B.Comp. Dissertation

A Real-time Caustics and Global Illumination Rendering Framework with RTX Technology

By

Sun Bangjie

Department of Computer Science

School of Computing

National University of Singapore

2019/2020

B.Comp. Dissertation

A Real-time Caustics and Global Illumination Rendering Framework with RTX Technology

By

Sun Bangjie

Department of Computer Science

School of Computing

National University of Singapore

2019/2020

Project No: H113650

Supervisor:
Dr. Low Kok Lim

Evaluators:
Dr. Low Kok Lim
Dr. Huang Zhiyong

Abstract

This dissertation presents a robust and noise-free rendering framework capable of generating caustics and global illumination in real time. It realizes path integral formulation of light transport with Microsoft's DirectX Raytracing (DXR) application programming interfaces (APIs) and NVIDIA GeForce RTX 20-series GPUs. The NVIDIA RTX technology, by building ray-tracing cores in the GPU hardware, optimizes and accelerates the testing of intersections between rays and geometries such as triangle patches. It, in turn, greatly improves the speed of techniques like path-tracing, bidirectional path-tracing and photon mapping, which were mostly used in offline rendering and needed hours or even days to render photo-realistic images. However, in order to keep rendering within the real-time budget, the total number of rays spawn is still limited. Due to such limitation in sampling, the resultant image will contain an observable amount of noise, which can be visually unsatisfying. It then requires denoising techniques to remove the noise and reconstruct the image. We can see the application of path-tracing and A-SVGF in games such as Quake II RTX, where good image quality and real-time performance can be achieved. However, since the denoising quality is highly dependent on the noise level of the source image, we see opportunities where low-variance rendering techniques such as bidirectional path-tracing can be adopted to further reduce the noise in the source image. We conduct several experiments to compare image quality and rendering speed of various scenes, and perform algorithmic analysis on the rendering techniques.

Keywords: Computer graphics, Caustics, Global illumination, Path tracing, Photon mapping, Denoising, DirectX Raytracing (DXR), RTX GPUs.

Implementation Software and Hardware: Microsoft Visual C++ 2017 (v141), Windows SDK Version 10.0.17763.0, Intel Core i7-9700K CPU @ 3.60GHz, ZOTAC Gaming GeForce RTX 2070 Super 8GB RAM.

Acknowledgments

The success and final outcome of this dissertation required a lot of guidance and assistance from many people and I am extremely privileged to have got this all along the completion of my dissertation. All that I have done is only due to such supervision and assistance and I would not forget to thank them. I respect and thank Dr. Low Kok Lim for providing me such a great opportunity to work on the topic of real-time rendering and giving me all support and guidance which made me complete the project duly. I am extremely thankful to him for providing such a nice support and guidance. I am thankful to and fortunate enough to get constant encouragement, support and guidance from all teaching staffs of the Department of Computer Science in NUS, which helped me in successfully completing the project work. I would also like to extend my sincere esteems to all my families and friends.

Contents

List of Figures

Chapter 1

Introduction

The goal of this dissertation is to build a rendering framework for generating caustics and global illumination in real time and noise-free. To meet our goal, we concentrate on the newly introduced hardware ray-tracing acceleration technology in GPU, empowered by NVIDIA GeForce RTX 20-series, as well as a robust Monte Carlo method for light transport simulation developed by Dr. Eric Veach [27]. By a real-time rendering framework, we mean one that produces images within 30 milliseconds, which correspond to more than 30 frames per second. We also aim to reduce the noise level in the output to yield images that are physically plausible and visually pleasing.

The NVIDIA RTX technology has created opportunities for real-time rendering of caustics and global illumination in games and films. By building ray-tracing cores in the GPU hardware, it optimizes and accelerates the testing of intersections between rays and geometries such as triangle patches. It, in turn, greatly improves the speed of techniques like path-tracing, bidirectional path-tracing and photon mapping, which were mostly used in offline rendering and needed hours or even days to render photo-realistic images. However, in order to keep rendering within the real-time budget (above 30 frames per second), the total number of rays spawn is still limited. One good indicator would be the number of rays generated per pixel, and the other would be the total number of rays per frame. Due to such limitation in sampling, the resultant image will contain an observable amount of noise, which can be visually unsatisfying. It then requires denoising techniques to remove the noise and reconstruct the image. Current denoising techniques include adaptive spatiotemporal variance-guided filtering (A-SVGF) [6], NVIDIA OptiX

AI-Accelerated Denoiser [5] and Blockwise Multi-Order Feature Regression [17]. We can see the application of path-tracing and A-SVGF in games such as Quake II RTX, where good image quality and real-time performance can be achieved. However, since the denoising quality is highly dependent on the noise level of the source image, we see opportunities where low-variance rendering techniques such as bidirectional path-tracing can be adopted to further reduce the noise in the source image.

In our research, we seek to develop a rendering framework that realizes path integral formulation of light transport [27] with Microsoft's DirectX Raytracing (DXR) application programming interfaces (APIs) and NVIDIA GeForce RTX 20-series GPUs. We conduct several experiments to compare image quality and rendering speed of various scenes, and perform algorithmic analysis on the rendering techniques adopted. In the following sections, we start with an overview of the rendering problem in the current production work and why it is important. We also present the high-level structure of our rendering framework. After this brief introduction, we summarize the original contributions of this dissertation, and outline its organization.

1.1 The problems in current production work

Currently, most games adopt rasterization and conventional ray-tracing rendering techniques, which are fast given the advancement of GPUs nowadays. However, these methods are unable to produce global lighting effects such as caustics and color bleeding. Approximation of such effects is often done in the post processing of rendered images, and is not physically plausible. In addition, post-processing often requires multiple rendering passes where only one effect is handled in each pass. It adds complexity to the rendering pipeline.

To achieve global illumination in one go, the most popular algorithm used for production work is path-tracing, which traces rays from the camera into the scene, and bounces the rays based on the hit points' material models. The algorithm has low complexity in both implementation and rendering, but the main problem is that it produces highly noisy images at low sample counts, and it converges to true colors very slowly. As shown in Figure 1.1, the Sponza scene [18] is rendered by the path-tracing algorithm with 1 sample per pixel. It is difficult to recognize

features of the building, such as flags. Thus, in games like Quake II RTX, a denoising pass is needed to remove random noises by blurring the image. Unfortunately, the performance of a denoiser is highly dependent on the noise level of the source image. As shown in Figure 1.2, even after we apply NVIDIA's Recurrent Autoencoder to denoise the image, the output image is still visually unsatisfying.

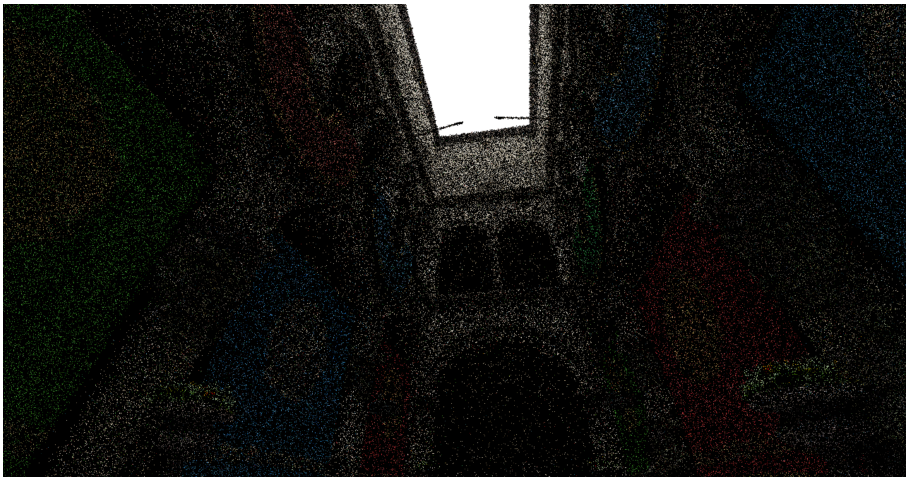

Figure 1.1: Sponza rendered with 1 sample per pixel

Figure 1.2: Sponza denoised by NVIDIA' Recurrent Autoencoder [5]

Despite a great deal of research in denoising techniques, it is important to reduce the noise produced by the rendering algorithm itself. In addition, since the growth in modern GPUs will allow an increasing number of samples per pixel in ray-tracing

related algorithms, it is essential to improve the noise reduction rate per unit increase in sample counts.

The importance of producing physically correct caustics and global illumination in real time is not only seen in games, but also applicable to film industries, and virtual reality. As film industries also aim to produce images as real as possible, the reduction in the rendering time will help improve the efficiency film production and certainly saving costs. Simulating objects and environments as closely as possible to our real world in virtual reality can enhance the immersive experience of users.

1.2 High-level structure of rendering framework

In this section, we will present the rendering framework making use of the path integral formulation, which serves as a foundation for bidirectional path-tracing and metropolis light transport methods [27]. The implementation of the framework is built on top of the Falcor rendering framework [3] developed by the NVIDIA team to avoid repetitive work needed for computer graphics researchers to start from scratch. The Falcor framework provides an abstraction of low-level DXR APIs, loading of scene geometries, as well as graphical user interfaces (GUIs).

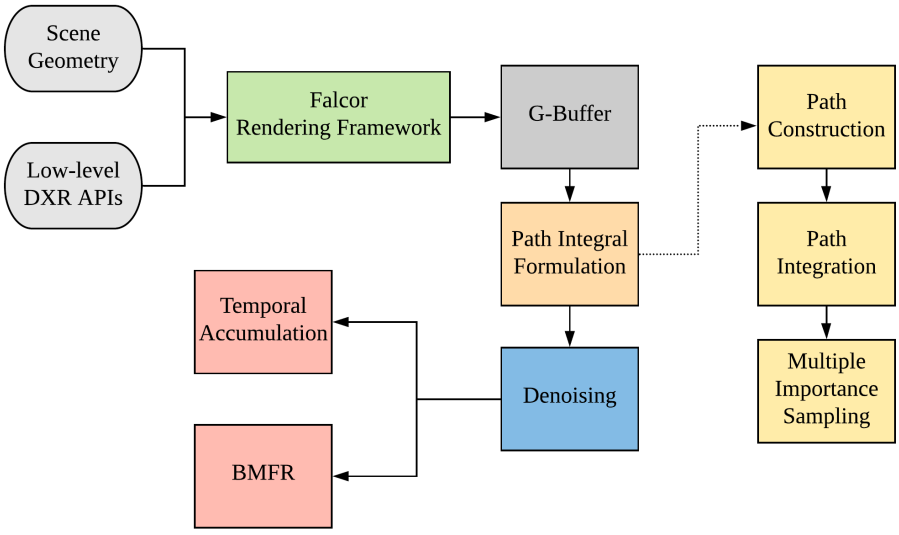

Figure 1.3: High-level diagram of rendering framework

With reference to Figure 1.3, our rendering framework contains three rendering passes: G-buffer, path integral formulation and denoising. G-buffer produces and

stores geometry and material information for all pixels. Path integral formulation consists of path construction, path integration and multiple importance sampling, which can be done in one rendering pass. The denoising pass consists of temporal accumulation and the Blockwise Multi-Order Feature Regression (BMFR) [17].

1.3 Summary of original contributions

Our contributions fall into two parts: a new path integral formulation rendering pass implemented with the latest RTX technology, and a new rendering framework that can produce noise-free caustics and global illumination in real time.

1.4 Reproducibility of work

The implementation of our rendering framework is available at Github repository at `https://github.com/SunBangjie/FYP-BidirectionalPathTracer`.

1.5 Thesis synopsis

The rest of this dissertation is organized as follows. In Chapter 2, we conduct a literature review and explain background knowledge and techniques. Chapter 3 explains the mathematical derivation of the path integral formulation of light transport in details. Chapter 4 provides essential information on RTX architecture and usage of DXR APIs, and presents implementation details of the rendering framework. Chapter 5 compare and analyze the results of our framework with reference images rendered with standard path-tracing. Chapter 6 discusses advantages and limitations of our framework. We conclude the entire dissertation and discuss further directions for future research in Chapter 7.

Chapter 2

Literature Review

In this chapter, we will describe relevant terms and definitions, introduce the background knowledge involved and review related work published.

2.1 Light patterns

Caustics are often seen when light rays are reflected or refracted by some optical medium and converge to a single point on a diffuse surface to produce area with above-average brightness [9], with reference to Figure 2.1. Examples of caustics can be easily produced by sunlight shinning on surface of water, or through glasses.

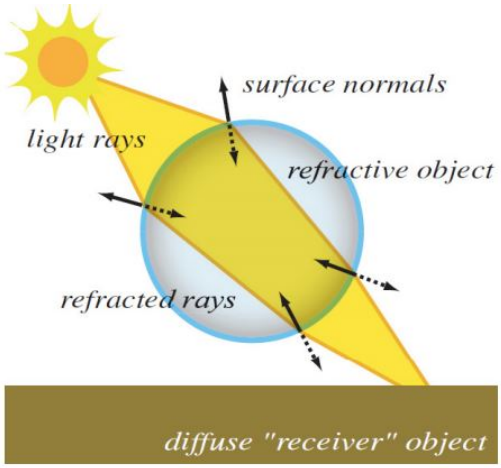

Figure 2.1: Caustics model

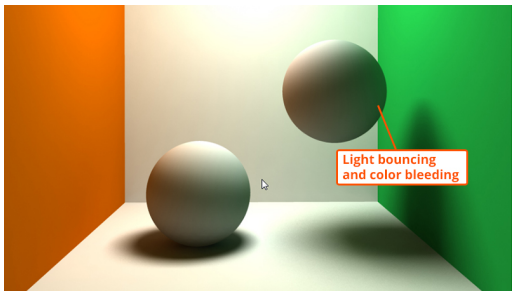

Figure 2.2: Color bleeding

Global illumination is a system that models how light is bounced off of surfaces onto other surfaces (indirect light) rather than being limited to just the light that hits a surface directly from a light source (direct light) [26]. For example, in Figure 2.2,

green color of the wall is cast onto the sphere on the right side of the image. This color-bleeding effect comes from indirect lighting because the green light is not cast directly from a light but rather is the result of a white light being cast onto the green wall which is then bleeding onto the nearby sphere.

2.2 Light transport mathematical framework

In this section, we describe basic terms and definitions used in light transport problems. Since in computer graphics, we simulate how light behaves in the real world, it is important to understand radiometry which describes the units of light. We also derive some mathematical equations to describe how light is transported from a light source to a view point.

2.2.1 Radiometry

In a light transport model, we can assume that the light moves in straight lines at infinite speed. This is the essential assumption for most ray-tracing algorithms, and even with this assumption, we are still able to simulate almost all the lighting effects visible to human eyes.

2.2.1.1 Flux

Flux, often denoted as ϕ, is the flow of energy measured in Watt. It is the amount of energy that passes through an area in a given time. We can write flux as:

$$\phi = \frac{dQ}{dt}$$

where dQ is a unit of energy and dt is a unit of time. Since flux is independent of time, it is the basic unit of many light transport algorithms.

2.2.1.2 Irradiance and radiosity

The irradiance E is defined as the amount of flux that is arriving at per unit of surface area dA. On the other hand, the radiosity B is the amount of flux that is leaving per unit of surface area. Radiosity is also known as radiant exitance M. These can be written as:

$$E = \frac{d\phi}{dA} \qquad B = M = \frac{d\phi}{dA}$$

2.2.1.3 Radiant intensity

The radiant flux per unit solid angle, $d\omega$ is measured by the term intensity I. It can be written as:

$$I = \frac{d\phi}{d\omega}$$

2.2.1.4 Radiance

Radiance is the primary way of describing the light in many rendering techniques and algorithms, since radiance is the value that mostly resembles color and it does not change over distance. We can say that radiance equals the amount of radiant flux found in a single light ray. To evaluate that, we need to find the amount of flux that exits or enters a single point on a surface, from or in a single direction. This is equal to the flux per area per solid angle, which can be written as:

$$L(x, \omega) = \frac{d^2\phi}{d\omega \, dA \, |\omega \cdot N_x|}$$

where $L(x, \omega)$ is the radiance of point x in the direction ω and N_x is the normal vector at point x. We can then find the flux by integrating over the surface area A in all directions Ω within the hemisphere above the point x:

$$\phi = \int_{x \in A} \int_{\omega \in \Omega} L(x, \omega) \, |\omega \cdot N_x| \, d\omega \, dx$$

2.2.2 Rendering equation

The rendering equation, proposed by James Kajiya [15] in 1986, is based on the physics of light and describes the law of conservation of energy during the flow of light through the rendered scene. The rendering equation has served as a fundamental basis for all algorithms simulating global illumination. Equation 1 describes the amount of light reflected from the surface point x in the direction ω_o, which can be computed as the emitted radiance plus the reflected radiance:

$$L_o(x, \omega_o) = L_e(x, \omega_o) + L_r(x, \omega_o) \tag{1}$$

$L_e(x, \omega_o)$ is the emitted radiance from point x in the direction ω_o. Only light sources and emissive materials can have positive emitted radiance, and other surfaces will have zero. $L_r(x, \omega_o)$ is the radiance reflected by the point x in the direction ω_o, from

all incoming lights. We can represent the incoming radiance for a single direction ω_i as $L_i(x, \omega_i)$.

To find out how much incoming radiance is reflected, we can multiply L_i by a factor called Bidirectional Reflectance Distribution Function (BRDF). It is a function f_r that defines the reflecting behavior of a surface from arbitrary incidences. For any light ray striking the surface at a specified angle of incidence, the BRDF gives the ratio between the radiance reflected in direction ω_o and the irradiance incident from direction ω_i. The BRDF is decribed by Equation 2 below:

$$f_r(x, \omega_i \rightarrow \omega_o) \equiv \frac{L_r(x, \omega_o)}{E_i(x, \omega_i)} \equiv \frac{L_r(x, \omega_o)}{L_i(x, \omega_i) \cos(\theta_i) d\omega_i} \tag{2}$$

where E_i denotes the irradiance (the amount of flux hitting the surface) and L_r is the radiance (the differential flux density emitted from the surface).

Thus, by integrating over all directions on the hemisphere above the point x, we can compute the reflected emittance as shown in Equation 3. The $cos(\theta_i)$ can be calculated using the dot product of w_i and normal vector N_x on point x according to the geometry definition of dot product.

$$L_r(x, \omega_o) = \int_{\omega_i \in \Omega} f_r(x, \omega_i \rightarrow \omega_o) L_i(x, \omega_i) |\omega_i \cdot N_x| d\omega_i \tag{3}$$

Therefore, after combining Equation 1 and Equation 3, we can get the rendering equation:

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_{\omega_i \in \Omega} f_r(x, \omega_i \rightarrow \omega_o) L_o(x', -\omega_i) |\omega_i \cdot N_x| d\omega_i \tag{4}$$

It is important to notice that we can replace $L_r(x, \omega_o)$ with $L_o(x', -\omega_i)$ where x' is the previous surface point that the light leaves before entering point x, and the outgoing direction ω_o at point x' is the opposite of the incoming direction ω_i at point x. Since both radiance terms are now L_o, we can drop the subscript for convenience.

2.3 Rendering techniques

There are numerous rendering techniques that attempt to simulate caustics and global illumination in graphics engines like Unity and Maya, as well as a lot of games such as Battlefield V. Some of them adopts the full ray-tracing approach, namely path-tracing or photon mapping, and some of them use a hybrid approach

that combines rasterization, also known as scanline rendering, and local ray-tracing to approximate lighting effects. In this section, we will review and evaluate these algorithms, and point out their advantages and limitations.

2.3.1 Path-tracing

The path-tracing approach was introduced by James Kajyiya [15] in 1986 together with the rendering equation, which quickly became a benchmark in computer graphics. Path-tracing attempts to trace the light paths from the camera into the scene, bouncing off both specular and diffuse surfaces until light rays intersect any light source or out of the scene. The main difference between path-tracing and conventional ray-tracing is that, ray-tracing samples lights directly when a ray hits a diffuse surface and stops further tracing. On the other hand, in path-tracing, the ray's new direction is sampled randomly among all possible directions within the hemisphere over the surface point, using the BRDF.

The path-tracing algorithm is unbiased as it does not introduce any systematic error, or bias, into the radiance approximation. It can produce reference image to compare against other rendering techniques. However, when the number of samples is insufficient, it generally causes highly noisy output image. Due to the fact that the probability of an intersection between a ray and a light source is very low, most traced paths will not contribute to the final image, causing high computational waste. It also cannot consistently handle caustics generated from a point light source, as it is highly unlikely to randomly generate the particular path that directly reflects into the point.

2.3.2 Forward ray-tracing

The path-tracing approach is one the of backward ray-tracing methods. On the other hand, we also have forward ray-tracing methods that trace the light particles (photons) from the light source to the surfaces and bounces off. Although forward ray-tracing can most accurately determine the color of each surface point, it is highly inefficient. This is because many rays from the light source never come through the view plane and into the eye. Tracking every light ray from the light source down means that many rays will go to waste because they never contribute to the final

image as seen from the eye. However, the main advantage of this approach is that it can generate caustics much more easily than path-tracing because every path that contributes to the image is originated from the light source.

2.3.3 Hybrid ray-tracing

Since both forward ray-tracing and backward ray-tracing have their drawbacks, we can combine them to form hybrid solutions. After a certain level of forward ray-tracing is performed, the algorithm records the data, and then goes on to perform backward ray-tracing. The final coloring of the scene takes both the backward ray-tracing and the forward ray-tracing calculations into account. We will discuss some of the major algorithms that adopt this hybrid approach in the following subsections.

2.3.3.1 Adaptive radiosity textures bidirectional ray-tracing

The bidirectional ray-tracing method making use of adaptive radiosity textures [11] consists of two rendering passes. The first pass emits rays from light sources into the scene. Whenever one of the rays hits a diffuse surface, its radiance is reduced according to the reflection and the energy difference is saved in an illumination map at intersections. The second pass carries out conventional ray-tracing, with the illumination maps providing additional information for the calculation of the surfaces' illumination. This method has limitation in the accuracy in the radiance approximation, and it is costly in terms of memory usage to store texture maps.

2.3.3.2 Bidirectional path-tracing

Bidirectional path-tracing is another hybrid ray-tracing solution. It first pursues the light rays through the scene, and stores the information of every hit point in the memory to construct a light path. Similarly, it also traces rays from the eye, and store all the hit points to construct an eye path. It then connects all the points from two paths and test the visibility between any two points. This technique can converge much faster than unidirectional path-tracing methods to the physically correct radiance as the number of samples increases, and hence it can produce less noise. However, the main problem of this algorithm is that it has relatively high time and space complexity.

In general, hybrid solutions will compromise speed and accuracy, but can produce images with relatively low noise compared to the path-tracing or forward ray-tracing approach.

2.3.4 Photon mapping

H.W. Jensen et al. [12] introduced the photon mapping method which can generate caustics and global illumination in two passes. The first pass focuses on the construction of photon maps, and the second pass render the scene by retrieving information stored in the photon maps. The main problem of this method is the difficulty in managing the huge mass of gathered information. The key to an acceptable computing time lies in the choice of the right data structure. The best structure to store a photon map is a balanced k-dimensional tree, which enables fast neighborhood searches, and is compact and efficient. One photon can be stored in only 20 bytes. However, since a large number of photons emitted from light sources are needed to produce visually pleasing images, the memory consumption might still be a problem.

Further work to improve photon mapping was seen in progressive photon mapping [25], where the photon tracing pass results in an increasingly accurate global illumination solution that can be visualized in order to provide progressive feedback. Progressive photon mapping uses a new radiance estimate that converges to the correct radiance value as more photons are used. It is not necessary to store the full photon map, and it is possible to compute a global illumination solution with any desired accuracy using a limited amount of memory. Stochastic progressive photon mapping [24] then extends the progressive photon mapping for simulating global illumination with effects such as depth-of-field, motion blur, and glossy reflections.

2.4 Denoising techniques

The rendering techniques mentioned in the previous section were mostly used for offline rendering in film industries before the rise of RTX technology. It may take hours or days to render an image at that time. Even with the latest RTX GPUs, we still have limitations in the number of rays that can be generated for a frame in order to keep rendering within the real-time budget. This, in turn, leads to a

noisy image which requires a denoising stage to remove the noise and reconstruct the image. In this section, we will discuss several types of denoising methods.

2.4.1 Sampling

Many different sampling methods were introduced to reduce noise in the final image rendered by path-tracing. The noise comes from stochastic sampling with biased noise generated by quasi-random sequences, weighing different rays outputs with multiple importance sampling (MIS) and next event estimation (NEE) [27]. A great deal of research handles denoising by improving the sampling method. Recently, the use of low discrepancy sampling [13] and tillable blue noise [4] has been used by Unity Technologies, Marmoset Toolbag and NVIDIA in real-time ray tracers. Christopher Kulla et al. [7] adopt importance sampling techniques to allow efficient calculation of direct and indirect lighting from arbitrary light sources in both homogeneous and heterogeneous participating media. Konstantin Shkurko et al. [22] introduce a new motion blur computation method for path-tracing that provides an analytical approximation of motion blurred visibility per ray. Rather than relying on timestamped rays and Monte Carlo sampling to resolve the motion blur, they associate a time interval with rays and directly evaluate when and where each ray intersects with animated object faces.

2.4.2 Signal processing

A conventional denoising approach would be signal processing. Such techniques, including Gaussian, Median [16], Bilateral, À-Trous [8], and Guided [14] filters, have been used to average out and blend regions with low variance. In particular, guided filters driven by feature buffers such as G-Buffer attachments have seen much success. Machine learning algorithms such as those employed by Intel's Open Image Denoise (OIDN), NVIDIA's Denoising Autoencoder [5] were used for offline renders, with NVIDIA [20] attempting to use trained neural networks in real-time.

2.4.3 Spatio-temporal

Spatial-temporal techniques have seen a resurgence in new literature. Examples include Spatio-Temporal Filter [19], the Spatio-Temporal Variance Guided Filter

(SVGF) [21], Spatial Denoising [1], Adaptive SVGF (A-SVGF) [6], Blockwise Multi-Order Feature Regression (BMFR) [17], and Temporally Dense Ray Tracing [2].

2.5 Summary

To build an unbiased rendering framework that approximates radiance of scene surfaces as accurately as possible, we have to manage the trade-offs between speed, memory consumption and noise level in the output image, given the limitations in the computational resources. Preliminary experiments were conducted to test the performance of RTX GPUs and the optimal number of rays that can be spawn for a frame within the real-time budget. We believe that a hybrid ray-tracing approach is a balanced one that can produce unbiased and accurate images with relatively low variance. Then we can apply light-weight spatio-termporal denoising techniques such as BMFR to handle the noise.

Chapter 3

Path Integral Formulation of Light Transport

In this chapter, we will derive the mathematical framework of the path integral method, and the rendering technique, bidirectional path-tracing, that arises from this method. We will then discuss an essential sampling method called multiple importance sampling which can be used in variance reduction.

3.1 Path integral mathematical framework

In Chapter 2, we have discussed the rendering equation which can be used to express the light transport problem. The rendering methods that arise from the rendering equation usually start at a point, such as a camera or a light source, and then recursively find the next point by shooting out a ray into the scene. This means that we can only find the next point by locally evaluating the current surface point. The path integral formulation method, on the other hand, gives a global view of the problem. For any two points in the scene, we can connect them by creating an edge between them and perform a visibility test to see if they are mutually visible.

3.1.1 Three point form

We will start by transforming Equation 4, which uses an integral over solid angle, into the light transport equation that uses an integral over surface area. In the rendering equation, we find the incoming radiance on a point x by integrating over all incoming directions. Another way is to integrate over all surface points in the scene. To do this, we have to replace the directions ω_i and ω_o with surface

points x' and x, where x' is the point that gives out energy and x is the point that receives energy. Thus, we can find the outgoing radiance from point x' to point x by rewriting the radiance term as:

$$L(x' \to x) = L(x', \omega_o)$$

where $\omega_o = \vec{x'x}$. The vector $\vec{x'x}$ represents the direction from x' to x. Next, we can rewrite the BRDF as:

$$f_r(x'' \to x' \to x) = f_r(x', \omega_i, \omega_o)$$

where $\omega_i = \vec{x'x''}$ and $\omega_o = \vec{x'x}$. Since we have to integrate over a set of surface points, A, instead of the solid angle, Ω, we need to find the relationship between unit area and the unit solid angle:

$$dw = \frac{dA\,|N_{x''} \cdot \vec{x''x'}|}{||x'' - x'||^2}$$

Then we can rewrite the rendering equation with respect to surface points in the area domain:

$$L(x' \to x) = L_e(x' \to x) + \int_{x'' \in A} L(x'' \to x') f_r(x'' \to x' \to x) G(x'' \leftrightarrow x') dx'' \quad (5)$$

where we can find the outgoing radiance from point x' to point x by integrating over all surface points $x'' \in A$. We can extract out the geometric term, $G(x'' \leftrightarrow x')$ to make the equation more compact. The G function is defined as:

$$G(x'' \leftrightarrow x') = V(x'' \leftrightarrow x') \frac{|N_{x''} \cdot \vec{x''x'}|\,|N_{x'} \cdot \vec{x'x''}|}{||x'' - x'||^2}$$

$V(x'' \leftrightarrow x')$ is the visibility function. $V(x'' \leftrightarrow x') = 1$ if we can shoot a ray from x'' to x' without hitting anything else, and $V(x'' \leftrightarrow x') = 0$ otherwise.

3.1.2 The Neumann Series Expansion

Since the Equation 5 has the same value L on both sides, it cannot be directly evaluated. Instead, we can recursively replace the value on the right side with itself. This is called the Neumann Series Expansion. Now, we can apply it to the equation,

and label the surface points hit in sequence as $x_0, x_1, x_2, x_3, ...$:

$$\begin{aligned} L(x_1 \to x_0) = L_e(x_1 \to x_0) &+ \int_A L_e(x_2 \to x_1) f_r(x_2 \to x_1 \to x_0) G(x_2 \leftrightarrow x_1) dA \\ &+ \int_A \int_A L_e(x_3 \to x_2) f_r(x_3 \to x_2 \to x_1) G(x_3 \leftrightarrow x_2) \\ &f_r(x_2 \to x_1 \to x_0) G(x_2 \leftrightarrow x_1) dA dA \\ &+ ... \end{aligned} \tag{6}$$

Intuitively we can see it as the summation of the radiance found in all the paths of length 1, 2, 3 and so on. We can write the Equation 6 in a compact form as:

$$L(x_1 \to x_0) = \sum_{i=1}^{\infty} L(\vec{p_i})$$

where $\vec{p_i}$ is a path of length i:

$$\vec{p_i} = x_0, x_1, x_2, ..., x_i$$

where x_0 is the point on the eye, and x_i is a point on the light source. $L(\vec{p_i})$ gives us the amount of radiance on path $\vec{p_i}$, and can be defined as:

$$L(\vec{p_i}) = \int \cdots \int_A L_e(x_i \to x_{i-1}) (\prod_{j=1}^{i-1} f_r(x_{j+1} \to x_j \to x_{j-1}) G(x_{j+1} \leftrightarrow x_j)) dA ... dA \tag{7}$$

3.1.3 Monte Carlo estimation

The overall problem we are trying to solve, is to find the radiance L_p that goes through a pixel p:

$$L_p = \int_D C(\vec{p})\, dD \tag{8}$$

where D is the set of all possible light paths in the scene, $\vec{p}$ represents a single light path and C is the measurement contribution function that measures the radiance contribution of the particular path. C is similar to Equation 7. For a particular path $\vec{p_i}$ of length i, it can be written as:

$$C(\vec{p_i}) = L_e(x_i \to x_{i-1}) (\prod_{j=1}^{i-1} f_r(x_{j+1} \to x_j \to x_{j-1}) G(x_{j+1} \leftrightarrow x_j)) W_e(x_1 \to x_0)$$

where $W_e(x_1 \to x_0)$ is the potential flowing through the view plane. For a pixel, $W_e = 1$ if a light ray hits it, and $W_e = 0$ otherwise.

With Monte Carlo estimation, we can approximate Equation 8 by taking the average of N randomly sampled paths:

$$L_p = \frac{1}{N} \sum_{i=1}^{N} \frac{C(\vec{p_i})}{P(\vec{p_i})}$$

where $P(\vec{p_i})$ is the probability distribution function (PDF) of sampling the path $\vec{p_i}$. The PDF of a path depends on all the choices made to create the path. Since we usually get the PDF in the solid angle domain when we use BRDF to sample a new direction, it is important to convert it into the area domain. The PDF with respect to surface area $P(x')$ can be expressed in terms of $P(\omega_o)$:

$$\begin{aligned} P(x') &= P(\omega_o) \frac{|N_{x'} \cdot \vec{x'x}| \, |N_x \cdot \vec{xx'}|}{||x - x'||^2} \\ &= P(\vec{x'x}) \, G(x' \leftrightarrow x) \end{aligned}$$

where x' is the current surface point and x is the next hit point in the sampled direction $\omega_o = \vec{xx'}$.

With all these derivations, we are now able to compute the radiance of each pixel by sampling and creating a number of paths in the scene. We can proceed to explain how a bidirectional path-tracing method can be used to construct such paths and gather them into the final results.

3.2 Bidirectional path-tracing

As introduced in Chapter 2, the bidirectional path-tracing is a hybrid ray-tracing approach that combines both the path-tracing and light-tracing methods to solve the light transport problem. It is also an algorithm arising from the path integral formulation. In this section, we will discuss the mathematical derivation of the algorithm.

3.2.1 Sampling the paths

In bidirectional path-tracing, we can trace rays from both the view point and the light source. We can create a subpath $P_{eye,s}$ of length s starting from the eye:

$$P_{eye,s} = x_0, x_1, x_2, ..., x_s$$

where x_0 is the view point. We also create another subpath $P_{light,t}$ of length t starting from the light:

$$P_{light,t} = y_0, y_1, y_2, ..., y_t$$

where y_0 is a point on the light. The length of each subpath s or t can be determined by either setting an upper limit or by using a method called Russian Roulette to randomly decide if the path should be terminated. We can then combine the two subpaths by forming an edge between x_s and y_t to create a complete path P_i of length $i = s + t + 1$ from the view point to the light:

$$P_i = x_0, x_1, x_2, ..., x_s, y_t, y_{t-1}, y_{t-2}, ..., y_0$$

To make sure it is a legal path we need to do a visibility test between points x_s and y_t.

3.2.2 Contribution of a path

The radiance flowing through the pixel p can be approximated using the Monte Carlo estimator:

$$L_p = \sum_{s=0}^{N_{eye}} \sum_{t=0}^{N_{light}} w_{s,t} V(s,t) C_{s,t} \tag{9}$$

where N_{eye} and N_{light} are the lengths of eye subpath and light subpath respectively. $V(s,t)$ is the visibility test between end points of two subpaths. $w_{s,t}$ is a weight function. $C_{s,t}$ is the contribution of the eye subpath of length s and the light subpath of length t.

Based on the variations of path lengths s and t, there are 4 distinct cases for us to deal with when evaluating $C_{s,t}$:

- When $s = 0$ and $t = 0$, there is only one point on each path. The light is directly visible to the eye. We can form an edge between x_0 and y_0, and the contribution is:

$$C_{0,0} = L_e(y_0 \to x_0) G(y_0 \leftrightarrow x_0)$$

- When $s > 0$ and $t = 0$, there is only one point on the light subpath, and more than one points on the eye subpath. This corresponds to the standard path-tracing algorithm. Instead of using y_0 as the point on the light source,

we can reduce the noise in the image by sampling a new point on the light source. The contribution in this case is:

$$C_{s,0} = \frac{L_e(y_0 \to x_s)}{P(y_0)} f_r(x_{s-1} \to x_s \to y_0) G(y_0 \leftrightarrow x_s) \\ (\prod_{i=1}^{s-1} \frac{f_r(x_{i-1} \to x_i \to x_{i+1}) \left| N_{x_i} \cdot x_i \vec{x_{i+1}} \right|}{P(x_i \vec{x_{i+1}})})$$

- When $s = 0$ and $t > 0$, there is only one point on the eye subpath, and more than one points on the light subpath. This corresponds to the light-tracing algorithm. The contribution in this case is:

$$C_{0,t} = \frac{L_e(y_0 \to y_1)}{P(y_0) P(y_0 \to y_1)} (\prod_{i=1}^{t-1} \frac{f_r(y_{i-1} \to y_i \to y_{i+1}) \left| N_{y_i} \cdot y_i \vec{y_{i+1}} \right|}{P(y_i \vec{y_{i+1}})}) \\ f_r(y_{t-1} \to y_t \to x_0) G(y_t \leftrightarrow x_0) \frac{W_e(y_t \to x_0)}{P(x_0)}$$

  Note that we have to take the potential flowing through the view plane, W_e, into account because not all rays can hit the view plane.

- When $s > 0$ and $t > 0$, there is more than one points on the each subpath. We can estimate the radiance that reaches the eye, on a path connected by the end vertices of both eye and light subpaths. The contribution is:

$$C_{s,t} = \frac{L_e(y_0 \to y_1)}{P(y_0) P(y_0 \to y_1)} (\prod_{i=1}^{t-1} \frac{f_r(y_{i-1} \to y_i \to y_{i+1}) \left| N_{y_i} \cdot y_i \vec{y_{i+1}} \right|}{P(y_i \vec{y_{i+1}})}) \\ f_r(y_{t-1} \to y_t \to x_s) G(y_t \leftrightarrow x_s) f_r(x_{s-1} \to x_s \to y_t) \\ (\prod_{i=1}^{s-1} \frac{f_r(x_{i-1} \to x_i \to x_{i+1}) \left| N_{x_i} \cdot x_i \vec{x_{i+1}} \right|}{P(x_i \vec{x_{i+1}})})$$

3.2.3 Weight of a path

For a path of length k, we can form $k + 1$ possible combinations of eye subpath of length s and light subpath of length t, as follows:

$$s = 0, \quad t = k$$
$$s = 1, \quad t = k - 1$$
$$\vdots$$
$$s = k - 1, \quad t = 1$$
$$s = k, \quad t = 0$$

We can treat the weight of a particular path as how likely the path will contribute to the final radiance, and we are doing a weighted sum of all the possible contributions of path with length k. Thus, the weights of all these contributions should sum up to one.

$$\sum_{s=0}^{k} w_{s,k-s} = 1 \quad \text{for} \quad k = 0, 1, 2, ...$$

We can have various ways to distribute the weights. We can see the $w_{s,t}$ as a distribution function, and use heuristics to solve it. The naive way to do it is to use a uniform distribution where each contribution has the same weight:

$$w_{s,t} = \frac{1}{s+t+1}$$

3.3 Multiple importance sampling

Using multiple importance sampling, we can distribute the weights in a better way to reduce noise. This method looks at how the same path can be sampled in different ways as well as the probability of sampling a particular path. Due to restrictions on computational resources, we adopt an one-sample method where we take only one sample from each possible path.

We can define the probability of sampling a particular path as p_i, and the total probability of sampling the path of length k is:

$$\sum_{i=0}^{k+1} p_i$$

We can then apply different heuristics to find $w_{s,t}$ based on the probabilities. One way is to use the balance heuristic to find the weight of a path with probability p_j:

$$w_{s,t} = \frac{p_j}{\sum_{i=0}^{k+1} p_i}$$

Another way is to use the power heuristic:

$$w_{s,t} = \frac{p_j^{\beta}}{\sum_{i=0}^{k+1} p_i^{\beta}} \tag{10}$$

where $\beta = 2$ is found to be the optimal [27].

The PDF p_j for a path can be evaluated by multiplying the PDF p^L of sampling a light subpath and PDF p^E for an eye subpath. We can define p_i^L and p_i^E to represent

the probabilities for generating the first i vertices of the light and eye subpaths respectively. These are defined as:

$$\begin{aligned} p_1^L &= P(y_0) \\ p_2^L &= P(y_{i-2} \to y_{i-1}) G(y_{i-2} \leftrightarrow y_{i-1}) p_{i-1}^L \quad \text{for} \quad i \geq 2 \end{aligned}$$

and similarly,

$$\begin{aligned} p_1^E &= P(x_0) \\ p_2^E &= P(x_{i-2} \to x_{i-1}) G(x_{i-2} \leftrightarrow x_{i-1}) p_{i-1}^E \quad \text{for} \quad i \geq 2 \end{aligned}$$

Using these probabilities, we can find the combined PDF p_j as:

$$p_j = p_s^L \, p_t^E$$

3.4 Summary

In this chapter, we have derived the mathematical framework for the path integral formulation and the bidirectional path tracing with multiple importance sampling. With the theoretical fundamentals, we can proceed to the implementation details of our rendering framework, as well as important optimizations.

Chapter 4

Implementation Details

In this chapter, we will introduce the DirectX Ray-tracing (DXR) pipeline and the usage of various shaders. We will then present the implementation details of the bidirectional path-tracing, introduced in Chapter 3, in our rendering framework.

4.1 Overview of shaders in DXR pipeline

DXR is designed to handle real-time ray-tracing and deal with the GPUs with RTX cores. We will look at the DXR pipeline in details and provide essential functions for ray-tracing.

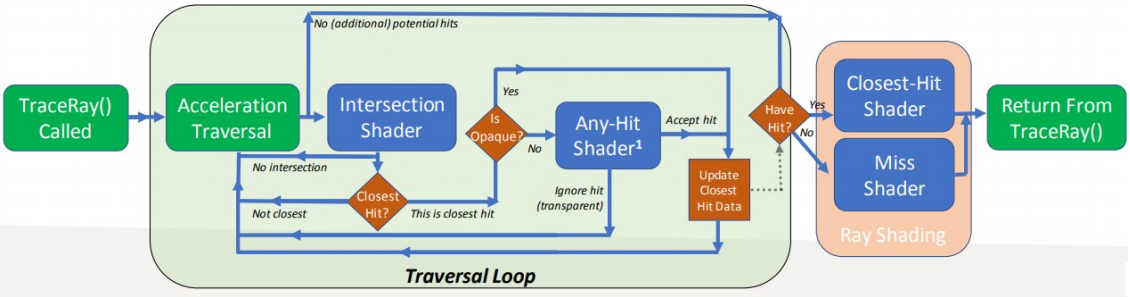

Figure 4.1: High-level diagram of DXR pipeline. Fixed stages are in green and programmable shaders are in blue. Modified from [29]

With reference to Figure 4.1, the DXR pipeline is split into five shaders:

1. **Ray generation shader** - defines how to start ray tracing and runs once per pass or algorithm.

2. **Intersection shader(s)** - define how rays intersect geometry and geometric shapes, and are widely reusable (often in-built and in-default).

3. **Miss shader(s)** - define behavior when rays miss geometry, e.g. user-defined behavior to use background color or read from environment maps.

4. **Closest-hit shader(s)** - run once per ray to shade the final hit.

5. **Any-hit shader(s)** - run once per hit to determine accept (when opague) or ignore hit (when transparent).

The DXR pipeline can be viewed as abstractions of various behaviours needed to perform ray-tracing. The ray generation shader is the entry point to spawn rays for each pixel. The TraceRay() method generates a ray, which can be described by the built-in data structure RayDesc. The ray is then tested with scene geometries with the acceleration traversal, empowered by the ray-tracing hardware. The intersection shader defines how to perform the testing of ray-geometry intersection. We usually keep it as default when handling triangle patches. When there is a hit, we can process it in the closest-hit shader. In this shader, we can extract the geometry and material information of the surface point, and perform direct and indirect lighting computation. When a ray does not hit anything, a miss shader will be called to handle it. The any-hit shader is used to test the transparency of any surface point, and will ignore the hit if it is transparent.

Algorithm 1: An abstraction of shooting a ray

Global: scene structure *scene*
Input: Ray origin o, ray direction d, minimum intersection distance $tmin$ and maximum intersection distance $tmax$
Output: Hit or miss information *payload*

Procedure `ShootRay`**:**
    initialize $RayDesc\ ray$
    $ray.Origin \leftarrow origin$
    $ray.Direction \leftarrow direction$
    $ray.TMin \leftarrow tmin$
    $ray.TMax \leftarrow tmax$
    initialize $RayPayload\ payload$
    $TraceRay(scene, ray, payload)$

We provide an abstraction of details needed to shoot a ray in Algorithm 1. It takes in the ray origin, direction, minimum and maximum intersection distances as parameters, and will get the hit or miss information stored in the RayPayload. Note that in the pseudo-code, we omit the indices of hit and miss shader to use in the TraceRay() function.

4.2 Implementation of bidirectional path-tracing

In this section, we will explain the implementation details of bidirectional path-tracing with pseudo-codes.

4.2.1 Overview

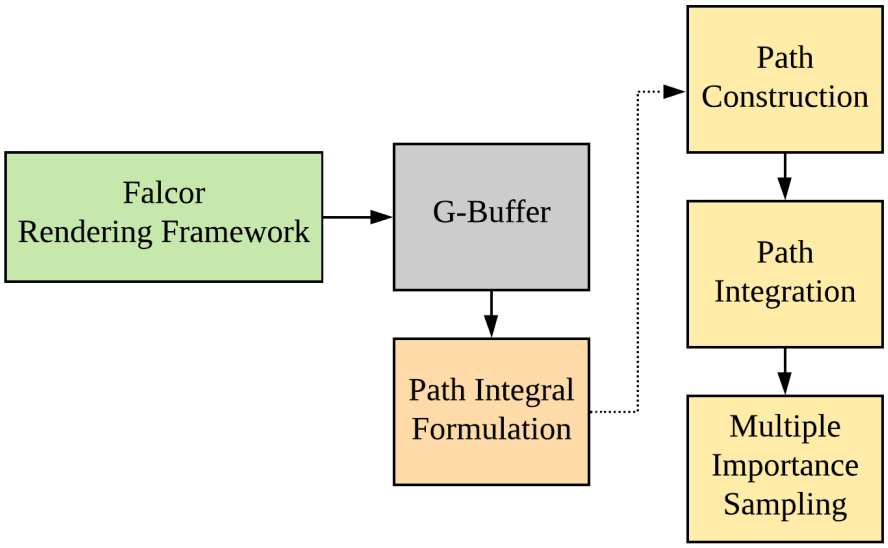

Figure 4.2: Overview of implementation of bidirectional path tracing

Since NVIDIA team provides the Falcor framework that caters to researchers with abstractions of scene geometry and material handling, low-level DXR APIs and GUIs, we decide to build our rendering framework on top of that to avoid repetitive work. With reference to Figure 4.2, the bidirectional path-tracing can be split into two main rendering passes, namely G-buffer and path integral formulation. G-buffer is used to extract scene information into textures so that subsequent rendering

passes can reuse the data. The path integral formulation evaluates the radiance of all pixels in the image.

4.2.2 G-buffer

G-buffer stores geometrical information of the scene for each pixel, including world position, world normal, view direction from the camera. It also contains material information such as diffuse color, specular color, roughness and type of material. It can have extra information such as index of refraction.

We can illustrate the implementation of shaders in a ray-traced G-buffer from the view of a pinhole camera with Algorithm 2. Note that we omit the miss shader in our pseudo-code because we can simply leave it as default.

Algorithm 2: Ray generation, closest-hit and any-hit shaders for G-buffer

Global: scene structure *scene*, camera settings including world position and coordinate frame axes, *camera*

Procedure `RayGenerationShader`**:**
- $curPixel \leftarrow DispatchRaysIndex().xy$
- $totalPixels \leftarrow DispatchRaysDimensions().xy$
- $pixelCentre \leftarrow (curPixel + (0.5, 0.5))/totalPixels$
- $ndc \leftarrow (2, -2) * pixelCenter + (-1, 1)$
- $rayDirection \leftarrow ndc.x * camera.U + ndc.y * camera.V + camera.W$
- $payload \leftarrow ShootRay(camera.position, rayDirection, 0, inf)$
- $outputTexture[curPixel] \leftarrow payload.attribute$

Procedure `ClosestHitShader`**:**
- Extract geometry and material information from the hit point.
- Store into payload.

Procedure `AnyHitShader`**:**
- Test if the hit point is transparent.
- **if** *transparent* **then**
  - Ignore hit
- **else**
  - Accept hit

4.2.3 Path construction

The path construction traces rays from both the camera and a sampled point on the randomly selected light source, and store essential information of all the hit

vertices. The path construction part is done in the ray generation shader, where we construct two sets of vertices, light paths and eye paths for each pixel. To do so, we first design a compact data structure $PathVertex$ to store the vertices. We need to have the following information of a vertex:

- Geometric information: it consists of the position P of the vertex in the world coordinate, the surface normal vector N and the normalized view direction vector V.
- Material information: it consists of the diffuse color dif, specular color *spec* of the material, and the roughness *rough* to simulate metallic material.
- Path information: it consists of a three-dimensional vector *radiance* that represents the radiance leaving the vertex, the probability distribution *pdf* of sampling the outgoing direction, and the type of BRDF *type* used for the vertex.

Since we have 6 three-dimensional vectors of floats, 2 floats and 1 unsigned integer, we can compact these attributes into an array of float4 of size 5, i.e. float4[5], and an unsigned integer. The total size of the data stored for one vertex is $5 \times 16 + 2 = 82$ bytes.

4.2.3.1 Eye path construction

To construct the eye path, we start tracing a ray from a sampled point on the camera, and store every hit point into the array of $PathVertex$ depending on the maximum eye path length S. Since we already have the primary hit information in G-buffer, we can simply extract it and start tracing the ray from the hit.

The first vertex v_1 of the path is a sampled point on the camera. For a pinhole camera, there is only one point to sample, which is the position of the camera itself. The normal vector is the z-axis of the camera coordinate frame. The potential of the camera is always 1 since the ray is calculated based on the pixel position and will always hit the image plane. The sampling probability is also 1 because we only have one choice of selecting the point on camera, and only one direction to sample for a particular pixel. We can keep the rest attributes as zero because they will not be used.

Algorithm 3: Sample BRDF

Input: Vertex v that contains geometric and material information
Output: Sampled direction L, sampling probability *pdf* and radiance *radiance* in the sampled direction

Procedure `SampleBRDF`**:**
  $radiance \leftarrow 0$
  $type \leftarrow SampleBRDFType(v.type)$
  $p \leftarrow GetProbabilityOfChosenType(type)$
  if $type = DIFFUSE$ **then**
    $L \leftarrow SampleHemisphere(v.N)$
    $pdf \leftarrow clamp(N \cdot L, 0, 1)/\pi$
    $radiance \leftarrow EvaluateDiffuse(v, L)/p$
  else if $type = REFLECTION$ **then**
    $L, pdf \leftarrow SampleReflectionLobe(v.N)$
    $radiance \leftarrow EvaluateReflection(v, L)/p$
  else if $type = REFRACTION$ **then**
    $L, pdf \leftarrow SampleRefractionLobe(v.N)$
    $radiance \leftarrow EvaluateRefraction(v, L)/p$
  return $L, pdf, radiance$

The second vertex v_2 of the path is loaded from G-buffer. We can get the geometric and material information. We will then sample the BRDF of the vertex to generate the direction L of the next ray segment, and compute the PDF of sampling. The radiance of the vertex in that particular direction is then evaluated. Algorithm 3 illustrates how to do so based on the material models such as lambertian diffuse models. Then we will shoot another ray starting from the vertex v_2 in the direction L, i,e. $ShootRay(v2.P, L, 0, inf)$. Then we can repeat the above to handle subsequent vertices hit.

With reference to Algorithm 4, we adopt an iterative ray tracing method to generate ray segments based on the current hit point. This allows us to transfer the BRDF samples from closest-hit shader to the ray generation shader via *RayPayload*. Note that payload here will contain all attributes of $PathVertex$ and three more fields: origin and direction of the next ray segment, and a boolean indicator if the ray should be terminated when it misses all scene geometry. Thus, we can compact them into an array of float4 of size 7, which requires $7 \times 16 = 112$ bytes for a payload.

Algorithm 4: Ray generation, closest-hit and miss shaders for eye path construction

Procedure `RayGenerationShader`**:**
  $curPixel \leftarrow DispatchRaysIndex().xy$
  initialize $PathVertex\ eyePath[MAX\_EYE\_PATH\_LENGTH]$
  $eyePath[0] \leftarrow ExtractCameraData(camera)$
  $primaryHit \leftarrow LoadFromGBuffer(curPixel)$
  $rayDirection, pdf, primaryHitRadiance \leftarrow SampleBRDF(primaryHit)$
  $rayOrigin \leftarrow primaryHit.P$
  for $i \leftarrow 2$ *to* $MAX\_EYE\_PATH\_LENGTH - 1$ **do**
    $payload \leftarrow ShootRay(rayOrigin, rayDirection, 0, inf)$
    if $payload.terminated = True$ **then**
      break
    $rayDirection \leftarrow payload.rayDirection$
    $rayOrigin \leftarrow payload.rayOrigin$
    $eyePath[i] \leftarrow ExtractDataFromPayload(payload)$

Procedure `ClosestHitShader`**:**
  $payload.rayOrigin \leftarrow v.P$
  $payload.rayDirection, payload.pdf, payload.radiance \leftarrow SampleBRDF(v)$

Procedure `MissShader`**:**
  $payload.terminated \leftarrow True$

4.2.3.2 Light path construction

The light path construction is similar to the eye path construction except that the first vertex is a sample on a randomly chosen light source, and we cannot make use of G-buffer for the primary hit. To get the first vertex, we can perform the following steps:

1. Randomly select a light source. Since in the Falcor framework, we have an array of light objects, we can generate a random number in [0, $N_l - 1$], where N_l is the number of lights. The sampling PDF is $\frac{1}{N_l}$

2. Sample the position on the chosen light and a direction of the light ray. The position, direction and the sampling PDF will depend on the light models such as point light, directional light and area light.

3. Store the position, sampling PDF, light intensity into a $PathVertex$.

Then, we can shoot rays iteratively to find the rest of the path vertices, similar to Algorithm 4.

4.2.4 Path integration

After constructing the two subpaths, we can evaluate the contribution of a particular path by handling the 4 cases mentioned in § 3.2.2. For case 1, we can simply add the emissive color loaded from G-buffer. For case 2, we will loop through all the vertices in the eye path, and do direct lighting computation on each vertex. Algorithm 5 illustrates how we handle this case by using an iterative approach to sum the weighted contributions from the eye path when the length of light path is 0.

Algorithm 5: Handle Case 2: Iterative way of path-tracing

Input: Eye path, $eyePath$, and the current pixel location, $curPixel$

Procedure `HandleCase2:`

$contribution \leftarrow 0$

for $i \leftarrow 1$ *to* MAX_DEPTH **do**

$unweightedContribution \leftarrow$
$eyePath[i-1].radiance * EvaluateDirectIllumination(eyePath[i])$

$contribution \leftarrow$
$contribution + GetWeight(i, 0) * unweightedContribution$

$outputTexture[curPixel] \leftarrow outputTexture[curPixel] + contribution$

Similarly, for case 3, we can use the same approach for the light path, except that we have to compute the pixel location given a direction to camera. Algorithm 6 shows how we handle case 3 with a pinhole camera model.

For case 4, we have to form a connecting edge between the end vertices from both the light subpath and eye subpath, and do a visibility test.

4.2.5 Multiple importance sampling

The multiple importance sampling finds the weight of a path parametrized by the eye path length s, and light path length t. We adopt the power heuristics as shown in Equation 10. We use Algorithm 8 to illustrate how the weight is computed making use of the *pdf* stored in a $PathVertex$.

Algorithm 6: Handle Case 3: Iterative way of light-tracing

Input: Light path, $lightPath$

Procedure `HandleCase3`**:**

$contribution \leftarrow 0$
for $i \leftarrow 1$ *to* MAX_DEPTH **do**
$directionToCamera \leftarrow normalize(camera.position - lightPath[i].P)$
$radiance \leftarrow EvaluateBRDF(lightPath[i], directionToCamera)$
$visible \leftarrow TestVisibilityToCamera(lightPath[i])$
if *visible* **then**
$pixelLocaition \leftarrow FindPixelLocation(directionToCamera)$
$contribution \leftarrow$
$contribution + GetWeight(0, i) * unweightedContribution$
$outputTexture[pixelLocation] \leftarrow$
$outputTexture[pixelLocation] + contribution$

Procedure `FindPixelLocation`**:**

$d1 \leftarrow dir \cdot camera.U$ / ||camera.U||
$d2 \leftarrow dir \cdot camera.V$ / ||camera.V||
$d3 \leftarrow dir \cdot camera.W$ / ||camera.W||
$ndc \leftarrow (d1/d3, -d2/d3)$
$pixelCenter \leftarrow ndc * (0.5, 0.5) + (0.5, 0.5)$
$pixelLocation \leftarrow round(pixelCenter * totalPixels)$
return $pixelLocation$

Algorithm 7: Handle Case 4: Forming connecting edge and test visibility

Input: Eye path, $eyePath$, light path $lightPath$, and the current pixel location, $curPixel$

Procedure `HandleCase4`**:**
- $contribution \leftarrow 0$
- **for** $i \leftarrow 2$ *to* MAX_DEPTH **do**
  - **for** $s \leftarrow 1$ *to* $MAX\_DEPTH - 1$ **do**
    - $t \leftarrow i - s$
    - $aL \leftarrow lightPath[t-1].radiance$
    - $aE \leftarrow eyePath[s-1].radiance$
    - $lightPathEnd \leftarrow lightPath[t]$
    - $eyePathEnd \leftarrow eyePath[s]$
    - $edgeDirection \leftarrow normalize(lightPathEnd.P - eyePathEnd.P)$
    - $fsL \leftarrow EvaluateBRDF(lightPathEnd, -edgeDirection)$
    - $fsE \leftarrow EvaluateBRDF(eyePathEnd, edgeDirection)$
    - $G \leftarrow EvaluateG(lightPathEnd, eyePathEnd)$
    - $unweightedContribution \leftarrow aL * aE * fsL * fsE * G$
    - $contribution \leftarrow contribution + GetWeight(s,t) * unweightedContribution$
    - $outputTexture[curPixel] \leftarrow outputTexture[curPixel] + contribution$

Procedure `EvaluateG`**:**
- $posA \leftarrow v1.P$
- $posB \leftarrow v2.P$
- $directionAB \leftarrow normalize(posB - posA)$
- $visible \leftarrow TestVisibility(posA, posB)$
- **if** *visible* **then**
  - $cosA \leftarrow |v1.N \cdot directionAB|$
  - $cosB \leftarrow |v2.N \cdot directionAB|$
  - $invLengthAB \leftarrow 1/length(posB - posA)$ **return** $cosA * cosB * invLengthAB * invLengthAB$
- **else**
  - **return** 0

Algorithm 8: Evaluate the weight using multiple importance sampling

Input: Eye path, $eyePath$, light path $lightPath$, eye path length s, light path length t

Procedure `GetWeight`**:**

$totalLength \leftarrow s + t$
$totalPdf \leftarrow 0$
for $i \leftarrow 0$ *to* $totalLength$ **do**
  $j \leftarrow totalLength - i$
  $pE \leftarrow eyePath[0].pdf$
  for $x \leftarrow 1$ *to* i **do**
    $pE \leftarrow pE * eyePath[x].pdf * EvaluateG(eyePath[x-1], eyePath[x])$
  $pL \leftarrow lightPath[0].pdf$
  for $y \leftarrow 1$ *to* j **do**
    $pL \leftarrow pL * lightPath[y].pdf * EvaluateG(lightPath[y-1], lightPath[y])$
  $totalPdf \leftarrow totalPdf + (pE * pL)^2$
  if $i = s$, $j = t$ **then**
    $currentPdf \leftarrow (pE * pL)^2$
return $currentPdf / totalPdf$

Chapter 5

Results

In this chapter, we will present rendering results in various scenarios. We will compare the our bidirectional path-tracing with the standard path-tracing with next event estimation (NEE). In the first section, we compare image quality using three different settings with the well-known Cornell Box [18]. The Cornell Box is a simple but powerful scene that allows us to experiment diffuse-diffuse, diffuse-specular and specular-specular interactions. In the second section, we compare the speed and memory complexity using the number of rays per pixel as an indicator. In Appendix A, we also present various complex scenes rendered to showcase the capability of our framework.

5.1 Image quality

The first criteria for comparison is the image quality. We will compare the noise level in the image without denoising, and compare the convergence rate when increasing number of samples per pixel. We will look at important features in images, such as caustics, color bleedings, reflections, ambient occlusions and hard shadows.

5.1.1 Output images without denoising

In the first experiment, we set all walls as diffuse so that we can see the interactions between diffuse surfaces. We also set two spheres as reflective in order to see mutual reflections. With reference to Figure 5.1, it is rendered by our bidirectional path-tracing. Compared to Figure 5.2, our result contains less noise on the walls and on the specular sphere. The ambient occlusions and shadows in our result are sharper,

which can be observed at the left side of the sphere and the ceiling in the reflection.

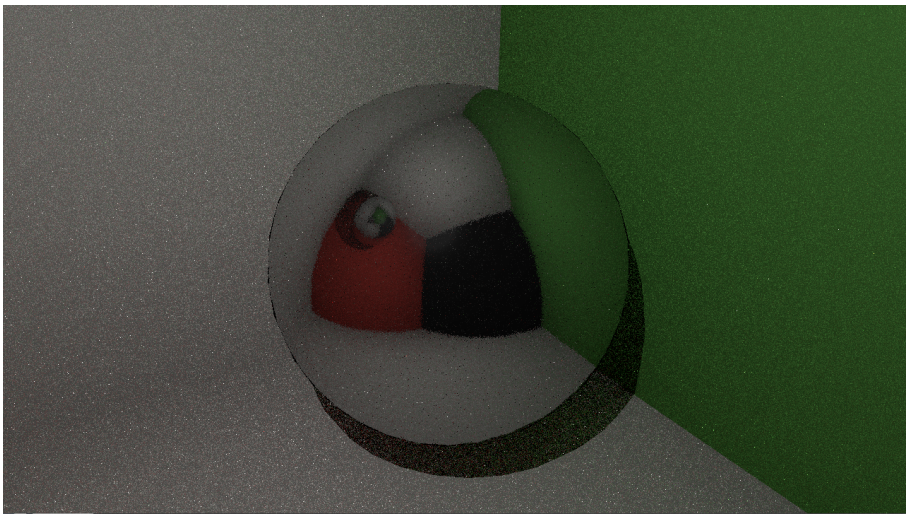

Figure 5.1: Bidirectional path-tracing: Cornell Box with two specular spheres

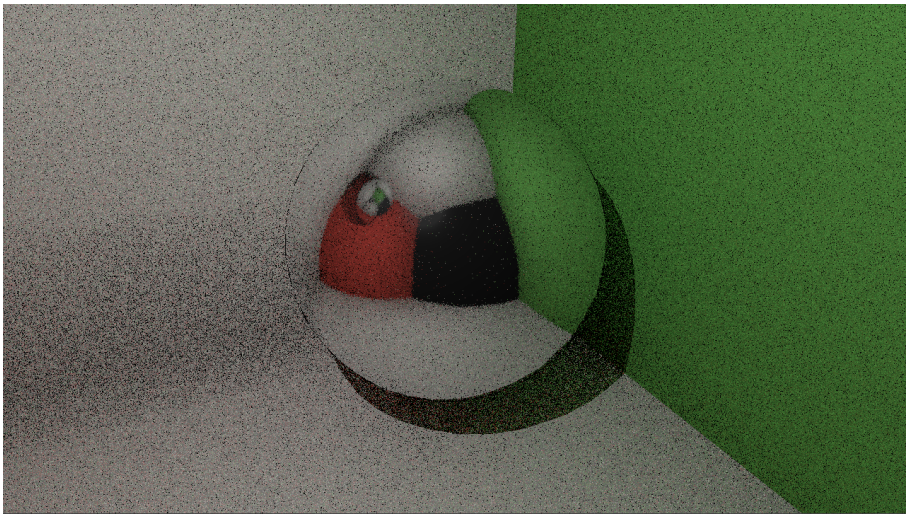

Figure 5.2: Standard path-tracing: Cornell Box with two specular spheres

Our second experiment has the back wall as specular and one specular sphere, because we want to investigate the noise level of a large flat specular surface, as well as the mutual reflections. In Figure 5.3, we can also observe that the noise level is lower than Figure 5.4 at both the specular and diffuse walls. The ambient occlusion at the sides of the back wall is also sharper.

In the third experiment, we simulate water reflections to produce caustics. Unlike the caustics by refraction through a glass ball, it is difficult to generate this effect in

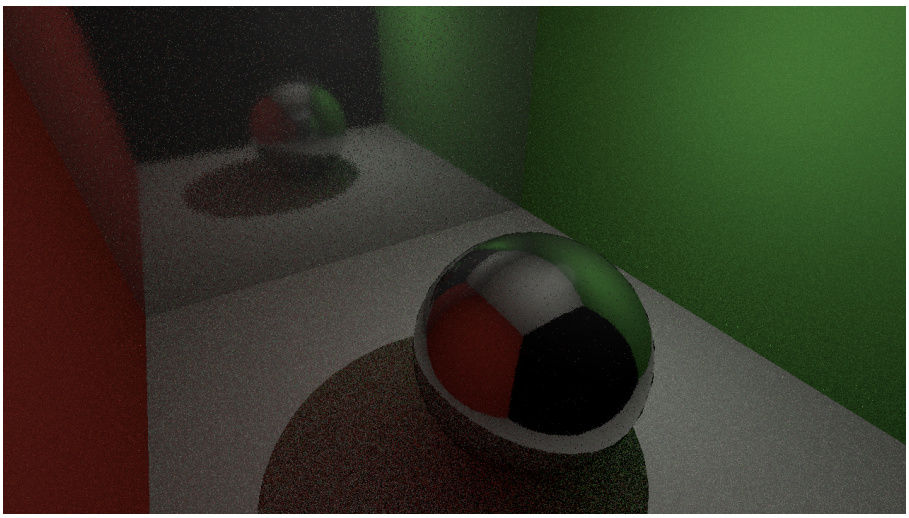

Figure 5.3: Bidirectional path-tracing: Cornell Box with one specular sphere and back wall

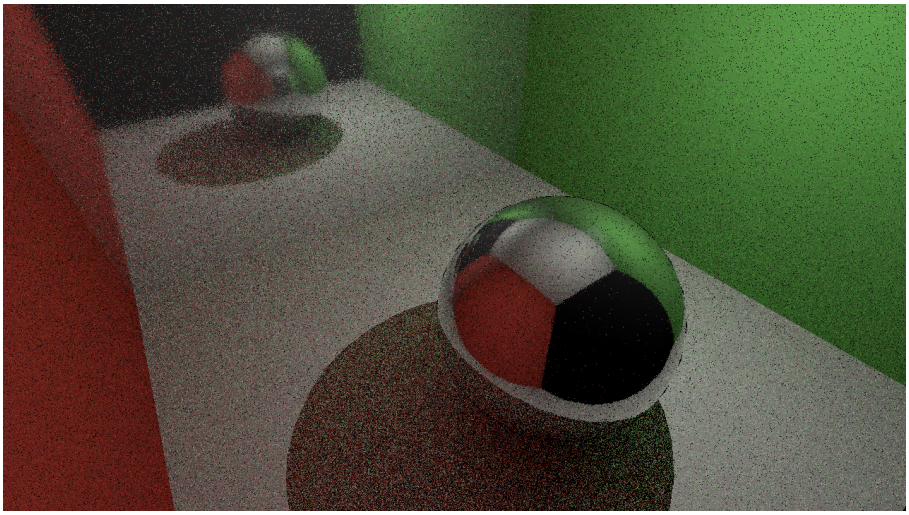

Figure 5.4: Standard path-tracing: Cornell Box with one specular sphere and back wall

the standard path-tracing algorithm. In Figure 5.5, the caustics on the back wall is obvious while we can hardly see any in Figure 5.6.

5.1.2 Images with temporal accumulation

The temporal accumulation pass takes the average of all the past frames. We can use this pass to denoise and control the number of samples per pixel. With the similar amount of rendering time, we can compare bidirectional path-tracing with 10

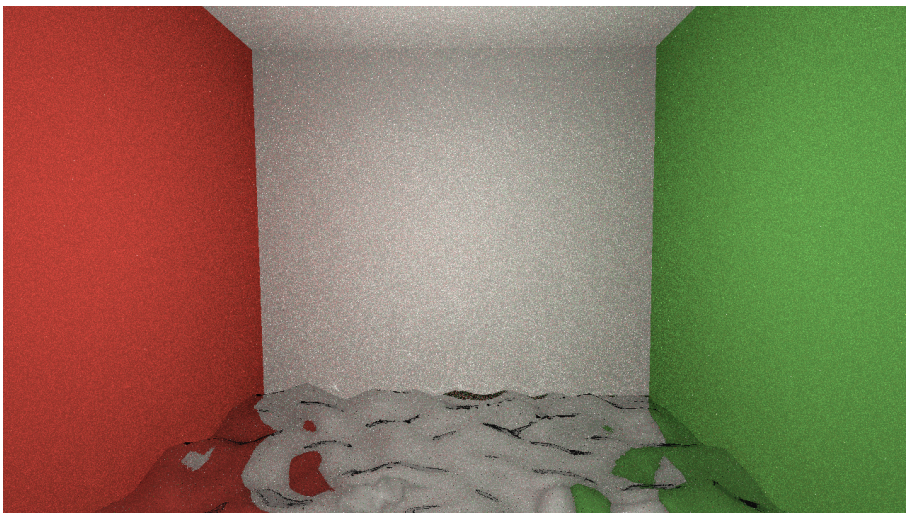

Figure 5.5: Bidirectional path-tracing: Cornell Box with reflective water surface

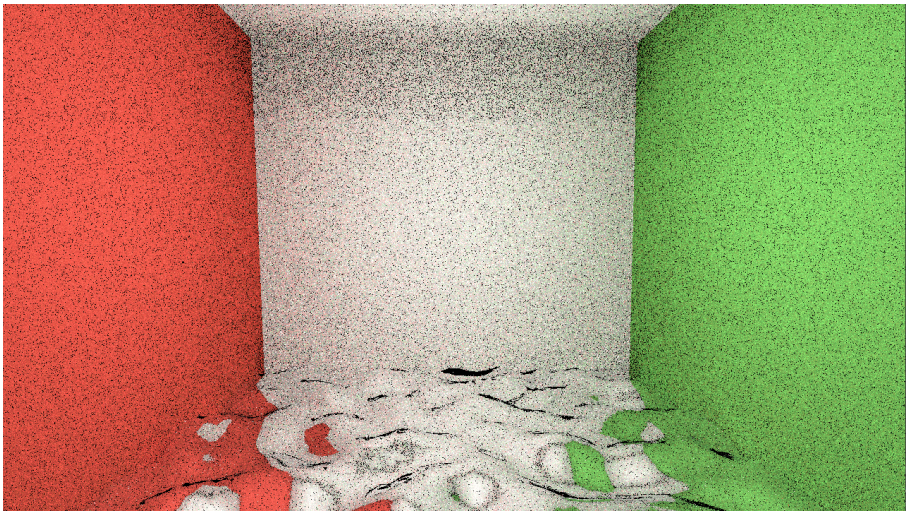

Figure 5.6: Standard path-tracing: Cornell Box with reflective water surface

frames accumulated and standard path-tracing with 40 frames accumulated. With reference to Figure 5.7 and Figure 5.8, the difference in the noise level is negligible as we can observe clearly every important features in the scene.

However, comparing Figure 5.9 and Figure 5.10, the caustics by water reflection are much clearer and sharper in our result, and we can only see a little of caustics on the right side of the wall in the image using path-tracing.

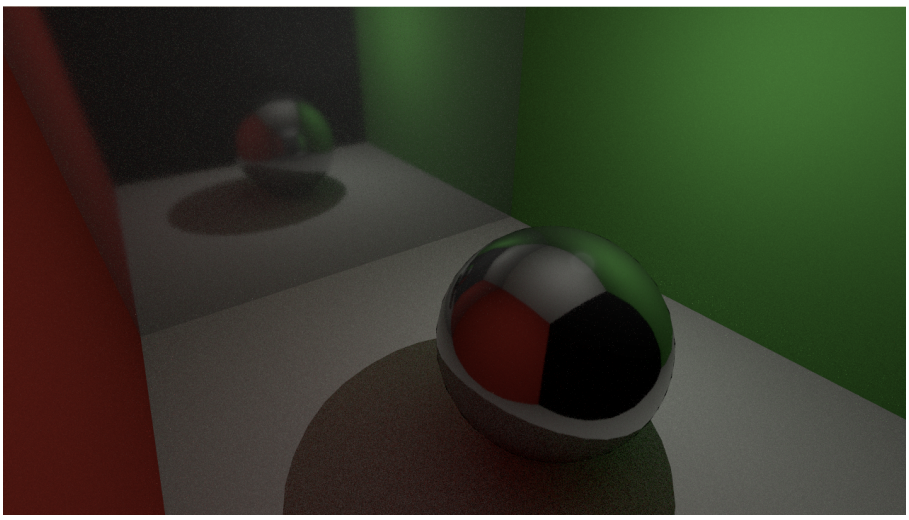

Figure 5.7: Bidirectional path-tracing with 10 frames accumulated: Cornell Box with one specular sphere and back wall

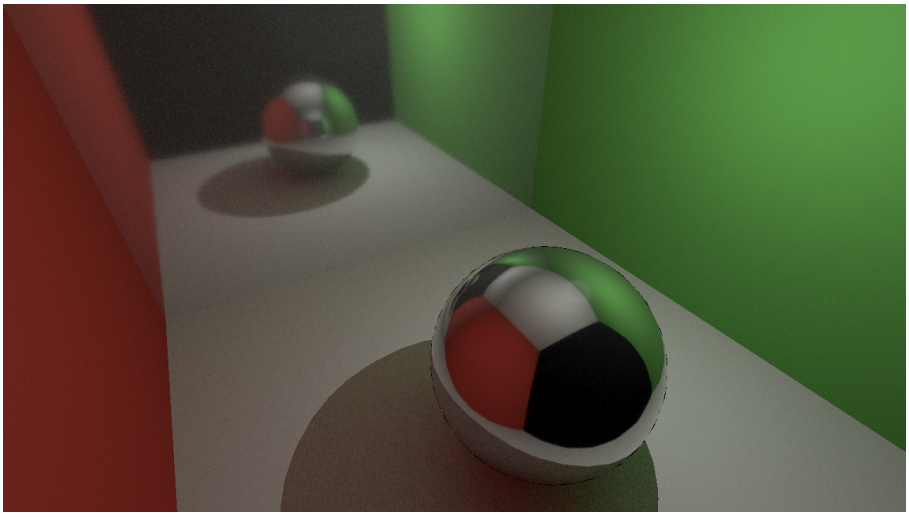

Figure 5.8: Standard path-tracing with 40 frames accumulated: Cornell Box with one specular sphere and back wall

5.1.3 Denoising by BMFR

After the denoising pass making use of BMFR, we can compare the denoised and raw images. With reference to Figure 5.11, Figure 5.12 and Figure 5.13, the left half of the images are denoised and the right half are not. We can see that the denoiser is able to maintain important features such as caustics, color bleedings, ambient occlusions, hard shadows and mutual reflections. However, we can still see

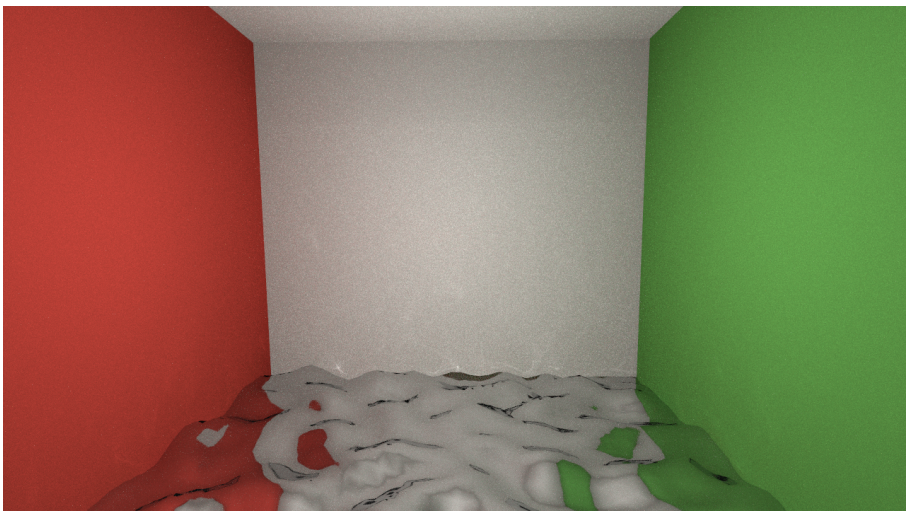

Figure 5.9: Bidirectional path-tracing with 10 frames accumulated: Cornell Box with reflective water surface

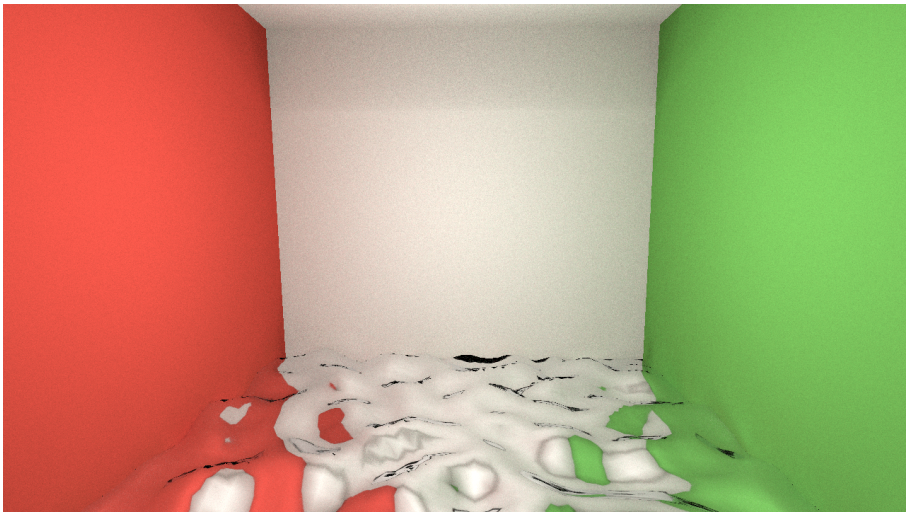

Figure 5.10: Standard path-tracing with 40 frames accumulated: Cornell Box with reflective water surface

a bit of jaggies at the edge of walls.

5.2 Speed and memory complexity

In this section, we will compare the speed and memory complexity between bidirectional path-tracing and standard path-tracing.

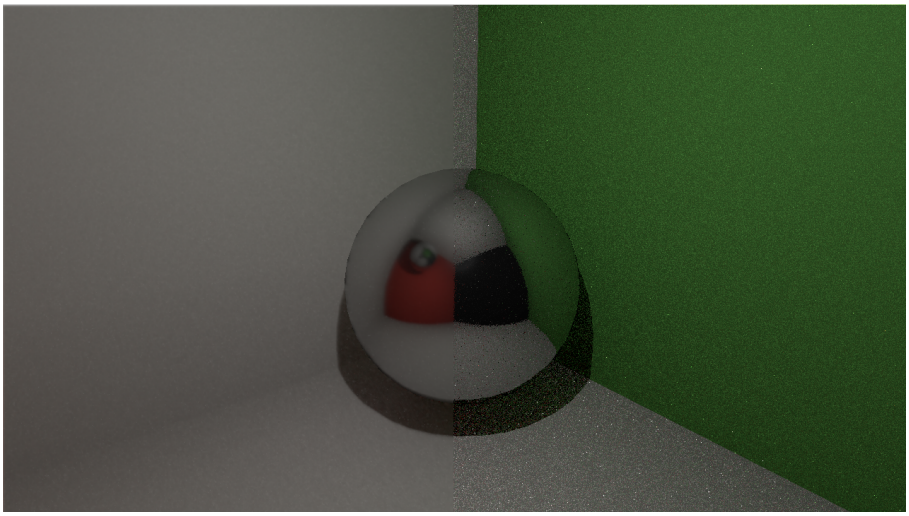

Figure 5.11: Cornell Box with two specular spheres: denoised (left half) and raw (right half)

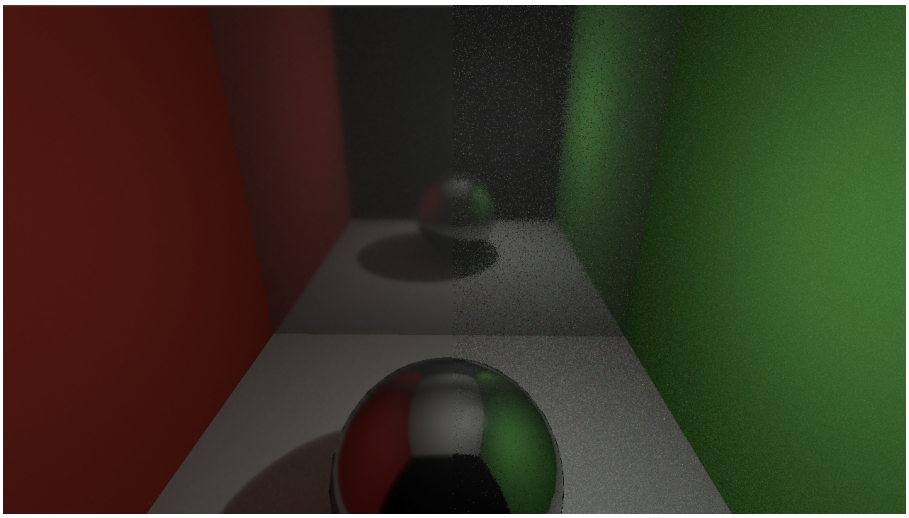

Figure 5.12: Cornell Box with one specular sphere and back wall: denoised (left half) and raw (right half)

5.2.1 Speed

We will evaluate the speed complexity by the number of rays generated per pixel because the testing of ray-geometry intersection is still the most expensive operation in ray-tracing even with the hardware acceleration.

In the standard path-tracing, one primary ray is generated per pixel. Once the ray hits a surface point, it bounces off by generating another ray, until the ray misses

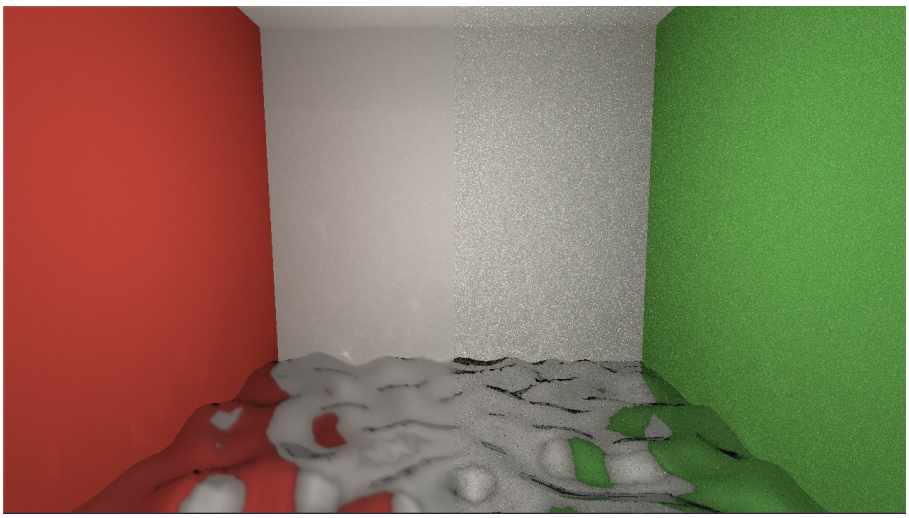

Figure 5.13: Cornell Box with reflective water surface: denoised (left half) and raw (right half)

all geometry or reach the maximum depth limit D. For each hit point, a shadow ray is shoot towards a random light source for direct illumination. This is required in the next event estimation to bring down the noise. Therefore, the total number of rays needed for one pixel is $2D$.

On the other hand, the bidirectional path-tracing requires $2D$ rays per pixel in the path construction step because light subpath and eye subpath spend D rays each. For testing visibility between end points, we need to generate $2 \times (1 + 2 + 3 + ... + D) = D + D^2$ rays. Therefore, the total number of rays needed for one pixel is $2D + D + D^2 = D^2 + 3D$.

Thus, we can see that the complexity of bidirectional path-tracing is quadratic and that of path-tracing is linear. The rendering time ratio is:

$$\frac{T_{bpt}}{T_{spt}} = \frac{D^2 + 3D}{2D} = 0.5D + 1.5$$

Since we use $D = 4$ in our experiments, the ratio is 3.5 and that means bidirectional path-tracing takes 2.5 times longer rendering time than path-tracing. That is also why we compare bidirectional path-tracing with 10 accumulated frames with standard path-tracing with 40 accumulated frames.

To render images with resolution of 1280×720, with Intel Core i7-9700K CPU @ 3.60GHz and ZOTAC Gaming GeForce RTX 2070 Super 8GB RAM, we are able

to achieve above 30 FPS for bidirectional path-tracing. With standard path-tracing, the FPS is beyond 120.

5.2.2 Memory

We will evaluate the memory complexity by the intermediate results stored in the GPUs. Since both bidirectional path-tracing and standard path-tracing uses G-buffer, the memory complexity is linearly proportional to the number of textures N_t for storing pixel information and the total number of pixels N_p. Since each texture is a two-dimensional array of float4, it takes $16N_p$ bytes per texture, and $16N_pN_t$ bytes in total. For bidirectional path-tracing, we need some additional space to store path vertices for each pixel, because ray generation shaders are executed in parallel. As discussed in § 4.2.3, we need 82 bytes for each path vertex, and the total additional space is $82 \times 2 \times D \times N_p = 164DN_p$ where D is the maximum path length. In our experiments, the G-buffer uses 8 textures, and the total number of pixel is $1280 \times 720 = 921600$. We can convert 921600 bytes to 0.88 Megabytes. The total memory usage for bidirectional path-tracing is $(16 \times 8 + 82 \times 2 \times 4) \times 0.88 = 692$ Megabytes while that for standard path-tracing is $16 \times 8 \times 0.88 = 113$ Megabytes. In both algorithms, the memory usage is acceptable given the current GPU RAMs.

5.3 Summary

After evaluating image quality, speed and memory complexity, we can see that bidirectional path-tracing is able to produce images with low level of noise, and after denoising, the important features remain clear and sharp. Although it is much more computationally expensive than standard path-tracing, we are still able to achieve above 30 FPS, which is in real-time. With the continuous advancement in GPUs, we believe that the FPS can be further increased to 60 and above.

Chapter 6

Discussion

In the previous chapters, we have presented the theory needed to implement a bidirectional path-tracer, our implementation details as well as the rendered results. There is, however, still some improvement that can be done. We will provide several possible solutions for that. We will also compare the bidirectional path-tracing with other rendering techniques to evaluate the advantages and limitations inherently existing in the algorithm itself.

6.1 Possible improvement in our implementation

The speed of our implementation can be improved by reducing the amount of visibility tests between the end vertices of subpaths. One way is to use group visibility testing described by Dr. Eric Veach [27], and another way is to use the probabilistic connections [23]. We did not focus on the optimization of bidirectional path-tracing because our emphasis is on the exploration and testing with the DXR. It is also possible to optimize the testing of ray-geometry intersection from the shaders by reducing the payload size carried along with each ray. This reduces the overhead in the hardware. These two areas of improvement can significantly increase the speed because these are usually the bottlenecks.

The accuracy of our implementation can also be improved by using the Russian Roulette method to randomly terminate a ray instead of setting a limit on the maximum ray depth. Our current implementation might introduce certain bias depending on the depth limit. For example, in Figure 5.1, it is supposed to have infinite reflections of spheres, but due to the depth limit, the reflection will end at certain depth. However, such bias is almost negligible given a reasonable depth limit.

Using the Russian Roulette will lead to additional time complexity, and thus it is a trade-off between accuracy and speed.

In our implementation, we only uses a few camera, light and material models. For cameras, we consider a pinhole camera and a thin lens camera to possibly simulate depth-of-field effects. It is possible to consider a prospective camera. For lights, we have a point light and a directional light. It will be great to add an area light and a spot light. For materials, we have lambertian diffuse and GGX [28] microfacet models for specular reflection and refraction. It is also good to have more material models to enhance the capability of rendering photo-realistic images. In our experiments, we presented three different settings to investigate the noise level and denoising quality of diffuse-diffuse, diffuse-specular and specular-specular interactions. With more camera, light and material models, we are able to come up with a great variety of scenes to test the robustness of our framework.

There is a problem in our implementation that we did not consider paths with no vertex on either light subpath or eye subpath because we do not have intersect-able geometry for lights and camera lens in our scenes. Considering such paths will only add waste to computational resources because it is impossible for a ray to reach such light or camera as the end point. However, if the light sources and cameras can be intersected by rays, and have significantly size in the scene, we have to consider those paths, which will increase the time complexity in the path construction stage.

In our experiments, it will be great if we can adopt statistical methods to analyze the amount of noise with respect to a reference image. For example, we can calculate the variance of colors on a surface, and compare the mean square distance of the pixel colors. Statistics will provide us good numerical indicators of noise.

6.2 Comparison with other rendering techniques

If we compare bidirectional path tracing with standard path tracing, we saw in Chapter 5 that bidirectional path tracing performs a lot better in terms of low noise level. This is especially in the scenes where there is little direct lighting and in the scenes containing caustics. However, bidirectional path tracing has some cases that it could perform poorly. For instance, it is not very efficient for outdoor scenes because shooting rays from the sun will lead to a very small probability of hitting

the scene. Another example is that we have a building with many rooms and each room has its own light source. If the camera is placed in one of the rooms and the doors between the rooms are closed, then bidirectional path tracing will need a large amount of time constructing the light subpath, which will hardly be connected to the eye subpath. If one of the doors is open, the probability might increase but still remains low.

Another well-known method is photon mapping, as introduced in Chapter 2. The benefit of photon mapping is that it requires a small number of rays and it can produce noise-free images. However, it is biased towards how many photons are used. If we use a small number of photons, it will create artifacts no matter how long we render it. Bidirectional path-tracing, on the other hand, is unbiased. If we use the Russian Roulette method, it is guaranteed not to have any artifacts after sufficient rendering time. In addition, the time and space complexity of photon mapping is highly dependent on the data structure used to store photons. If we use a large amount of photons to avoid artifacts, the construction, node insertion and searching time needed in a K-dimensional tree will increase significantly. Since we do not have hardware acceleration for kd-tree operations, the impact of RTX GPUs on photon mapping is limited.

Another rendering technique called metropolis light transport [27] is better at handling scenes that is lit through small openings such as doors and windows. This method is a popular extension to bidirectional path tracing to handle a wider range of scenes. Another possible extension is the vertex connection and merging [10] which makes use of both bidirectional path-tracing and photon mapping. These rendering techniques could be possible directions to go, but we still need to explore the performance of them on RTX GPUs.

Chapter 7

Future Work and Conclusion

7.1 Future research directions

We will provide several future research directions which we have considered but yet have time to do so. The most straightforward direction is to continue improving and optimizing the current framework as discussed in Chapter 6, so that it can be more efficient and capable of handling a wide range of scenes. The second direction would be focusing on realizing physically correct motion blur, which is mostly produced as a post-processing effect in current production work. Making the motion blur physically plausible will enhance the immersive experience of users in games, films and virtual environments. Another possible direction is to develop a lightweight denoiser specifically designed to handle the noise level in the current framework. Last but not least, we can add other rendering techniques such as metropolis light transport and vertex connection and merging into the framework.

7.2 Conclusion

In this dissertation, we have presented a robust and noise-free rendering framework that can generate caustics and global illumination in real-time. Although the current rendering speed is only above 30 frames per second, we believe that after further improvements and optimizations, it can have a great performance. With the increasing trend of GPU computational power, we can foresee the application of our work because of the fast convergence rate with an increasing number of samples per pixel. Our rendering frame can also be further extended to various techniques based on the path integral formulation of light transport.

Bibliography

[1] B. Abdollah-shamshir-saz, "Voxel based hybrid path tracing with spatial denoising", in *In Proceedings of I3D Symposium, Montreal, Quebec, Canada, 15-18 May 2018, 2 pages.*, 2018.

[2] P. Andersson, J. Nilsson, M. Salvi, J. Spjut, and T. Akenine-Möller, "Temporally dense ray tracing", in *High Performance Graphics*, ACM, 2019.

[3] N. Benty, K.-H. Yao, P. Clarberg, L. Chen, S. Kallweit, T. Foley, M. Oakes, C. Lavelle, and C. Wyman, "The Falcor rendering framework", `https://github.com/NVIDIAGameWorks/Falcor`, Mar. 2020. [Online]. Available: `https://github.com/NVIDIAGameWorks/Falcor`.

[4] A. Benyoub, "Leveraging ray tracing hardware acceleration in unity", in *Digital Dragons*, 2019.

[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", in *ACM Trans. Graph. 36, 4, Article 98*, ACM, 2017.

[6] C. D. Christoph Schied Christoph Peters, "Gradient estimation for real-time adaptive temporal filtering", in *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, ACM, 2018.

[7] M. F. Christopher Kulla, "Importance sampling techniques for path tracing in participating media", in *Computer Graphics Forum, Volume 31 Issue 4*, The Eurographs Association & John Wiley & Sons, Ltd. Chichester, UK, 2012, pp. 1519–1528.

[8] H. Dammertz, D. Sewtz, J. Hanika, and H. P. Lensch, "Edge-avoiding à-trous wavelet transform for fast global illumination filtering", in *High Performance Graphics*, ACM, 2010.

[9] A. Frühstück and S. Prast, "Caustics , light shafts , god rays",, 2011.

[10] I. Georgiev, J. Křivánek, T. Davidovič, and P. Slusallek, "Light transport simulation with vertex connection and merging", *ACM Trans. Graph.*, vol. 31, no. 6, 192:1–192:10, Nov. 2012, ISSN: 0730-0301. [Online]. Available: `http://doi.acm.org/10.1145/2366145.2366211`.

[11] P. S. Heckbert, "Adaptive radiosity textures for bidirectional ray tracing", in *ACM SIGGRAPH Computer Graphics*, 1990.

[12] P. H. C. Henrik Wann Jensen, "Efficient simulation of light transport in scences with participating media using photon maps", in *SIGGRAPH '98, ACM Press, New York, New York, USA*, 1998, pp. 311–320.

[13] W. Jarosz, A. Enayet, A. Kensler, C. Kilpatrick, and P. Christensen, "Orthogonal array sampling for monte carlo rendering", in *Computer Graphics Forum (Proceedings of EGSR)*, 2019.

[14] J. S. Kaiming He and X. Tang, "Guided image filtering", in *IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE*, IEEE, 2013.

[15] J. T. Kajiya, "The rendering equation", in *SIGGRAPH '86, ACM, New York, NY, USA*, 1986, pp. 143–150.

[16] M. Mara, M. McGuire, B. Bitterli, and W. Jarosz, "An efficient denoising algorithm for global illumination", in *High Performance Graphics*, ACM, 2017.

[17] K. I. Matias Koskela, "Blockwise multi-order feature regression for real-time path tracing reconstruction", in *Transactions on Graphics (TOG)*, ACM, 2019.

[18] M. McGuire, "Computer graphics archive", `https://casual-effects.com/data`, Jul. 2017. [Online]. Available: `https://casual-effects.com/data`.

[19] B. B. Michael Mara Morgan McGuire and W. Jarosz, "An efficient denoising algorithm for global illumination", in *High Performance Graphics*, ACM, 2017.

[20] P. S. Nima Khademi Kalantari Steve Bako, "A machine learning approach for filtering monte carlo noise", in *ACM Transactions on Graphics (TOG)*, ACM, 2015.

[21] C. Schied, A. Kaplanyan, C. Wyman, A. Patney, C. R. A. Chaitanya, J. Burgess, S. Liu, C. Dachsbacher, A. Lefohn, and M. Salvi, "Spatiotemporal variance-guided filtering", in *High Performance Graphics*, ACM, 2017.

[22] K. Shkurko, C. Yuksel, D. Kopta, I. Mallett, and E. Brunvand, "Time interval ray tracing for motion blur", in *IEEE Transactions on Visualization and Computer Graphics, Volume 24 Issue 12*, IEEE, 2017, pp. 3225–3238.

[23] P. Stefan, R. Ravi, D. Fredo, and D. George, "Probabilistic Connections for Bidirectional Path Tracing", *Computer Graphics Forum*, 2015.

[24] H. W. J. Toshiya Hachisuka, "Stochastic progressive photon mapping", in *SIGGRAPH Asia '09 ACM SIGGRAPH Asia 2009 papers*, ACM New York, NY, USA, 2009.

[25] H. W. J. Toshiya Hachisuka Shinji Ogaki, "Progressive photon mapping", in *SIGGRAPH Asia '08 ACM SIGGRAPH Asia 2008 papers*, ACM New York, NY, USA, 2008.

[26] Unity, "Unity manual", `http://docs.unity3d.com/Manual/index.html`, 2019. [Online]. Available: `http://docs.unity3d.com/Manual/index.html`.

[27] E. Veach, "Robust monte carlo methods for light transport simulation",, 1998.

[28] B. Walter, S. Marschner, H. Li, and K. E. Torrance, "Microfacet models for refraction through rough surfaces", in *Rendering Techniques*, 2007.

[29] C. Wyman, S. Hargreaves, P. Shirley, and C. Barr-Brisebois, "Introduction to directx raytracing", `http://intro-to-dxr.cwyman.org/`, 2018. [Online]. Available: `http://intro-to-dxr.cwyman.org/`.

Appendix A

Image Gallery

The models of the following scenes are taken from McGuire Computer Graphics Archive at `https://casual-effects.com/data/`. We render the following scenes in our framework, with 1000 accumulated frames and maximum ray depth of 4. All rendered within 1 minute.

Figure A.1: Arcade

Figure A.2: Bedroom

Figure A.3: Conference

Figure A.4: Pink room

Figure A.5: Salle De Bain

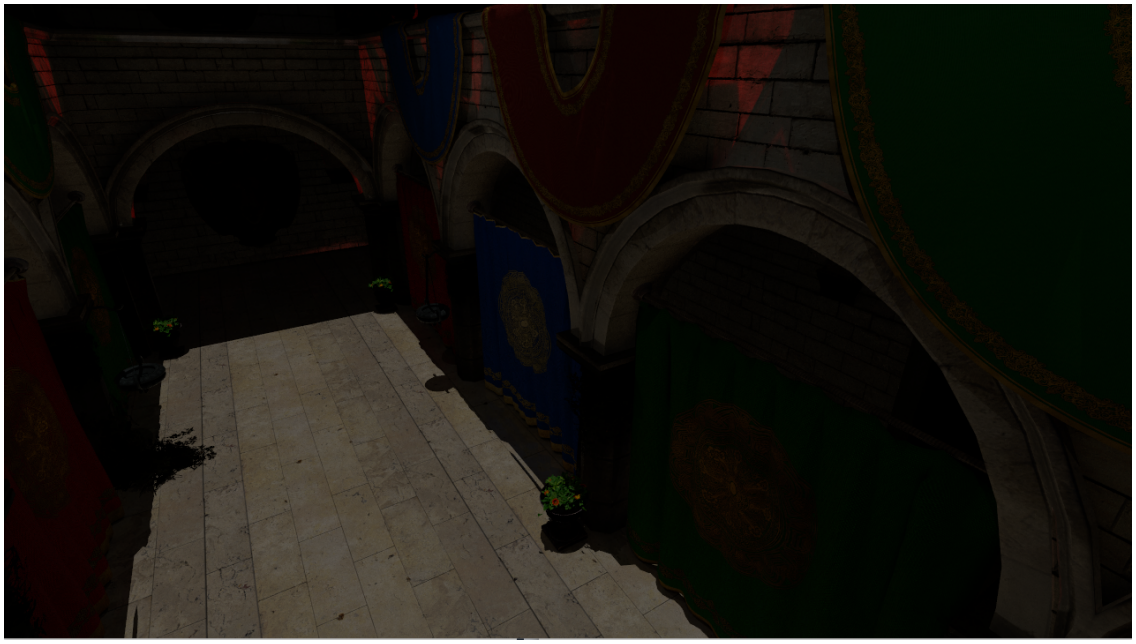

Figure A.6: Sponza