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 for 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 academia.

This project focuses on the widely-used rendering 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 standard path-tracing, 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 CPU implementations, and is even noticeably faster compared to one of the most academically renowned GPU renderer.

(a) A bathroom scene, rendered by path-tracing.

(b) A living room scene, rendered by path-tracing guided by reinforcement learning.

Figure 1: Images rendered by the software created in this project. Scenes (geometry, materials and textures) created by [2].

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 boost the efficiency of rendering. Firstly, this project utilizes GPUs (Graphics Processing Units), 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 renderer 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 were manufactured. In 1980, Whitted [21] 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 were not yet photorealistic.

The age of photorealism began in 1986 with the visionary work of Kaijiya [10]. This paper introduced an integral equation known as *the rendering equation*, which models the energy 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 to be used for real-time rendering applications[1] 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 [7, 19, 9] 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 ray-tracing, 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 [11, 12, 1].

Ever since the introduction of path-tracing, a variety of algorithms that builds on top of path-tracing have emerged. The most famous ones include bi-direction path-

[1] until perhaps, very recently. See NVIDIA RTX.

[2] Again, this is beginning to change with NVIDIA RTX.

tracing [20], Metropolis light transport [20, 13], energy redistribution path-tracing [4], and gradient-domain path-tracing [14]. One particularly interesting variant, which uses reinforcement learning to guide the generation of light rays [6], 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 [17] 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 [16], 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 is the `Mitsuba` renderer[3]. The GPU rendering routines of this project are 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 reduce control flow divergences and memory access latencies, so that the resulting implementation maximally utilizes the parallel architecture of GPUs.

[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.

This project made the observation that in certain lighting scenarios, the original path-tracing algorithm struggles to converge to a noiseless solution. To alleviate this problem, the project implements a variant of path-tracing, where reinforcement 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 measure 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, in a region of space S, integrating the irradiance of each point in S gives the total flux through S:

$$\Phi(S) = \int_S E(p)\,\mathrm{d}p \tag{2.1}$$

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 it is measured in 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 with.

In rendering, radiance often appears in the form $L_o(p, \omega)$ or $L_i(p, \omega)$, which denote the radiance going out of p and those 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.

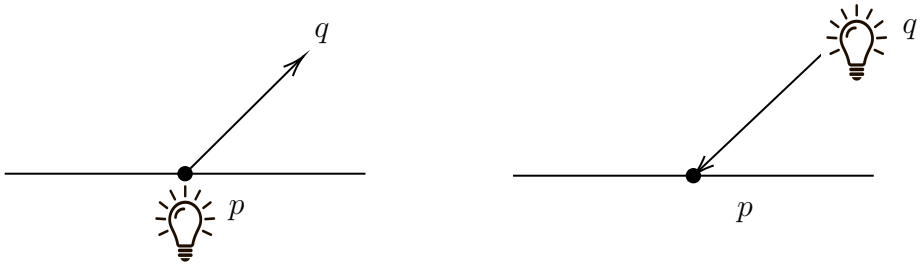


Figure 2.1: Example: consider two points p, q, and $\omega = \frac{q-p}{|q-p|}$. On the left, the radiance sent from p to q is $L_o(p, \omega)$. On the right, the radiance received by p from q is $L_i(p, \omega)$.

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:

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

Here, the support Ω is often a sphere or hemisphere of possible incoming directions, and the variable θ_i is 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 $\cos\theta_i$.

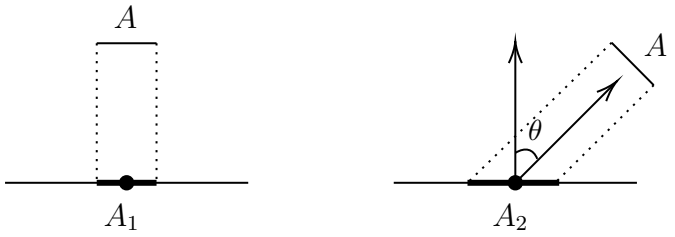


Figure 2.2: Example: on the right, the differential area A_2 illuminated by the ray is larger, because the ray is not perpendicular to the plane.

2.1.2 The BRDF

In order to model the outgoing radiances at each point, it is essential to account for the fact that surfaces reflect incoming light. That is, for any direction ω_i, the incoming radiance $L_i(p, \omega_i)$ may contribute to the outgoing radiance $L_o(p, \omega_o)$ of any other direction ω_o. 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

$$f_r(p, \omega_o, \omega_i) = \frac{\mathrm{d}L_o(p, \omega_o)}{\mathrm{d}E(p, \omega_i)} \tag{2.3}$$

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

Using equation 2.2, it can be derived that

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

which allows equation 2.3 to be re-written as

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

From this, it's straightforward to show that, for some C,

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

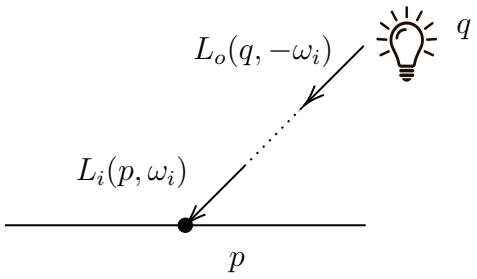
Different materials in the real world have drastically different BRDFs. The BRDF of a surface completely decides how it reflects light, which is a crucial factor of its appearance. This project implemented a range of different BRDFs corresponding to many different materials, but unfortunately, due to their complexity, details of these BRDFs could not be described here.

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 send outgoing radiances for two reasons only: they might emit light actively, and they reflect incoming light. Equation 2.6 already accounts for 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:

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

This is the famous rendering equation, originally proposed by Kaijiya [10].

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 $t(p, \omega_i)$ that computes the first surface point q intersected by a ray originated from p and travels in the direction ω_i, the rendering equation is re-written as:

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

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. For each point p visible from the camera, the rendering algorithm must compute $L_o(p, \omega_o)$, where ω_o points from p towards the camera.

2.2 Monte Carlo Integration

Equation 2.8 cannot be solved analytically. Thus, rendering software solves the equation numerically using the method of Monte Carlo integration, and the algorithms used by these software are referred as "integrators". Given an integral

$$I = \int_\Omega f(x) \, \mathrm{d}x,$$

a Monte Carlo integrator randomly samples N points $x_1, ..., x_n \in \Omega$ according to some probability density function (PDF) p, and computes

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

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 is 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 PDF p 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

$$p(x) = \frac{f(x)}{\int_\Omega f(x')\,\mathrm{d}x'}.$$

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

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

It is of course infeasible to obtain a perfectly proportional PDF, because doing so requires computing the value of $\int_\Omega f(x')\,\mathrm{d}x'$, 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

$$I = \int_\Omega f(x)g(x)\,\mathrm{d}x$$

In this case, if a renderer performs importance sampling according to distributions based on either f or g, one of these two would often perform poorly [17]. This is exactly the issue addressed by the technique of multiple importance sampling (MIS).

MIS proposes that, during Monte Carlo Integration, samples should be drawn from multiple distributions, chosen in the hope that at least one will match the shape of the integrand well. MIS weights the samples from each distribution in a way that

eliminates large variance spikes arising from mismatches between the sampling density and the integrand. 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 draws N samples $x_1, ..., x_N$ from p_f, and $y_1, ..., y_N$ from p_g, and applies 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

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

A full recount of why this weighting scheme reduces variance can be found in [17].

To apply MIS when solving the rendering equation, the renderer samples from a PDF p_{rad} that matches to the incoming radiance $L_o(t(p, \omega_i), -\omega_i)$, and another PDF p_{ref} that matches the reflection term $f_r(p, \omega_o, \omega_i) \cos \theta_i$. In the path-tracer implemented in this project, p_{rad} is a distribution that only samples ω_i that points to a light source, which is often much brighter than surfaces that indirectly reflect light. For p_{ref}, the project implemented specific sampling routings for each individual BRDF, thereby maximizing the similarity between the PDF and the BRDF at each point.

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. Specifically, 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, for any pair of mutually visible points p, p', 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 every point 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}. Using 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

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

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

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

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:

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

Here, each term represents the radiance contributed by paths of increasing length. 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:

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

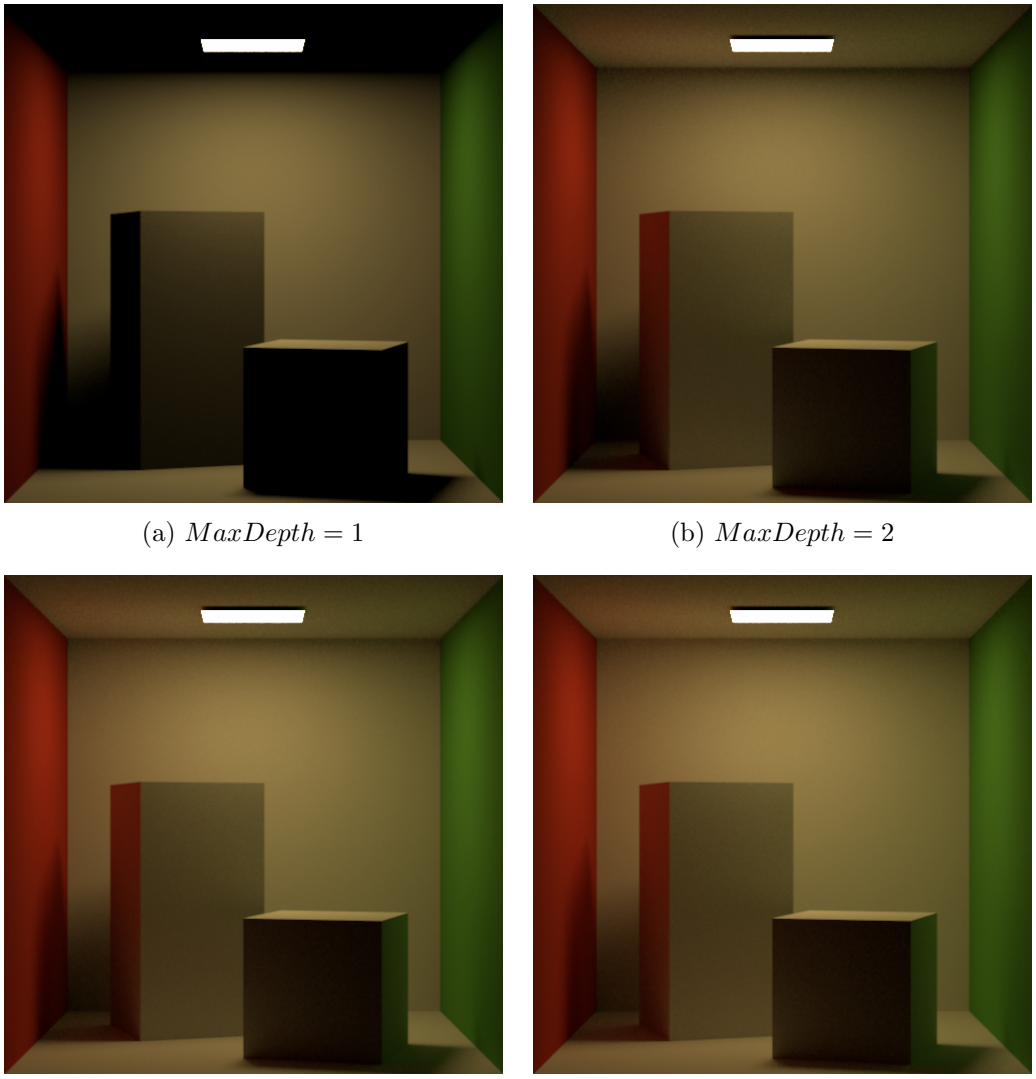
Substituting this into the previous equation gives

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

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 (see figure 2.3). More precisely, given a maximum amount of reflections $MaxDepth$, the radiance $L_o(p_1 \to p_0)$ can be estimated by

$$L_o(p_1 \to p_0) = \sum_{n=0}^{MaxDepth} \underbrace{\int_A \int_A \dots \int_A}_{n} L_e(p_{n+1} \to p_n) T_n \, \mathrm{d}p_n \, \mathrm{d}p_{n-1} \dots \mathrm{d}p_1 \tag{2.11}$$

If a Monte Carlo estimator is used to approximate the integrals, 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.



(a) $MaxDepth = 1$ (b) $MaxDepth = 2$

(c) $MaxDepth = 3$ (d) $MaxDepth = 4$

Figure 2.3: Effect of increasing $MaxDepth$

2.4 The Path-Tracing Algorithm

The path-tracing algorithm is parameterized by an integer, *spp*, which stands for samples per pixel. The algorithm samples *spp* random points on each pixel to be rendered, and generates an initial ray corresponding to each sample point. Starting from the ray, the algorithm constructs $MaxDepth * 2$ paths of increasing length, and accumulates the radiances carried by these paths. These radiances are combined by a MIS Monte Carlo estimator, which decides the final value of the pixel.

The path tracing integrator generates paths incrementally, starting from the camera location p_0. At each point p_{n-1}, the algorithm samples an incoming direction (for $p_1 \to p_0$, the direction is fixed by the configuration of the camera), and traces the incoming ray to find p_n. If the point p_n emits light actively, then this constitutes a path of $n-1$ reflection points. Moreover, p_n also serves as the next reflection point, so the algorithm samples a point p_{light} that illuminates p_n, and the radiance can be passed back to p_0 through a chain of reflections.

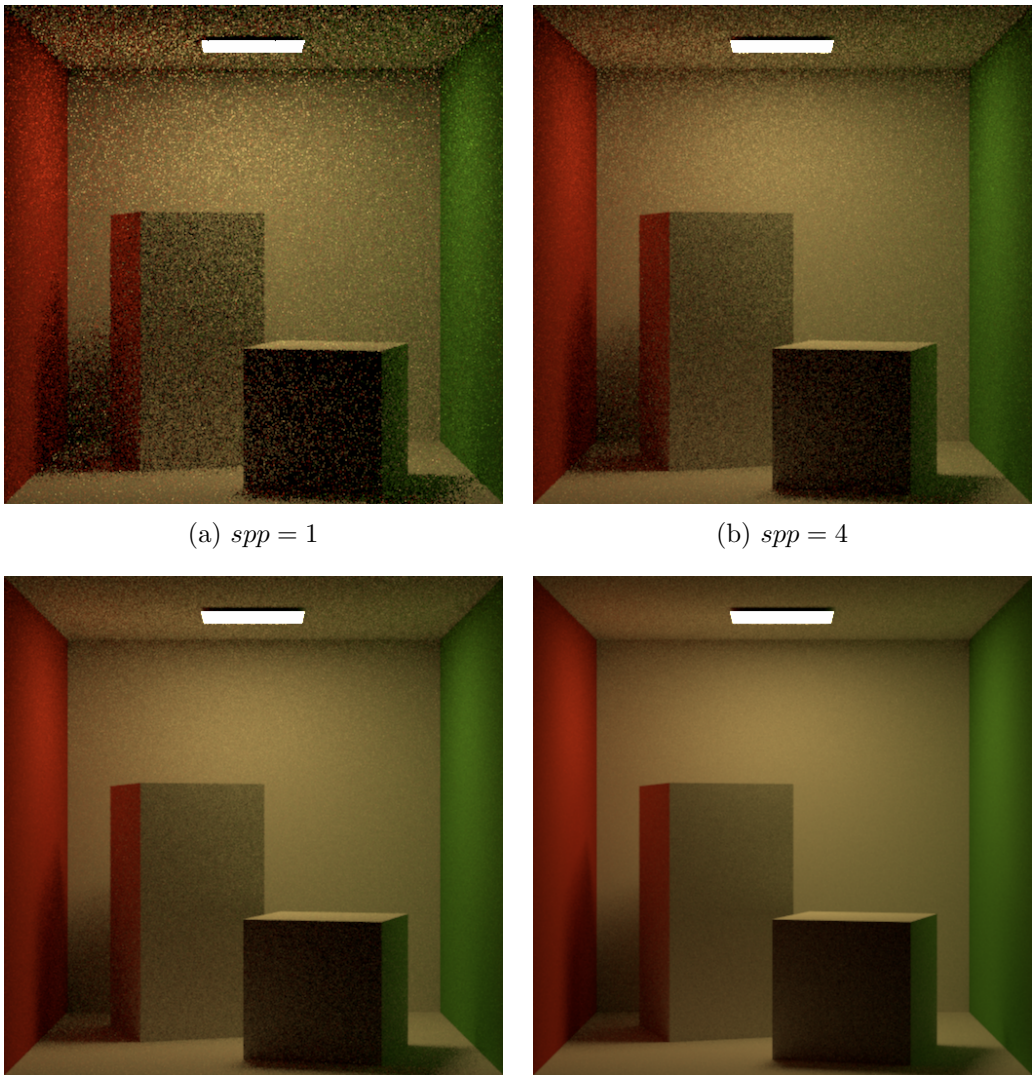
More precisely, the path tracing algorithm works as follows:

Algorithm 1: Path Tracing

```
1  foreach pixel in the image do
2      for i from 1 to spp do
3          Generate the initial ray r_0 going out of the camera position p_0;
4          for n from 1 to MaxDepth do
5              Find p_n by computing the intersection between r_{n-1} and the scene;
6              if p_n is on a light source then
7                  Accumulate the radiance of the path p_n → p_{n-1} → ... → p_0;
8              Sample a point p_light on a light source;
9              if the ray p_light → p_n is not occluded then
10                 Accumulate the radiance of the path p_light → p_n → ... → p_0;
11             Sample a direction ω_i from a PDF matching the BRDF at p_n;
12             Generate the ray r_n, originated at p_n and points towards ω_i;
13     Combine radiances computed by all samples to obtain final color of pixel;
```

Notice that, line 7 and 10 are two different ways a light emitter can illuminate p_{n-1}, and these two paths exactly correspond to the MIS procedure described in section 2.2.2. When accumulating their radiances, the MIS weight should be applied.

For each pixel, the integrator estimates its color using a total of $spp*2*MaxDepth$ paths. Thus, as the parameter *spp* increases, the variance of the Monte Carlo Estimator is reduced, which implies less noise in the rendered image. The following figure illustrates this effect.



(a) $spp = 1$ (b) $spp = 4$

(c) $spp = 16$ (d) $spp = 64$

Figure 2.4: Effect of increasing *spp*

For a typical scene, *spp* is required to be at least 1024 in order to reduce noise down to an imperceptible level. Common images to be rendered consist of around 1 million pixels, thus around 1 billion pixel samples are often needed. Moreover, each ray generated in the algorithm needs to be tested for intersection with the entire scene, which often consists of millions of geometric primitives. The next chapter of this report will discuss how this project copes with this humongous amount of computation and implements an efficient path-tracing renderer.

3 Implementing Path-Tracing

To maximize rendering efficiency, this projects implements path-tracing with parallelization on GPUs. This chapter begins with an introduction to GPU computing, and then discusses some of the most impactful performance optimizations employed by this project. These include careful structuring of GPU kernels, avoidance of performance degradation caused by polymorphism, and most importantly, usage of Bounding Volume Hierarchies to accelerate intersection detections.

3.1 The CUDA Programming Model

Originally built for real-time rendering, GPUs can handle a massive amount of geometries and pixels in parallel. The ability to do massively parallel computation motivated GPGPU (General-Purpose GPU) programming models, which became significantly useful for high performance computing. The software in this project is written using the CUDA programming model, developed by the NVIDIA Corporation.

CUDA employs the execution model known as SIMT (Single Instruction Multiple Threads). In this model, a large amount of threads can be spawned simultaneously, each running the same code on different data. CUDA threads are organized in groups of 32, known as *warps*. Each thread must execute the same instructions as the others in the same warp, or remain idle. When the threads within a warp access the memory, the entire warp can be paused and swapped out, so that a different warp can execute before the memory access finishes. Using this mechanism, the GPU hides memory access latencies by allowing very fast context switching. As a result, each physical core in the GPU can simultaneously handle multiple logical threads.

As an example, the GPU used for development of this project is an NVIDIA GTX1060 Mobile, which contains 10 *Streaming Multiprocessors*, each of which consists of 128 CUDA cores. Each streaming multiprocessor can have up to 2048 resident threads, giving a total of 20480 threads that can be simultaneously handled. Even though each GPU thread is not as fast as a CPU thread, the aggregated performance of the CUDA cores can still be many times faster than the CPU.

3.2 Wavefront Path-Tracing

In CUDA, a parallel GPU function that is invoked by the CPU is called a GPU *kernel*. When implementing algorithm 1 on GPUs, the most straightforward method is to implement the loop body (lines 3 to 12) as a GPU kernel, and invoke it for all pixel samples in parallel. This pattern of programming, where the entire computation is embodied in a single GPU kernel, is called the *Megakernel*[15] approach.

In contrast to megakernels, the *Wavefront* programming pattern divides the computation into many small kernels, where intermediate results are explicitly stored in memory. The CPU invokes these kernels one after another. On GPUs, regions of code that use a large amount of registers or have a high control flow divergence can significantly hurt the parallelization of the entire kernel, and thus the wavefront pattern can effectively contain these regions of code into separate kernels, so that the performance of the other kernels remain unaffected. However, the wavefront pattern incurs additional overhead by requiring more CPU-GPU communication, and memory IOs.

The project carried out numerous experiments to find an optimal way of dividing the rendering work into wavefronts. In the end, the program implemented is structured as indicated by the following pseudo-code:

Algorithm 2: Wavefront Path Tracing

```
1  foreach pixel sample in a parallel kernel do
2      generate the initial ray r_0 going out of the camera position p_0;
3  for n from 1 to MaxDepth do
4      foreach previously generated ray r in a parallel kernel do
5          Find p_n by computing the intersection between r and the scene;
6          if p_n is on a light source then
7              Accumulate the radiance of the path p_n → p_{n-1} → ... → p_0;
8      foreach newly found p_n in a parallel kernel do
9          Sample a point p_light on a light source;
10         Test whether the ray p_light → p_n occluded by some geometry;
11     foreach unoccluded p_light → p_n in a parallel kernel do
12         Accumulate the radiance of the path p_light → p_n → ... → p_0;
13     foreach p_n in a parallel kernel do
14         Sample a direction ω_i from a distribution matching the BRDF at p_n;
15         Generate a new ray, originated at p_n and travels in the direction of ω_i;
```

It was found that compared to a naive Megakernel implementation, this wavefront implementation is almost twice as fast.

3.3 Polymorphism

In GPU programming, a common source of performance degradation is dynamic-dispatch polymorphisms. On CPUs, dynamic-dispatch is easily implemented via virtual functions, but this can be costly for GPUs for two main reasons:

1. In the GPU, each thread in a warp must execute the same instructions as the others, or remain in idle. Thus, when different threads are dynamically dispatched to different code regions, control flow divergences occur, and the GPU becomes under-utilized.

2. Virtual functions make use of virtual tables and virtual pointers. During dynamic dispatch, the value of the program counter needs to fetched by reading the correct entry from the virtual table. This adds an additional level of indirection, which is costly on the GPU where memory IOs are more expensive.

The need for polymorphism in path tracing mainly comes from two components. Firstly, the scene to be rendered is often defined from various different geometric primitives (triangles, spheres, disks, etc.), and each primitive requires its own intersection detection code. Secondly, each object in the scene is associated with a certain surface material, and each material requires its own BRDF implementation. This project handles the polymorphism for these two components in different strategies.

For geometric primitives, this project chooses to completely eliminate the need for any primitive other than triangles. Before rendering begins, this project uses a preprocessing phase, where a triangle mesh is generated to represent each non-triangle primitive. This slightly increases rendering cost, because often a few thousands of triangles are needed to give a good approximation for shapes such as spheres. However, this overhead is outweighed by the benefit of reduced control flow divergences and memory IOs.

Unfortunately, polymorphism for materials cannot be handled in this manner, and thus it is impossible to avoid polymorphism all together. This project implements polymorphism not by virtual functions, but by using a templated `variant` class[1], where all polymorphic types are included as type arguments. As a pseudo-code example, the following type declaration could be used to define a general material type that could in fact be plastic, metal, or glass:

```
using Material = std::variant<Plastic, Metal, Glass>;
```

[1] `https://en.cppreference.com/w/cpp/utility/variant`

Implementation wise, a `variant` class is realized by *tagged union*s, which is in essence the same technique used to implement sum types in languages such as Haskell. Dynamic dispatch can be provided not via virtual functions, but by switching on an integer label attached to the object. This effectively avoids the extra memory operation incurred by dereferencing virtual pointers.

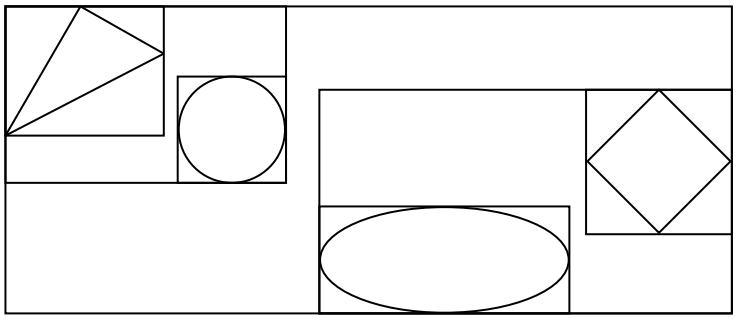
The usage `variant` solves problem 2, but problem 1 still remains because control flow divergence still exists whenever threads in the same warp process different materials. To solve this problem, this project employs the method of [15], which sorts the material evaluation tasks before they're executed. More precisely, in algorithm 2, before the kernel at line 9 is executed, this project inserts an extra phase where all tasks are sorted according to the material at their respective p_n. This step groups together threads that work on the same material, and there will only be a few warps that experience any divergences. Even though the sorting incurs a small extra cost, it was observed that it considerably improves the overall rendering performance.

3.4 Bounding Volume Hierarchy

In path-tracing, the most expensive step is almost always ray-scene intersection detection. This is very frequent operation, appearing in line 5 and 8 of algorithm 2. Thus, any performance gain in the intersection detection routine could prove to be significantly beneficial to the entire rendering procedure.

In a naive implementation, intersection detection would be performed by iterating through all geometries, and testing for intersection against each. With N being the amount of geometric primitives in the scenes, this is an $O(N)$ algorithm. Since N is often in the order of millions, this linear time algorithm is unacceptably slow. For this reason, this projects implements a data structure known as Bounding Volume Hierarchy (BVH), which reduces the complexity down to $O(\log N)$.

The idea of BVH is to partition the geometric primitives into a hierarchy of disjoint sets. The sets are organized into a binary tree, so that the root node is the set containing all primitives, and the leaves are singleton sets. Each interior node n has two disjoint children sets, and the union of the children sets is equal to n itself. In BVH, each set of primitives is represented by an Axis-Aligned Bounding Box (AABB), which is a box where each edge is parallel to a coordinate axis. Each AABB represents the set of all primitives that are completely enclosed by the box. The following image shows an example 2D BVH:



In this example, there are 4 primitives, each occupying an AABB as a leaf node. One AABB groups together the triangle and the circle, and another groups the ellipse and the rombus. These two AABBs are then grouped together by the root AABB.

3.4.1 BVH Traversal

In a BVH, if one AABB doesn't intersect the ray, all descendent boxes can be pruned from the search. This motivates the following algorithm, which finds the first intersection point between a ray and a BVH node by traversing recursively:

Algorithm 3: Recursive BVH Traversal

```
1  Function FindFirstIntersection(node, ray):
2      if node is a leaf node then
3          if ray intersects with the geometry of node then
4              return the intersection between ray and this geometry;
5          else
6              return "No Intersections";
7      if ray does not intersect the AABB of node then
8          return "No Intersections";
9      Recursively, call FindFirstIntersection(node.leftChild, ray) and
         FindFirstIntersection(node.rightChild, ray);
10     if both children do not intersect ray then
11         return "No Intersections";
12     else
13         return the shortest intersection with one of the children nodes;
```

Unfortunately, this recursive traversal is unsuitable for implementation on GPUs. Firstly, GPUs have very restrictive recursion depth limits (for CUDA, this is 24), and thus might fail to complete the traversal. Secondly, due to the massive amount of parallel threads, maintaining a recursion stack for each thread is significantly costly in terms of memory and register usage. For these reasons, this projects explicitly maintains a stack of nodes to be traversed, and operates on this stack iteratively.

In addition to eliminating recursion, this project also makes two important observations that lead to a fruitful optimization. Firstly, the project notices that if it is known that a node can only produce intersections further than the current shortest intersection found, then it is useless to push it onto the stack. Secondly, the project notices that the intersection distance between the ray and an AABB is a lower-bound of the shortest intersection distance between the ray and any geometry in the AABB. Motivated by these observations, this project keeps track of the distance of the shortest intersection found, and then makes the following optimizations:

- If a child node intersects the ray, but the ray-AABB intersection distance is further than the shortest intersection, then the node is discarded, because it cannot produce a shorter intersection.
- If both children nodes intersect the ray and are not discarded, then the node with the shorter intersection distance should be traversed first, because it has a higher chance of finding a better intersection.

Through experimentation, this project found that these techniques roughly double the efficiency of BVH traversal. The following pseudo-code summarizes the BVH traversal routine that this project implements:

Algorithm 4: Recursive BVH Traversal

```
1  Function FindFirstIntersection(root, ray):
2      Create a stack S, with root being the only element;
3      Let minDistance := ∞;
4      while S is non-empty do
5          Pop a node n from the stack;
6          if n is a leaf node then
7              if ray intersects with the geometry of node then
8                  Update minDistance if this intersection is shorter than
                    minDistance;
9          Compute leftMinDistance as the minimum intersection distance
            between the left child AABB of n with the ray;
10         Compute rightMinDistance as the minimum intersection distance
            between the right child AABB of n with the ray;
11         if exactly one child has intersection shorter than minDistance then
12             Push that child onto the stack;
13         if both children have intersections shorter than minDistance then
14             Push the further child onto the stack;
15             Push the shorter child onto the stack;
16     return the shortest intersection found;
```

3.4.2 BVH Quality Measure

BVHs differ in quality, and these differences have a hugely impact on performance. Firstly, a BVH is binary tree, so the balance of the tree is related to the cost of the traversal. More importantly, notice that in algorithm 4, it's possible that a ray r intersects with both children of a node n, and thus both subtrees need to be traversed. A "good" BVH must reduce the likelihood of this type of situation.

To formally quantify the quality of BVHs, this project employs the Surface Area Heuristic (SAH), first introduced in [8]. The SAH heuristic estimates the expected cost of a single BVH traversal for a random ray, and it is compute by

$$C_i \sum_{n \in I} \Pr(\mathcal{X}_n) + C_l \sum_{n \in L} \Pr(\mathcal{X}_n)$$

where I is the set of interior nodes, L the set of leaves, C_i the cost for intersection testing a single interior node, C_l the cost for intersecting testing a leaf node, and $\mathcal{X}_n$ the event that a random ray intersects the node n. SAH uses the observation that, for a convex shape S_1 enclosed within another convex shape S_2, the conditional probability $\Pr(\mathcal{X}_{S_1} | \mathcal{X}_{S_2})$ is exactly equal to $\frac{A(S_1)}{A(S_2)}$, where A is the function that computes surface area. Thus, the SAH heuristic becomes

$$C_i \sum_{n \in I} \frac{A(n)}{A(root)} + C_l \sum_{n \in L} \frac{A(n)}{A(root)}$$

Given a collection of primitives, finding the optimal BVH tree according to the SAH heuristic is an NP-hard problem [12]. This is not entirely surprising, considering the fact given any node with N geometries, there are $2^N - 2$ ways of constructing the two children nodes. However, not all hope is lost if the goal is to construct a BVH that's not necessarily optimal, but still reasonably good. This project implements a two-step system to approach high-quality BVH construction:

1. Firstly, construct a BVH using a naive heuristic: at each node, split the AABB at the mid-point of a coordinate axis, thereby dividing the geometries into two groups. This is known as a Linear BVH, which can be constructed efficiently on GPUs using the method of [11], but the tree created is of poor quality.

2. Secondly, optimize the previously created BVH tree, using the method of [12]. This step repeatedly solves the NP-hard optimization problem for very small *treelets*, which can still significantly improve the quality of the entire tree.

This two-step approach is similar to the BVH construction routine of NVIDIA's Optix library. Details for the two steps are described below.

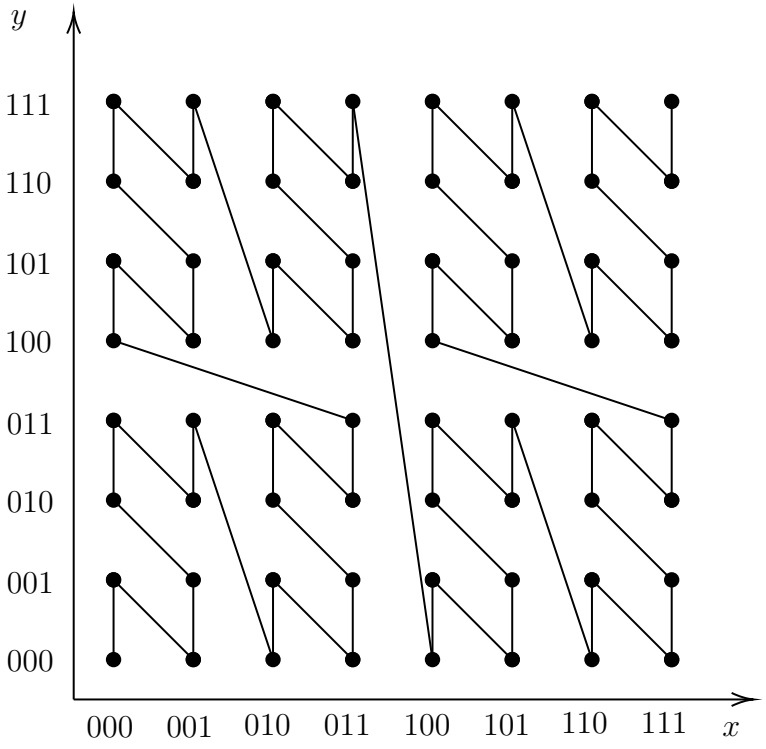
3.4.3 Linear BVH Construction

In Linear BVH, the AABB of each node is split at the mid-point of a coordinate axis, dividing the geometries into two groups. In three dimensions, the X, Y, and Z axises are used for splitting in a round-robin fashion, as the depth of the tree increases. A naive implementation of this splitting method would start at the root of the tree, and recursively divide each node using this heuristic. Unsurprisingly, this recursive algorithm parallelizes poorly. This motivates this project to employ a more sophisticated method described at [11], which uses a spatial encoding called *morton code* and a data structure known as *binary radix tree*.

The algorithm begins by taking the centroid of each geometric primitive. The algorithm scales and rounds each component of the centroid, so that it is represented by a 21-bit integer. Then, using the integer tuple (x, y, z) which represents the position of the centroid, the algorithm computes its *morton code*, which is a 63-bit integer defined as

$$M = x_0\ y_0\ z_0\ x_1\ y_1\ z_1\ ...\ x_{20}\ y_{20}\ z_{20}$$

where x_i is the ith bit of the x-coordinate, and the same for y_i and z_i. As an example, the image below illustrates a two dimensional 6-bit morton code. For each pair of (x, y) coordinate, its morton code M is the index of the point along the zig-zag curve which begins at the lower left corner.



The morton code has many useful properties. To begin with, the most significant bit of M determines exactly whether the primitive is located in the left half or the right half the scene, and the 2nd most significant bit determines whether it is located in the upper or the lower half. Continuing this pattern, for any rectangular region of the space defined by the first k significant bits, the $(k+1)$th significant bit would split that region of space into two parts, alongside one of the coordinate axises. This structure is in good correspondence with the mid-point BVH splitting strategy.

Given the morton codes of all primitives, one could straightforwardly construct a binary prefix tree where the ith level contains 2^i nodes, each of which corresponds to an assignment of the first i bits of a morton code. However, such a tree would not correspond to a useful BVH. This is because geometries are always distributed *sparsely* in the scenes, and thus only a very small portion of the 2^{63} morton code values are actually used by some geometry. Thus, the algorithm [11] implemented in this project uses a variant of prefix tree known as the radix tree, which compactly stores sparsely distributed keys.

For a set of N binary keys $k_0, ..., k_{N-1}$, a binary radix tree is binary tree whose leaves are the keys in sorted order. Each interior node n in the tree is also labeled by a bit sequence, which is the longest common prefix of all leaves in the subtree rooted at n. Using $\delta(i, j)$ to denote the length of the longest common prefix between k_i and k_j, the ordering of leaves implies that if $i \leq i' \leq j' \leq j$, then $\delta(i, j) \geq \delta(i', j')$. Thus, for a node whose leftmost descendent is k_i and rightmost descendent k_j, its prefix has length exactly $\delta(i, j)$.

In the radix tree, each internal node partitions its keys using the first differing bit, i.e. the $\delta(i, j)$th bit counting from 0. By definition, since $\delta(i, j)$ is the maximum length of common prefix, there must exists at least two keys that defer on the $\delta(i, j)$th bit. More precisely, let k_γ be the rightmost key in this subtree whose $\delta(i, j)$th bit is 0, then $k_{\gamma+1}$ must be such that the $\delta(i, j)$th bit is 1. The radix tree partitions this interior node so that the left child covers keys k_i to k_γ, and the right child covers keys $k_{\gamma+1}$ to k_j. An example radix tree with 8 5-bit keys are shown in the figure below.

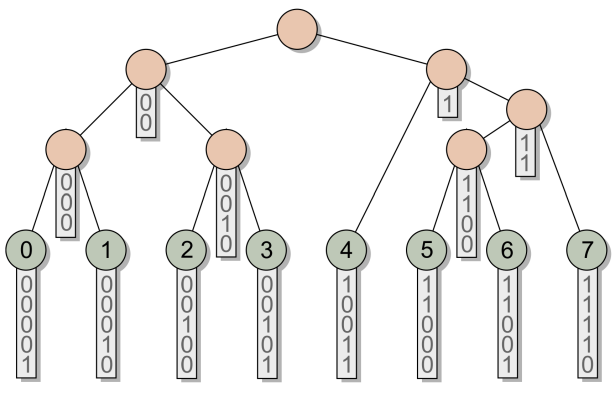


Figure 3.1: Example radix tree. Image credit [11]

In order to construct radix trees in parallel, the algorithm [11] establishes a connection between node indices and keys through a specific tree layout, which enables any interior node to be built simultaneously with its children. The layout stores interior and leaf nodes in two separate arrays, I and L. The leaf array is sorted on the keys increasingly, and the interior array is arranged such that

1. The root of the tree is at I_0.

2. For each interior node n which covers keys $k_i, ..., k_j$ and splits at k_γ: the left child is stored at I_γ if it is also an interior node, or at L_γ if it is a leaf. Similarly, the right child is stored at either $I_{\gamma+1}$ if it is interior, or at $L_{\gamma+1}$ if it is a leaf.

Notice that, these two rules enforce a special property: the index of an interior node coincides with the index of either its leftmost leaf descendent, or its rightmost leaf descendent. This property is visualized in the following image, where each horizontal bar represents the range of keys covered by an interior node.

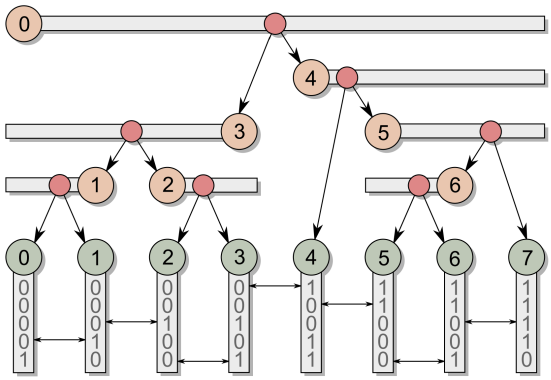


Figure 3.2: Radix tree storage layout. Image credit [11].

Given an interior node, it is not too hard to find out whether its index corresponds to the leftmost descendent, or the rightmost one. More precisely, for the interior node I_x, it suffices to compare $\delta(x, x+1)$ and $\delta(x, x-1)$. If the former is larger, then x must be the index of the leftmost descendent, and otherwise x coincides with the rightmost descendent. Furthermore, the smaller value of $\delta(x, x+1)$ and $\delta(x, x-1)$ provides a minimum threshold of the maximum common prefix length $\delta(i, j)$ for I_x, that is, $\delta(i, j) > \min(\delta(x, x+1), \delta(x, x-1))$. This is because, the minimum of $\delta(x, x+1)$ and $\delta(x, x-1)$ must equal the maximum common prefix length of the parent of I_x, and since L_x is the children, it must have a greater maximum common prefix length.

At this point, the algorithm knowns one of i and j, and it also knows a lower-bound $\delta_{\min}$ for $\delta(i, j)$. Without loss of generality, assume that the leftmost index i

is known. Notice that, j is a value that satisfies $\delta(i,j) > \delta_{\min}$, and it must be the maximum index that does so. Thus, it suffices to perform a binary search in the interval $[i+1, N]$, and find the largest j that satisfies $\delta(i,j) > \delta_{\min}$.

Having computed the leftmost and rightmost descendants i, j and the maximum common prefix length $\delta(i,j)$, it only remains to identify the index of the two children, γ and $\gamma+1$. Notice that, since γ is the rightmost index that still belongs in the left child, it is known that $\delta(i,\gamma) > \delta(i,j)$, and γ is the largest index in $[i,j]$ that satisfies this property. Thus, it is again straightforward to use binary search to find γ.

For each internal node I_x, the algorithm identifies its two children without requiring the children to have been constructed already. Thus, the algorithm can be parallelized across all nodes. The following pseudo-code summarizes this procedure:

Algorithm 5: Parallel Binary Radix Tree Construction

```
1  foreach internal node L_x in parallel do
2      δ_min := min(δ(x, x+1), δ(x, x−1)) ;
3      if δ(x, x+1) > δ(x, x−1) then
4          i := x;
5          Binary search to find the largest j such that δ(i, j) > δ_min;
6      else
7          j := x ;
8          Binary search to find the smallest i such that δ(i, j) > δ_min;
9      Binary search to find the biggest γ such that δ(i, γ) > δ(i, j);
10     Left child is I_γ as long as i ≠ γ, and L_γ otherwise;
11     Right child is I_{γ+1} as long as j ≠ γ+1, and L_{γ+1} otherwise;
```

One caveat with this algorithm is that no duplicates are allowed in the keys $k_0, ..., k_{N-1}$. However, morton codes for two geometries could potentially be identical, if they're really close together. This problem can be easily taken care of by appending extra bits after the morton code and ensuring that all binary keys are distinct.

Having constructed the binary radix tree, it only remains to convert it into a BVH by computing an AABB for each node. This can be done in parallel by having all threads start at leaf nodes, move up the tree, and repeatedly compute the AABB for the current node along the way. Whenever two sibling threads move up to process a common parent, the two threads carry out a simple consensus operation to decide which thread should drop out and terminate.

This project implemented this BVH construction in CUDA. Thanks to the parallel nature of this algorithm, the project observed that the running time of this BVH construction algorithm is almost negligible ($\sim$50ms for 1 million primitives) compared to the cost of the actual path-tracing phase ($\sim$10 minutes for a 1024 *spp* render).

3.4.4 BVH Optimization

The BVH trees constructed by the algorithm from the previous section can immensely accelerate intersection detections and thus rendering. However, the quality of these trees are still quite poor compared to trees generated with the guidance of the SAH heuristic. To address this issue, this project implements the algorithm described in [12], which optimizes existing BVH trees. With little running time cost, this optimization step boosts the rendering efficiency by about 300%.

It is believed that finding the optimal tree under the SAH heuristic is NP-hard [12]. However, intuitively, if the tree is optimal in every local *treelet*, the entire tree would be somewhat optimal. Formally, a *treelet* rooted at some node is defined to be a collection of immediate descendants of the root, consisting of $m-1$ internal nodes and m leaf nodes (which can still be internal nodes in the complete tree). The algorithm solves the SAH optimization problem using dynamic programming for each treelet with 7 leaves, in the hope that these local transformations improve the structure of the entire tree.

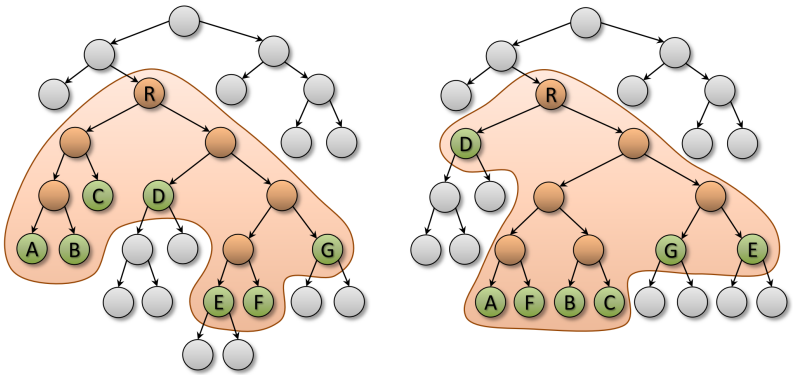


Figure 3.3: A treelet with 7 leaves and how it can be transformed. Image credit [12]

For each internal node n, the algorithm defines its cost to be

$$C(n) = \begin{cases} C_i A(n) + C(n_l) + C(n_r) & \text{if } n \text{ is an interior node in the full BVH} \\ C_l A(n) & \text{if } n \text{ is a leaf node in the full BVH} \end{cases}$$

where n_l and n_r are the left and right child. The cost defined this way is exactly the SAH cost multiplied by a constant factor of $A(root)$. For each internal node n that is the root of a treelet with 7 leaves, the algorithm optimizes $C(n)$ by finding the optimal structure of a local treelet. If more than one treelets with 7 leaves are rooted at n, the algorithm operates on the one where the leaves have the greatest surface area, which maximizes the potential for improvements.

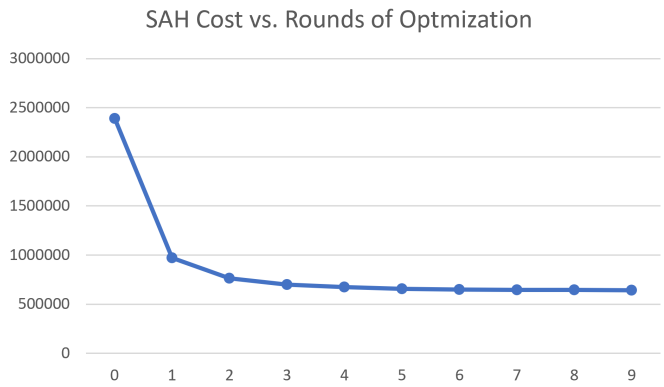
The algorithm optimizes treelets via dynamic programing. Specifically, it optimizes subsets of leaves of increasing size, and memoizes intermediate results. When working on a size-k subset S, it suffices to enumerate all possible binary partitions of S, and use memoized results to obtain optimal tree structure for each partition. Then, when the entire set of leaves is optimized, the treelet itself is also optimized. The procedure for optimizing each treelet is given in the following pseudo-code:

Algorithm 6: Treelet Optimization

```
Create memoization array C_opt of size 2^7, indexed by subsets of the leaves
  (empirically represented as bitmaps by 7-bit integers);
foreach leaf node n of the treelet do
    C_opt[{n}] := C(n);
for k from 2 to 7 do
    foreach subset S of the leaves of size k do
        C_opt[S] := ∞;
        A(S) := surface area of the AABB that encloses all primitives in S;
        foreach pair of disjoint nonempty subsets U, V, where U ∪ V = S do
            thisCost := A(S)C_i + C_opt[U] + C_opt[V];
            if thisCost < C_opt[S] then
                C_opt[S] := thisCost;
                Record (U, V) as current best partition for S;
Restructure the treelet using the best partitions found;
```

For treelets rooted at the same level in the BVH, this optimization step can be parallelized because the treelets do not overlap. Thus, this project implemented the algorithm in GPU, where nodes across each level are processed in parallel. Again, this step is extremely efficient, taking approximately 200ms for a scene with 1 million triangle. For such scenes rendered at 1024 *spp*, this optimization step could reduce rendering time from around 30 minutes to around 10 minutes.

Interestingly, the optimization does not reach a fixed point after processing each treelet once. The following image plots how the SAH (and thus rendering time) changes when the BVH is optimized repeatedly. This project runs the optimization for 3 rounds for each scene, which almost always reduces SAH to $\frac{1}{3}$ of the origin value.

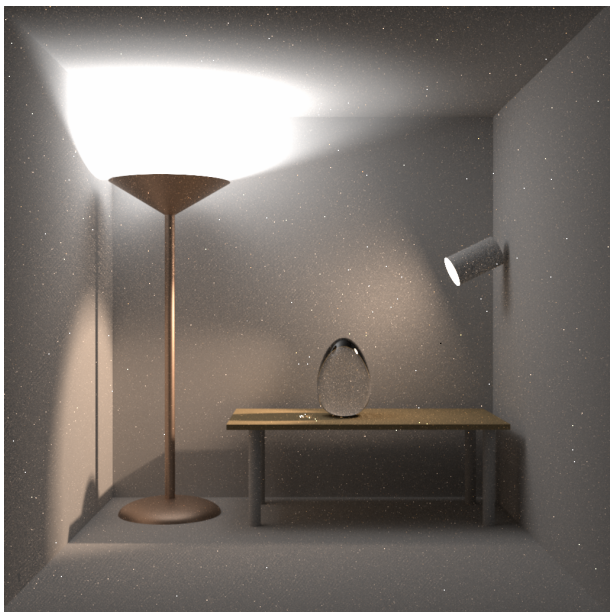


4 Reinforcement Learning Path Tracing

Traditional path-tracing struggles for scenes where indirect lighting dominates. To alleviate this problem, this project implements a variant of path-tracing described at [6], where reinforcement learning guides the generation of new rays.

4.1 Motivation

As described in section 2.2.2, the path-tracing integrator samples from two MIS distributions: p_{ref} which matches the local reflectance, and p_{rad} which only samples points on light sources. The efficiency of this method relies on the assumption that one of p_{ref} and p_{rad} is a good match for the integrand of equation 2.8. However, this is not always true, as exemplified by the scene rendered below.



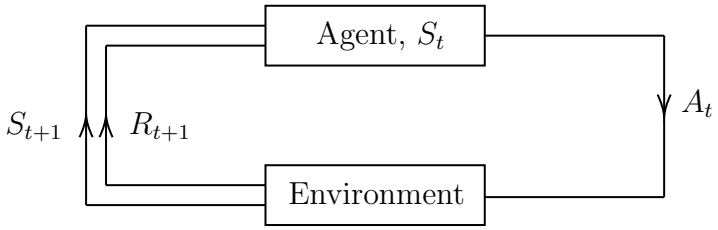
In this scene, both light sources are surrounded by opaque geometries, which means only a small region can be illuminated directly. Furthermore, the reflected radiance from this small region is the primary source of illumination for most surfaces in the scene. During path-tracing, the light sources sampled from p_{rad} are often

occluded, and hence p_{rad} is poor match for the integrand. The directions sampled from p_{ref} are also rarely useful, because only a small set of directions provide indirect illumination. As both MIS distributions are ineffective, the algorithm converges extremely slowly.

4.2 Method

This project implements the algorithm proposed by [6], which uses online reinforcement learning to identify a distribution p_{RL} of rays that is a good match for the integrand of the rendering equation.

In the reinforcement learning setting, an agent, starting from state S_1, interacts with an environment through a sequence of actions $A_1, A_2, ..., A_N$. After each action A_t, the agent receives a reward R_{t+1}, and transitions into a new state S_{t+1}. Often, the goal of the agent is to maximize the rewards it receives through the actions. The following figure illustrates this system.



A commonly used method of guiding an agent is to maintain a table Q, where the value $Q(s, a)$ is an indication of how good is the choice of taking the action a at state s. A well-known method for updating Q and moving the agent at the same time is the *Expected Sarsa* strategy. At each state s, the strategy chooses an action a from a distribution $\pi(s, a)$ which depends on $Q(s, a)$. This gives the agent a reward $R(s, a)$, and leads the agent to a new state S'. The algorithm then updates $Q(s, a)$ by the following formula:

$$Q(s,a) := (1-\alpha)Q(s,a) + \alpha\left(R(s,a) + \gamma \sum_{a'} \pi(s',a')Q(s',a')\right)$$

where $\alpha \in [0, 1]$ is a learning rate, and $\gamma \in [0, 1]$ a discount factor which reflects the fact that future awards are less important than immediate ones. Intuitively, this algorithm attempts to balance the *exploitation* of actions with high Q values and the *exploration* of actions with low Q values.

Path-tracing bears many similarities with expected Sarsa. Specifically,

- Each surface point p can be regarded as a state s.
- The incoming direction ω_i to be traced can be seen as the action taken a.
- The actively emitted radiance L_e received by p from ω_i can be viewed as the immediate reward $R(a, s)$.
- The reflectance term $f_r(p, \omega_o, \omega_i) \cos\theta_i$ corresponds to the discount factor γ.
- The incoming radiance L_i corresponds to Q.

In [6], it was proposed that the expected Sarsa algorithm can be used learn the distribution of Q (as thus L_i). When weighted (discounted) by the reflectance term $f_r(p, \omega_o, \omega_i) \cos\theta_i$, this distribution becomes a good distribution to sample incoming rays from, according to importance sampling.

In this correspondence between path-tracing and reinforcement learning, the domain of states S and actions A are both continuous, which makes a complete tabulated representation of Q impossible. To work around this, this project divides the entire scene into a 3D grid, typically of size $32 \times 32 \times 32$. All surface points within the same grid cell are considered to be at the same state s. Moreover, at each grid cell, the spherical domain of incoming directions ω_i is also divided into 128 equally-sized sectors, and the directions in each sector are mapped to the same action a.

During path-tracing, at each surface point p_n, instead of sampling a direction from the reflectance-matching p_{ref}, the algorithm uses the following sampling procedure:

Algorithm 7: Reinforcement Learning Guided Ray Generation

Finds the grid cell s_n that encloses p_n;
foreach sector a at the cell s_n **do**
    Sample a direction ω_i within a uniformly at random;
    Compute the discounted Q value $Q'(s_n, a) := f_r(p, \omega_o, \omega_i) \cos\theta_i Q(s_n, a)$;
Construct the discrete distribution over sectors $p_{sec,n}(a) = \frac{Q'(s_n,a)}{\sum_{a'} Q'(s_n,a')}$;
Sample a sector a_n from $p_{sec,n}$;
Sample a direction ω_i within a_n to trace the next ray towards;

After tracing ω_i from p_n to find the next surface point p_{n+1}, the algorithm uses expected Sarsa to update the Q table:

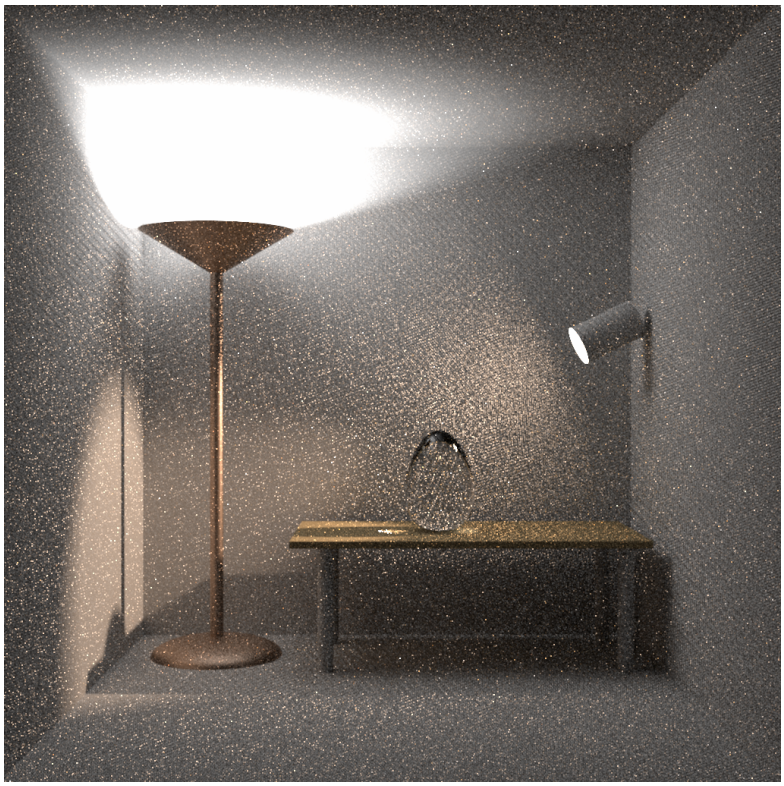
$$Q(s,a) := (1-\alpha)Q(s,a) + \alpha\left(L_e(p_{n+1} \to p_n) + \sum_{a'} p_{sec,n+1}(a')Q'(s_{n+1}, a')\right) \quad (4.1)$$

4.3 Implementation

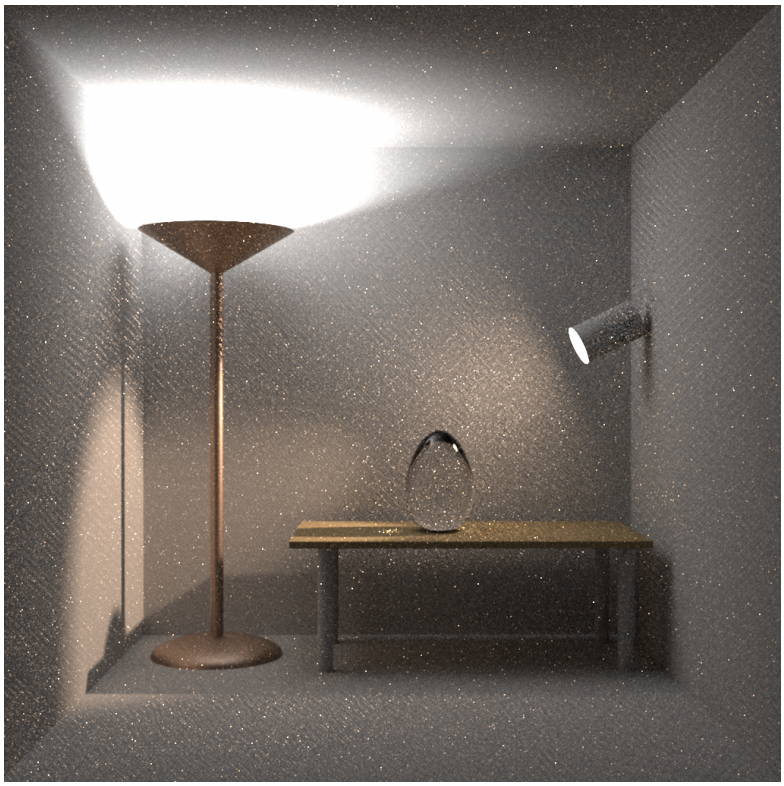
When implementing algorithm 7 on GPUs, different threads could operate on different grid cells. However, in CUDA, if all threads within the same warp read the same memory address, the hardware can *coalesce* the memory operation by broadcasting the same data content to all threads. To utilize this fact, this project included an additional sorting phase, which groups threads together according to the cell s_n that they operate on (similar to the material sorting stage described in section 3.3). It was observed that this improves the performance of computing the weighted Q values by around 50%.

Further, when formula 4.1 is used to update the Q table, the entry $Q(s, a)$ could be updated simultaneously by multiple threads. To avoid race conditions, this project accumulates proposal Q values using atomic operations, and then divides by the total amount of proposals in a later step. This ensures that all GPU threads cooperatively contribute to learning.

To illustrate the benefit brought by reinforcement learning, the figure on the next page compares the same scene rendered by traditional path-tracing and with reinforcement learning guided path-tracing. Both images are rendered with 64 samples per pixel, but the difference in noise levels between them is blatant.



(a) Traditional path tracing



(b) Reinforcement learning path tracing

Figure 4.1: Impacts of reinforcement learning. Both images rendered with 64 *spp*.

5 Results

In order to demonstrate its efficiency, the `Cavalry` renderer created in this project is compared with two of the most famous renderers used in academia:

- The `pbrt` renderer, version 3. This is a comprehensively documented and optimized CPU renderer, and it is accompanied and documented by the "PBRT" book [17]. At the time of writing, a 4th version of `pbrt` is under development. The 4th version will include GPU support, but due to stability issues, the beta version couldn't be used for comparison with `Cavalry`.
- The `Mitsuba` renderer. This is another well-known renderer with full GPU support. The renderer uses NVIDIA's Optix library for intersection detection, which is considered to have state-of-the-art performance.

The `Mitsuba` and `pbrt` renderers are compared with `Cavalry`'s implementation of both the traditional path-tracing and the reinforcement learning path-tracing algorithms. Notice that, `Mitsuba` and `pbrt` also implement other variants of path-tracing (e.g., bi-directional path-tracing), but only the traditional path-tracing integrator of these renderers are used for comparison.

The scenes used to compare these renders are created by Benedikt Bitterli, and they were downloaded from [2]. The creator defined these scenes for both `pbrt` and `Mitsuba`, with identical geometries and near identical materials. The `Cavalry` renderer created in this project parses the same input files as `pbrt`. It is thus ensured that all renderers are assigned the same task[1].

For each scene, the renderers are asked to render images with an increasing amount of samples per pixel, starting from 4 and doubling up to 1024. The image with 1024 *spp* is used as a reference, and the other images are compared with the reference by computing the mean-squared-error (MSE) of pixel values. The MSE is plotted against the time spent on rendering on a log-log scale. Since MSE (i.e., variance) is inversely

[1]One potential source of unfairness is that when rendering on `Mitsuba`, a rather naive random number generator is used, because the better RNGs of `Mitsuba` do not allow flexibly setting *spp*.

proportional to samples count (which is linear to rendering time), the log-log plots usually take the shape of a straight line with negative gradient.

A total of 4 scenes are selected. For each scene, an image rendered by `Cavalry` is shown, as well as the plot of MSE against rendering time. Notice that in the legends, `path` represents the path-racing implementation by `Cavalry`, and `rlpath` represents its reinforcement learning path-tracing implementation.

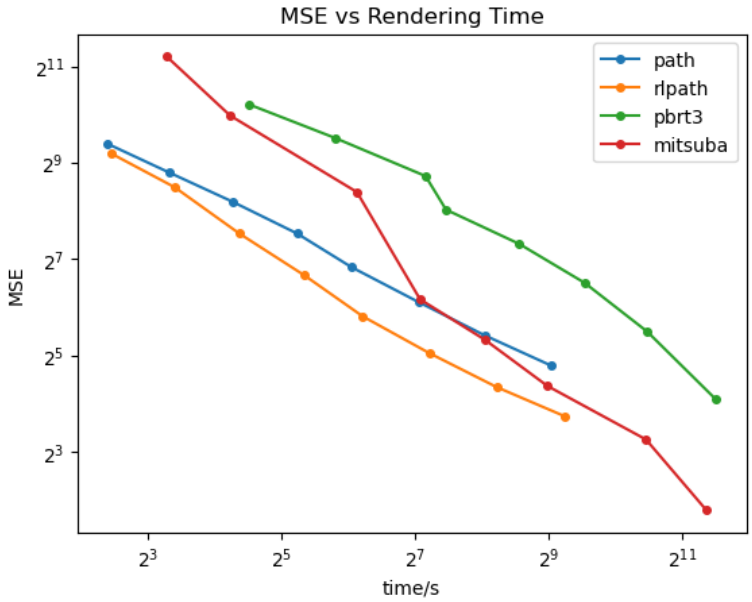


Figure 5.1: The "living-room-2" scene

As indicated by the plot, for this living room scene, the reinforcement learning integrator exhibits the greatest efficiency. `Mitsuba` is less efficient than `path` in the beginning, but overtakes at bigger sample counts. Naturally, all 3 GPU rendering algorithms outperform `pbrt`, which is CPU-only.

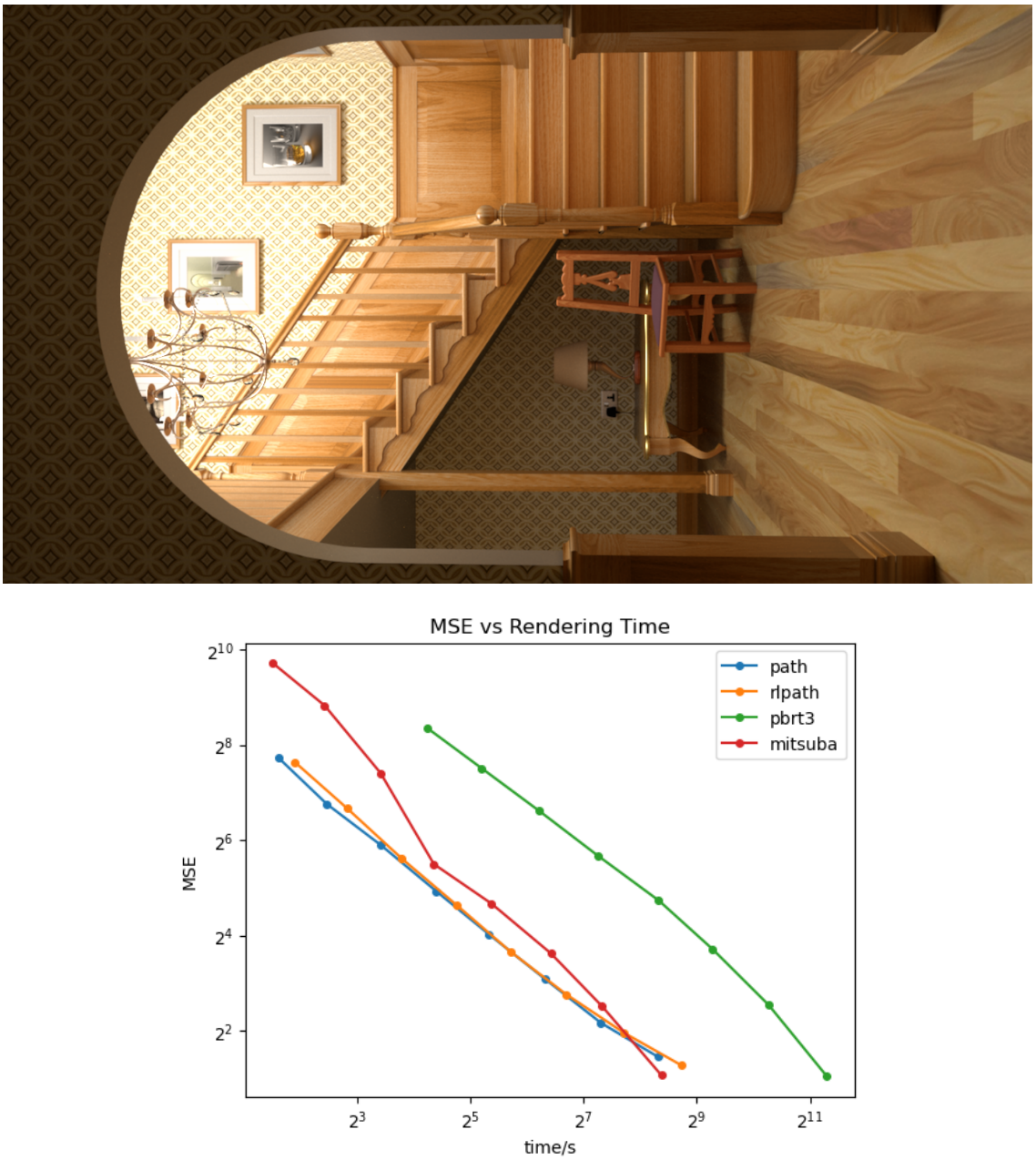


Figure 5.2: The "staircase" scene (rendered image rotated by 90 degrees)

For this staircase scene, the reinforcement learning path-tracing algorithm no longer has an advantage over traditional path-tracing. This is explained by the more forgiving lighting situation, where the majority of the scene (floor, stairs, wall in the back) is dominated by direct illumination from the light source. In contrast, in the living room, the light source is dimmer and the inter-reflections between objects are more important. Nevertheless, in the staircase scene, both integrators of `Cavalry` outperform `Mitsuba` (except at 512*spp*) and `pbrt`.

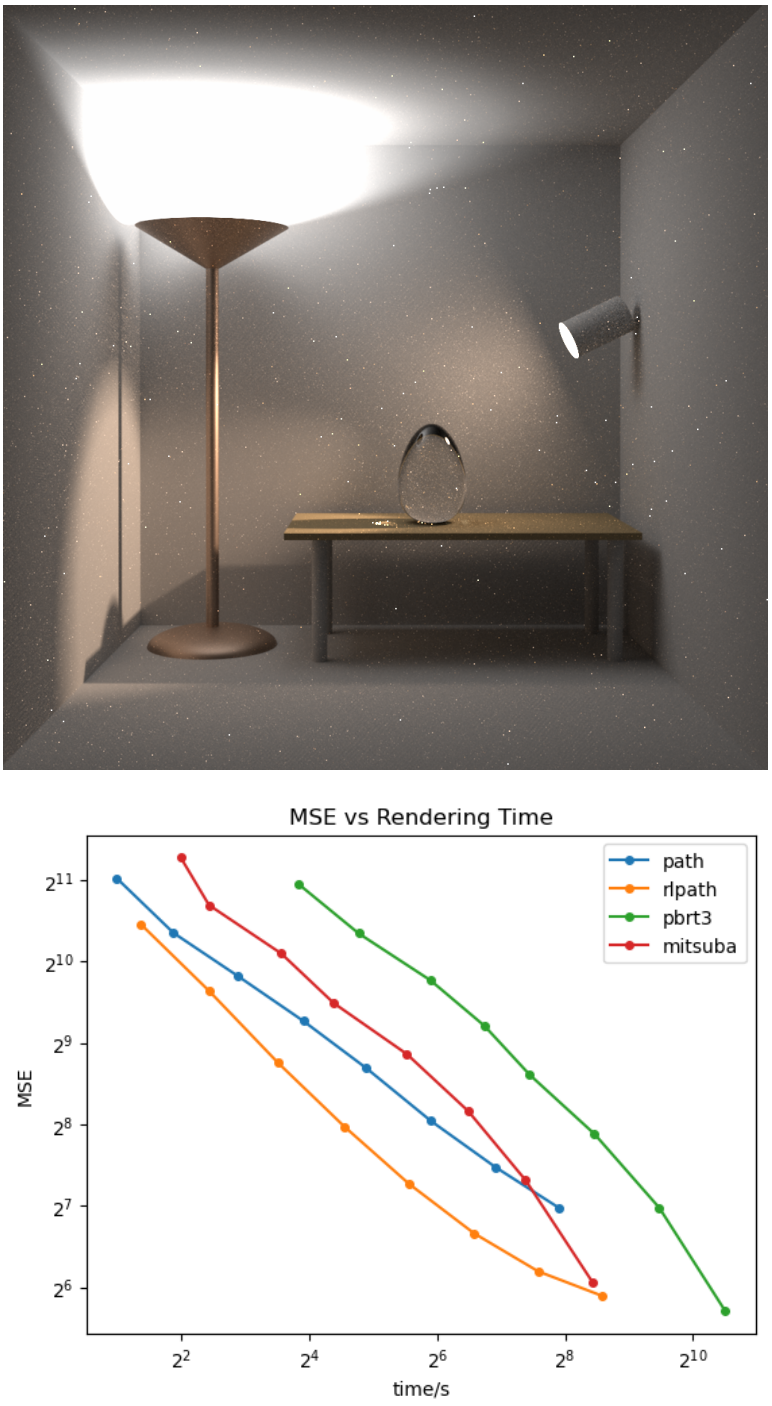


Figure 5.3: The "veach-bidir" scene

This is the scene with the difficult lighting scenario that was described in chapter 4. The advantage of `rlpath` over `path` (and the other two renderers) is the most significant here, because reinforcement learning can guide rays towards the bright region of dominant indirect illuminations.

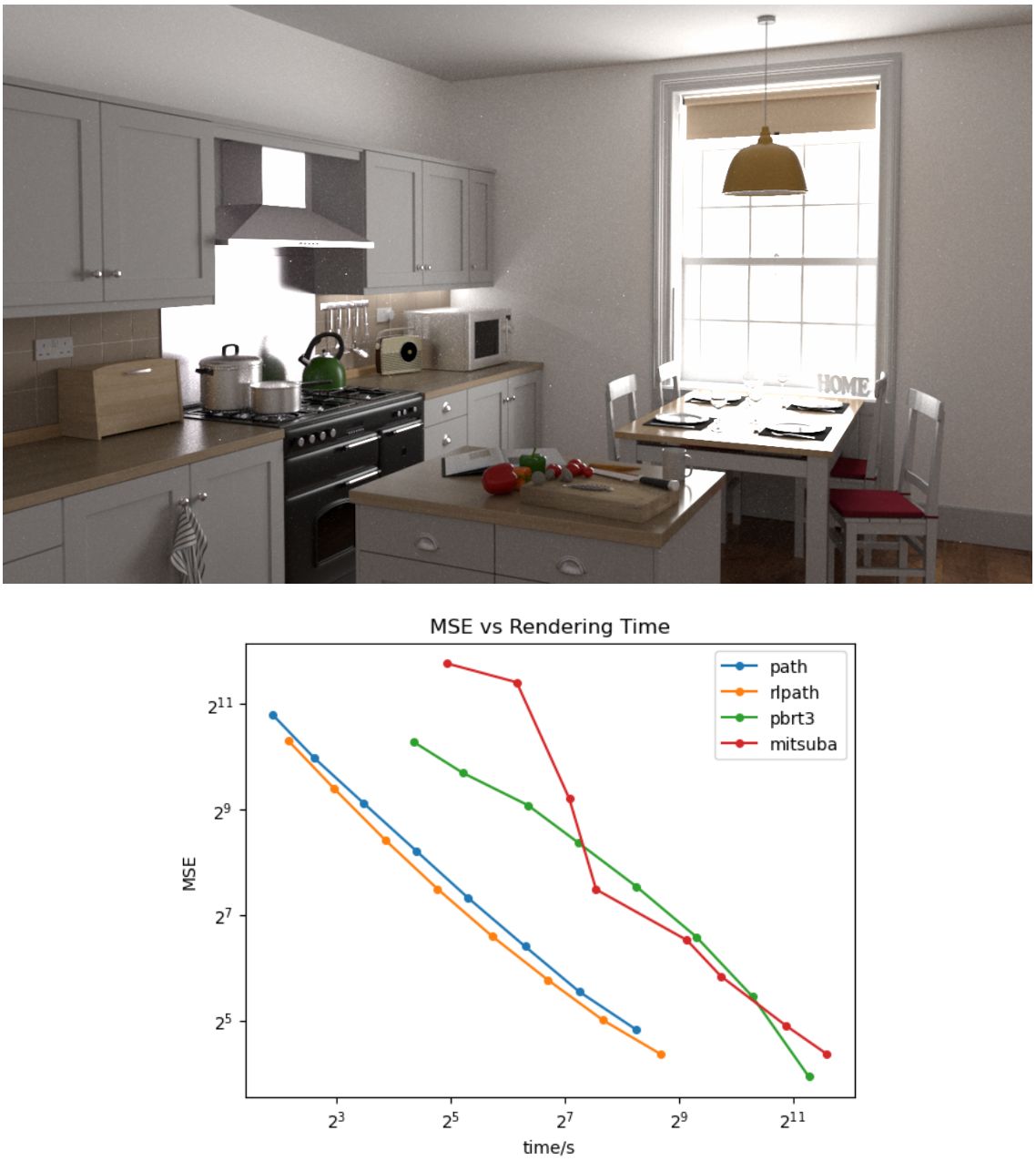


Figure 5.4: The "kitchen" scene

In this kitchen scene, the `rlpath` integrator slightly outperforms `path`, which significantly outperforms both `pbrt` and `Mitsuba`[2].

[2]Peculiarly, in this scene, the `Mitsuba` GPU renderer was even outperformed by `pbrt`. It is unclear as to why this happened.

To sum up, it was found that the performance of the `Cavalry` renderer created by this project significantly outperforms the `pbrt` CPU renderer, and in many occasions, it even surpasses the `Mitsuba` GPU renderer, which uses NVIDIA's state-of-the-art intersection detection library. Moreover, it was shown that except in scenes where direct lighting dominates, the reinforcement learning path-tracing algorithm prove to be more effective than the traditional path-tracing algorithm.

6 Conclusions

This project explored how to utilize GPUs for photorealistic rendering. The project implemented a parallel version of the path-tracing algorithm, as well as a variant of path-tracing where online reinforcement learning guides the generation of new rays. These algorithms are accelerated by a Bounding Volume Hierarchy system, where state-of-the-art BVH construction, optimization, and traversal procedures are implemented. The project demonstrates how these different algorithms can be combined and parallelized together to form an extremely efficient photorealistic rendering system. After around 200 commits and over 7 thousand lines of CUDA/C++ code, this project created the `Cavalry` renderer, whose full source code can be found at `https://github.com/AmesingFlank/Cavalry`.

6.1 Limitations & Future Work

This section begins by listing a few algorithmic aspects of `Cavalry` that are fairly important, but were not described in detail in this report due to the word-count limit:

- Quasi-random number generation:

  This project implemented a carefully designed random number generator, which has the *low-discrepancy* property and is specially suited for generating random samples used by a Monte Carlo integrator.

- Physically based BRDFs:

  In order to produce realistic images, it is vital that BRDFs of different materials be implemented in a physically correct manner. A family of BRDFs known for their physical accuracy are the Microfacet BRDFs [5], which are well implemented by this project.

This section also lists a few valuable features that would make `Cavalry` more complete and powerful as a renderer. Even though these features not yet supported, they can be added in future endeavors.

- Sub-surface scattering

  For most surfaces, when a light ray hits the surface at a point, the reflected rays start at the exact same point. However, this is not true for all objects in the real world. There are certain materials (e.g., jade) such that light enters the object at one point, and leaves the object at a slightly different point. These materials are described by bi-directional scattering-surface reflectance distribution functions (BSSRDFs), written as $S(p_o, w_o, p_i, w_i)$. The additional argument p_o to this reflectance function brings extra complexity to the renderer, but also enables it to capture a more complete range of real world objects.

Figure 6.1: Dragon model with sub-surface scattering. Rendered using `pbrt`.

- Participating media

  In the real world, the media through which light travels can sometimes scatter, emit, or absorb light. Modelling these interactions are essential for the rendering of phenomena such as fire and smoke.

- Other variants of path-tracing.

  Apart from the reinforcement learning integrator, many other variants of path-tracing exist, each with its own benefits. Two of the most powerful versions are Metropolis Light Transport, which uses a Markov chain whose stationary distribution converges to an optimal path-sampling distribution, and Gradient Domain Path-Tracing, which accelerates rendering by also estimating the difference between adjacent pixels. Implementing these algorithms would allow the `Cavalry` renderer to more robustly support a broader range of scenes.

6.2 Personal Reflections

Working on this project has been the most intensive intellectual endeavor I have made in my student career. Building the GPU renderer from scratch was a thorough training of comprehending and reproducing academic research, and it tested my abilities of organizing a complex code base. In the end, apart from the improvements mentioned in section 6.1, which I wish I made more progress on, I am in general very happy with and proud of the outcomes of this project.

One aspect of this project that I particular enjoyed is that, because physically based rendering is an extensively studied field, there is always a plethora of papers to refer to. I particularly enjoyed comparing academic results on the same topic from different years, and seeing how peoples' understanding of a problem progresses. This process gives me a comprehensive view of each algorithm: why does it work, what is it inspired from, and why has it been superseded by other approaches. These papers reflect just how much rendering technologies have improved over the past few decades, and it makes me excited about future developments of computer graphics.

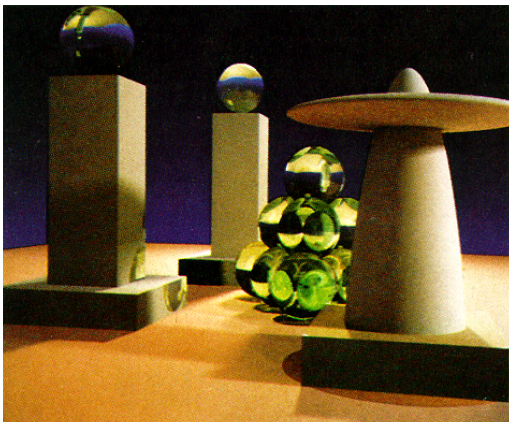


Figure 6.2: The image included in the original paper [10] that introduced path-tracing

The huge amount of existing rendering research can also be slightly frustrating once in a while, because it means it is difficult for me to devise novel algorithms. When implementing my renderer, there were quite a few times when a new idea of accelerating rendering emerged in my brain. On every occasion, a quick search on google scholar would lead to a well-written and in-depth paper that analyzes my "novel" proposal. As an example, I noticed that there are often regions of a rendered image that are significantly more noisy than other regions. Thus, I realized that if the renderer spends more time on these regions and less time in others, the quality of

the image could improve with no extra cost. Of course, I soon discovered an ample amount of research that proposes all kinds of adaptive sampling strategies (e.g., [18]).

Apart from the algorithmic side of rendering, this project also motivated me to put a lot of thought into writing well-structured and scalable code. The `Cavalry` renderer is designed so that new types of materials, geometries, and light sources and be added without any changes to the core path-tracing routine, and simultaneously, new variants of the path-tracing algorithm can be added without having to rewrite the code for these scene definition components. To achieve this kind of flexibility, the renderer underwent quite a few major refactors.

One observation that I made was that on some occasions, pursuing the scalability of the software sometimes means sacrificing performance or parallelizability. As an example, when organizing data, the array-of-structures pattern is often clearer and easier to modify, but the structure-of-arrays format enables better memory performance during parallelization. It was after many experiments when I settled into an architecture of the software that obtains a satisfying balance between these two aspects.

The completion of this project marks the end of 4 years of study at the computer science department of Oxford. Much like the project itself, my experience at Oxford had be delightfully challenging and extraordinarily rewarding. When I look back to the this project in the future, it will for sure be a pleasant reminder of the time I spent studying a field that I genuinely love.

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] Benedikt Bitterli. Rendering resources, 2016. https://benedikt-bitterli.me/resources/.

[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] Robert L Cook and Kenneth E. Torrance. A reflectance model for computer graphics. *ACM Transactions on Graphics (ToG)*, 1(1):7–24, 1982.

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

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

[8] Jeffrey Goldsmith and John Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications*, 7(5):14–20, 1987.

[9] 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.

[10] 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.

[11] Tero Karras. Maximizing parallelism in the construction of bvhs, octrees, and k-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.

[12] 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.

[13] 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.

[14] 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.

[15] Samuli Laine, Tero Karras, and Timo Aila. Megakernels considered harmful: Wavefront path tracing on gpus. In *Proceedings of the 5th High-Performance Graphics Conference*, HPG '13, page 137–143, New York, NY, USA, 2013. Association for Computing Machinery.

[16] 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.

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

[18] Fabrice Rousselle, Claude Knaus, and Matthias Zwicker. Adaptive sampling and reconstruction using greedy error minimization. *ACM Transactions on Graphics (TOG)*, 30(6):1–12, 2011.

[19] 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.

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

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