Efficient Photorealistic Rendering via Ray-Tracing on GPUs

Candidate Number: 1023011

University of Oxford

Computer Science 4th Year Project Report

Supervised by: Joe Pitt-Francis

Trinity 2020

Abstract

One of the ultimate goals of the field of computer graphics is to synthesize images that are indistinguishable from photographs. This task, often referred to as photorealistic rendering, is known for being extremely costly, due to the need of a physically-correct simulation of the transport of light in the scene. This project explores methods to accelerate rendering by utilizing the massively parallel computing architectures of GPUs, and creates a renderer whose performance tops some of the best renderers in the academia.

This project focuses on the widely-used simulation algorithm known as path-tracing. The project studies how to maximize the performance of path-tracing on GPUs, and implements state-of-the-art algorithms for the construction and traversal of the data structures used by this algorithm. Besides the standard path-tracing algorithm, the project also implements a newly emerged variant, where reinforcement learning is used to guide the the selection of light paths. The resulting renderer can robustly handle a variety of real-world materials, complex scene geometry, and difficult lighting situations. Moreover, comprehensive benchmarking indicates that the efficiency of this GPU renderer significantly exceeds other CPU implementations, and is even noticeably faster compared to one of the most academically renowned GPU renderer.

Contents

1 Introduction

1.1 Motivation

May it be glorious space battles with lasers blasting around, or a mountain-high gorilla wrestling a even bigger lizard, modern rendering technologies present our craziest fantasies to our eyes as if they are happening right in front of us. The contemporary film and television industry is highly dependent on this ability of rendering photorealistic images, and alongside the pursuit of wilder stories and imaginations, the scenes to be rendered are becoming increasingly complex and arbitrary.

Despite its importance, the task of photorealistic rendering has a computational cost that is matched by few others. As an example, recent Disney title films could require hundreds of CPU hours just to synthesize a single frame[3]. This immense cost is explained by the fact that there are complex interactions between light rays and scene geometry, and the rendering algorithm must comprehensively and accurately simulate these interactions in order to obtain a realistic image.

This project studies two orthogonal approaches that both boost the efficiency of rendering. Firstly, this project utilizes GPUs, whose massively parallel architectures allow a large amount of pixels to be processed at the same time. Secondly, from an algorithmic perspective, the project explores how reinforcement learning can be used to guide the algorithm to focus on the "important" parts of the scene. The project shows that both methods are indeed effective ways of improving rendering efficiency.

1.2 Related Work

The study of computer graphics began almost as soon as computers with screens are manufactured. In 1980, Whitted[17] proposed the algorithm known as *ray-tracing*, which is the algorithmic foundation of almost all photorealistic rendering algorithms used today. However, the version of ray-tracing described by Whitted was not based on a physically plausible model of light, and consequently the images generated was not yet photorealistic.

The age of photorealism began in 1986 with the visionary work of Kaijiya[8]. This paper introduced an integral equation known as *the rendering equation*, which models the power carried by light rays in a physically correct manner. Programs that produce images by computing solutions to this equation are called physically based renderers, and this method have since become the focus of decades of rendering research. In addition, Kaijiya's paper also described a variant of the ray-tracing algorithm known as *path tracing*, which solves the rendering equation using the technique of Monte-Carlo integration. This algorithm is still the core of most of the photorealistic renderers developed and used today, including the renderer implemented in this project.

Path-tracing is a powerful algorithm, but it is too costly[1] to be used for real-time rendering applications such as video games. As a result, real-time applications take a fundamentally different approach to rendering known as rasterization. Modern rasterization-based rendering pipelines can be very-fast: they can render tens or even hundreds of frames per second. However, despite the fact that a range of algorithms[6][15][7] exist that aims to make rasterization based algorithms as physically-correct as possible, the images produced by these renderers are still easily recognizable as computer generated. Thus, when the need for image quality dominates that for rendering speed, path-tracing is still the better choice. Notice that however, modern GPUs are designed to optimize for rasterization-based pipelines[2] and not for ray-tracing algorithms.

In a ray tracing algorithm, one of the most costly operations is ray-scene intersection. Intuitively, this is the task of finding the first intersection point between a light ray and the geometries in the scene. A naive linear search across all geometries is of course unacceptably slow, and a range of accelerating data structures have been created. This project employs the data structure of Bounding Volume Hierarchies (BVH), which recursively divides the scene into a tree of axis-aligned bounding boxes. Due to their recursive nature, the construction and traversal of these structures are difficult tasks on GPUs. For this reason, this project studied and implemented a series of GPU BVH techniques published by NVIDIA[9][10][1].

Ever since the introduction of path-tracing, a variety of algorithms that builds on top of patch-tracing have emerged. The most famous ones include bi-direction path-tracing[16], Metropolis light transport[16][11], energy redistribution path-tracing[4], and gradient-domain path-tracing [12]. One particularly interesting variant, which

[1] until perhaps, very recently. See NVIDIA RTX.
[2] this is beginning to change with NVIDIA RTX

uses reinforcement learning to guide the generation of light rays[5], is studied and implemented in this project.

One renderer of particular value to the graphics community is the `pbrt` renderer. Not only is this renderer open-source, it is also accompanied by an entire book[14] which happens to be the most authoritative textbook of physically based rendering. Moreover, because of its popularity, there is a large collection of beautiful scenes defined using `pbrt`'s input file format. For these reasons, this project decided to support `pbrt`'s scene definition files, which allows this project to be benchmarked and compared against `pbrt`.

Most of the ray-tracing renderers used tody, including `pbrt`, runs on the CPU. However, there is a GPU ray-tracing system known as OptiX[13], which is becoming increasingly popular. Developed by the GPU manufacturer NVIDIA, OptiX is heavily optimized, and provides a friendly ray-tracing library for developers. One famous renderer which uses OptiX as a backbone[3] is the `Mitsuba` renderer. The GPU rendering routines of this project is implemented from scratch, and its performance will be compared against `Mitsuba` (and therefore OptiX).

1.3 Project Outline

This project focuses on the GPU parallelization of ray-tracing algorithms for the efficient rendering of photorealistic images. A fully-featured GPU renderer is created, which supports a wide range of geometries, materials, and light sources. At the heart of the renderer are efficient implementations of two algorithms: the original path-tracing algorithm, and a variant of path-tracing guided by reinforcement learning.

In order to efficiently implement path-tracing, this project implemented state-of-the-art methods of performing ray-scene intersection detection. These includes algorithms for constructing BVH trees in parallel, optimizing the structure of constructed BVHs, and recursion-free methods of tree traversal. The project also makes numerous optimizations that reduces control flow divergences and memory access latencies, so that the resulting implementation maximally utilize the parallel architecture of GPUs.

This project made the observation that in certain lighting scenarios, the original path-tracing algorithm struggled to converge to a noiseless solution. To alleviate this problem, the project implements a variant of path-tracing, where reinforcement

[3]There is also an upcoming 4th version of `pbrt`, which also supports GPU via OptiX. However, the beta version is still unstable at the time this thesis is written

learning is used to guide the selection of light rays. The project investigates how the architecture of the GPU interacts with various aspects of the reinforcement learning routine, and implements a version that allows tens of thousands of GPU threads to efficiently cooperate during learning.

The renderer created by this project is named *Cavalry*[4]. It is created mainly using the C++ programing language on the CPU side, and the CUDA language for GPU computing. With a total of over 7 thousand lines, the complete source code of the software can be found at `https://github.com/AmesingFlank/Cavalry`, along with some additional information and screenshots.

[4]The name is inspired by a character in a video game called Overwatch. The character goes by the name "Tracer", and refers to herself as the "Cavalry".

2 Physically Based Rendering

In order to generate photorealistic images, renderers must find accurate solutions to the *rendering equation*. This chapter explains the meaning and intuition behind the equation, and describes how it can be solved by the path-tracing algorithm.

2.1 The Rendering Equation

The rendering equation describes the "strength" of light at each point in space and in each direction. To formalize the notion of strength, this section begins by introducing a few concepts from the study of radiometry.

2.1.1 Radiometry

In radiometry, there is a hierarchy of quantities that measures the strength of light in different contexts. The first quantity is *radiant flux*, which measures the total energy that travels through a region of space per unit time in the form of electromagnetic radiation. Radiant flux is often denoted by Φ, and its unit of measure is Watts(W).

In almost all regions in any scene, the radiant flux is not uniformly distributed, and thus one important quantity is the density of radiant flux with respect to area. This quantity is named *irradiance*, which is denoted by E, and measured in power per unit area ($W \cdot m^{-2}$). Intuitively, irradiance measures the amount of light received by a single point in space. For this reason, for a region of space S, integrating the irradiance of each point in S gives the total flux through S:

$$\Phi(S) = \int_S E(p)dp \tag{2.1}$$

For any point p, the irradiance $E(p)$ is not uniform across all directions, and thus it is also important to consider the density of irradiance in each direction ω. This quantity, $L(p, \omega)$, is called *radiance*, and its unit of measure for radiance is power per unit area per unit solid angle ($\mathrm{W} \cdot \mathrm{m}^{-2} \cdot \mathrm{sr}^{-1}$). Radiance is an especially important quantity, because it is a measure of the strength of a single ray of light, identified by

its direction ω and a point p which it passes through. Consequently, radiance is the physical quantity that ray-tracing algorithms constantly operates in.

In rendering, radiance often appears in the form $L_o(p, \omega)$ or $L_i(p, \omega)$, which mean radiance going out from the point p or entering into it, respectively. More precisely, $L_o(p, \omega)$ represents the radiance that travels from p and outwards in the direction ω, and $L_i(p, \omega)$ represents the incoming radiance that travels towards to p in the direction $-\omega$. The convention $L_i(p, \omega)$ might appear slightly counter-intuitive, since ω points in the opposite direction as the propagation of energy. However, the convenience of this notation will become apparent when the ray-tracing algorithm is formulated.

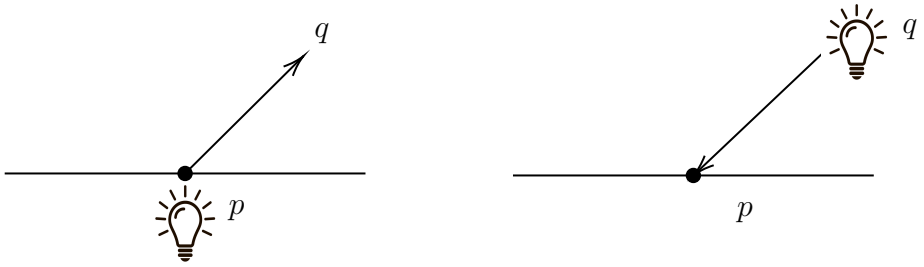


Figure 2.1: Example: consider two points p, q, and $\omega = \frac{q-p}{|q-p|}$. On the left, the radiance sent from p to q is $L_o(p, \omega)$. On the right, the radiance received by p from q is $L_i(p, \omega)$.

Similar to equation 2.1, which expresses flux as an integral of irradiance, it's also possible to obtain the incoming irradiance $E(p)$ by an integral across the incoming radiance from each direction. More precisely, the following relationship holds:

$$E(p) = \int_{\Omega} L_i(p, \omega_i) \cos\theta_i d\omega_i \tag{2.2}$$

Here, the support Ω is often a sphere or hemisphere of possible incoming direction, and the angle θ_i represents the angle between ω_i and the surface normal. The cosine term accounts for the fact that for incoming rays that are not perfectly perpendicular to the surface, the differential area illuminated by the ray is multiplied by a factor $\frac{1}{\cos\theta_i}$, and thus the contribution per unit area should be multiplied by $\cos\theta_i$.

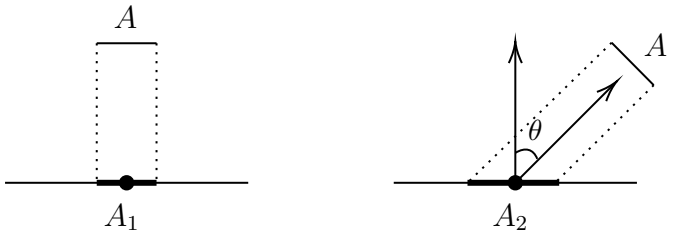


Figure 2.2: Example: on the right, the differential area A_2 illuminated by the ray is larger, because the ray is not perpendicular to the plane.

2.1.2 The BRDF

In order to accurately model the incoming and outgoing radiances at each point, it is essential to take into account the fact that surfaces can reflect incoming light. That is, for any direction ω_i, the incoming radiance $L_i(p, \omega_i)$ can contribute to the outgoing radiance $L_o(p, \omega_o)$ of another direction ω_o. For each surface point, this relationship is captured by the bi-directional reflectance distribution function (BRDF), written as $f_r(p, \omega_o, \omega_i)$. Formally, the BRDF is defined as

$$f_r(p, \omega_o, \omega_i) = \frac{dL_o(p, \omega_o)}{dE(p, \omega_i)} \tag{2.3}$$

where $dE(p, \omega_i)$ represents the differential incoming irradiance in the direction ω_i.

Using equation 2.2, it can be derived that

$$dE(p, \omega_i) = L_i(p, \omega_i) \cos\theta_i d\omega_i \tag{2.4}$$

which allows equation 2.3 to be re-written as

$$f_r(p, \omega_o, \omega_i) = \frac{dL_o(p, \omega_o)}{L_i(p, \omega_i) \cos\theta_i d\omega_i} \tag{2.5}$$

From this, it's straightforward to show that

$$L_o(p, \omega_o) = \int_\Omega L_i(p, \omega_i) f_r(p, \omega_o, \omega_i) \cos\theta_i d\omega_i + C \tag{2.6}$$

for some C.

Equation 2.6 shows that, given the incident radiances and the BRDF, the portion of outgoing radiances caused by reflections can be computed. As a result, the BRDF of a surface completely decides how the surface reflects light, which is a crucial factor of its appearances when viewed by a camera. Different materials in the real world have drastically different BRDFs, and it's vital for a physically based renderer to accurately implement these functions, if photorealistic images are to be rendered. A family of BRDFs known for their physical accuracy are the Microfacet BRDFs, which are well implemented by this project. Unfortunately, due to the complexity of these models, details of these BRDFs could not be described here.

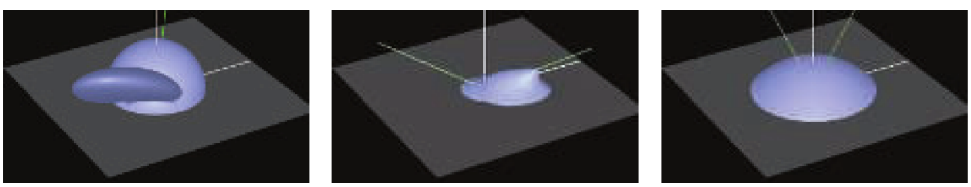


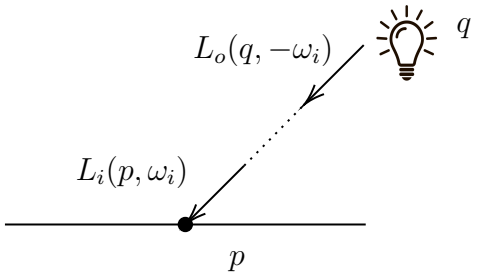
Figure 2.3: Shapes of example BRDFs. Image credit[2]

In order to fully model $L_o(p, \omega_o)$, it remains to describe the term C in equation 2.6. Surfaces in the real world sends outgoing radiances for two reasons only: they might emit light actively, and they reflect incoming light. Equation 2.6 already includes the reflected radiances, and thus it only remains to include actively emitted light. Writing $L_e(p, \omega_o)$ for the actively emitted radiance from p towards the direction ω_o, the following equation completely describes L_o:

$$L_o(p, \omega_o) = L_e(p, \omega_o) + \int_\Omega L_i(p, \omega_i) f_r(p, \omega_o, \omega_i) \cos\theta_i d\omega_i \tag{2.7}$$

which is the famous rendering equation, originally proposed by Kaijiya[8].

Under the assumption that radiance is constant along each ray, the incoming radiance $L_i(p, \omega_i)$ can be equated with the outgoing radiance from another point, q, as illustrated below.



Thus, defining the ray-tracing function $t(p, \omega_i)$ as a function that computes the first surface point q intersected by a ray from p in the direction ω_i, the rendering equation is re-written in a new form:

$$L_o(p, \omega_o) = L_e(p, \omega_o) + \int_\Omega L_o(t(p, \omega_i), -\omega_i) f_r(p, \omega_o, \omega_i) \cos\theta_i d\omega_i \tag{2.8}$$

Finding solutions to this equation is the ultimate goal of rendering, because the job of any renderer is to compute the amount of radiance received by a hypothetical camera placed in the scene. In other words, for each point p that is visible from the camera, the rendering algorithm must compute $L_o(p, \omega_o)$, where ω_o points from p towards the camera. The following section will begin the discussion of how this equation is solved in practice.

2.2 Monte Carlo Integration

Equation 2.8 cannot be solved analytically for all but the simplest scenes. Thus, rendering software solve the equation numerically using the method of Monte Carlo

integration. Given an integral

$$I = \int_\Omega f(x)dx,$$

the Monte Carlo estimator randomly samples N points $x_1, ..., x_n \in \Omega$ according to some distribution D, and computes

$$I_N = \frac{1}{N}\sum_{i=1}^{N}\frac{f(x_i)}{p(x_i)} \tag{2.9}$$

where $p(x_i)$ is the probability density of sampling x_i according to D. It can be proved that this indicator is both unbiased ($E[I_N] = I$) and consistent ($\lim_{N\to\infty} I_N = I$). When applied to the rendering equation, the support Ω is the sphere or hemisphere of directions, and the function f is computed by recursively estimating L_o, and then multiplying by the BRDF and the cosine of the direction.

2.2.1 Importance Sampling

In rendering, the variance $\mathbb{V}\text{ar}[I_N]$ of the Monte Carlo estimator manifest in the form of random noise in the resulting image. Thus, variance reduction techniques are vital for rendering high-quantity images efficiently. One of the most important such techniques is importance sampling.

In the monte carlo estimator, the distribution D used can be an arbitrary distribution. However, different distributions can lead to dramatically different variances. Importance sampling uses the fact that, informally, the variance $\mathbb{V}\text{ar}[I_N]$ is reduced as the shape of the PDF $p(x)$ becomes similar to the shape of the integrand $f(x)$. As an example, consider when the PDF is

$$p(x) = \frac{f(x)}{\int_\Omega f(x')dx'}.$$

In this case, p is always proportional to f, and the variance of the estimator is

$$\begin{aligned}\mathbb{V}\text{ar}[I_N] &= \frac{1}{N}\mathbb{V}\text{ar}_{x\sim D}\left[\frac{f(x)}{p(x)}\right] \\ &= \frac{1}{N}\mathbb{V}\text{ar}_{x\sim D}\left[\int_\Omega f(x')dx'\right] \\ &= 0\end{aligned}$$

It is of course infeasible to obtain a perfectly proportional PDF, because doing so requires computing the value of $\int_\Omega f(x')dx'$, which is the value to be estimated in the first place. However, even if p is similar in shape to f, variance can still be reduced.

2.2.2 Multiple Importance Sampling

When solving the rendering equation, the integrand is the product of two factors: the incoming radiance $L_o(t(p, \omega_i), -\omega_i)$, and the reflectance $f_r(p, \omega_o, \omega_i) \cos\theta_i$. More generally, the renderer is solving an integration of the form

$$I = \int_\Omega f(x)g(x)dx$$

In this case, if the renderer were to perform importance sampling to estimate this integral according to distributions based on either f or g, one of these two will often perform poorly[14]. This is exactly the issue addressed by the technique of multiple importance sampling, or MIS.

The basic idea of MIS is that, when estimating an integral, samples should be drawn from multiple distributions, chosen in the hope that at least one will match the shape of the integrand well. MIS provides a method of weighting the samples from each distribution that eliminates large variance spikes due to mismatches between the integrand and the sampling density. More precisely, given a PDF p_f that is a good match for f, and p_g that is a good match for g, MIS proposes to N samples $x_1, ..., x_N$ from p_f, and $y_1, ..., y_N$ from p_g, and apply the alternative Monte Carlo estimator:

$$I_N = \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)g(x_i)w_f(x_i)}{p_f(x_i)} + \frac{f(y_i)g(y_i)w_g(y_i)}{p_g(y_i)} \tag{2.10}$$

where the weights w_f and w_g are defined by

$$w_s(x) = \frac{p_f(x)^2}{p_f(x)^2 + p_g(x)^2}$$

A full recount of why this weighting scheme reduces variance can be found in [14].

In order to apply MIS to solving the rendering equation, the renderer need to sample from a PDF p_{rad} which matches to the incoming radiance $L_o(t(p, \omega_i), -\omega_i)$, and another PDF p_{ref} which matches the reflection term $f_r(p, \omega_o, \omega_i) \cos\theta_i$. In the path-tracing algorithm implemented by this project, p_{rad} is a distribution which only samples ω_i that points to a light source, which is often much brighter than surfaces that indirectly reflects light. For p_{ref}, which corresponds to surface reflection and is thus material dependent, the project implemented specific sampling routings for each individual material, thereby maximizing the similarity between the PDF and the BRDF of the materials.

2.3 The Path Space Formulation

The rendering equation as written in equation 2.8 is inconvenient for deriving algorithms, because the relationship between geometries in the scene is implicit in the ray-tracing function $t(p, \omega_i)$. This section rewrites the equation into a from that makes this relationship more explicit.

Firstly, equation 2.8 is to be transformed from an integral over directions into an integral over area. More precisely, the variable of integration ω_i, which represents an incoming direction, is to be replaced by a point p_{src}, which represents the source of the incoming ray. To achieve this change of variable, for any two points p, p' that are mutually visible, define

$$L_o(p' \to p) = L_o(p', \omega)$$
$$L_e(p' \to p) = L_e(p', \omega)$$

where $\omega = \frac{p-p'}{|p-p'|}$. Similarly, for three points p, p', p'', such that p is visible from p' and p' is visible from p'', the BRDF at p' is written as

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

where $\omega_i = \frac{p''-p'}{|p''-p'|}$ and $\omega_o = \frac{p-p'}{|p-p'|}$.

Note that, since not all points p_{src} is visible from p, the integrand needs to include a visibility term $V(p, p_{src})$, which takes the value 1 if p_{src} is visible from p, and 0 otherwise. Furthermore, the change of variable also incurs a Jacobian term, which is $\frac{\cos\theta'}{|p-p_{src}|^2}$, with θ' being the angle between the incoming ray and the surface normal at p_{src}. Use these terms, equation 2.8 is rewritten so that for a ray from a point p to the destination p_{dest}, the radiance is computed by

$$\begin{aligned} L_o(p \to p_{dest}) = \; & L_e(p \to p_{dest}) \\ & + \int_A f_r(p_{src} \to p \to p_{dest}) L_o(p_{src} \to p) V(p, p_{src}) \frac{cos\theta \cos\theta'}{|p - p_{src}|^2} dp_{src} \end{aligned}$$

where A is the domain of all surface points in the scene. To simplify notation, write the term $G(p, p_{src})$ as $V(p, p_{src}) \frac{cos\theta \cos\theta'}{|p-p_{src}|^2}$, the equation becomes

$$\begin{aligned} L_o(p \to p_{dest}) = \; & L_e(p \to p_{dest}) \\ & + \int_A f_r(p_{src} \to p \to p_{dest}) L_o(p_{src} \to p) G(p, p_{src}) dp_{src} \end{aligned}$$

The equation above is referred to as the surface form light transport equation, and it can be interpreted as a recursive definition for L_o. Naturally, one can expand this recursive definition, and obtain an infinite sum:

$$\begin{aligned} L_o(p_1 \to p_0) = & L_e(p_1 \to p_0) \\ & + \int_A L_e(p_2 \to p_1) f_r(p_2 \to p_1 \to p_0) G(p_2, p_1) dp_2 \\ & + \int_A \int_A L_e(p_3 \to p_2) f_r(p_3 \to p_2 \to p_1) G(p_3, p_2) f(p_2 \to p_1 \to p_0) G(p_2, p_1)\ dp_3 dp_2 \\ & + \ ... \end{aligned}$$

Here, each term represents the radiance contributed by paths of increasing length. For example, the first term in the sum represents the radiance directly emitted at p_1; the second term represents the radiance emitted at some point p_2 and reflected at p_1; the third term represents the radiance emitted at p_3 and reflected at p_2 and then again at p_1, and so on. For a path with n reflection points, the source emission is $L_e(p_{n+1} \to p_n)$, and it is weighted by a throughput term T_n, which is the product of the f_r and G at each reflection point:

$$T_n = \prod_{i=1}^{n} f_r(p_{n+1} \to p_n \to p_{n-1}) G(p_{n+1}, p_n)$$

which allows the previous equation to be written as

$$L_o(p_1 \to p_0) = \sum_{n=0}^{\infty} \underbrace{\int_A \int_A ... \int_A}_{n} L_e(p_{n+1} \to p_n) T_n dp_n dp_{n-1} ... dp_1$$

Notice that, the throughput term T_n computes the portion of radiance that remains after n reflections, which tends to decrease exponentially as n increases. For this reason, it is natural to only consider the first few terms of this infinite sum. More precisely, given a maximum amount of reflections $MaxDepth$, the radiance $L_o(p_1 \to p_0)$ can be estimated by

$$L_o(p_1 \to p_0) = \sum_{n=0}^{MaxDepth} \underbrace{\int_A \int_A ... \int_A}_{n} L_e(p_{n+1} \to p_n) T_n dp_n dp_{n-1} ... dp_1 \tag{2.11}$$

If a Monte Carlo estimator is used to approximate the value for each integral, the above formula becomes a recipe for a practical algorithm for computing radiances. This leads to the famous algorithm known as path-tracing, which is described in the next section.

2.4 The Path-Tracing Algorithm

The input of the path-tracing algorithm is a scene, defined by its geometries and materials, together with a camera, defined by its position, orientation, image resolution, and field of view (FOV) angle. Additionally, the algorithm is also parameterized by an integer *spp*, which stands for samples per pixel.

For each pixel on the image, the path-tracing algorithm samples *spp* random points on the pixel. For each of these random points, the algorithm generates $1 + MaxDepth * 2$ paths of increasing length, and accumulates the radiance carried by these paths. Finally, the radiances for all sample points in each pixel are combined by a Monte Carlo estimator, which decides the final value of the pixel.

The path tracing algorithm generates paths incrementally, where previous reflection points are reused. More precisely, after generating n reflection points $p_1, ..., p_n$, the algorithm does the following:

Algorithm 1: Incremental Path Construction

1. Samples a point p_{light} from a light source, and connect it to p_n. This generates a path of n reflections, whose radiance is to be accumulated;
2. Sample a direction ω_i from a distribution that matches the BRDF at p_n;
3. Trace the ray with origin p_n and direction ω_i. Let the intersection with the scene be p_{n+1};
4. If p_{n+1} is on a light source, then again this leads is a path of n reflections, whose radiance is accumulated;
5. Let p_{n+1} be the next reflection point;

Notice that, step 1 and 4 are two different ways a light emitter can illuminate the pont p_n, and these two different paths exactly correspond to Multiple Importance Sampling, described in subsection 2.2.2. Thus, when accumulating their radiances, the MIS weight should be applied.

To summarize, the path-tracing algorithm works as follows:

Algorithm 2: Path Tracing

```
1  foreach pixel in the image do
2      for i from 1 to spp do
3          Let p_0 be the location of the camera;
4          Randomly select a location inside the pixel, and generate a ray r
             corresponding to this location;
5          Compute the first intersection point p_1 between r and the scene;
6          Accumulate L_e(p_1 → p_0);
7          for n from 1 to MaxDepth do
8              Incrementally construct two further paths;
9              Accumulate the radiances from the newly generated paths,
                 weighted by MIS;
10     Combine radiances computed by each sample to obtain final color of pixel;
```

This algorithm is now the corner stone of photorealistic rendering. The remainder of this report will discuss the problem of efficiently implementing it on GPUs, and a variant of this algorithm where Reinforcement Learning is used to make smarter choices during importance sampling.

3 Implementing Path-Tracing

Bibliography

[1] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on gpus. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, page 145–149, New York, NY, USA, 2009. Association for Computing Machinery.

[2] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-time rendering*. Crc Press, 2019.

[3] Brent Burley, David Adler, Matt Jen-Yuan Chiang, Hank Driskill, Ralf Habel, Patrick Kelly, Peter Kutz, Yining Karl Li, and Daniel Teece. The design and evolution of disney's hyperion renderer. *ACM Trans. Graph.*, 37(3), July 2018.

[4] David Cline, Justin Talbot, and Parris Egbert. Energy redistribution path tracing. *ACM Transactions on Graphics (TOG)*, 24(3):1186–1195, 2005.

[5] Ken Dahm and Alexander Keller. Learning light transport the reinforced way. *CoRR*, abs/1701.07403, 2017.

[6] Randima Fernando. Percentage-closer soft shadows. In *ACM SIGGRAPH 2005 Sketches*, pages 35–es. 2005.

[7] Pat Hanrahan, David Salzman, and Larry Aupperle. A rapid hierarchical radiosity algorithm. In *Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, pages 197–206, 1991.

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

[9] Tero Karras. Maximizing parallelism in the construction of bvhs, octrees, and ¡i¿k¡/i¿-d trees. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*, EGGH-HPG'12, page 33–37, Goslar, DEU, 2012. Eurographics Association.

[10] Tero Karras and Timo Aila. Fast parallel construction of high-quality bounding volume hierarchies. In *Proceedings of the 5th High-Performance Graphics Conference*, HPG '13, page 89–99, New York, NY, USA, 2013. Association for Computing Machinery.

[11] Csaba Kelemen, László Szirmay-Kalos, György Antal, and Ferenc Csonka. A simple and robust mutation strategy for the metropolis light transport algorithm. In *Computer Graphics Forum*, volume 21, pages 531–540. Wiley Online Library, 2002.

[12] Markus Kettunen, Marco Manzi, Miika Aittala, Jaakko Lehtinen, Frédo Durand, and Matthias Zwicker. Gradient-domain path tracing. *ACM Transactions on Graphics (TOG)*, 34(4):1–13, 2015.

[13] Steven G Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, et al. Optix: a general purpose ray tracing engine. *Acm transactions on graphics (tog)*, 29(4):1–13, 2010.

[14] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically based rendering: From theory to implementation*. Morgan Kaufmann, 2016.

[15] Peter-Pike Sloan, Jan Kautz, and John Snyder. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 527–536, 2002.

[16] Eric Veach. *Robust Monte Carlo methods for light transport simulation*, volume 1610. Stanford University PhD thesis, 1997.

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