO. Finally, notice how the area

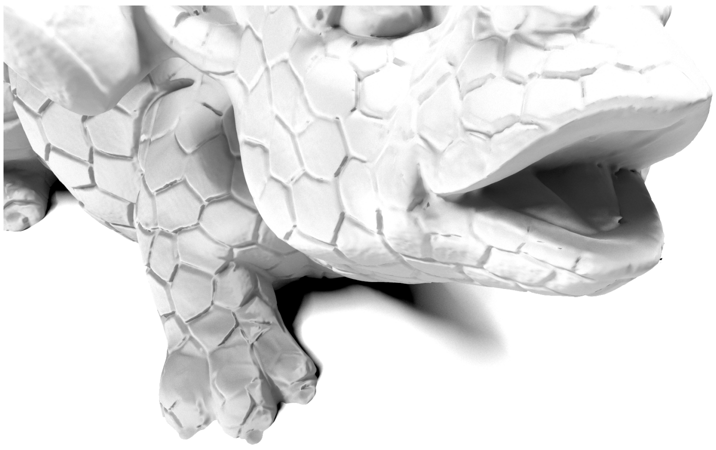


Figure 6.3: Dragon rendered with 8196 SPP from viewpoint 0 (model courtesy of Christian Schüller via PBRT scenes[36]).

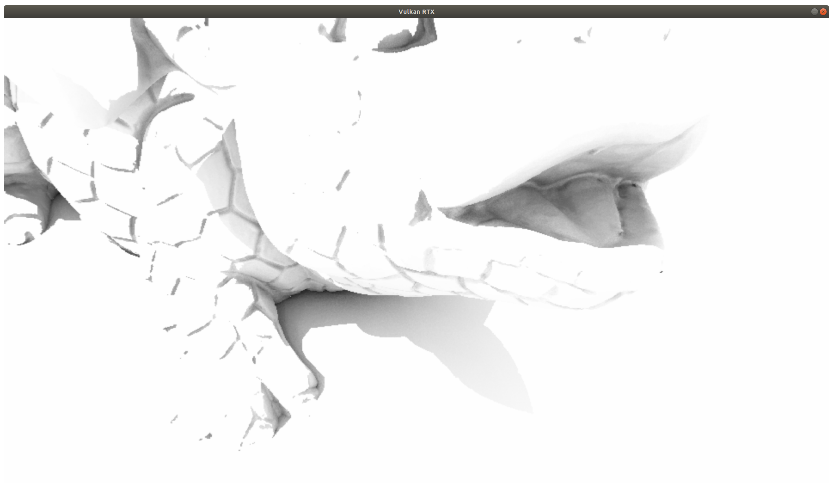


Figure 6.4: Dragon rendered with real-time AO from viewpoint 0 (model courtesy of Christian Schüller via PBRT scenes[36]).

beneath the head is less exposed to indirect light, and is comparable to that of the global illumination result.

For two more comparisons, see the Appendix, Section 7.3, Figure 7.1 - 7.4. Some of the comparisons shown there highlight how some scenes are are quite visibly less comparable.

6.1.2 Real-Time AO vs Offline AO

Another, even more relevant comparison is how well the real-time AO version is able to approximate the offline AO version. Although comparing against global illumination is interesting, offline AO is a more “realistic” target for real-time AO. If the results are highly comparable, this would indicate that high quality real-time AO is obtainable, and a viable option for games. Figure 6.5 and 6.6 shows such a comparison. From this example, it is clear that the real-time version is close to the offline version. However, there are some areas that suffer. Some slight sampling patterns can be seen below the car, above the exhaust, and underneath the rear wing in Figure 6.5. The same problem has been found in other comparisons as well.

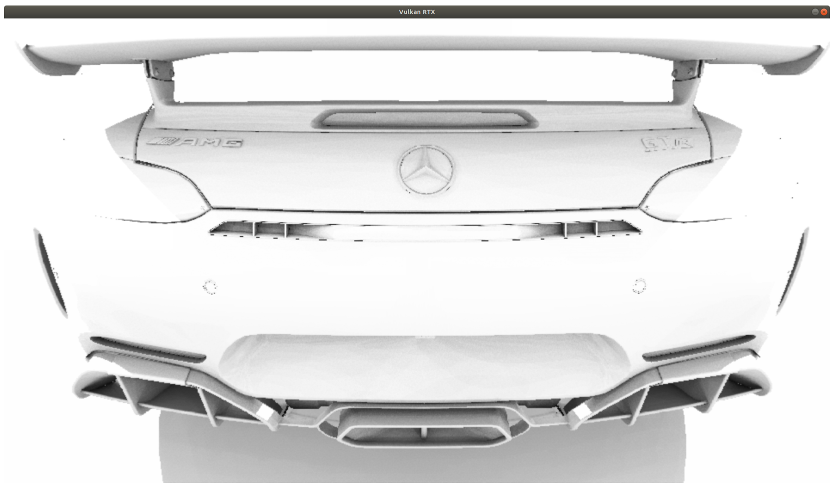


Figure 6.5: Mercedes rendered with **real-time** AO from viewpoint 1 (model courtesy of Thomas F[40]).

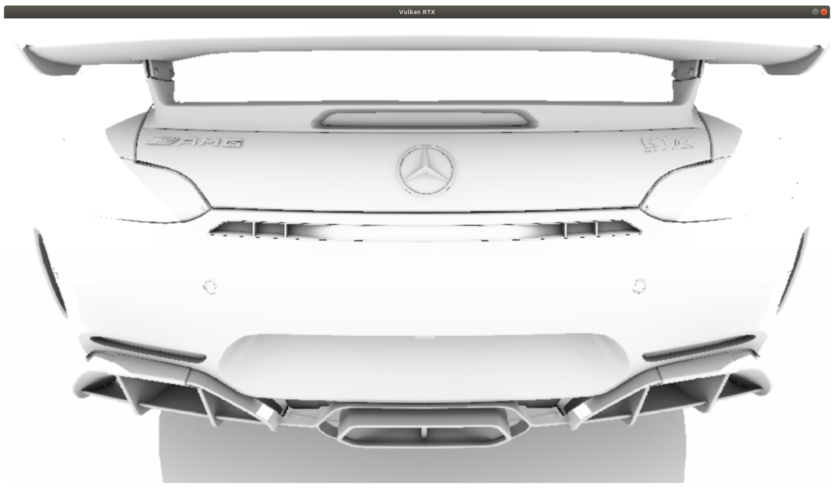


Figure 6.6: Mercedes rendered with **offline** AO from viewpoint 1 (model courtesy of Thomas F[40]).

To get deeper insight into how well the real-time version is able to approximate the offline version, a similarity measure can be used. "Given two sequences of measurements $X = \{xi : i = 1, ..., n\}$ and $Y = \{yi : i = 1, ..., n\}$, the similarity (dissimilarity) between them is a measure that quantifies the dependency (independency) between the sequences"[71]. Two similarity measures were chosen as metrics: *Tanimoto Measure*[71]:

$$S = \frac{X \cdot Y}{||X||^2 + ||Y||^2 - X \cdot Y} \tag{6.1}$$

and *Minimum Ratio*[71]:

$$S = \frac{1}{n} \sum_{i=1}^{n} min(\frac{x_i}{y_i}, \frac{y_i}{x_i}) \tag{6.2}$$

Table 6.1 shows how the two similarity measures rate the real-time version against the offline version. The similarity measures clearly show that real-time

Scene and viewpoint	**Tanimoto**	**Minimum ratio**
dragon.brhan, viewpoint=0	0.999933	0.996596
dragon.brhan, viewpoint=1	0.999974	0.998328
mercedes.brhan, viewpoint=0	0.999848	0.991831
mercedes.brhan, viewpoint=1	0.999795	0.990140

Table 6.1: Similarity between real-time AO and offline AO.

and offline AO results are *very* similar from a data standpoint. However, from a visual standpoint, there are some areas in which the real-time version is lacking.

6.1.3 Visibility Function Comparison

By changing the constant c in the visibility function, different results are obtainable. Unless comparing against the physically based result, there is no "correct" value. Generally, the value of c should be chosen so that it produces the most good looking results, subjectively speaking, for the given application.

A few examples are shown in Figure 6.7, 6.8 and 6.9. It is interesting to notice how a higher value of c, i.e. a quicker fall-off, results in less visible sampling patterns. The reason for this behaviour is that if the visibility function has $c = 1$, all samples are weighted equally. However, the larger c becomes, the less weight are given to samples that hit geometry that is far away. This is explained in greater detail in Section 5.3.2.

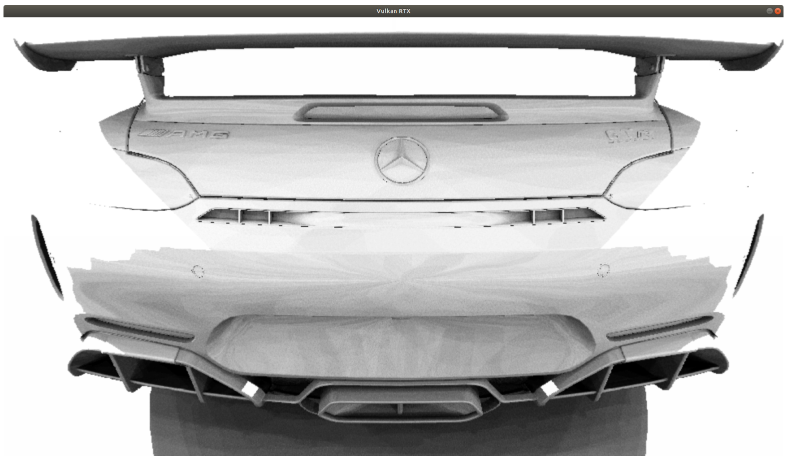


Figure 6.7: Mercedes rendered with real-time AO from viewpoint 1 with $\mathbf{c} = \mathbf{1}$ in the visibility function (model courtesy of Thomas F[40]).

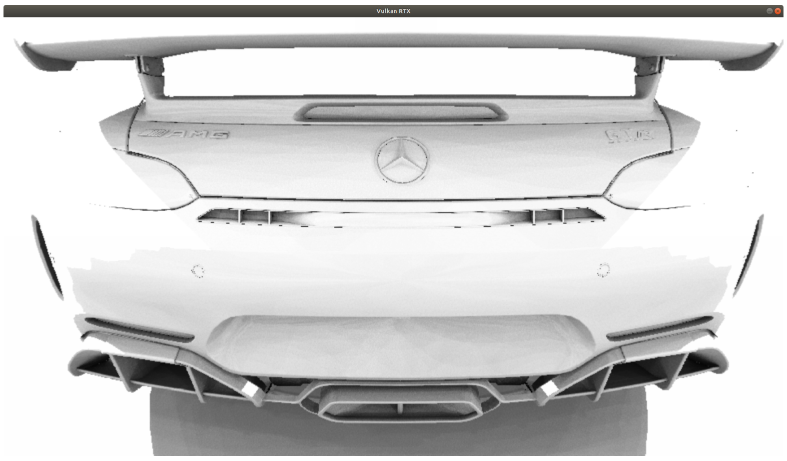


Figure 6.8: Mercedes rendered with real-time AO from viewpoint 1 with $\mathbf{c} = \mathbf{2}$ in the visibility function (model courtesy of Thomas F[40]).

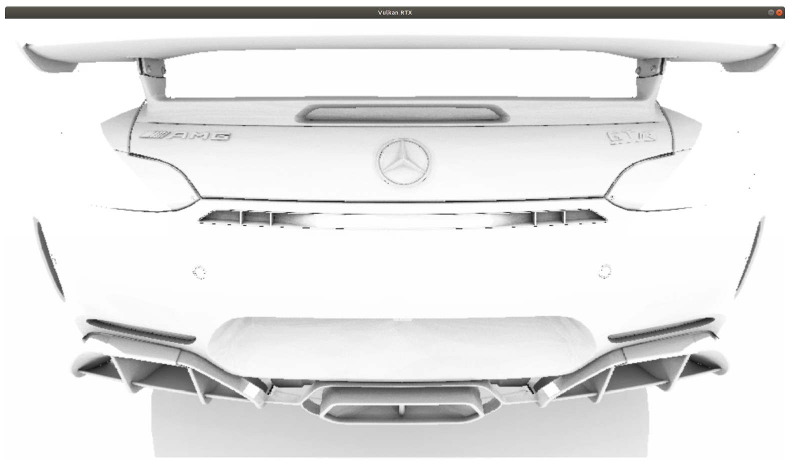


Figure 6.9: Mercedes rendered with real-time AO from viewpoint 1 with **c** = **32** in the visibility function (model courtesy of Thomas F[40]).

6.1.4 Box Filter Comparison

Due to the limited number of AO samples, it is virtually impossible to avoid some sampling patterns being present in the result. A way to try and reduce the pattern's impact, is through the use of a blur. However, blurring also reduces the overall detail in the image, and so, it must be used with caution. A comparison is shown between Figure 6.10 and 6.11.

Although some of the sampling patterns' impact is slightly lessened, what becomes most evident is the loss of overall detail. Even though a box filter was found to give the best result among the filters tested, it still suffers from this rudimentary problem. E.g. see how the lettering and the Mercedes logo are a lot less clear in the blurred result in Figure 6.11.

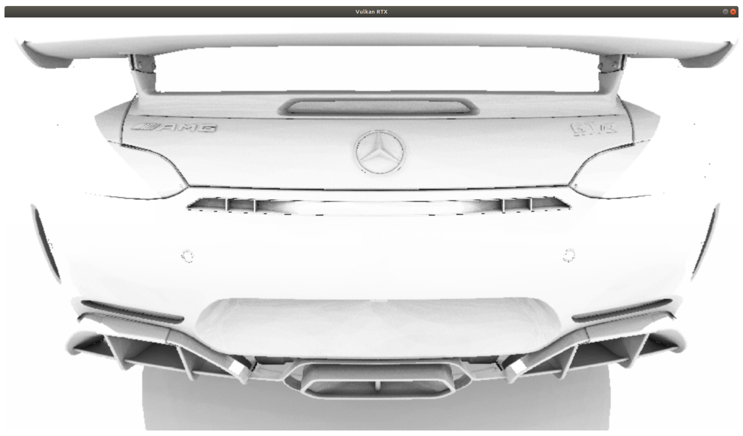


Figure 6.10: Mercedes rendered with real-time AO from viewpoint 1 **without** box filter (model courtesy of Thomas F[40]).

Furthermore, a measure of similarity can be used to confirm the visual impact blurring has. Table 6.2 shows the similarity between the *non*-blurred, and

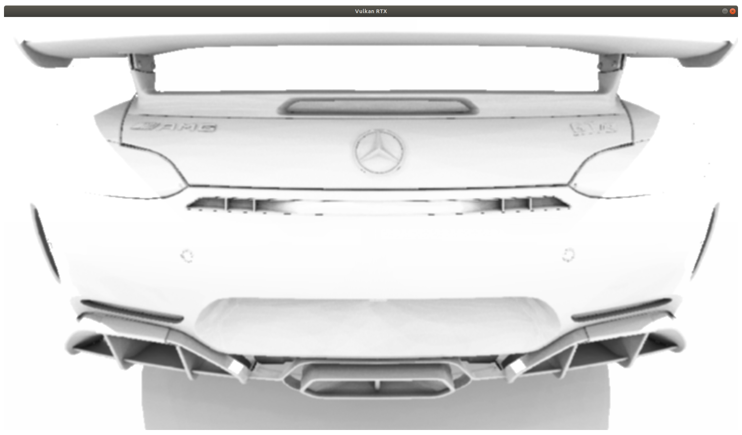


Figure 6.11: Mercedes rendered with real-time AO from viewpoint 1 **with** box filter (model courtesy of Thomas F[40]).

blurred real-time results, when measured against the offline result. Although only slight, there is a decrease in similarity between the blurred real-time result and the offline result, when compared to the *non*-blurred real-time result.

Scene	**Blurred**	**Tanimoto**	**Minimum ratio**
mercedes.brhan	No	0.999795	0.990140
mercedes.brhan	Yes	0.999283	0.985215

Table 6.2: Similarity between **non**-blurred real-time AO, and blurred real-time AO, when measured against offline AO.

6.1.5 Temporal Integration Comparison

The addition of temporal integration helps reduce sampling patterns, and noise, in the AO. A comparison can be seen between Figure 6.12 and 6.13.

Although not easy to see, there is a reduction in noise in Figure 6.13 compared to 6.12. This is especially noticeable under the rear wing, above the exhaust, and below the car. These are large areas in which noise is more visible. For smaller areas, noise is less visible, and thus, temporal integration does not yield much improvement.

As with blurring, a measurement of similarity shows that using temporal integration, slightly increases the similarity when measured against the offline result. This is shown in Table 6.3.

Scene	**Temporal Integration**	**Tanimoto**	**Minimum ratio**
mercedes.brhan	No	0.999662	0.986676
mercedes.brhan	Yes	0.999795	0.990140

Table 6.3: Similarity between temporally integrated real-time AO, and **non**-temporally integrated real-time AO, when measured against offline AO.

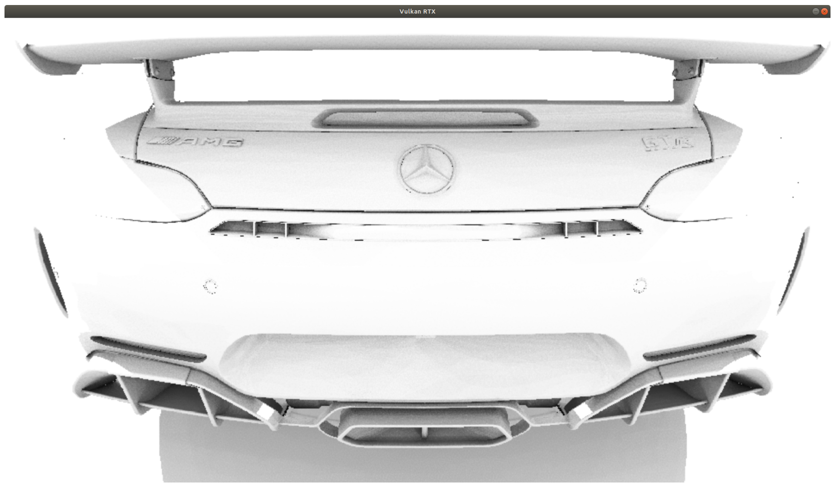


Figure 6.12: Mercedes rendered with real-time AO from viewpoint 1 **without** temporal integration (model courtesy of Thomas F[40]).

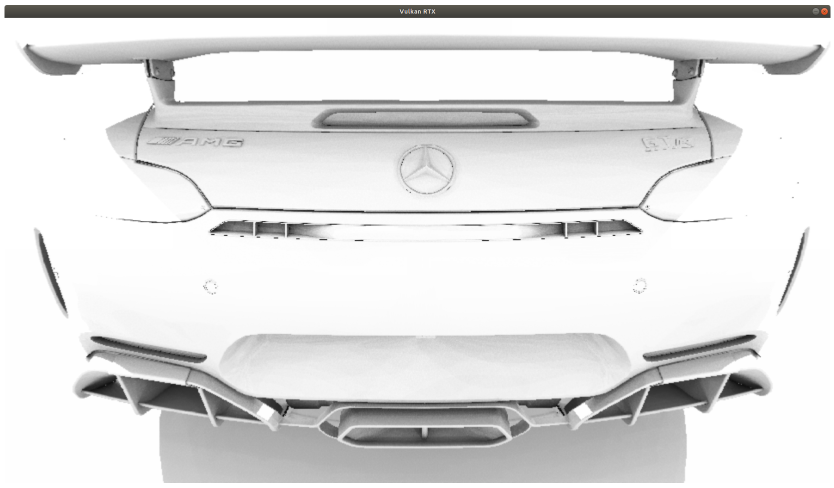


Figure 6.13: Mercedes rendered with real-time AO from viewpoint 1 **with** temporal integration (model courtesy of Thomas F[40]).

6.2 Performance

Two measures were taken when gathering performance timings:

- Only rendering is included in the timings, i.e. not other calculations performed within a frame.
- All timings were gathered when rendering *off-screen*. This means that no results were presented to the display, thus removing the effect of *VSync*[72], and other presentation overhead.

Table 6.4 shows an example of the timings gathered on an arbitrary run of the *mercedes.brhan* scene which has 2.45M triangles. This is the same scene which is displayed in Figure 6.13.

Pass	**Time (ms)**
Deferred pass	~2.89
AO pass	~5.18
Blur pass	~0.53
Temporal integration pass	~0.2
Total (AO only)	**~5.91**
Total (all)	**~8.8**

Table 6.4: Overview of timings for the various passes in the application. Timings are the average time of running for 10 seconds.

There are a few things to take note of here:

- The deferred pass is ray traced. This is not the optimal strategy, as it could just as well be rasterized, which still is faster than ray tracing. However, as this step will vary from application to application, it is not of large importance.
- The blur pass could be optimized to take advantage of the box filter's separable quality[73]. However, time was not spent on this.
- For more complex scenes with moving geometry, the temporal integration would be more complex and time consuming. However, it is unlikely that its time would exceed $1ms$.

The timings do however show that for the given scene, real-time AO is an option as it requires far less than $16.6ms$.

However, a very important thing to take note of here before continuing, is that the viewpoint of the camera, and the lighting in the scene, are incredibly important for the time is takes to calculate AO. When the same scene that gave the timings shown in Table 6.4 is rendered from a different viewpoint (shown in Figure 7.5 in the Appendix, Section 7.3), the total render time was reduced to $1.59ms$, with the AO requiring a mere $0.6ms$. This reduction can be explained by three reasons:

- As many of the rays do not hit any geometry, the intersection testing of the acceleration structure becomes faster.
- Most of the image does not have geometry in it. Thus, the deferred pass gets an early exit for a majority of the pixels.
- AO is only calculated for pixels that have geometry in them, and that are *not* directly illuminated. In this case, this percentage is relatively low. For a scene where most of the geometry does not have direct illumination, the AO would require more time.

To get a better understanding of how well the hardware accelerated ray tracing performs, additional tests with more complex scenes were performed. Table 6.5 shows some of these results. Even though the last entry in Table 6.5 has 3.5x the amount of triangles compared to the scene that gave the results in Table 6.4,

Triangle Count (M)	**AO Time (ms)**
4.9	∼1.2
6.7	∼4.3
8.6	∼5.6

Table 6.5: Rendering times for various scenes with increasing complexity. All scenes were rendered from the same static viewpoint. The scenes can be seen in respective order in the Appendix, Section 7.3, from Figure 7.6-7.8

the AO takes less time. This is due to the viewpoint. Even though the scene has a lot more triangles, they do not occupy as much of the image. This highlights an interesting point: intersecting a simpler acceleration structure many times is more costly than intersecting a more complex acceleration structure fewer times. I.e. deciding upon a good strategy for where and when AO should be calculated is of great importance.

Another important aspect of the performance is the time it takes to update the acceleration structure. There are two scenarios in which this becomes necessary:

- An mesh is altered internally: requires a rebuild of the bottom-level acceleration structure.
- A mesh is transformed within the scene: requires a rebuild of the top-level acceleration structure.

Due to time limitations, only the 2^{nd} option was investigated. The results for the same scenes used in Table 6.5, plus one additional scene, are shown in Table 6.6. This highlights an interesting point: even in a scene with 25.2 million

Triangle Count (M)	**Mesh Count**	**Rebuild Time (ms)**
4.9	4	∼0.157
6.7	5	∼0.16
8.6	6	∼0.16
25.2	15	∼0.18

Table 6.6: Top-level acceleration structure rebuild times for various scenes with varying complexity.

triangles, rebuilding the top-level acceleration structure does not require much time when compared to how long the AO calculations take.

However, some insight into bottom-level rebuilding times can be obtained. By taking a look at the initial build time for the entire acceleration structure (both bottom- and top-level), it is possible to reason about how long it should take to update the bottom-level acceleration structures. The initial build time for the scene with 25.2 million triangles is $\sim 230ms$. Although it is reasonable to expect the initial build to require more time than updates, it is also reasonable to expect that updating the bottom-level acceleration structures for the 15 meshes would take more than a few milliseconds. It would likely take somewhere between $100 - 200ms$, which is far too much for a real-time application.

Chapter 7

Limitations and Future Work

Given the results presented in Chapter 6, there are a few relevant questions one could ask:

- What limitations does the current implementation have?
- How can the results be improved upon in the future?

7.1 Current Implementation Limitations

The biggest limitation with the current real-time AO implementation, aside from the rebuilding of the bottom-level acceleration structures, is the limited sample count. It directly leads to the presence of sampling patterns in the final result. If not for this, the real-time AO version would be very close to the offline AO target (see the comparison made in Section 6.1.2). So long as the sampling patterns are present to the degree to which they are now, the implementation is almost "unusable" in any application that is to be placed in the hands of users.

To add to the problem of a low sample count, there is also the fact that the AO pass is being rendered at half resolution, i.e. 960x540. This makes the sampling patterns even more visible.

There is also the issue of performance. To get the current results, the AO pass needs roughly $5ms$ for the *mercedes.brhan* scene. This scene, although it has $\sim 2.45M$ triangles, is fairly simple; all of the geometry is clustered in one area. This makes the intersection of the acceleration structure faster than if there were many smaller clusters of geometry evenly spread throughout the scene. The geometry is also static, both internally and externally. In an application such as a game, this will not be the case. The acceleration structure will require regular updates, which will add to the total time of the AO implementation. As presented, this is likely to require many milliseconds, if not in the order of microseconds. This means that even if the AO calculations themselves were sped up a considerable amount, the implementation would still be bottlenecked by the acceleration structure updates.

Finally, the current implementation struggles to remove sampling artifacts. Other blurs were also tested, they did not yield as good results as the box filter which did a fairly poor job itself:

- Gaussian blur: failed to blur the sampling artifacts enough when using a small kernel. With a large kernel, a lot of detail in the AO was lost.
- Wider box blurs: the large kernels resulted in a loss of detail in the AO.
- Blurs with larger jumps in offsets, e.g. 1, 3, 5 instead of 1, 2, 3: loss of detail in the AO.

It is also worth noting that large kernels result in more texture loop-ups and reduced performance. Although the performance hit was negligible in comparison to the AO render pass, it still adds to the total time, and any time saved on blurring can be reinvested in AO calculations.

7.2 Implementation Improvement and Future Work

7.2.1 Sampling

Given the number of samples that are available for each pixel, one of the most important tasks is how to best sample the hemisphere. In the implementation presented, blue noise is used heavily. There does however exist other types of sequences that can be generated, e.g. *Sobol Sequence*. Doing further investigations into how to best pick the values used for sampling the hemisphere, might prove very valuable. For the sampling method, it is hard to do better than cosine-weighted sampling.

7.2.2 Sampling Pattern Mitigation

In the implementation presented, a simple box filter was used to try and reduce the sampling patterns that were present in the results. Although other filters were also tested, this part of the implementation did not receive a lot of focus. Finding a way to reduce the sampling patterns without much loss in overall detail would be very beneficial. Here various denoising algorithms could be investigated, that are better suited than a blur. If good filtering is achieved, the number of AO samples could also possibly be reduced, resulting in less time spent on the actual AO.

7.2.3 Temporal Integration

In an application were the camera and geometry move around, more care has to be taken when implementing temporal integration. This is explained in greater detail in [68], and would have to be implemented accordingly.

7.2.4 GPU Performance

Finally, although not up to the programmer implementing real-time AO, an increase in GPUs' ray tracing performance would yield better results. This is because more samples could be taken, or the number of samples could remain the same, yielding lower computation time. Also, improving the acceleration structure (re-)building time is *very* crucial for real-time AO to be a feasible option. With time, GPUs will get faster, and so, only the future will tell when real-time AO can become the standard for real-time global illumination approximation in games.

7.3 Conclusion

Given the presented implementation, real-time AO is *not* yet ready for use in real-time applications, especially games. The issue with AO quality due to sampling artifacts, the variable computation time which is viewpoint dependent, and the rebuilding of acceleration structures makes the technique hard to implement into a rendering pipeline with a limited budget, $< 5ms$ for AO.

Appendix

Source Code

All of the code for the path tracing engine is available in a public *GitHub* repository: *https://github.com/MulattoKid/BrhanRenderer*.

All of the code for the real-time AO implementation is available in a public GitHub repository: *https://github.com/MulattoKid/Vulkan_RTX*.

Both code bases have also been included in a *.zip* file when handing in the thesis.

.brhan File Format

The *.brhan* format allows for specifying the camera properties, number of samples per pixel, path depth, which geometric models to load, what transformations to apply to each model, what material a model should have etc. A full list of features can be found in the *README.md* in the GitHub repository for the path tracing engine.

Comparison Images

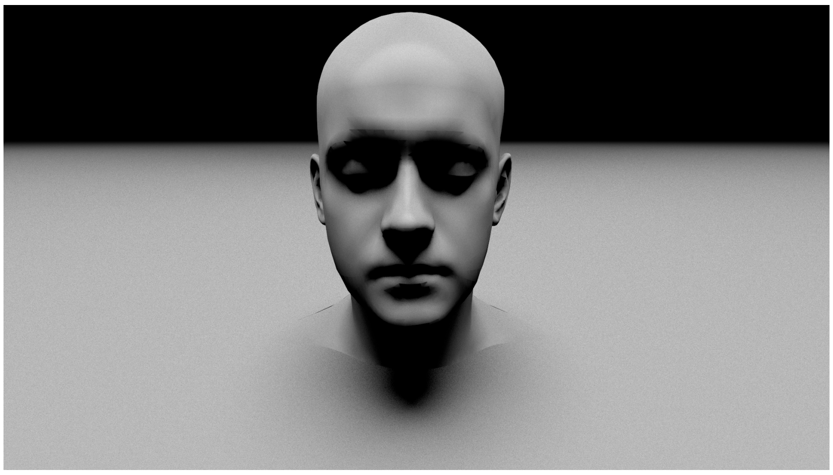


Figure 7.1: Head rendered with 2048 SPP (model via PBRT scenes[36]).

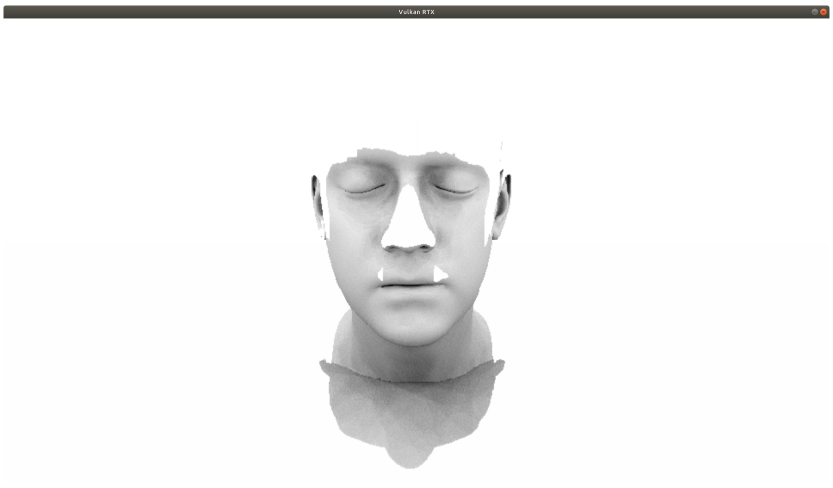


Figure 7.2: Head rendered with real-time AO (model via PBRT scenes[36]).

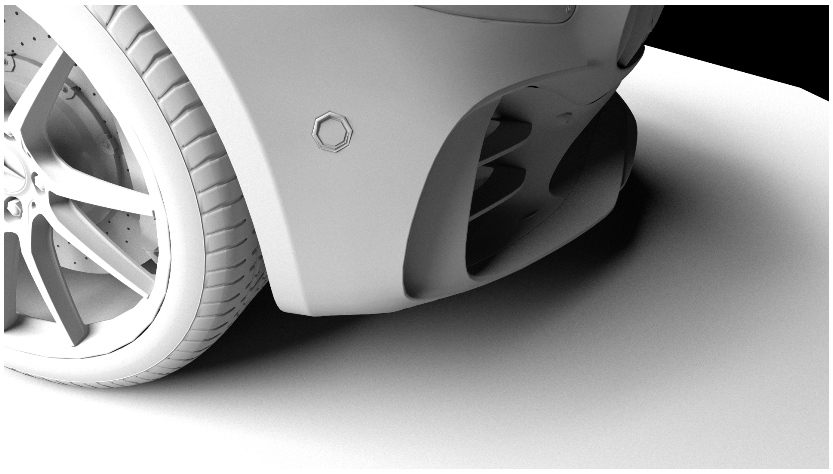


Figure 7.3: Mercedes rendered with 8196 SPP from viewpoint 0 (model courtesy of Thomas F[40]).

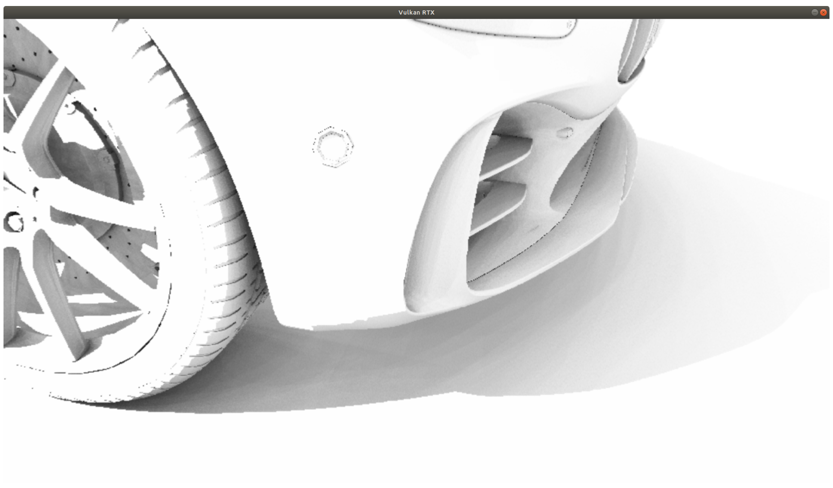


Figure 7.4: Mercedes rendered with real-time AO from viewpoint 0 (model courtesy of Thomas F[40]).

Additional Images

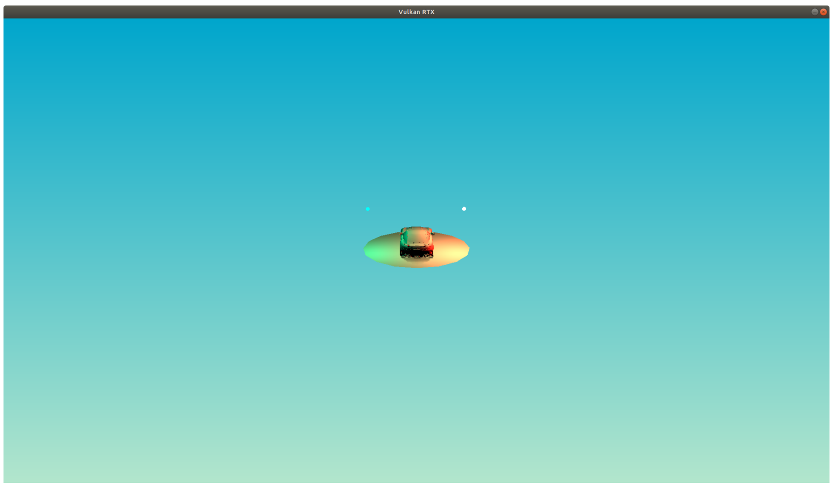


Figure 7.5: Mercedes rendered with real-time AO from far away (model courtesy of Thomas F[40]). Note that color is added for easier viewing.

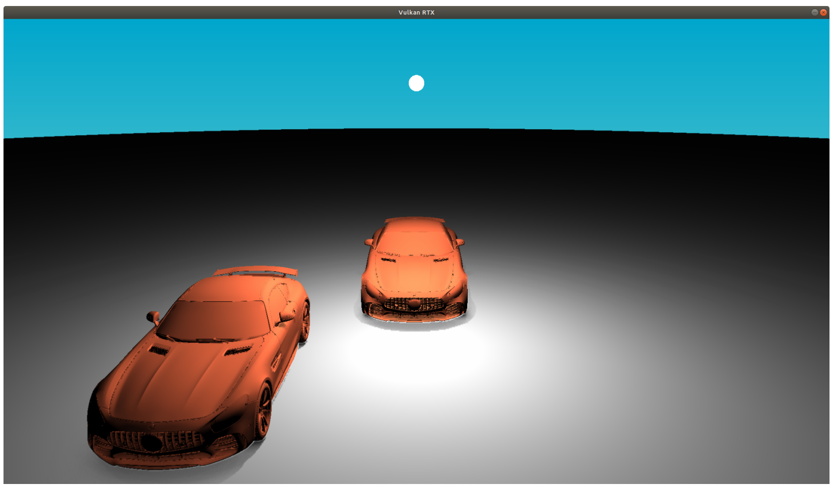


Figure 7.6: Scene with a complexity of 4.9M triangles.

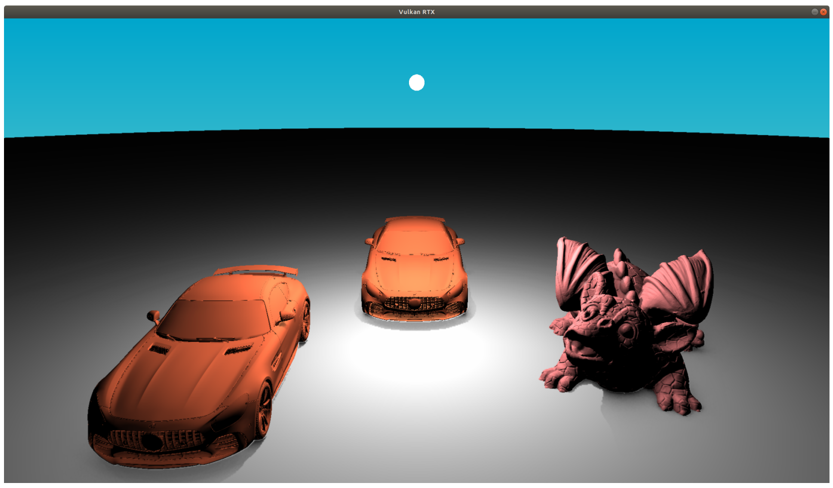


Figure 7.7: Scene with a complexity of 6.7M triangles.

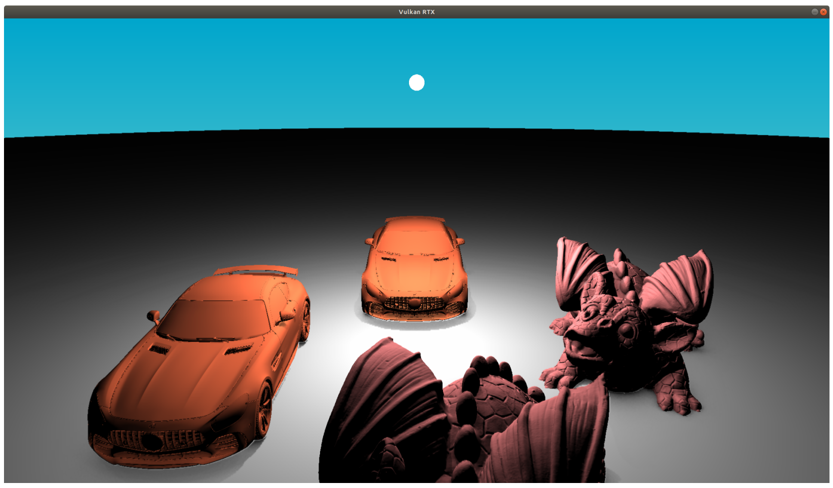


Figure 7.8: Scene with a complexity of 8.6M triangles.

Bibliography

[1] Mike Acton. Data-Oriented Design and C++. *https://youtu.be/rX0ItVEVjHc?t=1359* (accessed 01/29/2019), 2014.

[2] Michael Gallucci. 'Futureworld' - A Look Back at the First Movie With 3D CGI. *http://diffuser.fm/futureworld-movie/* (accessed 01/31/2019), February 2014.

[3] PC Plus. How special effects transformed the movies. *PC Plus*, 280, 2011.

[4] Pixar. *https://www.pixar.com/feature-films/monsters-university/* (accessed 01/29/2019).

[5] DOS games archive. *https://www.dosgamesarchive.com/download/wolfenstein-3d/* (accessed 01/29/2019).

[6] Fabien Sanglard. *GAME ENGINE BLACK BOOK: DOOM*. CreateSpace Independent Publishing Platform, 2018.

[7] Dark Side of Gaming. *https://www.dsogaming.com/screenshot-news/battlefield-5-open-beta-4k-screenshots-gallery/* (accessed 01/29/2019).

[8] Solid Angle. Ambient Occlusion. *https://docs.arnoldrenderer.com/display/A5AFMUG/Ambient+Occlusion* accessed(04.01.2019).

[9] Valve. Screen Space Ambient Occlusion. *https://developer.valvesoftware.com/wiki/Screen_Space_Ambient_Occlusion_(SSAO)* accessed(04.01.2019).

[10] Nvidia. Nvidia HBAO+ Technology. *https://www.geforce.com/hardware/technology/hbao-plus/technology* accessed(04.01.2019).

[11] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically Based Rendering - From Theory to Implementation*. Morgan Kaufmann, 3 edition, December 2016.

[12] Pete Shirley. *Ray Tracing in one Weekend*. Independent Publisher, January 2016.

[13] Unknown Author. Antialiasing and Anisotropic Filtering. *https://www.geforce.com/whats-new/guides/aa-af-guide#2*.

[14] Alan Wolfe. Generating Blue Noise Sample Points with Mitchell's Best Candidate Algorithm. *The blog at the bottom of the sea*, October 2017.

[15] Daniel L. Toth. On Ray Tracing Parametric Surfaces. *SIGGRAPH Comput. Graph.*, 19(3):171–179, July 1985.

[16] Tomas Möller and Ben Trumbore. Fast, Minimum Storage Ray-triangle Intersection. *J. Graph. Tools*, 2(1):21–28, October 1997.

[17] Scott Owen. Ray-Sphere Intersection. *SIGGRAPH Education*, 1999.

[18] Erik Reinhard, Brian Smits, and Charles Hansen. Dynamic Acceleration Structures for Interactive Ray Tracing. In Bernard Péroche and Holly Rushmeier, editors, *Rendering Techniques 2000*, pages 299–306, Vienna, 2000. Springer Vienna.

[19] Bui Tuong Phong. Illumination for Computer Generated Pictures. *Commun. ACM*, 18(6):311–317, June 1975.

[20] R. L. Cook and K. E. Torrance. A Reflectance Model for Computer Graphics. *ACM Trans. Graph.*, 1(1):7–24, January 1982.

[21] James T. Kajiya. The Rendering Equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150, August 1986.

[22] Gianluca Tosini, Ian Ferguson, and Kazuo Tsubota. Effects of blue light on the circadian system and eye physiology. *Molecular Vision*, 22:61–72, January 2016.

[23] Brent Burley. Physically-Based Shading at Disney. *Disney Animaion*, August 2012.

[24] Rod Nave. Index of Refraction. *http://hyperphysics.phy-astr.gsu.edu/hbase/Tables/indrf.html*.

[25] Max Born and Emil Wolf. Chapter 1 - basic properties of the electromagnetic field. In Max Born and Emil Wolf, editors, *Principles of Optics (Sixth Edition)*, pages 1 – 70. Pergamon, sixth edition edition, 1980.

[26] Rod Nave. Fresnel's Equations. *http://hyperphysics.phy-astr.gsu.edu/hbase/phyopt/freseq.html*.

[27] Andrey N. Kolmogorov. *Foundations of the Theory of Probability - Second English Translation.* Chelsea Publishing Company, 1956.

[28] Rory Driscoll. Better Sampling. *http://www.rorydriscoll.com/2009/01/07/better-sampling/* (accessed 0.5.17.2019).

[29] Yining Karl Li. Multiple Importance Sampling. *https://blog.yiningkarlli.com/2015/02/multiple-importance-sampling.html* (accessed 05.21.2019).

[30] Solid Angle. Arnold. *https://appleseedhq.net/* (accessed 02.14.2019).

[31] Matt Pharr, Wenzel Jakob, and Greg Humphreys. pbrt-v3. *https://github.com/mmp/pbrt-v3* (accessed 02.14.2019). Commit: f7653953b2f9cc5d6a53b46acb5ce03317fd3e8b.

[32] Paul Bourke. Object Files (.obj). *http://paulbourke.net/dataformats/obj/* (accessed 01.13.2019).

[33] Syoyo Fujita. tinyobjloader. *https://github.com/syoyo/tinyobjloader*.

[34] Sean Barrett. stb. *https://github.com/nothings/stb*.

[35] Standford University. The Stanford 3D Scanning Repository. *http://graphics.stanford.edu/data/3Dscanrep/* (accessed 05.01.2019).

[36] Matt Pharr, Wenzel Jakob, and Greg Humphreys. Scenes for pbrt-v3. *https://pbrt.org/scenes-v3.html* (accessed 04.09.2019).

[37] Jack Doweck. Inside Intel Core Microarchitecture and Smart Memory Access. *Intel Whitepaper*, 2006.

[38] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, J. Alan Miller, and Michael Upton. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal Q1*, 2002.

[39] Morgan McGuire. McGuire Computer Graphics Archive. *https://casual-effects.com/data/* (accessed 0.50.8.2019).

[40] Thomas F (@TehRenderGuy). Thomas F's Models. *https://www.behance.net/tehrenderguy*.

[41] Khronos Group. OpenGL Overview. *https://www.opengl.org/about/* accessed(03.28.2019).

[42] Khronos Group. Vulkan Overview. *https://www.khronos.org/vulkan/* accessed(03.28.2019).

[43] Microsoft. Direct3D. *https://docs.microsoft.com/en-us/windows/desktop/direct3d* accessed(03.28.2019).

[44] Apple. Metal 2 - Accelerating graphics and much more. *https://developer.apple.com/metal/* accessed(03.28.2019).

[45] Rendering Pipeline Overview. *https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview* (accessed 03.27.2019).

[46] Khronos Group. OpenCL Overview. *https://www.khronos.org/opencl/* accessed(03.28.2019).

[47] Nvidia. About CUDA. *https://developer.nvidia.com/about-cuda* accessed(03.28.2019).

[48] Khronos Group. Opengl Shading Language. *https://www.khronos.org/opengl/wiki/Core_Language_(GLSL)* accessed(03.28.2019).

[49] Max Liani. Renderman XPU: Development Update. *Pixar*, March 2018.

[50] Nvidia. RTX. IT'S ON. *https://www.nvidia.com/en-us/geforce/20-series/* accessed(03.28.2019).

[51] Khronos Group. Vulkan Specification Instances. *https://www.khronos.org/registry/vulkan/specs/1.1-extensions/html/vkspec.html#initialization-instances* (accessed 05.13.2019).

[52] Khronos Group. Vulkan Specification - Physical Devices. *https://www.khronos.org/registry/vulkan/specs/1.1-extensions/html/vkspec.html#devsandqueues-physical-device-enumeration* (accessed 05.13.2019).

[53] Khronos Group. Vulkan Specification - Devices. *https://www.khronos.org/registry/vulkan/specs/1.1-extensions/html/vkspec.html#devsandqueues-devices* (accessed 05.13.2019).

[54] Khronos Group. Vulkan Specification - Valiadation Layers. *https://www.khronos.org/registry/vulkan/specs/1.1-extensions/html/vkspec.html#fundamentals-errors* (accessed 05.13.2019).

[55] Khronos Group. Vulkan Specification - Extensions. *https://www.khronos.org/registry/vulkan/specs/1.1-extensions/html/vkspec.html#extendingvulkan-extensions* (accessed 05.13.2019).

[56] Khronos Group. Vulkan Specification - Features. *https://www.khronos.org/registry/vulkan/specs/1.1-extensions/html/vkspec.html#features* (accessed 05.13.2019).

[57] Khronos Group. Vulkan Specification - Queues. *https://www.khronos.org/registry/vulkan/specs/1.1-extensions/html/vkspec.html#devsandqueues-queues* (accessed 05.13.2019).

[58] Khronos Group. Vulkan Specification - Command Buffers. *https://www.khronos.org/registry/vulkan/specs/1.1-extensions/html/vkspec.html#commandbuffers* (accessed 05.17.2019).

[59] Khronos Group. Vulkan Specification - Image Layout Transitions. *https://www.khronos.org/registry/vulkan/specs/1.1-extensions/html/vkspec.html#synchronization-image-layout-transitions* (accessed 05.13.2019).

[60] Khronos Group. Vulkan Specification - Images. *https://www.khronos.org/registry/vulkan/specs/1.1-extensions/html/vkspec.html#resources-images* (accessed 05.13.2019).

[61] Khronos Group. SPIR-V Overview. *https://www.khronos.org/spir/* accessed(04.01.2019).

[62] Khronos Group. SPIR-V Overview. *https://github.com/KhronosGroup/SPIRV-Tools* accessed(04.01.2019).

[63] LunarG. SPIR-V Toolchain. *https://vulkan.lunarg.com/doc/sdk/1.1.101.0/windows/spirv_toolchain.html* accessed(04.01.2019).

[64] Khronos Group. Vk_nv_ray_tracing. *https://www.khronos.org/registry/vulkan/specs/1.1-extensions/html/vkspec.html#VK_NV_ray_tracing* accessed(03.29.2019).

[65] Khronos Group. Vulkan Specification - Descriptor Types. *https://www.khronos.org/registry/vulkan/specs/1.1-extensions/html/vkspec.html#descriptorsets-types* (accessed 05.17.2019).

[66] Jorge Jimenez, Xian-Chun Wu, Angelo Pesce, and Adrian Jarabo. Practical Realtime Strategies for Accurate Indirect Occlusion. *SIGGRAPH2016*, July 2016.

[67] Louis Bavoil, Miguel Sainz, and Rouslan Dimitrov. Image-Space Horizon-Based Ambient Occlusion. *ShaderX7 Advanced Rendering Techniques*, July 2008.

[68] Lasse Jon Fuglsang Pedersen. Temporal Reprojection Anti-Aliasing in INSIDE. *Game Developers Conference*, March 2016.

[69] Khronos Group. Vulkan - A Specification. *https://www.khronos.org/registry/vulkan/specs/1.1-extensions/html/vkspec.html#* accessed(03.29.2019).

[70] Jeff Bolz, Kerch Holt, Kenneth Benzie, Neil Henning, Neil Hickey, Daniel Koch, Timothy Lottes, and David Neto. GL_KHR_vulkan_glsl. *Khronos Group*, July 2018.

[71] Arthur Ardeshir Goshtasby. *Image Registration - Principles, Tools and Methods*. Springer, London, 2012.

[72] Nvidia Cooperation. Adaptive VSync. *https://www.geforce.com/hardware/technology/adaptive-vsync/technology*.

[73] Steven W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, San Diego, CA, USA, 1997.