Rúben André Campos de Carvalho

Combining the Ray Marching and Rasterization rendering models to provide real-time high-fidelity graphics

Computer Science Department
Faculty of Science, University of Porto
June 2020

Rúben André Campos de Carvalho

Combining the Ray Marching and Rasterization rendering models to provide real-time high-fidelity graphics

Master's Thesis

Supervisor: Verónica Orvalho

Computer Science Department
Faculty of Science, University of Porto
June 2020

This work is dedicated to my father who has been supporting me with all his strength throughout my whole life and who taught me the most important values in life. I also dedicate this work to my grandparents, who have been fighting to see me graduate my masters, and have always been rooting and praying for my best success in life. I hope you are reading this from up above wherever you are Kika.

Acknowledgments

I want to start by thanking professor Verónica Orvalho for allowing me to pursue the thesis topic I most wanted to follow, even though the domain of research is far off from the expected one at my masters course.

Additionally, I want to thank all the professors in the Department of Computer Science of University of Porto for bearing with my constant questions and interest about the topics lectured. Your passion about your research domains greatly motivated me to pursue this thesis work.

Finally, I would like to express my deepest gratitude for the support my father and grandparents gave me throughout my whole graduation, as well as my friends and work colleagues in finishing my thesis work.

Abstract

Providing photo-realistic computer-based images requires complex algorithms with high computational cost. Techniques such as Ray tracing are used in the implementation of these algorithms, which are still expensive in terms of the computation required despite the latest hardware-optimized GPUs built for this task. Therefore, there is a need to develop faster methods of rendering 3D scenes in order to provide graphical application users with a real-time photo-realistic interactive experience.

The main contribution of this thesis is a rendering model designed to be ran on the GPU based on ray marching that speeds up the rendering process of ray tracing algorithms. By utilizing sphere tracing in order to calculate intersections with scene objects, our method is able to provide real-time realistic graphics while not requiring as much computational power as the traditional ones. As part of this main contribution, we demonstrate how this model can be integrated with current hardware and graphics rendering pipelines. Using our system, software application users are now able to experience a photo-realistic interactive experience without having to rely on expensive computer hardware.

In addition to the developed ray marching system, we present an extensive analysis on current state-of-the-art rendering methods throughout this thesis. This serves as a building block to the comparison of the developed system against the most relevant photo-realistic rendering methods used in the present.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this introductory chapter, we start this thesis by introducing our motivation to build this new rendering model. Afterwards, we will take a shallow look at the architecture of the proposed model. We will then describe our objectives and contributions for this work. Finally, we will outline the different chapters along with their content included in this thesis, in order to guide the reader throughout this work.

1.1 Motivation

Providing real-time image rendering is one of the main goals when developing an interactive software application. In software development domains such as video-game and modelling it is also extremely important to provide photo-realistic graphics in order to create immersive environments. Accordingly, there has been extensive progress in the latest years on the development of hardware-optimized GPUs for global illumination rendering techniques such as ray tracing [25].

In contrast to the rasterization techniques used in the past, these new global illumination models are able to emulate how light behaves in the real world, generating images that are indistinguishable from real life pictures.

Nevertheless, the computation required to render highly-complex and detailed scenes in real-time is still extremely heavy. As a result, only the users that have expensive computer hardware are the ones who are able to enjoy a photo-realistic real-time experience.

We believe that every user should be able to enjoy photo-realistic real-time rendering, independent of having high-end computer hardware.

With this goal in mind, we developed a rendering model based on the ray marching rendering

model, sometimes also called sphere tracing, that provides graphic fidelity similar to the one produced by state-of-the-art global illumination rendering algorithms, while using significant less computational resources and providing real-time renders.

1.2 System Overview

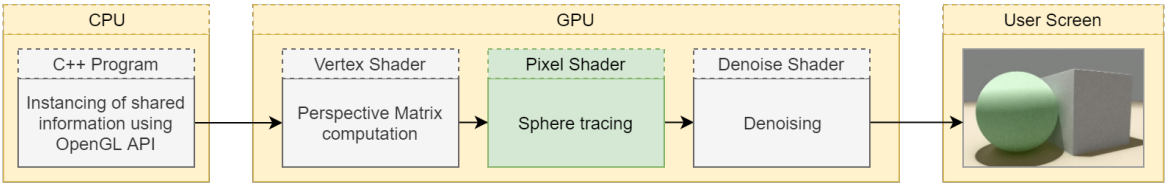

Figure 1.1: An overview of our rendering pipeline model.

The core of our system is a path tracing rendering engine. The implementation of this rendering engine is built on a pixel shader (Figure 1.1 in green), and uses sphere tracing to calculate the distance of each ray to the scene.

In contrast to a ray tracer, our model does not directly calculate the intersection between each cast ray and the scene objects. Instead, each object in the scene is represented by a signed distance function, through which we can query the object distance to each cast ray.

By utilizing this approach for calculating ray bounces and refractions, we are able to mitigate the high costs of calculating complex intersection functions each time we compute the color of a pixel. Additionally, we can easily represent objects that are easily described through mathematical functions, but are harder to represent through a normal vertex specification approach. An example of these objects are fractals. Their three-dimensional representation will be a topic to discuss throughout this work.

For each frame of the realistic visualization we want to produce, our pipeline operates in the order described in Figure 1.1. We will now detail what each stage of the pipeline is designed to do:

1. Starting at the CPU layer, we instance all the necessary variables that are going to be shared with the GPU shader programs. These include the camera position, orientation and lens properties, the resolution of the user screen and time ticks of the program.

2. Afterwards, at the GPU layer and in the vertex shader, we compute the perspective matrix based on the camera information. This calculation is essential to provide real-time interaction of the user with the scene.

3. The previous information is then passed on to the pixel shader, our rendering engine core. In this stage, our sphere tracing based path tracer calculates the color of

each pixel based on the scene information. Techniques such as multi-sampling [36] and next-event estimation [13] are utilized in order to provide high-quality realistic graphics with smooth frame rates. The mentioned techniques are performance and visual optimization methods that will be further explored and explained later in this work.

4. In the last last stage of our rendering pipeline, we feed a denoiser shader with the raw output of our path tracer. Since we want to provide real-time interaction, we must select a small number of samples for our path tracing engine. Consequently, we get a noisy output from our renderer. In order to solve this problem, we designed a denoiser that uses a temporal accumulation technique to remove noise from the images in real-time that is based on previous research on temporal anti-aliasing [20] (TXAA).

1.3 Objectives

Throughout this thesis, we will refer to the main objectives of our work in order to guide the reader through each chapter. We will now define the main objectives of this thesis:

Obj 1. To analyze and compare current state-of-the-art rendering techniques used for real-time graphics rendering.

Obj 2. To develop a rendering method that is capable of providing photo-realistic real-time 3D graphics in significant less computation resources than state-of-the-art methods.

Obj 3. To compare the developed system against state-of-the-art global illumination based rendering engines in terms of performance and graphic fidelity.

1.4 Contributions

We will now outline the contributions of our work. These contributions also help guide the reader through each chapter of this thesis.

Ctr 1. An extensive comparison between currently used rendering models for real-time high-fidelity graphics in the software domain. Additionally, we introduce the fundamentals of the new real-time ray tracing possibilities that emerged from recent GPU hardware development.

Ctr 2. A 3D real-time photo-realistic rendering model that allows every software application user to enjoy an high-fidelity and interactive experience without having to rely on expensive computer hardware.

1.5 Application

We will now explain how different software applications can benefit from utilizing our rendering model as their main core rendering system. In each of the following subsections, we describe how our system can be applied in the specified domain.

1.5.1 Game development

The system solution we present in this thesis allows game developers to render photo-realistic interactive scenes in real-time. Additionally, it allows the representation of geometry that is new to the game development domain, greatly expanding the types of environments that developers can create.

1.5.2 3D scanning

The ray marching technique has been used extensively in the past in order to model distance scans of real-world objects as three-dimensional objects [10]. Our system contributes to this research by providing immersive graphics on top of the scanned model visualization.

1.5.3 Art creation

As a final application of our system, we can use it in order to model artistic pieces. The door to non-solid geometry opens up a lot of possibilities for art creation. A good example of these possibilities can be found in Inigo Quillez's work [31], where he models alien environments through the use of distance field based geometry.

In order to show the possibilities of this new rendering technique, we present an artistic piece created by our rendering model in Figure 1.2. It was achieved by using a coloring technique called *orbit trapping* on top of a Mandelbulb fractal. This coloring technique will be further explained in this work.

1.6 Outline

This thesis is divided into six different chapters. A brief introduction of the content that is going to be presented in each chapter is written in the beginning of every chapter. This introduction helps guide the reader, giving a general overlook to what concepts are going to be presented in each chapter.

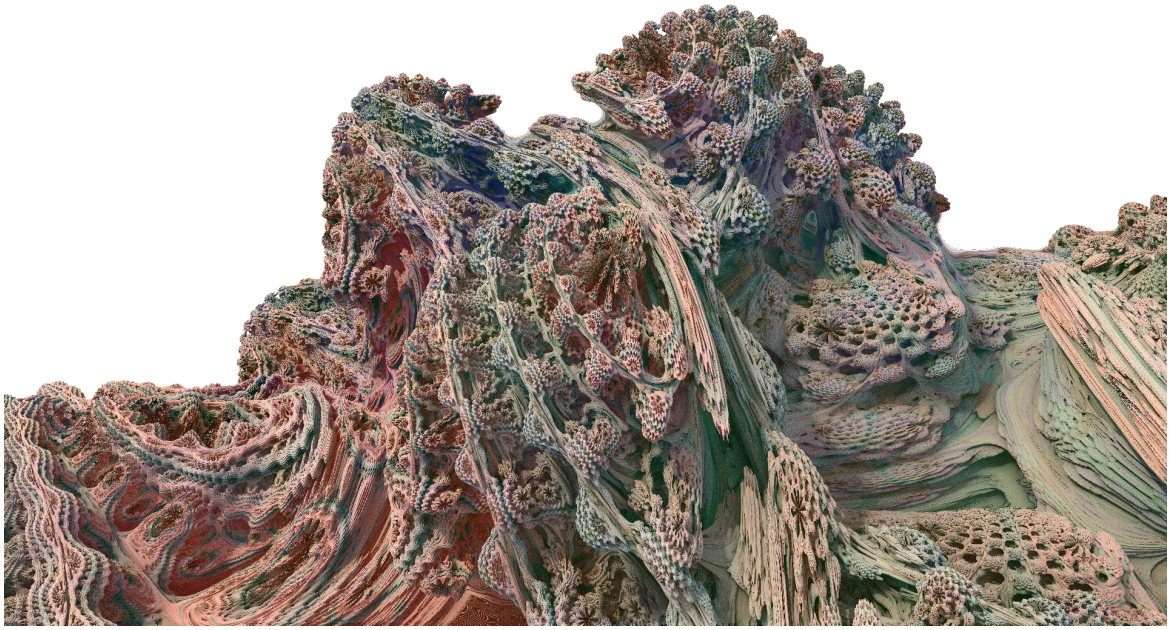

Figure 1.2: Colorful mandelbulb render. Produced by our rendering model in 1080p.

In this section, we outline what content is explored in each chapter of this thesis.

1. **Introduction** - The motivation, objectives, contributions and application domains of this work are introduced. Additionally, an overview of the system architecture is presented.

2. **Background** - Important graphics concepts are introduced in order to help the reader understand fundamental concepts that are used in further chapters of this thesis.

   This chapter is guided by multiple essential questions about computer-based image rendering. By following these questions and obtaining their answers, the reader is able to progressively grasp the main concepts of 3D graphical rendering. Moreover, the reader will be able to understand why certain techniques are more applied in current software applications than others.

3. **State of the art** - Real-time rendering methods that are used in the present are explained, along with the rendering techniques that they use in order to provide interactive photo-realistic experiences. Numerous challenges that these methods face are also outlined in this chapter.

4. **Proposed System** - We take a deep look into the architecture of the developed system. Furthermore, we learn the details of how the ray marching rendering method works and how we implemented it in our model. Additionally, we explain how different

optimization techniques are applied in order to provide faster computational speed while maintaining high graphics fidelity.

5. **Results and evaluation** - Multiple renders produced by our model are presented. Additionally, we compare the proposed system against state of the art methods such as Monte Carlo path tracing.

   This comparison will be extended to multiple quality levels such as time to render, graphical fidelity, implementation complexity and applicability in current software applications.

6. **Conclusions and future work** - A conclusion on the results of the evaluation of the proposed system is presented. In addition, we propose how the developed model can be further improved through software and hardware optimization steps. Finally, we discuss on what the future holds for real-time photo-realistic rendering.

Chapter 2

Background

In this chapter we will look into the most utilized rendering techniques in the present that allow content creators to develop photo-realistic real-time graphics. After finishing reading this chapter, the reader will be able to understand the fundamental techniques and different algorithms utilized in order to achieve 3D photo-realistic rendering. He will also be able to distinguish different types of rendering methods, along with their pros and cons, and how they can be applied depending on the task at end.

First, we will understand what real-time graphics are and how they are valuable in software applications. Then we dive into how the modern graphics pipeline works and what shaders are. We will then study the implementation, advantages and disadvantages of the rasterization rendering model, followed by the introduction of global illumination algorithms such as ray tracing and how they differ from the rasterization method. We conclude this chapter with the introduction of the ray marching model. This final section is crucial in order to understand the proposed system in this thesis.

2.1 The importance of real-time graphics

Before diving into the technical details of how real-time graphics are computed, it is of extreme importance for the reader to understand why providing fluid visuals in computer-generated imagery is valuable. In this section we answer the question *'What are real-time graphics and what is their purpose in current application development?'*.

Graphics rendering has been around for decades. In fact, mathematical models that allow us to represent real-world light behaviour date back to 1968 with IBM researcher *Arthur Appel* instigating the development of the graphics rendering domain through his work [3]. Although these mathematical models such as radiosity [12, 24] and ray tracing [17, 41]

existed for years prior to their adoption in real-time software applications, their algorithm implementations were still highly complex for the hardware that was available at the time [30]. As such, software applications that typically require interaction from the user were not capable of utilizing the proposed rendering techniques for 3D graphical images: the algorithms proposed were too expensive to be computable in real-time.

Faced with this problem, developers had to come up with a solution that allowed them to augment their software with graphical images while preserving the most important aspect in a software application: user interaction. As a result, developers adopted the rasterization rendering model [9], which is still the most used graphical rendering method for interactive graphical applications nowadays.

Real-time graphics are extremely valuable for domains such as video games, film making and 3D modelling, where the interaction of the user or the developer with the graphical environment is a must.

2.2 Achieving real-time graphics

Now that we are aware of the importance of real time graphics, it is important to understand how these can be achieved using modern hardware.

2.2.1 Defining real-time

Before diving into how real-time graphics can be achieved, we must first define our notion of what real-time really means. For the sake of the following arguments and information presented, we will define real-time graphics as graphical images that are provided to the user screen at a speed of *at least 24 frames per second*. This lower bound is the minimum speed of frame delivery that is needed for a human eye to perceive it has motion [34].

2.2.2 Rasterization

The most utilized rendering method in the present for interactive graphical applications is the rasterization rendering method. In this model, object surfaces are represented by triangles, called polygons, which are the core atomic sub-divisions of the object in 3D space.

Rasterization is commonly used for interactive graphical applications because it is extremely fast to compute the color of each pixel on the screen by following this approach. For the latest years, GPU hardware and graphical APIs have been designed with the rasterization process at its core [22]. As a result, the current graphical hardware we have installed in

our computers is capable of running the rasterization algorithm for millions of polygons in a scene in real-time. Consequently, this rendering model is an ideal candidate to build rendering engines for games and modelling applications, since these software applications require constant user interaction.

2.2.3 Modern graphics pipeline

In order to understand the connection between how the software and hardware was developed to serve the rasterization rendering technique, it is important to grasp the main concepts of the modern graphics pipeline. By following this section, the reader will be able to understand how the software APIs communicate with the GPU hardware in order to provide 3D polygon rendering in real-time for applications.

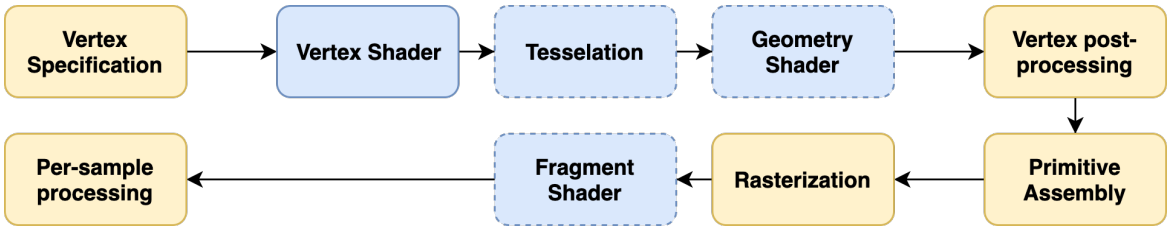

Figure 2.1: OpenGL rendering pipeline. Blue boxes represent programmable stages. Boxes with dotted lines represent optional stages.

The graphics pipeline is the process of operating over graphical data, initially vertices, from the stage it is specified at the CPU level, till the moment it is displayed on the output screen, as output from the GPU.

As seen in Figure 2.1, the modern graphics pipeline is defined by multiple stages, some of which are programmable. Programmable graphics hardware is extremely important in order to allow developers to have the freedom of deciding what transformations they want to apply over graphical data [40].

In the context of this work, we will now explain how a developer can configure these programmable stages in order to obtain the graphical images he wants to produce.

Shader programs

The programmable stages that are part of the graphics pipeline are called shaders. A shader is a program written by a developer that is compiled at the CPU and later ran at a specific stage of the rendering pipeline at the GPU.

All the shaders in a given pipeline are usually written in the same language, typically GLSL

for applications built using the OpenGL API. Conversely, the type of data a shader processes is dependant on the stage it belongs to.

For instance, the vertex shader operates on vertices. This means that it will be ran once for each vertex that is given as input from the vertex specification stage. Moreover, the fragment shader operates on pixels. As such, it is ran once per each pixel of the user screen. This process is repeated every frame. With modern-day screen resolutions boasting over 8 million pixels, it is easy to understand how expensive the fragment shader stage can be for shading computation.

Fortunately, this process of iterating over every pixel or every vertex is a good candidate for parallel processing, and GPU hardware is built with this in mind to vastly improve rendering throughput [6].

Shader programs must also be written with the parallel processing architecture in mind. What this means is that if we are writing a pixel shader, we must write code that is independent from the pixel color of neighbor pixels of the one we are computing.

2.3 Achieving photo-realistic graphics

In previous sections we introduced how developers are able to achieve real-time graphics rendering in order to provide interactive application users motion graphics. We introduced the rasterization rendering model, but have still not taken a look at the raw image output a non-configured rasterizer produces.

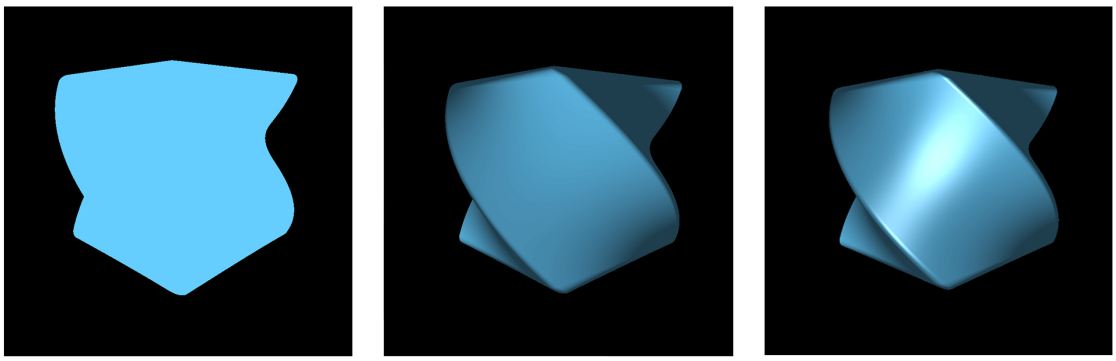

Figure 2.2: From left to right: Ambient light, diffuse light and specular highlights added on top of each other, produced by simple phong shading applied to a rasterized output of a twisted cube geometry.

In order to explain to the reader the difference between a raw rasterizer output and what can be achieved through shading methods using a fragment shader, we built a simple shader application using OpenGL and GLSL. Its output is shown in Figure 2.2.

Note how the object details are not recognizable by looking at the left image in Figure 2.2. This image was produced by simply coloring the rasterized pixels using a constant color, which we call the ambient light in this example.

Even though there is the presence of a three-dimensional object on the scene, by looking at the raw output from the rasterizer we can't visualize more than it's silhouette. In order to define the three-dimensional objects on our scene and make it look realistic, we need to add shading to our rasterizer output.

2.3.1 Shading

Shading is the process of adding depth to rasterized fragments in order to make them perceivable as three-dimensional objects [7]. Algorithms like Phong Shading [29] (Figure 2.2) are used for this matter. These shading algorithms calculate the amount of shading multiplier that must be applied to the color of each face of an object.

The shading calculation is typically based on the angle between the direction to the light source in the scene and the normal of each face of a given object. In short, the more angled a surface is compared to a light source, the darker it will be, and vice-versa.

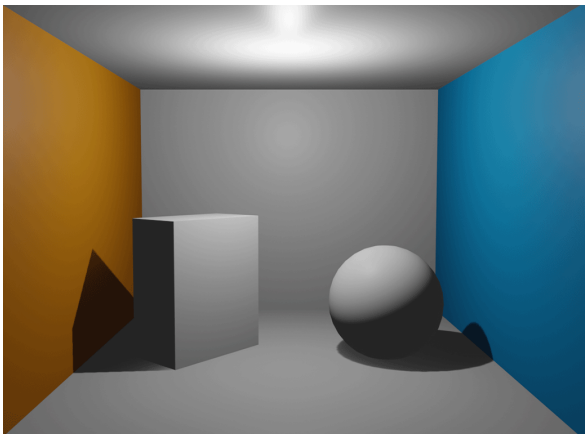

Figure 2.3: Phong shading technique applied to a rasterizer output. Render was produced using Blender 2.81.

Using the shading techniques described above, we produced a simple render that can be seen in Figure 2.3. While it is perceivable that the scene is three-dimensional, it is still far from what it would look like in real-life.

Can we identify what is the missing piece in order to trick our eyes that this may be a real-life picture?

2.3.2 Global illumination

It turns out that what is left in order to turn our rendered images into photo-realistic pieces is to account for global illumination.

Global illumination, sometimes also called indirect illumination, is an important real-world light phenomenon. It stands for the behaviour that light produces when it bounces multiple times in other scene objects before it reaches our eyes. This causes all object in a scene to contribute to the lighting of every other object in the same scene [35]. Although the difference may appear very subtle, developers quickly discovered the drastic impact it has on producing a realistic render image.

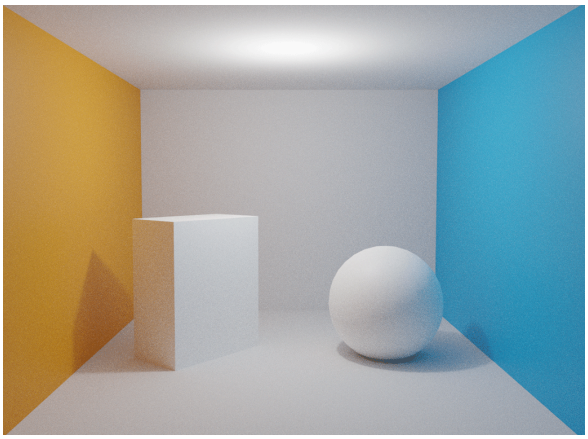

Figure 2.4: Global illumination based render produced using Blender 2.81, using cycles rendering engine.

After rendering the same scene as in Figure 2.3 but using a global illumination based shader, we can now see how dramatic the difference is in Figure 2.4. Since we are now accounting for light reflections between all objects on the scene, we obtain effects that are expensive to produce, such as light accumulation, color bleeding and ambient occlusion.

Even though global illumination based renders may look great, there is an expensive price to pay: the computation required to render a single frame may require an entire day for complex scenes [35]. As a consequence, it is easy to understand how this technique is not feasible for interactive software applications.

2.3.3 Global illumination based algorithms

We will now learn about the algorithms that are most utilized in the present in order to provide global illumination based realistic render images. We will first start by understanding the ray tracing technique, following up with its integration in the path tracing approach.

Afterwards, we will look into sphere tracing, a technique also called ray marching, and how it differs from the original ray tracing method. We will conclude on the application of these algorithms in current application software.

Ray Tracing

Ray tracing is a global illumination rendering method that originated from the necessity of providing high-fidelity realistic graphics.

The main idea of the ray tracing rendering model is to *shoot* one or more rays per pixel and calculate intersections of each ray with the scene objects. From each intersection, or hit, we can compute the amount of light that is absorbed by the hit object and reflected to other objects [30].

The amount of light an object absorbs and reflects is defined by its BSDF, short for bidirectional scattering distribution function [5]. This function models the different properties object surfaces have: specular surfaces such as mirrors should reflect most of the light, while diffuse surfaces should scatter the incoming light in a uniform manner (see Figure 2.6).

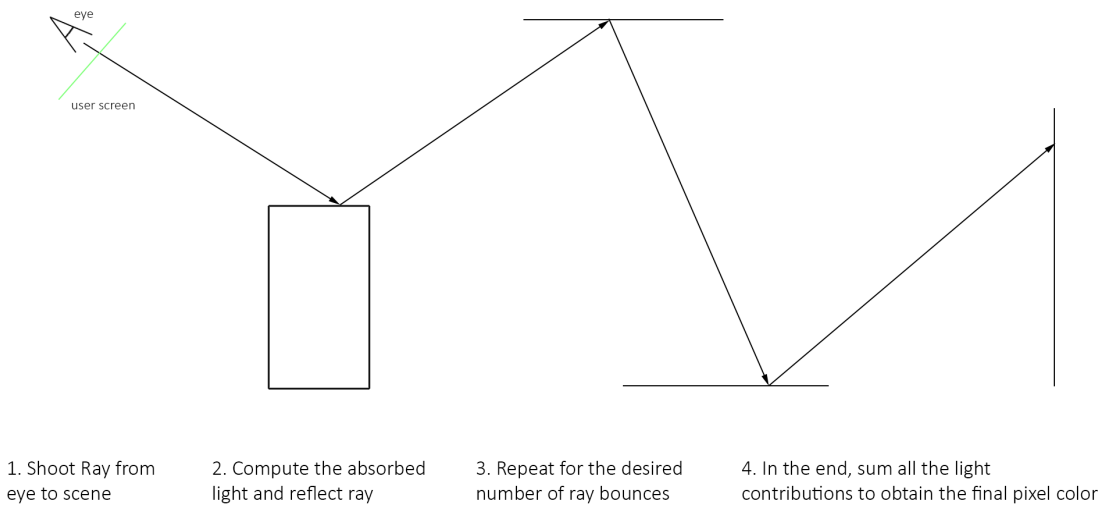

Figure 2.5: Illustration of a simplified backwards ray tracing computation of the color of a **single** pixel.

This process of tracing a ray intersection with the scene, reflecting it and applying the same process keeps on repeating itself, thus we typically end up with recursive algorithms for the ray tracing rendering approach [30].

It is important to note how the color for each pixel is calculated. A view-plane model is

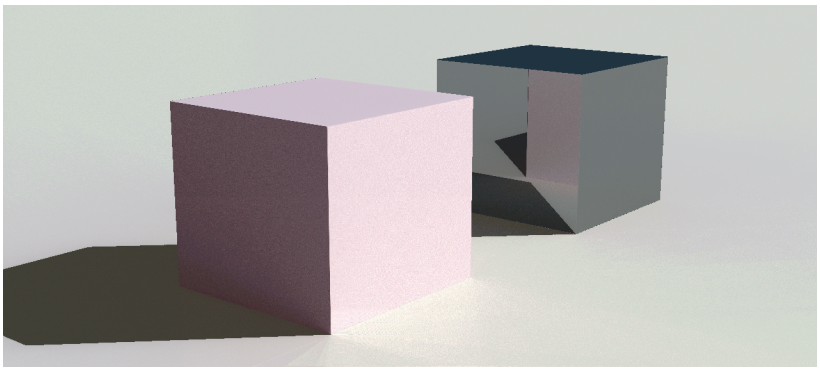

Figure 2.6: Two cubes with different BSDFs: diffuse (left) and specular (right). Scene rendered using our rendering model.

represented in the scene right after the eye, resembling a mapping of the 3D image we see into a 2D plane. Each ray casted through the view-plane is in charge of coloring the pixel it intersects.

We are currently at a stage where we believe it is beneficial to introduce the term *ray bounces*. This term stands for the number of bounces we want to account for the light behaving in the scene. The higher the number of bounces, the more realistic the rendered image will be. On the other hand, the lower the number of bounces, the quicker the computation will be done, since for each cumulative bounce we have to recursively call the tracing function again. In the example of Figure 2.5, we have a diagram where the color of each pixel is influenced by 3 light bounces.

Ray tracing might seem like the most optimal way of developing photo-realistic renders but there is a big caveat to this technique. In order to understand this fundamental drawback, we are first going to learn about the rendering equation [17].

The rendering equation

Let's start with the basic idea:

For each point in the real world, the amount of radiance that the point outputs is equal to the sum of the radiance it emits, plus the radiance it reflects.

As a direct correlation of this, we can also conclude that every ray of light in a scene contributes to the shading of every surface point.

The mathematical equation that describes the mentioned physical behaviour of light is called the rendering equation, and is defined below [17].

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_\Omega f_r(x, \omega_i \rightarrow \omega_o) L_i(x, \omega_i)(\omega_i \cdot n) d\omega_i \tag{2.1}$$

- $L_o(x, \omega_o)$ is the outgoing radiance at the point x in the ω_o direction.
- $L_e(x, \omega_o)$ is the emitted radiance at the point x in the ω_o direction. This value is only different from 0 for points that are light sources.

In other words, each time we are tracing a ray and it hits a diffuse surface, we should divide its radiance in multiple other infinite reflection vectors over a semi-hemisphere normal to the surface the point we hit belongs to. A visualization of the proposed computation is presented in Figure 2.7.

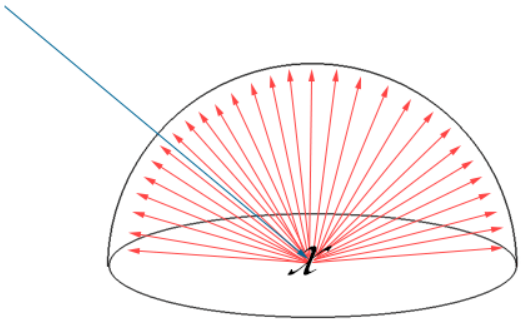

Figure 2.7: Visualization of the rendering equation integral.

The main problem with the Ray tracing approach is now clear: if we want to model realistic surfaces, we surely want to have diffuse surfaces in our scene. As a consequence, we must infinitely iterate over the tracing computations in order to create a truly photo-realistic scene. This, put in mathematical terms, means that the rendering equation is infinitely recursive. Consequently, this equation cannot be solved analytically because the number of integration domains is also infinite [17].

Path Tracing

In order to solve the problem of representing diffuse surfaces in a realistic manner by using ray tracing, the path tracing algorithm was created.

The path tracing algorithm presents a solution in order to approximate the value of the rendering equation, since computing its real value would be infinitely expensive in terms of computation steps [21]. It is similar to the ray tracing technique, with the difference that it models the diffuse element of surfaces in a realistic way.

The solution is simple: by following a monte carlo approach to solve for ray reflections on diffuse surfaces, we are able to compute an approximation of how exactly light would bounce around in a given scene [36].

A monte carlo path tracer randomly chooses different reflection vectors when light bounces around the scene. While in theory we would need an infinite amount of reflections to capture the radiance coming from all directions, if we randomly pick a certain number of reflection vector samples when we hit a point in the scene, we can progressively get closer to a realistic solution while preserving computation time [36]. After we define a specific number of samples S_n, the path tracer calculates S_n times the color of each pixel in the screen for a given scene and blends the computed pixel color values into a final color.

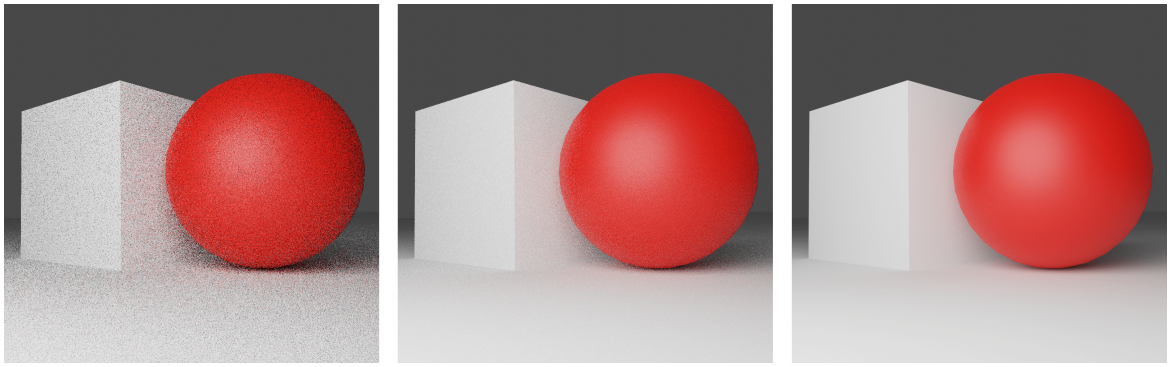

Figure 2.8: Path tracer samples comparison. From left to right: 1, 4 and 16 samples render of the same scene. Renders were all produced in Blender 2.81 using Cycles renderer.

One of the main drawbacks of utilizing a low-sample approach to rendering a photo-realistic scene with the path tracing technique is the amount of noise produced in the final render. From Figure 2.8 we can see that low-sample renders have a lot more noise than the renders with a lot more samples.

The more samples we use, the better the final image will reflect reality. This reality is typically mentioned in research as *ground truth*. Nevertheless, we should always keep in mind that this additional image fidelity comes at the price of rendering times higher in orders of magnitude when compared to low-sample renders.

Ray Marching

We will now introduce the ray marching technique. In this section, we will explain how this technique differs from ray tracing and what can be achieved with its use.

Ray marching, as a rendering technique, is similar to the ray tracing technique: for each pixel on the screen, we shoot one or more rays and trace their reflection behaviour in the

scene the number of times we decide to, until we are satisfied with the result.

However, there is a fundamental difference between these two techniques. In ray tracing, we calculate the intersection of each ray with all of the scene objects at each bounce step. In contrast, in the ray marching approach we march each ray until it hits an object on the scene [39].

There are two main problems that are bound to happen with this marching approach if we were to use a fixed marching step distance. Both these two problems can be visualized in Figure 2.9.

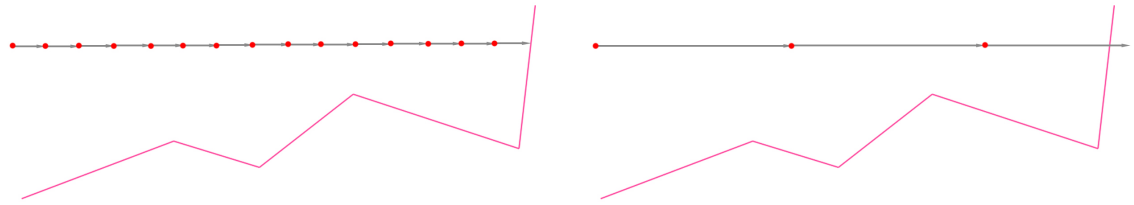

Figure 2.9: A visualization of the under-stepping (left image) and over-stepping (right image) problems when marching a ray using a fixed-distance step. Both images define an object surface (pink) and the stepping of a single ray in an horizontal direction, until the ray hits the surface (left) or over-shoots it (right).

The first is under-stepping. This problem happens when we pick a small step distance for our marching algorithm. This would mean that we would rely on marching the ray countless times until we would hit an object in the scene. Consequently, this approach would prove to be inefficient.

The second is over-stepping. This problem occurs when we select a step size that is too big for the objects in our scene. Since we would over-shoot our ray through the object, in this case we would have inconsistencies in our rendering.

How can we guarantee a step size that minimizes the number of steps required to hit an object and that also does not overshoot the objects in the scene?

In order to answer this question, let's first look into the Ray marching algorithm presented in Table 2.1. For each marching step of a ray, we first calculate the minimum distance from the point to the scene by calculating the signed distance function for the current point we are marching. A signed distance function is a mathematical function that given a certain point p, returns us the distance of p to the object that SDF represents. Additionally, it is important to note that for a given point p, the distance of p to a scene composed by multiple

Algorithm 1: Ray marching algorithm

```
totalDistance ← 0
for step ← 0; step ≤ MAX_MARCHING_STEPS do
    currentPoint ← rayOrigin + totalDistance * rayDirection
    distance ← sceneSDF(currentPoint)
    if distance < ϵ then
        return totalDistance
    end if
    if totaldistance ≥ MAX_DISTANCE then
        return MAX_DISTANCE
    end if
    totalDistance += distance
end for
```

Result: Distance to the hit object

Table 2.1: Ray marching algorithm

objects can be obtained by calculating the minimum of all SDFs evaluated at p.

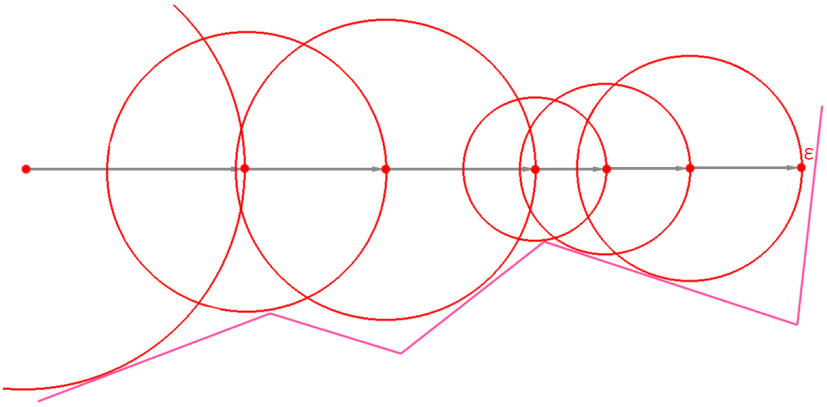

Figure 2.10: A visualization of the ray marching technique applied with a variable sized step based on the minimum distance to the scene.

By utilizing the minimum distance to the scene as our step size, we are sure to never overshoot an object, while keeping the number of steps required to march until we hit an object at minimum [14]. Following this approach, we will never actually hit an object with our ray march. Instead, we will consider a collision when the ray is really close to an object surface. This close distance threshold is represented by the ϵ value in the algorithm specification presented in Table 2.1 and in Figure 2.10.

Since the ray marching approach relies on representing objects through the use of signed distance functions, this brings up a compromise on how we are able to represent objects in our scene [14].

On one hand, we have more difficulty representing common real-word objects that are typically modeled through their vertex points. On the other hand, we are able to represent three-dimensional mathematical structures such as fractals which are shown in Figure 2.11 (right).

Additionally, the duplication of objects is cost free with the SDF representation, since we can use the modulus operator on top of the SDF result to *warp space* and create infinite copies of the same object. As an example to the domain repetition technique, the scene represented in Figure 2.11 (left) has the same computational complexity as a scene with a single cube would have.

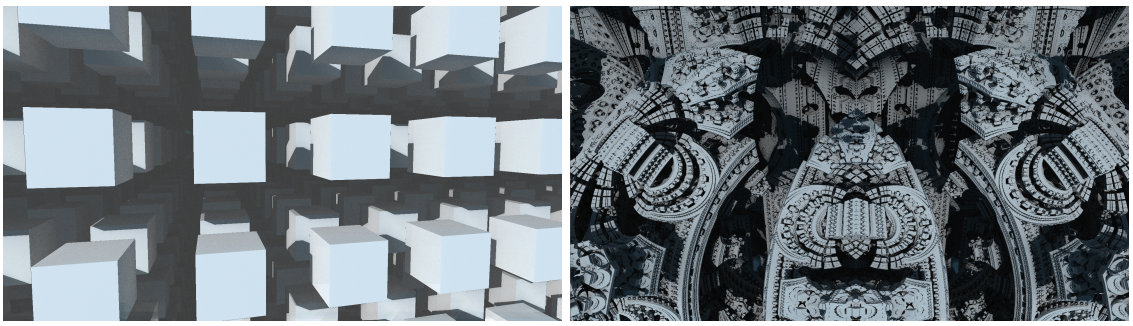

Figure 2.11: Domain repetition based on signed distance function representation of a cube (left) and 3D fractal render (right). Both renders were produced by our system.

2.4 Summary

Real-time graphics are essential for providing an interactive experience to software application users.

Throughout this chapter, we learned that there are two main different rendering techniques:

First, we have those which are not based on global illumination. These are commonly used for real-time graphics.

On the other hand, we have global illumination based rendering techniques. These are able to provide high-fidelity graphics that look just like real-life images.

Our goal is to combine both rendering techniques in order to provide a rendering system that is able to compute high-fidelity graphics in real-time.

Chapter 3

State of the art

In this chapter we will learn about the techniques that are used in order to provide real-time photo-realistic graphics in current software applications. In each of the following sections, we will describe a technique. Afterwards, we will give some examples on how the idea is applied in applications in the present. Additionally, we will describe the advantages and disadvantages of each approach. Finally, we conclude each section by commenting on how its information will help drive our work.

Firstly, we will explore the light mapping technique and how it is able to emulate realistic static environments.

Secondly, we will dive into how developers are able to use recent hardware development such as Nvidia's new Turing architecture in order to provide real-time ray tracing computation through an hybrid graphics rendering pipeline model.

Finally, we will take a look at state of the art research on the ray marching technique based on distance fields, in order to drive our reasoning behind choosing this rendering model for our proposed system.

3.1 Light Mapping

We will now discuss a technique that is widely used in order to create photo-realistic environments with effects such as global illumination and realistic shadows in software applications. This technique is utilized in conjunction with the rasterization rendering model. As a consequence, it is able to provide a real-time interaction with the user.

Explanation

The technique we are referring to is called light mapping. In reality, the high-fidelity effects that are provided by this technique are not dynamic. Instead, pre-computed light values from a scene are stored into an additional texture that is mapped into each object. This technique was invented and first implemented in the game *Quake*, and was initially called surface caching [2].

By following this approach, the lighting stays static even when certain objects of the scene change. As such, light maps are most commonly used when lighting big environments such as buildings, since their lighting is not going to change for a given scene.

The light mapping technique is usually combined with simple dynamic lighting models. In this way, the rendering engine only has to compute simplified light shading on the objects of the scene that are dynamic.

Application

The light mapping technique is currently widely used for shading static video-game scenes. Most of the developed rendering engines that are being used today for video-game production rely heavily on this technique in order to provide photo-realistic environments.

An example of the light-map technique usage is seen in Figure 3.1, where a render from a game that is built using Epic Games's *Unreal Engine 3* is displayed. Other famous rendering engines like *Crytek*'s *CryEngine* and *DICE*'s *Frostbite* engine also rely on using the light mapping technique together with path tracing approaches in order to bake the lighting into object texture and uv maps [27, 11].

Additionally, architecture simulation applications also make use of this technique in order to provide realistic-looking virtual house models for customers to explore.

Advantages

The main advantage of the light mapping technique is that it is able to provide photo-realistic graphics without increasing the run-time computation required to render a scene when compared to a lighting model such as the Blinn-Phong model.

With this rendering technique, we can store expensive computations of the path tracing model for a given scene and quickly sample those baked computations in real-time for the user. Since we don't have hardware for now that is capable of rendering a complex scene in real-time using only path tracing [18], this technique is extremely valuable in order to

Figure 3.1: An in-game render of one of Mirror's Edge gameplay levels. All light effects that are visible on the scene walls are produced by the light mapping technique. Since most of the solid structure is static, the application of this technique is seamless.

provide realistic visuals in real-time for applications.

Disadvantages

The main drawback of the light mapping technique is that it only works for static objects. While this is fine for architecture simulations, on the other hand video-games and movies typically require constant object interactions for a given scene.

One approach of optimizing this disadvantage that is commonly used in rendering engines is to effectively pre-compute the static light computations required for the static elements in the scene. Later, in run-time, the lights produced by dynamic scene elements are calculated on the spot, using simple lighting models such as the Blinn-Phong or Lambert models.

While the mentioned approach is extremely effective, visual differences between how realistic static objects are lighten and how dynamic objects behave against light can be perceived when the light mapping technique is utilized. We believe that these differences can impair the immersion of a user in the virtual environment.

3.2 Hybrid graphics rendering

In this section we will discuss a different rendering pipeline architecture than the one presented previously in the Background section. This new architecture recently came into wide use in current AAA game titles due to the hardware-dedicated ray tracing GPUs

launched recently by Nvidia.

Explanation

While we believe that ray tracing is the future for realistic graphics rendering, applying this expensive algorithm for complex scenes in our applications is still not doable in real-time due to hardware limitations [18]. In order to reap the visual benefits of the ray tracing technique, as well as the performance benefits of the rasterization technique, the hybrid graphics rendering pipeline model was envisioned.

As the name implies, the hybrid graphics pipeline consists in mixing the two main approaches for rendering, rasterization and ray tracing, in order to provide fantastic visuals at a low performance cost when compared to using a path tracing engine for example.

The main idea is to render a three-dimensional scene using normal rasterization approaches first. At the same time, values such as the normal vector to each primitive are stored in a collection of textures called the G-buffer.

Afterwards, shadows, reflections, global illumination and ambient occlusion light effects are ray-traced using a path-tracer with a sample count of mostly around the 1spp mark (one sample per pixel).

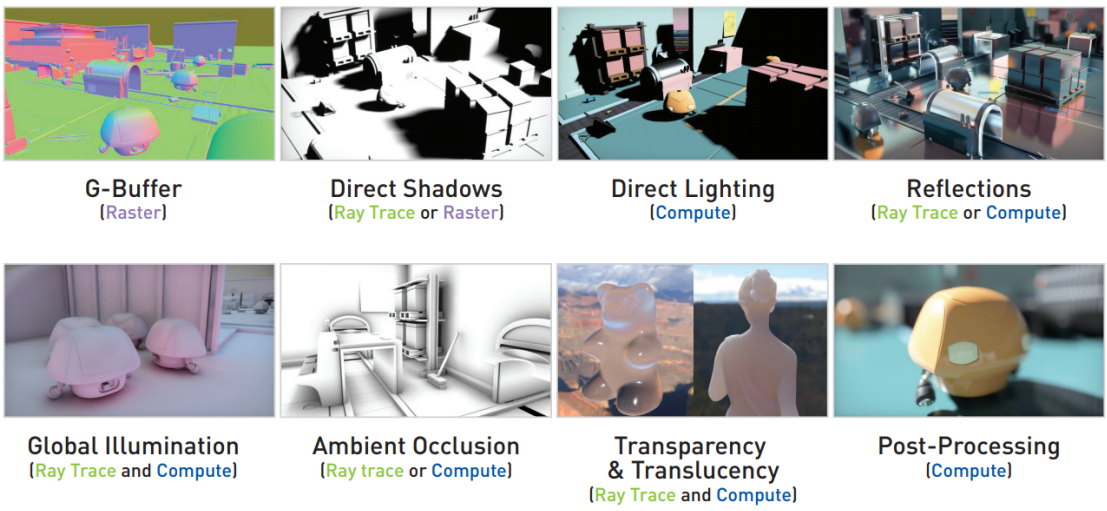

Figure 3.2: A general overview of the selected hybrid rendering steps in EA's PICA PICA DirectX RTX ray tracing project [4]

While 1spp may produce a lot of visual noise, de-noising techniques are applied on top of each step's output in order to mask the output noise. By maintaining such a low sample count and applying a de-noising step after, the path-tracing computation can be done in

real-time while providing a smooth looking final image. Refer to Figures 3.3 and 3.4 to understand what a 1spp output would look like if it was not de-noised before being shown to the user.

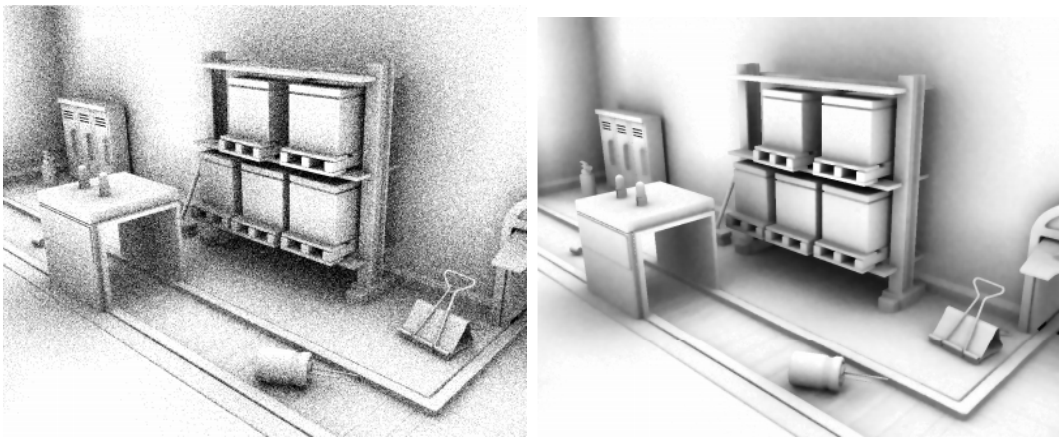

Figure 3.3: De-noising technique applied to ambient occlusion 1spp path tracing output. Left image is the raw output. Right image is the de-noiser output. Images taken from the PICA PICA project [4].

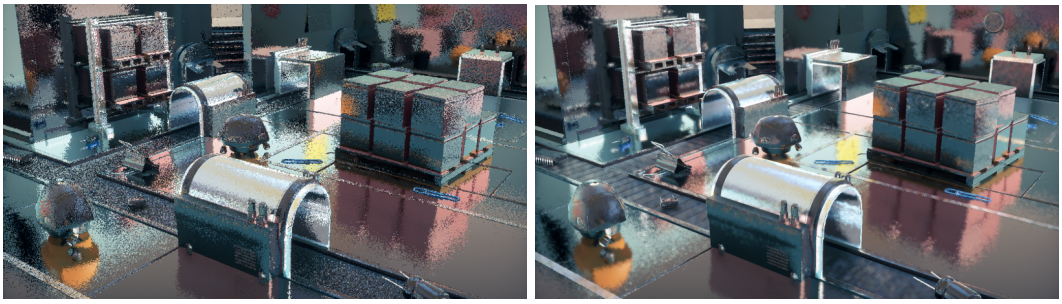

Figure 3.4: De-noising technique (right) applied to reflection path tracing output (left). In the case of calculating ray reflections, the developers took it up a notch and decided to only reflect 1 ray for each 4 pixels, thus decreasing the sample count to 0.25spp. Images taken from the PICA PICA project [4].

Finally, all the outputs from each processing step are mixed to generate a final frame image.

Advantages

The main advantage of this rendering architecture is the fact that is extremely adaptable to different performance and visual requirements of a given application. This method is able to provide beautiful real-time photo-realistic graphics.

Disadvantages

While the hybrid architecture is extremely adaptable to different applications, it poses a difficult challenge for developers of software applications: the implementation must be designed by the developers [18]. As a consequence, these design decisions must be taken into account in the application development life-cycle. In other words, the hybrid graphics pipeline approach is not a general solution to the problem of providing photo-realistic graphics in real-time.

3.3 Ray marching distance fields

In this section, we will look into recent research on the ray marching rendering model based on distance fields. Moreover, we will explore the limitations of this rendering method, as well as the unique value it can provide in graphics rendering.

Application and previous work

The ray marching technique, also commonly mentioned as sphere tracing, is currently used in rendering engines for volumetric rendering [28] and procedural terrain generation [26]. Most of the current applications of this technique are based on volumetric rendering due to the nature of how we can easily map volumetric spaces into mathematical functions.

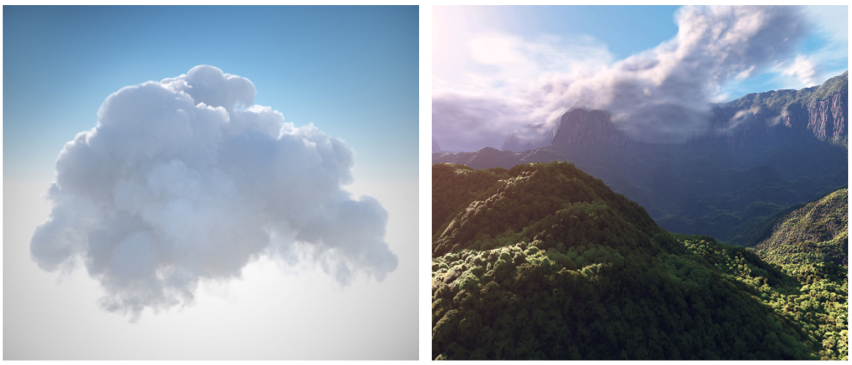

Figure 3.5: Renders produced by the ray marching technique. On the left, we have a volumetric cloud render produced by OctaneRender 3. On the right, we have a beautiful terrain scenery produced by Inigo Quilez [32], available at his graphics blog.

While rendering volumes is the domain where the ray marching algorithm is being mostly applied, using the sphere tracing technique in order to represent solid objects is not uncommon, since the technique was first applied in order to represent three-dimensional fractals

[15].

Later research allowed developers to approximate ambient occlusion light effects using the same technique, in much less computation than global illumination based methods [1]. Additionally, research on representing most of the basic solids as signed distance functions [31] allows us to mix the mentioned shading techniques with solid geometry in order to test if this method is viable for portraying realistic three-dimensional scenes.

Shading of simple solids

In order to prove the solid shading capabilities of the ray marching algorithm, we decided to create a simple shader application and implement the most important lighting effects that are desired in a photo-realistic experience. This experimental application will be useful for the development of the proposed system in this work.

In this section we will guide the user on how we implemented each of the shading properties, or channels, that we believe are important in order to provide realistic graphics.

Albedo, Diffuse and Specular

For albedo, we simply paint each pixel with the color of the hit solid. For the diffuse and specular channels we used a simplified version of the Blinn-Phong algorithm for light shading.

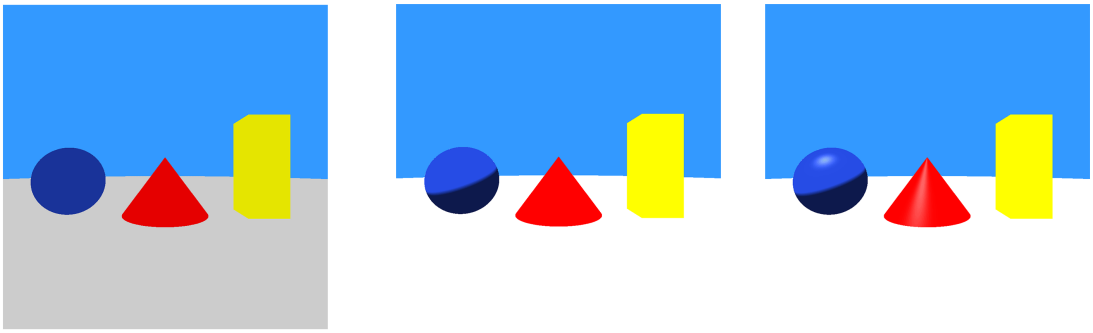

Figure 3.6: From left to right: implementation of albedo, diffuse and specular highlights in a simple ray marching based shader. Each channel is represented additively from left to right.

Soft shadows and penumbra, Ambient occlusion and Reflections

Inigo Quilez mentions in one of his articles that *one of the many advantages of distance fields, is that they naturally provide global information* [33]. This global information that

he mentions allows us to calculate otherwise expensive shading effects in low computational steps for distance fields. By using the number of steps we march for each ray and the minimum distance that the ray marched close to the scene, we are able to almost freely calculate effects such as soft shadows and ambient occlusion.

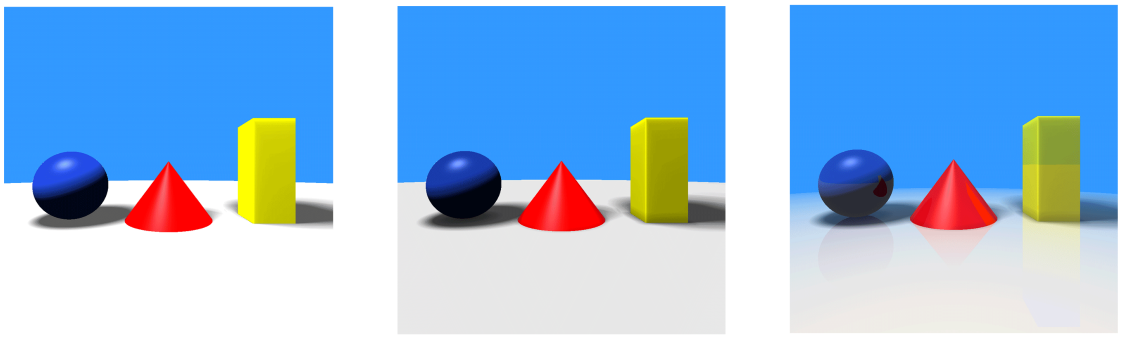

Figure 3.7: From left to right: implementation of soft shadows, ambient occlusion and reflections represented additively in each channel.

Reflections are finally added by bouncing each ray once when it hits an object on the scene. The color of the second object the bounce ray hits is captured, and mixed with the color of the first hit object.

Advantages

In the previous section we proved that the ray marching technique can be used with solid geometry in order to represent common real-world geometry. Additionally, global illumination techniques such as soft shadows, penumbra and ambient occlusion can be calculated cheaply with the sphere tracing approach. While these techniques are extremely helpful in order to simulate realistic light behaviour, the randomness necessary in order to model diffuse surfaces still requires a path tracing monte-carlo approach. As can be seen in Figure 3.7, we are not accounting for light bounces around the scene. We are also not accounting for the physical behaviour light has when it reflects from a surface and scatters around the scene.

The main advantage of ray marching distance fields is the unique geometry that can be represented through the use of this method. Later on our work, we will show examples of this in our three-dimensional fractal geometry renders.

Another advantage of using distance fields as the representation of our scene is that we can model a complex scene in substantially less memory than rasterization based rendering models: since the scene is modeled by a multitude of mathematical functions instead of

millions of vertices, it is easy to understand why less memory is utilized.

Disadvantages

There are two big drawbacks to using the ray marching approach in order to render solid geometry.

First we have the problem of representing a three-dimensional scene. Since all objects must be represented by mathematical functions, this poses a real problem for developers that want to model their environments. Although there are applications and techniques that can read the distance of all points to a given primitive and store that information in a texture that represents the SDF of the modeled object [19], these methods are still very expensive and not computable in real-time.

The second drawback is when bouncing rays around the scene using the ray marching approach. Since each object is represented by a distance field, we don't store the normal values of this distance field *apriori* like the rasterization process does. Consequently, every time we want to calculate the reflection direction of a ray we are marching, we must calculate the derivative of distance field function in three different close points to the surface we hit in order to know the normal value of the distance field at that point.

3.4 Summary

From analyzing the multiple rendering techniques that we described in this chapter, we can conclude that providing high-fidelity graphics in real-time is still a balancing act that requires the introduction of specific optimization steps to a rendering system.

While some of these optimizations steps can lead to limitations in a specific rendering system, our goal is to utilize some of the optimizations that these models introduce such as hybrid rendering and light mapping in order to build a highly performant rendering model.

In the next chapter, we will take a look into the proposed rendering model in this thesis and how it uses the Ray marching technique. Additionally, we will describe how our model aims to solve the disadvantages of the other proposed models in this chapter.

Chapter 4

Proposed System

In this chapter we will look into the architecture and implementation of the developed system in this thesis. Firstly, we will explore how data flows in our proposed rendering pipeline as well as our thought process when designing the data flow architecture. Secondly, we will document our approach in implementing what we call the *path marching* algorithm. The core implementation is done in a shader program, and mixes both the ray marching and path tracing ideas. Lastly, we will explore how we solved the noise problem we obtain when generating low-sample frames in order to maintain high-fidelity graphics and still provide them in real-time.

4.1 System Architecture

The model we propose in this thesis implements all the different stages described in Figure 4.1. We will now explain the data-flow of our proposed rendering pipeline, dividing the explanation of each rendering stage into different sections.

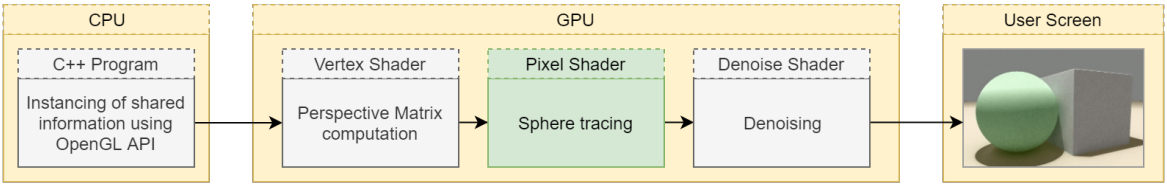

Figure 4.1: An overview of how data flows in our pipeline model.

4.1.1 Shared variables initialization

The first step of our rendering pipeline model is ran at the CPU level. This step consists of a program written in the C++ language. We decided to use C++ because it provides an object oriented style of programming that allows us to follow best coding practices. Additionally, it also provides the low-level manipulation necessary to work with buffers and other types of data that are fed to the graphics card API.

The API that we decided to use in order to communicate with the graphics card was OpenGL. Although other APIs like DirectX and Vulkan were also available, we decided to use OpenGL due to it's simplicity and the fact that it is a cross-platform open-source library.

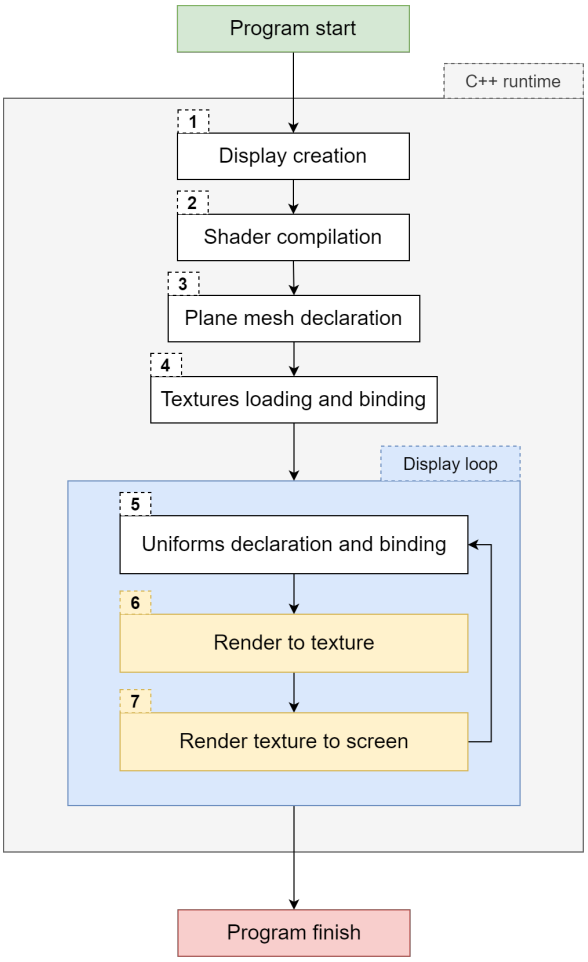

Figure 4.2: An overview of the CPU stage of our pipeline.

This program is in charge of initializing all the important processes in order to use the shader program in which the core implementation of our system lies at.

The initialization process starts by creating the display and binding it to OpenGL (1). We used the SDL 2.0 library for window management and communication with the OpenGL

API. We decided to use SDL because it is an open-source cross-platform solution that is easy to use and that works natively with C++ [23].

After this first step, we compile the shader code that is written in GLSL (2). When writing shader programs for the graphics card to run, these must be first compiled at the CPU level. Later on, the compiled information is sent to the GPU for it to process and run.

Afterwards, we move to step three where we create a simple quad mesh that consists of two triangles (3). This quad mesh occupies the whole camera screen, and is where we are going to paint the rendered scene. This technique is commonly used when implementing pixel shader based rendering engines that do not take use of vertex space based scenes such as rasterization.

At the next step, we load texture data from .png files and bind that data to the GPU (4). Two textures are used in the context of our rendering model. The first is a blue noise texture which acts as a normally distributed pseudo random number generator from which we are going to sample random values in our shader. This random sampling technique will be explained in further sections of this chapter. The second texture serves as a memory texture where we store the previous rendered frame. This texture is utilized for a temporal accumulation based de-noising solution described also in a future section.

The following steps (5,6,7) all take part of the display loop. The display loop is a loop that is ran at every frame until the user display is closed by the user. In it, the information related to the camera position, orientation and other properties is calculated and binded to the shader programs through the use of uniforms at each iteration (5). Uniforms are variables that can be shared between the CPU and the GPU and accessed at the shader stage.

Afterwards, at each iteration of the display loop, the ray marching algorithm is ran in the shader and the output render stored in the memory texture (6). Finally, the contents of the memory texture are rendered to the quad mesh that occupies the whole screen (7).

4.1.2 Vertex Shader

From now on we will be describing the rendering stages that are ran at the GPU side.

The vertex shader stage is a necessary step in our rendering pipeline, since the vertex shader is also required in the modern OpenGL pipeline. In this stage, we declare all varying variables using the previously uniform values that are passed by the CPU.

A varying is a variable that is used in order to share information between shader programs.

Additionally, we compute the camera matrix based on the scene camera's right, up and front three-dimensional vectors. Since we only have four vertices in our scene (the ones from the

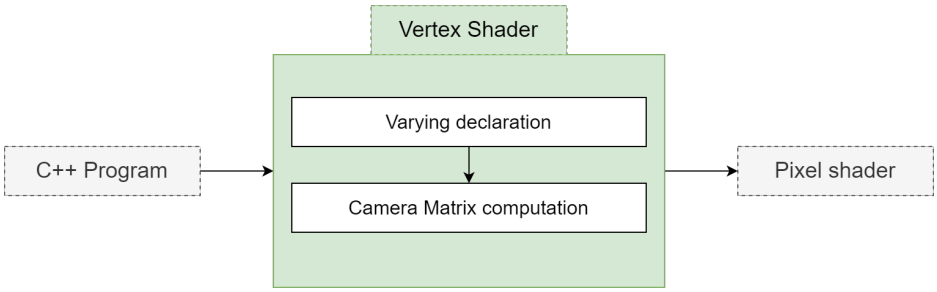

Figure 4.3: An overview of the vertex shader stage of our pipeline.

quad mesh), this computation will only be done four times in each frame rendered.

By applying this computation at an earlier stage, we are able to mitigate performance issues that doing this computation in the pixel shader would create, since every computation in the pixel shader is done once per pixel, per rendered frame.

4.1.3 Pixel Shader

The core implementation of the proposed system is built at the pixel shader level. In this section, we will detail the implementation of the ray marching based path tracing rendering engine that we propose as the main contribution of this thesis. In order to guide the overview, we will use numbers to refer to the different steps shown in Figure 4.4.

Our pixel shader implements a path tracing real-time engine. Contrary to a common path tracer which utilizes the ray tracing technique in order to render photo-realistic graphics using the CPU, we ported this implementation to the GPU side and decided to use a small number of samples and bounces in order to provide the path tracer output in real-time for the user. Consequently, we had to implement multi-sampling. The sampling loop can be visualized in red in Figure 4.4.

Note that each of the following steps that are going to be described are called once per pixel on the user screen. This logic is described inside the green container in Figure 4.4.

Calculating ray direction

The first step of our system is to calculate the ray direction at each pixel coordinate (1). For this matter, we utilized the following equations:

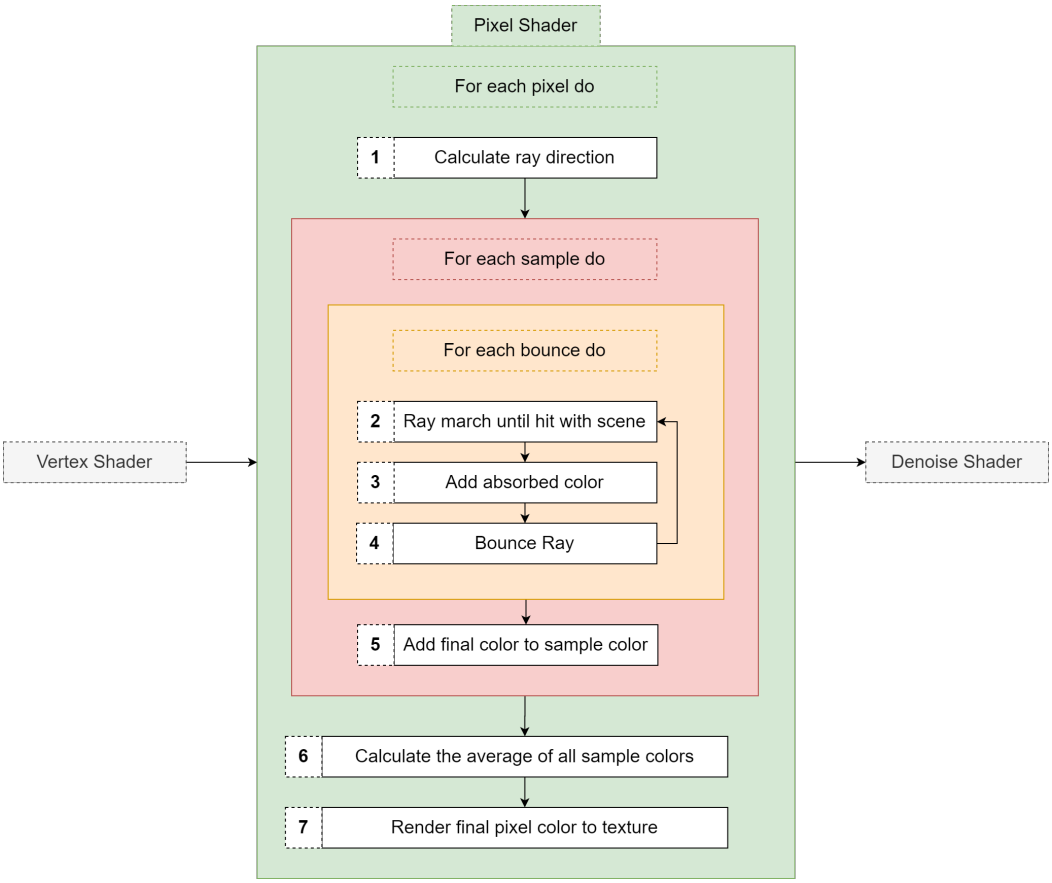

Figure 4.4: An overview of the pixel shader stage of our pipeline.

$$r_x = p_x - s_w/2$$
$$r_y = p_y - s_h/2$$
$$r_z = s_h/tan(radians(fov)/2)$$

Where r_x, r_y and r_z are the ray's x, y and z vector components, p_x and p_y are the current pixel screen coordinates and s_w and s_h are the user screen width and height. The camera matrix that was previously calculated in the vertex shader stage is then applied to the normalized ray vector coordinates in order to obtain a normalized ray direction at each pixel.

This computed information is later used at the ray marching step in order to find the intersection of each ray with the scene.

Multi-sampling

We decided to implement the multi-sampling technique in order to reduce output noise from our path tracer when modelling diffuse surfaces.

Contrary to reflecting a ray that hits a perfectly specular surface, for example a mirror, when a ray hits a diffuse surface the light rays scatter around in an uniform manner due to surface imperfections.

Since we want to model these imperfections, we utilize the monte-carlo approach in order to estimate this light behaviour quickly. By using this approach of selecting a random ray direction and repeating the marching algorithm on the selected ray, there will be visible noise in the rendered image.

Multi-sampling, as the name indicates, refers to the approach of repeating the whole ray tracing/marching process again while selecting another seed for our random generator. Since the reflection direction computation depends on the random generator we have, we will obtain different noise patterns in each different sample we process. Consequently, if we mix multiple samples by applying the average color of each pixel to the final image, we will greatly reduce the noise in the final render. However, this approach causes computation complexity to also grow linearly with the number of chosen samples.

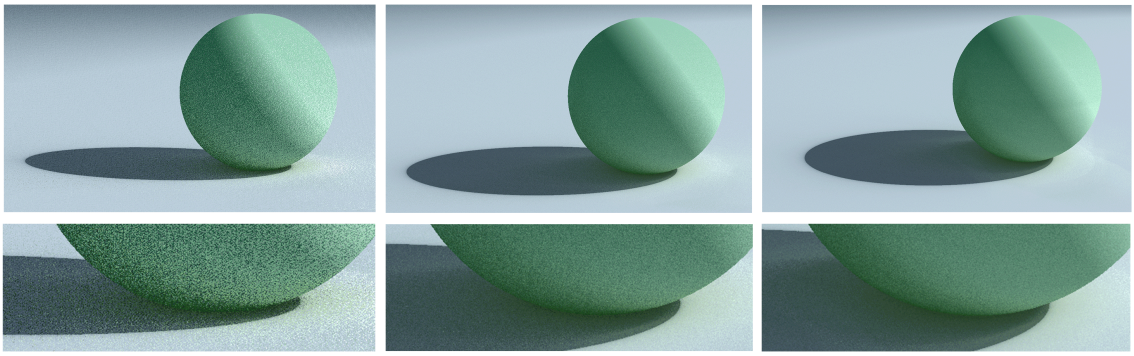

Figure 4.5: A comparison of sample count used in our rendering model. Figures are 1, 8 and 16 sample renders of the same scene, from left to right respectively. Notice that we start to get diminishing returns past the 8 sample count mark.

Representing the 3D scene

In order to represent the scene we want to render, we decided to take a simple approach and declare the objects of a scene statically in the pixel shader program. This does not mean that objects are not dynamic in terms of movement, just that there won't be any dynamically created or erased elements in our scene.

Our world representation is based on the previously introduced distance field mathematical representation of domains, and uses signed distance functions to represent solid geometry [31].

In our implementation, we made use of GLSL structs in order to maintain code readibility and usability. We use structs to represent objects in the scene. Each object has the following information:

- Center - vec3 that represents the center point of the object.
- Size - float that represents the size of the object.
- Type - int that represents the type of solid the object is.
- Albedo - vec3 that stores the RGB values for the color of the object.
- Emission - double that stores how much light an object emits.
- SurfaceType - int that represents what type of surface the object has such as diffuse, specular or refractive.
- Id - int that is an unique identifier of the object in a given scene.

We designed a structure that allows to represent all the solids we want to render in the most general way possible, since we know that all solids will have a center and a size.

The implementation of what the size dimension represents varies from solid to solid. In the case of a sphere the size represents its diameter. On the other hand, for a cube it represents its side length. Moreover, this object structure may also represent non-solid geometry objects such as three-dimensional fractals.

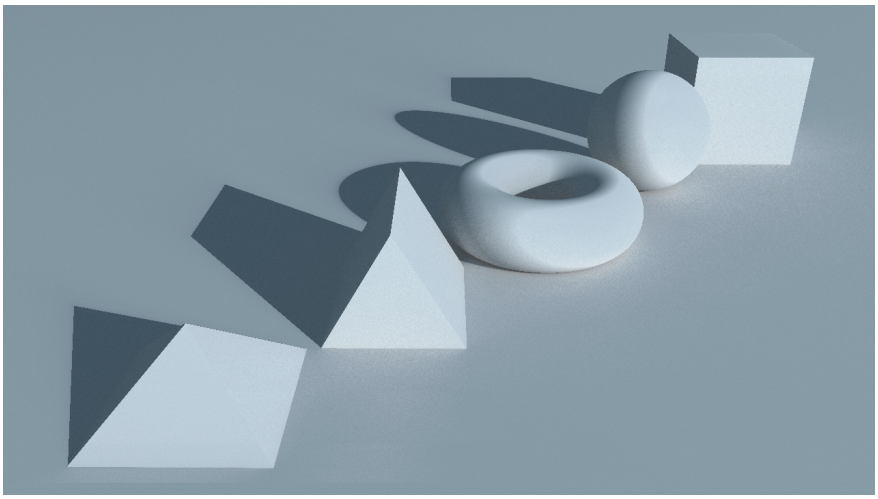

Figure 4.6: A showcase of the types of solids we can render in our model.

Finally, we also added support for primitive combinations of geometry. The union, subtraction and intersection of two objects are the combination operations we implemented.

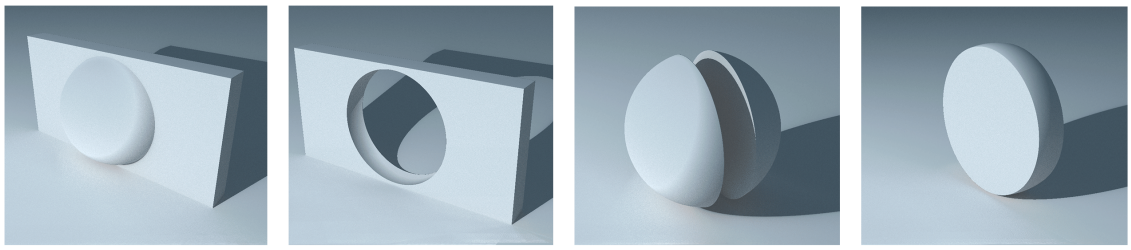

Figure 4.7: An example of the shapes that can be created with the simple use of a box and sphere geometry by using geometry combinations. Images rendered using our system.

Ray marching loop

The ray marching loop is represented in steps 2, 3 and 4 in Figure 4.4. This loop consists in calculating the intersection of each ray with the scene (2) by marching the ray until the distance from it to the scene is less than a pre-defined ϵ. Note how this method is different from the ray tracing method, where you test intersections with each object in the scene directly using mathematical intersection functions for each different type of solid.

After the marching step, the color of the hit object is mixed with the current pixel color (3) we are executing the ray marching algorithm on. The mixing factor varies depending on the number of bounces. In other words, the more bounces, the less the color of the hit object contributes to the final pixel color.

The last step of the ray marching loop is to calculate the reflection direction of the ray, which is a new ray called the bounce ray (4). Finally, if the number of bounces we are iterating over is less than the pretended amount of bounces we repeat the same logic with the new calculated bounced ray. When we reach the number of bounces we want, we break out of the ray marching loop and are ready to finalize our render process.

Note that in order to calculate the new bounce ray direction, we must know the normal vector of the surface we hit. Normally, when using a vertex-based approach such as the way objects are represented in a rasterization scene, this normal information is stored in the object model file. In contrast, when using a distance field based approach to represent our objects we don't typically store that information in a file. Instead, normals are calculated at run-time when a ray hits a scene object.

In order to calculate a surface normal, we use a three-point derivative of the object's SDF at that same surface.

After the calculation of the normal vector to the surface the ray hit, we select a random reflection ray based on the normal direction we computed. In order to sample our random vector direction in a uniform manner, we use a blue noise texture.

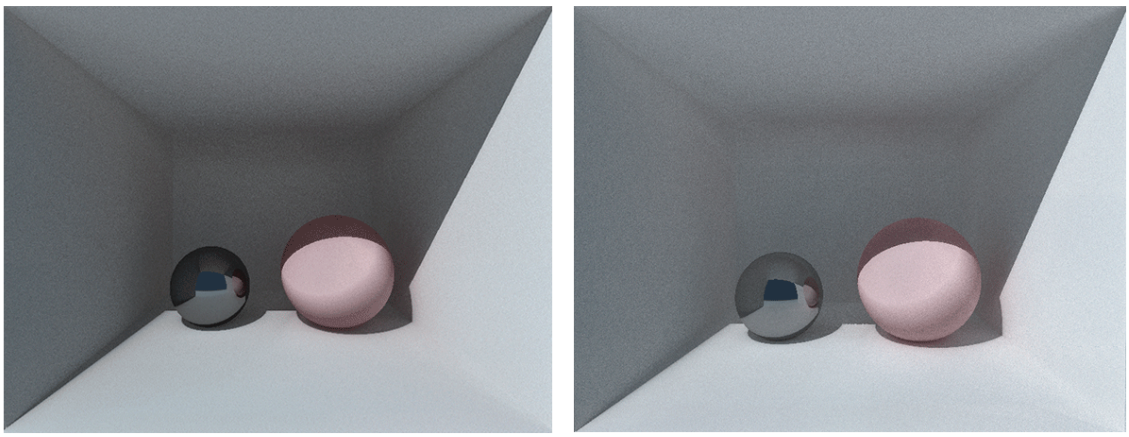

Figure 4.8: 3 samples (left) vs 8 samples (right) render of the same scene using our rendering model. Note how light gathers more in the 8 sample render.

This blue noise texture stores random RGB values that are uniformly distributed. By sampling each pixel of this texture based on the coordinates of the current pixel and bounce depth we are iterating, we are able to achieve a uniform noise distribution in our renders.

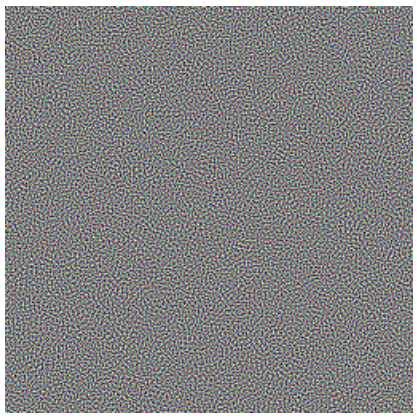

Figure 4.9: Visualization of the blue noise texture used to sample uniform random values in the pixel shader.

Averaging sample color

After the main ray marching loop, we mix each calculated pixel color from all samples (5) (6). All samples are treated equally. This means that we compute the average of all sample colors to generate the final pixel color that we are going to render.

Rendering to texture

Finally, the previous frame output of our path marching engine that is stored in the memory texture is read, and the current rendered frame is blended at a low percentage rate and stored again in the same texture in order to emulate the *temporal accumulation de-noising* technique. We will explain what this technique is and how it is achieved in the next section of this chapter.

4.1.4 Denoise Shader

The denoise shader stage is the last rendering step of our proposed pipeline model. In this section we will explain the whole denoising process and how the denoise shader operates over the memory texture we have previously mentioned.

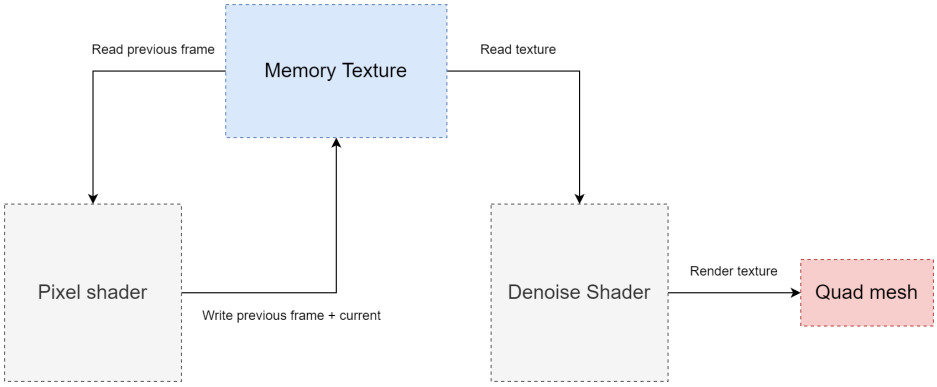

Figure 4.10: An overview of the proposed denoising pipeline.

The purpose of the denoise shader is to remove noise that is generated by the path marching algorithm. Our denoising process is capable of removing noise from single sample image renders in real-time, providing renders that are significantly cleaner than when not using a denoiser. See Figure 4.11 where we compare the raw output against the denoised output.

The denoiser we built uses the *temporal accumulation technique*, which exploits temporal properties of real-time rendering and is based on previous work by Daniel Scherzer [37] on this domain.

The technique consists in storing a buffer of previous n frames and mixing them together in real-time to create a final image.

The same technique we are applying could also be called *temporal sampling*, since we are effectively mixing n samples into one over time. Nevertheless, we are not doing this in the same frame. Due to this approach, we can maintain a real-time experience for the user while providing him the the best fidelity graphics.

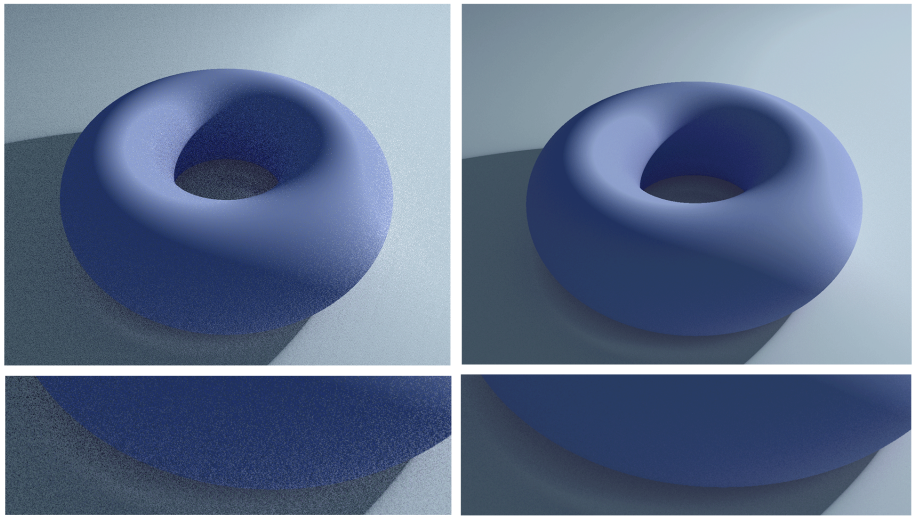

Figure 4.11: A visualization of the temporal accumulation based denoising technique used in our system. On the left image we have a single sample noisy render. On the right, we have the same single sample render, but accumulated over 20 frames.

The downside of this denoising technique is that when the user moves the camera, we must throw away previously computed frames. This causes noise to be visible on the frames in which the user is interacting with the scene. Additionally, we can't denoise dynamic scenes. If we try to apply this technique over a moving object, we get a blurred render like the one seen in Figure 4.12.

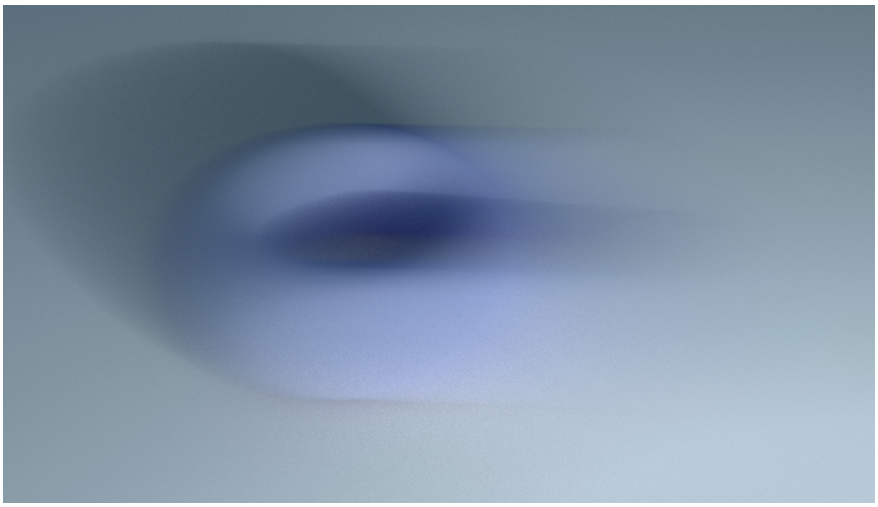

Figure 4.12: Temporal accumulation denoising applied to a moving object.

In order to implement the architecture that is explained in Figure 4.10, we utilized OpenGL Framebuffer Objects. An FBO, short for framebuffer object, allows developers to render offscreen without disturbing the main screen.

An FBO with the memory texture attached is binded to the pixel shader as the default READ and WRITE location for pixels. This means that all the output of the pixel shader will be rendered to the attached texture instead of the main screen.

Afterwards, the denoise shader reads the stored pixel data from the memory texture and renders it to the quad mesh that occupies the whole user screen. This logic is detailed in Figure 4.10.

4.2 System Limitations

In this section we will go over the main limitations of our system. While we tried to reduce optimization *hacks* to a minimum, some optimization techniques were necessary in order to provide a realistic real-time experience for the user.

4.2.1 Next-event estimation

One topic that we have yet to describe our approach on is emissive objects. Following the common approach when implementing a path tracing engine, we decided to allow the existence of emissive objects.

An emissive object is an object that emits light. These objects allow light to shine on our scene, thus lighting the environment.

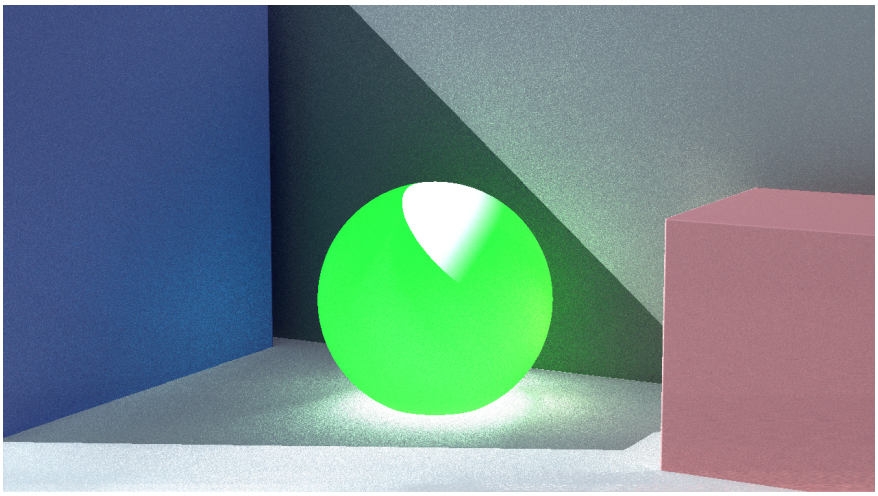

Figure 4.13: A render of an emissive green sphere produced by our rendering model.

Our implementation of this type of emissive objects is simple: if an object has an emission term that is not null, we multiply the color of the rays that hit that object by the emission term.

While this approach is physically correct, a big problem arises from it. Since we are depending on three-dimensional objects in order to light our environment, the smaller the emissive object is, the lower the probability of a random ray reflecting of a surface and hitting the emissive object is as well. Thus, we end up with renders like the one seen in Figure 4.14 where we get extremely noisy illumination in the left image.

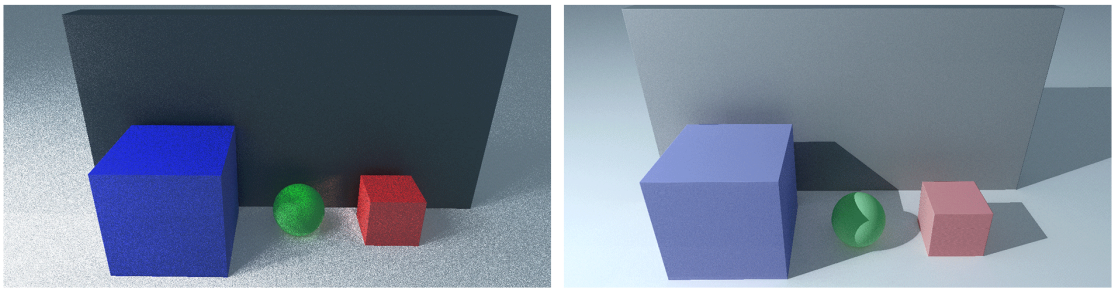

Figure 4.14: A comparison between a raw lighting render based only on emissive objects (left) and a render that utilizes the next-event estimation technique (right). Both images rendered using our system.

In order to solve this problem, we used a technique called next-event estimation [13].

This optimization technique consists in calculating the angle between the direction to the light and the normal of the surface at each ray hit with the scene. Similarly to the Blinn-Phong lighting model, we combine this multiplier with the sample color of each pixel in order to obtain a render that converges to ground truth in substantial less samples. Since in this approach we only need to calculate the direction to the scene light, we decided that the light object is a simple point with no mass. We will refer to this light point as our global light.

While this technique is extremely beneficial in order to produce real-time renders that are also visually pleasing, we decided to include it in this work as a limitation simply due to the fact that it is an approximation of how light behaves in the real world.

As a result of using this optimization technique, we must shade the surfaces that are occluded from the light in another manner, creating hard shadows such as the ones seen in Figure 4.14 in the right render. While this light behaviour does not work for soft light scenes, we believe it produces a type of light that resembles sunlight correctly, since it shines bright and casts hard shadows. Additionally, the computational complexity of this approach grows linearly with the amount of global lights there are in the scene.

Finally, we want to note that even though we used the next-event estimation technique for portraying global light behaviour, we also allow the existence of local emissive objects like shown in Figure 4.13.

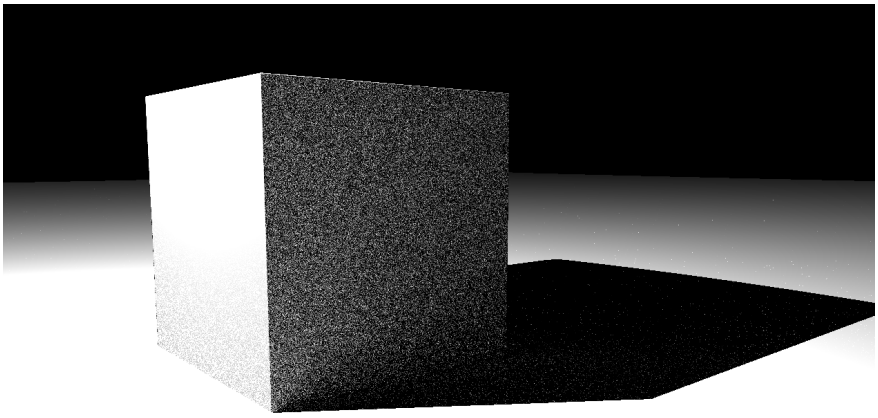

Figure 4.15: Single sample render produced with the next-event estimation technique. Non occluded surfaces quickly converge to ground truth, while occluded surfaces will require more samples since their lighting is based solely on bounced rays.

4.2.2 Scene declaration

Another main limitation of our system relies on the way we define our scene environment. In order to build a new environment using our model, the developer must change the scene objects declaration inside the shader program. Additionally, we can only represent a scene that is a combination of the solid geometry we support such as cubes, spheres and prisms.

Lukasz Tomczak proposes in his work [39] an efficient method of storing 3D mesh data in it's distance field representation by utilizing a texture and sampling that same texture in the signed distance function of our scene. Succinctly, the process consists in bounding the mesh we want to render in a three dimensional box, measuring the distance from each point in space to the mesh and storing that distance information in a three-dimensional texture. Later, in the pixel shader, we sample our distance field estimate directly from the 3D texture, instead of utilizing the pre-defined solid geometry signed distance function evaluators.

While our main focus for our work is not the representation of complex geometry such as meshes, Tomczak's work would greatly benefit the way we represent our three-dimensional scene in terms of design and developer experience.

4.3 Summary

We were able to design and implement a model that does not depend on specific hardware or APIs. This is extremely important in order to provide every user with the same experience

despite his computer's hardware.

Furthermore, we developed a sphere tracing based model which can represent both distance fields and common geometry. This greatly extends the domain areas in which our system can be successfully applied.

In the next chapter, we will be comparing our system against other state of the art methods in order to evaluate the results of our solutions to the limitations of current state of the art methods.

Chapter 5

Results and evaluation

In this chapter, multiple renders produced by our path marching rendering model are presented. Additionally, we will compare the proposed system for this thesis against state of the art rendering methods such as monte carlo path tracing and rasterization based models. The comparison will be extended to multiple quality levels such as time to render, graphical fidelity, implementation complexity of the algorithm and applicability of the rendering method in current software applications.

System used for benchmarking

Before diving into each case study of our rendering model, we want to describe the specifications of the system that was utilized in order to benchmark all the study cases that follow this section.

In Table 5.1, we give a detailed account of the specification of the software and hardware of the machine we used for all test cases documented in this work, as well as for the render images presented.

Finally, we want to note that all the results for rendering time and graphics fidelity presented in the following sections are rendering in 720p resolution, unless specified otherwise.

Operating System	Windows 10.1 Version 1809 (build 17763.1098)
CPU	Intel i5-6600k
GPU	MSI Radeon R9 390 8GB GDDR5
GPU driver version	Radeon Software Adrenalin 2020 Edition 20.1.3

Table 5.1: Specification of the system used for benchmarking and comparison studies.

5.1 Selecting the optimal bounce count

The number of bounces that we take into account each time we march a new ray is critical to the image fidelity we want to provide. This parameter introduces a compromise between image fidelity and the time to render of the final image: the higher the number of bounces, the more realistic the image will look but will be more expensive to compute.

In order to start this chapter, we will introduce a case study on finding the optimal ray bounce count for our rendering model. This case study explores how we can provide real-time graphics while maintaining a real-time performance by changing the ray bounce count parameter.

For this case study, we decided to render multiple different images of the same simple three-dimensional scene. The scene consists of three colored spheres on top of a white plane, with a white wall as their background. Furthermore, we decided to use a specular material with a specular roughness of 0.3 for all the spheres, since this type of material better reflects the changes in the number of ray bounces.

In Figure 5.1, we visually compare the differences between different ray bounce numbers.

Looking at the visual results, we observe that images rendered with a bounce count of over 4 ray bounces are mostly identical. The only minor difference we observed in high sample counts, namely 16 and 32, was how ambient light gathers around the scene in a more intense manner. This phenomenon we just described goes in line with the expected behaviour from a big ray bounce count.

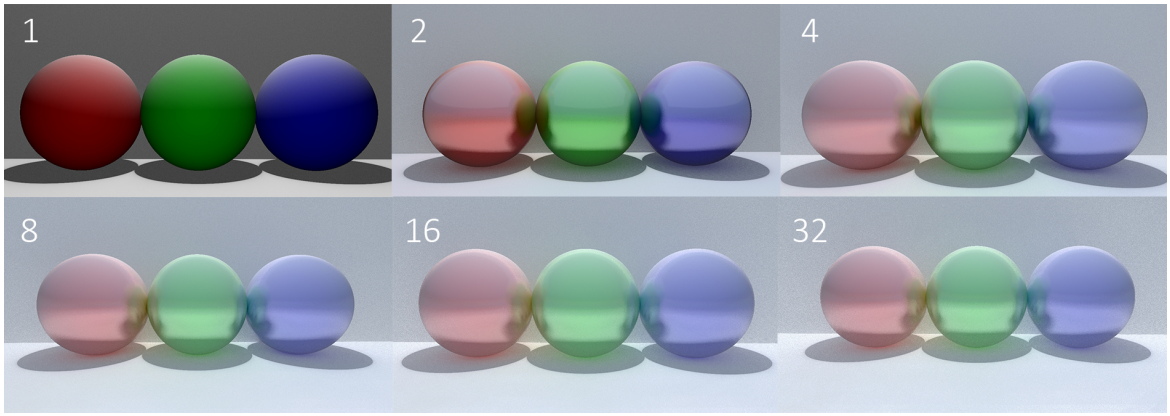

Figure 5.1: Visual comparison between different bounce counts for the same scene. The number of bounces used to render each image is displayed at the top left.

In order to understand the differences each ray bounce count imposes in the time to render of the final image, we decided to plot the different data we measured from the multiple renders in Figure 5.1. The visualization of how each bounce count affects time to render is displayed

in Figure 5.2.

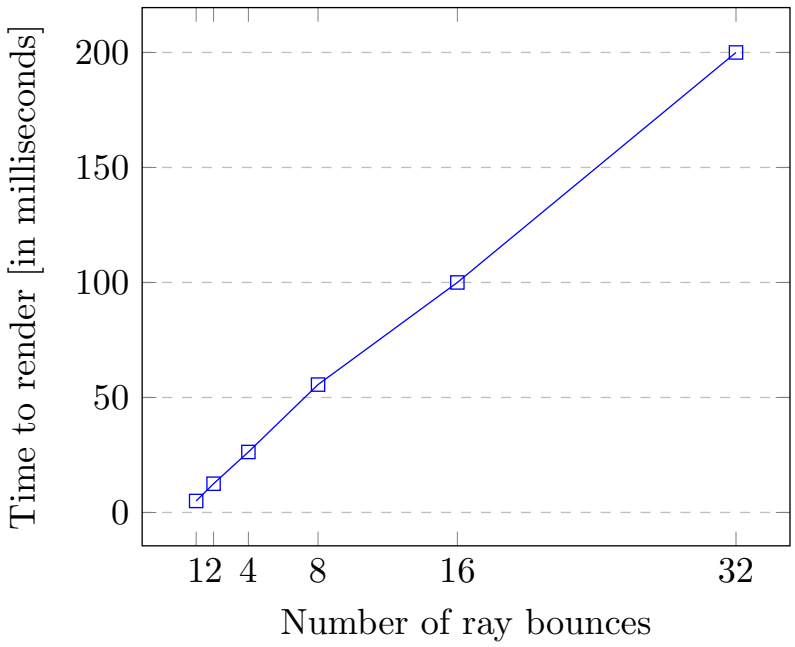

Figure 5.2: Data visualization of how changing the number of ray bounces affects the total rendering time of our model for the renders shown in Figure 5.1.

Previously in this work, we introduced our definition of real-time graphics. To re-iterate over the definition we decided upon, real-time graphics must be provided at 24 frames per second or faster. In other words, the time to render of our method for each frame should be under the $41.6ms$ mark if we want to provide real-time graphics for the user of our application.

From Figure 5.2, we observe that in order to provide each frame in less that 41.6 milliseconds, we must pick a number of ray bounces that is smaller than 7.

From this case study, we conclude that our range of optimal ray bounces is between the 4 and 6 range for a simple scene. Since we don't observe major differences between the 4 and 8 bounce count renders presented in Figure 5.1, we decided to set our default number of bounces to 4 for the rest of the case studies of our model.

5.2 On samples and temporal denoising

In this next section, we will evaluate the effectiveness of the denoising algorithm used in our application.

In order to do this evaluation, we must first understand that the temporal denoising technique works in the same way as the multiple-sampling technique but applied over multiple frames.

A visualization of this temporal method is presented in Figure 5.3, where we can observe the progressive denoising applied over time to a render of fractal geometry.

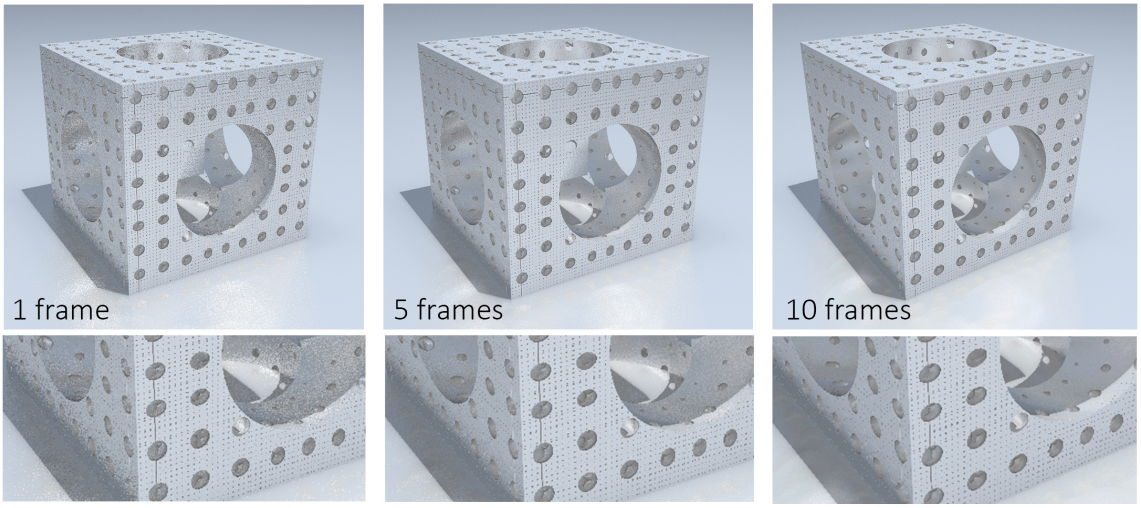

Figure 5.3: A comparison between noise accumulation in different count of total frames elapsed.

Applying the denoising process seen in Figure 5.3, we are able to provide the user with a time to render of $76.9ms$ between frames. While this is only 13fps and is surely beneath our limit for real-time graphics rendering, rendering the same scene using a multi-sample approach would require 10 samples for it to converge to the last render seen in Figure 5.3 (10 frames accumulation). The suggested 10 sample render would require 10x the rendering time of the denoised render, thus providing the each frame at $770ms$, rendering the application unusable from a user interaction standpoint.

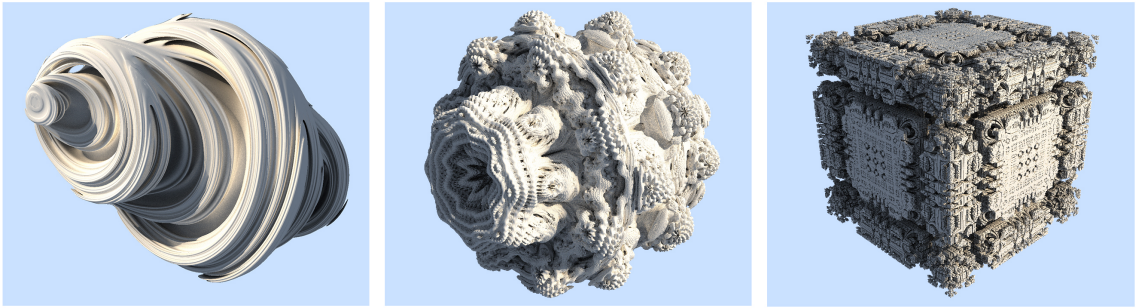

Figure 5.4: Multiple fractal geometry renders produced by our rendering model.

While 13fps is below our real-time standards, the complexity of the signed distance function of each fractal presented in Figure 5.4 makes exploring this type of geometry a bit slower when compared to simple scenes composed by simple geometry. Nevertheless, we believe that the ability to interact with the fractal geometry is extremely useful to produce artistic

renders.

5.3 Exploring different materials

We will now dive into the different materials that are available in our path tracing system. For each type of material, we will present several renders to show how each material behaves visually in a given scene.

Additionally, we will compare the renders of each material produced by our system against Blender's state of the art Cycles path tracing engine. The Blender application version we used was 2.1.

For Cycles's configuration, we decided to go with 128 samples and 4 diffuse light bounces. For our rendering model, we utilized 4 light bounces with single sample frames accumulated over a second.

5.3.1 Diffuse material

In Figure 5.5, we present a render of a simple scene composed by a sphere and a plane. Both the objects in the scene have the same type of diffuse white material. We decided to use a sphere for this visualization since it is an object that greatly details every light effect.

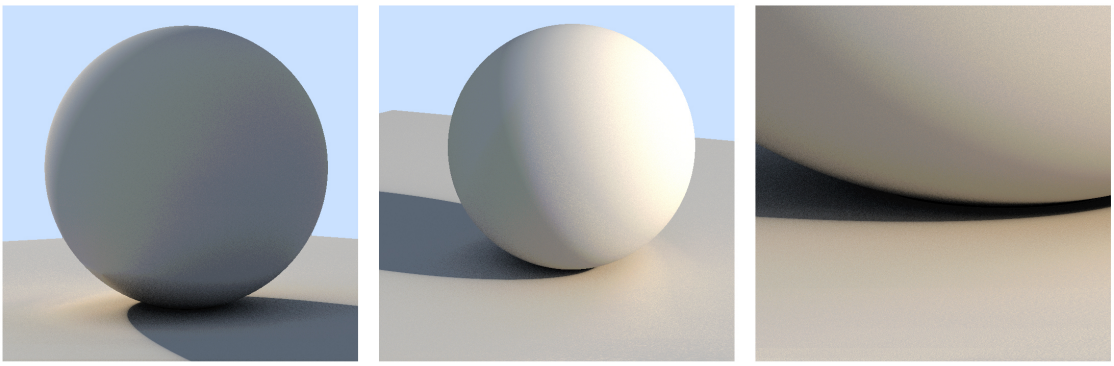

Figure 5.5: Diffuse sphere and plane renders produced by our rendering model. Time to render of 7ms (142 fps).

On the other hand, we present our comparison renders of the same scene in Figure 5.6.

Looking at both render results from our model and Blender's engine, we observe that the most important light phenomena are indeed present: ambient occlusion under the sphere, specular highlights in the normal direction of the scene light, ambient light bleeding into the

scene's shadows and even the shadows that are casted from the sphere's own shadows to its own back surface (first render from the left in Figure 5.5).

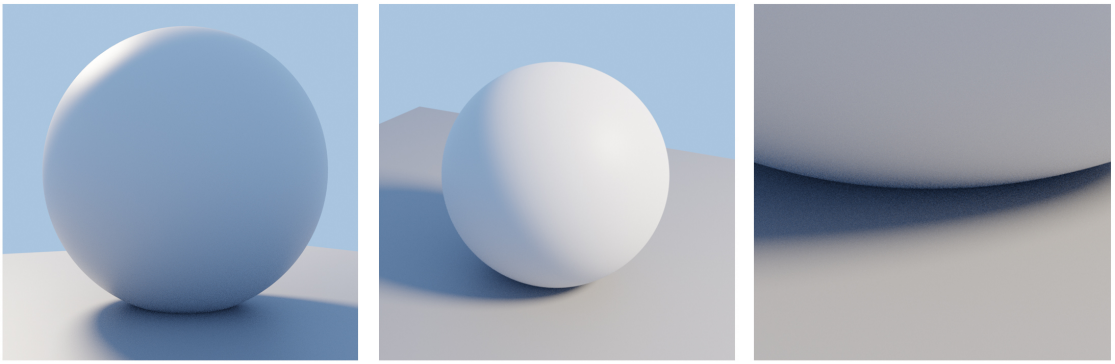

Figure 5.6: Diffuse sphere and plane renders produced by Blender's Cycles rendering engine. Time to render average of 2:09 minutes.

For this test experiment of modelling diffuse surfaces, the main visual difference we noticed between our model and the state of the art one was on the softness of shadows. To explain, our model uses a next-event estimation approach with an additional simplification that every scene global light has no dimensions, i.e. is a single point with three-dimensional coordinates. Consequently, we will never obtain soft shadows using our simplification approach.

There is a possible solution to this problem that relies on using the number of marched steps in order to shade the shadows accordingly. This optimization step was utilized previously in our ray marching test shader application created in Section 3.3 of this thesis.

In other words, we know that the closer a ray will pass by another object, the more marches the algorithm will run over that same ray. Using this global information, we can shade the shadows according to the number of marching steps that were executed over each ray.

A result of the application of this approach can be shown in Figure 5.7. Since this application increases the overall rendering times by 12%, we decided to not implement it in our final rendering model.

In Table 5.2 we present a render time comparison between the three distinct renders presented previously. From this table, we observe that our rendering model performs faster than Cycles in 4 times the order of magnitude of its rendering time.

While the difference is absurd, we must note that we are comparing a 128 sample render against a 1 sample real-time denoised output. The denoised output from our model is provided at an extremely low time to render, but it comes at the price of visible noise when the user interacts with the camera position.

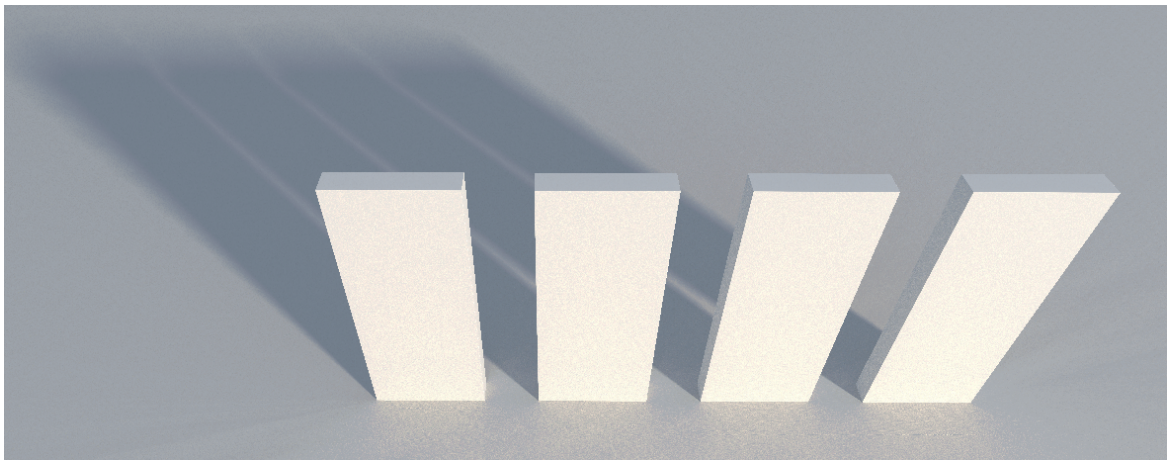

Figure 5.7: Ray marching dynamic soft shadows produced by our model. Render time increased 12% when compared to not computing soft shadows.

	Our Model	Blender 2.1 Cycles
Back of Sphere	7ms	2 min 9 sec
Front of Sphere	6ms	1 min 3 sec
Sphere detail	10ms	3 min 14 sec

Table 5.2: Comparing Figures 5.5 and 5.6 rendering times.

5.3.2 Specular material

The second type of surface material we have implemented in our rendering model is the specular material. Based on a parameter, we can configure the roughness of each specular surface in a given scene.

In Figures 5.8 and 5.9 we present the renders produced by our rendering engine and Cycles, respectively.

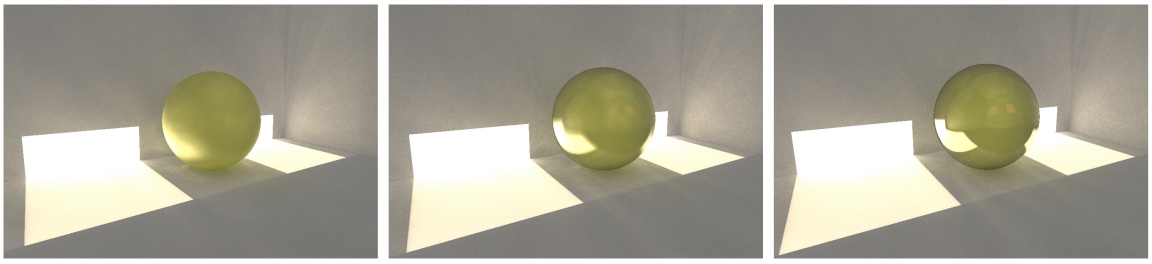

Figure 5.8: Specular material sphere render produced by our rendering model and delivered at 34fps.

In each figure, we present 3 different render images produced by the two rendering models.

Specular roughness	Our Model	Blender 2.1 Cycles
0.9 roughness	29ms	2 min 55 sec
0.5 roughness	27ms	2 min 50 sec
0.05 roughness	26ms	2 min 46 sec

Table 5.3: Comparing Figures 5.8 and 5.9 rendering times.

From left to right, each render represents a specular yellow sphere inside a room with a specular roughness of 0.9, 0.5 and 0.05, respectively.

For this evaluation test of specular materials, we can observe that our ray marching based representation of specular objects has the same visual behaviour as the one produced by state of the art rendering models such as the one presented in Figure 5.9: reflected illumination is captured and bounced around the scene in the same manner for specular surfaces.

Nevertheless, we believe our ray marching based path tracer to be a bit behind the state of the art path tracing engines when modeling smooth light bounces around the room represented in this test case scene.

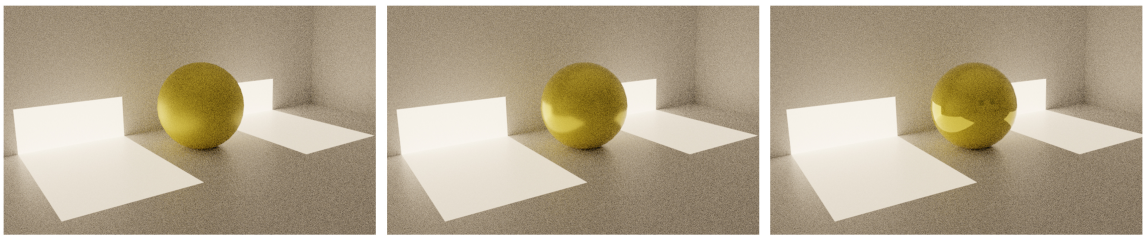

Figure 5.9: Specular material sphere render produced by Cycles. Average time to render of 2 min 51 sec.

Our model lacks the softness presented in renders of path tracing engines. This is due to the next-event estimation simplification technique we utilize in our model. We will explore the results of this limitation later when we evaluate emissive objects.

Alternatively, we must note the massive difference between the time to render of both models. While our rendering model lacks the light softness Cycles provided, it was able to deliver the images in Figure 5.8 in under 29ms, while it took more than 2 minutes for Cycles to produce a noisy render of the same scene.

5.3.3 Refractive material

For the last material comparison between our proposed rendering model and Blender's Cycles path tracing engine, we are going to explore refractive materials. Our goal with implementing

refractive materials in our model was to allow the representation of glass surfaces and objects.

Refractive materials are the *holy grail* of path tracing engines. The renderization of a refractive surface requires computing caustics. Caustics are the light rays that pass through refractive materials such as glass and water, gathering around specific spots in the scene.

Caustics are extremely expensive to compute: one must account for the forward ray tracing technique which consists in tracing multiple rays from a light source into the scene. This computation is extremely expensive. Consequently, path tracing engines typically only use the backwards ray tracing approach which consists in tracing each ray from the eye or camera of the scene.

When caustics want to be added, a technique introduced first by Lafortune [21] called bi-directional path tracing is usually implemented.

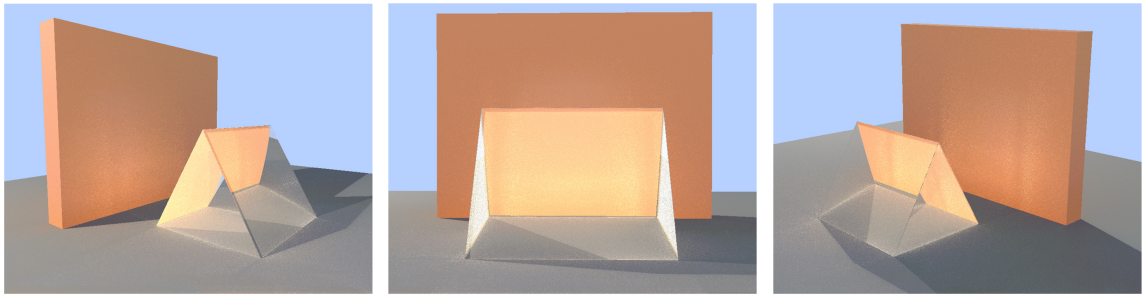

Figure 5.10: Refractive hollow prism render produced by our rendering model and delivered at 32fps.

Since we want to maintain an interactive experience for the users of our application, providing ground truth based caustics was not a possibility. Nevertheless, we implemented a simplification of the caustics light behaviour, shading each surface according to whether it is occluded by another refractive surface. This simplification step provided a good-looking and fast solution to the caustics problem.

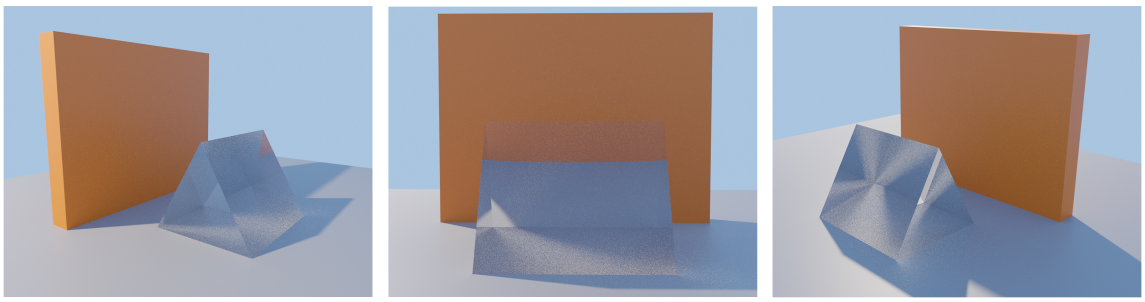

Figure 5.11: Refractive material render produced by Cycles using realistic caustics. Time to render varied from 40 seconds to 1 minute.

	Our Model	Blender 2.1 Cycles
Left Side	29ms	40 sec
Front Side	37ms	1 min
Right Side	31ms	59 sec

Table 5.4: Comparing Figures 5.10 and 5.11 rendering times.

Comparing Figures 5.10 and 5.11, we notice that the reflection and refraction behaviour is pretty similar, with the big differences being in caustics not having the same physical behaviour.

While the quality of the caustics produced by our method is far from ground truth, we are able to provide the renders in Figure 5.10 at over 32 frames per second, while the state of the art path tracing engine Cycles required one minute to render the same scenes.

In Table 5.4, we present render time comparisons between the two models for the same scene.

5.4 On global illumination and emisive objects

In this section we will introduce two study cases about global illumination.

5.4.1 Global illumination

The first study case is a render of a simple scene composed by two spheres inside a room with no roof. Let's use it to analyze and compare the effects of global illumination produced by our rendering model against Blender's Cycles path tracer.

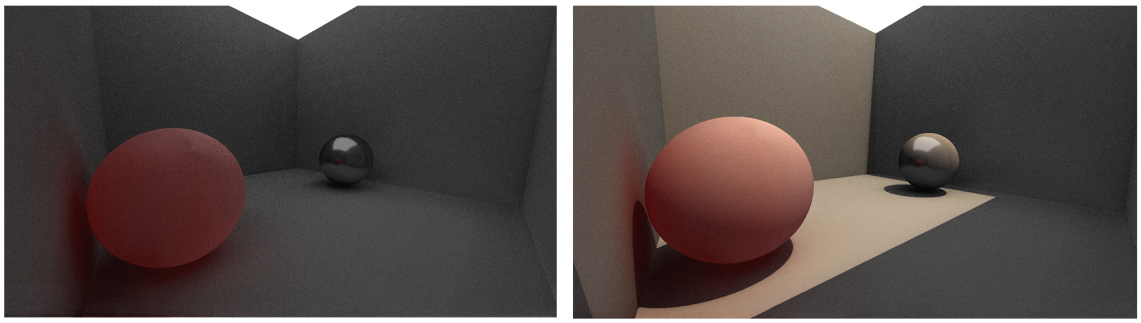

Figure 5.12: Global illumination effects compared without (left) and with global light (right). Both renders produced by our rendering model and provided in real-time at over 40 frames per second.

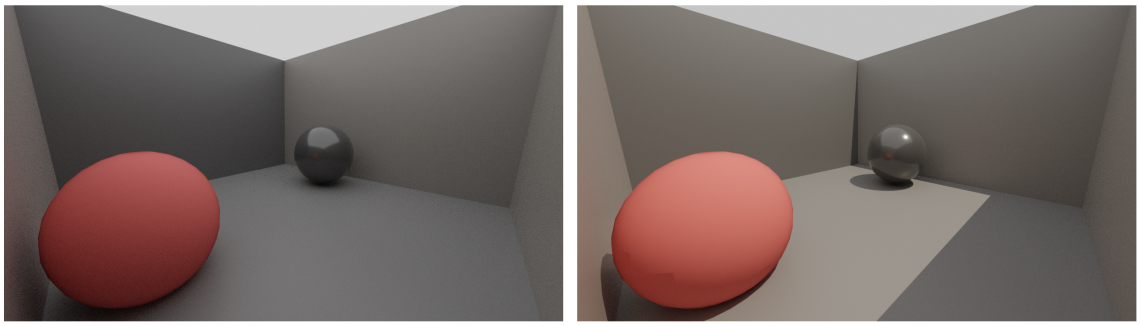

Figure 5.13: Global illumination effects compared without (left) and with global light (right). Both renders produced by Cycles, taking over a minute to render.

Analyzing Figures 5.12 (our model) and 5.13 (Cycles), we observe that the render produced by our proposed model has more noise. Nevertheless, we believe that for this scene our model provides an higher dynamic range of light when compared to Cycles's render: shadows are darker and the ambient occlusion factor is higher.

Although this makes the version rendered by our proposed model look more realistic, we can't say that this is a clear win, since these variables are easily configured in both models in order to match the wanted visuals without using extra computational power.

Secondly, we notice the limitation of the vertex approach to path tracing in Blender's Cycles renders shown in Figure 5.13: the red sphere's model is not completely smooth, thus creating small shading artifacts in the render that has global light (Figure 5.13, right render).

Thirdly, we notice the visual effects of a previously mentioned limitation in our system: the specular light effect that is created by the global light in the scene reflecting off the metallic sphere is not present in our model. This happens because we are utilizing the previously discussed technique called next event estimation in order to quickly converge the path tracing calculations that are present in the algorithm of our rendering engine. Consequently, we model all global lights by a single point. This makes it impossible to get their three-dimensional reflections, since global lights are not three-dimensional objects in our model.

Finally, we want to touch on the topic of render times for this case study. Following the same conclusion we had for previous case studies about rendering different types of materials, we observe that our model is capable of providing real-time realistic scenes that take state of the art path tracing engines more than a minute to render. This major difference between rendering times can be observed in Table 5.5.

	Our Model	Blender 2.1 Cycles
Ambient light	14.3ms	1 min
Global light	25ms	1 min 15 sec

Table 5.5: Comparing Figures 5.12 and 5.13 rendering times.

	Our Model	Blender 2.1 Cycles
Emissive objects	14.3ms	2 min

Table 5.6: Comparing the rendering times between our model and Cycle's emissive object scene renders displayed in Figure 5.14.

5.4.2 Emissive objects

The second study case is a comparison between two renders of a scene composed by multiple emissive spheres inside a specular, closed room. This study case is extremely valuable in order to compare the fidelity of the emission of lights in a given scene between our model and state of the art path tracing engines.

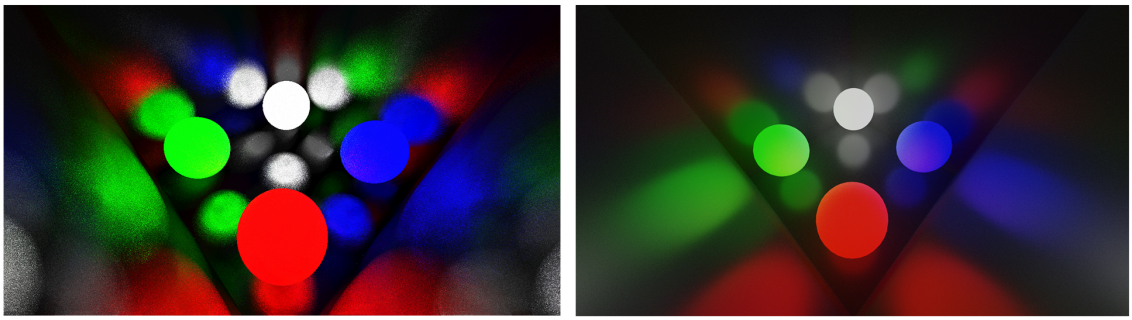

Figure 5.14: Comparison between our model (left) and Cycles (right) renders of a scene with emissive spheres inside a specular room.

By looking at Figure 5.14, we observe that the render produced by our model has a lot more noise when compared to the render produced by Cycles. Since we are not utilizing the next event estimation technique for three-dimensional objects, we suffer in render fidelity from this limitation and would have to significantly boost our sample count in order to obtain a render image that is similar to the one Cycles produced for the given scene.

Nevertheless, we still obtain the important light behaviour that is observed when the red light that is emitted by the red sphere hits the blue sphere and mixes together with the blue light in order to manifest a purple color.

	Our Model	Blender 2.1 Cycles
Mandelbulb 8-Powered	45.5ms	29 sec

Table 5.7: Comparing the rendering times between our model and Cycle's mandelbulb geometry renders displayed in Figure 5.16.

5.5 Fractal geometry

In this experiment, we will compare the capabilities of rendering three-dimensional fractals of our model against the state of the art path tracer used in Blender.

Since Blender has no native support for representing three-dimensional objects based on their distance fields, in order to create a mandelbulb fractal for our rendering comparison we had to follow the workflow displayed in Figure 5.15. The mentioned workflow was utilized in order to create a .obj file that represents a vertex based approximation of the mandelbulb fractal geometry.

In order to obtain our object file we first utilized Mandelbulb 3D [16], an open source program used to model and render the mandelbulb fractal. Since this utility has no option to export the model of the fractal as an .obj file, we had to export the model as a sequence of voxel slices stored as PNG files. Afterwards, we used an additional open source utility, Fiji [38], that allows us to export the sequence of voxel slices as an .obj file and finally import it directly into Blender.

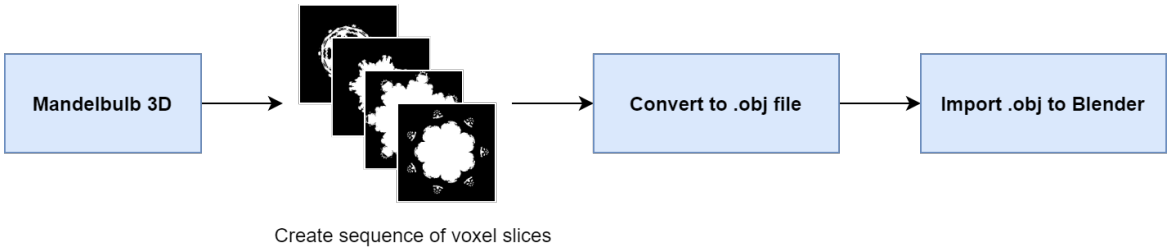

Figure 5.15: Workflow used in order to create a vertex-based object model of a mandelbulb fractal to be used in Blender.

In Figure 5.16 we present a comparison between the render produced by our model and Cycles for the mandelbulb three-dimensional fractal. Analyzing both renders, we come to the conclusion that the diffuse light softness provided by Cycles path tracing system is not achievable by our model. Since our model is only using a bounce count of 4, we are limited in terms of graphics fidelity to the render showed in the left image of Figure 5.16.

Nevertheless, we are able to provide the render shown at a frame rate of 22 fps, while Blender's Cycles rendering engine took half a minute to render the same scene. Additionally, due to the ray marching approach and how we compute the mathematical distance field for

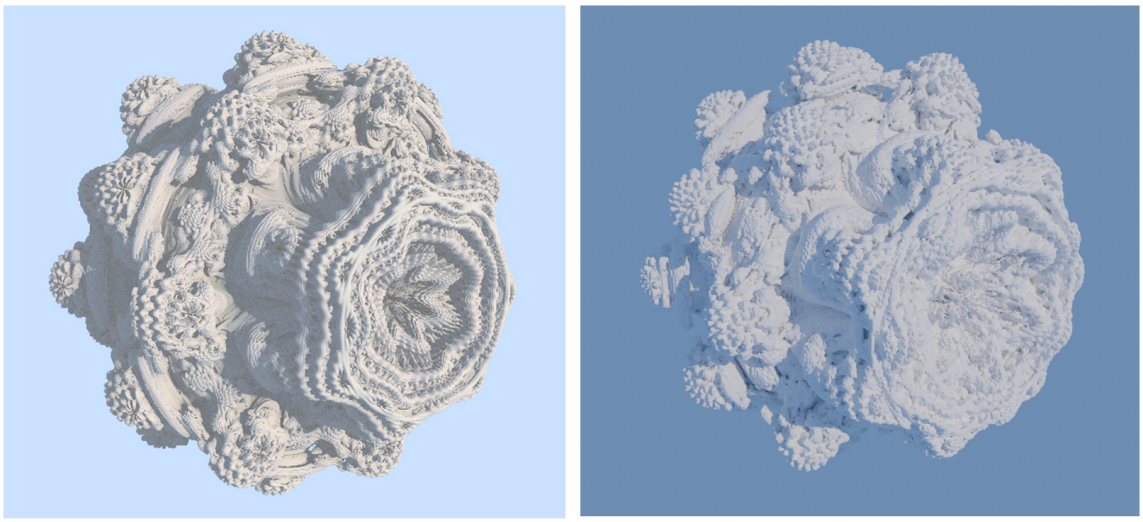

Figure 5.16: Comparison between the render produced by our model (left) of the Mandelbulb fractal against Blender's Cycles render of the same geometry (right).

the fractal geometry, we are able to add interesting coloring to our models without increasing the computation cost of the overall scene.

The technique we used in our rendering model in order to color fractal geometry is called orbit trapping. Orbit trapping consists in storing the minimum or maximum distance a point has become close to another given point in space during its recursive fractal mathematical computation. The application of this technique provides extremely interesting and pleasing visualizations of the fractal structures like shown in Figure 5.17.

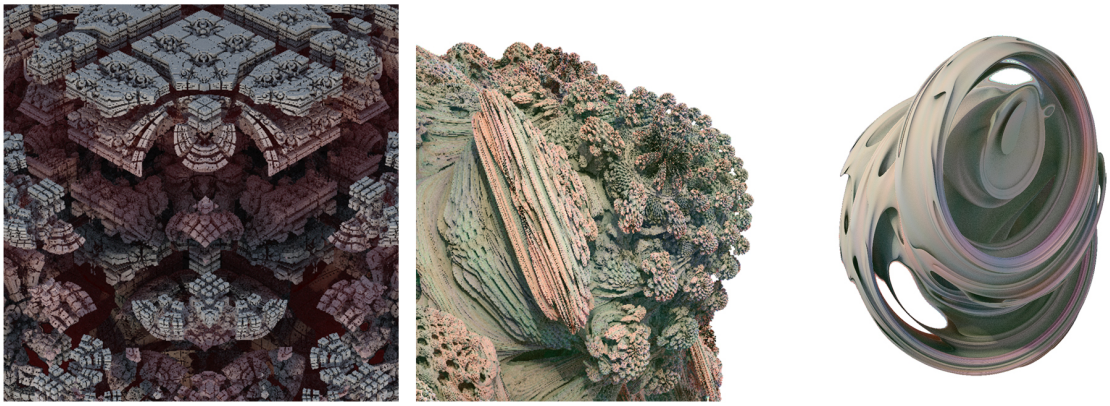

Figure 5.17: Visualization of the orbit trapping technique used to color three-dimensional fractals. Renders produced by our system.

	Our Model	Blender 2.1 Cycles
High fidelity graphics	yes	yes
Real-time rendering (over 24 fps)	yes	no
Path tracing denoising	yes	yes
Solid geometry	yes	yes
Distance field based geometry	yes	no
Textures	no	yes
Physics based materials	no	yes
Caustics	yes (simplified)	yes

Table 5.8: A comparison of available key features between our model and Cycles state of the art path tracer.

5.6 Evaluation of the results

To conclude this chapter, we want to go over the different results we obtained when comparing our model against Blender's Cycles state of the art path tracing engine. In order to conclude on this comparison, we decided to create Table 5.8, were we evaluate whether each model has the different specified key features for realistic and interactive graphics rendering.

To summarize the results, our model beat the state of the art path tracing engine Cycles in time to render by a long-shot for the same scenes. The quality of the graphics lacked in some domains, such as the previously mentioned softness of lighting, due to using lower ray bounce counts for our configuration parameters. Additionally, our implementation of realistic materials is not physics based in order to save on computation time. This contrasts with Cycles PBR materials, which in turn created a visual difference on the quality of diffuse and specular reflections between our comparisons.

Furthermore, our denoising method proved successful in delivering the same fidelity of graphics provided by renders of over 32 samples, but only using 1 sample per pixel/ray without any visible noise after accumulating frames for a second.

Next, our method proved to be able to represent both solid and distance field based geometry, although the latter not in real-time for some scenes, which is something that Cycles is not able to replicate.

5.7 Summary

To conclude on the results of this chapter, we were able to create a rendering model that is capable of providing photo-realistic real-time 3D graphics in significant less computation

resources than state-of-the-art methods (objective 2 of this work).

Nevertheless, we must also underline some of the limitations of our system. For instance, since the rendering of caustics was simplified, our model lacked the graphics fidelity generated by state of the art models for caustics rendering. Secondly, our model also displayed some artifacts when rendering specular materials inside closed environments.

In the next chapter we will further discuss these results and what we propose to work on for the future of our work.

Chapter 6

Conclusions and future work

In this last chapter, we will evaluate our work methodologies and results throughout the development of this thesis. Additionally, we will provide our opinion on what could be further researched for the benefit of the ray marching and ray tracing domains of graphics rendering. Finally, we will comment on what the future holds for high fidelity real-time graphics rendering and what the current limitations of this area are.

Researching and development

In terms of research, we must admit that it was not easy to find answers to the problems we faced. Most of the technical solutions we implemented in our thesis are combinations of years of research on the computer graphics domain and are heavily based on papers presented mostly in the Siggraph computer graphics conferences.

On top of that, since representing solid geometry in the ray marching domain is a topic that is usually not researched upon, we had to put together different approaches for the problems of modeling light reflections and emission, for example.

Developing the code-base and implementing the theoretical solutions was even more complex than researching them, since little to no examples are available. Some solutions were based in already implemented solutions available at *Shadertoy*, an online platform where people can code and share their pixel shader implementations.

Nevertheless, we noticed that most of the developers that code shaders tend to hide their solutions behind cryptic code statements. In order to avoid following their footsteps, we made sure that we wrote clean, readable code for anybody to understand in our C++ application and our GLSL shader implementations.

On the same topic of development, we also want to note that debugging shader code was one of the main bottlenecks of developing our rendering model. We have yet to find a real-time solution that allows us to style, lint and compile our shader code while using the Visual Studio Code platform we used to develop our system.

Evaluation

For the evaluation of our methods, we were limited by a single set of hardware for testing purposes, the one described in Table 5.1.

Event though recent developments in graphics hardware have allowed specific implementations of ray tracing, for example RT Cores on Nvidia graphics cards, we consider this approach as a fork that limits the standardization of ray tracing for real-time applications. As a consequence, this work was primarily tested using an AMD graphics card that does not support a custom ray tracing solution. This lack of additional testing may have induced a little bit of uncertainty in our comparisons, but it is not anything that should be considered critical to the conclusions presented in this work.

Furthermore, we also note that comparisons against rasterization-based renders were not introduced. We decided to not include these as we had no solid grounds on where we should put the basis of this rendering system for comparison purposes: whether to use baked lighting or real-time rasterization lighting solutions, for example, was a difficult ground to agree on.

Nevertheless, we are sure that a rasterization based approach would lead to better results for solid geometry in terms of computational speed, but would lack lighting fidelity when compared against our model and the state of the art path tracing engine used in this work.

Future work

Despite our conclusion that the model we proposed in this work is capable of delivering high-fidelity graphics in real-time, we are aware that it has a few limitations that have already been noted throughout this thesis. Consequently, we wish to specify a couple of improvements that our work would benefit from. These improvements can be seen as an extension of our work, further developing this domain of graphics rendering we explored.

Scene representation

Like we mentioned in Section 5.2.2, we believe that the main limitation of our system relies on the way we define our scene environment. In order to solve the limitation of having

to represent every object through a signed distance field estimator function, we propose a solution that is based on Lukasz Tomczak's work [39]. The solution consists in storing any 3D mesh data as a three-dimensional texture that stores the distance of each point to the surface of the object. Later, when running our ray marching algorithm, we sample the texture in order to obtain the distance of each point in space to the 3D model.

Solving this great limitation in our system would allow vertex-based scenes to be ported into the ray marching approach directly. Additionally, it would allow us to introduce the ray marching technique in the hybrid graphics pipeline model, where it could be used to calculate high-fidelity effects such as reflections and ambient occlusion in vertex-based rasterized scenes.

Using different APIs

Despite the fact that our rendering model was only tested in a single machine, we would also like to extend the results of our work into other sets of hardware. More specifically, we would like to develop a version of our rendering model built using the DirectX 12 graphics API. We suggest that this graphics API, coupled with nVidias's new graphics cards, would allows us to use the ray tracing turing architecture in order to compute our ray marching solution in less time.

Additionally, we would also like to test a Vulkan based implementation, since the Vulkan graphics API provides higher performance for the computation of graphics, when compared to OpenGL, which was the graphics API we decided to utilize in order to develop our system.

Final thoughts

Unfortunately, providing high-fidelity graphics in real-time is still a balancing act due to the lack of computational power available. Optimization solutions have to be researched in order for us to be able to experience truly realistic virtual environments and be able to interact with them in real-time. Most of these solutions are also simplifications to the way lighting is calculated, thus moving further away our renders from ground truth.

Even so, we have managed to develop a high-performant and realistic graphics renderer that uses path tracing in order to provide global illumination based renders in real-time. Through the implementation of this system, we believe that we have opened the door to ray marching based realistic rendering of solid geometry.

Additionally, we hope that this thesis allowed readers that are new to the graphics domain to understand the fundamentals of high-fidelity graphics rendering through our detailed

explanations.

Furthermore, we believe that we are going to see spectacular improvements on the research domain of computer graphics in the next years, due to the attention that recent GPU hardware development has brought to the scene. We already have the tools necessary in order to render virtual words that are indistinguishable from real-life. What is left is to have the computational power necessary to compute them in real-time.

Finally, we want to share that the distribution model of our system is open-source. The complete code-base of the proposed system in this thesis is available online as a Github project [8]. We invite everyone that has interest in this research domain to improve our rendering system through their contributions.

Bibliography

[1] Evans A. Fast approximations for global illumination on dynamic scenes. *ACM SIGGRAPH 2006 Courses*, 25:153–171, 2006.

[2] Michael Abrash. Quake's lighting model: Surface caching. 1996.

[3] Arthur Appel. Some techniques for shading machine renderings of solids. *AFIPS '68 (Spring) Proceedings of the April 30–May 2*, pages 37–45, 1968.

[4] Henrik. BARRÉ-BRISEBOIS, Colin. HALÉN. Pica pica nvidia turing. *Real-Time Ray Tracing sponsored session, SIGGRAPH*, 2018.

[5] Bartell et al. The theory and measurement of bidirectional reflectance distribution function (brdf) and bidirectional transmittance distribution function (btdf). *Radiation Scattering in Optical Systems*, 257:154–160.

[6] Ian Buck and Pat Hanrahan. Data parallel computation on graphics hardware. *Stanford Graphics*, 2003.

[7] A. Butt. Data parallel computation on graphics hardware. *Information and Communication Technologies*, pages 65– 65, 09 2005.

[8] Rúben Carvalho. Path marcher. `https://github.com/Arukiap/Path-marcher`, 2020.

[9] Robert L. Cook et al. The reyes image rendering architecture. *ACM SIGGRAPH Computer Graphics*, 21(4):95–102, 1987.

[10] Brian Curless. From range scans to 3d models. *ACM SIGGRAPH Computer Graphics*, 33(4):38–41, 2000.

[11] Pierre-Yves Donzallaz. Shining the light on crysis 3. *Game developers conference*, 2013.

[12] Cindy M. Goral et al. Modeling the interaction of light between diffuse surfaces. *ACM SIGGRAPH Computer Graphics*, 18(3):213–222, 1984.

[13] Hanika et al. Manifold next event estimation. *Eurographics Symposium on Rendering*, 34(4), 2015.

[14] John C. Hart. The visual computer. *The visual computer*, pages 527–545, 1996.

[15] Defanti T. A Hart J. C. Efficient antialiased rendering of 3-d linear fractals. *ACM SIGGRAPH Computer Graphics*, 25:91–100, 1991.

[16] Julius Horsthuis et al. Mandelbulb 3d software. `https://www.mandelbulb.com/`, 2009.

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

[18] Sergei Karmalsky. Global illumination in metro exodus: An artist's point of view. *Game developers conference*, 2019.

[19] Kehl et al. Coloured signed distance fields for full 3d object reconstruction. *British Machine Vision Conference*, 2014.

[20] J. Korein and N. Badler. Temporal anti-aliasing in computer generated animation. 17(3):377–388, 1983.

[21] Lafortune. Mathematical models and monte carlo algorithms for physically based rendering. *PhD thesis*, 1996.

[22] Samuli Laine and Tero Karras. High-performance software rasterization on gpus. *Nvidia Research*, 2011.

[23] Simple DirectMedia Layer. `https://www.libsdl.org/`, 2020.

[24] T. Nishita and E. Nakamae. Half-tone representation of 3-d objects with smooth edges by using a multi-scanning method. *Journal of IPSJ*, 25(5):703–711, 1984.

[25] Nvidia. Nvidia turing gpu architecture *whitepaper*, 2018.

[26] Jacob Olsen. Realtime procedural terrain generation. 2004.

[27] Yuriy O'Donnell. Precomputed global illumination in frostbite. *Game developers conference*, 2018.

[28] Carlos Parga and Sebastián Palomo. Efficient algorithms for real-time gpu volumetric cloud rendering with enhanced geometry. *Symmetry*, 10(4):125, 2018.

[29] Bui Tuong Phong. Illumination for computer generated pictures. *ACM*, 18(6):311–317, 1975.

[30] Timothy J. Purcell et al. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21(3):703–712, 2002.

[31] Inigo Quilez. Distance functions. `https://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm`.

[32] Inigo Quilez. Terrain raymarching. `https://www.iquilezles.org/www/articles/terrainmarching/terrainmarching.htm`, 2002.

[33] Inigo Quilez. Penumbra shadows in raymarched sdfs. `https://www.iquilezles.org/www/articles/rmshadows/rmshadows.htm`, 2010.

[34] Paul Read et al. Restoration of motion picture film. *Gamma Group*, pages 24–26, 2000.

[35] Mathieu Robart et al. Global illumination for advanced computer graphics. *Digest of Technical Papers - International Conference on Consumer Electronics*, 2008.

[36] P. Robert and G. Casella. *Monte Carlo Statistical Methods*. 2004.

[37] Daniel Scherzer et al. Exploiting temporal coherence in real-time rendering. *ACM SIGGRAPH ASIA Courses*, 24:1–26, 2010.

[38] Johannes Schindelin et al. Fiji Software. `https://fiji.sc/`, 2011.

[39] Lukasz Tomczak. Gpu ray marching of distance fields. pages 57–61, 2012.

[40] Li-Yi Wei. A crash course on programmable graphics hardware. *Microsoft Research Asia*, 2005.

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