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**

1.1 Motivation . . . 1

1.2 Related Work . . . 1

1.3 Project Outline . . . 3

2 Physically Based Rendering **5**

2.1 The Rendering Equation . . . 5

2.1.1 Radiometry . . . 5

2.1.2 The BRDF . . . 7

2.2 Monte Carlo Integration . . . 8

2.2.1 Importance Sampling . . . 9

2.2.2 Multiple Importance Sampling . . . 10

2.3 The Path Space Formulation . . . 11

2.4 The Path-Tracing Algorithm . . . 13

3 Implementing Path-Tracing **15**

3.1 The CUDA Programming Model . . . 15

3.2 Wavefront Path-Tracing . . . 16

3.3 Polymorphism . . . 17

3.4 Bounding Volume Hierarchy . . . 20

3.4.1 BVH Traversal . . . 21

3.4.2 BVH Quality Measure . . . 22

3.4.3 Linear BVH Construction . . . 24

3.4.4 BVH Optimization . . . 28

4 Reinforcement Learning Path Tracing **30**

Bibliography **31**

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[19] 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[9]. 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][17][8] 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[10][11][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[18], Metropolis light transport[18][12], energy redistribution path-tracing[4], and gradient-domain path-tracing [13]. 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[16] 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[15], 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.

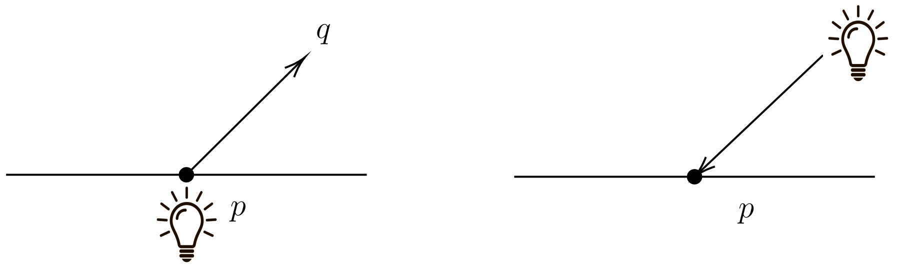


Figure 2.1: Example: consider two points p, q, and $\omega = \frac{q-p}{|q-p|}$. On the left, the radiance sent from p to q is $L_o(p, \omega)$. On the right, the radiance received by p from q is $L_i(p, \omega)$.

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 $\cos\theta_i$.

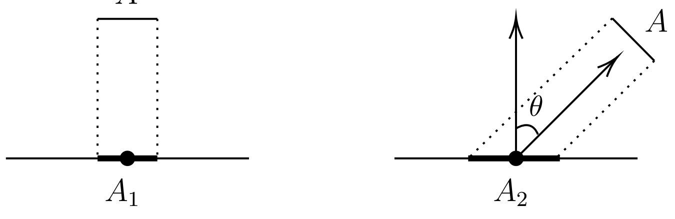


Figure 2.2: Example: on the right, the differential area A_2 illuminated by the ray is larger, because the ray is not perpendicular to the plane.

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.

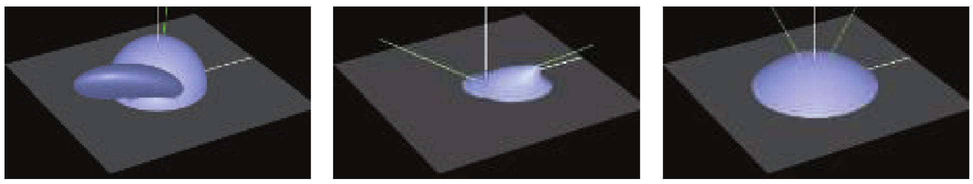


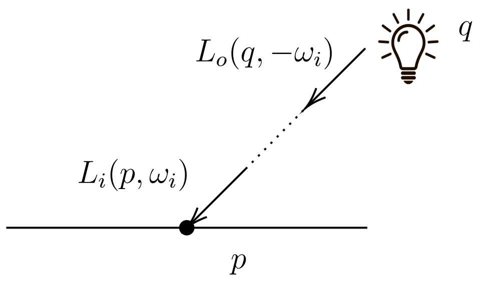
Figure 2.3: Shapes of example BRDFs. Image credit[2]

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[9].

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[16]. 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 [16].

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 $p_1, ..., p_n$, whose throughput is T_n, the algorithm does the following:

Algorithm 1: Incremental Path Construction

```
1 Samples a point $p_{light}$ from a light source.;
2 Test whether the ray $p_{light} \to p_n$ is occluded by some geometry. If not, then
    this generates a path of $n$ reflections, whose radiance is to be accumulated;
3 Sample a direction $\omega_i$ from a distribution that matches the BRDF at $p_n$;
4 Trace the ray with origin $p_n$ and direction $\omega_i$. Let the intersection with the
    scene be $p_{n+1}$;
5 If $p_{n+1}$ is on a light source, then again this leads is a path of $n$ reflections,
    whose radiance is accumulated;
6 Let $p_{n+1}$ be the next reflection point;
7 Let $T_{n+1}$ be $T_n f_r(p_{n+1} \to p_n \to p_{n-1}) G(p_{n+1}, p_n)$;
```

Notice that, step 2 and 5 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 $L_e(p_1 \to p_0)$;
7          for $n$ from 1 to $MaxDepth$ do
8              Incrementally construct two further paths, using algorithm 1;
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

Path-Tracing is an extremely costly algorithm. In a common rendering task, there're around one million pixels, each of which requires roughly one thousand samples to eliminate noise. More over, each ray generated in the algorithm needs to tested for intersection with the entire scene, which often consists of millions of geometric primitives. In order to cope with this humongous amount of computation, this projects implements path-tracing with parallelization on GPUs, and utilizes a data structure known as Bounding Volume Hierarchy (BVH) for accelerating intersection tests.

This chapter begins by giving a short introduction to GPU computing, and then explains the GPU parallelization of the path-tracing algorithm. Some of the highly impactful GPU optimization techniques employed are then described. Finally, the chapter discusses the state-of-the-art BVH construction algorithms implemented in this project.

3.1 The CUDA Programming Model

Originally built for real time rendering, GPUs are designed to handle a massive amount of geometries and pixels in parallel. The ability to do massively parallel computation motivated GPGPU (General-Purpose GPU) programming models to arise, which became significantly useful for scientific computing purposes. 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 GPU threads can be spawned simultaneously, each running the same sequence of code on different sets of data. GPU threads in CUDA 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. However, different warps do not need to remain in sync. When the threads within a warp access the memory, the entire warp can be paused and swapped out, so that a different warp can start executing 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 (known as a CUDA core) 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 multiprocessors 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 GPU programming, a parallel GPU function that is invoked by the CPU is often referred to as a GPU *kernel*. When implementing algorithm 2 on GPUs, the most straightforward method would be to implement the loop body (lines 3 to 9) as a GPU kernel, and invoke it for all pixel samples in parallel. This pattern of programming, where the entire CPU computation is embodied in a single GPU kernel, is called the *Megakernel*[14] approach.

In contrast to megakernels, the *Wavefront* programming divides the computation into many smaller GPU kernels, where intermediate results are explicitly stored in memory. The CPU then invokes these kernels one after another. On GPUs, regions of code that uses a large amount of registers or has 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 single kernel, 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 3: Wavefront Path Tracing

```
1  foreach pixel sample in a parallel kernel do
2      generate the initial ray 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      foreach newly found $p_n$ in a parallel kernel do
7          Sample a point $p_{light}$ on a light source;
8          Test whether the ray $p_{light} \to p_n$ occluded by some geometry;
9      foreach unoccluded $p_{light} \to p_n$ in a parallel kernel do
10         Evaluate the BRDF at $p_n$, and accumulate the radiance of this path;
11     foreach $p_n$ in a parallel kernel do
12         Sample a direction $\omega_i$ from a distribution matching the BRDF at $p_n$;
13         Generate a new ray, originated at $p_n$ and travels in the direction of $\omega_i$;
```

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

3.3 Polymorphism

When implementing a GPU path-tracer, an unexpected source of performance degradation is the need for dynamic-dispatch polymorphism. In CPU programming, dynamic-dispatch is easily implemented via virtual functions, but this can be costly for GPUs for the following 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, 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.

3. The virtual tables for GPU code and CPU code are stored at different addresses. Thus, extra care needs to be taken when objects are copied from CPU to GPU, or vice versa.

The need for polymorphism in path tracing mainly comes from two components. Firstly, the scene is often defined from a range of different geometric primitives (triangles, spheres, disks, etc.), and each primitives requires a different intersection detection procedure. Secondly, there is a material system, where each object could be associated with a certain surface material, and each material requires its own implementation of BRDFs. 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 its 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 can could in fact be plastic, metal, or glass:

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

Implementation wise, a `variant` class is implemented by *tagged union*, 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 labelled attached to the object. This avoids the extra memory operation incurred by dereferencing virtual pointers, and removes the problem of inconsistent virtual table locations across the CPU and the GPU.

Although the usage `variant` solves problem 2 and 3, problem 1 still remains, because control flow divergence still exists whenever threads in the same warp are processing surfaces of different materials. To solve this problem, this project employs the method of [14], which sorts the material evaluation tasks before they're executed. More precisely, in algorithm 3, 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 guarantees that threads that work on the same material are

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

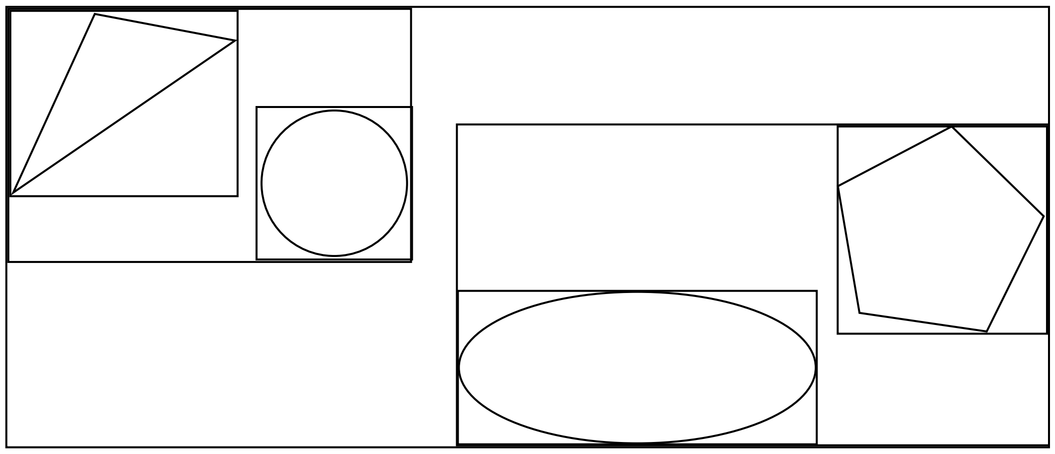
grouped together, and there will only be a few warps that experience any divergences. Even though the sorting incurs a small extra cost, it was observed that material sorting considerably improves the overall rendering performance.

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 3. 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 has two disjoint children sets, and the union of the children sets is equal to the interior node. 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 a total of 4 geometric primitives, each of which occupies a single box as a leaf node. One AABB groups together the triangle and the circle, and another AABB groups the ellipse and the pentagon. These two AABBs are then grouped together by the root AABB, which encloses all primitives.

3.4.1 BVH Traversal

BVHs can significantly accelerate ray-scene intersection tests, because once it is known that an AABB doesn't intersect the ray, all descendent boxes and geometries can be pruned from the search. This motivates the following algorithm, which finds the first intersection point between a ray and a scene by traversing the BVH tree in a recursive top-down manner:

Algorithm 4: Recursive BVH Traversal

```
Function FindFirstIntersection($node$, $ray$):
    if $node$ is a leaf node then
        if $ray$ intersects with the geometry of $node$ then
            return the intersection between $ray$ and this geometry
        else
            return “No Intersections”;
    Test for intersection between the ray and the AABB of $node$;
    if $ray$ does not intersect the AABB of $node$ then
        return “No Intersections”;
    Recursively, call FindFirstIntersection($node.leftChild$, $ray$) and
     FindFirstIntersection($node.rightChild$, $ray$);
    if both children do not intersect $ray$ then
        return “No Intersections”;
    else
        return the shortest intersection with one of the children nodes;
```

Despite its simplicity, this recursive traversal algorithm is unsuitable for a GPU implementation. Firstly, GPUs have a very restrictive recursion depth limit (for CUDA, this is 24), and thus might not be able to complete the traversal. More importantly, 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 the elimination the recursion, this project also makes two important observations that leads to a fruitful optimization. Firstly, the project notices that it is unnecessary to push a node onto the stack, if it is known that this node can only produce an intersection that is further than the current shortest intersection found. Secondly, the project notices that the intersection distance between the ray and an AABB serves as a lower-bound estimation of the shortest intersection distance between the ray and any geometry in the AABB. Motivated by these observations, the BVH implementation of this project makes the following optimizations:

- The distance of the shortest intersection found so far is recorded.
- 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 5: Recursive BVH Traversal

```
Function FindFirstIntersection($root$, $ray$):
    Create a stack $S$, with $node$ being the only element;
    Define $minDistance := \infty$;
    while $S$ is non-empty do
        Pop a node $n$ from the stack;
        if $n$ is a leaf node then
            if $ray$ intersects with the geometry of $node$ then
                Update $minDistance$ if this intersection is shorter than
                  $minDistance$;
        Compute $leftMinDistance$ as the minimum intersection distance
          between the left child of $n$ with the ray;
        Compute $rightMinDistance$ as the minimum intersection distance
          between the right child of $n$ with the ray;
        if Exactly one child has intersection shorter than $minDistance$ then
            Push the child into the stack;
        if both children have intersections shorter than $minDistance$ then
            Push the further child;
            Push the shorter child;
    return the shortest intersection found;
```

3.4.2 BVH Quality Measure

The previous subsection have focused on how a BVH is utilized to accelerate rendering, but the more difficult problem remains: how to construct high-quality BVHs efficiently.

BVHs differ in quality, and these differences are hugely impactful for performance. Firstly, a BVH is binary tree, so the balance of the tree is related to the cost of the traversal. However, the structure of the tree is hardly the only concern. Notice that in algorithm 5, its possible that for a ray r and a node n, r intersects with both children of n, and thus need to traversal both subtrees. 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 by Goldsmith and Salmon in [7]. The SAH heuristic is an estimation for the expected cost of single BVH traversal of 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 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[7]. 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[11]. This is not entirely surprising, considering the fact given any node with N geometries, there're $2^N - 2$ ways of constructing the two children nodes. However, not all hope is lost if the goal is to simply to construct a BVH that's not necessarily optimal, but still reasonably good. This project implements a two-step system to approach BVH construction:

1. Firstly, construct a BVH using a relatively naive heuristic: at each node, divide the geometries into two groups by splitting them at the mid-point of the AABB in one of the coordinate axises. This is known as a Linear BVH or LBVH, which can be constructed very efficiently on GPUs using the method of [10], but the tree created is of poor quality.

2. Secondly, optimize the previously created BVH tree, using the method of [11]. This step repeatedly solve the NP-hard optimization problem for very small *treelets*, which can still significantly improve the quality of the entire tree.

Details for each of the two steps are described below.

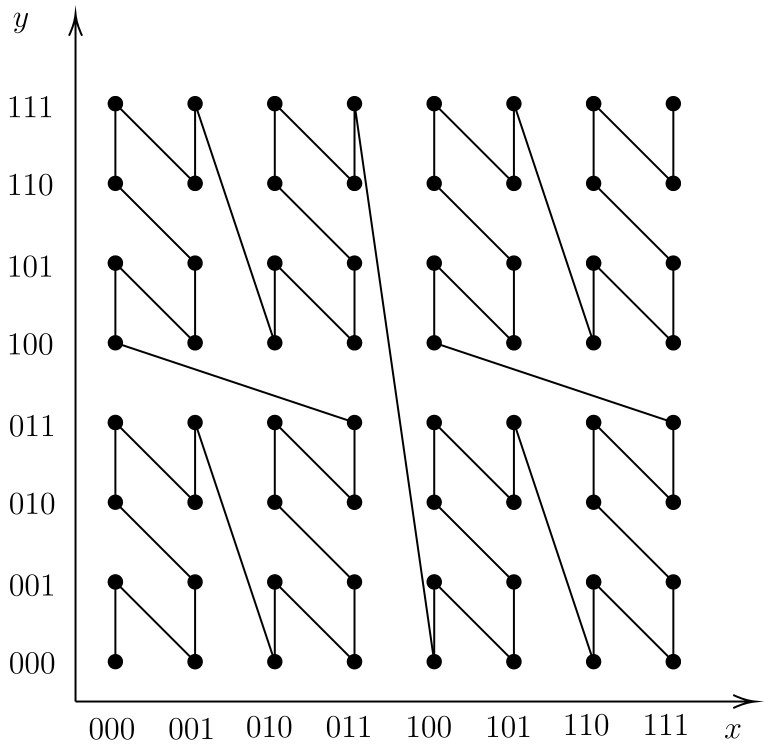
3.4.3 Linear BVH Construction

In LBVH, the geometries in each AABB are divided into two groups by splitting at the mid-point of the AABB in one of the coordinated axises. 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 divides each node using this heuristic. Unsurprisingly, this recursive algorithm parallelizes poorly. This motivates this project to employ a more sophisticated method described at [10], 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

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

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[10] implemented in this project uses a version of prefix tree known as the radix tree, which compactly supports 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 is the tree is also labeled by a bit sequence, which is the longest common prefix of all leaves in the subtree rooted at this interior node. Using $\delta(i,j)$ to denote the length of the longest common prefix between k_i and k_j, the ordering of leaves implies 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.

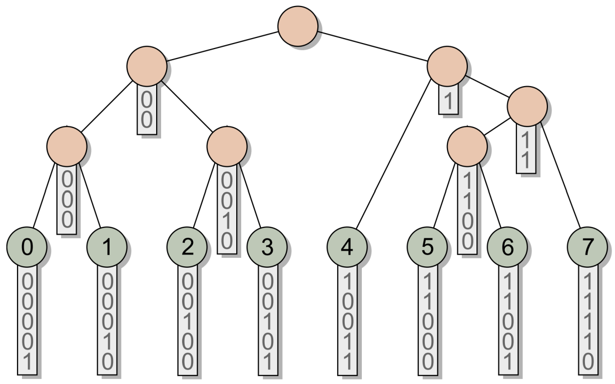


Figure 3.1: Example radix tree. Image credit [10]

In order to construct radix trees in parallel, the algorithm[10] establishes a connection between node indices and keys through a specific tree layout, which enables interior nodes to be built simultaneously with their children. The layout stores interior and leaf nodes in two separate arrays, I and L. The leaf array is sorted 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 enforces 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 vertical bar represents the range of keys covered by an interior node.

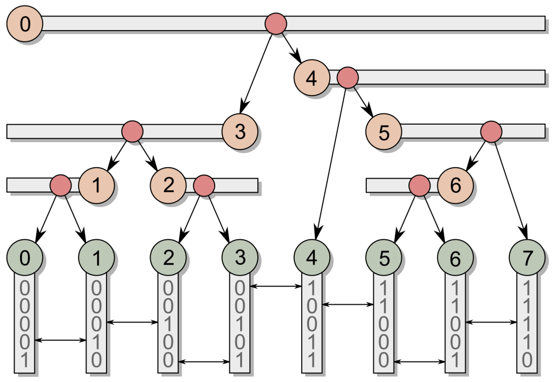


Figure 3.2: Radix tree storage layout. Image credit [10].

Given an interior node, it is not too hard to find out whether its index correspond 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 6: Parallel Binary Radix Tree Construction

```
1  foreach internal node $L_x$ in parallel do
2      $\delta_{\min} := \min(\delta(x, x+1), \delta(x, x-1))$ ;
3      if $\delta(x, x+1) > \delta(x, x-1)$ then
4          $i := x$;
5          Binary search to find the largest $j$ such that $\delta(i,j) > \delta_{\min}$;
6      else
7          $j := x$ ;
8          Binary search to find the smallest $i$ such that $\delta(i,j) > \delta_{\min}$;
9      Binary search to find the biggest $\gamma$ such that $\delta(i,\gamma) > \delta(i,j)$;
10     Left child is $I_\gamma$ as long as $i \neq \gamma$, and $L_\gamma$ otherwise;
11     Right child is $I_{\gamma+1}$ as long as $j \neq \gamma + 1$, and $L_{\gamma+1}$ 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 on GPUs using CUDA. Thanks to the massively 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) compare to the cost of the actual path-tracing phase.

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 that are generated with the guidance of the SAH heuristic. To address this issue, this project implements the algorithm described int [11], which optimize 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[11], however, the algorithm follows the intuition that if the tree is optimal in every local *treelet*, the entire tree would be somewhat optimal. Formally, a *treelet* of size n rooted at some node is defined to be a collection of immediate descendants of the root, consisting of $n-1$ internal nodes and n 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 of size 7, in hopes that these local transformations improve the structure of the entire tree.

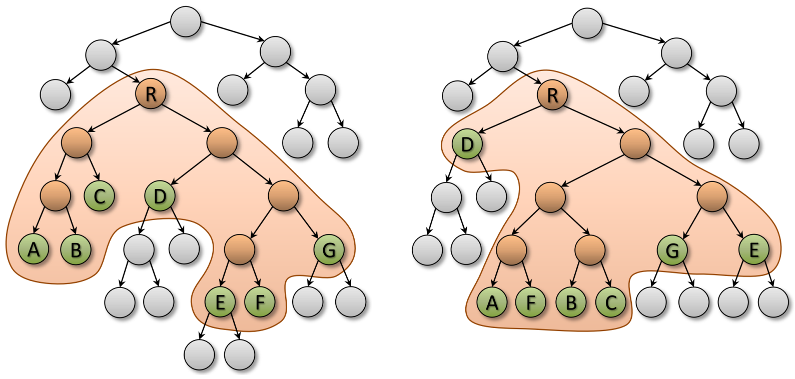


Figure 3.3: A treelet of size 7 and how it can be transformed. Image credit [11]

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 of size 7, the algorithm optimizes $C(n)$ by finding the optimal structure of a local treelet. If more than one treelet of size 7 is rooted at n, the algorithm operates on the one where the leaves have the greatest surface area, which maximizes the potential for improvements.

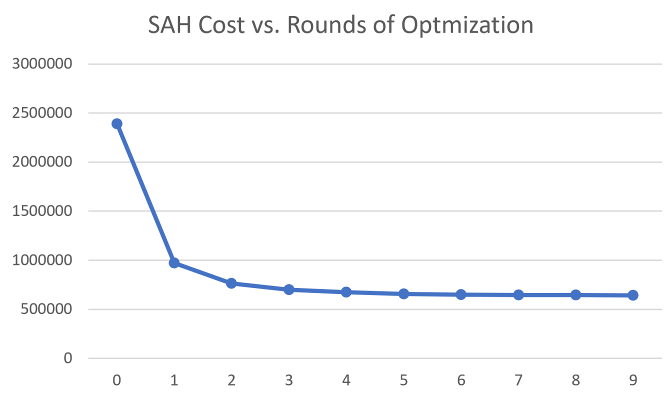
The algorithm optimizes treelets using dynamic programing. Specifically, it optimizes subsets of the leaves of increasing size, and memoize intermediate results. When working a subset of size n, it suffices to enumerate all possible binary partitions of this subset, and use memoized results to obtain 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 7: Treelet Optimization

```
1  Create memoization array $C_{opt}$ of size $2^7$, indexed by subsets of the leaves
     (empirically represented as bitmaps by a 7-bit integers);
2  foreach leaf node $n$ of the treelet do
3      $C_{opt}[\{n\}] := C(n)$;
4  for $k$ from 2 to 7 do
5      foreach subset $S$ of the leaves of size $k$ do
6          $C_{opt}[S] := \infty$;
7          $A(S)$ := surface area of the AABB that encloses all primitives in $S$;
8          foreach pair of disjoint nonempty subsets $U, V$, where $U \cup V = S$ do
9              $thisCost := A(S)C_i + C_{opt}[U] + C_{opt}[V]$;
10             if $thisCost < C_{opt}[S]$ then
11                 $C_{opt}[S] := thisCost$;
12                 Record $(U, V)$ as current best partition for $S$;
13 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 around 200ms for a scene with 1 million triangle. This cost is extremely low compared to the actual rendering time.

Interestingly, this optimization step does not reach a fixed point after optimizing each treelet once. The following image plots how the SAH cost (and thus rendering cost) changes when the BVH for a scene is optimized repeatedly. This project runs the optimization for 3 rounds for each scene, which almost always reduces SAH cost to one third of the origin value.



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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

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

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