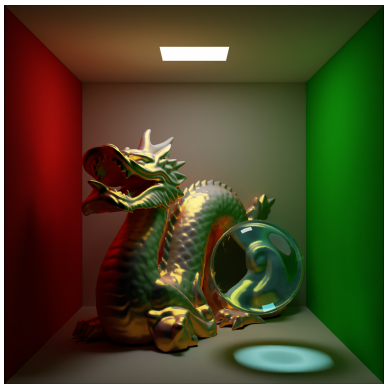
TÉCNICO LISBOA



Interactive Physics-Based Rendering with AI-Accelerated Denoiser

Gonçalo Filipe Dias Soares

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. João António Madeiras Pereira

Examination Committee

Chairperson: Prof. Paolo Romano

Supervisor: Prof. João António Madeiras Pereira

Member of the Committee: Prof. Fernando Pedro Reino da Silva Birra

October 2020

Resumo

O Ray Tracing tem sido, por muito tempo, a melhor escolha para produzir imagens fotorrealistas; no entanto, a qualidade da imagem tem um alto custo de desempenho. Além disso, métodos de Monte Carlo como estes têm a caracteristica de produzir imagens com alta variância se não forem renderizados por longos períodos de tempo. Os métodos de reconstrução são uma solução recente que visa remover o ruído das imagens geradas desta forma. Com o NVidia RTX, o traçar de raio foi aprimorado pelo hardware dedicado para calcular estruturas de aceleração e interseções de entre raios e primitivas geométricas. Isto permite que as simulações de transporte de luz sejam mais rápidas do que nunca. Neste trabalho, pretendemos comparar diferentes tipos de algoritmos de transporte de luz quando usados em conjunto com métodos de reconstrução, de forma que podemos, em última análise, responder se devemos continuar a investir em algoritmos mais complexos, agora que a reconstrução é uma nova ferramenta ao nosso dispor.

Palavras-chave: Tempo Real, Monte Carlo , Path tracing , Bidirectional Path tracing , AI Denoiser;

Abstract

Ray Tracing has, for long, been the best choice to produce photo-realistic images, however, the image quality comes at a high cost in performance. Furthermore, Monte Carlo methods such as these are known to produce high variance images if not rendered for long periods of time. Reconstruction methods are a recent solution that seeks to remove high variance from images generated this way. With NVidia RTX, ray tracing was enhanced by providing dedicated hardware to compute Acceleration Structures and ray-primitive intersections. This enables light transport simulations to be faster than ever before. In this work, we intend to compare different types of light transport algorithms when used in conjunction with reconstruction methods, so that we can ultimately answer if we should keep invest in more complex algorithms, now that reconstruction is available.

Keywords: Real-Time Rendering , Monte Carlo , Path tracing , Bidirectional Path tracing , AI Denoiser;

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Real-Time Ray tracing has for long been considered the technology of the future, a far to reach dream where rendering photo-realistic images by simulating the physical behavior of light could be calculated fast enough so that we can interact with the scene and perceive the changes in real-time. This future is now possible with the introduction of hardware-accelerated ray tracing simulation provided by the NVidia RTX family of graphics cards.

Typical production renderers render an image by using a vast number of samples per pixel to render the best-looking image possible, however recent work in image reconstruction, also referred to as denoising, has opened the possibility of generating images with far fewer samples per pixel, and therefore less time, with substantial quality.

Ray Tracing is a technique used to simulate the physical interactions of the light in a scene, and over the years, multiple algorithms have been developed to solve this problem, with varying complexity, image quality results, and performance. Now that it is possible to make these simulations at never before seen rates, it is relevant to compare these different ray tracing algorithms that emerged over the years, when implemented on graphics cards, and used in conjunction with reconstruction methods.

1.2 Objectives

The main objective of our research is to create a renderer capable of implementing multiple Light Transport algorithms, with the ability to systematically compare them to each other. Furthermore, with the rise of denoising algorithms, we seek to answer how these behave with more complex algorithms such as Bidirectional Path Tracing and Vertex Connection and Merging, verifying how a simple algorithm with a denoiser compares to the state of the art methods, and ultimately see if complex algorithms have become unnecessary. In other words, we are trying to make an application that helps find answers to the following questions:

- Can we use Ray Tracing algorithms in real-time?

- How well is a denoiser capable of reconstructing images with low samples?
- Is it worth using complex light transport algorithms over simpler ones with denoising?

1.3 Contributions

To answer the questions above, we developed a renderer that leverages the graphics card RTX capabilities; we achieve this by utilizing the Ray Tracing extension for the Vulkan low-level graphics API. Our implementation has an architecture that allows changing between different light transport, multiple scenes, and the ability to denoise the generated image either every frame or after reaching a defined number of accumulated samples or total elapsed time. This denoising is computed and accelerated by the graphics card CUDA cores exposed via the Optix Framework. Optix provides a function that can take as input nosy images generated by light transport algorithms with a small number of samples and output images with much higher quality, resembling images generated with many more samples. This work has a focus on making an application that facilitates the creation of different ray tracing algorithms so that it is easy to add new algorithm implementations into our working pipeline, and compare it with the previously created algorithms. We implemented two light transport algorithms for this thesis's scope: Path tracing and Bidirectional Path Tracing.

We gathered data from multiple scenes rendered with both algorithms, with and without denoising the output so that we can compare the performance and image quality of both algorithms to be able to answer the proposed questions.

Chapter 2

Related Work

Rendering is the process of producing an image from the description of a 3D scene. Physically-based rendering techniques focus on simulating reality by using principles of physics to model the interaction between light and matter [1]. In section 2.2, we will be discussing some of these algorithms, comparing their strengths and weaknesses. The most common solutions produce images with a lot of noise before converging to a correct representation of the described scene; in section 2.3 we will look at how reconstructing low sampling images compare to images with a high number of samples.

2.1 Real-time Rendering

Real-time rendering is concerned with rapidly making images on the computer. In a real-time application, an image appears on the screen, the viewer interact with application, and this feedback affects what is generated next [2]. Rasterisation is still the most common and efficient way of rendering in real-time, however of the available physically based rendering algorithms, stochastic ray-tracing based methods (path tracing, and derived methods) are favored due to their elegance and scalability to achieve photo-realism with minimal algorithmic complexity [3]. These methods work by solving the rendering equation [4] and are usually referred to as light transport simulations. The rendering equation requires an infinite number of light rays to calculate a single pixel's color in an image. That is impossible to compute, so stochastic based methods solve the rendering equation by calculating an increasing number of rays and applying a mathematical function to accumulate each ray's contribution. With this type of technique, the more rays we send and therefore the more time we wait, the better the resulting image is. In real-time applications, we have a finite time to render each image; so while using stochastic methods, we are limited to a few rays per pixel [5] which will result in a nosy image e.g. Figure 2.1.

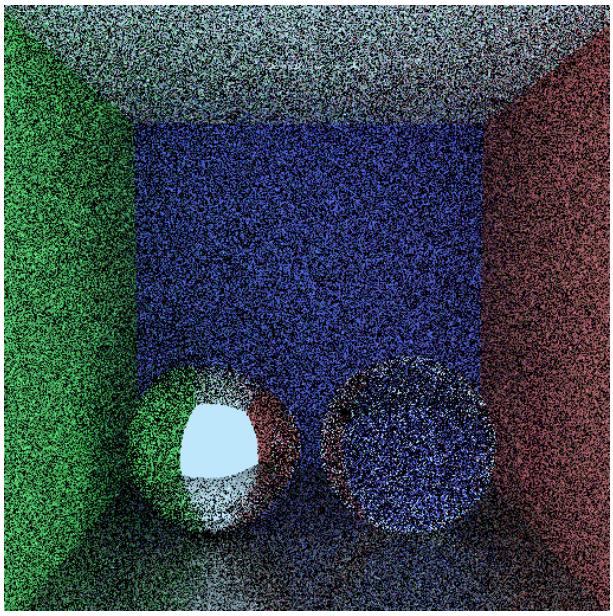


Figure 2.1: Noisy/high variance image produced by Monte Carlo methods

2.2 Ray Tracing

Ray tracing is a general term that covers a vast set of algorithms that aim to achieve an approximate solution to the rendering equation [4] via Monte Carlo sampling. These algorithms all rely on tracing rays, thus the name, intersecting them with different scene objects and gathering information of the surfaces hit.

2.2.1 Rendering Equation

The rendering equation introduced by James Kajiya [4], describes, in mathematical form, the way that light interacts with a point in a surface (see Fig. 2.1).

$$L_o(x, w_o) = L_e(x, w_o) + \int_{\Omega} f_r(x, w_i, w_o) L_i(x, w_i)(w_i \cdot n) dw_i \tag{2.1}$$

Notation	Meaning
x	point in a surface
w_o	outgoing direction
w_i	incoming direction
n	surface normal of point x
Ω	unit hemisphere centered on point x
$L_o(x, w_o)$	radiance of point x in the outgoing direction w_o
$L_e(x, w_o)$	radiance emitted by the point x in the outgoing direction w_o
$f_r(x, w_i, w_o)$	proportion of irradiance reflected of refracted from w_i towards w_o
$L_i(x, w_i)$	radiance at point x incoming from w_i

Table 2.1: Rendering Equation mathematical notation

The outcome of solving this equation is the color of the point x when perceived from a direction w_o, therefore photo-realist images can be generated by computing this equation for every pixel of said image.

According to this equation the color of a point perceived from a certain direction $L_\omega(x, w_\omega)$ is equal to the radiance/light emitted by that point, expressed as $L_e(x, w_\omega)$ plus the every radiance incoming from every direction on an hemisphere Ω around that point, influenced by the material and incident angle, expressed as $\int_\Omega f_r(x, w_i, w_o)L_i(x, w_i)(w_i \cdot n)dw_i$

The incoming light, $L_i(x, w_i)$, is affected be the material it is intersecting with; this intersection is covered in the equation by the function $f_r(x, w_i, w_\omega)$, that depends on the incoming and outgoing directions: w_i, w_ω.

2.2.2 Monte Carlo Sampling

Since the integral in the rendering equation does not have a closed form, it is not analytically solvable, however an approximation to the solution can by given using monte carlo sampling:

$$\int f(x)dx \approx E\left[\frac{1}{k}\sum_{i-1}^{k} f(Xi)\right] \tag{2.2}$$

Monte Carlo sampling states that an integral can be estimated by computing uniformly distributed samples X_i and calculating their average. This method is applied in light transport algorithms to get an approximate solution to the rendering equation[2.1], by shooting rays in directions sampled from a uniformly distributed function and calculating their interaction with the scene. By increasing the number of samples we accumulate and average together the result of the weighted sum approximates to solution of integral. Therefore the more rays we shot into the scene the closer we are to the actual solution to the rendering equation.

2.2.3 Global Illumination

Global illumination is a term used to represent the combination of direct and indirect lighting, characterized by multiple ray bounces between light sources and the scene objects [4]. Algorithms that model global illumination generate at least one plausible path between the light sources and the camera and compute each light's contribution along that path. This class of algorithms is capable of rendering effects such as caustics and color bleeding, as shown in figure 2.2 and 2.3.

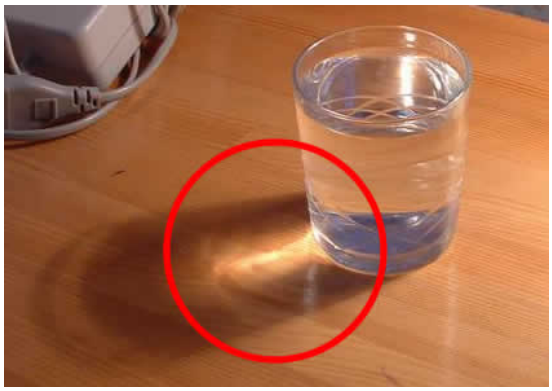


Figure 2.2: Caustics [6]

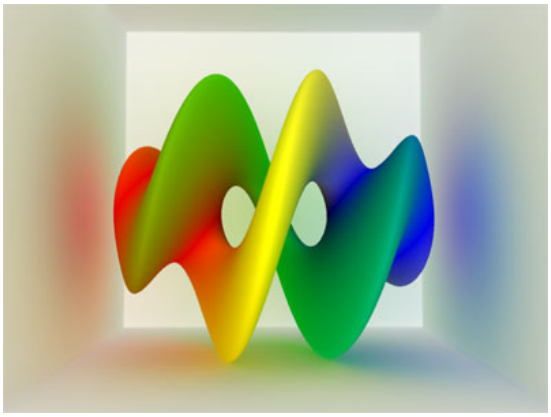


Figure 2.3: Color Bleeding [7]

2.2.4 Path Tracing

Path Tracing [4] is one of the simplest light transport simulation algorithms that are capable of rendering scenes with Global Illumination. As tracing paths is an essential building block for all algorithms introduced in the later sections, good understanding and optimization of Path Tracing has a significant impact on their performance [8].

Path Tracing consists of sampling random paths across the scene. This method starts by casting a ray from the camera, called a primary ray; it is then traced into the scene geometry and on each surface of the geometry hit, we recursively create another ray in a stochastic direction.

Russian Roulette

To prevent infinitely long paths, caused by rays bouncing forever, Russian Roulette path termination is applied to each path vertex, which determines if the given path should be terminated. The path termination usually is based on the properties of the surface associated with the vertex, if the probability of that path contribution is low, that path is terminated and the contribution of remaining paths are augmented by the probability of termination.

Next Event Estimation

In Kajiya's original algorithm, a path only contributes to a pixel color when the path terminates on a light; depending on the scene this can be a difficult task or even impossible if the scene has only point lights. To bypass this constraint on each ray's hit position, we additionally sample a random light, and if this light is not obstructed by any other scene geometry that point is directly illuminated, and so we add its contribution to the path, this technique is called Next Event Estimation (NEE).

Importance Sampling

Monte Carlo Methods may sample the integrated function non-uniformly, however this biases the desired estimation and can lead to incorrect results. To maintain these methods unbiased it is necessary to

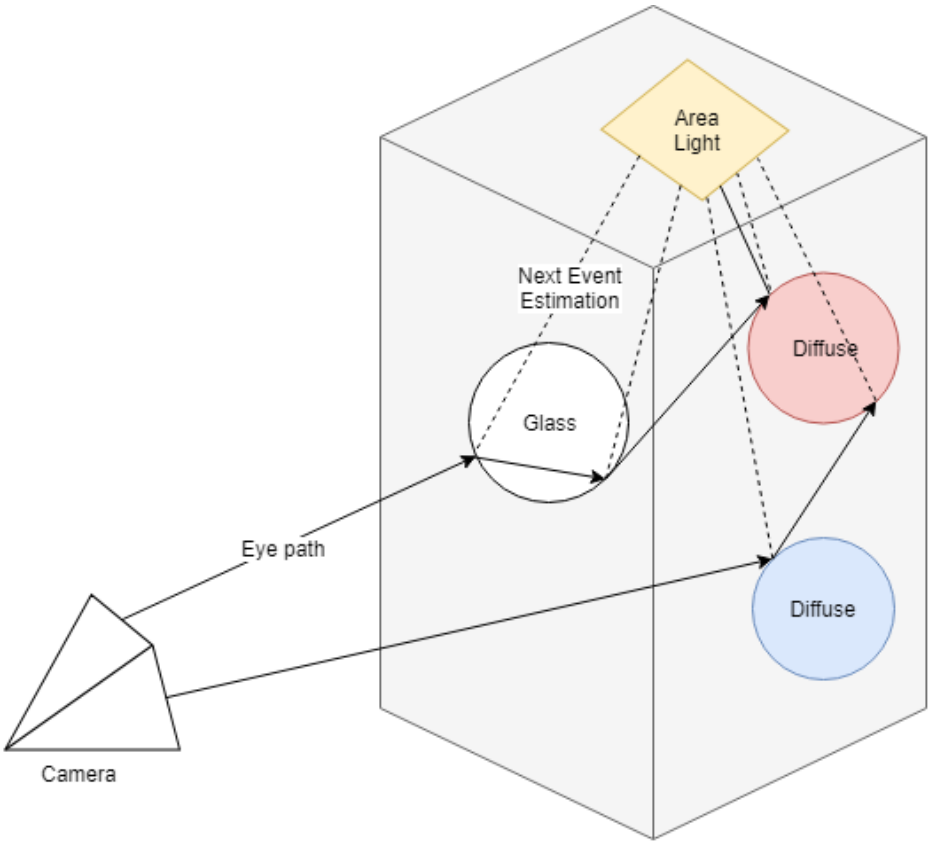


Figure 2.4: Schematic representation of Path Tracing with Next event estimation

weight the these samples by the probability of sampling that value. This weight function pdf is the probability density function of the sampling operation, for the rendering equation that is the cosine weighted function, but we can also sample the material or light sources.

2.2.5 Bidirectional Path Tracing

The Bidirectional Path Tracing algorithm [9] can be viewed as a generalization of the Path Tracing algorithm and it was created as an attempt to outperform it on complex scenes that rely on indirect illumination. Unidirectional Path Tracing relies on tracing a path from the camera lens to emissive objects (e.g, area lights, fire) and calculating the illuminance contribution to the color of a pixel by the light that was intersected by the path. On more complex scenes where there are few light sources, or the lights require multiple and improbable bounces, very few paths contribute to the pixel color, as shown in Figure 2.5.

Bidirectional Path Tracing mitigates this problem by tracing path not only from the camera but also from each light source. Paths originated from the camera are referred by camera or eye sub-paths, and paths originated from light sources are called light sub-paths. The algorithm recursively extends both paths, adding one vertex at a time, just like Path Tracing. After generating both paths, each vertex of the camera subpath is connected to the light sub-path vertices, similar to Unidirectional path tracer with Next Event Estimation but instead of sampling every light source we sample every vertex of each light sub-path. To achieve better performance it is common to reduce the number of connections by stochastically selecting the light vertices that are connected. A path in a bidirectional path tracer can

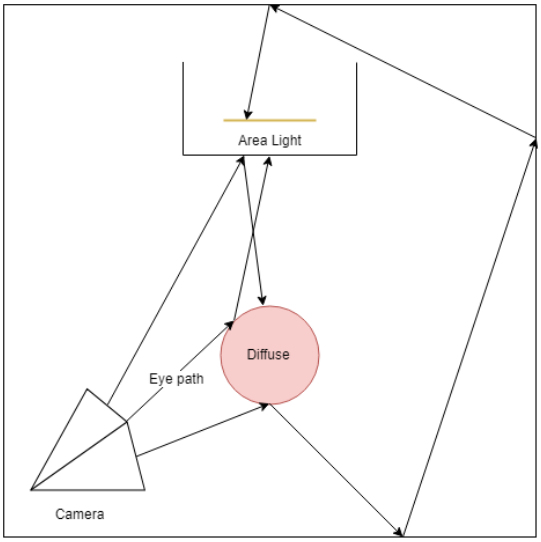


Figure 2.5: Representation of a difficult scene of Path Tracing. Note that Next Event Estimation is not represented.

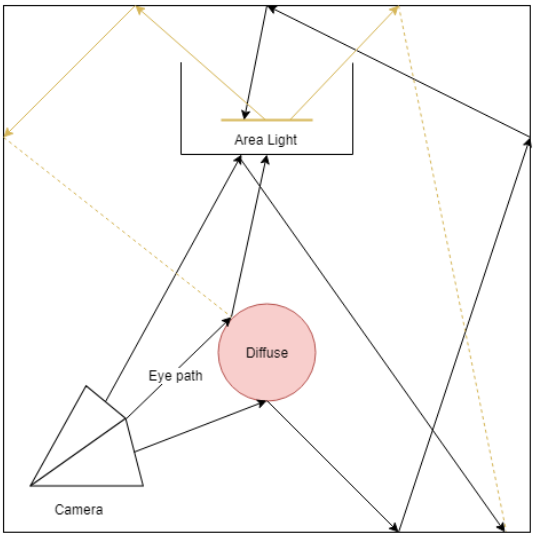


Figure 2.6: By connecting eye sub-paths with light sub-paths, the final render converges faster.

be constructed in multiple ways; it is necessary to have a way of determining how much should a path contributes to a pixel's final color. This problem is addressed by Importance Sampling 2.2.4.

2.2.6 Photon Mapping

While Path Tracing and Bidirectional Path Tracing offer great flexibility to render a wide variety of scenes, some effects, such as those associated with reflected caustics represented in Figure 2.8, remain particularly hard to capture. Photon Mapping techniques require both light and eye sub-paths just like Bidirectional Path Tracing. However light sub-path vertices are called photons and each eye subpath vertex connects all the photons inside a certain radius. The main idea of this approach is to cast rays from the light sources, store photons generated by these paths in a structure that can be used efficiently for queries. After populating the structure with photons we trace eye sub-paths by casting rays starting from the camera, once with hit a surface we calculate the contribution of the path by querying the photon structure for nearby photons. Conceptually this is a simple algorithm to grasp; Figure 2.10 is a visual representation of the general technique.

2.2.7 Vertex Connection and Merging

Bidirectional Path Tracing and Photon Mapping are notorious for being very versatile rendering algorithms, however, both are not without problems, for example, Photon Mapping approaches have difficulties producing noise-free renders of diffuse surfaces illuminated by light sources that are far away, as demonstrated in Figure 2.9. It has been acknowledged that these two families of algorithms are complementary in terms of the types of light transport effects they can efficiently capture [10]. Vertex Connection and Merging is an algorithm that aims to leverage the advantages of both methods by combining vertex connection techniques from Bidirectional Path Tracing and Vertex Merging techniques from

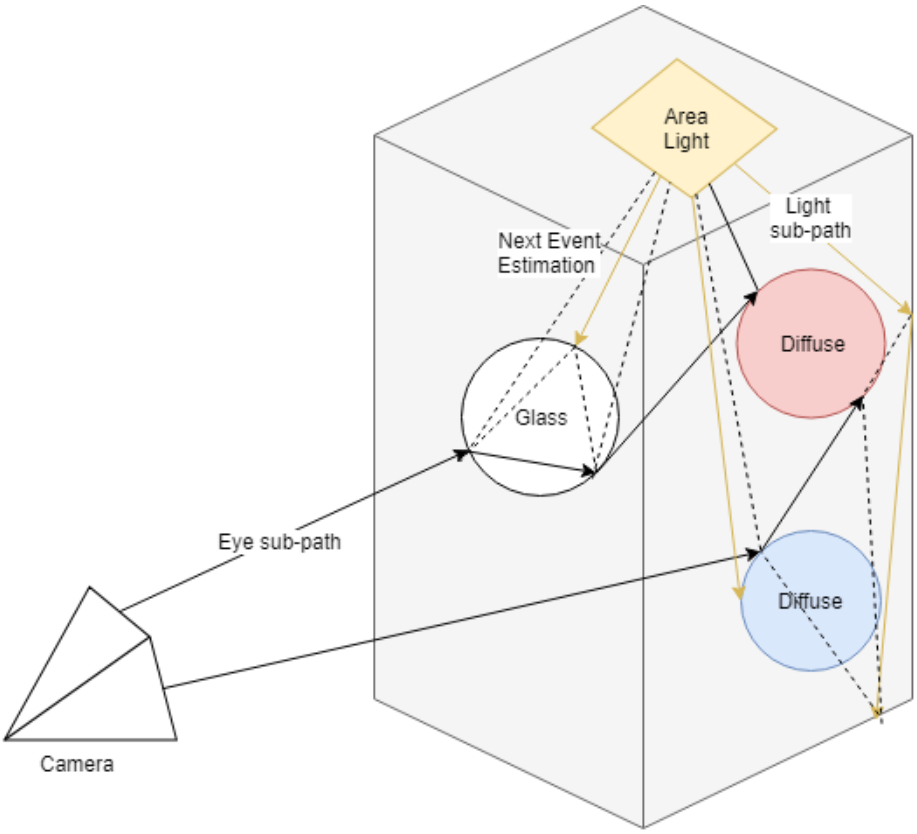


Figure 2.7: Schematic representation of eye and camera sub-paths in a Bidirectional Path Tracing.

Figure 2.8: Reflection of caustics in a Bidirectional Path Tracer (Left) and the expected result (Right) Images rendered using smallvcm renderer [10]

Figure 2.9: Illumination of diffuse (background) and specular (car) with Photon Mapping and Bidirectional Path Tracing Images rendered using smallvcm renderer [10]

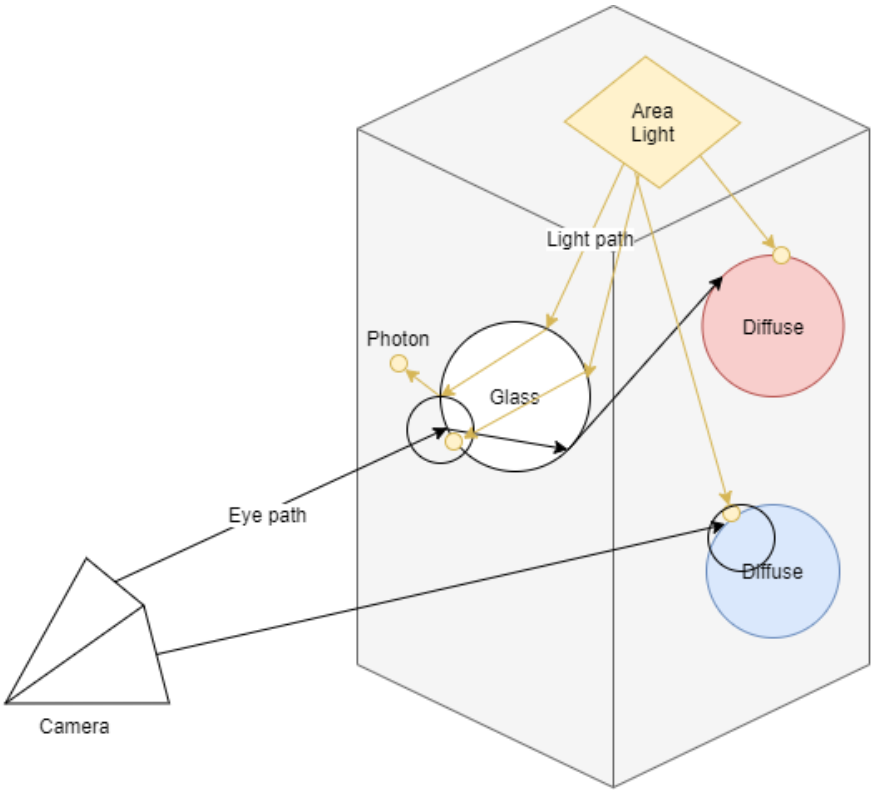


Figure 2.10: Schematic representation of Photon Mapping

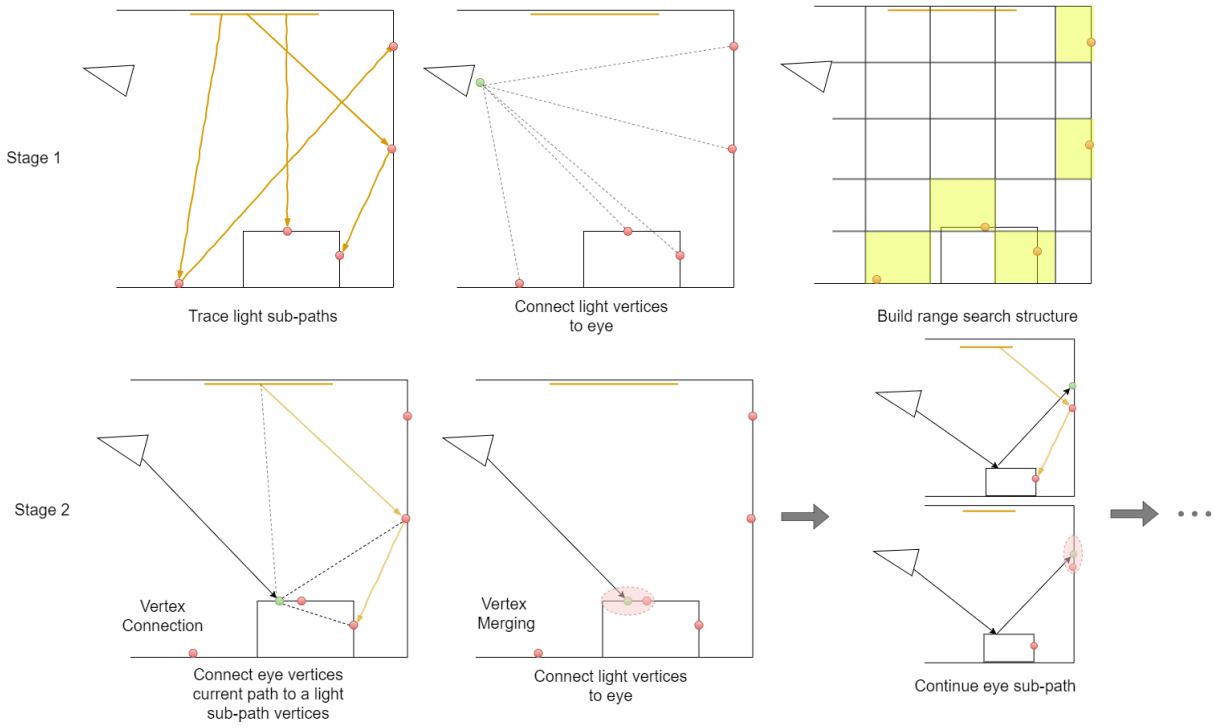


Figure 2.11: Schematic representation of Vertex Connection and Merging

Photon Mapping via the preciously introduced Multiple Importance Sampling [11].

Vertex Connection and Merging is a two-stage algorithm, the stages are visually represented in Figure 2.11; the first stage is focused on building a structure of light vertices similar to photon structure in Photon Mapping. In the original paper, the authors start by tracing multiple light sub-paths and generating a set of light vertices, represented on the Figure 2.11 by red dots, the authors then connect these vertices to the eye, creating a link between the light path and a pixel, and then they populate a light structure with the light sub-path vertices. In the second stage for every pixel, a sub-path is traced where every eye vertex is connected to a light source and afterward to the light sub-path associated with the corresponding pixel. Finally, the authors merge the vertex from the current eye sub-path to every vertex that is within a certain radius. The estimated contribution of the full path is then multiplied by a weight calculated using Recursive Multiple Importance Sampling, therefore effectively combining Bidirectional Path Tracing and Photon Mapping.

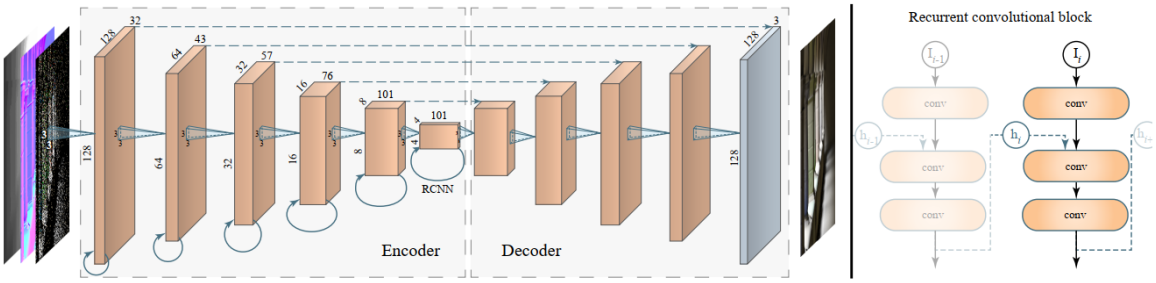


Figure 2.12: Architecture of the recurrent auto-encoder proposed in [15]

2.3 Reconstruction

In recent years, the algorithm we've introduced before have emerged as the rendering techniques of choice for film and visual effects [12]. All of these techniques are based on Monte Carlo sampling which inevitably leads to noisy outputs, when the sample count is low, which spurred the development of reconstruction kernels [13]. These kernels allow noise-free reconstruction of images with dozens to hundreds of samples per pixels [14]. Since with Monte Carlo sampling every time we render a frame, the output will be slightly different. When generating image sequences this problem is aggravated, because the difference between frames produces a flickering effect, which is not desired.

2.3.1 Recurrent Denoising Autoencoder

In this section, we present a basic overview of a reconstruction method, note that details of the implementation are not mentioned. The work of Chakravarty Chaitanya [15] describes a machine learning technique for reconstructing an image sequence rendered using Monte Carlo methods. Using a variation of a deep convolutional network the authors can take as input an extremely noisy image, with a low count of Samples Per Pixel (SPP), and produce as output an image that resembles the same image with much more SPP.

This algorithm uses a Recurrent Neural Network represented in Figure 2.12, and has a large training set consisting of input sequences and the desired output sequences. This way the neural network learns to map noisy inputs to noiseless outputs.

Chapter 3

Implementation

3.1 Overview

To test the different algorithms, we needed to create a rendering engine so that we could implement different algorithms to solve the rendering equation. This section will describe the evolution of said engine and introduce a high-level description of its architecture. We started by creating an engine using the Optix [16], which is a programmable general-purpose ray tracing framework implemented on the GPU. Optix is an algorithm agnostic renderer, meaning that it does not define its ray tracing or path tracing algorithm; the user is responsible for programming it. In Optix, we define a set of Cuda programs and create a context with them. These programs are then used by the Optix rendering pipeline to produce an image. Figure 3.1 provides the scheme of the pipeline; the yellow boxes are the programmable components of the pipeline.

Our Solution consists of implementing different Ray Generation Programs, in which we define the light transport algorithm. These programs are compiled during the build phase and can be swapped between each other at runtime. This feature promotes the comparison between them without the need to recompile the program. Optix offers an API with a few post-processing functions, including a Denoiser [15], as described in the paper.

After implementing a simple Path Tracer with the Optix framework, we compared it with a similar implementation using the Ray Tracing extension for the Vulkan Graphics API. Our Optix Path Tracer was around 24 times slower than its Vulkan counterpart, running at under 10 fps while the Vulkan one ran at 170 fps with the same scene, path length, and samples per pixel. We looked into optimizing our Optix implementation by following the example implementation provided by NVidia. However, these examples had similar performance as our implementation. After profiling the application, see figure 3.3, we realized that the examples utilized less than 50% of the RTX graphics card capabilities we tested. Facing these unsatisfactory results, we decided to re-implement our engine, now utilizing Vulkan for our ray tracing needs, in conjunction with the Optix denoiser. Our implementation is hosted in [17].

The Vulkan Ray Tracing Pipeline, described in more detail in section 3.2, at a conceptual level, is very similar to Optix(see Fig 3.2). However, Vulkan being a low-level graphics API provides much more

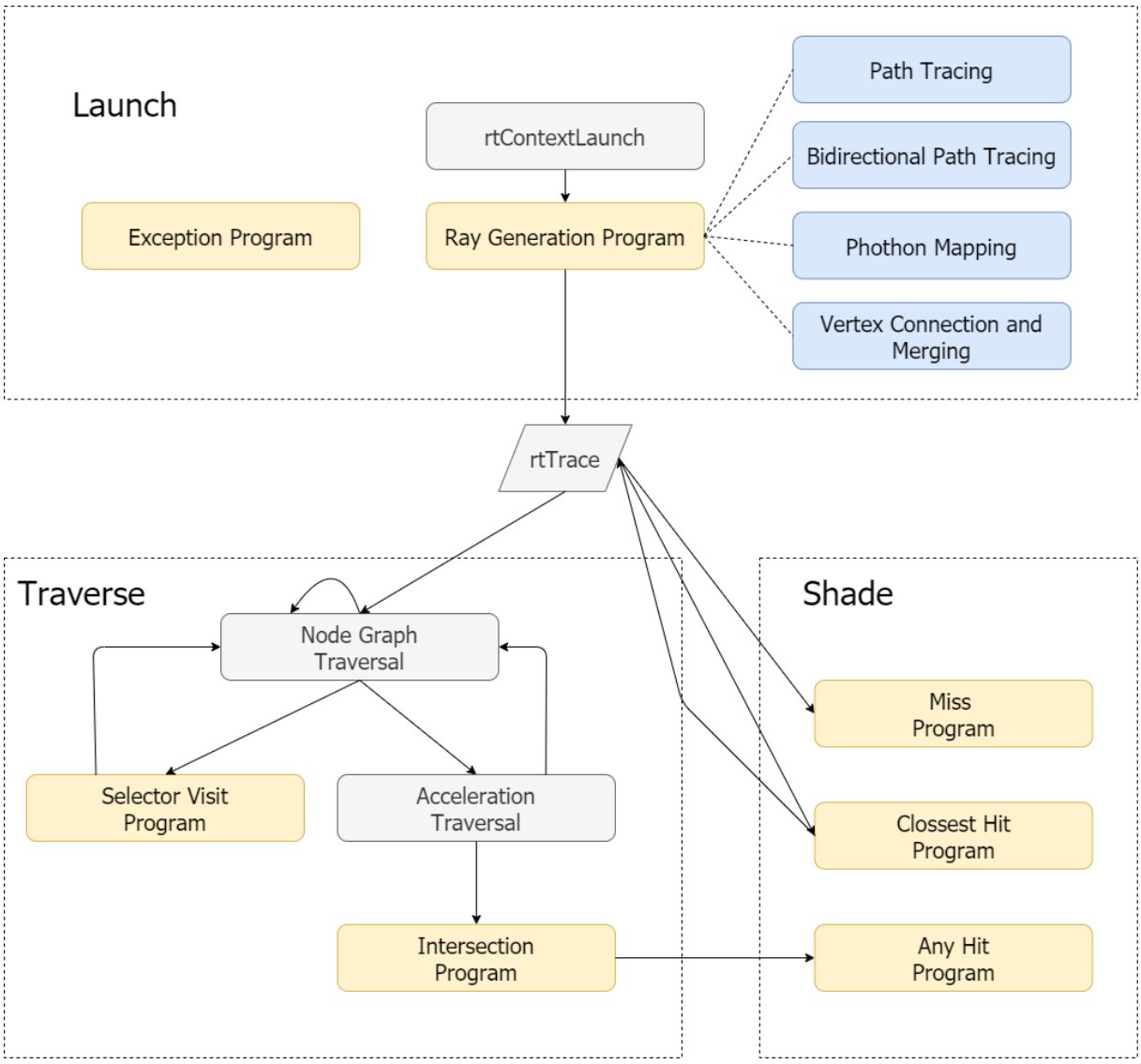


Figure 3.1: Pipeline defined in the Optix framework

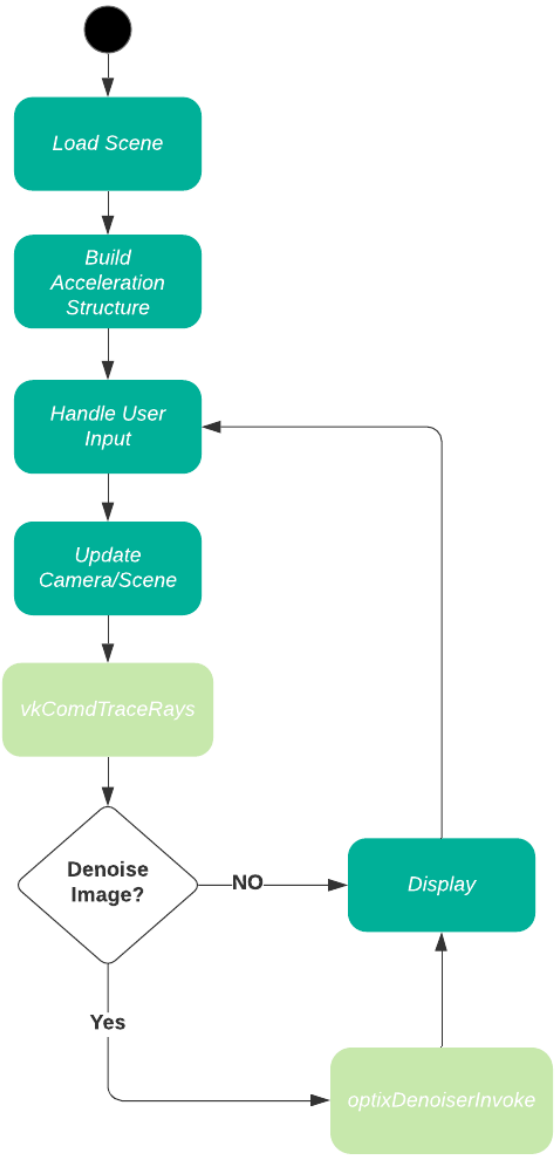


Figure 3.2: Pipeline defined in the Optix framework

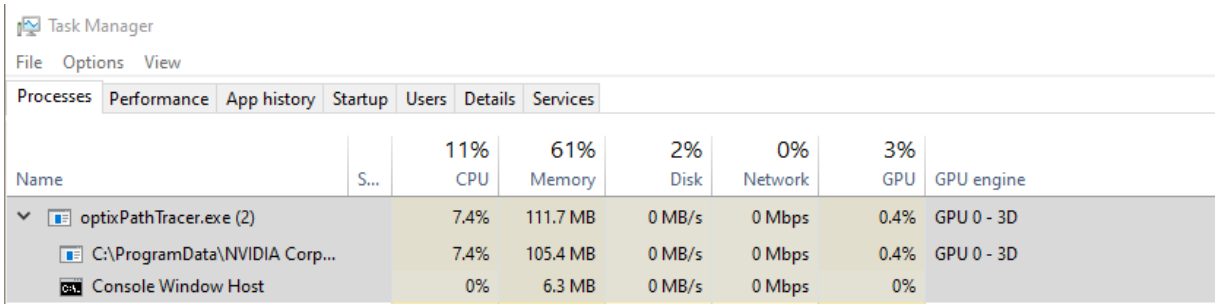


Figure 3.3: Optix application resource usage, note that the GPU usage is very low. Profiling done using Windows Task Manager

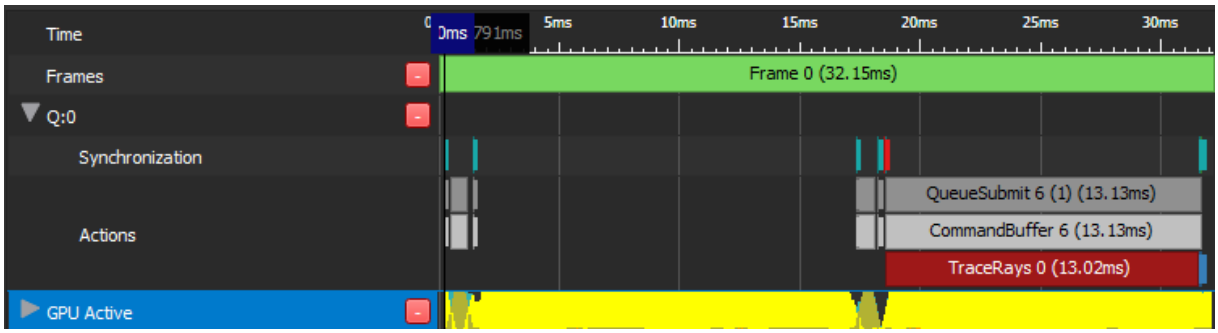


Figure 3.4: Profile frame of our application, note that the GPU was almost always at full capacity. Profiling done using NSight Graphics

fine-grained control of the device resources. These efforts paid off as our engine now was able to utilize nearly 100% of the graphics card capabilities (see Fig 3.4)

3.2 Vulkan Ray tracing Pipeline

To render images using Vulkan's Ray Tracing Pipeline [18], we first need to setup up the ray tracing pipeline, managing how the Acceleration Structures are laid out in Device Memory. Then we build these Acceleration Structures with the geometry of the scene. It is also necessary to configure how updates to these structures occur when the scenes change over time, but since our scenes are static, we do not need to handle complex AS updates.

The Ray Tracing depends on implementing a set of ray generation, intersection, hit, and miss shaders. The relationship is between these shaders is represented in figure 3.5. In the next few sections of this chapter, we explore our implementations of said shaders.

3.2.1 Ray Generation

Ray generation shaders are the starting point to all ray tracing work, they run for each pixel on a multi-threaded 2D grid. They are generally responsible for tracing rays into the scene and writing the algorithm's final output to memory, e.g., the screen or a texture.

Our Solution implements two different ray generation shaders: a Path Tracer and a Bidirectional Path Tracer.

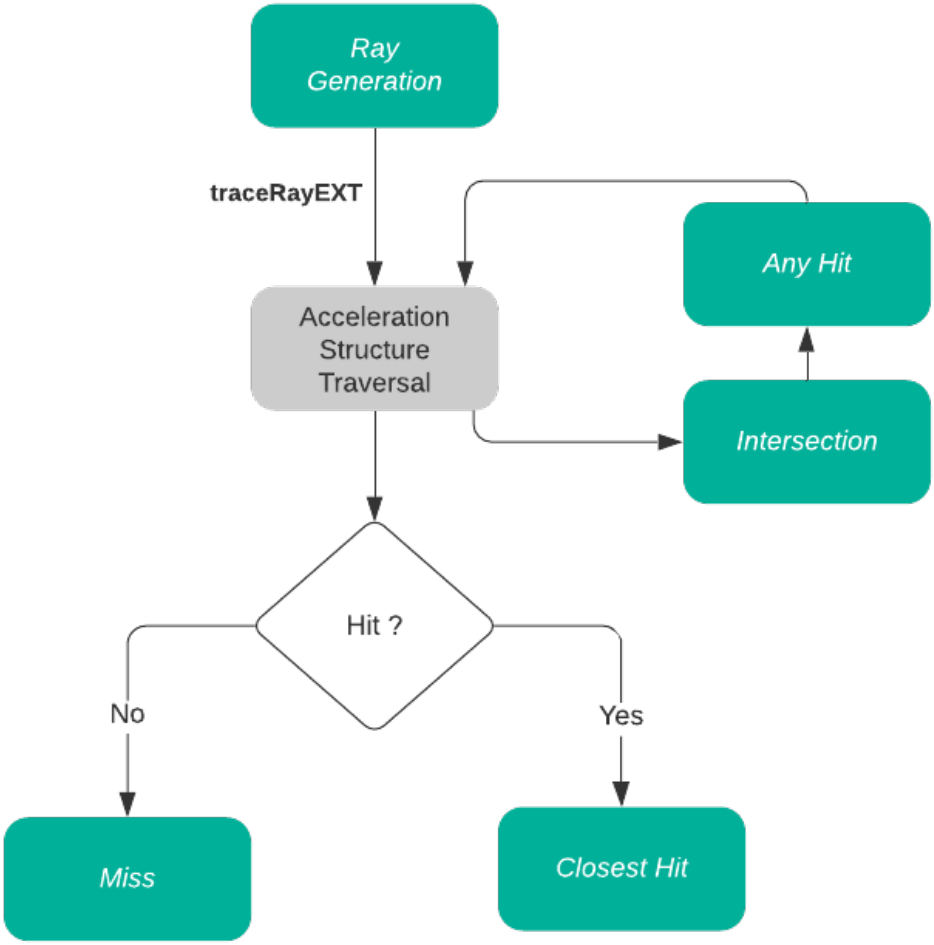


Figure 3.5: Vulkan's Ray Tracing Pipeline

Path Tracing

With Path Tracing the color of a pixel is given by the averaged sum of all the paths that starts from that pixel and eventually hit an emissive object. In 3.1, we can see the pseudo-code for this algorithm, a pixel start with a color of zero (black), then we start a path with a direction defined by the camera. Our path tracing generation shader calculates the color of a given pixel, by calculating the contribution of each path of that pixel, then it is responsible for adding that color to the contribution of previous frames, followed by applying post-processing effects, like exposure, tone mapping, and gamma correction; afterward, we can finally save that pixel color to a texture.

```
void main() {
    vec3 pixel_color = vec3(0);

    for (int sample = 0; sample < number_of_samples; sample++) {
        origin = pixel + random_offset(); // anti-aliasing

        pixel_color += traceEyePath(origin, direction);
    }

    pixel_color /= number_of_samples;

    accumulateWithPreviousFrames(pixel_color);
    applyPostProcessing(pixel_color);
    storePixel(pixel_color);
}
```

Listing 3.1: path.rgen

The responsibility of calculating the contribution of a path comes from ***traceEyePath***, which starts a path from the given origin to the given direction. On each ray intersection, we check if the material we intersected is emissive; if it is, we terminate that path and return its color; otherwise, we sample the Bidirectional Scattering Distribution Function of that material, that allows us to calculate the direction of the path's next bounce. We then trace a ray from that intersection towards a selected point in a light or emissive object. If there is no intersection, it means that point is directly illuminated, and therefore a contribution is added to that path. A given intersection's contribution is calculated based on the material we hit divided by the probability of the ray bouncing towards that chosen light. Finally, we decide if we want to continue extending the current path with the Russian Roulette Termination method, were we select termination probability *p* based on the path contribution RGB values and then randomly decide if that path is terminated, amplifying the contributions that survive this termination by $1/p$.

```
vec3 traceEyePath(vec3 origin, vec3 direction) {
    vec3 color = vec3(0);

    for (int bounce = 0; bounce < path_lenght; bounce++) {
        // return with intersection information (Hit)
        traceRay(scence, origin, direction);

```

```
            if (Hit is Emissive) {
                return color * emissive;
            }

            // Get intersected material, the next ray origin and the scattered direction
            sampleBSDF();

            if (Ray missed) {
                return backgroung_color;
            }


            // Next Event Estimation
            bool in_shadow = traceShadowRay();
            if (not in_shadow) {
                color += Material Color / Ray Probability;
            }

            // Terminate paths with low contribution
            russian_roulette_termination();
        }

        return color;
    }
```

Listing 3.2: Path Tracing traceEyePath

Bidirectional Path Tracing

Bidirectional Path Tracing distinguishes itself from a regular Path Tracer by generating a path from the light sources as well as from the camera, followed by the connection of these paths. Our Bidirectional Path Tracing ray generation shader 3.3 is very similar to our path tracing one. The critical difference is that we start by construction light paths that originate from light sources and bounce through the scene, saving information about the position, the intersected materials, and the probabilities these bounces, given by the BSDF of said materials. Then when tracing paths from the camera on the ***traceEyePath*** function, represented by Listing 3.4, we connect these each vertex to the light path vertices. For that, on each intersection, we cast a shadow ray towards each light path vertex, and if no other object is occluding the two vertices, we calculate the contribution of this path and divide it by its probability.

```
    void main() {
        vec3 pixel_color = vec3(0);

        for (int sample = 0; sample < number_of_samples; sample++) {
            origin = pixel + random_offset(); // anti-aliasing

            constructLightPath();

            pixel_color += traceEyePath(origin, direction);
```

```
        }

        pixel_color /= number_of_samples;

        accumulateWithPreviousFrames(pixel_color);
        applyPostProcessing(pixel_color);
        storePixel(pixel_color);
    }
```

Listing 3.3: bdpt.rgen

```
vec3 traceEyePath(vec3 origin, vec3 direction) {
        vec3 color = vec3(0);

        for (int bounce = 0; bounce < path_lenght; bounce++) {
            // return with intersection information (Hit)
            traceRay(scence, origin, direction);

            if (Hit is Emissive) {
                return color * emissive;
            }

            // Get intersected material, the next ray origin and the scattered direction
            sampleBSDF();

            if (Ray missed) {
                return backgroung_color;
            }

            // Sample the light vertices calculated on the constructLightPath() step
            for (int l = 0; l < LIGHTPATHLENGTH; l++) {
                bool in_shadow = traceShadowRay(light_paths[l]);
                if (not in_shadow) {
                    // The color is scaled by the ray probability and a weight
                    // calculated using Multiple Importance Sampling
                    color += Material Color / Ray Probability / MIS weight;
                }
            }

            // Terminate paths with low contribution
            russian_roulette_termination();
        }

        return color;
    }
```

Listing 3.4: Bidirectional Path Tracing traceEyePath

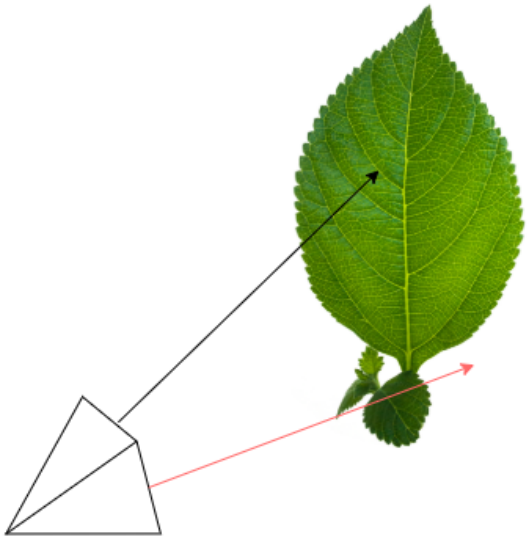


Figure 3.6: Any Hit shader - The red ray intersection is discarded

Intersection Shaders

Intersection shaders are used to implement ray-primitive intersection algorithms, other than ray-triangle intersections, that have built-in support. They can be used to allow the scenes to have custom primitives like hair or spheres. Our implementation provides a sphere intersection shader.

Hit Shaders

There are two types of Hit shaders in Vulkan, the Any Hit and Closest Hit. A Hit shader is run every time an intersection between a ray and some geometry primitive is found, in arbitrary order. The Closest Hit shader is only executed for the closest primitive to the ray origin. One example use for Hit shader would be to discard ray-primitive intersections when we hit a geometry that samples from a texture, and the point we hit is transparent, as shown in figure 3.6. However, our implementation does not use any Hit shaders as they are optional in the pipeline, and our scenes did not require them.

Our Closest Hit shaders are only responsible for creating a Ray Payload with the geometry information and material information of the intersected primitive, which will be used by the Ray Generation shaders.

Miss Shaders

Miss shaders run when a ray fails to intersect with any geometry primitive in the scene. These shaders are typically used to return the scene's background color or sample an environment map.

3.3 Optix Denoiser

Since we decided to produce our Monte Carlo algorithms using Vulkan, that means the images are stored in Vulkan buffers, and Optix uses Cuda buffers to do its calculations, so to enable this interop between Vulkan and Optix, we need to translate the Vulkan buffers to Cuda buffers and vice-versa. Our solution revolves around getting a Vulkan Image generated by the Monte Carlo algorithm; copying said image to a Cuda buffer with the appropriate layout so that we can invoke the Optix denoiser with it, which generates a new Cuda buffer with the denoised output; finally, we need to copy this denoised Cuda buffer back to a Vulkan image with the correct layout so that it can be displayed on the screen.

Chapter 4

Testing and Results

4.1 Evaluation Methodology

To evaluate the different algorithms we use a set of scenes, with different characteristics, e.g. reflection of caustics, glossy materials, or difficult to reach light sources, to benchmark each algorithm's strengths weaknesses, we implemented on a graphics card.

In order to evaluate the success of each algorithm, we will measure renders with varying amounts of accumulated samples with the following metrics.

4.1.1 Frames Per Second

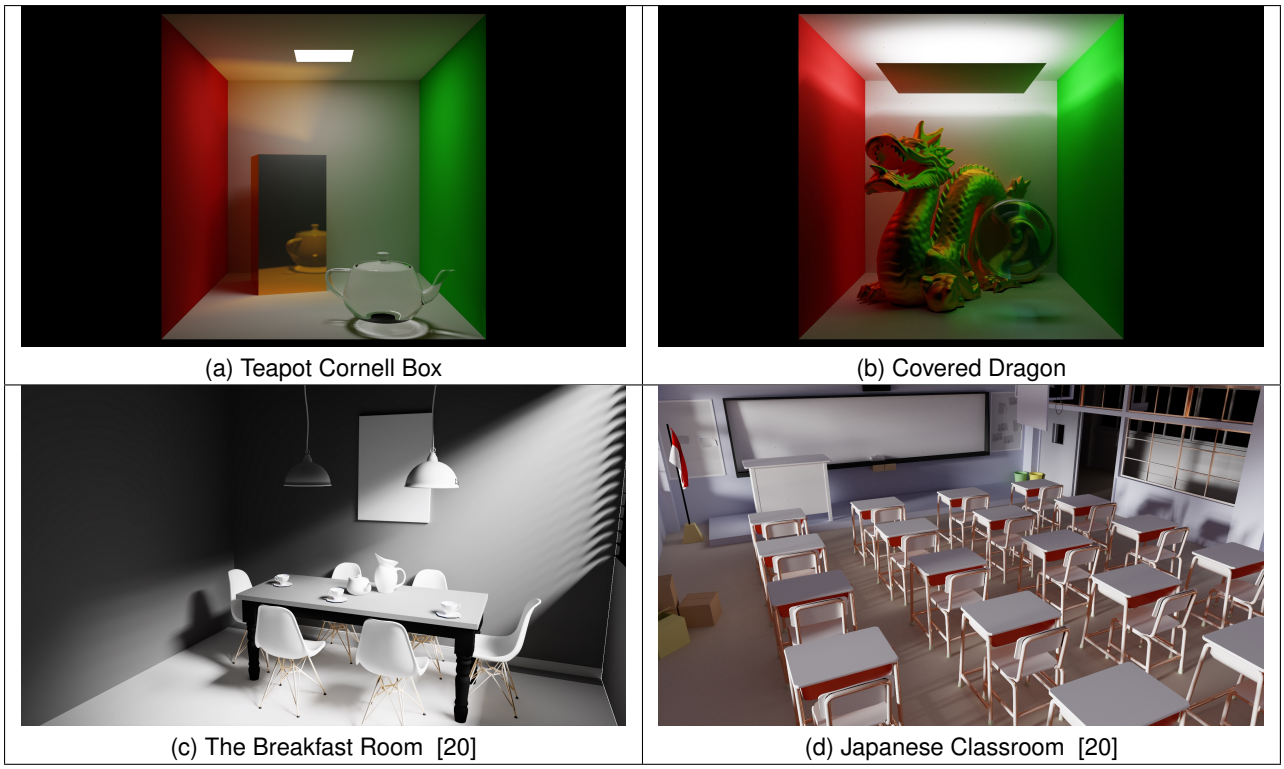
This metric measures the number of frames a given algorithm can produce per second. We will record the average and the standard deviation of the frames per second. With this metric we gain information about the time need to render a single frame and if the algorithm can maintain a stable throughput.

4.1.2 Structural Dissimilarity

Structure Dissimilarity Index Metric (DSSIM) is an image quality metric that gives a positive value where 0 means its an identical image and $DSSIM > 0$ represents the amount of difference between the images. Structural Similarity Index Metric is one other popular metric used when comparing Images and DSSIM can be derived from SSIM [19] where $DSSIM = 1/SSIM - 1$ This metric as seen much use to compare the output of a Monte Carlo renders to its reference image.

4.1.3 Time to Converge

In this scenario we test the algorithms in a static scene and with a stationary camera, this way we can measure how long, and additionally how many iterations are necessary for a certain algorithm to achieve a defined DSSIM between the output and a reference image.



(a) Teapot Cornell Box

(b) Covered Dragon

(c) The Breakfast Room [20]

(d) Japanese Classroom [20]

Figure 4.1: Scenes selected for testing.

4.2 Test Scenes

For testing purposes, we chose scenes that either have been used to benchmark Ray Tracing application or are able to showcase most features of our application. These scenes are shown in Figure 4.1.

We chose this set of scenes to provide a wide range of characteristics and different complexities. The ”Teapot Cornell Box” is a simple scene with very few primitives, featuring as the name suggests, the Cornell Box with a teapot, showcasing different materials, such as diffuse, glass, and colored metal. The ”The Covered Dragon” utilizes the same Cornell Box, though we added a rectangle covering the light source, this scene serves to test how the implemented light transport algorithms behave when the light sources are hard to reach. This scene also includes the Standford Dragon, composed of approximately 5,500,000 triangles, and a sphere to showcase that renderer supports custom primitives, besides triangles. ”The Breakfast Room” [20] features multiple different geometries, the light source in this scene is outside the room and enter it through a window with blinders. Finally, we chose the ”Japanese Classroom” [20], a commonly used scene to benchmark ray tracing applications and denoisers.

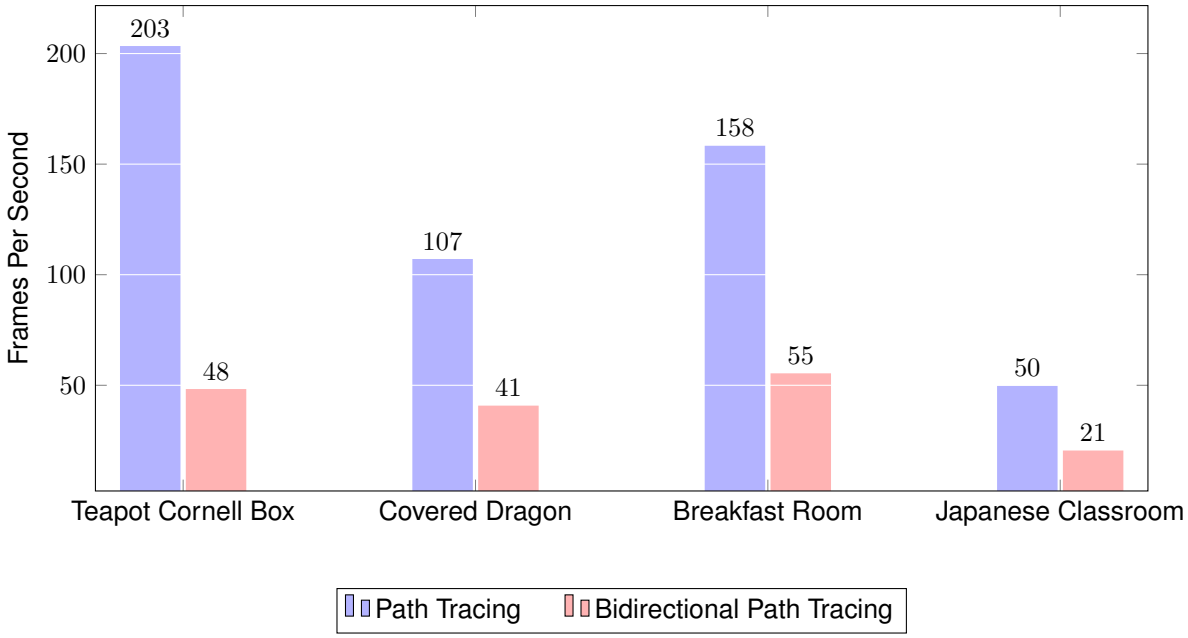


Figure 4.2: Average Frame Rate on 1920x1080 Resolution

4.3 Results

This section will present and analyze the results we gathered from rendering our test scenes with both Path Tracing and Bidirectional Path Tracing. We will also see how the recursion step affects performance as well as image quality. Every test was made on the same machine utilizing the NVidia RTX 2070 graphics card, every test we made was GPU bound, so the only component that matters is the graphics card. Every frame, we trace a single path per pixel, meaning that we calculate one sample per pixel per frame. Both algorithms had a maximum path length of 32, which represents the maximum number of bounces a path can make through a scene. Appendix A contains more images rendered by our application, and Appendix B has Tables and Figures with more test results.

4.3.1 Performance

We start by analyzing the performance of both ray tracing algorithms; Figure 4.2 shows the average frames per second that the scene was being rendered at (see Table 4.1 for more details). Here we can see that that at the resolution of 1920x1080 Path tracing can achieve real-time rates in every scene and that Bidirectional Path Tracing is more expensive performance wise but can still achieve iterable frame rates on the scenes we tested.

The table 4.2 reflects the duration of denoising a single frame. The duration of the denoiser is independent of the scene and ray tracing algorithm, the only factor that affects its duration is the image resolution. For example, a denoised frame of the scene *Breakfast Room*, at the resolution of *1920x1080*, using Path Tracing takes $6.3 + 14.5 = 20.9$ milliseconds, which means that this scene runs at $1/0.0209 = 47.8$ frames per second with the denoiser running on every frame.

	PT		BDPT	
	Frame Duration [ms]	Frames Per Second	Frame Duration [ms]	Frames Per Second
Teapot Cornell Box 1280x720	2.3	416.2	9.6	103.9
Covered Dragon 1280x720	4.6	219.7	11.3	87.0
Breakfast Room 1280x720	2.9	332.7	8.5	118.3
Japanese Classroom 1280x720	9.3	106.8	22.2	45.4
Teapot Cornell Box 1920x1080	4.8	208.3	16.7	48.3
Covered Dragon 1920x1080	9.5	104.7	24.4	40.8
Breakfast Room 1920x1080	6.3	158.7	18.6	55.4
Japanese Classroom 1920x1080	20.2	49.4	52.4	20.5

Table 4.1: Scene Frame Rates

Image Resolution	640x360	854x480	1280x720	1920x1080
Denoiser Duration [ms]	2.8	4.4	7.5	14.5

Table 4.2: Denoiser Durations

4.3.2 Image Quality Assessment

In this section, we take a look at the images the algorithms can produce and how they improve in quality over time, and how the application of the reconstruction step affects the image quality. The Figure 4.3 is the result of accumulating eight samples, meaning that every pixel only has a contribution from a maximum of eight paths. We can see that with this low sample count, the image contains a high amount of variance. For comparison Figure 4.4 shows the result from rendering the same scene for an equal amount of time with the Bidirectional Path Tracer, where we are only able to accumulate 4 samples. Figure 4.5 is the outcome of denoising Figure 4.3. And finally figure 4.6, shows a side by side comparison of these images, where we can see how the denoiser helps reducing variance, from a noisy image.

The charts depicted in figures 4.7 and 4.8 exhibit how the rendered image quality progressively increases by accumulating samples. Figure 4.8 is a zoom in of figure 4.7 regarding a time interval till 3 seconds These results come from computing the DSSIM value between a generated image and the reference for that scene, utilizing a command line tool[21]. Here we can see that both algorithms converge to the reference image, reflected by the DSSIM decreasing over time until getting close to 0. In this scene, we can see that Bidirectional Path Tracer produces higher quality images than a Path Tracer. However, a Path Tracer with Denoising converges much faster to the target image, when comparing with methods without the denoising step.

Figure 4.3: ”Japanese Classroom” - Path Tracing - 8 Total Accumulated Samples - Rendered in 16 milliseconds

Figure 4.4: ”Japanese Classroom” - Bidirectional Path Tracing - 4 Total Accumulated Samples - Rendered in 16 milliseconds

Figure 4.5: ”Japanese Classroom” - Path Tracing with Denoising - 8 Total Accumulated Samples - Rendered in 16 milliseconds + 14.5 milliseconds of denoising

Figure 4.6: ”Japanese Classroom” - Side to Side comparison rendered in 16 milliseconds. Path Tracing (Top-Left), Bidirectional Path Tracing (Top-Right), Path Tracing Denoised (Bottom)

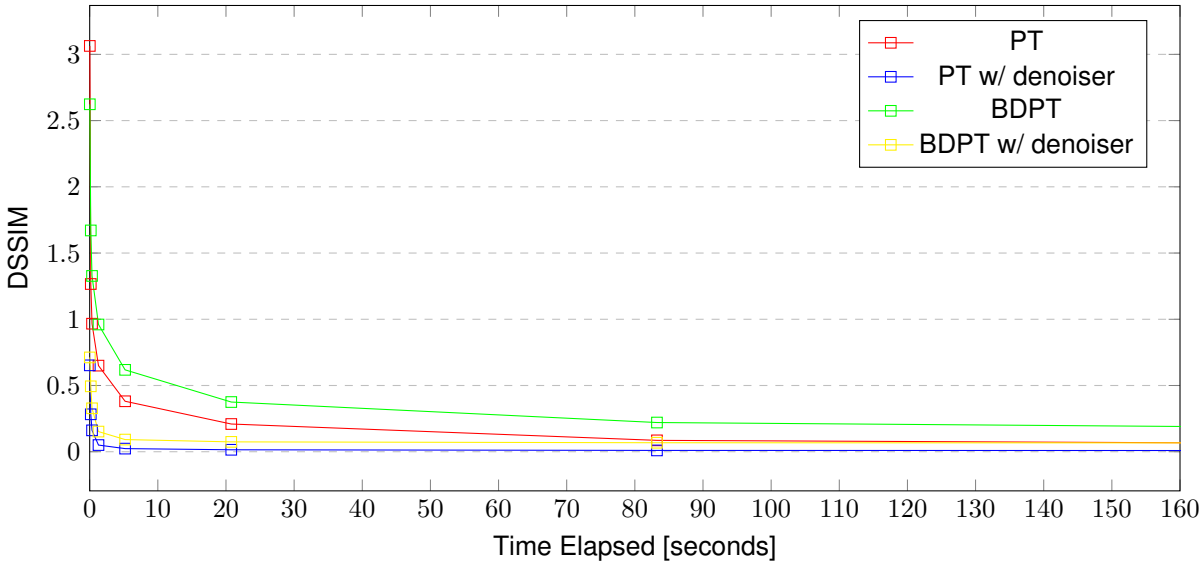


Figure 4.7: ”Japanese Classroom” Structural Dissimilarity over Time

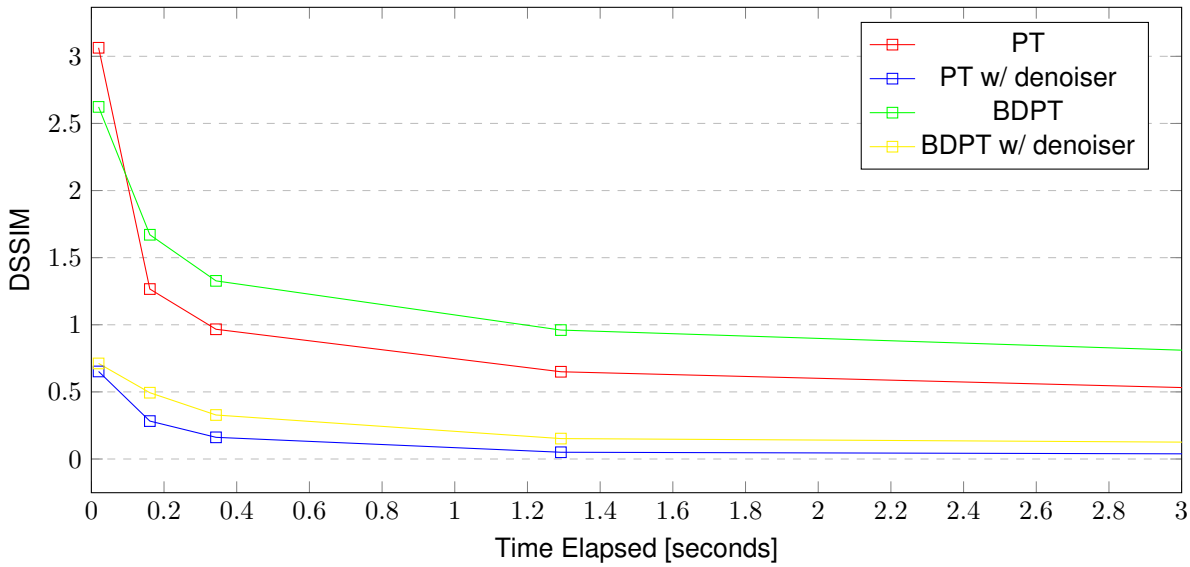


Figure 4.8: ”Japanese Classroom” Structural Dissimilarity over Time, Zoom in of the first 3 seconds

Algorithm	Time [s]	Accumulated Samples	DSSIM	Denoised DSSIM
PT	0.02	1	3.06367398	0.65222797
PT	0.161	8	1.26624604	0.28258904
PT	0.343	16	0.96657428	0.16146306
PT	1.292	64	0.64997775	0.05030262
PT	5.196	256	0.38104223	0.02352854
PT	20.773	1024	0.20920310	0.01472628
PT	83.245	4096	0.08585196	0.00990925
PT	331.638	16384	0.02705733	0.00731504
BDPT	0.048	1	2.62315406	0.71196505
BDPT	0.161	4	1.67064845	0.49393342
BDPT	0.343	7	1.32711299	0.32785179
BDPT	1.292	25	0.96057530	0.15214008
BDPT	5.196	49	0.61820777	0.09179273
BDPT	20.773	405	0.37519143	0.07441921
BDPT	83.245	1562	0.22043505	0.06788891
BDPT	331.638	6299	0.12450785	0.06423006

Table 4.3: ”Japanese Classroom” Image quality..

Chapter 5

Conclusions

For most scenes, we can exceed Real-Time rates using Path Tracing, and Bidirectional Path Tracing can keep up with a Real-Time frame rate at 1920x1080 resolution, provided that though we are only calculating one path per pixel every frame. Overall, results seem to indicate that using Path Tracing in conjunction With a Denoiser is the best approach to obtain the highest image quality in the shortest amount of time. Modern denoising powered by deep learning is an excellent solution that is able to transform low sampling images to images with much higher quality. Even images with high sample counts can benefit from denoising their output, to help clear out some high variance that is very hard to converge. Complex ray tracing algorithms may still be able to outperform the simpler Path Tracer under challenging scenes, such as scenes where light sources are very hard to reach. However, for the vast majority of scenes, light sources are not that hard to intersect, so Path Tracing performs very well in these cases.

5.1 Achievements

Our work tries to pave the way for more research work of light transport algorithms by providing an application to compare implementations of said algorithms leveraging hardware acceleration by utilizing the GPU. Our main goal was to establish a base architecture so that other Monte Carlo methods could be integrated and studied. We managed to successfully implement a ray tracing renderer on the GPU capable of using a Denoiser, supporting multiple scenes and a diverse set of materials. We also implemented two light transport algorithms: Path Tracer and Bidirectional Path Tracer, that we tested extensivelly so that we could answer the problems we were trying to solve.

5.2 Future Work

During the development of the application, we had to compromise on this work's scope; initially, we planned to implement two more light transport algorithms: Photon Mapping and Vertex Connection and Merging. We feel confident that we built a solid base for these algorithms to be implemented on, and

there are more of such algorithms that could be of interest to study when accelerated by the graphics card. The Optix denoiser is not spatiotemporally stable, meaning that if we move the camera while denoising, a flicker artifact is produced; another direction of extending our work would be implementing a spatiotemporally stable denoiser to remove this type of artifact and rendering animation in real-time.

We hope that the growth of GPU accelerated ray tracing keeps evolving so that we can afford even more rays per pixel in the near future to allow for entirely path traced games and faster 3D renderers.

Bibliography

[1] M. Pharr, W. Jakob, and G. Humphreys. *Physically based rendering: From theory to implementation*. Morgan Kaufmann, 2016.

[2] T. Akenine-Moller, E. Haines, and N. Hoffman. *Real-time rendering*. AK Peters/CRC Press, 2018.

[3] J. Bikker and J. van Schijndel. The brigade renderer: A path tracer for real-time games. *International Journal of Computer Games Technology*, 2013, 2013.

[4] J. T. Kajiya. The rendering equation. In *ACM SIGGRAPH computer graphics*, volume 20, pages 143–150. ACM, 1986.

[5] C. R. A. Chaitanya, A. S. Kaplanyan, C. Schied, M. Salvi, A. Lefohn, D. Nowrouzezahrai, and T. Aila. Interactive reconstruction of monte carlo image sequences using a recurrent denoising autoencoder. *ACM Transactions on Graphics (TOG)*, 36(4):98, 2017.

[6] J. Gurney. Caustics image, 2010. URL `www.interstation3d.com/tutorials/mray_caustics/images/real01.jpg`.

[7] J. Gurney. Color bleeding image, 2010. URL `gurneyjourney.blogspot.com/2010/05/color-bleeding.html`.

[8] T. Davidovič, J. Křivánek, M. Hašan, and P. Slusallek. Progressive light transport simulation on the gpu: Survey and improvements. *ACM Transactions on Graphics (TOG)*, 33(3):29, 2014.

[9] E. P. Lafortune and Y. D. Willems. Bi-directional path tracing. 1993.

[10] I. Georgiev. Implementing vertex connection and merging. 2013.

[11] I. Georgiev, J. Krivánek, T. Davidovic, and P. Slusallek. Light transport simulation with vertex connection and merging. *ACM Trans. Graph.*, 31(6):192–1, 2012.

[12] A. Keller, L. Fascione, M. Fajardo, I. Georgiev, P. H. Christensen, J. Hanika, C. Eisenacher, and G. Nichols. The path tracing revolution in the movie industry. In *SIGGRAPH Courses*, 2015.

[13] M. Zwicker, W. Jarosz, J. Lehtinen, B. Moon, R. Ramamoorthi, F. Rousselle, P. Sen, C. Soler, and S. eui Yoon. Recent advances in adaptive sampling and reconstruction for monte carlo rendering. *Comput. Graph. Forum*, 34:667–681, 2015.

[14] C. Schied, A. Kaplanyan, C. Wyman, A. Patney, C. R. A. Chaitanya, J. Burgess, S. Liu, C. Dachsbacher, A. E. Lefohn, and M. Salvi. Spatiotemporal variance-guided filtering: real-time reconstruction for path-traced global illumination. In *High Performance Graphics*, 2017.

[15] C. R. A. Chaitanya, A. Kaplanyan, C. Schied, M. Salvi, A. E. Lefohn, D. Nowrouzezahrai, and T. Aila. Interactive reconstruction of monte carlo image sequences using a recurrent denoising autoencoder. *ACM Trans. Graph.*, 36:98:1–98:12, 2017.

[16] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich. Optix: A general purpose ray tracing engine. *ACM Transactions on Graphics*, August 2010.

[17] G. Soares. Vulkan path tracer with optix denoiser integration, 2020. URL `https://github.com/goncalofds/lift`.

[18] J. B. Daniel Koch Tobias Hector and E. Werness. Ray tracing in vulkan, 2020. https://www.khronos.org/blog/ray-tracing-in-vulkan.

[19] Zhou Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004.

[20] B. Bitterli. Rendering resources, 2016. https://benedikt-bitterli.me/resources/.

[21] kornelski. Image similarity comparison simulating human perception, 2020. https://github.com/kornelski/dssim.

Appendix A

Images

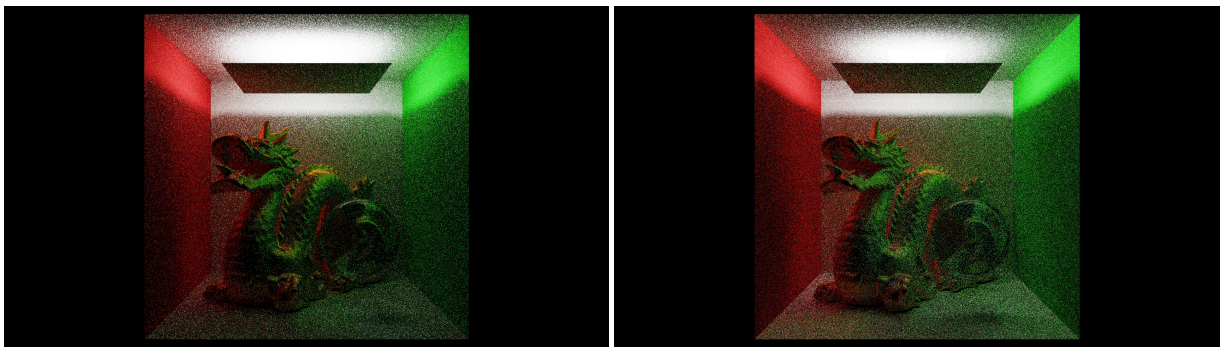


Figure A.1: ”Covered Dragon” Scene Algorithm Comparison - Path Tracer 64 samples (Left) Bidirectional Path Tracer 25 samples (Right)

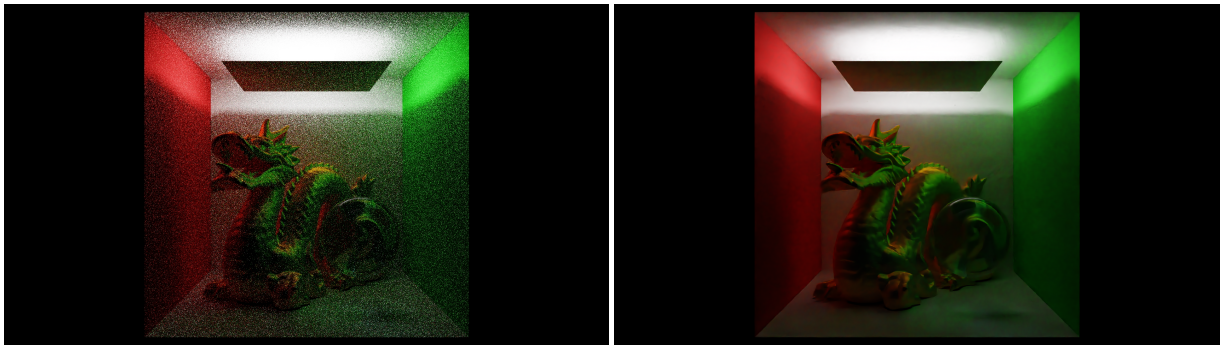


Figure A.2: ”Covered Dragon” Scene Comparison - Path Tracer 64 samples (Left) Path Tracer 64 samples Denoised (Right)

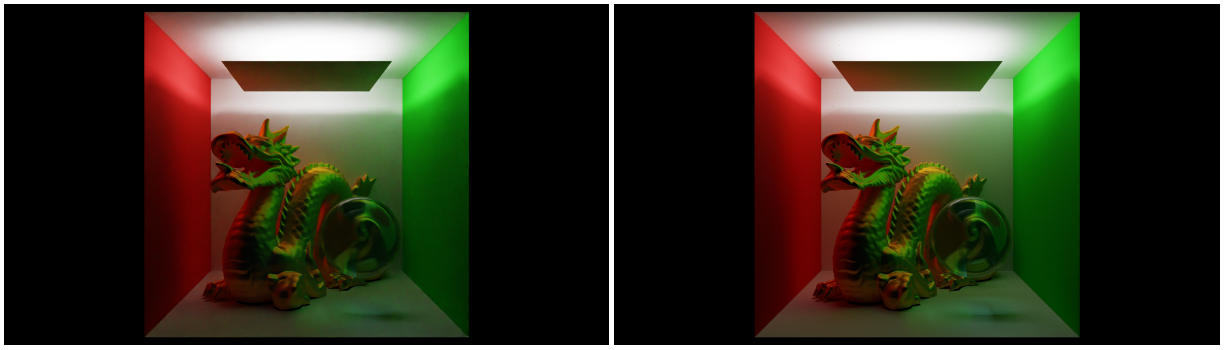


Figure A.3: ”Covered Dragon” Scene Comparison - Path Tracer 1024 samples Denoised (Left) Path Tracer Reference (Right)

Appendix B

Additional Results

Algorithm	Time [s]	Accumulated Samples	DSSIM	Denoised DSSIM
PT	0.005	1	0.39787133	0.05277607
PT	0.039	8	0.26160547	0.02572754
PT	0.078	16	0.23111120	0.01830349
PT	0.315	64	0.15784105	0.00733049
PT	1.248	256	0.07981995	0.00335810
PT	5.019	1024	0.03635421	0.00210892
PT	20.165	4096	0.01291957	0.00147753
PT	96.396	16384	0.00424378	0.00114861
BDPT	0.021	1	0.51703851	0.06156353
BDPT	0.039	2	0.51923883	0.06253241
BDPT	0.078	4	0.39809779	0.06832022
BDPT	0.315	15	0.25522760	0.06919949
BDPT	1.248	60	0.16639614	0.06581431
BDPT	5.019	238	0.11592539	0.06380560
BDPT	20.165	954	0.08846302	0.06315037
BDPT	96.396	4610	0.07062779	0.06270523

Table B.1: ”Teapot Cornell Box” Image quality..

Algorithm	Time [s]	Accumulated Samples	DSSIM	Denoised DSSIM
PT	0.009	1	0.67127678	0.24979752
PT	0.076	8	0.53247813	0.14160439
PT	0.151	16	0.49367307	0.10571623
PT	0.604	64	0.41864464	0.05444154
PT	2.420	256	0.31710149	0.02478038
PT	9.459	1024	0.20818218	0.01338085
PT	38.492	4096	0.10746079	0.00801028
PT	154.397	16384	0.04065980	0.00525720
BDPT	0.024	1	0.70049155	0.23078512
BDPT	0.076	4	0.60348738	0.15192630
BDPT	0.151	7	0.53577495	0.10813048
BDPT	0.604	25	0.38973450	0.06516590
BDPT	2.420	49	0.24681416	0.04564624
BDPT	9.459	405	0.15275928	0.03949232
BDPT	38.492	1562	0.08936442	0.03638964
BDPT	154.397	6299	0.05268307	0.03479745

Table B.2: ”Covered Dragon” Image quality..

Algorithm	Time [s]	Accumulated Samples	DSSIM	Denoised DSSIM
PT	0.006	1	2.33128486	0.55346293
PT	0.049	8	3.49258348	0.33203885
PT	0.1	16	3.04279178	0.22883567
PT	0.4	64	1.88681548	0.10495783
PT	1.606	256	1.15137247	0.04917518
PT	6.393	1024	0.65067683	0.02310055
PT	25.504	4096	0.32394587	0.00891766
PT	102.059	16384	0.13719233	0.00383266
BDPT	0.018	1	2.28904392	0.55360024
BDPT	0.049	3	2.80307390	0.49734572
BDPT	0.1	6	3.19377557	0.37768401
BDPT	0.4	23	2.36387127	0.19685868
BDPT	1.606	89	1.37773956	0.11487482
BDPT	6.393	347	0.72884005	0.07396845
BDPT	25.504	1405	0.36259777	0.05088362
BDPT	102.059	5633	0.15782825	0.03924146

Table B.3: ”Breakfast Room” Image quality..

Algorithm	Time [s]	Accumulated Samples	DSSIM	Denoised DSSIM
PT	0.009	1	0.67127678	0.24979752
PT	0.076	8	0.53247813	0.14160439
PT	0.151	16	0.49367307	0.10571623
PT	0.604	64	0.41864464	0.05444154
PT	2.420	256	0.31710149	0.02478038
PT	9.459	1024	0.20818218	0.01338085
PT	38.492	4096	0.10746079	0.00801028
PT	154.397	16384	0.04065980	0.00525720
BDPT	0.024	1	0.70049155	0.23078512
BDPT	0.076	4	0.60348738	0.15192630
BDPT	0.151	7	0.53577495	0.10813048
BDPT	0.604	25	0.38973450	0.06516590
BDPT	2.420	49	0.24681416	0.04564624
BDPT	9.459	405	0.15275928	0.03949232
BDPT	38.492	1562	0.08936442	0.03638964
BDPT	154.397	6299	0.05268307	0.03479745

Table B.4: ”Covered Dragon” Image quality..