FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Real-Time Ray Tracing

in Quake III

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Visual Computing

eingereicht von

Lukas Lipp, BSc
Matrikelnummer 01425235

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Dipl.-Ing. Christian Freude

Wien, 29. Oktober 2020

Lukas Lipp	Michael Wimmer

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Real-Time Ray Tracing

in Quake III

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Visual Computing

by

Lukas Lipp, BSc
Registration Number 01425235

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Dipl.-Ing. Christian Freude

Vienna, 29th October, 2020

Lukas Lipp

Michael Wimmer

Erklärung zur Verfassung der Arbeit

Lukas Lipp, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 29. Oktober 2020

Lukas Lipp

Danksagung

Ein großes Dankeschön geht an meine Familie ohne deren Unterstützung ein solches Studium nicht so einfach möglich gewesen wäre. Ich möchte auch all denjenigen danken die mich in diesen Jahren begleitet haben und mir mit Rat und Tat zur Seite gestanden sind.

Ein großer Dank geht an das Dekanat Informatik (Förderungsstipendium), welches mir einen Großteil der benötigten Hardware, für diese Diplomarbeit, finanziert hat.

Acknowledgements

A big thanks to my family for every bit of support they offered me over these years. Without their help, all of this would not be possible. Shout outs to my friends and the people who helped me during all these years.

Another big thank you goes to the Office of the Dean of TU Wien Informatics, for the financial support for the necessary hardware for this thesis.

Kurzfassung

Diese Arbeit behandelt die Erweiterung der bekannten Quake III Spiel-Engine mittels Ray Tracing in Echtzeit. Hierbei wird untersucht, inwieweit Ray Tracing mithilfe der neuesten Grafikkarten-Generation von NVIDIA implementiert werden kann. Ermöglicht wird die Berechnung von Ray-Tracing Effekten in Echtzeit durch eine dedizierte Hardware-Unterstützung, welche Ray Tracing Algorithmen beschleunigen soll. Zugriff zu diesen Features wird durch eine neue API ermöglicht. Zudem werden Strategien besprochen wie Offline Ray Tracing Algorithmen in einer Real Time (Online) Umgebung effizient angewendet werden können.

Hierfür wird zuerst Quake III mit einem Vulkan Backend ausgestattet, um in weiterer Folge, Distributed Ray Tracing zu implementieren, welches dann für das Darstellen der gesamten Spielewelt zuständig ist. Lediglich die user interface Elemente werden vom Rasterizer übernommen.

Das Ziel ist es zu analysieren, inwieweit es möglich ist, mittels RTX Ray Tracing in eine Spiele Engine einzubauen und welche Abstriche an Qualität und Performance man in Betracht ziehen muss unter der Voraussetzung 30/60 Bilder pro Sekunde zu erreichen. Zudem werden Strategien besprochen, welche Qualität, Performance oder beides verbessern.

Abstract

This work discusses the extension of the popular Quake III game engine using real-time raytracing. It investigates how ray tracing can be implemented using the most recent graphics card generation by NVIDIA, which offers dedicated hardware support and acceleration via an new API. In addition, strategies will be discussed about how offline ray-tracing algorithms can be transformed to an online real-time context.

In order to implement ray tracing, Quake III needs to be extended with a Vulkan backend. Next, distributed ray tracing is implemented and is used to render the whole game world except for the user interface (UI) elements. The UI will be handled by the rasterizer.

The performance and efficiency of ray tracing in a game engine using the RTX hardware features is analyzed and discussed. The focus lies on how quality and performance relate to each other, and how far ray tracing can be pushed with still acceptable frame rate of around 30/60 frames per second. Furthermore, implementation strategies that improve the quality, performance or both will be discussed.

Contents

Kurzfassung **xi**

Abstract **xiii**

Contents **xv**

1 Introduction **1**
1.1 Motivation 1
1.2 Problem Statement 2
1.3 Contribution 4
1.4 Structure of this Work 4

2 Background **7**
2.1 Ray Tracing 7
2.2 Path Tracing 9
2.3 Monte Carlo Method 11
2.4 Illumination 14
2.5 Denoiser 18
2.6 RTX 18

3 Related Work **21**
3.1 Ray Tracing and Path Tracing 21
3.2 RTX 24

4 Architecture **27**
4.1 Offline/Online Ray Tracing Differences 27
4.2 Distributed Ray Tracing 29
4.3 Lights Culling 29
4.4 Denoiser 31
4.5 Temporal Anti-Aliasing 31

5 Quake III modifications **33**
5.1 Refactoring the original Quake III Code 34
5.2 Implementing Vulkan in addition to OpenGL 36

5.3 Other necessary Quake III modification 37
5.4 Distributed Ray-Tracing Integration 38

6 Distributed Ray Tracing Implementation **41**
6.1 Acceleration Structures 42
6.2 Ray-Tracing Pipeline 43
6.3 Primary Ray Stage 44
6.4 Transparency 45
6.5 Secondary Ray Stage 47
6.6 Direct Illumination Stage 47
6.7 Accessing Object Data in Shader 48
6.8 Combination of Multiple Textures 49
6.9 Lights 50
6.10 Denoiser 51
6.11 Composition 52
6.12 Temporal Anti-Aliasing 52

7 Evaluation **53**
7.1 Hardware 53
7.2 Visual Results 54
7.3 Performance 57
7.4 Discussion of Noise Origins 62
7.5 Problems and Limitations 62

8 Conclusion and Future Work **65**

List of Figures **67**

List of Tables **71**

List of Algorithms **73**

Glossary **75**

Acronyms **77**

Bibliography **79**

CHAPTER 1

Introduction

1.1 Motivation

Computer graphics, which is a sub-field of computer science, is focused on generating or manipulating images with the aid of a computer. It is a crucial part in digital photography, film, video games, cell phones and more. Since the goal of computer graphics is to efficiently produce results based on defined requirements, the complexity of the used algorithms also increases further. Quality factors of the image can be resolution, low number of artifacts, amount of details or complexity of the content. Since images and computer screens can have millions of pixels and standard central processing units (CPUs) can only handle one pixel at a time (or more, e.g., 8 pixels if multithreading is available), it can be difficult to execute complex calculations for all the pixels in a given time frame. To solve this problem, graphics processing units (GPUs) were developed, which are designed to execute instructions in parallel. The advantage is that these operations are mostly identical and are therefore optimal for an execution in parallel (single instruction, multiple data (SIMD)). This enables the calculation of multiple pixels at once, and therefore decreases the required time for the same calculation done on the CPU. Since a huge motivation of computer graphics is to achieve photo realistic results, there is active research regarding improving the algorithm and hardware in order to reach this goal. There are many different use cases where photo-realism is desired. One example would be for architects or designers since they want to create virtual environments that represent the final product as closely as possible. Also, the movie industry makes use of computer graphic to produce movies completely from imagination or adding 3D models and effects to existing scenes captured from the real world. In both cases it is desired to close the gap between fake and reality. The use case that is most related to this thesis are computer games. Their goal is to immerse the player in a virtual world, which becomes easier the more realistic the game's visuals look. Especial with Virtual Reality (VR) headsets emerging, photorealism is becoming an important topic in order to provide an immersive experience for the player.

There are various techniques that can be utilized to render a scene in the most realistic way possible. One key component for achieving a realistic image is the correct shading of the objects based on all the light sources. There are many approaches, which differ in the accuracy of their result to achieve this. The accuracy is often in relation to the necessary computational cost. Games, for instance, use fast but less accurate algorithms, while professional applications require exact algorithms that require much more computational effort and are thus slower. Therefore, offline implementations have fewer time constraints and produce better visual results, while online implementations have real-time constraints but produce a lower quality result since they need to make use of simple approximations. One approach to achieve results with a high accuracy is called ray tracing and is mostly used to calculate illumination, reflections and shadows. The idea behind it is to shoot a ray from the eye through a scene and test it for intersection with the scene. Through this technique, a physically based simulation of the light can be calculated.

Ray tracing was barely used in computer games because of the real-time constraints, limited computing power of most machines and the missing hardware acceleration through graphics cards. With the introduction of programmable shaders and the support for dynamic loops in shaders, which were introduced in Shader Model 3.0, the implementation of ray tracing became feasible. In 2018 NVIDIA released their new RTX GeForce graphics card generation [nvg], which are the first consumer-grade GPUs with dedicated ray-tracing cores. These processors are optimized for ray tracing and should increase the performance of ray-tracing algorithms. NVIDIA also developed an interface which should unify their usage.

This new technology gives us the opportunity to accelerate offline ray-tracing approaches to make them usable in an online context. It is now the question to which extent this new hardware support allows the use of ray tracing to calculate physically correct lighting in an established game engine. Since the source code of Quake III is open source, we decided to use Quake III for this purpose and extend it with ray-tracing effects for illumination, shadows and reflections/refractions.

1.2 Problem Statement

The original implementation from Quake III is from 1999 and uses the OpenGL API for rendering, which currently does not support RTX (also, the implemented OpenGL version is version 1 and thus very old). Therefore, the code needs to be refactored and a renderer based on the modern Vulkan API [vul] must be implemented.

For the ray-tracing implementation we opted for distributed ray tracing instead of path tracing to keep the already high implementation overhead of this thesis manageable. Distributed ray tracing skips some aspects compared to path tracing but introduces the same problems we would face with path tracing. Since both depend on the use of random samples, noticeable noise will be contained in the produced images if the amount of used samples per pixel (SPP) is too low. Even with hardware acceleration from dedicated processors provided through NVIDIA's new RTX technology we face the problem that a naive ray-tracing implementation is still too computationally expensive. Therefore, the implementation must be improved through various optimizations in order to produce a clean output image every 16.6 ms (equals 60 frames per second (FPS)).

Old games, as it is the case with Quake III, often produce its lighting with the help of light maps. These precalculated textures are generated with more lights than computers at that time could handle during the game's execution. At run-time, the light maps can then be used to produce the desired shading quality of the scene with just a minimal performance impact. This means Quake III provides no predefined lights, and in order to produce a comparable shading of the scene, numerous lights must be created, which further increases the workload on the GPU.

Correctly blending transparent textures is also a nontrivial task when using ray/path tracing. In a rasterizer-focused engine, textures are designed to blend from back to front. A ray/path tracer does this from front to back in most cases.

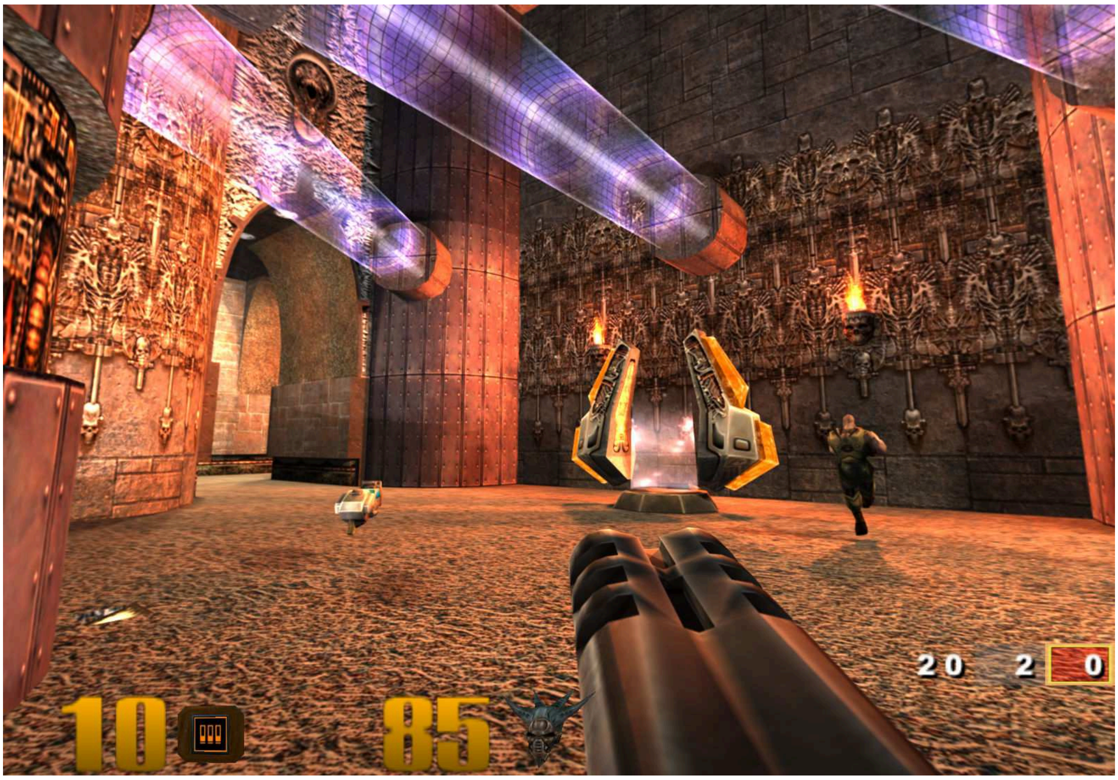


Figure 1.1: Ray-traced Quake III.

1.3 Contribution

We will discuss important aspects of modifying ray-tracing algorithms for real-time applications and explore the possibilities of them in a commercial game engine.

The effects we use for distributed ray tracing are soft shadows and anti-aliasing. We turned off depth of field throughout our analysis because its visual contribution was unsatisfying and did not go well with the feeling of the game. Since the amount of SPP we can calculate each frame in a real-time application is limited, anti-aliasing and especially soft shadows add noticeable noise to the ray-traced output. The more SPP we use, the less noise is produced by these effects, in the optimal case we would prefer the use of only one SPP. The second type of noise we experience is produced by subsampling of the lights. This noise compared to the noise produced by the distributed ray-tracing effects is from a discrete distribution instead of a continuous distribution. Nevertheless, a denoiser is effective against both types.

In order to make real-time ray tracing feasible, we start with the implementation of a denoiser. Schied et al. [SPD18] developed a denoiser for real-time path tracing which can produce a clean image with the use of just one SPP. It is called adaptive spatiotemporal variance-guided filtering (A-SVGF), is state-of-the-art and uses the knowledge from multiple previous frames to clean up the current image. Since the denoiser's output quality depends on the degree of noise in the input image, we must reduce the noise as much as possible. Therefore, we introduce a light-culling strategy which should help to save performance, and thus to improve the visual quality using the potentially visible set (PVS) provided by Quake III. Originally, Quake III uses the PVS in combination with the rasterizer for geometry culling. We can repurpose it for our light-culling strategy and cull lights that have no contribution on a per-pixel basis. This strategy should not be limited to Quake III's PVS but also be applicable to any PVS. With the help of light culling, it is now possible to produce results comparable in terms of visual quality to the results produced by the denoiser while still being able to maintain an acceptable frame rate. An additional strategy we employ is called light subsampling, which improves the performance but results in a high amount of noise, since the number of contributing lights is too low.

When we combine the denoiser, light culling and light subsampling, we can produce a high-quality shading of the scene with the lowest impact on performance out of all combinations. Based on these observations, we will discuss how quality and performance relate to each other.

1.4 Structure of this Work

In the next chapter, we will discuss the background knowledge for the fundamental parts of this thesis. Ray tracing, path tracing and other important technical terms related to these technologies will be discussed.

In Chapter 3, we look at research literature about ray tracing in an offline/online context and the RTX technology. This is necessary to understand the current state of research for these topics and to build a solid knowledge foundation for the various aspects of this thesis.

In Chapter 4, we will discuss the overall architecture of the implementation. We start with a discussion on how to transition from an offline to an online ray-tracer, and what changes are necessary to achieve acceptable performance of 30 or 60 FPS. Thereafter, the structure for the distributed ray tracing backend will be explained from a high-level point of view. Here, different parts that produce the final output, but also architectural strategies (e.g., our light culling approach) used in the implementation are discussed.

Chapter 5 is used to point out all the changes that were required in order to make the original Quake III source code compatible with modern computers and further enable the implementation of a ray-tracer. This also includes the implementation of a Vulkan backend.

In Chapter 6, the implementation of the distributed ray-tracer will be explained in more detail. In contrast to Chapter 4, where we talked about the high-level architectural view, this time we will take a look at the low-level implementation. This chapter will not contain Quake III related code and therefore can be seen as a general implementation of a real-time distributed ray-tracer.

Chapters 7 and 8 are used to evaluate the results by comparing and analyzing different aspects of the final implementation with focus on quality and performance. In the last chapter we will draw a conclusion based on this gained knowledge about the possibilities of real-time ray tracing in commercial game engines.

CHAPTER 2

Background

There are many different variations of ray and path tracing, which differ in visual quality, computational efficiency and accuracy of the models used for simulating light. Since we cannot focus on all of them, we will only discuss variations that are relevant for this thesis. The goal of this chapter is to discuss the fundamental concepts for ray tracing and path tracing and therefore only the simplest forms of these techniques are discussed. More advanced solutions to the discussed problems below are included in the next chapter, which deals with recent research publications in the field of ray/path tracing.

The fundamental idea of ray and path tracing is to shoot a ray through a scene of objects and check if the ray hits one of those objects. There are some difficulties we need to overcome in order to make this feasible on a computer. One problem is finding a way to organize all objects of the scene in a way that it is possible to check them for collision as fast as possible. One approach is to use a bounding volume hierarchy (BVH), which is a tree like data structure that organizes the objects based on their bounding volume. This has the advantage that we can eliminate all branches which are not relevant for a given ray and speed up the calculation of the intersection. Another problem especially for real-time ray tracing is how quickly the BVH can be updated for moving and deforming objects in the scene. Based on solutions for these problems ray and path-tracing algorithms can be realized. There is a lot of research happening which the focus on improving these basic ray and path-tracing implementations. The goal mostly lies on improving quality and performance. In the following sections we will talk about some of these basic approaches.

2.1 Ray Tracing

2.1.1 Whitted Ray Tracing

Whitted ray tracing [Whi80] is the simplest form of ray tracing and has the advantage that it is deterministic, in contrast to path tracing, which uses stochastic processes that manifest themselves as noise in the images. If the surface that is hit is reflective, the ray gets reflected by the normal of the surface at the intersection point. For transparent surfaces like water or glass the ray splits up into an reflection and refraction ray. The Fresnel equation is used to calculate the directions of the two new rays starting from the point of intersection. In case of a opaque surface the algorithm stops and uses a illumination model such as the Phong illumination model to calculate the shading of the object at the intersection point. So called shadow rays are used to check if a given point is in shadow. This is done by shooting rays from a given point in the direction of all light sources. See Figure 2.1 for a graphical representation of one example ray. Pseudocode for Whitted ray tracing can be found in Algorithm 2.1.

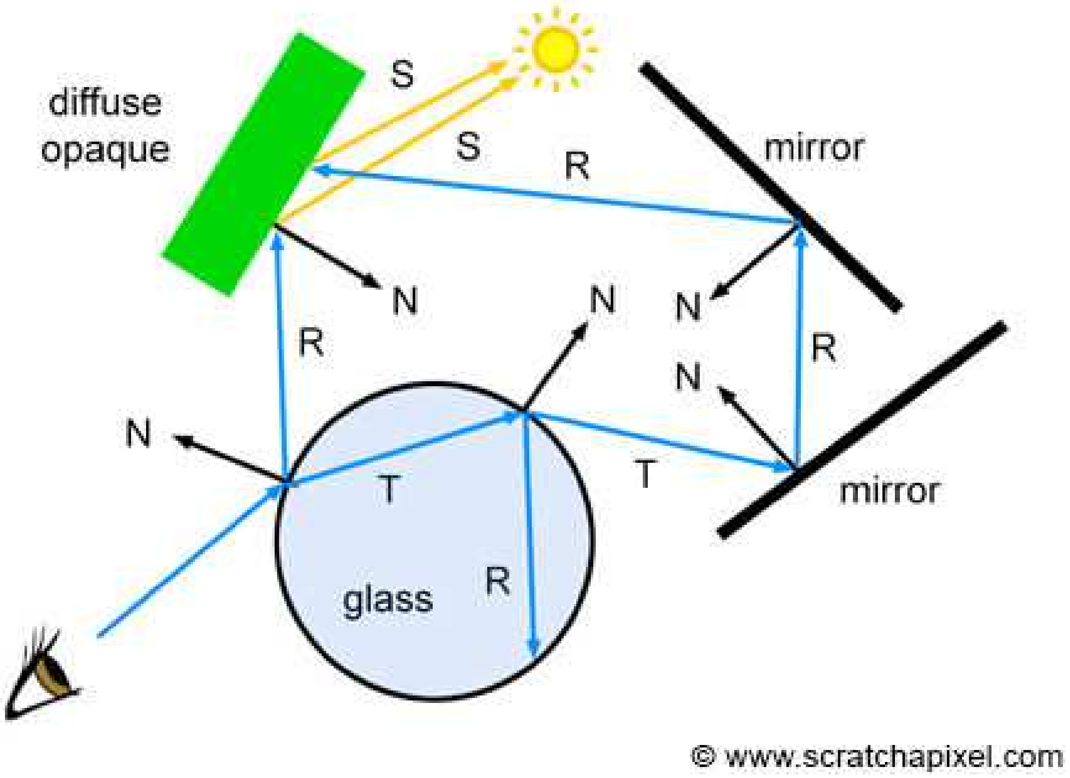


Figure 2.1: Illustration of a ray path in case of a Whitted ray-tracer. The ray gets reflected and refracted multiple times till it hits a opaque surface and stops. R = Reflection Ray; T = Refraction or Transmission Ray; S = Shadow Ray; N = Normal; [wrt]

2.1.2 Distributed Ray Tracing

Another ray-tracing algorithm is called distributed ray tracing. It is based on Whitted ray tracing and extends it with effects for soft shadows, anti-aliasing, motion blur, depth of field and glossy reflections [CPC84]. In Table 2.1 the effects are explained in more detail. Since these effects depend on Monte Carlo (MC) sampling (explained in a later section), the end result can contain noise, which needs to be filtered out or reduced by using more samples per pixel (SPP), which means that for each pixel multiple rays with different random values are traced and their contributions are averaged. With the help of the MC method, the result gets more precise with each sample and converges against the exact solution (ground truth).

soft shadows	Instead of only using a fixed position from the surface of the light to calculate the shadow, we select a random position.
anti-aliasing	Normally the ray is shot through the pixel center. When we randomly select the position inside the pixel, we can reduce aliasing in the resulting image.
motion blur	Blurs the vision depending on the movement and the movement speed of the camera. This is done by shooting multiple rays, with each ray intersecting the scene at a different random time. All the results are then averaged.
depth of field	Blur object by their distance to the camera. The distance at which objects are sharp/blurred depends on the set aperture and focal length. These two values define a focal point through which the ray has to travel. Before shooting the ray, a random offset is added to the start point of the ray.
glossy reflections	By randomly changing the direction of the reflection ray when a reflective surface was hit, a blurred reflection image can be calculated.

Table 2.1: Distributed ray tracing effects. Note that all these effect introduce noise to the image. In order to reduce/remove this noise more than one SPP is required.

2.2 Path Tracing

In order to get one step closer to modeling the light distribution of the real world, one important factor is still missing from ray tracing. In the real world, not only the direct light contributes to the shading of a surface but also the indirect light. Since every object that receives light also emits some light from its hemisphere to its surrounding, this is an endless chain of dependencies. In Equation 2.1 a formal definition of the light distribution called the rendering equation can be seen. The formula contains an integral over the hemisphere, which again depends on the lighting of all the points visible in this hemisphere. Since such recursive integrals are too hard or even infeasible to calculate conventionally or analytically, MC integration is used, where we basically choose one random bounce direction from the hemisphere and limit the bounce depth by a fixed value. There is a more advanced technique for setting the bounce depth, which is called Russian Roulette Path Termination. After each bounce a random decision is made, which depends on the throughput (contribution) of the ray, if the ray should be terminated or continued. This is connected to an important characteristic the indirect lighting has.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek, WIEN Your knowledge hub

Algorithm 2.1: Whitted Ray Tracing Pseudocode

```
Function trace(ray, depth):
    if depth >= MaxDept then
        return black
    end
    checkIntersection(ray, scene)
    if ray hit nothing then
        return black
    else
        if material == opaque then
            shootShadowRays()
            return shading
        else if material == mirror then
            reflectRay()
            return trace(reflectedRay, depth++)
        else if material == transparent then
            reflectRay()
            refractRay()
            return blend(trace(reflectedRay, depth++), trace(refractedRay,
             depth++))
    end

Function main():
    for each pixel do
        initRay()
        trace(ray, 0)
    end
```

With every bounce, the contribution to a pixel decreases, while the calculation complexity increases. Ray tracing with the addition of these (indirect) bounces is called path tracing and is used for global illumination (GI). Since path tracing needs a high amount of random samples, the end result is in comparison to ray tracing, prone to noise when the SPP is low. This characteristic makes path tracing unattractive for heavily time dependent use-cases. There are strategies which apply simplifications or modifications to the path-tracing algorithm in order to get results faster, but this comes along with the loss of accuracy. As a result, the image will not converge exactly against the ground truth and is thus called biased.[Kaj86]

$$
\begin{aligned}
&L_o(x,x') = g(x,x')\Big[L_e(x,x') + \int_\Omega f(x,x'',x''')L_i(x',x'')dx''\Big] \\
&where: \\
&g(x,x') \quad \text{Geometry term} \\
&L_o(x,x') \quad \text{Light received by camera} \\
&L_e(x,x') \quad \text{Light emitted by surface} \\
&L_i(x',x'') \quad \text{Light received from hemisphere} \\
&\int_\Omega ...\ dx'' \quad \text{Integral over hemisphere} \\
&f(x,x'',x''') \quad \text{Bidirectional reflectance distribution function (BRDF)}
\end{aligned}
\tag{2.1}
$$

2.3 Monte Carlo Method

In order to realise distributed ray tracing and path tracing, we need to make use of MC integration. This technique is used to solve definite integrals, which cannot be solved analytically because they are too complex. As stated in the previous section, the rendering equation cannot be calculated conventionally or analytically, therefore it is approximated by solving the integral through randomly sampling within the integral bounds. All results are weighted by the probability of each sample and summed up. Based on the properties of MC integration, the result converges closer to the ground truth with each additional sample. This can be used to calculate a single, simple integral like $\int_a^b sin(x)dx$, or multiple larger integrals, like it is the case with the Rendering Equation (Equation 2.1). This can easily be shown visually (see Figure 2.2), by drawing a function inside a box of known size and test the function for multiple random samples that are within this box. If the sample is below the function, it is a hit, otherwise a miss. The ratio of the total number of samples used, compared to how many of this samples were hits, divided by the domain area contains our approximation of the functions integral. In the following subsections we will discuss the MC estimator in more detail.

2.3.1 Estimator

Let us take a look at the MC estimator from a formal point of view. In order to integrate a function $f(x)$ within the continuous domain Ω (Equation 2.2), we must calculate its estimator (Equation 2.6).

$$I = \int_\Omega f(x)dx \tag{2.2}$$

In order for this to work, we have to make use of the cumulative distribution function (CDF) and the probability density function (PDF). The CDF describes the probability that a random variable X takes on a value being less or equal than a threshold x (Equation

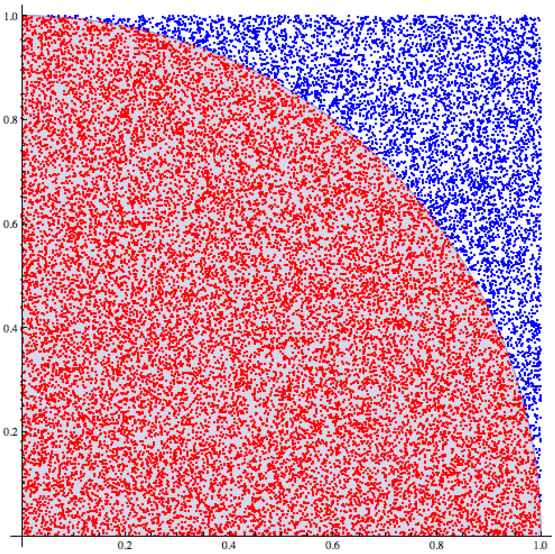


Figure 2.2: Approximation of π through the MC method with 27000 sample points. We are calculating the ratio between the area below the curve and the 1×1 square. This can be expressed as $\frac{\frac{\pi}{4}r^2}{r^2}$. Since we are only considering a fourth of a circle, the result is $\frac{\pi}{4}$ and has to be multiplied by 4 to get the approximation of π.

2.3). The PDF is the derivative of the CDF and describes how likely it is that a random variable X takes on a value being equal to x (Equation 2.4).

$$CDF_X(x) = P(X \leq x) \tag{2.3}$$

$$PDF_X(x) = \frac{d}{dx} CDF_X(x) \tag{2.4}$$

This enables the calculation of the probability that a random variable X takes on a value that is within the bounds of a interval $[a, b]$. This can also be interpreted as the weighted combination of all PDF's across a given interval (Equation 2.5). If we integrate over the full extend of the distributions domain, the result is always 1.

$$P(a \leq x \leq b) = \int_a^b PDF_X(x)dx \tag{2.5}$$

The estimator and its variance for function $f(x)$ can then be defined as shown in Equation 2.6 and 2.7.

$$E\left[f(x)\right] = \int_{\Omega} f(x) PDF_X(x)dx \tag{2.6}$$

$$\sigma^2\left[f(x)\right] = E\left[(f(x) - E[f(x)])^2\right] \tag{2.7}$$

Furthermore, based on these definitions, we can show that for any constant c the following statements about the estimator and variance are true (see Equation 2.8 and 2.9).

$$E[cf(x)] = cE(f(x)) \tag{2.8}$$

$$\sigma^2\left[cf(x)\right] = c\sigma^2\left[f(x)\right] \tag{2.9}$$

This means the expected value of the sum of multiple random variables is the sum of their individual expected values (Equation 2.10). With these properties we can define a simpler expression for the variance (Equation 2.11).

$$E\left[\sum_i f(x_i)\right] = \sum_i E\left[f(x_i)\right] \tag{2.10}$$

$$\sigma^2\left[f(x)\right] = E\left[f(x)^2\right] - E\left[f(x)\right]^2 \tag{2.11}$$

We can use this knowledge to integrate the one dimensional function $f(x)$ from a to b with N random samples $x_i \in [a, b]$ and a PDF of $\frac{1}{(b-a)}$. Equation 2.12 shows the MC estimator for computing I from Equation 2.2.

$$E\left[f(x)\right] = (b-a)\frac{1}{N}\sum_{i=0}^{N} f(x_i) \tag{2.12}$$

The more samples we use, the more the variance decreases, and the estimate converges towards the ground truth. The quality of this process depends on the random samples selected, which we will discuss next. For ray/path tracing there is an additional improvement that is often used called next even estimation (NEE). We will cover NEE in the Illumination Section.

2.3.2 Random Values

One important part of MC Integration, which can optimize how fast the result converges against the ground truth, are the random values that are used. The random values that are calculated from microscopic phenomena are called true random number. True random numbers can but, in most cases are not well distributed (non uniformly). Since it is beneficial for MC to use uniformly distributed random numbers, we cannot use true random numbers. The second option is to use random numbers generated by an algorithm. These are called pseudo-random numbers, which are predictable and, in most cases, non uniformly distributed. When multiple such non uniformly distributed random values are shown as a grey scale image, we call it white noise (shown in Figure 2.4). There exist algorithms, which can produce uniformly distributed random numbers to a certain degree. These sequences are called low discrepancy sequences. Discrepancy describes how uniformly distributed a sequence is. A discrepancy of 0 means that a sequence is completely equidistributed, while a high discrepancy means the values are on the contrary not uniformly distributed. This is visualized in Figure 2.3 for the often used Halton and Sobol sequence.

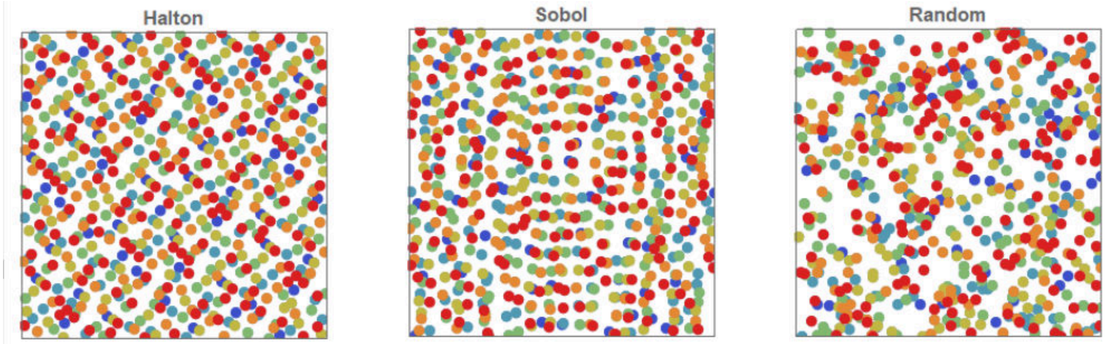


Figure 2.3: 620 random 2D points calculated with the use of the Halton and Sobal sequence, in comparison to an ordinary random generator. [quae]

These low-discrepancy sequence generators differ in the degree of uniformity they can produce, together with the computational effort these calculations require. Especially in a real-time use case, calculating these uniformly distributed random numbers on the fly is too taxing for the computer, therefore they can be precomputed and saved in the form of a texture. Such a random number sequence is often called blue noise when visualized (seen Figure 2.4). In other words this means that, similar samples contained in the texture are as far spread out as possible over a given domain. This can be visualized by applying a blur filter (like Gaussian blur) on a blue noise texture. Figure 2.4 shows this in comparison to a white noise texture. Since the white noise values are not evenly distributed across the texture, the blurred image contains clearly visible high frequency noise. On the other hand, the blurred blue noise texture contains nearly no noise, except for the edges. As previously mentioned, the use of low-discrepancy random values can decrease the time the MC method requires to converge. This also means the resulting data has less variance and in the case of ray/path-tracing the resulting image contains less noise. This is especially beneficial when a denoiser is used, since the algorithm can work with data that contains more valuable information (we will talk about the denoiser in the next sections).

2.4 Illumination

Illumination produced by ray/path-tracing algorithms is based on the light transport theory. This means, energy (light) is transferred by a ray between points in the scene/space. The materials of the surfaces that a single ray hits during the traversal of the scene can be described as an arrangement of letters as defined by Paul Heckbert [Hec90]. This is often

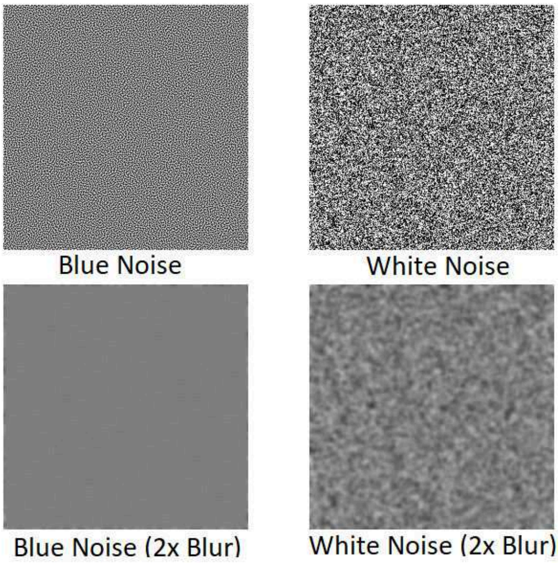


Figure 2.4: White noise vs blue noise texture when a blur is applied. The blue noise has a uniform blur since its values are more uniformly distributed than the white noise values.

called Heckbert's notation. These letters describe a complete light path traveling through the scene. In other words, it describes the order of surfaces the ray hits on its way from the light to the eye. Heckbert's notation uses the following letters to label a light path. L = Light; D = Diffuse; S = Specular; E = Eye; In the direct and indirect light subsection we will talk about some example light paths using this notation. It is important to mention that theses notations start from the light, while most implementations start from the eye. In the following we will also talk about aspects we have to consider in order to calculate the illumination correctly with the MC method.

2.4.1 Direct Light

When a object is hit, the first bounce off its surface in the direction of the eye is called direct light. This means the shading of a point is directly influenced by a light source. These light paths have the notations of LE, LSE or LDE. This means the ray from the light hits an object and after that the ray travels directly to the eye. The noise produced by the direct light, when stochastically sampled, exhibits higher frequencies.

2.4.2 Indirect Light

The indirect lighting is described by the light that bounces of an object and shades all its surrounding surfaces. This can be defined as LDDE in Heckbert's notation, but there are many more variations like for instance LDDDDDE (5 diffuse bounces). Since the contribution of the light decreases with each bounce, it can, depending on the application, be sufficient to have a small amount of bounces. Compared to direct light, the noise of the indirect light is low frequent when stochastically sampled.

Since distributed ray tracing does not calculate the indirect illumination, we do not utilize hemisphere sampling in our implementation, but we included it in this section for completeness.

2.4.3 Hemisphere Sampling

Once a surface is hit from a ray coming from the eye or another surface, we have to consider the incoming light from the whole hemisphere for that hit point. A visual representation can be seen in Figure 2.5. In the rendering equation, this integral is described as the integral over Ω. Ω is 2π which is the surface area of the hemisphere (2π steradians or solid angels). Since we cannot consider all direction from the hemisphere, we have to limit the amount we sample. This is important since we have to incorporate the probability of selecting a single direction when we use the MC method. There are two often used sampling strategies with each having a different PDF, which we will explain in the next subsections.

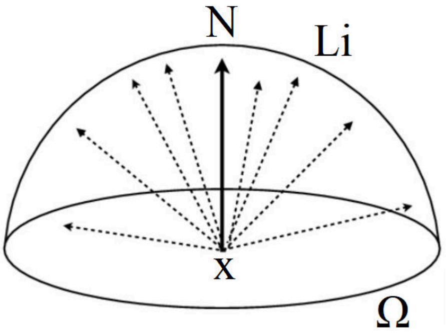


Figure 2.5: The hemisphere defined by Ω that needs to the sampled for incoming light in order to calculate the shading of point x.

Before we look at these sampling strategies, we have to explain a few things. In comparison to Equation 2.1, the Rendering Equation is often written as seen in Equation 2.13 with the $cos(\theta)$ explicitly stated. θ is the angle between the incoming light direction and the normal at point x and can also be expressed as the dot product of said vectors. This $cos(\theta)$ term is based on the Lambert's cosine law and describes how the light intensity decreases with a increasing θ. This property will be important when we talk about cosine weighted hemisphere sampling.

$$L_o = L_e + \int_{\Omega} BRDF * L_i * cos(\theta)\, dx \tag{2.13}$$

A simple form of the BRDF is called Lambertian BRDF and is defined as $\frac{c_{diff}}{\pi}$ with c_{diff} being the diffuse color of the object (albedo). The $\frac{1}{\pi}$ factor is needed as a normalization factor since integrating $cos(\theta)$ over the hemisphere results in π. The Lambertian BRDF does not incorporate the view vector, therefore it is view independent.

Uniform Hemisphere Sampling

Uniform sampling gives each direction along the hemisphere the same PDF of $\frac{1}{2\pi}$. Based on Equation 2.13 the estimator for the integral can be constructed as shown in Equation 2.14.

$$\begin{aligned} E\left[L_o\right] = L_e + \frac{1}{N}\sum_{i=0}^{N}\frac{\frac{c_{diff}}{\pi}L_i\,cos(\theta)}{\frac{1}{2\pi}} &= L_e + \frac{1}{N}\sum_{i=0}^{N}\frac{c_{diff}}{\pi}L_i\,cos(\theta)\;2\pi \\ &= L_e + \frac{2c_{diff}}{N}\sum_{i=0}^{N}L_i\,cos(\theta) \end{aligned} \tag{2.14}$$

Cosine Weighted Hemisphere Sampling

Cosine weighted hemisphere sampling makes use of the fact that incoming rays from the lower parts of the hemisphere, do not contribute as much light to the point as rays that come from the top of the hemisphere. As mentioned earlier, this phenomena is described by Lambert's cosine law. We want to proportionally generating fewer rays at the bottom of the hemisphere, compared to the top. This can be achieve by using $\frac{cos(\theta)}{\pi}$ for the PDF. The resulting estimator can be seen in Equation 2.15 .

$$\begin{aligned} E\left[L_o\right] = L_e + \frac{1}{N}\sum_{i=0}^{N}\frac{\frac{c_{diff}}{\pi}L_i\,cos(\theta)}{\frac{cos(\theta)}{\pi}} &= L_e + \frac{1}{N}\sum_{i=0}^{N}\frac{\frac{c_{diff}}{\pi}L_i\,cos(\theta)\;\pi}{cos(\theta)} \\ &= L_e + \frac{c_{diff}}{N}\sum_{i=0}^{N}L_i \end{aligned} \tag{2.15}$$

2.4.4 Next Event Estimation

Instead of choosing random direction within the hemisphere, we can also select directions that go in the direction of a light source. This saves us the calculation of paths that will not hit a light. When we shoot a ray to all lights (N is the number of lights) through this approach, hitting a light has a probability of $\frac{1}{N}$. If we shoot a ray to only one light, we must multiply this probability by N for MC to converge properly. This probability only applies when the selection of a light is uniformly distributed. For more intelligent light sampling strategies, a more sophisticated probability must be computed. Using NEE can decrease the noise and make the MC method converge faster. It also improves the performance when a denoiser is used.

2.5 Denoiser

Since the results from ray/path-tracing algorithms can contain noticeable noise, filters are used to reduce this noise. This implementation is called a denoiser and can be very useful especially for real-time applications. One of the simpler solutions for such a denoiser would be, for example a blur filter like the Gaussian filter. The drawback is, that we lose a lot of details, since we just blur the input image. In case of a movie or video game, one solution would be to reuse the information from the previous frames and apply it to the current frame to reduce the noise. With the use of motion vectors, which describe the change of pixels between the previous and the current frame, we can map the information on a per pixel basis from the previous to the current frame. When using a denoiser for the production of a ray-traced video clip, the denoiser already knows the future movement of the camera and can incorporate this information. In case of a video game where the future movement of the camera is unknown, it is especially complicated to reuse previous information. Therefore, the denoiser needs to be flexible when it comes to keeping or discarding information.

2.6 RTX

In order to utilize the RTX ray-tracing technology with Vulkan, we need to understand a few aspects beforehand. This section will discuss acceleration structures (ASs) and ray-tracing pipelines. Both are terms which will frequently appear throughout this thesis.

2.6.1 Acceleration Structures

All the geometry that we want to use for ray tracing, needs to be saved in a data structure that is suitable and optimized for ray intersection testing. NVIDIA uses so called acceleration structures, which are build and updated on the GPU with the help of the Turing ray-tracing cores. This structure is divided into a two-level tree and splits into top and bottom level ASs. The bottom level ASs contain all the geometrical information and are therefore expensive to create and update, while the top level ASs contain just references to the bottom level ASs. See Figure 2.6 for a graphical representation. This structure can be seen like a scene graph where the leafs are objects in the scene with their own corresponding model-matrix. Since it is not possible to get vertex data (like positions, uvs, normals, etc) from each individual bottom ASs, we need to use a separate buffer, which we can use to look up all the data via indices and instant IDs in shader.

2.6.2 Ray-Tracing Pipeline

In order to make use of the ASs, a ray-tracing pipeline needs to be constructed. The pipeline shares some similarities with a rasterizer pipeline, like the use of descriptor sets for binding data (textures, buffers, etc.) to the pipeline. One key difference is that the ray-tracing pipeline uses a shading binding table to invoke different shaders during the lifetime of a trace call. There are 5 different shader types in total but only 3 are

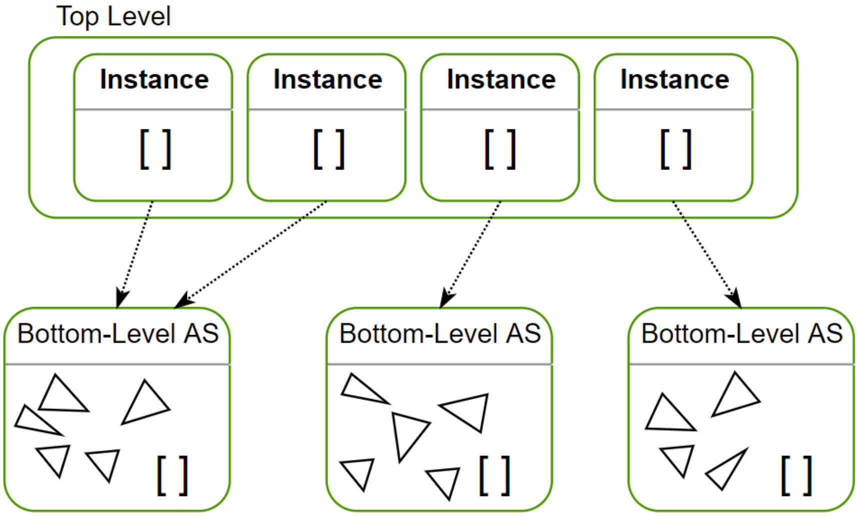


Figure 2.6: Top and bottom level ASs and how they are related. [nv]

mandatory. A payload is used to communicate data between all these shader stages. In the following enumeration we explain the different shader types.

Ray Generation Executed for each pixel and is used to shoot a ray through the top level AS that contains the bottom ASs. This is the starting point of the ray-tracing pipeline.

Miss Executed when no object is hit. Can be used to sample a sky map for example.

Closest Hit From all object hit along the ray, this shader is only called for the closest surface.

Any Hit (optional) This shader is called for every intersection along the ray. One use case is, that it can be used to ignore intersection, so they will not be considered for the closest hit.

Intersection (optional) Executed before the any hit shader and can be used to implement intersection handling.

The shading binding table can contain multiple shaders for the same shader stage (e.g. multiple closest hit shaders), which are then indexed based on a previously defined offset. This means, we can use different shaders for different materials/objects. It is also possible to give certain objects different flags, which can then be used to exclude them from the trace.

Figure 2.7 shows the steps involved when shooting a ray. It starts with the ray generation stage, which is used to shoot a ray at a top level AS. During the traversal of the AS,

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

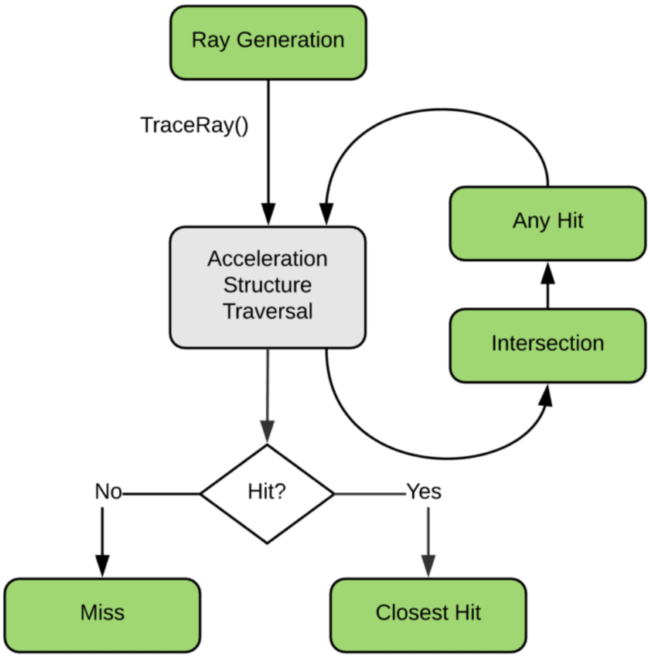

Ray Generation
TraceRay()
Any Hit
Acceleration Structure Traversal
Intersection
No
Hit?
Yes
Miss
Closest Hit

Figure 2.7: Graph showing the flow of a ray-tracing call. [nv]

the any hit and intersection shaders can be invoked for each object the ray intersects with. Once the ray reaches its maximum distance, the closest hit shader gets called for the closest object hit during the AS traversal. If no object was hit, during the traversal, the miss shader gets called. A ray payload is used to transfer data between these stages. There are flags which can be used to modify the behavior of the traversal, like terminating the ray on first hit (useful for shadow rays) or ignoring all (non-)opaque objects. Once we got the result, it needs to be saved to an image buffer, which then needs to be copied to the swapchain for presentation.

CHAPTER 3

Related Work

3.1 Ray Tracing and Path Tracing

In the previous chapter we mentioned some problems we need to overcome for ray/path-tracing to function correctly and efficiently. This section will present some state-of-the-art research that focuses on improving the current solutions for some of these problems. Additionally one topic will be discussed which focuses on using ray-tracing for a different rendering problem.

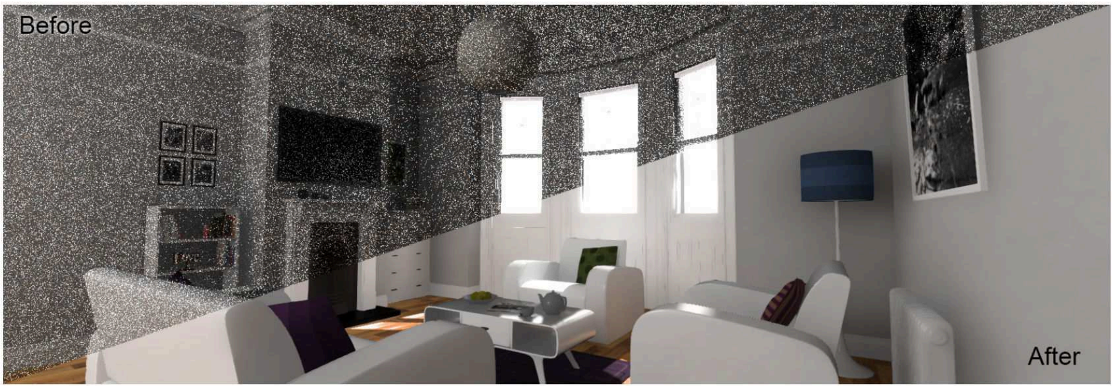


Figure 3.1: Denoising results from the approach presented by Moreau et al. (Left: before / Right: after) [MMBJ17]

We explained that one strategy to reduce noise is to increase the SPP. If the sample count is too low, the result exhibits a lot of noise. For real time rendering the frame time is important, therefore distributed ray-tracing or path-tracing is limited in its sample count. A paper from 2017 introduced a method, which can denoise ray-traced global-illumination with a low number of samples [MMBJ17]. The idea behind the algorithm is, to separate direct light from indirect light and apply temporal kernels, which grow in blurry or under sampled regions. The results are later recombined to create the final image (see Figure 3.1). Denoising results from ray-tracing algorithms with a low number of SPP is an important topic to allow the use of ray-tracing algorithm in real time rendering applications.

Another problem of interactive ray-tracing applications with a low number of SPP count is, that they often suffer from objectionable temporal artifacts when a new frame is freshly rendered. A method which solves this problem is called stable ray tracing [CSK$^+$17]. Instead of using temporal post-processing filters like TAA, they use a combination of sample re-projection and explicit hole filling. The idea is to cache shading location from the previous frame and reuse all locations which are still visible for the current image. Hole filling is then used in regions where the density of visible re-projected points is low. The last step is to generate the final image from the shaded samples.

Adaptive Spatiotemporal Variance-Guided Filtering (A-SVGF) is another solution to denoise images having a low sample count [SPD18]. The key aspect of this algorithm is, that it analyzes the signal over time and derives an accumulation factor between the previous and current frame for each pixel. This can reduce ghosting and improve stability. A-SVGF was designed with real time applications in mind and runs at 1080p in around 2 ms on modern hardware.

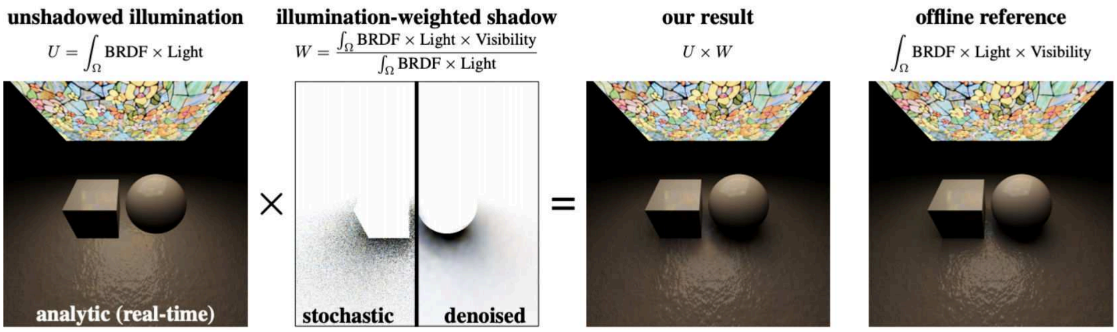

unshadowed illumination
$U = \int_\Omega \text{BRDF} \times \text{Light}$
illumination-weighted shadow
$W = \frac{\int_\Omega \text{BRDF} \times \text{Light} \times \text{Visibility}}{\int_\Omega \text{BRDF} \times \text{Light}}$
our result
$U \times W$
offline reference
$\int_\Omega \text{BRDF} \times \text{Light} \times \text{Visibility}$
×
=
analytic (real-time)
stochastic
denoised

Figure 3.2: Illumination results from the approach of Heitz et al in comparison to offline results. [HHM18]

Speeding up the calculations of realistic illumination and shadows for a scene is always a non-trivial task. The authors of following paper propose an approach where analytical illumination is combined with stochastic raytraced shadows [HHM18]. The results are comparable in terms of realism with an accurate offline approach. (see Figure 3.2). In order to calculate the illumination part, a modified version of the rendering equation is used, which does not utilize the visibility factor. For the shadow calculation part, random samples are used to compute the visibility of the light source. The resulting image is, because of the nature of random samples, noisy and needs to be denoised for a smoother result. The last step is to combine both parts, which results in the final shadowed illumination of the scene.

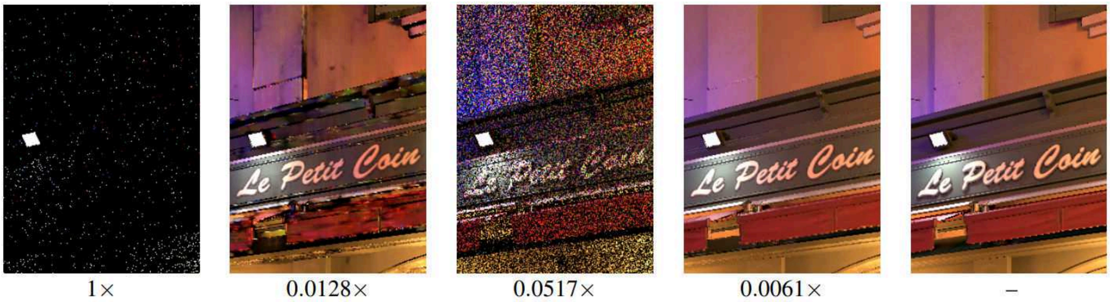


Figure 3.3: First two images are 1 spp with uniform sampled lights and denoised with SVGF [SSK+17] over 5 frames. Next two images are 1 spp with BVH sampled lights and denoised with SVGF over 5 frames. The last image is the ground truth. [MPC19]

We have already mentioned that all objects of a scene need to be organized in order to increase the speed of intersection testing with a ray. The same can be done with the light sources as well. In order to keep the amount of shadow rays to trace low, not all lights should be considered for every shading point. Therefor, lights need to be selected stochastically. NVIDIA recently published a paper [MPC19], which tries to solve the problem by giving more important lights a higher probability of being selected for a given shading point. The idea is to build a BVH over the lights (a "light BVH") and traverse the tree to find the best lights. Every cluster has a given priority, with more important cluster having a higher priority. At each branch of the BVH traversal a random decision is made about which path to take. This approach is also suitable for dynamic lights, because the authors also present a strategy to update the BVH every frame. The idea is to keep the topology and just refit changes instead of recreating the entire hierarchy every frame. See Figure 3.3 for the results.

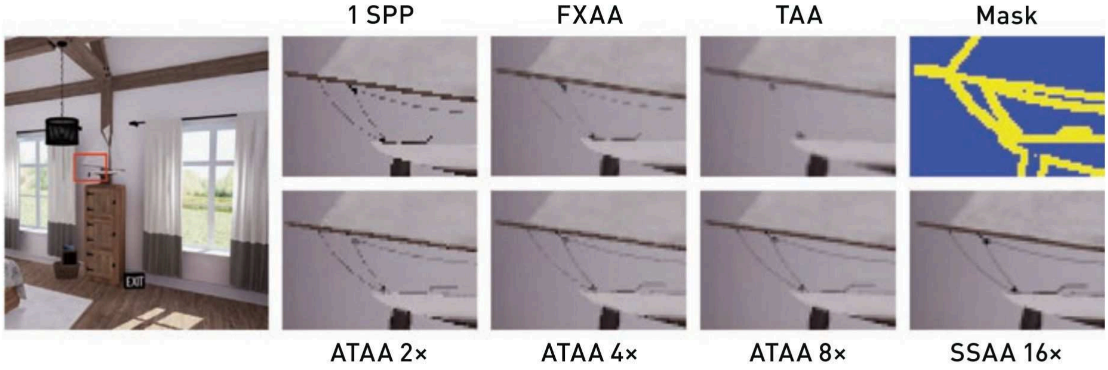


Figure 3.4: ATAA in comparison to FXAA and TAA. [MSG+19]

There is also active research for using ray tracing to solve other problems of computer graphics. A recent paper showcased a method to improve anti-aliasing for deferred rendering, which should deliver results close to SSAA (Super Sampled Anti-Aliasing) (see

Figure 3.4) [MSG+19]. The authors call it adaptive temporal anti-aliasing (ATAA) and it uses temporal anti-aliasing (TAA) in combination with fast approximate anti-aliasing (FXAA) and adaptive ray tracing. All this is done within a time frame of 16ms, which is equal to 60 frames per second and is the target by most game engines. The overall idea is to run TAA on most pixels and for every pixel where it fails, a more robust heuristic based on sparse ray tracing is used. Therefore, ray tracing is only applied on a few pixels, which saves some computational work. Since this technique heavily depends on ray tracing, a GPU that accelerates ray tracing is mandatory.

In 2004, there was an attempt to implement ray tracing into Quake III. Performance wise, they achieved 20 frames per second which is impressive considering this was in 2004. There is no source code available, therefore the information we have about this project is very limited. [quad]

3.2 RTX

There were past attempts for hardware accelerated ray tracing, with RTX being the most recent attempt and the first product targeted for the masses. One of the first question that arise is, how does it compare to software-based ray tracing. This topic is discussed in a paper written by Zakharov et al. [SGFV19] The authors ray trace multiple scenes with an RTX 2070 with hardware acceleration and an GTX 1070 with a software fall back. The results are then compared. Since Vulkan and DirectX, both support RTX ray tracing, both were used for testing. In Figure 3.5 the results are shown. The RTX 2070, with RTX activated, has a lead of twice or more million rays traced per second compared to the tests where RTX was disabled or not available. For two tests they used Hydra (marked with _hydra in Figure 3.5), which is an open source renderer, with the source code available on GitHub. For the other two tests they implemented a ray tracer themselves.

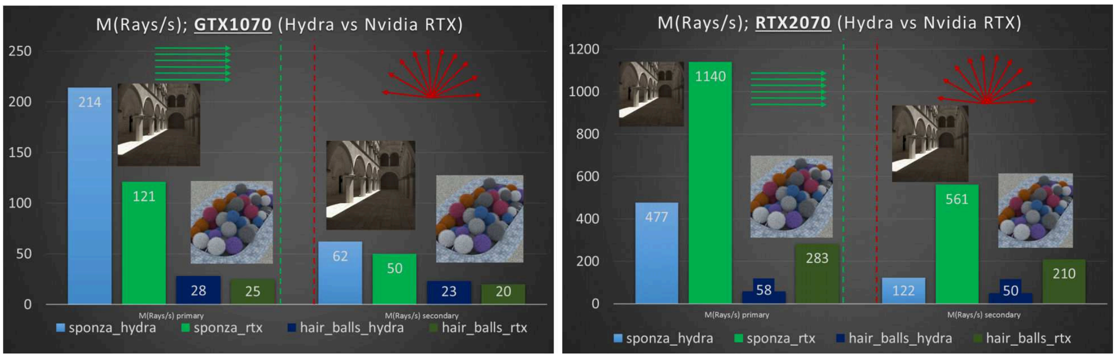


Figure 3.5: Million rays traced per second comparison of GTX1070 (left) and RTX2070 (right). [SGFV19]

There are also attempts about using RTX functionality outside the indented scope. Wald et al. [WUM+19] use the ray tracing cores of the Turing architecture, not to solve

TU Bibliothek Your knowledge hub WIEN
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

classical ray-tracing problems, but for finding the tetrahedron from a given tetrahedral mesh that contains a give point. In order to allow that, they describe and evaluate three different approaches to reformulate this problem for RTX. The strategy is, to use hardware accelerated BVH traversal and triangle intersection when tracing rays against the tetrahedrons faces.

Another not intended use case for the ray-tracing cores is presented in this next paper. Salmon et al. [SS19] used OpenMC, which is a community-developed Monte Carlo neutron and photon transport simulation code (making use of MPI and OpenMP) and modified it in order to use the ray-tracing cores from the Turing architecture for acceleration. The model, which showed the biggest increase, was 20.1x faster with RTX enabled on Turing compared to the test where RTX was disabled. They presented the results for five different models, which were tested with 1.000.000 particles each. The performance uplift for RTX-On compared to RTX-Off was 6.0x on average.

The Turing architecture does not only provide functionality to accelerate ray tracing, but also has hardware for accelerated matrix multiplication. These cores are called tensor cores and are useful for machine learning for example. Raihan et al. [RGA19] explain the architecture of these cores and also modeled them in a GPU simulator which resulted in a accuracy of 99,6% compared to the NVIDIA Titan V GPU.

There are open source projects on GitHub [git] which make use of hardware accelerated ray tracing to some extend in an offline and online context. One notable use cases for hardware accelerated ray tracing is the implementation of Global illumination (GI) for Quake II by NVIDIA (see Figure [quaa]). In this implementation, Quake II Pro (which is an improved version of Quake II) was extended with ray traced global illumination. In Figure 3.6, a comparison of the original and the modified version of Quake II can be seen. A-SVGF [SPD18] is used to denoise the noisy end result. The output of the path-tracer is noisy, because they only use one shadow ray per pixel with the addition of one or two bounces (each bounce shoots one shadow ray). Quake II RTX is based on q2vkpt [quab] which was one of the inspirations for this thesis. Since such an implementation is not trivial and there is no documentation available which explains the overall implementation of Quake II RTX or q2vkpt, it is difficult to use them as a reference. Therefore, one of the goals for this thesis is, to provide a documented implementation of a real time ray tracer and show its potential using a more advanced game engine.

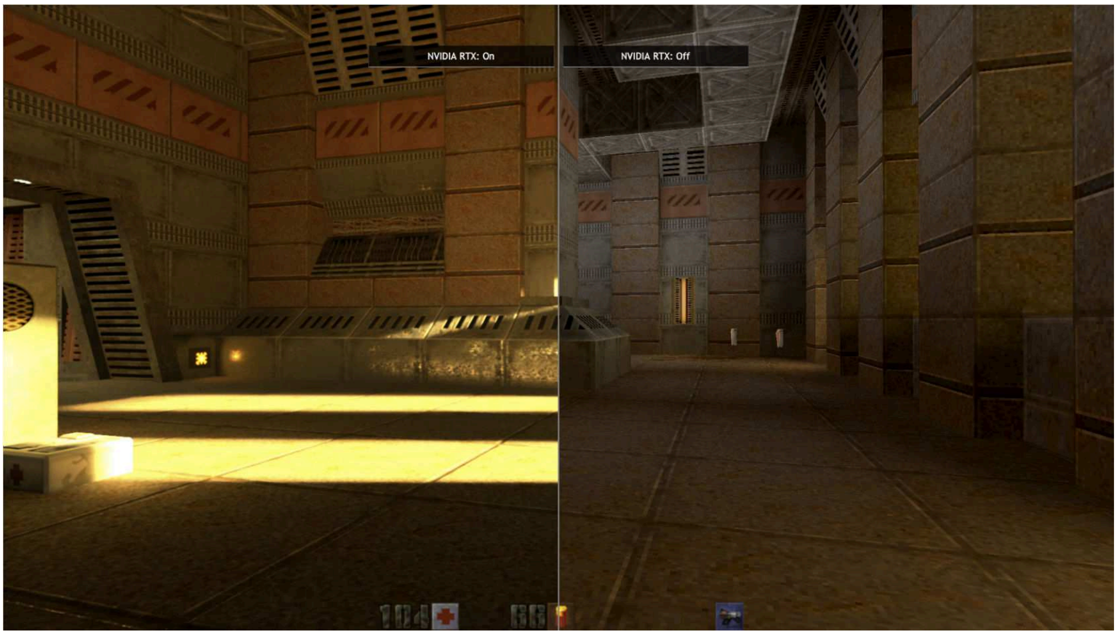

NVIDIA RTX: On
NVIDIA RTX: Off

Figure 3.6: NVIDIA RTX Tech Demo of Quake II (Left: with RTX / Right: original). [quaa]

CHAPTER 4

Architecture

4.1 Offline/Online Ray Tracing Differences

When we think about ray tracing, we mostly think about very computationally expensive algorithms, which produce high-quality results that are closer to photo-realism than a rasterized result and are still mostly used in professional software. Games or heavily time-dependent applications were never considered for ray tracing except for a few exceptions, like certain effects (e.g., screen-space reflections). Even though we now have the hardware to fully integrate ray tracing in a real-time application, we still need to modify the typical ray-tracing approach, since the original offline approach is not suited for real-time use. The problem is that the amount of SPP we can calculate each frame is limited by the FPS we want to achieve. In order to get a noise-free result, in theory we would require an infinite amount of SPP (except for the most basic ray-tracing approach, which is called Whitted ray tracing [Whi80], since it is deterministic). Even with more powerful hardware than we have nowadays, this would still be a problem. Therefore, we need to limit algorithms like distributed ray tracing or global illumination through setting a maximum on sample and depth count, with the requirement to still produce a result in a certain amount of time (depending on the FPS we want to achieve). Limiting sample and depth count is a first step but is still not enough. It can be helpful to bias the calculations in order to get results faster. For random decisions, it is important to stochastically use decisions which provide the most information instead of trying a true random path and having a high chance of ending up with a decision that does not contribute valuable information to the result. This results in multiple different ray-tracing implementations that try to handle these problems better. For the implementation of a real-time ray/path tracer, the high-level architecture mostly looks the same. Therefore, we can provide a general solution for the design of the overall structure and explicitly state differences if there are any. In Table 4.1, the different techniques we will discuss, together with a pro/cons list of their feature differences, are listed.

Type	Pros	Cons
Whitted ray tracing [Whi80]	noiseless deterministic reflection/refraction	no indirect lighting hard shadows aliasing
Distributed ray tracing [CPC84]	soft shadows anti aliasing glossy reflections motion blur depth of field glossy reflections	no indirect lighting noise
Path tracing (GI) [Kaj86]	pros from Dist RT indirect lighting	noise

Table 4.1: Pros and cons of different ray/path-tracing techniques.

As mentioned before, the discussed ray-tracing algorithms share the same foundation, therefore the following presented strategies can be applied to all of them. Merely for global illumination, an extra stage is required. In order to create an online variant of different ray-tracing techniques, we need to split the offline algorithms into multiple parts and handle them separately. This comes with the advantage that each stage can be individually denoised with seperate denoisers adjusted for each stage. For Whitted ray tracing and distributed ray tracing, the individual stages are: first hit, secondary hit and direct illumination, which we will discuss in detail in the next paragraph. In case of global illumination (path tracing), an indirect illumination stage is added. When we look at the algorithm for an offline ray/path-tracing approach, all these stages are hard to differentiate from the code since they are somehow implicitly contained in the algorithm.

In order to understand the different parts of the pipeline, we will discuss each stage in detail. The order of the listing is the same as the execution order in the code. It is possible to implement all these stages in one shader and execute them in one trace call, but for reasons like the use of individual denoisers per stage, it is recommend to use a G-Buffer and split these stages into different shaders with individual trace calls.

Primary Ray Stage Trace one ray per pixel from the camera position until the first object is hit. Save its color, position, material and other features to the G-Buffer.

Secondary Ray Stage This stage gets executed for materials that require additional rays like glass, water, mirrors or objects that make use of the alpha channel for see-through regions. If no according object was hit, this stage can be skipped. Depending on the implementation, this stage can be executed multiple times in order to reach a certain reflection/refraction depth, or just once when recursions are used. Update G-Buffer with new data for affected pixels.

Direct Illumination Stage For each pixel, the direct illumination will be calculated based on the information gathered from the primary and secondary ray stage. This stage shoots shadow rays to one or more lights to find out if a given point is visible from a light source. A better way to do this is to have a list of lights that have a higher probability to be visible from the given position and (depending on the quality) shoot one or more shadow rays in the direction of these lights.

Indirect Illumination Stage This stage is only used for path-traced global illumination and should handle the bounces for the indirect light. Based on the hit position we got from the previous stages, we reflect the incoming ray in a random direction along the hemisphere of the surface and save the information of the object it hits. This can be done multiple times depending on the required amount of bounces. For each bounce, one or more shadow rays are calculated from the new hit position. One or two bounces should be enough in most cases for a reasonable quality/time trade off.

4.2 Distributed Ray Tracing

In order to implement distributed ray tracing, three stages are required (primary ray stage, secondary ray stage and direct illumination stage). Distributed ray tracing has the advantage that it is based on Whitted ray tracing, which is deterministic and therefore can be implemented without the use of a denoiser. Even though the noise introduced by the additional distributed ray-tracing effects is visible, its extent is low enough that it does not distort the result in a way that is unpleasant to look at, and can easily be removed with Temporal anti-aliasing (TAA), for example. For the random samples, values from blue-noise textures are used. This has the advantage that the values are evenly distributes in comparison to white-noise textures.

We use a G-Buffer for the implementation of this thesis to save all the data. The first hit stage saves the position, normal, color, material and the cluster to the G-Buffer. Additionally the blended colors from all the transparent objects (from the eye to the first opaque object) that were hit along the ray are also saved to the G-Buffer. A good way to decrease the time required to trace one frame is to only use one light source per pixel. This introduces a lot of noise that needs to be handled by a denoiser. In order to reduce the noise, we can also use all available lights for the scene. Since shooting shadow rays to all lights can result in many lights contributing nothing to the result, we can exclude some lights that are not visible for a given position and thus increase the performance. This will be explained in more detail in the next section. In Chapter 7 we will take a look at how this affects the performance of our implementation.

4.3 Lights Culling

In order to increase the frame rate of the ray tracer or reduce the noise of the result, it is important to cull lights which are not visible from a given position. This is a nontrivial task since culling important lights can result in inconsistent lighting across the scene. There are different approaches to achieve this, but since we already have a PVS available,

#									
3	23	34	74	0	0	0	0	0	0
1	103	0	0	0	0	0	0	0	0

Table 4.2: Example for the light visibility texture. Each row represents a cluster. The first column shows how many lights are visible, and the numbers after that are the indices of the lights. In this example cluster 0 is visible from 3 lights with indices 23, 34, 74. Cluster 1 is visible from 1 light with index 103.

it made sense to try to reuse it for light culling. Though, this implementation depends on the PVS provided by Quake III, it should be possible to apply it on any PVS. The PVS is basically a binary array with bits set at certain positions which define which clusters are visible from a specific cluster. A cluster is an axis-aligned bounding box (AABB), defining a volume in space, and everything inside of it is part of the cluster. In order to get the information for a specific cluster, we need to read the correct portion of the PVS array. This is done by offsetting the pointer from the start of the PVS array by the stride for one cluster times the cluster index `vis = PVS + clusterIdx * clusterStride`. With this information we can check all the other clusters if they are visible from the current cluster. In the case of Quake III this is done through checking if the following expression equals true `(vis[cluster»3] & (1«(cluster&7)))`.

4.3.1 Cluster Merging

Quake III makes use of the clusters on a per-object basis, but we want to use it on a per-pixel basis. So we want to check for each pixel which cluster contains the current world-space position and what clusters are visible from it. Since we want to assign only one cluster to each triangle, it can happen that this is not possible since each vertex of the triangle is in a different cluster. In this case we need to merge the clusters into a single new cluster with the combined visibility information of all three vertices.

4.3.2 Light Texture

We save all lights in an array and upload it to the GPU. Then we create a texture with #cluster rows and maximum #lights columns, which contains all lights that are visible from the cluster of the according row. The first column is the number of lights visible from the cluster in the specific row. From column 1 till #lights, the indices for the corresponding lights inside the light array are stored. See Table 4.2 for an example of the content of this light visibility texture. When calculating the illumination for a given pixel, we look up the according row, and iterate through all lights or, randomly select one light.

4.4 Denoiser

The denoiser we use is called Adaptive Spatiotemporal Variance-Guided Filtering (A-SVGF) [SPD18] and consists of multiple parts, with some parts being executed before the first ray-tracing stage and some after the last ray-tracing stage. The denoiser should take an image that was rendered with only one random light source per pixel and produce a noise-free result. In the first step, the denoiser tries to project information like vertex position and object id from the previous frame to the current frame. Since we can already use the projected information for the primary ray stage, it is important to do this before any rays are traced. Once all the ray-tracing stages are executed, we calculate the gradients for the luminosity changes between the current and previous frame. With the help of the output from the ray tracer and the previous accumulated history, we can calculate weights which define how much information from the previous frames should be reused for the current frame. This helps in eliminating artifacts like ghosting. If the history for a given pixel is no longer valid, we discard it completely. Since we sparsely sample the gradient, we need to reconstruct it with a guided filter like À-Trous. The reconstructed information is added to an accumulation buffer. It is important to not add filtered data to the accumulation buffer since it can result in a growing blur effect. The last step is to use TAA to get an even smoother result. For a more detailed explanation of A-SVGF, refer to the paper [SPD18].

4.5 Temporal Anti-Aliasing

Uses the results from the previous frame to improve the quality of the current frame. For this implementation TAA is not only helpful for removing aliasing on edges, but also to remove left-over firefly noise after the denoiser step. It also helps with removing noise produced by the distributed ray-tracing effects.

CHAPTER 5

Quake III modifications

Quake III was released in 1999 and was therefore not developed with modern computers or operating systems in mind. In order to make the code compile, outdated dependencies needed to be replaced on top of other fixes. We also moved the code from 32-bit to 64-bit, which resulted in memory-alignment problems and binary operations that needed further attention. Since Quake III is a commercial game developed by id Software, the amount and complexity of the code is considerably high. Luckily, there are resources online from people who have analyzed the code to some extent. These documentations often lack low-level details, but they are helpful to understand the overall structure of the code. Fabien Sanglared's website [quac] gives a good explanation about the different parts/modules of the Quake III source code. From his research we found out that Quake III's engine uses a virtual machine (VM) to execute the game logic for the UI, Server and Client parts. The VM supports three different execution modes. `VMI_BYTECODE` uses the precompiled bytecode, which is saved in the assets of Quake III, and interprets it, while `VMI_NATIVE` uses dynamically linked libraries for these parts. `VMI_COMPILED` is used to translate the bytecode to native instructions, but was removed since we do not need it. We mostly use `VMI_NATIVE`, which can be activated with the arguments `"+set vm_game 0 +set vm_cgame 0 +set vm_ui 0"`, since it is more comfortable to work with. Each of the three processes (for server, client, ui) is executed in a VM and uses system calls to make changes to the engine e.g. load models, add models to scene, render scene. The UI process is used for setting up and maintaining the UI. For the single-player game mode, Quake III creates a server on localhost and then creates a localhost client that connects to the server. The server side keeps track of the current game states and distributes it to the client, which then renders the current frame based on this state. The engine is linked to two separate static libraries, one that is called bot.lib and one that is called renderer.lib. In order to get human-like behavior for the enemies, bot.lib is used, and for all the rendering operations, renderer.lib is used. The engine takes the system calls from the virtual machines and passes them to these two implementations if required by the process. In Figure 5.1 a visualization of the Quake III architecture can be found.

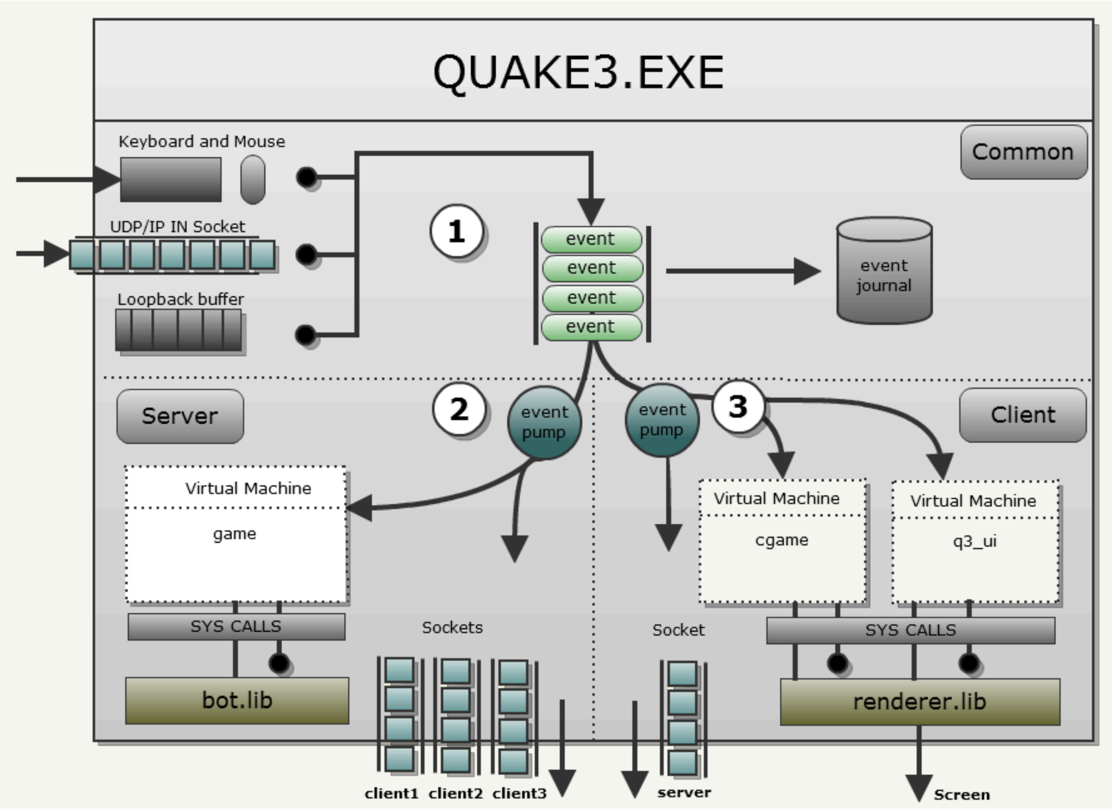

QUAKE3.EXE
Keyboard and Mouse
Common
UDP/IP IN Socket
1
event
event
event
event
event journal
Loopback buffer
Server
2
event pump
event pump
3
Client
Virtual Machine
game
Virtual Machine
cgame
Virtual Machine
q3_ui
SYS CALLS
Sockets
Socket
SYS CALLS
bot.lib
renderer.lib
client1
client2
client3
server
Screen

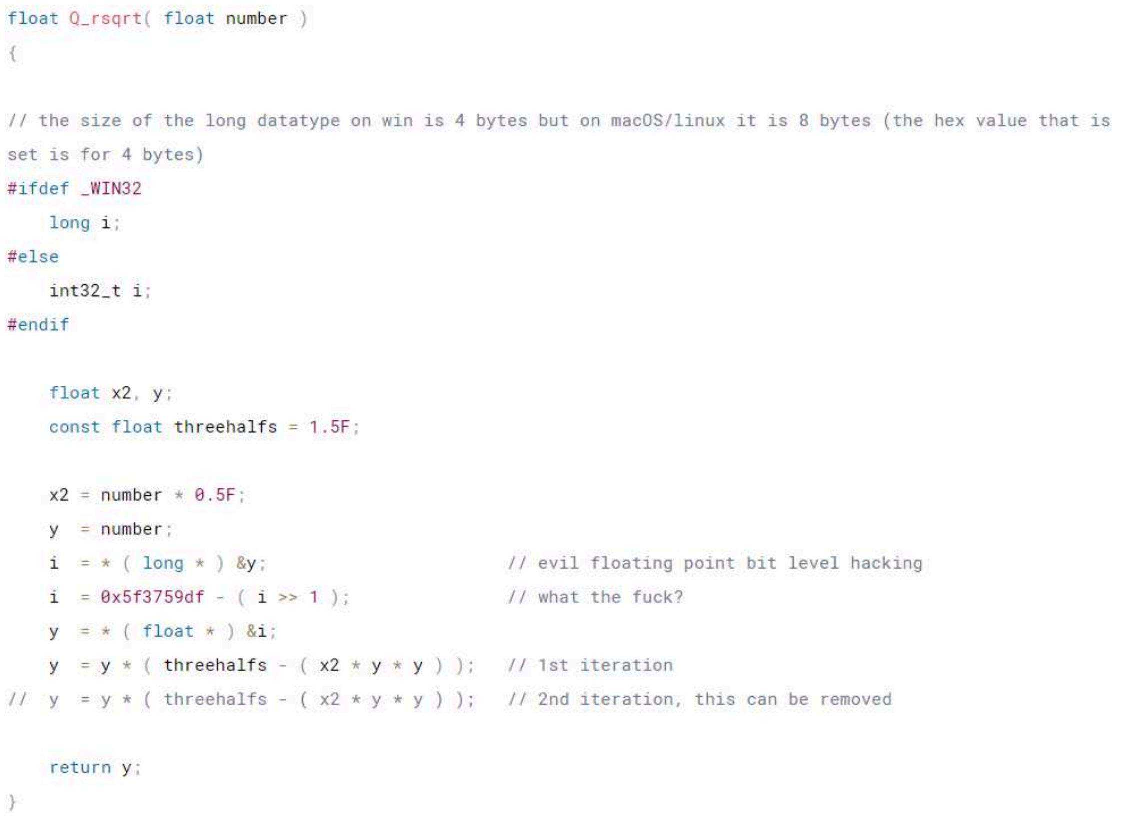
Figure 5.1: The diagram shows all the different parts of Quake III and how they are connected with each other. The server side uses one virtual machine while the client side uses two virtual machines. The engine provides system calls to the artificial intelligence module and the renderer module. [quac]

5.1 Refactoring the original Quake III Code

First of all, it was necessary to replace some dependencies that where outdated. The code that was used to load images and decompress jpeg files did not work anymore, therefore we used stb and TinyJPEG to replace it. Another problem was that binary operations that are used on data types whose size depends on the OS (32 bit or 64 bit environment) can produce wrong results when moved from 32-bit to 64-bit. Table 5.1 shows the differences regarding the long integer type. On Windows, there is no difference between 32 and 64-bit long types but since Quake III runs on Windows, Linux and macOS, we had to consider it. One example for a binary problem caused by the altered size of the type is the famous inverse square root hack used in Quake III. It is an approximation of the ground truth that was four times faster than other methods at that time. The code uses the magic number `0x5F3759DF` and one iteration of Newton's method for the calculation. Since bit operations are involved, we needed to change the `long` type to `int32_t` for Linux/macOS. See Figure 5.2 for the code.

OS	Architecture	Size of "long" type
Windows	IA-32	4 bytes
	Intel® 64	4 bytes
Linux	IA-3	4 bytes
	Intel® 64	8 bytes
mac OS	Intel® 64	8 bytes

Table 5.1: Table that shows the different sizes between 64bit and 32bit environments of long int in bytes. [int]



```
float Q_rsqrt( float number )
{

// the size of the long datatype on win is 4 bytes but on macOS/linux it is 8 bytes (the hex value that is
set is for 4 bytes)
#ifdef _WIN32
    long i;
#else
    int32_t i;
#endif

    float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y  = number;
    i  = * ( long * ) &y;                       // evil floating point bit level hacking
    i  = 0x5f3759df - ( i >> 1 );               // what the fuck?
    y  = * ( float * ) &i;
    y  = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
//  y  = y * ( threehalfs - ( x2 * y * y ) );   // 2nd iteration, this can be removed

    return y;
}
```

Figure 5.2: The quick inverse square root hack used in Quake III, with modifications for x64.

Another fix was required for the VM, because the VM uses bit operations for the execution of the bytecode. The problem was that the size of a pointer on x86 is 4 bytes and on x64 it is 8 bytes. Quake III was developed with 4-byte pointers in mind, which resulted in wrong strides for instructions on x64. This could be solved with the use of `intptr_t`.

There were other bugs that needed to be fixed but the above-mentioned ones where the most notable ones.

5.2 Implementing Vulkan in addition to OpenGL

Quake III originally shipped with OpenGL 1, which means the developer had to use a fixed-function pipeline without the use of buffers (at least when certain extensions where not available). Since this was very limiting, they built a dual-core renderer with material-based shading on top of OpenGL. RTX ray tracing does not work with OpenGL at all, therefore we needed to implement a renderer using the Vulkan API. This is not trivial since OpenGL 1 is very old and therefore completely different compared to modern rendering APIs in terms of features but also structure. At that time, most tasks where done on the CPU instead of the GPU, like the calculation of the next positions in case of an animated mesh. Since the differences in structure between OpenGL 1 and Vulkan are so big, it was not possible to implement Vulkan in the most optimal way. In order to do so, it would most likely require a rewrite of a big part of the engine.

For each object, Quake III uses multiple render stages, which are rendered on top of each other and blended accordingly. For example, a computer terminal model has one texture for the base color, one texture for details on top of the base color and one animated texture for the pulsing lights. On top of these 'normal' stages, so-called light maps are added, which contain the baked lighting for the objects in the scene. Therefore, for each object there can be multiple textures with different blend operations. Additionally, each stage can have two different textures, which are combined via an add or multiply operation (this is an OpenGL extension and must be supported and activated).

In order to implement Vulkan, we need to recreate the functionality for everything that was mentioned in the previous paragraph. Since all vertex data is handled by the CPU and each vertex is rendered with OpenGL immediate mode (which means, with every frame all vertex data needs to be send to the GPU all over again; no persistent memory on the GPU for an individual object), we need to use memory allocated on the device (GPU) that is mapped to the host (CPU). Vulkan provides the following flags for said behavior: `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT` (see Figure 5.3). This allows us to write directly to the GPU memory and is useful because for each new frame, we need to transfer all vertex data to the GPU again. Since we use multiple images in our swapchain, one buffer for each swapchain image is necessary in order to avoid race conditions.

For the rasterizer we only need one vertex and one fragment shader. The textures are accessible through a `sampler2D` array with dynamic size since we will need access to all textures at once later for the ray-tracing implementation. This functionality is an extension of Vulkan, therefore we need to enable `GL_EXT_nonuniform_qualifier` in GLSL and `VK_EXT_descriptor_indexing` as a device extension. After that we need to activate the following device features in the following struct:
`struct VkPhysicalDeviceDescriptorIndexingFeaturesEXT`

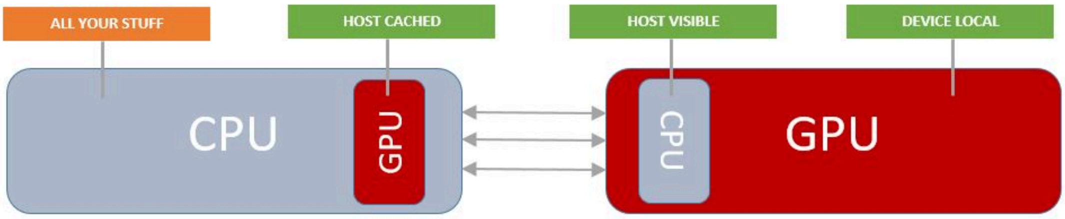


Figure 5.3: Different memory configurations in Vulkan.

`runtimeDescriptorArray`,
`descriptorBindingVariableDescriptorCount` and
`descriptorBindingPartiallyBound`

Another way to use different textures with each object would be to bind different descriptors when rendering different objects. For the ray tracer later on this will not work, since we do not know beforehand which object we will hit, and we cannot change the descriptor between ray hits. Therefore, it is necessary to have access to all the textures through an array. Addition and multiplication of textures, in case of two textures per stage, clipping of vertices outside of the clip planes and discarding of vertices based on the alpha value was implemented with the use of conditions in the shader and can be activated through push constants. Since Quake III was not designed with Vulkan in mind, where state changes (pipeline, descriptor, etc) are very expensive, the draw commands are not sorted in a way that the amount of state changes are reduced. Therefore, it was beneficial to implement the whole functionality into a single shader, and bind all the required data through a single descriptor, so the amount of pipeline and descriptor set changes is reduced.

From a visual point of view, there is no noticeable difference between the OpenGL and Vulkan renderer. In terms of performance, the OpenGL renderer is slightly faster. The performance of Vulkan heavily depends on the number of state changes, how memory is accessed and how fast commands are recorded. All of this is not optimal in our implementation and would require a major rewrite of the original code to fix. OpenGL and Vulkan can be switched through the game settings or changing the flag `r_glDriver` to "`vulkan-1`" in the `q3config.cfg` file.

5.3 Other necessary Quake III modification

Before we can implement the ray tracer, we need to make some additional modifications to Quake III. One of those changes was to remove all render stages that contained light maps. Since we want to calculate our lighting entirely with ray tracing, we want to reduce the baked lighting as much as possible. For the specular highlights, this is most of the time not possible since they are part of the general texture (see Figure 5.4).

Quake III also uses environment maps to simulate reflection on certain objects. Through uv changes, which are in relation to the location of the player, it is possible to simulate a view-dependent reflection (e.g., glass windows). It was important to remove those since we want to add our own ray-traced reflections for those objects.

There were also some other textures and render stages that we needed to remove. One example would be light flares for some of the light sources or tin effects for metal objects. We also did some modifications and rearrangements for some render stages in order to fix multiple problems that occurred (e.g., wrong surface flags for certain objects).

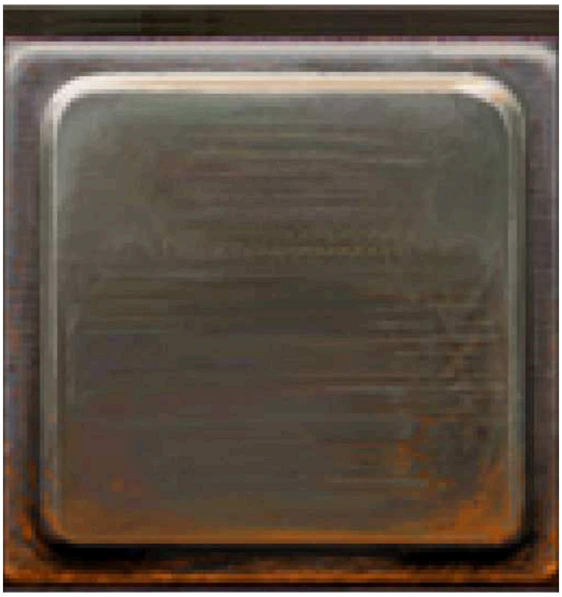


Figure 5.4: A cropped part of a texture from Quake III used for metal walls. In the upper part of the image, you can see the baked specular highlights, which cannot be easily removed. In the lower part, some baked shadows can be seen.

5.4 Distributed Ray-Tracing Integration

Since the architecture for the ray tracer is very different compared to the architecture of the rasterizer, we needed to implement it separate to the code of the rasterizer. Therefore, we could not modify existing code, and needed to write the ray-tracing backend from scratch. For the rasterizer, all objects that are not visible from the current view position are culled through frustum culling and the use of the PVS. The PVS contains data that describes which clusters are visible from a given cluster and is useful to cull objects from clusters currently not visible. A cluster is an AABB and can be used to describe all objects that are within the defined bounds of the cluster. This results in only certain objects of the scene getting added to the render list, which saves performance in case of the rasterizer but is a problem for the ray tracer since needed geometry can be missing. For example, objects outside the visible area that cast a shadow inside a visible area would be absent since they are culled. Therefore, we dismiss this method and instead always use all geometry from a loaded map. This is implemented in `tr_bsp.c`, which contains the code for loading the map of the current game level. Here we added code for building the bottom acceleration structures (ASs) containing the current map. We use multiple bottom ASs because we separate objects depending on the number of updates they require during run-time (we will explain this later in more detail). For entities (player, weapon or power-up models) we create one bottom AS each and instantiate it in case of multiple instances.

Then the different ray-tracing pipelines (as explained in Section 4.1) are created together with two descriptor sets. One for the texture array and one for all the other buffers required by the ray tracer. There are three ray-tracing pipelines in use. The first pipeline is for the first ray, the second one is for secondary rays like reflection/refraction rays, and the last one is for the direct illumination. They are executed in this order.

Once all the bottom ASs have been built, the rendering loop is executed. At first all the ASs that require updates will be updated or rebuilt and then added to a trace list (gets cleared at the beginning of each frame), which is then used to build the top AS for the current frame. After that, our global uniform buffer object, which contains, amongst other things, the inverse view/projection matrix, gets updated and the pipelines are executed in said order. The last step is to write the ray-tracing result to the swapchain image.

CHAPTER 6

Distributed Ray Tracing Implementation

In this chapter we will discuss the low-level implementation details for the distributed ray tracer. For the most parts, this will be a look on the general implementation with as few references to Quake III as possible. Table 6.1 contains all the console commands added to Quake III with this implementation.

`rt_printPerfStats`	Print performance stats to console
`rt_illumination`	Turn on/off ray traced illumination
`rt_cullLights`	Cull unnecessary lights
`rt_numRandomDL`	Number of random lights for direct illumination (0 = all)
`rt_pause`	Pause the game
`rt_accumulate`	Pause the game and accumulate all images
`rt_antialiasing`	Turn on/off anti aliasing
`rt_softshadows`	Turn on/off soft shadows
`rt_dof`	Turn on/off depth of field
`rt_aperture`	Set the aperture for dof
`rt_focallength`	Set the focal length for dof
`rt_denoiser`	Turn on/off the denoiser
`rt_taa`	Turn on/off temporal anti aliasing

Table 6.1: Commands that can be used to adjust different settings/values of our implementation.

6.1 Acceleration Structures

First, we take a look at how we can build the acceleration structures for our geometries in the best possible way. In general, we can split the geometry into static types, which will never change, and dynamic types, which deform during gameplay. Since updating ASs is expensive, it should be done carefully. One strategy to do this is to separate the geometry into multiple ASs to avoid updating geometries inside the AS that do not require an update. We split our geometry into gameworld data and entity data. In case of a completely static game world, the whole mesh could be put into a single AS, but since the game world also contains deforming geometry, we need to split it further into a static and a dynamic gameworld AS. For the entities, which are objects that appear and disappear frequently depending on the gamestate, it is important to use multiple ASs since we need the functionality of adding, updating and removing them independently. Since the geometry of the entities can change often, it would be a waste of resources to update also the static objects in the AS. It can also happen that an entity is used multiple times in the scene, therefore it is advised to, instead of duplicating the object, instantiating it multiple times in the top-level AS from a single bottom-level AS.

For our implementation we used 5 different bottom-level AS:

Static World AS with Static Data Contains the geometry of the gameworld where the topology or its data (normals, uvs or textures) does not change.

Static World AS with Dynamic Data Contains the geometry of the gameworld where the topology does not change but its data (normals, uvs or textures) does change.

Dynamic World AS Contains the geometry of the gameworld where the topology does change.

Static Entity AS Contains entity geometry where the topology does not change.

Dynamic Entity AS Contains entity geometry where the topology does change, and AS updates are required.

Since a single static entity AS can be used in multiple instances in the top-level AS, we always use a new separate data buffer (normals, uvs...) each frame for each instance, to avoid problems when data changes for multiple instances differently (e.g., different speeds for texture animations). This removes the necessity to differentiate between static entities where the data changes or not. In all cases where the geometry of the AS or its data changes, we need to use separate buffers for each swapchain image in order to avoid race conditions.

6.2 Ray-Tracing Pipeline

In order to use the ASs we just built, we need to create a ray-tracing pipeline. One essential part of the pipeline is the shader binding table, which contains all the shaders we want to use for the pipeline. It is possible to create a single shader binding table and use it for multiple different pipelines with the use of offsets for indexing into the shader binding table. In this implementation, we used multiple different shader binding tables for each ray-tracing pipeline.

Attachments	
Top Level AS	Contains the scene we currently use for ray tracing.
Global UBO	Contains configuration values and the inverse view and inverse projection matrix, but also variables that toggle certain features on and off.
G-Buffer	Multiple buffers used to exchange data between different stages of the ray tracer.
Index Buffers	Multiple buffers which contain the index data for the geometry used in the ASs.
Vertex Buffers	Multiple buffers which contain the vertex data for the geometry used in the ASs.
Cluster Buffers	Multiple buffers which contain the clusters for all triangles from the ASs.
Cluster Visibility Table	Texture containing data with regard to cluster to cluster visibility.
Light Visibility Table	Texture containing data about which lights are visible from which cluster.
Light List	Contains an array of all the lights in the scene. For each light, different attributes of the light are saved (position, color, etc.).
Blue Noise Texture	Texture array with blue noise textures.
Sky Map	Cube map texture containing the sky texture used in the miss shader.

Table 6.2: All the ray-tracing specific attachments we use.

Another important part are the descriptor sets that need to be bound in order to access our buffers. We use two different sets. The first set contains a dynamically sized array of image samplers. This means we have access to all currently loaded textures. This descriptor is also used by the rasterizer pipelines. The second descriptor set is for ray tracing only, and contains all the data we need for ray tracing. Since some attachments have different buffers depending on the current image in the swapchain, we create one descriptor set for each swapchain image. This allows us to bind a different set each frame, instead of updating one set which is slow. The content of the descriptor set is listed in Table 6.2.

The ray payload we use for the communication between shaders is shown in Table 6.3:

Ray Payload	
`vec2 barycentric`	The barycentric coordinates of the triangle that was hit.
`uint instanceID`	Instance ID for the AS instance that was hit.
`uint primitiveID`	Primitive ID which is the index of the triangle that was hit inside the AS.
`float hitDistance`	Distance to the triangle that was hit.
`vec4 transparent`	Summed up transparency along the ray.
`float maxTransDistance`	Current max distance for the transparent calculation. This value is used to decide in which order the textures from transparent objects need to be blended.

Table 6.3: The ray payload we use to transfer data between the ray generation, any hit, closest hit and miss shader.

Once an any hit, closest hit or miss shader is called, the information of the hit is written to the payload and passed to the ray generation shader for further use.

6.3 Primary Ray Stage

The primary ray stage is the first ray-tracing stage that gets executed and is used to find the closest opaque object. In order to do this correctly, we need two rays. The first ray is used to find the distance to the closest opaque surface and the second one is basically the same but with the maximum distance set to the distance we got from the first ray. This is needed, since we want to use the any hit shader for completely transparent objects that do not require shading, (particles like smoke or transparent flame textures) between the eye and the first opaque surface (more details in the next section). If the maximum length for the ray exceeded the first opaque surface, the any hit shader would get called for transparent objects behind the first opaque object too. This can result in transparent objects behind the opaque surface shining through the opaque surface. See Figure 6.1 for an example of such a scenario. We use the flag `gl_RayFlagsCullNoOpaqueNV` for the first of the two rays, which skips all surfaces that are not opaque and thus saves some performance. A small offset needs to be added to the distance we get from the first ray in order to compensate numerical errors and make sure the second ray reaches the opaque surface again. Once the second ray is shot, we take the information from the surface we hit and read the object data (pos, uv, normal etc.) from the according buffers. In a later section we will talk about the details how these buffers are accessed. The data is then written to the G-Buffer, which consists of the following buffers as described in Table 6.4:

G-Buffer	
`albedo`	Color of the object at the pixel position.
`position`	The position of the pixel in world space. The 4th component contains the cluster id for the pixel.
`normal`	The normal of the pixel in world space.
`reflection/refraction`	Reflection and refraction color for areas that contain such surfaces.
`transparent`	Contains the color for transparent surfaces located between the view position and the first opaque surface.
`object`	Contains the barycentric coordinates, primitive ID and instance ID of the object at the pixel position.
`motion`	Contains the motion vectors from the previous frame to the current frame.
`view/material`	The ray direction with origin from the camera position that was used for this pixel. The 4th component contains the material ID of the object hit.

Table 6.4: The G-Buffer we use.

Algorithm 6.1: Pseudocode for blending transparent textures along a ray

```
Function alpha_blend(vec4 top, vec4 bottom):
    return vec4(top.rgb + bottom.rgb * (1 - top.a), 1 - (1 - top.a) * (1 - bottom.a))
Function any_hit_main():
    // premultiply alpha
    texture.rgb *= texture.a
    if ADD_BLEND then
        payload.transparency.rgb += texture.rgb
    else if ALPHA_BLEND then
        if payload.max_transparent_distance < gl_HitTNV then
            payload.transparency = alpha_blend (payload.transparent, texture)
        else
            payload.transparency = alpha_blend (texture, payload.transparent)
        end
```

6.4 Transparency

Semi-opaque objects (which require shading) are handled in the second ray stage (more details in the next section). For completely transparent objects (which require no shading), we make use of the any hit shader. When we trace a ray, we collect all the completely transparent surfaces along the ray with the any hit shader and store their color in a separate variable. We use the any hit shader only for surfaces that do not require shading, since we can not shoot shadow rays from an any hit shader, and collecting all

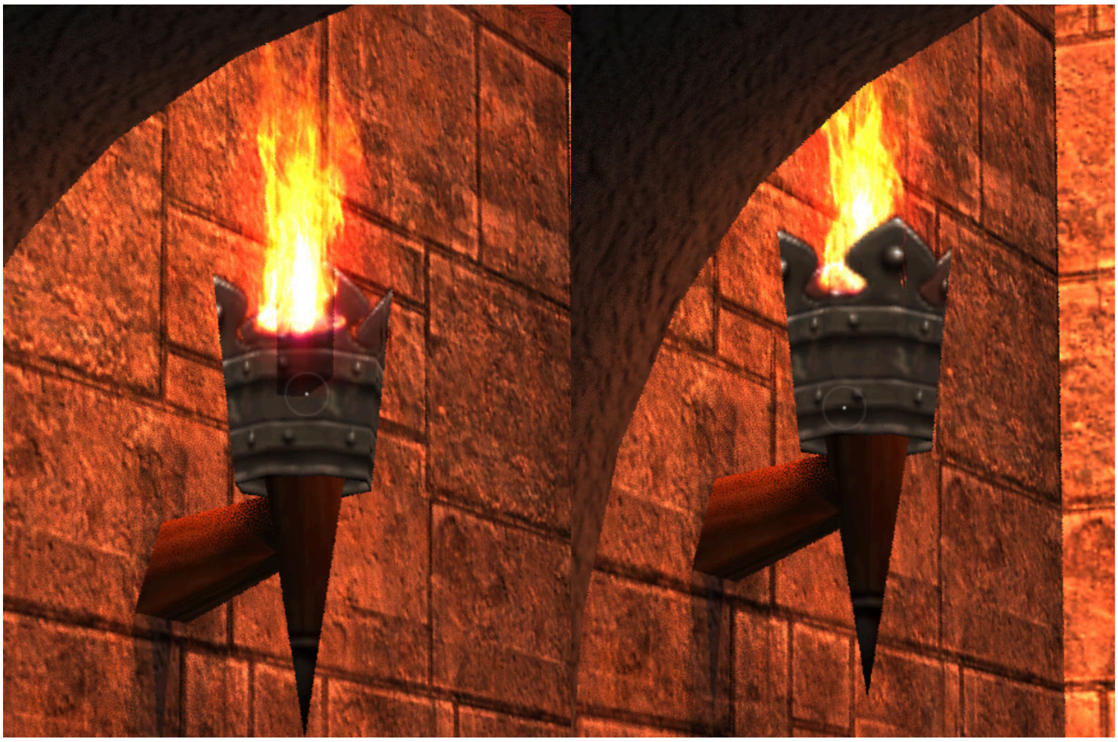


Figure 6.1: The left image shows a scenario where the transparent surface (flame) behind the metal parts leaks through since the order of the objects for which the any hit shader gets executed is random, and therefore objects behind a opaque surface can also be included. The right images shows the scenario where the maximum ray length is the exact length from the camera to the metal ring and therefore only objects between the camera and the metal ring are considered for the transparency calculation.

necessary positions, normals, etc. along the way, required for shading, is not practical. In order to execute the any hit shader only for completely transparent objects, we used a different instance offset, in comparison to the opaque objects, for indexing into the shader binding table. For opaque objects, only the closest hit shader is called, for transparent surfaces, only the any hit shader is called. We use two different operations for adding up the transparency. One is `ADD_BLEND`, which adds the colors, and the other one is `ALPHA_BLEND`, which tries to blend the colors by their alpha values. See Pseudocode 6.1 for the code used for these operations. Since the order of the objects for which the any hit shader gets called is random, we have to differentiate if a surface is in front or behind the previously hit transparent objects. We do this by saving the distance to the furthest away already processed transparent surface and compare it with the distance to the current transparent surface hit. In the case of the `ALPHA_BLEND` operation, we have to swap the blend order if the distance to the current object is smaller/larger than the maximum distance saved. This strategy can produce wrong results when the ray intersects with three or more transparent objects and the any hit shader gets called for the objects in the middle last. For the use case of this thesis, quality and performance of this method were good enough, but if a more accurate blending is required, this strategy needs to be improved further.

Once we accumulated all the colors from the transparent surfaces along the ray, we save the final value to the G-Buffer. In the composition stage, the transparent surface colors are blended on top of it.

6.5 Secondary Ray Stage

In this stage we trace all the additional rays required by certain materials like glass, mirrors or semi-opaque objects. This stage uses recursion inside the hit shader to trace the ray until an opaque surface was hit, or the maximal depth count is reached. Once an opaque surface is hit, its data is passed to the ray-generation shader, where the data gets saved to the G-Buffer for the affected pixels.

Semi-opaque objects, compared to other transparent objects, require shading, which is complicated to do with the any-hit shader approach discussed in section 6.3 and 6.4. These surfaces are problematic, since they are basically opaque surfaces and thus require shading, but have certain areas that are transparent (see Figure 7.5 for an example). This is the result of older games trying to skip the detailed geometry of an object and use textures instead to fake it (e.g., a fence with transparent holes as texture, on a rectangular mesh). For these objects, we check the alpha channel and if it is below 0.5, we continue the ray in the same direction. If it is above 0.5, we have hit an opaque part of the surface and stop (0.5 was the value that worked best).

In case of a mirror, we reflect the ray with the help of the normal of the object and continue the ray in the new direction.

For glass or water, the ray splits into one reflection and one refraction ray.

For these pixels, we overwrite the data from the first ray stage inside the G-Buffer with the new hit data from this stage. In the next stage, this information is then used for shading.

6.6 Direct Illumination Stage

This stage is used to calculate the shading for the surfaces hit in the primary ray and secondary ray stage. For the calculation of the shading we need the albedo, position, normal and the cluster for the pixel, which we can read from the G-Buffer. Then we iterate over all the lights that are visible from the cluster of the pixel and shoot a shadow ray in the light's direction. Since we only want to check if there is anything between the point and the light source, we can use the ray flags `gl_RayFlagsTerminateOnFirstHitNV | gl_RayFlagsSkipClosestHitShaderNV` to terminate the ray as soon as any surface is hit, resulting in a performance increase. The result is 0 if an object was hit (spot is in shadow), or 1 when no objects was hit (spot is illuminated). Basically the formula cancels out the shading term if the light is not visible because it is multiplied with 0. If the denoiser is activated, we can use just one random light source per pixel to get a noise-free image. See Algorithm 6.2 for a simplified version of this procedure.

Algorithm 6.2: Pseudocode for calculating shading

```
Function calcShading():
    vec3 shadeColor
    for each light visible do
        float lit = shootShadowRay()
        lit *= numLights / numLightsUsed
        shadeColor += lit * LdotN * lightIntensity
    end
    return (shadeColor * albedo / M_PI)
```

6.7 Accessing Object Data in Shader

Since we have multiple ASs, accessing the corresponding data is not trivial. The buffer we describe in this chapter can contain more values than listed, but for clarity we only list the necessary data. When a ray hits an object, we get the information shown in Table 6.5.

Build In Variables	
`gl_InstanceID`	The index of the instance in the top level AS
`gl_PrimitiveID`	The index of the triangle starting from the beginning of the current instance
`barycentric`	The barycentric coordinates of the hit point inside the triangle

Table 6.5: The data provided by the any hit and closest hit shader which is used to describe a intersection with a triangle.

With this information, we need to access our buffers containing the instance data and the index, vertex and cluster data. With the instance id we can access the instance data buffer which contains information for accessing the correct data from the index, vertex and cluster buffers. In Table 6.6 the content of the instance buffer can be seen.

The type variable describes whether the data of the object is contained in the `world_static`, `world_dynamic_data`, `world_dynamic_as`, `entity_static` or `entity_dynamic` buffer. The index buffer contains an array of unsigned integers that are used for indexing into the vertex buffer. With the formula `offsetIDX + (primitiveID * 3) + 0|1|2` we get the correct triangle indices from the index buffer. With the indices from the index buffer we can read the correct data from the

Instance Buffer	
`modelmat`	Model matrix used to transform the vertices to world coordinates
`type`	Describes which index, vertex and cluster buffers contain our data (Static, Dynamic, etc.)
`offsetIDX`	Offset for the index buffer
`offsetXYZ`	Offset for the vertex buffer
`cluster`	The cluster this instance is inside (only used for entities)

Table 6.6: This data is set for each individual instance in the top level AS.

vertex buffer. The following formula gives the correct offset from the beginning of the buffer: `offsetXYZ + idx_(0|1|2)`. The content of the vertex buffer can be seen in Table 6.7.

Vertex Buffer	
`pos`	Vertex position
`normal`	Normal of the vertex
`color`	Color of the vertex
`uv`	UV coordinate for the vertex
`texture idx`	Texture of the object the vertex belongs to
`material`	Material of the object the vertex belongs to

Table 6.7: The vertex buffer contains all the object data needed after a ray intersects a triangle.

In order to get the hit position, UVs and other data of the hitpoint inside the hit triangle, we need to interpolate the values we got from the vertex buffer with the help of the barycentric coordinates. The following formula gives use the correct hit position: `v0 * bary.x + v1 * bary.y + v2 * bary.z`. We do this for all values that require interpolation and save them in a struct for further use. The cluster index can be read from the according cluster buffers with an index calculated with the following formula: `offsetIDX + primitiveID`.

6.8 Combination of Multiple Textures

Since objects can have multiple textures in addition to the color values assigned to each vertex, we had to implement a functionality to blend them accordingly. For each of the textures of the object, we have information about the blending strategy we need to use. This includes if the texture requires vertex colors and if it needs to be blended by alpha, addition or multiplication. Each texture is defined by an index and can be accessed from the texture array. To get the final color, we go through all the available textures of the object, read them with the according texture indices and UVs and sum them up based on the contained blend information. Algorithm 6.3 shows a simple pseudo code for this procedure.

Algorithm 6.3: Pseudocode for reading textures

```
Function getTexture():
    vec4 result = vec4(0.0)
    for each texture of object do
        vec4 tex = texture(idx, uv)
        if color then
            tex *= color
        if blend then
            result = blend(result, tex)
        else if add then
            result += tex
        else if multiply then
            result *= tex
    end
```

6.9 Lights

A big problem is that Quake III has no light sources. Therefore, we needed to define them ourselves. We used the geometry of objects that we thought would make an acceptable light source. We split each light source consisting of more than one triangle into multiple triangle-shaped light sources. The data found in Table 6.8 defines one light source.

Light	
`vec3 pos`	Position of triangle vertex 0
`uint offsetIDX`	Offset for the indexbuffer
`uint offsetXYZ`	Offset for the vertexbuffer
`int cluster`	Cluster of the light
`vec4 color`	Color of the light
`vec3 normal`	Normal of the light
`float size`	Size of the light
`vec3 AB`	Direction from vertex 0 to vertex 1
`vec3 AC`	Direction from vertex 0 to vertex 2

Table 6.8: Data of one light source.

When we calculate the lighting for a given position, we take the cluster of this position, and look up the number of lights visible from this cluster (first column of the light texture; see table 4.2). Then we iterate over all lights from 1 till the number of lights and calculate the shading by shooting shadow rays in the direction of the lights.

6.10 Denoiser

The denoiser we use is called A-SVGF [SPD18]. The stages of the denoiser can be seen in Table 6.9 (listed in order of execution).

A-SVGF Stages	
`RNG`	Calculate the random seed for the current frame.
`Forward Projection`	Try to project the information from the previous frame to the current frame.
`Gradient`	Calculate the gradient based on the luminosity from the previous and current frame.
`Gradient À-Trous`	Filter the sparse gradient image with a guided filter to produce a dense result.
`Temporal Filter`	Calculate the weights for the accumulation factor between the previous and current frames.
`À-Trous`	Construct a dense lighting information from a sparse lighting information.
`TAA`	Clean up the image.

Table 6.9: A-SVGF stages listed in order of execution.

The buffers used by the denoiser can be seen in Table 6.10.

A-SVGF Buffers	
`RNG`	Contains the current seed for each pixel.
`Gradient Sample Pos`	The position we used from each 3x3 area for forward projection.
`Position Forward`	Forward projection of the previous position.
`Object Forward`	Forward projection of the previous object data (instance id, primitive id and barycentric coordinates).
`Moment Histlength`	Contains the change in luminosity and the history length.
`Color History`	Contains the accumulated illumination.
`Grad Atrous A/B`	Two buffers are used for ping/bong filtering of the sparse gradient image.
`Atrous A/B`	Two buffers are used for ping/bong filtering of the sparse and noisy reconstructed illumination.

Table 6.10: Buffers that are needed by the denoiser.

Since the implementation is mostly identical to the reference, please take a look at the paper for further details [SPD18].

6.11 Composition

This final stage is used to produce the final output image. It combines different images based on the current configuration. This stage also blends the colors from the transparent surfaces onto the final output image.

6.12 Temporal Anti-Aliasing

This optional stage is executed after the composition stage and uses the results from the previous frame in combination with motion vectors to further clean up the current image. At first, we look at the color distribution around the current pixels in a 3x3 window in the current frame (we calculate mean and standard deviation). Then we use the motion vector to get the correlating color from the previous image using bicubic filtering. We clamp the color we get from the previous frame to the range of the distribution $\mathcal{N}(\mu, 3\sigma)$ we calculated in the beginning and add 0.1 of it to the current pixel color.

CHAPTER 7

Evaluation

7.1 Hardware

Since RTX is still a new technology, the computational power for accelerated ray tracing is quickly exhausted, especially on the lower end RTX GPUs like the RTX 2060. Therefore, we used an RTX 2080 Ti, which is the fastest consumer-grade GPU currently available that is based on the Turing architecture, and provides us with enough headroom for our analysis. The complete list of the used hardware can be found in Table 7.1. This implementation should also run on some older Nvidia GPUs since Nvidia offers a software fallback for RTX Ray Tracing on non-Turing cards, but we have not tested this, so we can no guarantee it.

In terms of extension and feature requirements for the GPU, there is not much to say since only the Turing-generation GPUs support RTX Ray Tracing, and all those cards also support the latest extensions and features of Vulkan$^{\text{TM}}$.

CPU	Intel i7 9700k @ 4.90 GHz
GPU	GeForce RTX 2080 Ti with 11 GB GDDR6
Memory	32 GB DDR4 @ 3200 MHz

Table 7.1: Hardware of the computer used for the evaluation

As mentioned, we use a very powerful GPU, but it is still very easy to reach the GPUs limits with ray tracing. Even with a game like Quake III, which is from 1999 and should easily run on modern hardware, these limits are clearly visible when no denoiser is used. The models and other assets of Quake III have a low complexity since it is an older game but it can still be challenging to achieve 60 frames per second at a resolution of 1920x1080 or 2560x1440 without a denoiser. A detailed analysis of the performance follows in the next sections. This means that there is a lot of potential for research and development in terms of hardware but also algorithms in order to achieve good performance with even more demanding content.

Some configurations of the ray tracer can impact the performance noticeably, like reflection and refraction depth, bounce depth in case of path tracing or number of shadow rays. Since these settings are also responsible for the quality of the rendering result, it is important to find a good middle ground.

7.2 Visual Results

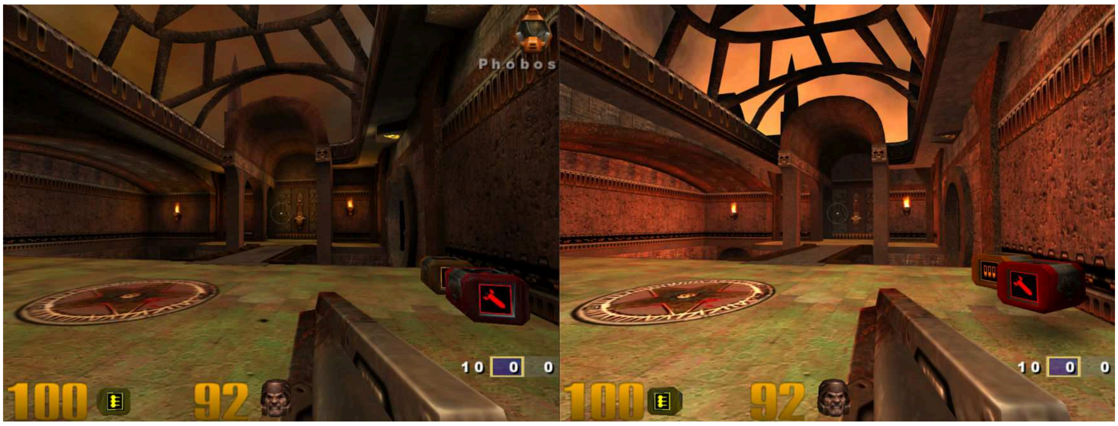


Figure 7.1: The left shows the original Quake III rasterizer. The right image is the result produced by our ray-tracing implementation. The scene is taken from the map q3dm2.

In this section, we will look at the visual result of our implementation. The first big question is how does our implementation compare to the original graphics of Quake III. When we take a look at Figure 7.1 we can see a comparison of a scene rendered by the original Quake III rasterizer and our ray-tracing implementation. The first difference we see are the soft shadows added to the scene, like the shadows below the ammo boxes. The second difference is that the shading produced by the ray tracer seems a bit different in terms of color/lighting compared to the original renderer. The reason for this is that Quake III does not have explicit lights defined that we can use. Quake III gets all its shading from light maps, therefore we had to define our own lights. Since we do not know how the lights were configured for the calculation of the original light maps, we cannot reproduce the exact same lighting configuration. This results in a slightly different shading of the scene and makes the comparison difficult. In order to mitigate this, we would have to create a custom map where we know the exact lighting configuration, but this would be beyond the scope of this thesis.

Figure 7.2 shows the difference between the output of the ray tracer with the denoiser disabled (use of all 146 lights) and the case where TAA is additionally enabled, both in comparison to the ground truth produced by accumulating 1500 frames from the paused game. The left image contains noise from the soft-shadow effect. The noise is not very

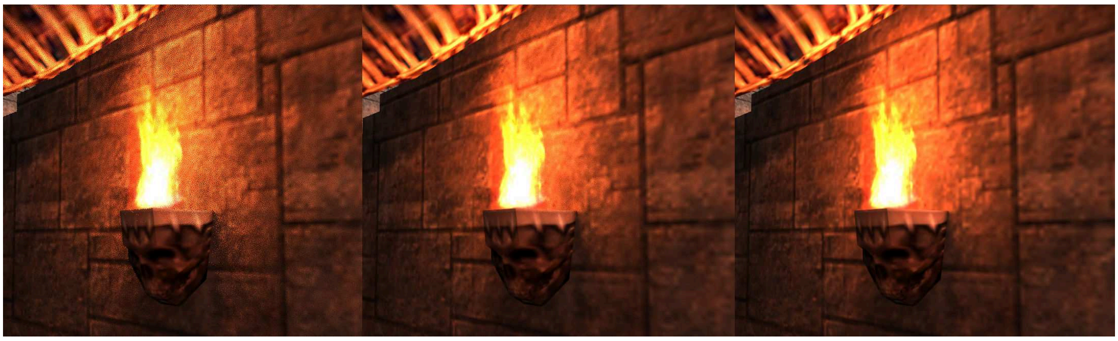


Figure 7.2: The left shows the ray-traced result without a denoiser and all 146 lights per pixel. The middle image shows the same but with TAA activated. The right image is the accumulation of 1500 frames with the same configuration as the left image. The scene is taken from the map q3dm4.

noticeable and is even less noticeable when the images are in motion. We can also enable TAA to achieve a result with no perceivable noise, which is close in terms of quality to the accumulated image. The middle is the result with TAA activated, and the right image is the accumulated result.

Figure 7.3 shows multiple shots of the same scene with different settings. The top left image shows the scene with illumination disabled, which means removing all the light maps. The other three images show the scene when we activate ray-traced illumination. The top right shows the scene when we use no denoiser and all the lights for each pixel. In the bottom left only one light source is used per pixel, which results in a lot of noise. This configuration is the input for the denoiser, with the result being visible in the bottom right image. When we compare the first image with no illumination with both images on the right, it is clear to see that the quality improves when ray-traced illumination is activated. Especially the shadows from the pillars, or the shading on the weapon, floor and wall, make the scene more realistic. When we compare the results on the right with each other we can see a bit of a difference when using a denoiser and when using no denoiser. The result from the denoiser is a bit more washed out, which means it removes some details of the shadows, and makes them look more subtle. For example, the shadow from the left pillar is less pronounced and the penumbra seems a bit large compared to the image where we used no denoiser. In Section 7.4 we will take a look at another problem that is connected to this observation. Also, the difference in color between areas which are in shadow and which are fully lit is much lower. This is probably the result of the denoiser using edge-preserving filters to fill the gaps of the little data it has each frame. This results in a lighting information closer to the mean of the surrounding pixels.

Randomly selecting one light from all the lights of the scene, or only from a smaller subset can influence the amount of noise that is contained in the output image. This can carry over to the denoiser and can influence the quality of its output. In Figure 7.4 we can see the difference between light culling off/on. The top two images show the result

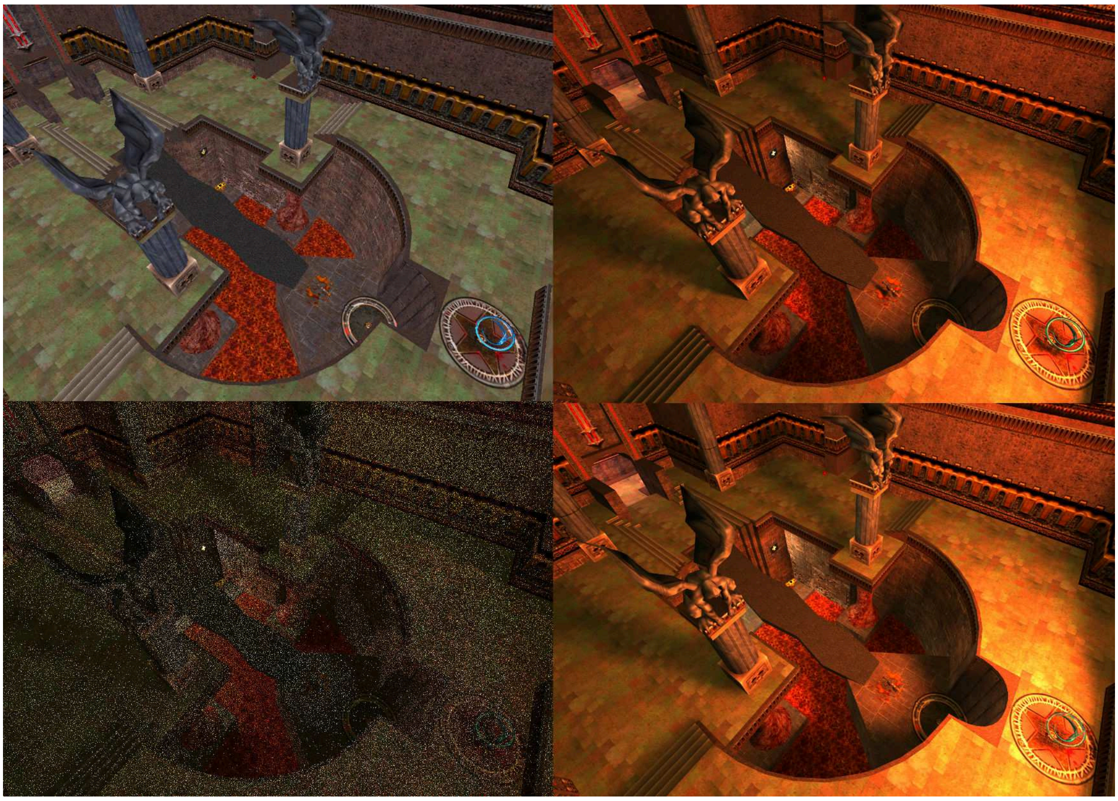


Figure 7.3: The top left image shows Quake III without light maps, which is equal to no illumination. Bottom left shows Quake III rendered with the ray tracer and one light per pixel. Top right is the ray traced result when using all 106 lights per pixel and no denoiser. The bottom right is the ray traced result with one light per pixel and a denoiser. The scene is take from the map q3dm7.

when we do not use light culling and the bottom row shows the results when light culling is enabled. All four images use one random light per pixel. We can see the top left image contains more noise than the bottom left image, which makes the top image look darker since more lights are selected that do not contribute to the lighting of the pixel. With this information, the denoiser also produces a slightly darker and less detailed result. The image from the denoiser on the bottom right is overall brighter. Another difference that is present when light culling is deactivated but cannot be displayed by a screenshot is the slight alteration of shading produced by the denoiser between each frame. It is very noticeable during run-time, through randomly appearing dark spots. The reason for this is that we have hit too many useless lights and thus, the denoiser does not have enough data to correctly reconstruct the lighting in a given area. This is less pronounced or not visible at all when light culling is on. If it is not explicitly mentioned, we always have light culling enabled.

The last visual aspects we will talk about are transparent surfaces like glass and water.

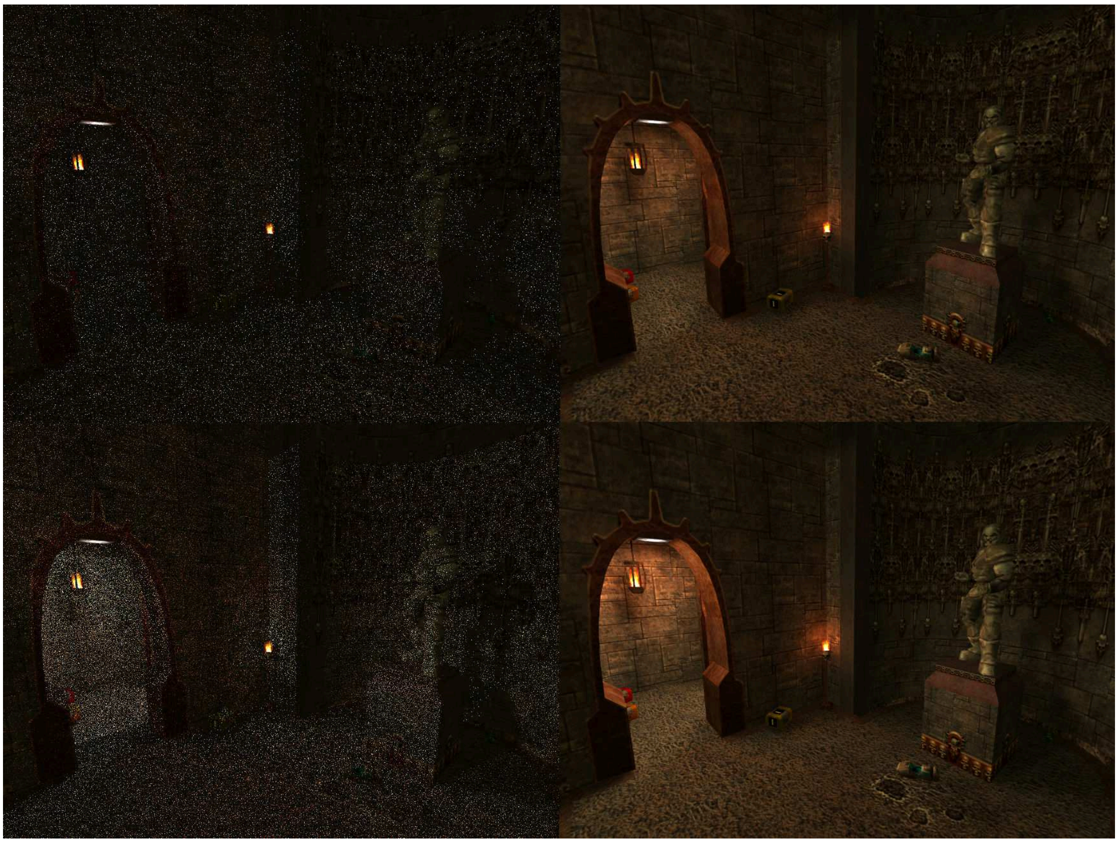


Figure 7.4: The top left image shows Quake III with the denoiser deactivated, one light per pixel and light culling off. Bottom left shows Quake III with the denoiser deactivated, one light per pixel and light culling on. Both images on the right show the images from the left processed by the denoiser. The scene is take from the map q3dm9.

Figure 7.5 shows a semi-opaque model inside a glass tube. This is a difficult situation to handle since we need two refraction rays to reach the wall behind the glass tube, and the first refraction ray can get reflected multiple times inside the tube. It is possible to just use one of the two rays at each branch and try to remove the noise with a denoiser, but for a better comparison with the ray-tracing configuration where we use no denoiser, we traced all the rays recursively. Our maximum reflection/refraction depth is 5, which means we can look through up to 5 glass surfaces positioned behind each other.

7.3 Performance

In this section we take a look at the performance and how different settings influence it. One thing to keep in mind is that, since RTX Ray Tracing is still a new topic, our implementation certainly has room for improvements. The optimizations that are part of our implementation are light culling and a denoiser.

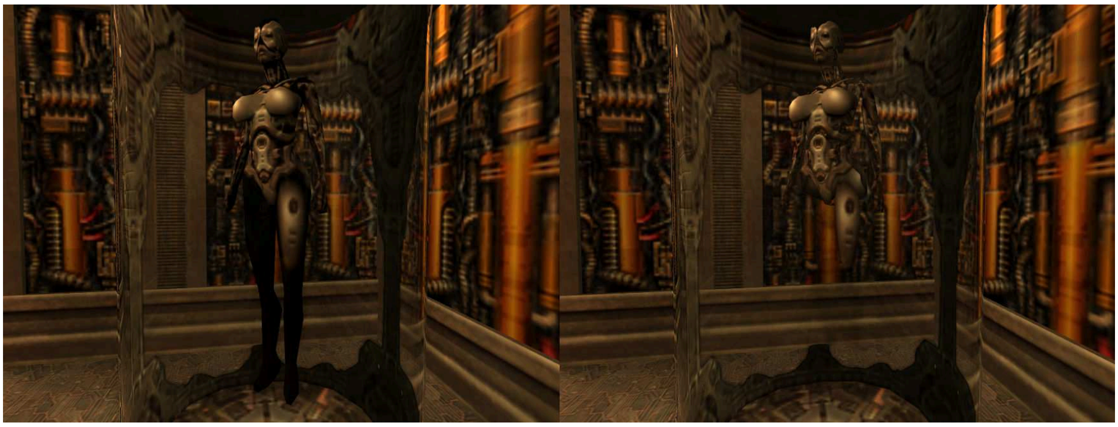


Figure 7.5: Glass tube with a semi-opaque model in between (from map q3dm0). Left image shows the scene with the secondary ray stage functionality for semi-opaque model deactivated.

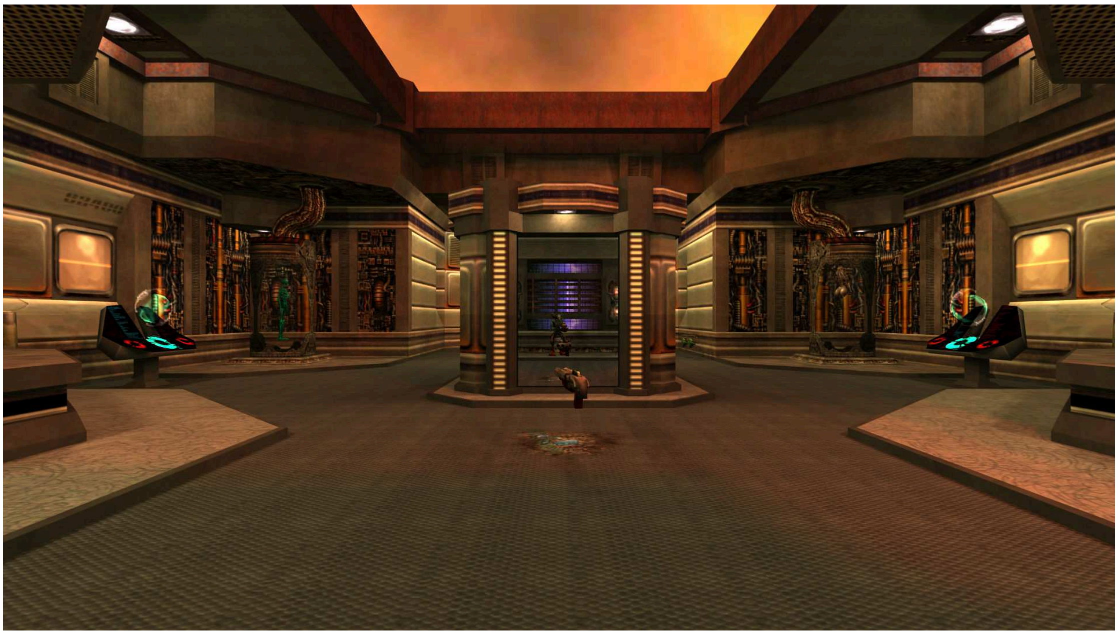


Figure 7.6: The scene we used for performance analysis with a mirror in the middle and two glass tubes on the sides. Scene is from the map q3dm0.

The two main aspects the performance of our implementation depend on are, the number of lights we use per pixel and the resolution we use for the calculation of our image. Since distributed ray tracing has no indirect lighting stage, the path depth has no impact on the performance in most cases. The only exceptions are glass and mirror surfaces, which only heavily impact the performance if there are multiple such surfaces next to each other. In the case of no denoiser, we have no other option then to use all lights to get a nearly noise free image. This highly affects the performance in a negative way. When we use a denoiser, we can resort to only one light per pixel, which results in a performance increase even with the denoiser using up to 2ms of the frame time. As reference, in order to get 60 frames per second, a frametime of 16.6 ms is required. We tested the performance of our implementation across all the commonly used display resolutions from 1280x720 pixels (720p) up to 2560x1440 pixels (1440p). The test scene we use is from the Quake III map q3dm0 at the initial player position (see Figure 7.6). Our observations can be seen in Figure 7.7. The grey bar describes the run where we used no denoiser and shot shadow rays to all 135 light sources. The orange bar describes the case where one shadow ray and no denoiser was used and the blue bar describes one shadow ray and an active denoiser. Without a denoiser we reach the limit at 1440p where the frametime goes above 16.6 ms. For all the resolutions where we used a denoiser, the results are below 16.6 ms (above 60 FPS). Even tough in most cases, using no denoiser can achieve more than 60 FPS, the framerate can drop way below that in areas where more lights are used. When using a denoiser, we have the advantage of limiting the maximum number of lights per pixel to one, which results in a stable frametime. On the other hand, in the mentioned areas the variance can increase, since the probability of selecting a certain light source decreases, which can result in a lower-quality denoising result. Figure 7.8 shows how much time the denoiser needs each frame.

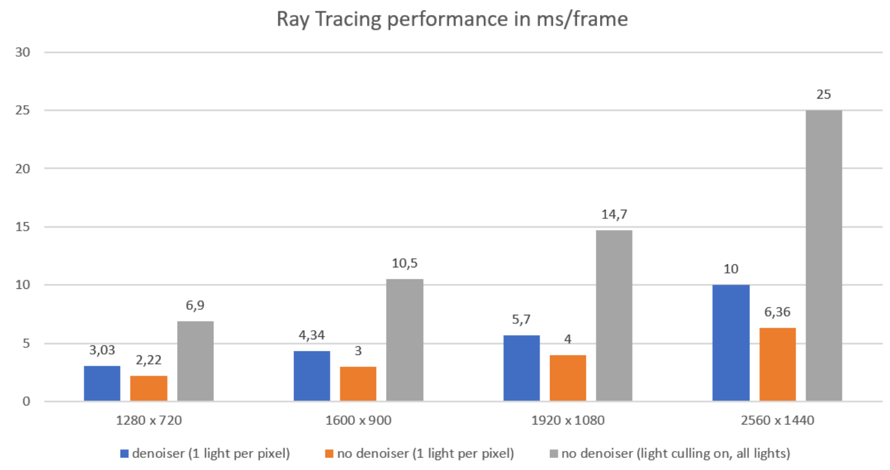


Figure 7.7: Ray-traced Quake III performance across multiple common display resolutions when using a denoiser compared to using no denoiser.

In order to analyze the impact of our light culling implementation in terms of performance, we render our scene with the denoiser deactivated and shoot shadow rays to all 135 lights. When we do not cull any lights, we have to test all lights in the scene for visibility. This results in many shadow rays traced that do not contribute to a pixels shading. Figure 7.9 shows the comparison of light culling activated and deactivated. In the case where light culling is activated, most of the lights that are not visible from the given triangle are not

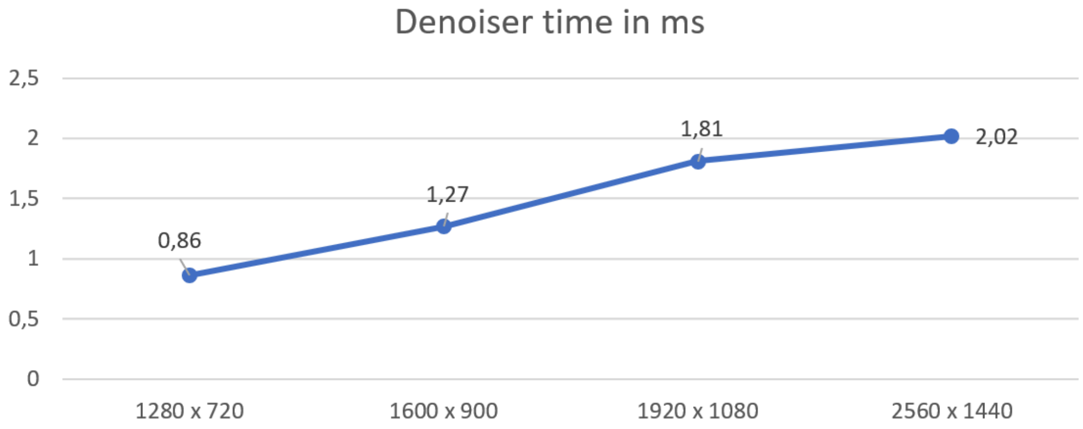


Figure 7.8: The time the denoiser needs every frame at multiple resolutions.

considered for the lighting calculation. This results in an increased performance of more than double the fps. When we use a denoiser, light culling reduces the variance of the random distribution for selecting a useful light and thus improves the visual quality and stability.

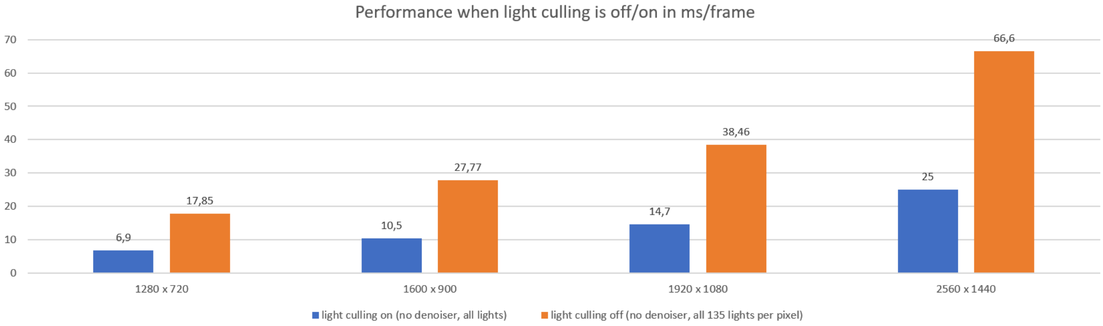


Figure 7.9: The difference in performance when all 135 lights are used, in comparison to when we cull the lights that do not contribute to a pixel's shading.

7.3.1 Performance affected by SPP

By setting the amount of lights to consider per pixel to different values, we can change the amount of noise in the image. This can improve the visual quality but comes with a performance drawback. In Figure 7.10 this can be seen for different sample counts (without a denoiser) and a single result with 1 SPP + denoiser. We compare these results with the ground truth, which is a image accumulated from 1500 frames, with one SPP and denoiser/TAA disabled. We only included one image for the situation when the denoiser is on, since the difference between different sample counts is practically not visible. This can be seen quantitatively in the left graph of Figure 7.11, where we calculated the Structural Similarity Index (SSIM) between the ground truth and different sample counts. When a denoiser is used, the result is the same for all the different sample counts. In case of no denoiser, the SSIM increases with more samples per pixel. The last two data points show the case where we additionally activated TAA. When we use all lights in combination with TAA, we can achieve a visual result which is very close or even better when compared to the case when using a denoiser. This means it is possible to achieve visual results close to the ground truth without using a denoiser. In the case when a denoiser is used, TAA improves the quality even further, though there is no difference between using all lights or just one light except for performance. In terms of performance, there is a clear downwards trend with an increased sample count. The graph in Figure 7.11 shows this for sample counts from 1 to 16. Since in this case, sample count means the number of lights we consider per pixel, and not calculating the ray for one pixel multiple times, the decrease is not as drastic as one would expect. For instance, if we calculated each pixel four times for each frame in case of a traditional 4 SPP implementation, the FPS would drop by three fourths. This is not practical for real-time usage since we divide our FPS by the SPP and this would result in a huge performance loss. In our implementation, the performance loss from 1 SPP to 4 SPP is around a third (around 150 FPS (18 ms/frame) from 450 FPS (4 ms/frame)). See the left graph of Figure 7.11, which shows the drastic performance decrease in case of tradition samples. The ratio of gained visual improvement compared to the accompanied performance loss is better through this approach. The right graph of Figure 7.11 shows that the visual quality is slightly better when traditional samples are used compared to just light samples, but the performance loss with each additional sample is much more drastic (left graph).

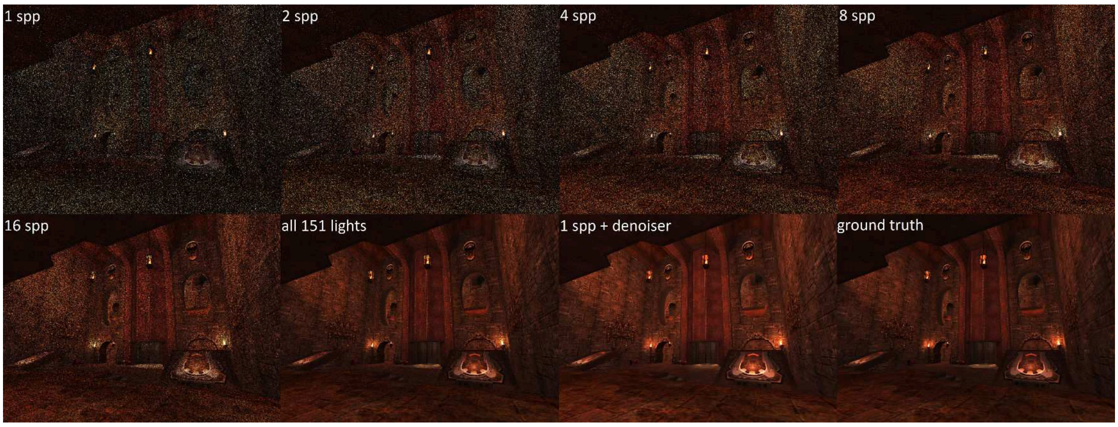


Figure 7.10: Top row from left to right 1 spp, 2 spp, 4 spp, 8 spp. Bottom row from left to right 16 spp, all 151 lights, 1 spp with denoiser, 1500 images accumulated at a resolution of 1024x768 pixels (ground truth). Light culling was always on. (from map q3dm15)

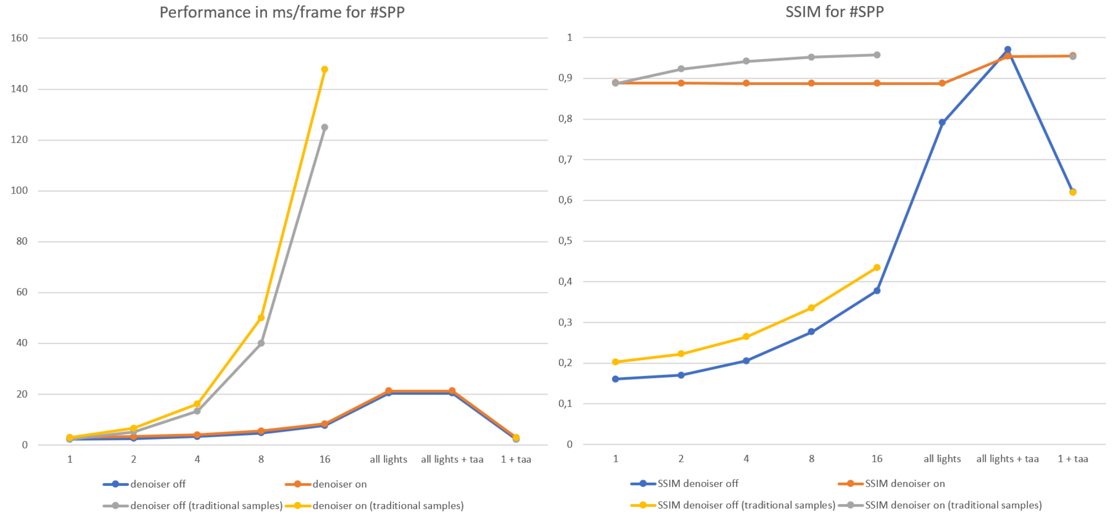


Figure 7.11: The left graph shows ms/frame for different sample counts. The right graph compares the visual results for different sample counts using SSIM. Each image is compared to the ground truth (accumulation of around 1500 images). Values are captured from the scene shown in Figure 7.10.

7.4 Discussion of Noise Origins

During the analysis of our implementation we found certain sources of noise that needed to be discussed in more detail. This was especially noticeable at a certain location within the Quake III map q3dm7. Here we have a large light source which is slightly offset into the back wall. If we limit the scene's lights to just this single light and turn off all effects except for soft shadows, we can notice noise within the penumbra but also outside of it in the fully lit areas. This noise is produced because of two reasons. First, some shadow rays get occluded by the protruding surfaces around the light source and second, because of the distance and angle-based dampening of the light's strength. When all this is removed, we get a result without any noise outside the penumbra. See Figure 7.12 for the visual comparison. This kind of problem is not present when using a denoiser or TAA.

7.5 Problems and Limitations

When no denoiser is used and light culling is on, we need to consider all lights that remain after light culling in order to get a noise-free result. The problem in this case is that, when moving through the scene, the frame rate can vary a lot in situations where the amount of lights changes a lot. Even if the FPS are above 60, these spikes can be noticeable through stutter. Figure 7.13 visualizes these frame time spikes for the map q3dm0 in the time span of around one minute. We could limit the maximum number of

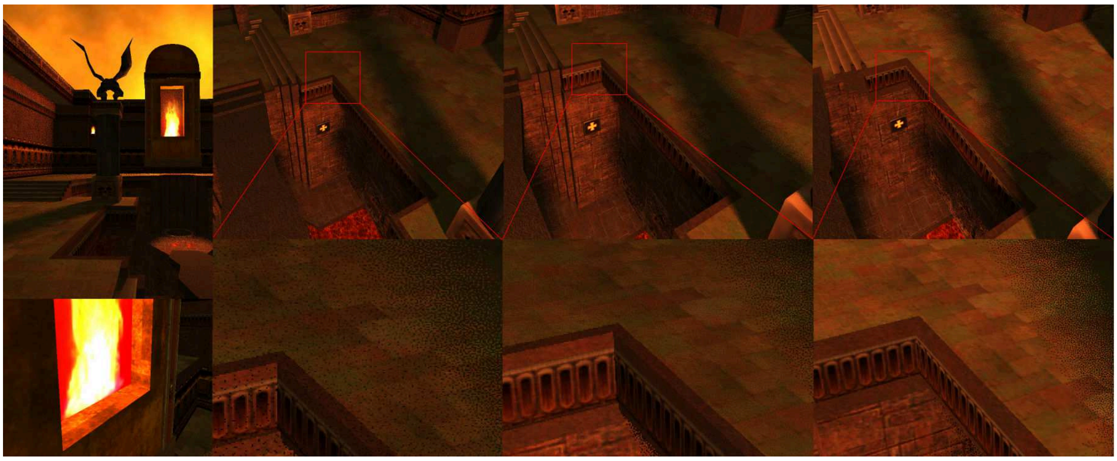


Figure 7.12: The first image shows our example scene. The second image shows the noise with the large light source being partially occluded by the protruding surfaces around it. The third image shows the noise with the light moved forward to remove all possible occlusions. The fourth image shows the noise with the L dot I term additionally set to 1. In all three images the penumbra can be seen on the right side, and the fully lit area on the left side.

lights to a value greater than one, but this would result in an inconsistent noise level between frames, scaling with the total number of lights an area has. Another solution would be to evenly distribute the light sources in the scene so that we have about the same amount of lights for each triangle. Another solution would be to disable light culling and use all lights all the time, which results in a constant performance decrease. The best solution would be to limit the lights per pixel count to 1 and use a denoiser.

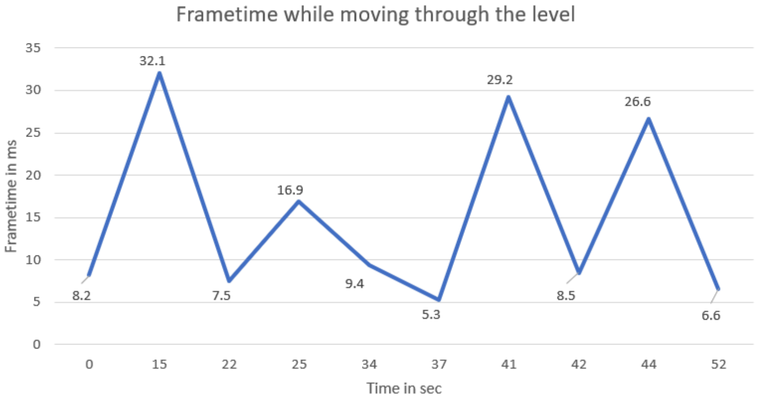


Figure 7.13: Frametime while moving through the level q3dm0 at a resolution of 720p with denoiser disabled and the use of all lights per pixel.

The same goes for reflections and refractions. If the camera is close to a reflective/refractive surface, resulting in a big part of the screen being covered by it, a lot of reflection and refraction rays need to be calculated. This reduces the frame rate depending on how much of the screen is covered with the surface. A solution would be to reduce the complexity for reflection and refraction rays by limiting the maximum depth. For materials like glass or water, when a solution is used that traces all reflection/refraction rays (in order to produce no noise), each additional reflection/refraction ray can create another pair of reflection/refraction rays, which results in a tree-like structure of dependency. For two glass panes next to each other, this can result in more than 4 rays per pixel. A solution would be to randomly decide at each hit if we want to spawn a reflection or refraction ray and only follow one ray. This would be similar to the path-tracing approach. Since we skip information with this method, the result will contain noise. Denoising such a situation is not trivial and requires additional adjustments.

Another problem in our implementation is that it is difficult to adjust the strength of the lights in a way that certain areas are not underexposed, while other areas are not overexposed (this is also reinforced by Quake III not providing predefined lights, therefore we have to define our own lights). In this case, the use of a tonemapper can help with achieving a shading which does not produce such extreme values.

One problem produced by the denoiser can be seen in Figure 7.14. The denoiser blurs and softens shadows when directly compared to the ground truth. This effect increases the further away the object is from the camera. The reason for this is that objects that are further away have less temporal history because of the fewer pixel they occupy on the screen. Since the denoiser makes use of blur filters to produce dense results from sparse inputs, fewer pixels produce a less detailed reconstruction of the shading as shown in Figure 7.14. The left and middle image show the same object from a close and far distance with the denoiser activated. We can see that the middle image loses a lot of detail compared to the ground truth (right images), which we acquired through the accumulation of multiple images without a denoiser. In Figure 7.3, this problem can also be seen.

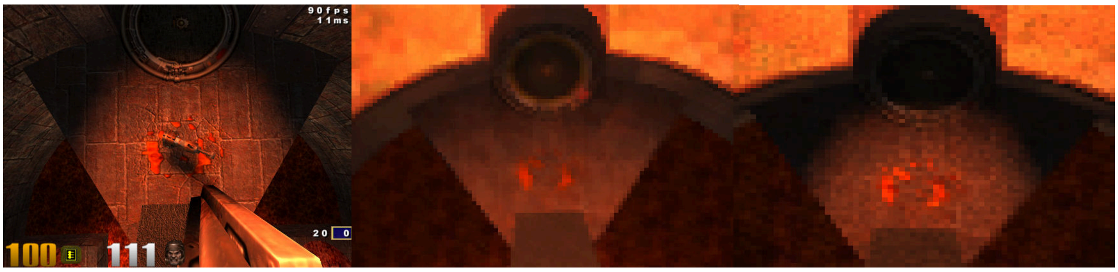


Figure 7.14: Left image shows the scene with the denoiser on, from a close distance. The middle image shows the result when the denoiser is on and the camera is further away from the object. The right image shows the accumulated image (no denoiser) from the same distance as the middle image. The shadows in the middle image are much lighter compared to the left and right image. The scene is taken from the map q3dm7.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

CHAPTER 8

Conclusion and Future Work

Overall, the results we achieved are promising for the future of completely ray-traced games with the help of more capable hardware and improved implementation strategies. Currently, the low complexity of older games is essential for the feasibility of a fully ray-traced game. The outdated architecture of these old games, on the other hand, makes it hard to implement ray tracing at its fullest potential.

Implementing ray tracing without a denoiser or light culling is not advised, since the disadvantages are very prominent. The only scenario where an implementation without a denoiser can be useful is when development time is short, a light-culling functionality that is perfectly adjusted for each triangle is available and the lights are evenly distributed across the scene to decrease the occurrence of frametime spikes. When no denoiser is used, activating TAA for the final output can increase the visual quality a lot. Even though using a denoiser is very helpful, no current denoiser is perfect, and using a denoiser is always an exchange of quality for performance (only one light per pixel is required to produce a noise-free image but quality compared to the ground truth is lower). The quality of the light culling can have a big impact on the performance of the denoiser. The more unnecessary lights are culled, the more effective the denoiser can operate, and thus, can further close the gap between the denoiser's output and the ground truth. Our proposed light-culling strategy removed many discrepancies between frames and improved the general quality of shading/shadows of the final output. Also, the performance gains when using all lights after culling compared to all lights without culling were significant. Still, the most exciting results were when we combined the denoiser with light culling and subsampling of the lights. No other combination could produce such a good quality/performance ratio and the analysis shows that this is a feasible way to implement a completely ray-traced legacy game. Since the PVS of Quake III was not originally designed for such a precise application, we could probably get even better results with a more appropriately designed PVS.

All in all, the implementation of a real-time ray tracer is a non-trivial task and has its upsides and downsides compared to a rasterizer. The upside is that certain effects like reflections are straightforward to implement. On the other hand, there are a lot of knobs to adjust. For example, getting multiple transparent surfaces to blend correctly along a ray can be trial and error depending on the complexity of the scene and the consistency of the data. Considering ray tracing at the beginning of the engine design can help a lot with avoiding such problems.

From a visual point of view, it is difficult to produce better results with a real-time ray tracer without introducing too much of a performance hit in comparison to a modern rasterizer, since rasterizers of modern engines set a very high bar in terms of visual quality. In older games, where the geometry is very simple, it is possible for a ray tracer to produce better results than the original renderer, tough in comparison with a state-of-the-art game, it still cannot produce an equal quality. Once the hardware and software has evolved further, we will see state-of-the-art games that produce better results with a ray tracer than a rasterizer.

Possible future work and improvements are, on the one hand, to iron out some bugs that are still present, and on the other, further improve the implementation (e.g., add indirect illumination and other effects). Additionally, the code will be hosted on GitHub and will be available for everyone to use and modify. In general, we hope this thesis can help people with a similar goal in mind, and encourages the use of ray tracing in a real-time renderer.

TU Bibliothek WIEN Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Figures

1.1 Ray-traced Quake III. 3

2.1 Illustration of a ray path in case of a Whitted ray-tracer. The ray gets reflected and refracted multiple times till it hits a opaque surface and stops. R = Reflection Ray; T = Refraction or Transmission Ray; S = Shadow Ray; N = Normal; [wrt] 8

2.2 Approximation of π through the MC method with 27000 sample points. We are calculating the ratio between the area below the curve and the 1×1 square. This can be expressed as $\frac{\frac{\pi}{4}r^2}{r^2}$. Since we are only considering a fourth of a circle, the result is $\frac{\pi}{4}$ and has to be multiplied by 4 to get the approximation of π. 12

2.3 620 random 2D points calculated with the use of the Halton and Sobal sequence, in comparison to an ordinary random generator. [quae] 14

2.4 White noise vs blue noise texture when a blur is applied. The blue noise has a uniform blur since its values are more uniformly distributed than the white noise values. 15

2.5 The hemisphere defined by Ω that needs to the sampled for incoming light in order to calculate the shading of point x. 16

2.6 Top and bottom level ASs and how they are related. [nv] 19

2.7 Graph showing the flow of a ray-tracing call. [nv] 20

3.1 Denoising results from the approach presented by Moreau et al. (Left: before / Right: after) [MMBJ17] 21

3.2 Illumination results from the approach of Heitz et al in comparison to offline results. [HHM18] 22

3.3 First two images are 1 spp with uniform sampled lights and denoised with SVGF [SSK+17] over 5 frames. Next two images are 1 spp with BVH sampled lights and denoised with SVGF over 5 frames. The last image is the ground truth. [MPC19] 23

3.4 ATAA in comparison to FXAA and TAA. [MSG+19] 23

3.5 Million rays traced per second comparison of GTX1070 (left) and RTX2070 (right). [SGFV19] 24

3.6 NVIDIA RTX Tech Demo of Quake II (Left: with RTX / Right: original). [quaa] . . . 26

5.1 The diagram shows all the different parts of Quake III and how they are connected with each other. The server side uses one virtual machine while the client side uses two virtual machines. The engine provides system calls to the artificial intelligence module and the renderer module. [quac] . . . 34
5.2 The quick inverse square root hack used in Quake III, with modifications for x64. . . . 35
5.3 Different memory configurations in Vulkan. . . . 37
5.4 A cropped part of a texture from Quake III used for metal walls. In the upper part of the image, you can see the baked specular highlights, which cannot be easily removed. In the lower part, some baked shadows can be seen. . . . 38

6.1 The left image shows a scenario where the transparent surface (flame) behind the metal parts leaks through since the order of the objects for which the any hit shader gets executed is random, and therefore objects behind a opaque surface can also be included. The right images shows the scenario where the maximum ray length is the exact length from the camera to the metal ring and therefore only objects between the camera and the metal ring are considered for the transparency calculation. . . . 46

7.1 The left shows the original Quake III rasterizer. The right image is the result produced by our ray-tracing implementation. The scene is taken from the map q3dm2. . . . 54
7.2 The left shows the ray-traced result without a denoiser and all 146 lights per pixel. The middle image shows the same but with TAA activated. The right image is the accumulation of 1500 frames with the same configuration as the left image. The scene is taken from the map q3dm4. . . . 55
7.3 The top left image shows Quake III without light maps, which is equal to no illumination. Bottom left shows Quake III rendered with the ray tracer and one light per pixel. Top right is the ray traced result when using all 106 lights per pixel and no denoiser. The bottom right is the ray traced result with one light per pixel and a denoiser. The scene is take from the map q3dm7. . 56
7.4 The top left image shows Quake III with the denoiser deactivated, one light per pixel and light culling off. Bottom left shows Quake III with the denoiser deactivated, one light per pixel and light culling on. Both images on the right show the images from the left processed by the denoiser. The scene is take from the map q3dm9. . . . 57
7.5 Glass tube with a semi-opaque model in between (from map q3dm0). Left image shows the scene with the secondary ray stage functionality for semi-opaque model deactivated. . . . 58
7.6 The scene we used for performance analysis with a mirror in the middle and two glass tubes on the sides. Scene is from the map q3dm0. . . . 58

7.7 Ray-traced Quake III performance across multiple common display resolutions when using a denoiser compared to using no denoiser. 59
7.8 The time the denoiser needs every frame at multiple resolutions. 60
7.9 The difference in performance when all 135 lights are used, in comparison to when we cull the lights that do not contribute to a pixel's shading. 60
7.10 Top row from left to right 1 spp, 2 spp, 4 spp, 8 spp. Bottom row from left to right 16 spp, all 151 lights, 1 spp with denoiser, 1500 images accumulated at a resolution of 1024x768 pixels (ground truth). Light culling was always on. (from map q3dm15) 61
7.11 The left graph shows ms/frame for different sample counts. The right graph compares the visual results for different sample counts using SSIM. Each image is compared to the ground truth (accumulation of around 1500 images). Values are captured from the scene shown in Figure 7.10. 62
7.12 The first image shows our example scene. The second image shows the noise with the large light source being partially occluded by the protruding surfaces around it. The third image shows the noise with the light moved forward to remove all possible occlusions. The fourth image shows the noise with the L dot I term additionally set to 1. In all three images the penumbra can be seen on the right side, and the fully lit area on the left side. 63
7.13 Frametime while moving through the level q3dm0 at a resolution of 720p with denoiser disabled and the use of all lights per pixel. 63
7.14 Left image shows the scene with the denoiser on, from a close distance. The middle image shows the result when the denoiser is on and the camera is further away from the object. The right image shows the accumulated image (no denoiser) from the same distance as the middle image. The shadows in the middle image are much lighter compared to the left and right image. The scene is taken from the map q3dm7. 64

List of Tables

2.1 Distributed ray tracing effects. Note that all these effect introduce noise to the image. In order to reduce/remove this noise more than one SPP is required. 9

4.1 Pros and cons of different ray/path-tracing techniques. 28
4.2 Example for the light visibility texture. Each row represents a cluster. The first column shows how many lights are visible, and the numbers after that are the indices of the lights. In this example cluster 0 is visible from 3 lights with indices 23, 34, 74. Cluster 1 is visible from 1 light with index 103. . 30

5.1 Table that shows the different sizes between 64bit and 32bit environments of long int in bytes. [int] 35

6.1 Commands that can be used to adjust different settings/values of our implementation. 41
6.2 All the ray-tracing specific attachments we use. 43
6.3 The ray payload we use to transfer data between the ray generation, any hit, closest hit and miss shader. 44
6.4 The G-Buffer we use. 45
6.5 The data provided by the any hit and closest hit shader which is used to describe a intersection with a triangle. 48
6.6 This data is set for each individual instance in the top level AS. 49
6.7 The vertex buffer contains all the object data needed after a ray intersects a triangle. 49
6.8 Data of one light source. 50
6.9 A-SVGF stages listed in order of execution. 51
6.10 Buffers that are needed by the denoiser. 51

7.1 Hardware of the computer used for the evaluation 53

List of Algorithms

2.1 Whitted Ray Tracing Pseudocode 10

6.1 Pseudocode for blending transparent textures along a ray 45
6.2 Pseudocode for calculating shading 48
6.3 Pseudocode for reading textures 50

Glossary

AABB A box that is aligned with the axes of the coordinate system, defining a volume in this n-dimensional space.. 30

AS A data structure that accelerates ray intersection tests. 18, 39

BRDF A function that defines how light is reflected at an surface. 11

BVH A tree structure containing multiple objects from a scene, organized by their bounding volumes. 7

CPU A electronic circuitry which executes instructions, called a program. It is the main part of a computer. 1

engine Provides all the functionality that a game requires. 33, 36

FPS The number of frames a 3D applications can display in a second. 3

G-Buffer A collection of buffers which are used to store data of each pixel for the current rendered frame. 28, 29, 44, 45, 47, 71

GI Path Tracing method to calculate correct lighting for a 3D Scene. 10, 25

GPU A electronic circuitry which is designed to execute multiple instructions in parallel. 1

MC A method which uses repeated random sampling to obtain numerical results. 8

PVS A data structure that saves which objects are visible from a certain position. 4

SIMD Multiple simultaneous (parallel) computations from a single process (instruction). 1

SPP The amount of ray traced per pixel. More rays result in a better approximation of the ground truth. 3, 9

SSIM Describes the difference between two images. 61

swapchain The images used for rendering. There are multiple images in the swapchain. so one image can be displayed while another can be used to draw to. 36, 43

TAA A technique which uses the information from previous frames to reduce aliasing in the current frame.. 29

VM The emulation of a computer system. 33

VR Makes use of a head mounted display to fully immerse a player in a virtual world. 1

Acronyms

AABB Axis-Aligned Bounding Box. 30, 38, *Glossary:* AABB
AS Acceleration Structure. 18–20, 39, 42–44, 48, 49, 67, 71, *Glossary:* AS

BRDF Bidirectional reflectance distribution function. 11, *Glossary:* BRDF
BSP Binary Space Partitioning. *Glossary:* BSP
BVH Bounding Volume Hierarchy. 7, 23, 25, *Glossary:* BVH

CPU Central Processing Unit. 1, 36, *Glossary:* CPU

FPS Frames Per Second. 3, 5, 27, 59, 61, 62, *Glossary:* FPS

GI global illumination. 10, 25, 28, *Glossary:* GI
GPU Graphics Processing Unit. 1, 3, 30, 36, 53, *Glossary:* GPU

MC Monte Carlo. 8, 9, 11–17, 67, *Glossary:* MC

PVS Potentially Visible Set. 4, 29, 30, 38, 65, *Glossary:* PVS

SIMD Single Instruction, Multiple Data. 1, *Glossary:* SIMD
SPP Samples Per Pixel. 3, 4, 9, 10, 21, 22, 27, 61, 71, *Glossary:* SPP
SSIM Structural Similarity Index. 61, 62, 69, *Glossary:* SSIM

TAA temporal anti-aliasing. 29, 31, 51, 54, 55, 61, 62, 65, 68, *Glossary:* TAA

VM Virtual Machine. 33, 35, *Glossary:* VM
VR Virtual Reality. 1, *Glossary:* VR

Bibliography

[CPC84] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '84, page 137–145, New York, NY, USA, 1984. Association for Computing Machinery.

[CSK^{+}17] Alessandro Dal Corso, Marco Salvi, Craig Kolb, Jeppe Revall Frisvad, Aaron Lefohn, and David Luebke. Interactive stable ray tracing. In *Proceedings of High Performance Graphics*, HPG '17, New York, NY, USA, 2017. Association for Computing Machinery.

[git] GitHub. `https://github.com/`. Accessed: 2019-05-12.

[Hec90] Paul S. Heckbert. Adaptive radiosity textures for bidirectional ray tracing. *SIGGRAPH Comput. Graph.*, 24(4):145–154, September 1990.

[HHM18] Eric Heitz, Stephen Hill, and Morgan McGuire. Combining analytic direct illumination and stochastic shadows. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '18, New York, NY, USA, 2018. Association for Computing Machinery.

[int] Size of 'long integer' data type (C++) on various architectures and OS. `https://software.intel.com/en-us/articles/size-of-long-integer-type-on-different-architecture-and-os`. Accessed: 2019-05-12.

[Kaj86] James T. Kajiya. The rendering equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, page 143–150, New York, NY, USA, 1986. Association for Computing Machinery.

[MMBJ17] Michael Mara, Morgan McGuire, Benedikt Bitterli, and Wojciech Jarosz. An efficient denoising algorithm for global illumination. In *Proceedings of High Performance Graphics*, HPG '17, New York, NY, USA, 2017. Association for Computing Machinery.

[MPC19] Pierre Moreau, Matt Pharr, and Petrik Clarberg. Dynamic many-light sampling for real-time ray tracing. In *High Performance Graphics*, 2019.

[MSG+19] Adam Marrs, Josef Spjut, Holger Gruen, Rahul Sathe, and Morgan McGuire. *Improving Temporal Antialiasing with Adaptive Ray Tracing: High-Quality and Real-Time Rendering with DXR and Other APIs*, pages 353–370. 01 2019.

[nv] Nvidia. `https://www.nvidia.com/`. Accessed: 2019-05-12.

[nvg] Nvidia Geforce RTX. `https://www.nvidia.com/de-at/geforce/20-series/rtx/`. Accessed: 2019-05-12.

[quaa] Quake II RTX. `https://www.nvidia.com/de-at/geforce/campaigns/quake-II-rtx/`. Accessed: 2019-05-12.

[quab] Quake II VKPT. `http://brechpunkt.de/q2vkpt/`. Accessed: 2019-05-12.

[quac] Quake III Analysis. `https://fabiensanglard.net/quake3/index.php`. Accessed: 2019-05-12.

[quad] Quake III Ray Tracing from 2004. `http://www.q3rt.de/`. Accessed: 2019-05-12.

[quae] The Unreasonable Effectiveness of Quasirandom Sequences. `http://extremelearning.com.au/unreasonable-effectiveness-of-quasirandom-sequences/`. Accessed: 2019-05-12.

[RGA19] M. A. Raihan, N. Goli, and T. M. Aamodt. Modeling deep learning accelerator enabled gpus. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 79–92, March 2019.

[SGFV19] Vadim Sanzharov, Alexey Gorbonosov, Vladimir Frolov, and Alexey Voloboy. Examination of the nvidia rtx. pages 7–12, 11 2019.

[SPD18] Christoph Schied, Christoph Peters, and Carsten Dachsbacher. Gradient estimation for real-time adaptive temporal filtering. *Proc. ACM Comput. Graph. Interact. Tech.*, 1(2), August 2018.

[SS19] Justin Lewis Salmon and Simon McIntosh Smith. Exploiting hardware-accelerated ray tracing for monte carlo particle transport with openmc. In *Benchmarking and Simulation of High Performance Computer Systems*, 2019.

[SSK+17] Christoph Schied, Marco Salvi, Anton Kaplanyan, Chris Wyman, Anjul Patney, Chakravarty Chaitanya, John Burgess, Shiqiu Liu, Carsten Dachsbacher, and Aaron Lefohn. Spatiotemporal variance-guided filtering: real-time reconstruction for path-traced global illumination. pages 1–12, 07 2017.

[vul] Vulkan SDK. `https://www.khronos.org/vulkan/`. Accessed: 2019-05-12.

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

[wrt] Whitted Ray Tracing Explanation. `https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-overview/light-transport-ray-tracing-whitted`. Accessed: 2019-05-12.

[WUM+19] Ingo Wald, Will Usher, Nate Morrical, Laura M. Lediaev, and Valerio Pascucci. Rtx beyond ray tracing: Exploring the use of hardware ray tracing cores for tet-mesh point location. In *High Performance Graphics*, 2019.