Approaches to Accelerating Rendering Speeds

Author
Rami Khammash

School of Mathematics and Computer Science

A thesis presented for the degree of
MSc

Supervisor
Dr. Murdoch Jamie Gabbay

Heriot-Watt University
April 14, 2024

Declaration of Authorship

I, Rami Khammash, declare that this thesis titled, 'Approaches to Accelerating Rendering Speeds' and the work presented in it is my own. I confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Signed: RAMI KHAMMASH

Dated: 13/04/2024

Approaches to Accelerating Rendering Speeds

Rami Khammash

Abstract

Computational demands of rendering can be significant, especially for complex scenes with high-resolution geometric data and intricate lighting. This necessitates the development of efficient techniques to accelerate rendering speeds. In what follows, we will explore the theoretical foundations of rendering and various acceleration techniques. We will cover topics including rendering algorithms, shading calculations, acceleration data structure, optimizing ray/object intersection tests, and the impact of parallelism on rendering times. We will implement a rendering engine and investigate methods to improve its performance. We conclude by analyzing how our acceleration techniques improve the speed of our rendering engine.

Table of Contents

List of Figures

List of Tables

.

Abbreviations

BRDF	**B**idirectional **R**eflectance **D**istribution **F**unction
BVH	**B**ounding **V**olume **H**ierarchy
CPU	**C**entral **P**rocessing **N**nit
FOV	**F**ield **O**f **V**iew
GPU	**G**raphics **P**rocessing **U**nit
LBVH	**L**inear **B**ounding **V**olume **H**ierarchy
SAH	**S**urface **A**rea **H**euristic
SIMD	**S**ingle **I**nstruction **M**ultiple **D**ata
SPMD	**S**ingle **P**rogram **M**ultiple **D**ata
VFOV	**V**ertical **F**ield **O**f **V**iew

Chapter 1

Introduction

Rendering is the process of drawing 3D images on computers. It simulates light-object interactions for image creation. Rendering can vary in complexity, ranging from offline rendering for movies, where time is abundant and emphasis is on achieving visual fidelity, to real-time rendering for video games, where time is limited and thus a compromise in the visual quality is needed for rendering speed. There are plenty of rendering algorithms, and their introduction is delegated to §2.2.1, where a brief description of some of them can be found.

Regardless of the rendering algorithm used, the ultimate goal of rendering remains the same: to generate visually compelling images that engage the viewer. The evolution of rendering techniques has been closely linked with the advancement of both computer hardware and software. Today's rendering capabilities enable the creation of increasingly immersive and realistic digital experiences.

In rendering, there are four main performance goals we seek to achieve [Akenine-Moller et al., 2019]:

1. Increased number of frames per second.
2. Higher resolution.
3. Simulating physically based materials and lighting.
4. Increased geometric complexity.

We will explore how to realize these goals in our research.

GPUs are becoming increasingly powerful with each new generation. This has allowed them to handle an astronomical[1] number of calculations per second, paving the way for programmers to leverage them for intensive computing. In light of the rapid advancements in hardware, the possibility of rendering complex scenes on consumers' computers is more realizable than ever. In what follows, we will explore approaches that attempt to address this goal and delve into how research has advanced over the years.

[1] Billions of floating-point operations per second [Pharr and Fernando, 2005]!

Figure 1.1: *Hyperion* is a physically-based rendering system developed by Walt Disney Animation Studios. Hyprion uses path tracing (see §2.2.1.5) as its main rendering algorithm. Disney used Hyperion to render many of its recent feature-films, including *Moana* (bottom left) [Burley et al., 2018].

1.1 Motivation

Accelerating rendering has been a subject of research long before the capabilities of graphics hardware were understood. Researchers posed several questions, like: How can we render frames in as few seconds as possible? How can photorealistic graphics be achieved in a realistic time? And so on...

Given unlimited resources and unlimited time, some rendering algorithms can converge to the ground truth result. The problem was that users neither have unlimited time nor unlimited resources. Therefore, the shift was toward utilizing the CPU and hoping that Moore's law would come to the rescue, enabling future chips to catch up with the time-complexity of rendering algorithms. Furthermore, researchers have introduced approaches to speed up rendering times, utilizing techniques such as parallel processing and implementing acceleration data structures. Some of these approaches stood the test of time and are still prevalent in implementations of rendering engines. While algorithmic improvements paved the way, a breakthrough in hardware technology was the ultimate catalyst for achieving significant acceleration in rendering speeds. Exploring these acceleration methods is the motivation of our research.

1.2 Aims

The goal of this research is to explore techniques that accelerate rendering times. We also aim to compromise the visual quality loss that may ensue from optimizing some parts of the rendering pipeline. Speed is not the only goal of rendering; we also want to render production-quality images.

This research will also explore some of the background topics, specifically the shading models. These are important to understand. They do not directly contribute to the acceleration goal, but they ensure visual fidelity is realized. Furthermore, to effectively optimize shading calculations, a thorough understanding of shading principles is needed.

The aims of this research are summarized below:

1. **Aim 1**: Implement an engine capable of rendering primitives (spheres, boxes, triangles, etc...) with lighting and materials.
2. **Aim 2**: Enhance the engine's performance, primarily utilizing parallelism and acceleration data structures to achieve speed improvements.

1.3 Objectives

The objectives of our research are summarized in the following list, in order of priority:

1. Survey the literature to explore the background of rendering algorithms and investigate rendering acceleration schemes.
2. Write a rendering engine using C++. The rendering engine will be implemented from scratch (i.e., no rendering/graphics APIs will be used).
3. Enhance the engine's speed by implementing an acceleration data structure and parallelizing the engine's code via OpenMP.
4. Leverage GPU acceleration with CUDA to enhance the engine's performance.
5. Implement different heuristics to optimize the performance of the acceleration data structure.
6. Add support for rendering complex primitives (meshes).
7. Explore other optimization techniques to expedite the rendering engine.

Chapter 2

Literature Review

2.1 Introduction

In this chapter, we will conduct a literature review on rendering acceleration. We will summarize each paper and investigate how it relates to the research we intend to carry out. Furthermore, we aim to investigate gaps and future challenges that some of the research papers outline.

The outline of this chapter is summarized below:

- **Background:** This section focuses on exploring rendering algorithms and shading models.
- **Acceleration Structures:** This section explores acceleration structures designed to expedite rendering calculations and reduce computational overhead.
- **Optimizing Intersection Tests:** This section discusses the importance of optimizing ray/primitive intersection algorithms.
- **CPU Accelerated Rendering:** This section discusses how multithreading can expedite rendering.
- **CUDA and the GPU:** This section explores the use of CUDA for GPU-accelerated rendering.

2.2 Background

2.2.1 Rendering Algorithms

In this subsection, we review different rendering algorithms. We give a brief history of each, and discuss their advantages and limitations. We aim to highlight the differences between these algorithms as their terminology tends to be inadvertently used, and abused, in many instances. As part of our discussion, we will analyze how some algorithms build on their predecessors to optimize rendering speeds.

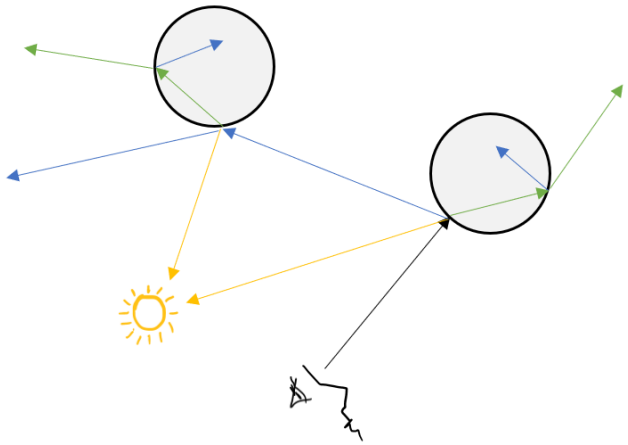


Figure 2.1: *Whitted Ray Tracing.* The eye shoots a ray towards the scene that bounces around, generating different types of rays.

2.2.1.1 Rasterization

The process of finding the pixels of an image that are occupied by a geometric primitive is called ***rasterization*** [Shirley et al., 2021]. Thanks to its speed, rasterization has long been used in real-time rendering. Rasterization requires the utilization of GPUs for shading purposes. This rendering technique has proven to be useful for displaying complex scenes. A recent study was able to render 2 billion points in real time [Schütz et al., 2022]. As we will see in the following subsections, there are rendering algorithms that are more photorealistic, albeit slower, than rasterization.

2.2.1.2 Ray Casting and Ray Tracing

The concept of ***ray tracing*** dates back to the 16$^{\text{th}}$ century when it was first described by the German Renaissance artist Albrecht Dürer [Caulfield, 2022]. Ray tracing addresses the problem of projecting 3D scenes onto a canvas. In the last century, computer scientists drew inspiration from this artistic concept and began digitally emulating it. Since then, especially with the advancement in the computing capabilities of the GPU, ray tracing has emerged as one of the predominant rendering algorithms (alongside rasterization). Ray tracing, as an algorithm, is straightforward: A virtual camera emits light rays from its position, passing through each pixel on the image plane, and these rays are "traced" as they interact with the scene. Whenever a ray intersects an object in the scene, the properties of that object are used to determine how light interacts with the object's surface, including determining visual effects such as color, reflectivity, and transparency.

Arthur Appel pioneered the use of computers to produce shaded images [Appel, 1968]. His key idea was to "*shoot random light rays from the light source at the scene and project a symbol from the piercing point on the first surface the light ray pierced.*" This algorithm became known as ***ray casting***. At an intersection point, ray casting generates two types of rays: the *view ray*, emitted from the camera's origin through the virtual screen, and the *shadow ray*, originating from the intersection point towards the light to detect any

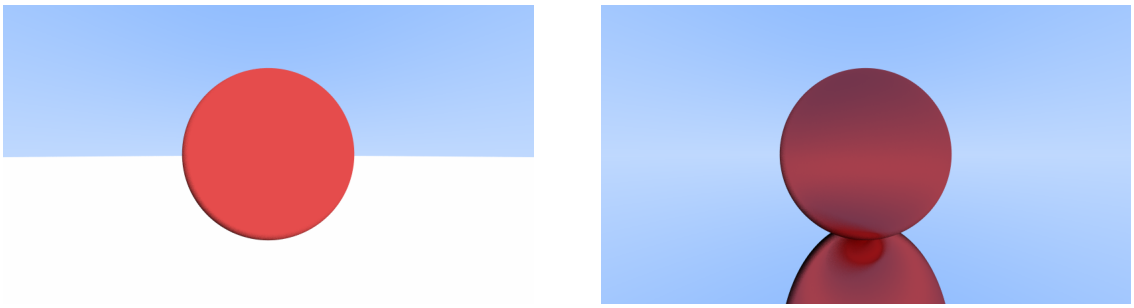


Figure 2.2: *Rendering a Specular Scene.* The left scene is rendered with no recursion while the right scene is rendered with a recursion of depth = 50. Notice that the mirror-like reflection is realized in the right image: this is the power of Whitted's deeply recursive ray tracing algorithm.

object along its path.

Although the terms *ray tracing* and *ray casting* are sometimes used interchangeably, they differ in one fundamental characteristic. Unlike in ray casting, a ray in ray tracing does not halt when it intersects an object; instead, it keeps bouncing around the scene, effectively "tracing" its attenuation as it rebounds between objects, until it eventually halts with a "final" color. This implies that *ray tracing* is the more expensive rendering technique, yet it is the more physically accurate of the two.

Turner Whitted was the first to employ recursion as a mechanism for ray tracing to make it a physically based rendering technique[1] [Whitted, 1979]. The idea is to create a tree of rays extending from the camera's origin to the first intersection point and from there to other intersection points and to the illumination source (see Figure 2.1). To avoid infinite recursion, a maximum recursion depth can be set. Recursion allows rendering materials that have a specular reflection (see Figure 2.2).

2.2.1.3 Distributed Ray Tracing

Distributed ray tracing is an idea that extends ray tracing [Cook et al., 1984]. Aliasing is a noticeable problem in images rendered by ray tracing. A key observation is that one can avoid aliasing by simply shooting more rays toward the scene and averaging their total contribution. This process is known as *multisampling*. Figure 2.3 shows the effect of multisampling in rendering. The essence of distributed ray tracing is that, without drastically changing the details of the ray tracing algorithm, one can also benefit from sampling to give rise to further visual effects like depth-of-field and motion blur.

Distributed ray tracing is one method to address aliasing; however, it suffers from a significant drawback: It needs many samples-per-pixel to render a noise-free images. While this may be acceptable for simple scenes, it would be impractical for complex ones. In §2.2.1.5, we will define what exactly rendering algorithms aim to solve, and we will discuss a rendering algorithm that relies on a more efficient sampling strategy, allowing for faster rendering.

[1] A physically based rendering technique aims to render realistic images by modeling the physics of light.

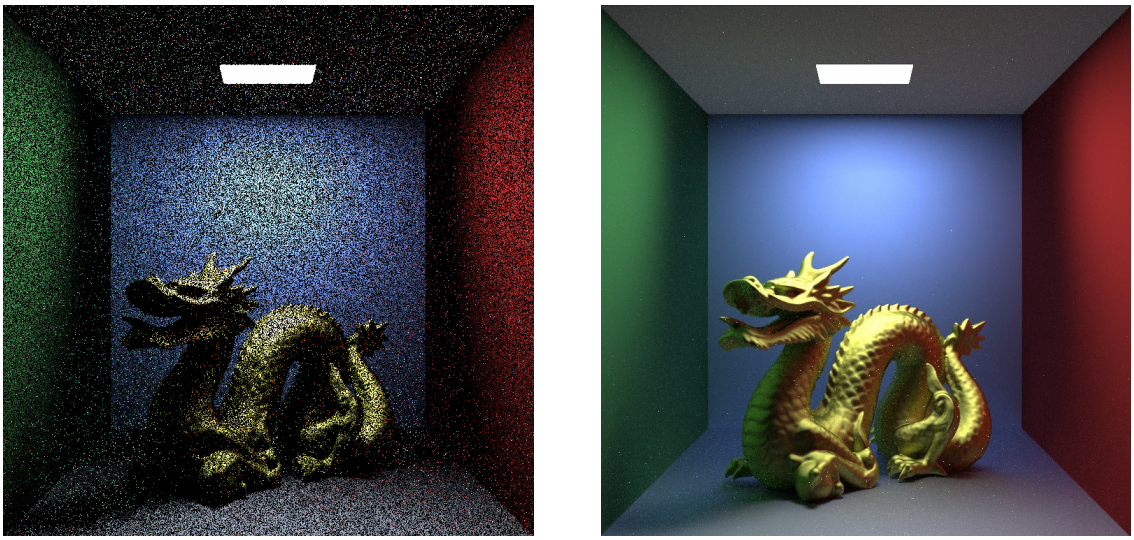


Figure 2.3: *Anti-aliasing via Multisampling.* The left scene is rendered without multisampling. The scene on the right is rendered using 3,000 samples-per-pixel. The difference in the noise level is noticeable.

2.2.1.4 Radiosity

One problem with traditional ray tracers is that they cannot simulate the reflection of light between diffuse surfaces. ***Radiosity*** is a technique that does this. It is yet another approach for rendering. The idea of radiosity is to divide the surfaces in the scene into very small areas, also called *patches*. The *radiance* from a point on a surface can be formulated in terms of patches; it is essentially the sum of the emitted and reflected energies at that point in a certain direction. The goal is to measure the amount of light reaching a certain point from all directions. This latter quantity is called the *irradiance*. Hence, if radiance is given, we can derive all other radiometric quantities. The equation of irradiance is given by [Shirley et al., 2021]:

$$H = \int_{all\,\mathbf{k_i}} L_f(\mathbf{k_i})\, cos\theta_i\, d\sigma_i \tag{2.1}$$

where:

H = irradiance
$\mathbf{k_i}$ = incoming direction
L_f = field radiance (i.e., a measure of the brightness of a surface in a particular direction)
θ_i = angle between $\mathbf{k_i}$ and the surface normal at the point of intersection
$d\sigma_i$ = differential solid angle associated with incoming light direction $\mathbf{k_i}$

Radiosity is not an all-encompassing solution to the rendering problem. It does not account for specular reflections, which makes it a poor-choice rendering algorithm for scenes that contain many objects that have specular surfaces. What is important about radiosity is that it fills a gap that ray tracing fails to address. In the next section, we will see how Equation 2.1 can be further extended to an equation that generalizes rendering algorithms.

2.2.1.5 Path Tracing

Ideal and perfectly diffuse surfaces do not exist in real life. All surfaces are made up of microfacets that differ in the way they reflect incoming light. Thus, it would be helpful to combine the mathematical formulation of radiosity with the way ray tracing handles different shading cases into a single rendering algorithm.

Kajiya [Kajiya, 1986] derived an integral equation that generalizes various rendering algorithms. Kajyiya's key observation was that all rendering algorithms attempt to model the physical phenomenon of light scattering off various types of surfaces. In consequence, we can view various algorithms as accurate approximation to the solution of a single equation.

The idea of Kajyia's equation follows. Consider the intersection point on a surface. To find the intensity of light leaving that point, we simply sum two quantities: the emitted light and the total light intensity scattered toward the intersection point. Kajiya's equation is known as the *rendering equation*, and it is shown below:

$$L_s(\mathbf{k}_o) = L_e(\mathbf{k}_o) + \int_{\text{all } \mathbf{k_i}} \rho(\mathbf{k}_i, \mathbf{k}_o) L_f(\mathbf{k}_i) \cos\theta_i \, d\sigma_i \tag{2.2}$$

where:

$\mathbf{k_o}$ = direction towards the eye
$\mathbf{k_i}$ = incoming direction
L_s = surface radiance (i.e., observed radiance)
L_f = field radiance
ρ = BRDF
$d\sigma_i$ = differential solid angle associated with incoming light direction $\mathbf{k_i}$
θ_i = angle between $\mathbf{k_i}$ and the surface normal at the point of intersection

The rendering equation does not have an analytical solution, because it is infinitely recursive. The light energy hits a surface from a ray that was bounced from a previous surface, which in turn was bounced from yet another surface, and so on. In the same paper Kajyia proposed an approximate solution to the equation motivated by the intuition that one can sample more rays toward the important directions (such as toward lights). This is known as *importance sampling*, and it is a relaxation that his proposed solution exploits.

Kajyia introduced a Markov Chain Monte Carlo algorithm to solve the rendering equation. A *Markov Chain* is a sequence of states generated through a random process, where each state is drawn from a probability space. In rendering, we can think of this space as the set of points on a surface. Instead of performing recursive ray tracing, and branching the ray tree at each intersection, we follow only one of the ray tree branches. This resembles tracing a path in the ray tree, and this is why this algorithm is called ***path tracing***. This solution significantly improves performance in scenes with many reflective and refractive surfaces (see Figure 2.4).

Figure 2.4: Pixar's first use of physically based rendering through path tracing was in its two movies: *Monsters University* and *The Blue Umbrella*, ©Disney/Pixar 2013 [Hery et al., 2013]. This scene is particularly *hard* to render, because of the presence of many light sources (neon signs, car headlights, traffic lights, etc...).

Unlike the case for offline rendering (as in movies), physically based Monte Carlo path tracing is not yet prevalent in the video game industry, where demand for real-time rendering is high. The problem with path tracing is its tendency to generate noisy images. One way to tackle this problem is to combine path tracing with the idea of distributed ray tracing (see §2.2.1.3), meaning we simply use multisampling to generate a less noisy image. Thanks to importance sampling, this approach usually requires significantly fewer samples-per-pixels compared to standalone distributed ray tracing. Consequently, this approach indirectly accelerates rendering by leveraging path tracing in conjunction with the idea of distributing the directions of the rays.

2.2.2 Illumination and Shading

The rendering algorithms presented in the previous section answer the *visibility* problem: determining what objects are visible to the observer. In rendering, our goal extends beyond verifying the visibility of three-dimensional objects on the screen; we also aim to give these objects a visual appearance. This requires the use of a *shading model*: A mathematical description of how an object's color varies based on several factors, like surface orientation (determined by the surface normal), view direction, and lighting. Hence, the aforementioned rendering algorithms solve even a more general problem, that of *global illumination*: Determining how to add more realistic light to a virtual scene.

To achieve accurate rendering of different surface materials, it is essential to understand how light interacts with them. Extensive research has been devoted to formulate this interaction using several shading models, with many studies recognized as fundamental to the field of computer graphics. Since we will implement

some of these models in our rendering engine, it is worthwhile to examine them.

We suggest referring to Figure 2.5 while reading this section, as it can be very helpful in understanding the notation used in shading.

2.2.2.1 Reflection Models

Lambertian reflection is the simplest shading model; it describes the behavior of light interacting with diffuse surfaces. It adheres to Lambert's cosine law, which states that the intensity of light (I) observed from rays scattering off diffuse surfaces is proportional to the cosine of the angle between the normal $\vec{N}$ to the surface and the observer's view direction. The ***Lambertian reflection model*** is:

$$I = surface\_color \times max(0, \vec{N} \cdot \vec{L}) \tag{2.3}$$

Blinn [Blinn, 1977] improved this model by introducing an ambient term. Ambient light is the indirect, uniform light that illuminates a scene's background. Thus, the perceived intensity function for a diffuse surface becomes:

$$I = ambient\_color + surface\_color \times max(0, \vec{N} \cdot \vec{L}) \tag{2.4}$$

See Figure 2.6 for a render of diffuse materials.

A diffuse material is *view independent*: An observer's perception of the surface's color does not depend on the direction from which she looks. Many surfaces in nature are not diffuse; they have some degree of shininess that produces highlights. We call such surfaces *specular*, and they produce *specular reflection*. Contrary to diffuse, a specular material is *view dependent*: Look at a lake from different directions, and you will see different highlights reflected. Equation 2.4 does not account for such surfaces. Blinn, inspired by Phong shading [Bui-Tuong, 1975], extended this equation to account for specular reflection. Blinn's idea was to produce a reflection that is brightest (i.e., perfect mirror) when the view direction $\vec{V}$ and the reflected light direction $\vec{L}$ from the intersection point are symmetrically positioned across the surface normal.

How do we exactly tell how close are we to a perfect mirror reflection? We simply evaluate the direction of maximum highlights represented by the bisector of the angle between $\vec{V}$ and $\vec{L}$. We call this vector $\vec{H}$; it is the bisector of the angle between $\vec{V}$ and $\vec{L}$.

By simple vector algebra, we can deduce that:

$$\vec{H} = \frac{\vec{V} + \vec{L}}{||\vec{V} + \vec{L}||} \tag{2.5}$$

A perfect mirror happens when $\vec{H}$ and $\vec{N}$ are parallel (that is, their dot product is 1). How dim or bright the specular component is thus depends on how far/close $\vec{H}$ is to $\vec{N}$, which can be mathematically obtained by their dot product $\vec{H} \cdot \vec{N}$.

Equation 2.4 becomes:

$$I = ambient\_color + surface\_color \times max(0, \vec{N} \cdot \vec{L}) + specular\_color \times max(0, \vec{N} \cdot \vec{H})^p \tag{2.6}$$

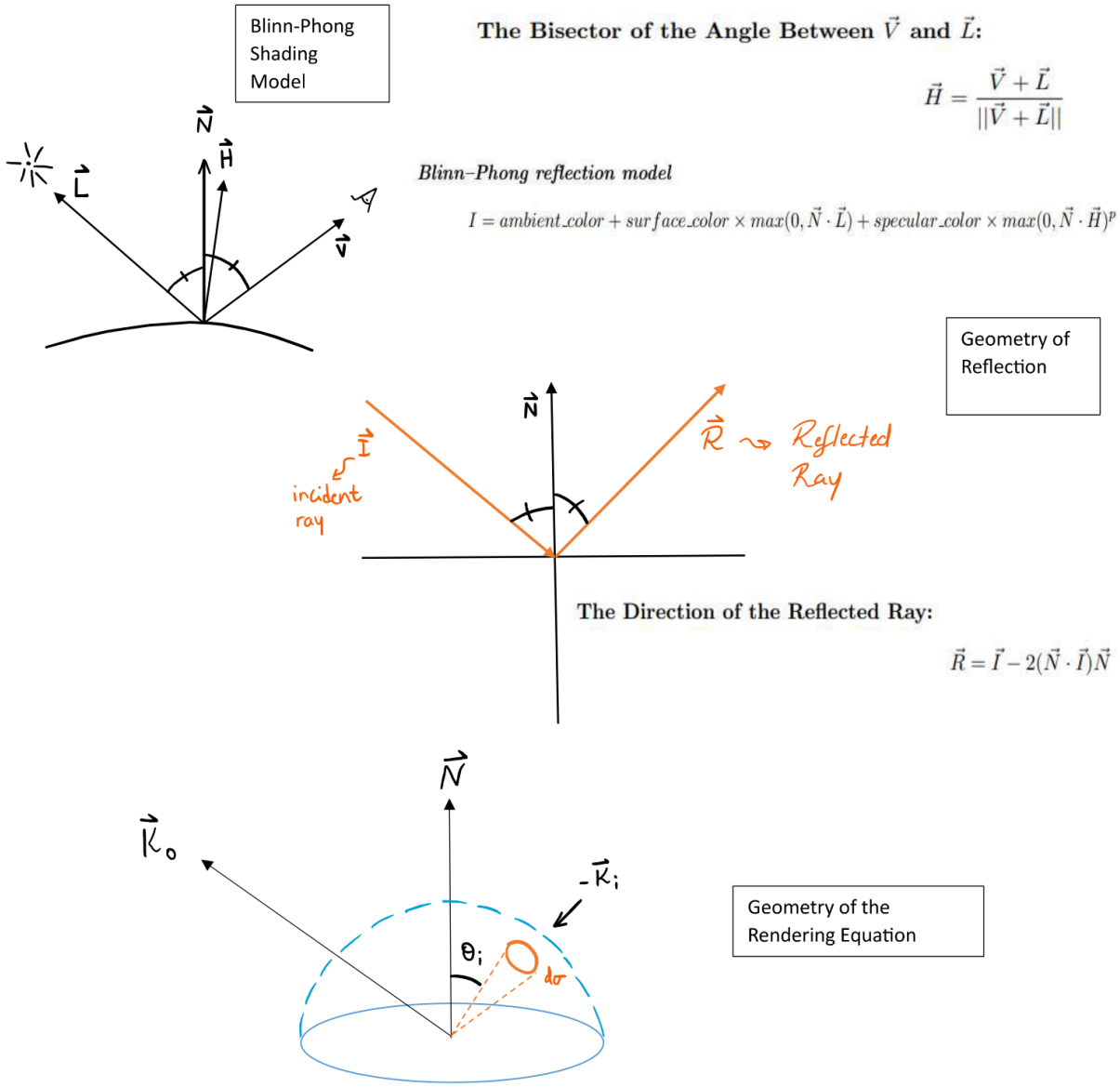


Figure 2.5: *The Geometry of Shading.*

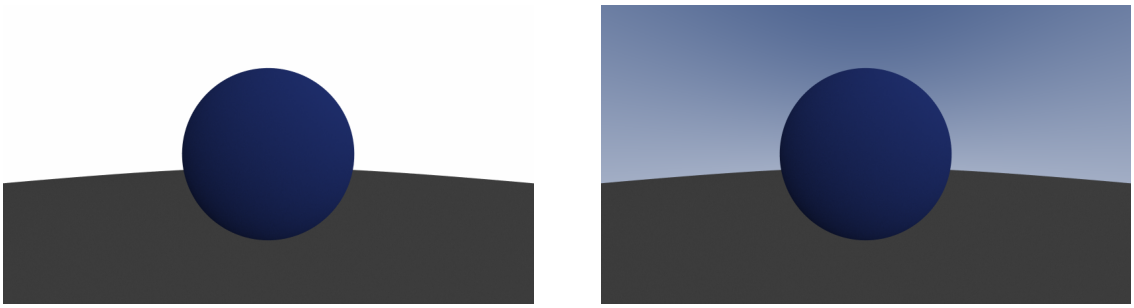


Figure 2.6: *Two Diffuse Scenes.* The left scene has a high ambient factor, while the scene to the right has a low ambient factor.

Equation 2.6 is known as the ***Blinn–Phong reflection model***. We note the presence of an exponent (p) in computing the specular component. Phong observed that large specular highlights may be created if the dot product between $\vec{N}$ and $\vec{H}$ is not raised to an exponent $(\vec{N} \cdot \vec{H})^p$ [Bui-Tuong, 1975]. This exponent is referred to as *Phong exponent*.

2.2.2.2 The Bidirectional Reflectance Distribution

The shading models we presented in §2.2.2.1 are special cases of a more general shading function. A general shading function is one that is able to characterize the way a surface reflects light, and it depends on two variables:

- $\mathbf{k_i}$: The angle between the incoming light and the surface normal.
- $\mathbf{k_o}$: The angle between the surface normal and the location of the observer - also known as the *outgoing angle*.

Such a function is called the ***Bidirectional Reflectance Distribution Function*** or *BRDF* for short. A BRDF is thus a function whose input is the direction of incident light and whose output is the direction of reflected light towards the observer.

$$BRDF(\mathbf{k_i}, \mathbf{k_o}) = \rho(\mathbf{k_i}, \mathbf{k_o}) \tag{2.7}$$

All the shading models that we discussed, and many others, are examples of BRDFs. Accurate numerical calculation of BRDFs is time consuming; therefore, in computer graphics, a BRDF is defined as a probability distribution function (PDF) that calculates the probability of the scattering of incoming light towards an outgoing direction [Lafortune et al., 1997].

2.3 Acceleration Structures

The most computationally demanding aspect of rendering is determining whether rays intersect with objects on the scene[2]. That number can easily be in billions for complex scenes and it is, in fact, linear in the number of objects in the scene (i.e., $O(N)$, where N is the number of objects in the scene). Optimizing this subroutine has been an area of ongoing research. Researchers have posed a key question: Since rays are repeatedly tested on all objects in the scene, why not adopt the idea of *binary search* to make the ray/object intersection routine sub-linear? The answer is embodied in the idea of *spatial data structures*.

2.3.1 Spatial Data Structures

To achieve sublinear runtime, we need a data structure to organize the objects in the scene so that we can traverse them efficiently. Such a data structure is called a ***spatial data structure***. Spatial data structures adopt two schemes [Shirley et al., 2021]: *object partitioning schemes* and *space partitioning schemes*. These two schemes are defined as follows:

- *Object partitioning schemes* group objects into a hierarchy by dividing them into disjoint groups, which may overlap in space.
- *Space partitioning schemes* partition space into disjoint regions. One object may end up overlapping more than one region.

2.3.2 Bounding Volume Hierarchies

If a spatial data structure is a container of objects, then a ***Bounding Volume Hierarchy*** (BVH) is a way of organizing these objects in space. Hierarchical organizations are favored because of the speed in which we can traverse them. Querying a binary search tree is an $O(log(N))$ operation. It is common to use *Axis Aligned Bounding Boxes* (AABBs) as bounding volumes for three primary reasons:

1. Boxes are easy to define and take little storage space. To define each box, only the two opposite corners are needed.
2. It is easy to construct the bounding box of a primitive.
3. The ray/box intersection test is fast to compute.

In §5.7, we explain how to implement a BVH with axis-aligned boxes as bounding volumes. A detailed survey of other bounding volumes can be found in Meister et al.'s paper [Meister et al., 2021].

2.3.3 Heuristics

To construct efficient BVHs, we need a rule to determine how to partition the bounding volumes into children nodes in the hierarchy tree. This rule is usually guided by a split ***heuristic***, which aims to

[2]For simple scenes, 75% of rendering time is spent on calculating ray/object intersections [Whitted, 1979].

balance the tree and make ray traversals more efficient, among many other things.

In their paper [MacDonald and Booth, 1990], MacDonald et al. described two heuristics for constructing space subdivisions that partition objects using binary trees (also known as *k-d trees*). Particularly, one of the heuristics provides an approach to k-d trees construction. MacDonald et al. presented a mathematical framework to derive some intersection estimates. These intersection estimates are then fed to an equation, called the *tree cost function*, to give an upper bound for the number of rays that intersect objects in the scene. This function, in turn, can be used to calculate the *object cost per ray*, a weight associated with testing a ray for intersection with an object in the scene. Subsequently, these equations can be used to construct a k-d tree with a minimum total cost, and this rule has come to be known as the *surface area heuristic* (SAH).

One problem with the SAH is that its calculation requires measuring the surface area of each node's bounding volume and estimating the associated costs that we discussed. This can make the implementation of SAH somehwat complex. Nonetheless, we will revisit this heuristic in §2.6.3, where we will discuss how BVHs are implemented on the GPU.

The following is a list of other heuristics that can be used to construct BVHs:

- Sorting the primitives along an axis and then dividing them into two sub-lists [Shirley et al., 2021]. Unlike the SAH, this heuristic is easy to code.
- Extending the approach stated above with an extra step: Choose axes more carefully each time. One way to accomplish this is to choose the axis so that the sum of the volumes of the bounding boxes of the two sub-trees in minimized. Compared to the heuristic above, this approach works better for chaotic scenes [Shirley et al., 2021].
- Constructing the hierarchies such that the two sub-tress contain approximately the same number of primitives [Shirley et al., 2021].
- Constructing a BVH incrementally by inserting objects on the scene into the hierarchy as soon as they are encountered. This approach is particularly useful in applications where we do not know the components of the whole scene in advance[3] [Bittner et al., 2015].

2.4 Optimizing Intersection Tests

Intersection tests are used to determine the visibility of objects in the scene. If a ray sent from the camera intersects an object, then that object is visible. A rendering engine can have many objects; but, usually there are three main ones: triangles, spheres, and boxes. Triangles can be used as the building blocks of meshes, while spheres and boxes can be used as the bounding volumes of the nodes in a BVH.

Ray/triangle intersection tests are essential to the idea of ***tessellation***, a connected mesh of triangles which approximates a surface [Snyder and Barr, 1987]. Since many objects are represented as a collection of triangles, it is important to optimize ray/triangle intersection tests. There are many ray/triangle intersection algorithms; however, two of them stand out for their speed thanks to their utilization of barycentric coordinates: the Möller–Trumbore algorithm [Möller and Trumbore, 2005] and Synder-Barr

[3]Some examples include VR and AR applications.

algorithm [Snyder and Barr, 1987].

For ray/sphere intersection tests, there are two approaches. The first is the *geometric solution*, and it is based on the geometry of the ray/sphere intersection. The second is the *algebraic solution*, and it is is based on substituting the ray's equation in the sphere's equation to determine the intersection time [Glassner, 1989]. When a BVH is implemented, the intersection of a ray and a bounding volume, which is usually represented as an AABB, is one of the most calculated operations during rendering, so optimizing it is crucial for rendering efficiency. For ray/AABB intersection tests, most algorithms are based on the slab method [Kay and Kajiya, 1986]. This algorithms has been improved ever since, whether by using IEEE numerical properties to make it more robust [Williams et al., 2005], or by multiplying by the reciprocal of the corresponding component of the ray direction as instead of performing several divisions, thus saving time [Shirley, 2016][Barnes, 2019], among other improvements.

2.5 CPU Accelerated Rendering

In this section, we define the CPU and discuss how it can be used to accelerate rendering.

2.5.1 The CPU and Parallel Computing

The ***CPU***, also known as the *Central Processing Unit*, is the core component of a computer system. It is responsible for executing instructions and handling different kinds of operations. We can think of the CPU as the brain of the computer: It performs arithmetic and logic operations, interprets instructions from running programs, and manages communication between various hardware components. The CPU consists of cores that are capable of executing their own set of instructions concurrently, thanks to the parallelism inherent in its design. Using this parallelism, we can enhance the computational performance of computer programs. One way a programmer can take control of the parallelism of the CPU is to use ***OpenMP***: An API and a collection of compiler directives for programs written in C++ that provides support for parallel programming. OpenMP provides a programming interface for shared-memory parallelization in which multiple threads can efficiently access and modify shared data within a parallel region [Peterson and Silberschatz, 1985].

2.5.2 Parallel Rendering

Rendering is an *embarrassingly parallel* problem[4]; therefore, researchers have investigated how it can benefit from parallel programming. One of the earliest investigations into parallelizing rendering engines was conducted by Crockett in his seminal work [Crockett, 1997]. Crockett identified several practical considerations that arise when parallelizing rendering, and we present a non-complete list of these considerations below:

[4] An embarrassingly parallel problem is one that lends itself to partitioning into independent subtasks for parallel execution.

- Shared-memory systems (which OpenMP is built for) provide efficient access to a global address space, minimizing manual coordination of tasks and data access in parallel regions of the code.
- Shared-memory systems maximize the range of practical algorithms.
- When using shared-memory systems, resource contention must be addressed. One solution is to decompose the problem into tasks, which eliminates memory hot spots and minimizes critical section operations.

Krogh et al., [Krogh et al., 1995] presented one of the earliest implementations of a parallel rendering engine, which utilized message-passing for communication between processors. A notable gap in their work is that their engine renders spheres only.

Reinhard and Jansen [Reinhard and Jansen, 1997] presented two approaches to parallelize rendering:

1. **Demand-driven:** Assigns coherent sub-tasks to processors with a low load by dividing the scene into sub-parts and assigning each part to a different processor. This approach evenly distributes the load across processors; however, it may suffer from communication bottlenecks.
2. **Data-parallel:** Schedules tasks by distributing data across processors based on a spatial subdivision that rays are traced through. When a ray enters a new voxel[5], it is transferred as a task to the processor that owns the voxel's primitive. This approach makes the algorithm scalable; but, it may suffer from load imbalance.

Reinhard et al. proposed a hybrid implementation that combines both of these approaches. One limitation of their research is that no attempt was done to parallelize path tracing; they only demonstrated the hybrid approach for implementing ray tracing.

Parker et al. [Parker et al., 2005] discussed parallelizng interactive rendering. The paper began by introducing the simplest parallel ray tracing implementation: primary/secondary demand-driven scheduling through parallelizing the rendering loop. The authors argued that this approach incurs excessive synchrinization overhead, as each pixel representes an independent task. To address the issue, Parker et al. proposed reducing synchronization overhead by assigning groups of rays to each processor. Larger groups lead to lower synchronization overhead. Nonetheless, this approach also presents challenges, as larger groups can lead to increased idle time for processors, resulting in poor load balancing.

[He et al., 2021] presented two ways to optimize the primary/secondary demand-driven scheduling approach just discussed. The first approach is to use an operation pool, which collects all rays emitted from the pixel points. Operations in the pool are then distributed among several threads. The second approach is to use the OpenMP directive: *omp schedule*. This directive assigns the work to threads line-by-line, instead of pixel-by-pixel. Their approach was to use *static, 1* (that is, static scheduling and chunk size equal to 1) scheduling. They argue that this approach is ideal, because neighbor pixels are more likely to have similar recursion depths. By distributing the work line-by-line, threads are more likely to receive comparable workloads. This minimizes time loss due to load imbalance. Using 8 threads, they achieved a speedup of 4.84 with the straightforward approach and 5.77 with the static scheduling approach.

Kadir and Khan [Kadir and Khan, 2008] described an implementation of a parallel ray tracer using *MPI*[6]

[5] In the context of 3D computer graphics, a voxel refers to a 3D pixel.

[6] MPI, short for Message Passing Interface, is an API that allows for communication among processes in parallel in a distributed memory architecture https://www.open-mpi.org/doc/v4.0/

and OpenMP. They implemented a ray tracer for both distributed memory and shared memory systems. The idea of their implementation is to use *load balancing* to distribute work evenly across multiple processors or threads. Load balancing distributes rendering tasks to ensure that each processor handles a similar number of pixels. Specifically, they used *static load balancing*; it is the simplest form of load balancing and involves dividing the work into equal chunks and assigning each chunk to a processor. Finally, they evaluated the performance of their implementation using several metrics.

One of the evaluation metrics they used to benchmark their implementation is *speedup*, which is a ratio of the execution time on a single processor to the execution time on multiple processors. They achieved a speedup of up to 1.6 when using OpenMP on 8 threads. For the case of MPI, they achieved a speedup ratio of approximately 1.45 on 8 processors.

There are some limitations in the implementation presented by Kadir et al. First, their rendering engine only renders spheres; there is no support for other primitives such as triangles, boxes, etc. Second, the load balancing technique they implemented is not dynamic; hence, it may not scale well to different scenes.

2.6 CUDA and the GPU

This section will provide a brief overview of the GPU and CUDA, followed by a discussion of GPU-based rendering and how GPUs are used to accelerate rendering. The ***GPU*** is an electronic component capable of handling computationally expensive tasks. GPUs are highly parallelizable: One can exploit parallelism to enable GPUs to divide a problem into millions of separate tasks and tackle them all at once. One way to run code on the GPU is to use CUDA. ***CUDA*** is an API developed by Nvidia; it allows programmers to write software that utilizes the GPU [Sanders and Kandrot, 2010].

As we discussed in §2.5.2, we can write parallel code on the CPU to accelerate rendering; however, the architecture of the CPU limits the degree of parallelism that we can benefit from, because a CPU contains far fewer cores than a GPU. To further accelerate rendering performance, it would be desirable to have the CPU control the main flow of the program and give instructions to the GPU to carry out the huge burden of intense calculations.

2.6.1 The Architecture of the GPU

GPUs were designed to lighten the workload on the CPU when demanding applications are run, especially those involving 3D graphics [Cioli et al., 2010]. GPUs can be regarded as a collection of multicore processing units that share memory. They adhere to the *single-program, multiple-data* (SPMD) parallel programming paradigm, where multiple cores execute the same program on different segments of the data simultaneously.

Traditionally, the approach taken to improve CPUs was to rely on adding more transistors to enhance the performance of sequential code; however, transistor scaling is becoming harder [Jennings, 2019]. GPUs, on the other hand, present a compelling alternative; they are refined for graphics rendering, and their programmability performance is effectively enhanced by the use of additional transistors [Purcell et al., 2005].

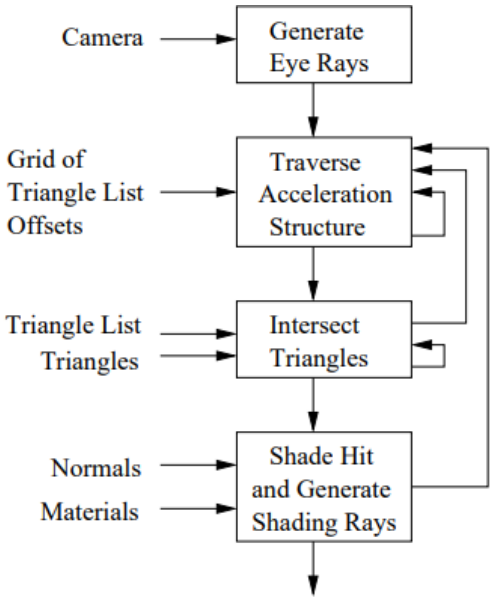


Figure 2.7: A flow diagram for a streaming ray tracer. *Hyperion* (see Figure 1) is an example of a streaming ray-tracer [Karlsson and Ljungstedt, 2004].

2.6.2 Implementing Rendering Algorithms on the GPU

Purcell et al. presented a framework to map ray tracing onto graphics hardware [Purcell et al., 2005]. The idea is to redefine ray tracing as a *streaming computation*: A computation that executes a program on each element of the input stream and direct the result to an output stream, as opposed to processing the required data as a sequential stream of elements (see Figure 2.7). This allows the GPU to run a vertex program on a stream of vertices and a fragment program on a stream of fragments. This is a high view of the programmable graphics processor: It is treated as a paralleizable computational device capable of speeding up expensive algorithms like ray tracing.

There are certain limitations of implementing parts of the rendering pipeline on the GPU that we must be aware of. Aila et. al. [Aila and Laine, 2009] explored the limitations of ray traversal on GPUs. Mainly, the research paper focused on exploring the challenges of traversing acceleration structures and ray/object intersections. We shall focus on reviewing the second challenge, and defer discussing accelerated structures on GPUs to a later subsection (§2.6.3). The authors argue that the performance of ray/object intersection is limited by several factors:

- Computation.
- Memory bandwidth.
- Parallel work distribution.
- Other unknown factors.

They approached their assessement by implementing optimized versions of some of the fastest GPU ray traversal routines in CUDA. The implementations are then compared to a custom simulator that estimates the theoretical upper bound performance for a specific NVIDIA GPU.

Furthermore, Aila et al. outlined three ways to improve *Single Instruction, Multiple Data* (SIMD) efficiency and performance:

- *Replacing terminated rays.* Long-running rays may cause starvation of resources; therefore, they can be periodically replaced with new ones that commence from the root.
- *Utilizing work queues.* Work queues provide a way of ensuring virtually optimal SIMD utilization regardless of affecting factors.
- *Using wide trees.* The idea is that trees with branching factor higher than two make memory fetches more coherent and can lead to enhanced SIMD efficiency.

The limitation Aila et al's research is that their implementations don't include textures, shading, and building and maintaining acceleration structures. These elements are known to significantly impact the optimization of rendering on GPUs.

Cioli et al. explored ways to improve the performance of ray tracing using a GPU [Cioli et al., 2010]. Their approach involved developing three distinct GPU implementations of the ray tracing algorithm, each iteration building upon the strengths and addressing the shortcomings of its predecessor. The goal of their research is to show that GPU aided ray tracing can run faster than a ray tracer implemented on the CPU. The key observation is that ray tracing can benefit from SPMD parallelism. This is a different approach from the previous paper, which focused on the SIMD paradigm. The idea is to extend the implementation from the CPU to the GPU and associate a distinct thread with each ray sent to the scene. Cioli et al. utilized the *grid construction algorithm*, a type of uniform space subdivision acceleration structures (see §2.3.1). The three algorithms follow a five-step procedure:

1. Load the scene data from a text file.
2. Invoke the grid construction algorithm.
3. Transfer the data from the CPU to the GPU.
4. Perform ray calculation (shading, lighting, etc...) to calculate each pixel's color.
5. Copy back the data from the GPU to the CPU and render the scene on the screen.

The evaluation methodology was carried out using three types of test cases:

1. Different distribution of objects.
2. Different number of objects in the scene.
3. Using scenes from other research papers.

Their experimental results showed that the speedup achieved from the grid construction algorithm is heavily dependent on the choice of the grid size. An optimal grid size can substantially increase the performance of the ray tracer. Moreover, their results showed that it is important to leverage the different memory levels (global, texture, and constant memory) of the GPU. Furthermore, it is always beneficial to optimize the code in a way that minimizes the number of calculations. In the case of their research, Cioli. et. al implemented an efficient ray/triangle intersection algorithm. The future work that this paper proposes includes evaluating the use of different acceleration structures and extending the ray tracing algorithm to multiple GPUs.

2.6.3 Constructing Accelerating Structures on GPUs

As previously mentioned, one way of creating acceleration structures is to build spatial hierarchies. The challenge any implementation of a spatial data structure faces is the overhead cost of organizing the objects in space into a hierarchy. Since the trend of programming on the GPU is toward achieving parallelism, it is desirable to construct BVHs in parallel to ensure they are scalable to future architectures.

Lauterbach et. al. investigated the construction of accelerated structures on GPUs [Lauterbach et al., 2009]. Their research presented two novel parallel algorithms capable of leveraging GPUs to construct BVHs. They then combined these two algorithms into a hybrid algorithm. Their results showed that the construction cost can significantly benefit from the use of graphical hardware. The algorithms presented in the paper were implemented in CUDA.

The first algorithm uses spatial *Morton codes* to reduce the problem of constructing a BVH into a sorting problem. A Morton code is a mapping from multi-dimensions to one dimension that preserves locality. The idea Lauterbach et. al. presented was to use Morton codes to order the objects in the scene. The motivation is that a Morton index code is directly computable from its geometric coordinates. Once each object in the scene has a Morton code assigned to it, a BVH can be easily constructed by bucketing the objects based on the bits of their Morton code. This bucketing strategy is applied recursively to construct the hierarchy. Each layer of the hierarchy corresponds to a recursion layer. The strategy creates the initial split in a two-step routine. First, it examines the most significant bit of each object's code. Second, it allocates those with 0 bits to the first child and those with 1 bit to the second child. The routine is then recursively invoked on each child with only one difference: at each recursion layer, we look at the next most significant bit.

This algorithm, called *Linear Bounding Volume Hierarchy* (LBVH), aims to minimize the cost of BVH construction. LBVH employs the Morton code strategy to construct BVHs on the GPU. LBVH construction time is linear, $O(N)$, and the depth of the hierarchy produced is $O(log(N))$. This beats the baseline BVH algorithm, which takes an $O(Nlog(N))$ construction time.

The second algorithm the paper presented was a top-down approach that employs the surface area heuristic (SAH) (discussed in §2.3.3) to construct BVHs. The BVHs built are optimized for fast ray tracing. The SAH computation is performed as usual, generating k uniformly sampled split candidates for each of the three axes. The algorithm then uses $3k$ threads to test all samples concurrently. Then, a parallel reduction using the minimum operator finds the split candidate sample with the lowest cost. The candidate is compared to the SAH cost of not splitting the current node and makes a decision about which node to split.

Finally, Lauterbach et al. suggested a hybrid approach to GPU construction of accelerated structures. The idea is to combine LBVH and the SAH construction algorithm into a unified algorithm. The three algorithms the research paper presented were benchmarked on different scenes. The following results were obtained:

- The LBVH algorithm is highly scalable.
- The construction time of LBVH is linear, meaning it is very fast.

Some of the gaps of their works are summarized below:

- Their construction algorithm does not support systems with multiple GPUs.
- The Morton-based sorting algorithm can be further enhanced for improved efficiency.

Pantaleoni and Luebke [Pantaleoni and Luebke, 2010] improved on the LBVH algorithm by introducing a hierarchical algorithm, *Hierarchical LBVH* (HLBVH), that is more time and storage efficient. Their idea was to reformulate the LBVH algorithm by splitting it into a two-level hierarchy, sorting primitives into coarse and fine grids[7] for faster ray tracing. In the same paper, they extended HLBVH by using a hybrid approach that combines the HLBVH algorithm with SAH sweep builder to construct high-quality SAH-optimized trees at a lower cost than using the SAH sweep builder alone[8]. Their algorithms were implemented in CUDA.

Pantaleoni and Luebke suggested that their top-level SAH building procedure had the potential to be further optimized, which could have led to a reduction in memory usage. This gap was addressed by Chitalu et al. [Chitalu et al., 2020]. They proposed an *ostensibly-implicit tree*[9] for BVH construction that leveraged local atomic operations to synchronize threads within a group. This approach reduced overhead and memory consumption, and it eliminated the need for post-processing to compact data. As of today, Chitalu et al.'s algorithm is the **fastest** construction algorithm.

2.6.4 Heterogeneous Approaches to Accelerating Rendering

It is not necessary to solely utilize either the GPU or the CPU for rendering. We can also benefit from *heterogeneous computing* - the use of multiple types of cores. There are certain difficulties in utilizing different core types to cooperate. For one, both the GPU and the CPU share the same memory architecture, and there is a possibility of competition for resources between the two core types. Specifically, this arises when the CPU is involved in intensive calculations, forcing the GPU to compete with it for shared resources.

The idea Barringer et. al [Barringer et al., 2017] presented was to develop a hybrid rendering system that divides the work between the CPU cores and the GPU. The communication between the CPUs and the GPU is done via shared memory. The idea is to organize rays in large packets that can be flexibly distributed among the two different cores. The outline of the work distribution between the GPU and the CPUs is as follows:

- GPU: handles the bulk of ray queries, including BVH traversal and triangle intersection.
- CPUs: shoot rays into the scene, schedule ray queries, and perform shading calculations at the intersection point on the surface of objects in the scene.

The allocation of work is done via large buffers of random rays, which are called *ray streams*. The idea

[7] Coarse and fine grids refer to two different levels of granularity in the hierarchical LBVH construction algorithm. In the first level, primitives are sorted into a coarse grid according to a part of their Morton code. In the second level, primitives within each voxel of the coarse level are sorted according to the remaining bit.

[8] We discussed SAH trees in §2.3.3, and we specified that their construction can be slow; hence, Pantaleoni and Luebke applied SAH to the already formed clusters in their HLBVH's coarse grid, minimizing the overhead.

[9] They are called *ostensible* because they are implicitly represented as binary trees, but they can be transformed into an explicit representation during runtime if needed.

is to stockpile an adequate number of rays to keep the resources of the CPUs and the GPU saturated. Ray streams that send work to the GPU are called *ray dispatch threads* and those that send work to the CPU are called *host threads*. A central scheduler manages the workload the ray dispatch threads and host threads send, ensuring that the CPUs and the GPU are operating to the fullest extent. A rendering algorithm is implemented on top of the scheduling algorithm, and the scheduler invokes callbacks to it.

Barringer e. al. evaluated their work using six test scenes of varying complexity. Their system has proven to have low overhead. Moreover, vectorized shading has shown to improve performance by a considerable factor. On the other hand, their evaluation showed that heterogeneous approaches increase the power usage, and heterogeneous systems are prone to the problems we stated earlier. For future works, this research suggested examining whether rendering performance can be further enhanced by grouping expensive shading tasks and offloading their computation to the GPU.

Chapter 3

Methodology, Evaluation & Requirements

3.1 Introduction

This research focuses on investigating ways to expedite rendering. The two objectives are to implement a rendering engine and accelerate its speed. In this chapter, we will list the features we aim to implement and discuss the evaluation strategy we will use.

3.2 Requirements

We will categorize functional and non-functional requirements into four distinct groups:

1. **Minimal Usable Subset Requirements:** These requirements are essential for the success of this project, and failure to meet them may hinder meeting essential project objectives.
2. **High-Priority Requirements:** These requirements can add substantial value to our research; however, omitting them will not cause the research to fail.
3. **Desirable Requirements:** These requirements aren't necessary for completing our research. If they are not implemented, there is minimal to no impact on the completion of our dissertation.
4. **Future Requirements:** These requirements belong to future works and will not be implemented.

3.2.1 Functional Requirements

Minimal Usable Subset Requirements:

1. **Camera:** The rendering engine must provide a camera for viewing the scene.
2. **Rendering Objects:** The engine should render primitives (such as spheres, triangles, boxes, etc.).

3. **Lighting Support:** The engine must support lighting and should simulate its effects based on the shading model. This is essential, because if there is no light we cannot see our rendered images.

High-Priority Requirements:

1. **Support for Materials** The engine should support rendering different materials (e.g. diffuse).
2. **Clear Rendered Images:** The engine should support anti-aliasing and importance smapling to reduce noise in the rendered scenes.
3. **Acceleration Structures:** Implementation of BVH and using axis-aligned boxes as bounding volumes to accelerate ray/object intersection tests.
4. **Multi-Threading Support:** Utilization of multi-threading (through OpenMP, for example) to leverage CPU parallelism for accelerated rendering.
5. **Hardware Acceleration:** Integration with hardware acceleration to accelerate rendering.

Desirable Requirements:

1. **Multiple Light Sources:** Implementing various lighting models (like ambient lighting and directional lighting).

Future Requirements:

1. **Real-time Procedural Generation:** The rendering engine could be enhanced to support real-time procedural generation.
2. **Machine Learning Integration:** Exploring deep learning techniques to accelerate rendering.

3.2.2 Non-functional Requirements

Minimal Usable Subset Requirements:

1. **Code Documentation:** The code must be well documented and clear.
2. **Use of Version Control:** We must track code and dissertation writing changes and maintain a history of development.

High-Priority Requirements:

1. **Code Profiling:** Utilize profiling tools to identify performance bottlenecks and optimize the code accordingly. Document the optimization process and results in the dissertation.
2. **File Format Compatibility:** The system should support reading common 3D file formats.

Desirable Requirements:

1. **Error Handling and Reporting:** Provide clear, informative error messages to facilitate debugging.

Future Requirements:

1. **User Interface:** There will be no user-friendly interface (a GUI).
2. **Integration with Game Engines:** We will not integrate our engine with any game engines.

3.3 Evaluation Strategy

This research aims to examine several aspects, as outlined below:

- **Evaluate the performance of rendering acceleration techniques:** We will measure the render times of different acceleration techniques to determine their efficiency.
- **Compare and contrast different rendering acceleration techniques:** This involves conducting evaluations to identify the strengths and weaknesses of different acceleration techniques relative to each other, aiming to understand the trade-offs between performance and quality.
- **Identify limitations and propose potential improvements.**

3.3.1 Metrics

Evaluation metrics are quantifiable measures used to assess the performance and quality of a system. These metrics should serve as a standardized way to compare different approaches and identify the most effective solution for a given research question. Below, we will define the metrics that we will use. In our case, we can categorize evaluation metrics into three main areas:

1. **Rendering Performance:**
   (a) *Render Time.*
   (b) *Total Number of Ray-Object Intersection Tests.*
2. **Parallelism:** A measure of the engine's efficiency in utilizing machine resources for parallel processing to improve performance.
   (a) *Speedup*: A measure of the relative performance gain of a parallel program compared to its serial counterpart.
   $$\text{Speedup Ratio} = \frac{\text{Sequential Execution Time}}{\text{Parallel Execution Time}} \tag{3.1}$$
   (b) *Efficiency*: A measure of how well a parallel algorithm performs in comparison to the theoretical maximum efficiency that could be achieved with a given number of processors.
   $$\text{Efficiency} = \frac{\text{Speedup Ratio}}{\text{Number of Cores Used by the Parallel Algorithm}} \tag{3.2}$$
3. **Code Usability:** Assesses the readability and reusability of the code, as well as its extensibility.

3.3.2 Evaluation

In this subsection, we outline our approach to evaluating our research.

1. **Rendering Performance**:
   (a) By analyzing the *render time*, we can benchmark our engine's performance against other rendering engines. Moreover, we can use the *render time* to measure how much speedup was achieved by, for example, implementing acceleration structures or utilizing parallelism.

(b) We use the *total number of ray/object intersection tests* to test the efficiency of our optimizations.

2. **Parallelism:**
   (a) *Speedup* > 1 indicates performance improvement through parallelization. We aim to parallelize our code such that we utilize the available machine's units to improve the engine's performance.
   (b) *Efficiency* can be used to compare different parallel implementations. By definition, the maximum theoretical *efficiency* is 1. *Efficiency* close to 1 indicates that the parallel algorithm is utilizng the available processors efficiently, while *efficiency* close to 0 indicates poor processors utilization. We aim to make our parallelism as efficient as possible. We note that the amount of logical processors a programmer may use from the available pool can usually be controlled. In the case of OpenMP, for example, the programmer sets the number of threads, and the OS maps those threads to cores.
3. **BVH Build Time:** We will evaluate different BVH (Bounding Volume Hierarchy) implementation strategies by comparing the time it takes for each strategy to construct the hierarchy.

Chapter 4

Project Management

4.1 Introduction

Managing our workflow is an integral part of our research. We use two management tools: *GitHub*[1] and *Overleaf*[2].

We use GitHub for the following reasons:

- **Version Control**: GitHub enables us to track our changes over time. We can easily manage our project files using it.
- **Remote Access**: If we have our project folders in our repository in GitHub, we can access them from anywhere and using any device. This makes our work more flexible.
- **Backup**: As our project files are stored on GitHub, we can always revert to older versions of our code if necessary. This ensures that we have a backup in case the main computer used for the project encounters any technical issues that renders it unusable and its files inaccessible.

We use Overleaf for the following reasons:

- **Using LaTeX**: LaTeX is a powerful system to produce high-quality writing. It is a standard system in research. LaTeX facilitates inserting equations, charts, graphs, and figures into the dissertation's text.
- **Collaboration**: Overleaf allows for collaboration on the same document; hence, the supervisor and the student can both make changes and review feedback simultaneously.
- **Backup**: Overleaf is another version control tool that can be used to retrieve the documents if the main computer is damaged.

[1] https://github.com/.
[2] https://www.overleaf.com/.

4.2 Professional, Legal, Ethical and Social Issues

4.2.1 Professional Issues

Project development will comply with the British Computer Society's Code of Conduct, and the source code will be developed following modern software development practices.

4.2.2 Legal Issues

This research adheres to all laws and regulations. Data used is taken from publicly accessible sources or with proper authorization and complies with applicable data protection laws and guidelines. Copyrighted materials, if utilized, will respect the requisite permissions and licenses.

4.2.3 Ethical and Social Issues

This research project does not involve sensitive data or human subjects; thus, there is no risk breaking ethical guidelines.

On the aspect of social issues, a rendering engine itself raises no concerns, but products that utilize it can do so. Examples of these products include video games. The following is a non-complete list of possible concerns:

- Playing video games has the potential to increase stress levels [Pallavicini et al., 2022].
- Video games can contribute to loneliness [Lemmens et al., 2011].
- Immersive games environments that have detailed graphics can cause addiction.

On the other hand, some products that use rendering algorithms can have societal benefits. We list some of these applications' benefits below:

- Playing video games can contribute to increased well-being [Johannes et al., 2021].
- Rendering in medical imaging is used by doctors to facilitate surgeries [Erickson, 2017].
- Rendering programs are used for data visualization in many scientific fields, such as astrophysics.

Chapter 5

Implementation

In this chapter, we aim to discuss the implementation of our rendering engine. While we won't be able to cover every aspect, we will mostly focus on the features we implemented to accelerate rendering. Additional details about the implementation can be found in the Appendix §A.3.

5.1 Preliminaries

In order to render any scene, a convenient way to represent and manipulate 3D vectors is needed. To address this need, we implement the `Vec3D` class, which encapsulates 3D vectors operations. With `Vec3D`, we can implement other core classes like `Ray` and `Camera`. The details of their implementation can be found in the Appendix (§A.3.2), as they are somewhat standard among many rendering engines.

In our engine, we need a way to represent 3D objects, or ***primitives***. These surfaces can be implicitly represented; for example, a sphere is defined by a radius and a center point. A key idea is that a point in 3D space is nothing but a location represented by a 3D vector, which can be represented by our `Vec3D` class. To handle different types of primitives, we create a base class called `Primitive`. This class acts as a parent to various primitives (derived classes) used in our engine. These classes implement functions essential to rendering, such as intersection tests and shading evaluation. Below are the derived classes:

- `Triangle`
- `Sphere`
- Axis aligned rectangles (namely, `XY_Rectangle`, `XZ_Rectangle`, `YZ_Rectangle`). Together, these classes can form a box.

When we send a ray from the camera into the scene, we are essentially collecting a bunch of information (see §2.2.1.2), such as the time of the closest intersection, the material of the intersected surface, the normal at the intersection point, and so on. This is realized by implementing a structure that we call `Intersection_Information`, which we pass as a parameter to our ray/primitive intersection method to keep track of the required intersection information.

5.2 A Basic Ray Tracer

As discussed in §2.2.1.2, ray tracing is a very simple algorithm: We visit every pixel of the screen and determine whether or not any primitive lies in that pixel and perform shading calculations accordingly. This iterative process is commonly referred to as the ***render loop***, and its details can be summarized in a few lines as shown below:

Algorithm 1 The Render Loop

1: **for** each pixel in the image **do**
2: Construct a ray from the camera origin in the direction of the pixel
3: Find the first object intersected by the ray and perform all shading calculations at that point
4: Set pixel color to the value computed from the hit point, light, and **n**
5: **end for**

Each scene can be parameterized uniquely. The set of the parameters of a scene include the dimension of the output image, camera configurations, rendering settings, and the composition of the scene. The latter is specified by both primitives and lights. To guarantee this uniqueness, and to facilitate reproducibility, we encapsulate these parameters within a `struct` named `Scene_Information`. This is similar to the use of a `struct` in §5.1 to encapsulate the intersection information at an intersection point. The following code snippet is the general way of representing a scene in our engine:

```
Scene_Information some_scene(){
    Scene_Information scene_info;
    // Set image settings
    ...
    // Set rendering settings
    ...
    // Set camera settings
    ...
    // Initialize materials
    ...
    // Initialize primitives
    ...
    // Add primitives to the list
    ...
    // Create the BVH
    ...
    // Add lights to the list
    ...
    return scene_info;
};
```

Listing 5.1: The `Primitive` class.

For instance, initializing a rendering setting, such as the samples-per-pixels to be used, is a straightforward call:

```
scene_info.samples_per_pixel = 500;
```

The returned scene information (**Line 19**) is then passed to the function that runs the render loop.

5.3 Implementing Distributed Ray Tracing

Recall that a distributed ray tracer is a ray tracer that can handle "soft" phenomena, such as anti-aliasing (§2.2.1.3). A ray tracer sends a single ray per pixel, while a distributed ray tracer sends many rays through each pixel and average their results. To achieve this, we just need to add one line to the render loop algorithm we presented above:

Algorithm 2 The Render Loop with Antialiasing

```
1: for each pixel in the image do
2:     for s ← 0 to samples_per_pixel - 1 do
3:         Construct a ray from the camera origin in the direction of the pixel
4:         Find the first object intersected by the ray and perform all shading calculations at that point
5:         Set pixel color to the value computed from the hit point, light, and n
6:     end for
7: end for
```

5.4 Shading and the Material System

We reviewed some shading models in §2.2.2 and discussed the role of the BRDF (which we denoted as ρ) in rendering in §2.2.2.2. In our engine, we perform shading calculations using a specific function that we call `radiance()` (we discuss shading more in §5.5.2). We implement multiple versions of this function, ranging from a basic one that simply calculates the color of a pixel based on the intersection information to a more sophisticated one that can sample the PDF of several surfaces, allowing for more accurate physically based rendering. The general algorithm of `radiance()` is shown below:

Algorithm 3 Color Radiance

1: **function** RADIANCE($\overrightarrow{r_{in}}$, *primitives_list*, *depth*, *background_color*)
2: **if** *depth* ≤ 0 **then**
3: **return** Color(0, 0, 0)
4: **end if**
5: **if** $\overrightarrow{r_{in}}$ doesn't intersect any primitive $\in$ *primitives_list* **then**
6: **return** *background_color*
7: **end if**
8: *color_from_emission* $\leftarrow$ assign light color if intersected object's material is emissive
9: **if** there are no shading calculations at the intersection point **then**
10: **return** *color_from_emission*
11: **end if**
12: perform any required shading calculations (evaluate $\rho()$, PDF, etc...)
13: **return** *color_from_emission* $+\frac{\rho() \times \textbf{\textit{radiance}}()}{pdf}$
14: **end function**

`radiance()` simply iterates over the list of primitives in the scene. If there is no intersection, the background color is returned; otherwise, it performs shading calculations to get the color of the pixel, and recurs, multiplying by the BRDF and dividing by PDF (more on that later). If the recursion depth is exceeded, the algorithm halts. We can now modify the render loop to include the `radiance()` function:

Algorithm 4 Render Loop with *radiance*() accumulated over all samples

1: **for** each pixel in the image **do**
2: **for** $s \leftarrow 0$ **to** *samples_per_pixel* - 1 **do**
3: Construct a ray from the camera origin in the direction of the pixel
4: pixel color $\leftarrow \sum_{s=0}^{samples\_per\_pixel-1} radiance()$
5: **end for**
6: **end for**

As previously mentioned, we can extend `radiance()` to other versions:

- `radiance_sample_light_directly()`: Calculates the radiance by directly sampling lights. In this version, we need to pass a list of lights in the scene, and perform sampling calculations as an extra step in **line 12**.
- `radiance_mixture()`: Calculates the radiance using a mixture distribution approach, combining light source sampling with importance sampling based on the BRDF of each primitives. Again, we perform any required additional calculations in **line 12**.

To enable these extensions, we need to look at path tracing via *Monte Carlo* integration.

5.5 Monte Carlo Rendering

The ***Monte Carlo*** method is a probabilistic solution to a nonprobabilistic problem. Recall that in §2.2.1.5 we discussed the use of the Monte Carlo method to solve the rendering equation. In this section, we will see how we used the Motne Carlo method in implementing our engine.

The essence of the Monte Carlo method is simple: Use random samples to approximate the average value of a function (in our case, the rendering equation) over the region of interest (in our case, the hemisphere of directions), and then weight the result by the *probability distribution function* (PDF)[1] that determines how the random samples were drawn.

Suppose we have an integral (say, $\int_{x \in S} g(x)\, d\mu$) that we want to approximate. The Monte Carlo method can be summarized in three steps (it is really just like averaging):

1. **Random Sampling.**
   - Draw random samples x_i from a distribution. The distribution is defined according to a certain PDF $p(x)$.
2. **Add All Samples Together.**
   - Sum up all the random samples that we generated, weighting each by the PDF: $\sum_{i=1}^{N} \frac{g(x_i)}{f(x_i)}$, where N is the number of random samples (x_i's) generated.
3. **Weight the Summation.**
   - Divide the summation by the number of the random samples N we drew: $\frac{1}{N} \sum_{i=1}^{N} \frac{g(x_i)}{f(x_i)}$.

Hence, to approximate the rendering equation (Equation 2.2) via Monte Carlo simulation, we simply calculate[2]:

$$L_s(\mathbf{k}_o) \approx \frac{1}{N} \sum_{i=1}^{N} \left[L_e(\mathbf{k}_o) + \frac{\rho(\mathbf{k}_i, \mathbf{k}_o) L_f(\mathbf{k}_i) \cos\theta_i \, d\sigma_i}{p(\mathbf{k}_i)} \right] \tag{5.1}$$

The last step tells us that the more samples we generate, the better our approximation will be; this is enforced by the *the law of large numbers*. The larger N is, the more confident we are of our approximation. This is really what we have been doing all along: We take more samples-per-pixel to get a less-noisy image (i.e., closer to the ground truth); however, we have seen that we need many samples-per-pixel to get a denoised image (this is a penalty of the *law of diminishing returns*), and this really slows down our rendering engine. Hence, we want a way to make our rendering equation converge to the ground truth result with as few samples-per-pixel as possible, accelerating our engine in the process. This is done through *importance sampling*, which we discuss in the next section.

5.5.1 Importance Sampling

Recall that the first step of the Monte Carlo is concerned with drawing random samples from a distribution $p(x)$, but how do we get $p(x)$? One way would be to choose a $p(x)$ that is uniform over the hemisphere of

[1] This is why we divide by the PDF in the `radiance()` function.

[2] This is noting but an *expected value*, and one can use the *linearity of expectation* to obtain Equation 5.1.

directions. This amounts to taking $p(x) = \frac{1}{2\pi}$ (i.e., dividing by the curved surface area of the hemisphere). However, such a uniform random sampling approach is not optimal. To see this, simply substitute it in Equation 5.1, and assume we have a Lambertian (i.e., perfect diffuse) BRDF, so ρ is equal in all directions: $\rho = \frac{1}{\pi}$. We can notice that a *cosine* term still exists in the equation. So for a diffuse material, a uniform pdf is simply *imperfect*. Therefore, the key question is how can we choose a PDF so that we can converge to the ground truth results with as few samples-per-pixel as possible? The answer to this question is *importance sampling*: Choosing the PDF p intelligently such that wherever p is large where the integrand is large, we get more samples in important regions, which essentially means **faster** convergence to the ground-truth result.

There are two important properties of importance sampling:

1. Any choice of PDF p always converges to the ground-truth answer; however, the closer p approximates g, the faster the convergence.
2. A linear mixture of PDFs is itself a PDF, and is referred to as a *mixture density*.

5.5.2 Implementing Importance Sampling in Shading Calculations

Shading can be summarized in three steps:

1. Construct the BRDF of the material.
2. Decide in which direction should we trace the next ray.
    - This is done through the PDF.
3. **Importance Sampling Step:** Choose the PDF intelligently so that it cancels out some BRDF terms.
    - This not only simplifies calculation (i.e., faster rendering) but also reduces the variance (noise) of our Monte Carlo approximation.

To implement PDFs, we start with creating a template class called `PDF`, and then implement derived classes for any PDFs we choose to incorporate. In our case, we implement the following:

1. `Mixture_PDF`: Allows for calculating mixture densities.
2. `Cosine_Weighted_PDF`: Generates directions based on a cosine weighted distribution. This is the true Lambertian reflection, so we use it for the `Diffuse` material.
3. `Primitive_PDF`: Samples directions toward the primitives themselves.
4. `Specular_PDF`: Generates directions based on a cosine power cosine-weighted distribution. We use it for the importance sampling version of the `Phong` material.
5. `Uniform_Hemispherical_PDF`: Generates directions by uniformly sampling the hemisphere, forming a diffuse reflection that is not Lambertian. Used for the `Uniform_Hemispherical_Diffuse` material.
6. `Uniform_Spherical_PDF`: Similar to the previous, but it generates directions by uniformly sampling the sphere.

A valid question arises: How do we generate non-uniform random samples? The answer lies in using the *inverse transform sampling* method, which is based on function inversion. This method can be summarized in 4 steps:

Sampling Strategy	PDF of Directions	Cartesian Coordinates
Uniform Sphere	$p(\omega) = \frac{1}{4\pi}$	$(2\sqrt{r_2(1-r_2)}\cos(2\pi r_1), 2\sqrt{r_2(1-r_2)}\sin(2\pi r_1), 1-2r_2)$
Uniform Hemisphere	$p(\omega) = \frac{1}{2\pi}$	$(\sqrt{1-r_2^2}\cos(2\pi r_1), \sqrt{1-r_2^2}\sin(2\pi r_1), 1-r_2)$
Cosine-weighted Hemisphere	$p(\omega) = \frac{\cos\theta}{2\pi}$	$(\cos(2\pi r_1)\sqrt{r_2}, \sin(2\pi r_1)\sqrt{r_2}, \sqrt{1-r_2})$
Power-cosine-weighted Hemisphere	$p(\omega) = \frac{\cos^n\theta}{2\pi}$	$(\cos(2\pi r_1)\sqrt{1-r_2^{(\frac{2}{n+1})}}, \sin(2\pi r_1)\sqrt{1-r_2^{(\frac{2}{n+1})}}, r_2^{n+1})$

Table 5.1: Sampling Strategies and Corresponding PDFs. Both r_1 and r_2 are uniform random numbers $\in [0, 1]$. The n in the last cell is the specular exponent, which is used in the importance sampling version of the Phong material [Lafortune and Willems, 1994] [Simon, 2015] [Shirley et al., 2021].

1. Generate a uniform random number r_1 from the uniform distribution $\in \mathcal{U}(0, 1)$.
2. Compute θ for which $F(\theta) = r_1$, which is essentially the inverse $F^{-1}(r_1)$.
3. Use this θ as a random number generated from the nonuniform distribution.
4. Repeat the same process (1-3) for ϕ using another random number r_2.

Now that we represented the random numbers using polar coordinates, we will have to convert them to Cartesian coordinates. Table 5.1 summarizes all the sampling strategies that we used in our implementation. The derivation of these Cartesian coordinates is shown in the Appendix (§A.3.1), alongside the mathematics of sampling light sources.

To incorporate this in the `radiance()` function (see §5.4 and the extensions proposed), we simply add the following lines:

```
...
auto light_ptr = std::make_shared<Primitive_PDF>(lights, rec.p);
Mixture_PDF mixture_pdf(light_ptr, surface_pdf_ptr);

scattered_ray = Ray(rec.p, mixture_pdf.generate_a_random_direction_based_on_PDF(), r.
    get_time());
double new_pdf = mixture_pdf.PDF_value(scattered_ray.get_ray_direction());

if (new_pdf == 0.0)
    return color_from_emission;

return color_from_emission + rec.mat_ptr->BRDF(r, rec, scattered_ray, surface_color) *
    radiance_mixture(scattered_ray, world, lights, depth-1, background) / new_pdf;
```

Listing 5.2: `radiance_mixture()`.

5.6 Implementing Intersection Tests

Intersection tests determine what objects are visible in the scene. They are the most called functions; hence, it is important to optimize them (see §2.4). In this section, we will discuss their implementation in our engine.

5.6.1 Ray/Sphere and Ray/Triangle Intersection Tests

For ray/sphere intersection, we implement the *algebraic solution* and *geometric solution*. The first substitutes the ray into the sphere's equation to find the intersection time. The second considers the intersection geometry. Details can be found in the Appendix §A.3.3.1. In addition, for ray/triangle intersection, we also implement two methods: *Möller–Trumbore* and *Snyder & Barr*. Both leverage barycentric coordinates. Details can be found in §A.3.3.2.

5.6.2 Faster Ray/Box Intersection Tests

When it comes to ray intersection tests, the most significant speed improvement can be achieved by optimizing ray/AABB intersection tests. The ray/AABB intersection is one of the most called functions in rendering (§2.3). Certain optimizations of the ray/AABB intersection algorithm can result in more than a twofold increase in rendering speeds compared to using the basic intersection method (the *slab method*).

Overall, we implemented a total of five ray/box intersection algorithms, four of which were already seen in the literature review chapter (§2.4):

Algorithm	Reference
Slab Method	[Kay and Kajiya, 1986]
Branchless Method	[Barnes, 2019]
Optimized Slab Method	[Williams et al., 2005] & [Marrs et al., 2021]
Kensler Method	[Shirley, 2016]

Table 5.2: Ray/triangle intersection algorithms and references.

The fifth algorithm is one that we wrote, and it is an observation made on the *branchless method*. Instead of calling `fmin()` and `fmax()` when evaluating `t_0` and `t_1`, we use a swap operation, `swap(t_0,t_1)`, to ensure that `t_0` is the smaller value and `t_1` is the larger value, without the need of explicitly invoking two extra comparisons. The code becomes:

```
1 bool our_ray_AABB_intersection(const Ray& r, double t_min, double t_max)
     const {
2     Vec3D ray_origin = r.get_ray_origin();
3
4     for (int i = 0; i < 3; ++i) {
5         double inv_direction_i = r.inv_direction[i];
```

```
        double t0 = (minimum[i] - ray_origin[i]) * inv_direction_i;
        double t1 = (maximum[i] - ray_origin[i]) * inv_direction_i;

        if (t0 > t1)
            std::swap(t0, t1);

        t_min = std::max(t0, t_min);
        t_max = std::min(t1, t_max);

        if (t_max <= t_min)
            return false;
        }
    return true;
}
```

Listing 5.3: Our ray/AABB intersection algorithm.

5.7 Implementing the BVH

The goal of the hierarchy is to surround each object in the scene with an axis-aligned bounding box (AABB), and test for ray/AABB before testing for ray/primitive intersection. The benefit is that the rays that miss the AABBs are cheaper to compute than a brute force search through the scene. The caveat is that rays that intersect the AABBs are more expensive to compute that a brute force search. Therefore, a good hierarchy is one that tightly encapsulates objects in the scene and is roughly balanced, reducing intersection tests and the cost of traversing the hierarchy. We start by discussing how we implement AABBs, and then we discuss how we construct the BVH.

5.7.1 Implementing AABBs

The core of any BVH is the bounding volume of its objects, which in our case are bounding boxes. AAABs are defined by two points: The maximum and minimum coordinates. Our `AABB` class is shown below.

```
class AABB {
public:
    AABB(const point3D& MIN, const point3D& MAX) : minimum(MIN), maximum(
       MAX) {
        // Calculate the centroid of the AABB
        ...
    }

    bool intersection(const Ray& r, double t_min, double t_max) const {
        // CHOOSE THE PREFFERRED METHOD FOR RAY/AABB INTERSECTION TEST
```

```
    }

    point3D get_min() const { return minimum; }
    point3D get_max() const { return maximum; };
    point3D get_centroid() const { return centroid; }
private:
    point3D minimum;
    point3D maximum;
    point3D centroid;
};

AABB construct_surrounding_box(AABB box_0, AABB box_1) {
    point3D minimum(fmin(box_0.get_min().x(), box_1.get_min().x()),
                    fmin(box_0.get_min().y(), box_1.get_min().y()),
                    fmin(box_0.get_min().z(), box_1.get_min().z()));

    point3D maximum(fmax(box_0.get_max().x(), box_1.get_max().x()),
                    fmax(box_0.get_max().y(), box_1.get_max().y()),
                    fmax(box_0.get_max().z(), box_1.get_max().z()));

    return {minimum, maximum};
}
```

Listing 5.4: The `AABB` class.

5.7.2 Constructing the Hierarchy

Since we will test the intersection of the rays with each node in the BVH, we make the BVH a subclass of the `Primitive` class. As we discussed in §2.3.2, there are many ways to build a BVH. Specifically, we have the following design choices:

- How the split axis is chosen.
- How to group the objects in the scene along the split axis.
- The choice of the comparator, which decides how objects are compared before being grouped.

Let's discuss how we implement each. There are many ways to choose the split axis. For example, we can choose it at random [Peter Shirley, 2023b], or choose the axes in a rotational manner (first X, then Y, then Z) [Shirley et al., 2021]. We implement both of these choices. Second, we want to group the objects in the scene so that we ultimately have a binary tree that is roughly balanced and has the boxes of the sibling subtrees overlap as little as possible. One heuristic to achieve this is to sort the primitives along an axis before dividing them into two sublists. We can use partial sorting instead of a full sort, and this can be more efficient. Finally, regarding the comparator's choice, we can compare bounding boxes by their centroid coordinate [Shirley et al., 2021] [Pharr et al., 2023], by their minimum coordiante [Peter Shirley, 2023b], or by their maximum coordinate. We implement these three choices.

The hierarchy is constructed recursively. We start by the first primitive in the list, make it a root of the BVH, and bound it with an AABB. Then, we bound each of its left and right children with a box. We continue in this manner until we reach the leaves of the tree. Invoking the heuristics that we discussed at every recursive call. All in all, the BVH implementation is shown below:

```
class BVH : Primitive {
   BVH(const Primitives_Group &list) :
            BVH(list.primitives_list, 0, list.primitives_list.size(),
               0.0, 0.0, 0) {}

    BVH(const std::vector<std::shared_ptr<Primitive>> &src_objects,
        size_t start, size_t end, double time0, double time1, int
           axis_ctr) {
    auto objects = src_objects;

    // Choose an axis at random, or keep rotating between the axes
    // int axis = random_int_in_range(0,2);
    int axis = axis_ctr % 3;

    auto comparator = // CHOOSE A COMPARATOR

    // Length of the primitives list
    size_t N = end - start;

    // Build the tree
    if (N == 1) {
            // Set both the left and right pointers of the current node
               to the single object, the root
            left = right = objects[start];
        } else if (N == 2) {
            // Compare the two objects using the chosen comparator
            if (comparator(objects[start], objects[start+1])) {
                left = objects[start];
                right = objects[start+1];
            } else {
                left = objects[start+1];
                right = objects[start];
            }
        } else {
            auto m = start + N/2;

            // Sort objects using the respective comparator...
            // std::sort(objects.begin() + start, objects.begin() + end,
               comparator);

```

```
37              // ... or use partial sort (usually faster and does the job)
38              // std::partial_sort(objects.begin() + start, objects.begin()
                    + m, objects.begin() + end, comparator);
39
40              // ... or use nth_element (fastest!)
41              std::nth_element(objects.begin() + start, objects.begin() + m
                    , objects.begin() + end, comparator);
42
43              // Recursively construct the left and right subtrees
44              left = std::make_shared<BVH>(objects, start, m, time0, time1,
                     axis_ctr+1);
45              right = std::make_shared<BVH>(objects, m, end, time0, time1,
                    axis_ctr+1);
46          }
47
48          AABB box_left, box_right;
49
50          BBOX = construct_surrounding_box(box_left, box_right);
51      };
52  }
```

Listing 5.5: The `BVH` class.

As the code snippet shows, we can flexibly add any comparator of our choice. For example, to compare bounding boxes by their centroid coordinates, we can use the following function.

```
1  inline bool compare_AABBs_centroid_coord(const std::shared_ptr<Primitive>
       a, const std::shared_ptr<Primitive> b, int axis) {
2      AABB box_a;             // AABB of object a
3      AABB box_b;             // AABB of object b
4
5      // Compare centroids
6      return box_a.get_centroid().V[axis] < box_b.get_centroid().V[axis];
7  }
```

Listing 5.6: Centroid comparator

Similarly, we can choose whatever heuristic (`sort()`, `partial_sort()`, or `nth_element()`) we want to group the primitives along the split axis.

5.7.3 Optimizing the Hierarchy Construction

In the BVH implementation just shown (Listing 5.5), we are sending the full list of primitives through the recursive calls at each recursion level; however, this is very inefficient. Instead, we can send partial lists. To do this, we add the following two lines after **Line 42**:

```
// Don't send the full lists, just send partial copies of them
    auto first_half = std::vector<std::shared_ptr<Primitive>>(objects.
       begin(), objects.begin() + m);
    auto second_half = std::vector<std::shared_ptr<Primitive>>(objects.
       begin() + m, objects.end());
```

Listing 5.7: Optimized BVH construction

This change greatly enhances the rendering speed. We will see this in the results chapter.

5.8 Parallel Rendering with OpenMP

In §2.5.2, we explained why rendering is embarrassingly parallel. A key idea to parallelize rendering is that the computation of each pixel's color is independent. This means that what what we are truly parallelizng is the render loop (Algorithm 4).

Our first strategy uses OpenMP's *worksharing-loop construct*:

Algorithm 5 Parallel Render Loop with Using the Parallel For Loop Strategy

```
#pragma omp parallel for collapse(2) schedule(dynamic)
for each pixel in the image do
    for s ← 0 to samples_per_pixel - 1 do
        Construct a ray from the camera origin in the direction of the pixel
        pixel color ← ∑_{s=0}^{samples_per_pixel−1} radiance()
    end for
end for
```

We use the *collapse(2)* clause to indicate that we wish to collapse the two *for loops* into a single iteration space and divide the work based on the scheduling method. A rule of thumb is to use *static* scheduling if each iteration takes roughly equal time and use *dynamic* scheduling otherwise. In rendering, the workload significantly differs across various iterations, so we use *dynamic* scheduling.

Another parallelizaiton strategy is to divide the workload over columns. To do this, we first decide the number of threads to use and then determine the number of columns to process per thread. We then initiate a parallel region. Inside the parallel region, we retrieve the ID of the current thread. We then go over the columns that this specific thread is responsible for. Figure 5.1 shows a simplified example of how this strategy is used.

The last parallelization strategy we implement involves the use of parallel tasks. Given an image size, we consider its total number of pixels (image width × image height). We start by dividing the image into a specific number of regions. The master thread enters the parallel region and encounters the task directive, pushes the stack to the stack queue, and any available thread in the thread pool can process this task[3].

[3]This parallelism style is inspired by OpenMP's official guide and other OpenMP texts [Eijkhout, 2017] (chapter 24)

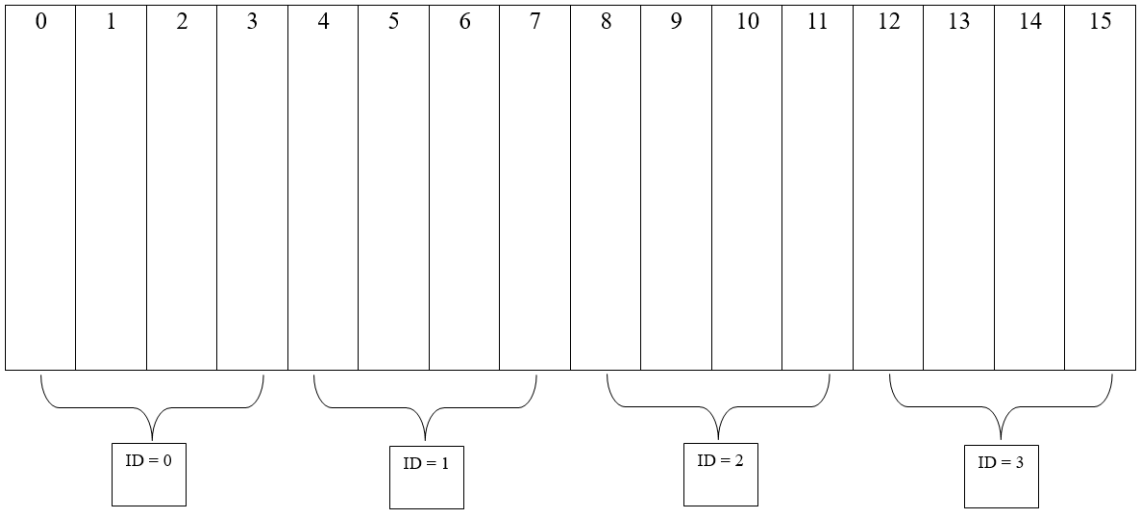


Figure 5.1: *Parallel Columns Strategy.* In this example, assume the image width (i.e., the number of columns) is 16 and the maximum number of threads available is 4. The workload follows the figure.

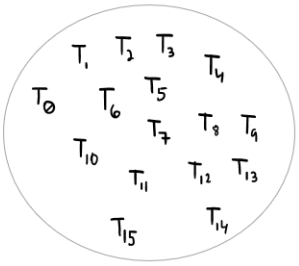
A key note is that the iterations of the loop (and thus the creation of new tasks) are executed one after another without waiting for the newly created tasks to finish. For an example of this, see Figure 5.2. The full algorithm is shown in Listing 5.8[4].

Regardless of the parallelization strategy that we use, we need to be careful when writing the colors to the output file. Since some threads could finish before others, we don't want the pixels of our final image to be in the wrong order. For this purpose, we make use of C++ `Vector` class and initialize a matrix, `pixel_colors`, whose cells represent the colors of each pixel in the image (**line 3**). In the color accumulation step in the render loop, we write the color of the pixel to this matrix instead of the final image (**line 28**). We then iterate over the matrix and write the colors to the final image file in the correct order.

```
1  void render(Scene_Information scene_info) {
2      ...
3      std::vector<std::vector<Color>> pixel_colors(image_height, std::vector<Color>(
           image_width, Color(0, 0, 0)));
4      int num_of_pixels = image_width * image_height;
5      #pragma omp parallel
6      #pragma omp single
7      {
8          int num_regions = // SET TO DESIRED;
9          int iter_per_region = num_of_pixels / num_regions;
10         for (int region = 0; region < num_regions; ++region) {
11         #pragma omp task
12             {
```

[Trobec et al., 2018] (chapter 3).

[4]It might be strange that we use a *single directive* after a *parallel directive*; however, this is the correct way to ensure that only a single thread creates the tasks and pushes them into the queue. If we omit the *single directive*, we will get an unexpected behavior [Eijkhout, 2017].



Thread Pool

Thread	Region	Pixels Interval
T_8	0	[0,5000)
T_2	1	[5000,10000)
T_6	2	[10000, 15000)
T_0	3	[15000,20000)
T_{12}	4	[20000,250000)
T_{19}	5	[25000,30000)
T_{13}	6	[30000, 35000)
T_{11}	7	[35000, 40000)
T_{10}	8	[40000, 45000)
T_5	9	[45000, 50000)
T_7	10	[50000, 55000)
T_3	11	[55000,60000)
T_4	12	[60000, 65000)
T_{14}	13	[65000,70000)
T_{15}	14	[70000,75000)
T_1	15	[75000,80000)
T_0	16	[80000,85000]
. . .		
T_{12}	31	[155000, 160000]
. . .		
T_8	112	[560000,565000)
. . .		. . .
T_{14}	127	[635000,640000)

Figure 5.2: *Parallel Tasks Strategy.* Here, assume we are given an 800×800 image (=640,000 pixels) and that the number of threads available is 16. The table shows one possible thread allocation to regions.

```
                int start_iter = region * iter_per_region;
                int end_iter = (region + 1) * iter_per_region;

                for (int iter = start_iter; iter < end_iter; ++iter) {
                    int i = iter % image_width;
                    int j = iter / image_width;

                    Color pixel_color(0.0, 0.0, 0.0);
                    for (int s = 0; s < samples_per_pixel; ++s) {
                        auto u = (i + random_double()) / (image_width - 1);
                        auto v = (j + random_double()) / (image_height - 1);

                        Ray r = cam.get_ray(u, v);
                        pixel_color += radiance_mixture(r, world, lights, max_depth);
                    }
                    pixel_colors[j][i] = pixel_color;
                }
            }
        }
    }
    ...
}
```

Listing 5.8: *Parallel Tasks Strategy.* This snippet shows part of the main `render()` function.

5.9 GPU Rendering with CUDA

In our literature review chapter (§2.6), we explored how the GPU can be used to accelerate rendering, and we discussed how CUDA can be used to achieve this acceleration. In this section, we will explain how we implemented the GPU version of our rendering engine.

Our code thus far has been written with C++; hence, it makes sense that our rendering engine will have two different versions: One version, written in C++, that runs on the CPU, and another version, written in C-style and is CUDA compatible, that runs on the GPU. Our CPU version uses *doubles* for computations; however, it is more efficient to use *floats* for the GPU. Moreover, we no more have the luxury to use most of C++ functionality with CUDA, such as the standard library, which includes `std::vector` and smart pointers. For this reason, we started by rewriting the entire engine, using arrays and raw pointers, and updating necessary variable to be of *float* type. Additionally, almost all functions must be updated to have the `__device__` qualifier (or sometimes the `__global__` identifier), which signifies that the associated function is to be executed in the device memory, the memory associated with the GPU.

There are more significant changes that had to be made. We outline some of them in the following subsections.

5.9.1 Launching Rendering Commands

We start by allocating an image-sized frame buffer on the host. This is similar to what we did when we implemented multithreading on the CPU. This buffer will hold the colors of the pixels, which will be calculated by the GPU. This is one way to move the data from the device memory to the host memory. Essentially, what this means is that rendering will be done on the GPU, while other processes, such as loading meshes and writing the final rendered image to the output file, will be done on the CPU. The reason for this design choice is that CUDA doesn't support file I/O, so it is suitable to handle such processes on the CPU.

Threads are launched by a kernel call. To orchestrate the computation of each pixel of the output image independently, we use the the threads and blocks identifiers: `threadIdx`, `blockIdx`, and `blockDim`. The render loop then continues as we have seen before (§5.4). Lastly, before entering the multisampling loop, we must initialize the state of the random number generator for each pixel in the image grid. Ultimately, the rendering code on the GPU will look like this:

```
1  __global__ void render(Vec3D *pixel_colros, int image_width, int image_height, int
       sanples_per_pixel, Camera **camera, Primitive **lst, curandState *rand_state) {
2      int i = threadIdx.x + blockIdx.x * blockDim.x;
3      int j = threadIdx.y + blockIdx.y * blockDim.y;
4
5      if((i >= image_width) || (j >= image_height))
6          return;
7
8      int pixel_index = j * image_width + i;
9
10     // Unique seed for each pixel
```

```
11      curand_init(3019 + pixel_index, 0, 0, &rand_state[pixel_index]);
12      curandState local_rand_state = rand_state[pixel_index];
13
14      // CODE INSIDE THE MULTISAMPLING LOOP REMAINS THE SAME
15      ...
16
17      pixel_colros[pixel_index] = pixel_color;
18  }
```

Listing 5.9: GPU version of the `render()` function. See Listing 5.8 for the code inside the multisampling loop.

The `render()` kernel is launched with the desired number of blocks and threads per block:

```
render<<<blocks,threads>>>(...);
```

5.9.2 Sending Meshes to the GPU

Meshes are read on the host and then loaded to the device for rendering; this process can be summarized as such:

1. Initialize variables to store the number of vertices and faces of a 3D model read from an .obj file. These will help us allocate enough space for the data in the host memory.
2. Using a helper function (`get_num_vertices_and_faces()`) that preprocesses the .obj file to be loaded, we determine the number of vertices and faces that the model has. This allows us to initialize two dynamically allocated arrays: `vertices` and `faces`.
3. Populate these arrays with the data obtained from loading the model from the .obj file (by calling the `load_model()` function).
4. Send this information from the host memory to the device memory. To accomplish this, we allocate memory on the device via `cudaMalloc` and copy the `faces` array to the device memory using `cudaMemcpy`, allowing it to be rendered on the GPU.

A key questions is why did we create a dynamic array for `vertices` if we don't send it to the device memory along with the `faces` array? It's important to remember that CUDA doesn't support the use of `std::vector`, yet we need a way to build the `faces` array (which are triangles), and this relies on having an array of vertices.

5.9.3 Freeing Up Memory

As mentioned in the introduction of this section, CUDA doesn't support smart pointers. This means that we need to manually delete any memory we allocate. For this reason, we implemented *destructors* for all classes that allocate memory. For example, when a triangle is constructed, it also creates a new material pointer; hence, the *destructor* of the `Triangle` class will take care of freeing this memory. Similar care must be taken to free any other dynamically allocated memory, such as the `vertices` and `faces` arrays that we discussed in §5.9.2.

5.9.4 Implementing a BVH on the GPU

There are some changes that have to be made in order to implement a BVH on the GPU when compared to our CPU implementation (§5.7). We list these changes below:

1. Since we can't use the standard C++ library with the CUDA code, we rely on the `thrust` library and use `thrust::sort()` instead of `std::nth_element()`.
2. We need to create the partial lists that will be sent in the recursion calls differently. In the CPU implementation, we relied on the copy constructor of the `vector` class; for the GPU implementation, we will allocate the partial lists dynamically. This means that we have to ensure that we allocate enough memory for each.
3. Since we can't use smart pointers, we resort to using raw pointers, specifically for representing the left and right subtrees of the hierarchy.
4. The `BVH` class of the GPU code must implement a destructor to free the dynamically allocated memory.

Some of the changes mentioned above are highlighted in the BVH construction. Particularly, the GPU version differs from the CPU version in the case when there are more than two objects remaining to be inserted into the hierarchy within the scene. The listing below shows the difference between the two versions in constructing the hierarchy (compare it to Listing 5.5 (**Line 32 - Line 45**) after the `else` statement):

```
1  else {
2              thrust::sort(l, l + n, comparator);
3              auto m = n / 2;
4
5              // Allocate memory for each half
6              Primitive **first_half = new Primitive*[m];
7              Primitive **second_half = new Primitive*[n - m];
8
9              // Copy the first half
10             for (int i = 0; i < m; ++i)
11                 first_half[i] = l[i];
12
13             // Copy the second half
14             for (int i = 0; i < n - m; ++i)
15                 second_half[i] = l[m + i];
16
17            // Launch recursion
18             left  = new BVH(first_half, m, time0, time1, local_rand_state
                   , axis_ctr + 1);
19             right = new BVH(second_half, n - m, time0, time1,
                   local_rand_state, axis_ctr + 1);
20         }
```

Listing 5.10: GPU version of the `BVH()` function.

Chapter 6

Results

In this chapter, we benchmark the acceleration techniques that we implemented and analyze their effectiveness in accelerating rendering. The following are links to our repositories:

- CPU Version
- GPU Version

6.1 Test Environment

6.1.1 Software

Component	Specification
OS Manufacturer	Microsoft Corporation
Tech Stack	C++, OpenMP, CUDA
gcc version	13.1.0
OpenMP Version	4.5

Table 6.1: System specifications: software.

6.1.2 Hardware

Component	Specification
Processor	11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz, 2304 Mhz, 8 Core(s), 16 Logical Processor(s)
GPU	NVIDIA GeForce RTX 3080
CUDA Version	release 12.3, V12.3.107

Table 6.2: System specifications: hardware.

Figure 6.1: *The Effect of Importance Sampling.* The scene on the left uses importance sampling, resulting in significantly less noise compared to the scene to the right, which doesn't use it. Both scenes were rendered with 500 samples-per-pixel.

6.2 Faster Rendering with Importance Sampling

The result of the probabilistic framework we discussed in section §5.5 is that we can optimize the allocation of computational resources by favoring paths that contribute more significantly to the final image, thereby reducing variance and improving convergence speed. This is shown in Figure 6.1. In essence, with an equal samples-per-pixel count, we utilized importance sampling to render a less noisy image. To further reduce the noise in the image to the right, we'll need more samples, meaning we have to run our engine for a longer time. We won't provide an exact speedup measurement since the two scenes are materially different (i.e., they reflect light differently), making it difficult to determine when noise levels are similar.

6.3 Faster Intersection Tests

In this section, a comparison between all the ray/primitives intersection algorithms implemented in our engine (§5.6) is given. Moreover, we show how optimizing the ray/AABB intersection algorithm leads to faster rendering. A summary of the results is shown at the end.

6.3.1 Ray/Sphere Intersection Performance Comparison

Let's start by inspecting the results of the two ray/sphere intersection algorithms we implemented.

Number of Tests	Algebraic Solution (secs)	Geometric Solution (secs)	Speedup from the Algebraic Solution
50,000,000	4.827	6.057	+25.48%
150,000,000	14.482	18.238	+25.94%
300,000,000	28.785	36.467	+26.69%
500,000,000	49.054	61.824	+26.03%
750,000,000	72.834	93.623	+26.68%

Table 6.3: *Ray Intersection Tests Comparison.* Each results is taken as the median of 5 runs.

The difference between the two algorithms is evident. The algebraic solution outperforms the geometric solution by a considerable factor.

6.3.2 Ray/Triangle Intersection Performance Comparison

Let's examine the results of our ray/triangle intersection tests, which we present in Table 6.4. Each result represents the median of five runs. We use the runtime of Möller-Trumbore as a reference, so the relative runtime of Snyder & Barr (denoted T_{Snyder}) to Möller-Trumbore (denoted $T_{\text{Möller}}$) is given by the expression $\frac{T_{\text{Snyder}} - T_{\text{Möller}}}{T_{\text{Möller}}}$.

Number of Tests	Snyder & Barr (secs)	Möller-Trumbore (secs)	Speedup from Möller-Trumbore
1,000,000	0.152	0.148	+2.7%
50,000,000	6.938	6.911	+0.39%
150,000,000	19.682	19.610	+0.37%
300,000,000	39.350	39.243	+0.27%
500,000,000	65.450	65.379	+0.10%
750,000,000	98.384	98.235	+0.15%

Table 6.4: *Ray Intersection Tests Comparison.* Each results is taken as the median of 5 runs.

We observe a negligible difference between the two ray/triangle intersection algorithms. There are not many papers that compare the two algorithms together. For example, in their original paper, Möller and Trumbore [Möller and Trumbore, 2005] compared their algorithm to Badouel's algorithm. The research carried out by Shumisky [Shumskiy, 2013] compareed Möller-Trumbore algorithm with the watertight Ray/Triangle intersection algorithm [Woop et al., 2013].

6.3.3 Ray/AABB Intersection Performance Comparison

Finally, we present the results of ray/box intersection tests on the scene shown in Figure 6.2, but this time using a plot (Figure 6.3). We see some interesting results:

Figure 6.2: *A Rabbit and a Teapot inside a Cornell Box.* This scene was used to compare ray/AABB intersection algorithms. Tests used different samples-per-pixel counts (2, 4, 8, 16, 32, & 64).

1. The *optimized slab method* and *Kensler's method* are both the most efficient ray/AABB intersection algorithms. Using either, we got a speedup of more than 2 times compared to using the *slab method*.
2. Our idea of optimizing Tavian's *branchless method* has actually paid off: We can notice that our method is ≈10% faster thanks to replacing `max()` and `min()` calls with a `swap()` call.

6.3.4 Summary

As mentioned in our literature review (§2.4), intersection tests often become the bottleneck in the performance of a rendering engine; thus, optimizing them is necessary. The results show that our optimization of the ray/AABB intersection algorithm (§5.6.2) has accelerated the render speeds by two times. Speed improvements were also observed when using the *algebraic solution* for ray/sphere intersection. Finally, our tests showed no significant speed difference between the *Möller–Trumbore* and *Snyder & Barr* algorithms. Yet, as noted in §2.4, the *Möller–Trumbore* algorithm is preferred due to potential visual artifacts with the latter.

6.4 BVH

6.4.1 Benchmarking BVH Implementations

We will test our BVH implementations (§5.7) using the scenes shown in Figure 6.4. Let us start with Figure 6.4a, because it has fewer triangles. We wish to answer the following questions:

- Does passing a partial list through the recursive calls result in a noticeable speedup compared to

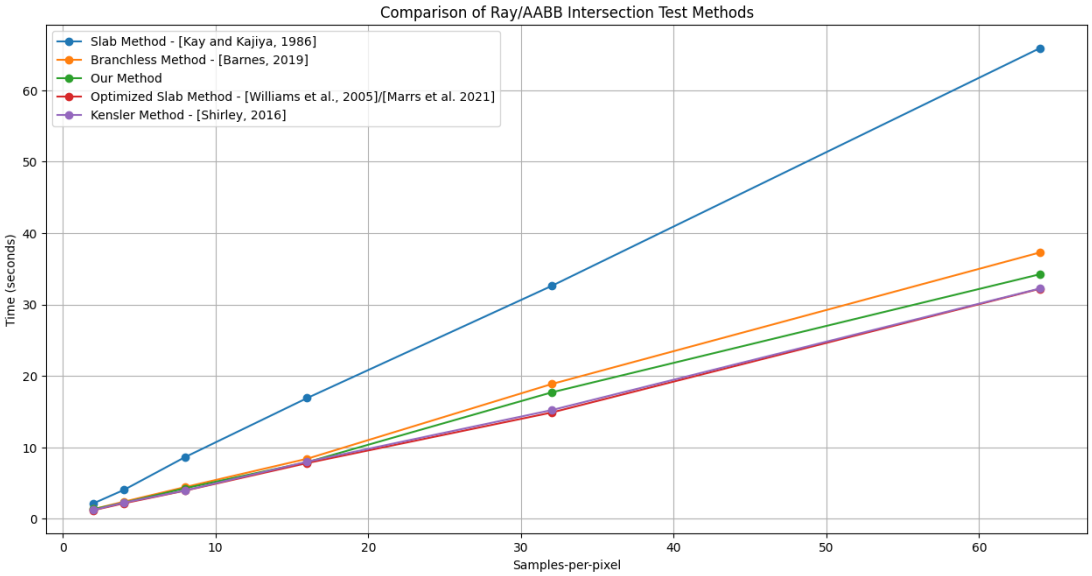


Figure 6.3: *Ray/AABB Intersection Methods Comparison. Williams et. al.'s* and *Kensler's* methods are faster than other intersection tests, aligning with our discussion in the literature review chapter (§2.4).

passing the entire list?
- Does the sorting operation matter?
- Can the way we choose the split axis make our engine run faster?
- Does it matter whether we compare bounding boxes by their minimum, maximum, or centroid coordinates?

The results are shown in Table 6.5. Below, we list some observations, many of which address the questions posed at the beginning of this section

- By modifying the basic BVH implementation, we were able to approximately half the total render time of the first scene.

To check this, observe the first row in the table, which represents the result obtained using the basic BVH implementation (§5.7.2), and compare it to the last two rows, which represent the results obtained using our modified BVH implementation (§5.7.3). The results clearly indicate that when passing the full list of primitves through the recursive calls, the engine wastes a lot of time in building the BVH. This may be acceptable for a scene with few triangles (Figure 6.4a); however, as we will see later, the BVH, which is supposedly an acceleration technique, becomes the bottleneck of the performance.

- The modified BVH implementation builds the hierarchy significantly faster than the naïve one.

This can be observed by comparing the *Build Runtime* (*secs*) column of first half of the table with the second half, and it further supports our previous claim in regards to which implementation is more efficient.

(a) *Scene* 1. In addition to the axis aligned rectangles, this scene is made up of 16,256 triangles to represent the two meshes.

(b) *Scene 2.* In addition to the axis aligned rectangles, this scene is made up of 106,290 triangles to represent the two meshes.

Figure 6.4: *Test Scenes for BVH Implementations. Scene 2* is significantly more complex than *Scene* 1, because the first contains 6 times the number of triangles.

- Choosing the axis in a rotational manner is a more efficient heuristic than choosing it at random.

To see this, we simply compare the first 3 entries with the last three entries of each of the 6 sections in the table. If we do so, we can see there is approximately a 10% speedup when rotating between axes. Choosing which axis to split on by rotating between them is used in some of the BVH implementations that we discussed [Shirley et al., 2021].

- The sorting operation matters. Partial sorting via `nth_element()` is much faster than `sort()`.

The table tells us that this improvement can be between 5% to 17%, depending on the other parameters.

- Sorting the bounding boxes by the minimum coordinates results in slower render time compared to when sorting them by their centroid or maximum coordinates.

Sorting by the centroid coordinate is commonly adopted by most BVH implementations that we have seen in the literature review chapter [Lauterbach et al., 2009] [Pharr et al., 2023].

This concludes our analysis of the BVH implementations performance in rendering the scene in Figure 6.4a. These observations generally hold for many scenes that we have tested our BVH implementations on.

Let us now verify how this holds up for a more complicated scene. To do that, we investigate the difference

BVH Implementation	Split Axis Choice	Split Strategy	Comparator	Build Runtime (secs)	Total Runtime (secs)
Full-List Recursive Calls	Random	std::sort	Min	2.883	8.450
Full-List Recursive Calls	Random	std::sort	Max	2.891	7.530
Full-List Recursive Calls	Random	std::sort	Centroid	2.913	7.749
Full-List Recursive Calls	Rotational	std::sort	Min	2.857	8.249
Full-List Recursive Calls	Rotational	std::sort	Max	2.919	7.395
Full-List Recursive Calls	Rotational	std::sort	Centroid	2.903	7.386
Full-List Recursive Calls	Random	std::partial_sort	Min	2.838	8.392
Full-List Recursive Calls	Random	std::partial_sort	Max	2.857	7.655
Full-List Recursive Calls	Random	std::partial_sort	Centroid	2.829	7.587
Full-List Recursive Calls	Rotational	std::partial_sort	Min	2.831	8.152
Full-List Recursive Calls	Rotational	std::partial_sort	Max	2.832	7.333
Full-List Recursive Calls	Rotational	std::partial_sort	Centroid	2.832	7.251
Full-List Recursive Calls	Random	std::nth_element	Min	2.713	8.057
Full-List Recursive Calls	Random	std::nth_element	Max	2.728	7.551
Full-List Recursive Calls	Random	std::nth_element	Centroid	2.727	7.677
Full-List Recursive Calls	Rotational	std::nth_element	Min	2.711	7.965
Full-List Recursive Calls	Rotational	std::nth_element	Max	2.705	7.292
Full-List Recursive Calls	Rotational	std::nth_element	Centroid	2.700	7.227
Partial-List Recursive Calls	Random	std::sort	Min	0.288	5.950
Partial-List Recursive Calls	Random	std::sort	Max	0.275	5.147
Partial-List Recursive Calls	Random	std::sort	Centroid	0.285	5.268
Partial-List Recursive Calls	Rotational	std::sort	Min	0.302	5.582
Partial-List Recursive Calls	Rotational	std::sort	Max	0.309	4.879
Partial-List Recursive Calls	Rotational	std::sort	Centroid	0.308	4.829
Partial-List Recursive Calls	Random	std::partial_sort	Min	0.241	5.939
Partial-List Recursive Calls	Random	std::partial_sort	Max	0.228	5.156
Partial-List Recursive Calls	Random	std::partial_sort	Centroid	0.234	5.195
Partial-List Recursive Calls	Rotational	std::partial_sort	Min	0.225	5.617
Partial-List Recursive Calls	Rotational	std::partial_sort	Max	0.234	4.736
Partial-List Recursive Calls	Rotational	std::partial_sort	Centroid	0.234	4.800
Partial-List Recursive Calls	Random	std::nth_element	Min	0.099	5.792
Partial-List Recursive Calls	Random	std::nth_element	Max	0.105	5.296
Partial-List Recursive Calls	Random	std::nth_element	Centroid	0.10	5.077
Partial-List Recursive Calls	Rotational	std::nth_element	Min	0.099	5.346
Partial-List Recursive Calls	Rotational	std::nth_element	Max	0.099	4.513
Partial-List Recursive Calls	Rotational	std::nth_element	Centroid	0.099	4.556

Table 6.5: BVH implementations performance comparison.

between the naïve BVH implementation and our improved BVH implementation in rendering the scene shown in Figure 6.4b. The results are shown in Table 6.6. Our implementation can build the hierarchy approximately 230 times faster, resulting in faster rendering speeds. Moreover, the naïve BVH implementation spends 96% of the engine's total rendering time in constructing the hierarchy, which is very wasteful and makes it the bottleneck of the performance.

6.4.2 Summary

Our results show that randomly choosing which axis to split on is inefficient; rather, rotating between the axes on each split leads to faster rendering times. Moreover, partially sorting the objects along the split axis accelerates rendering and doesn't yield poorer quality trees, so it should be used instead of fully

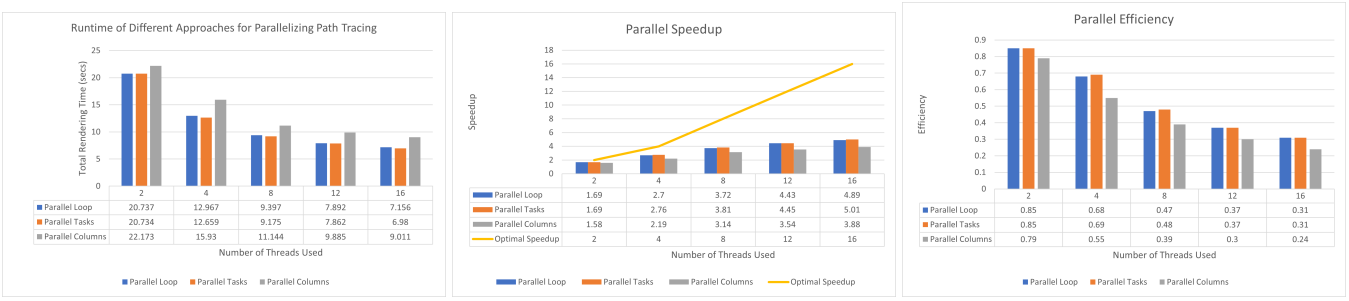
BVH Implementation	Split Axis Choice	Split Strategy	Comparator	Build Runtime (secs)	Total Runtime (secs)
Full-List Recursive Calls	Random	std::sort	Min	0.834	6.032
Partial-List Recursive Calls	Rotational	std::nth_elemnt	Centroid	191.682	197.941

Table 6.6: *Speedup Gained from Our BVH Implementation.* Both scenes were rendered with 10 samples-per-pixel. Our implementation is considerably faster.

Figure 6.5

sorting the objects. Finally, sorting by the centroid or the maximum coordinate of the bounding boxes is both more efficient than sorting by the minimum coordinate. In a nutshell, we were able to improve the BVH implementations given in some references ([Shirley et al., 2021] and [Peter Shirley, 2023b]).

Further improvements are still possible. Other common choices for which axis to split on include trying each axis and choosing the one that leads to the least total surface area of the child nodes' bounding boxes or selecting select the axis corresponding with the largest extent when projecting the bounding box centroid for the current set of primitives [Pharr et al., 2023]. Both of these two choices are commonly used when implementing SAH-based BVHs (§2.3.2). However, it seems hard to beat the average $O(N)$ guaranteed (on average) by `nth_elemnt()` when it comes to partially sorting the objects along the split axis.



(a) *Total Runtime.* (b) *Speedup.* (c) *Efficiency.*

Figure 6.6: An analysis of the performance of three different parallelization techniques.

6.5 CPU Parallelism

6.5.1 Benchmarking the Parallel Strategies

In this section, we will present the results of parallelizing our path tracer using the three approaches we discussed in §5.8. We will examine how our findings compare to those of other researchers, which we discussed in our literature review chapter (§2.5.2).

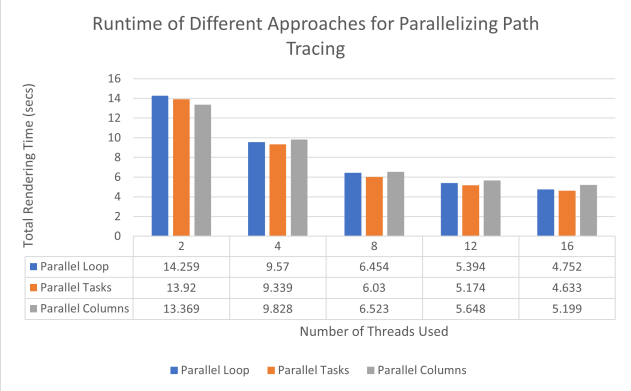
Our first evaluation is based on rendering the scene shown in Figure 6.5. The results are shown in Figure 6.6. We first note that the median serial execution time for the scene is 34.961 seconds[1]. The fastest strategy is using *parallel tasks*, resulting in a speedup of up to 5 times. We observe that the runtime decreases as we add more threads. This supports our claim that path tracing is an embarrassingly parallel problem.

The *parallel tasks* version performs slightly better than the *parallel loop* version. Research has shown that tasks-based parallelism is generally more optimal than nested[2] [Terboven et al., 2012]. In the literature review chapter, we have discussed that Part et al. [Parker et al., 2005] found that Parallelizing the render loop often leads to excessive synchronization overhead. They also discussed that one way of reducing the overhead is to assign groups of rays to each processor (see §2.5.2). This is similar to what we do: Spawned rays are assigned by generated tasks, which are assigned to free processors by the scheduler. The *parallel columns* strategy doesn't perform well in this particular scene. The reason for this is that Figure 6.5 is a square image (800 pixels × 800 pixels). We will see that the *parallel columns* strategy actually performs well once we have a rectangular image, where the number of columns exceeds the number of rows.

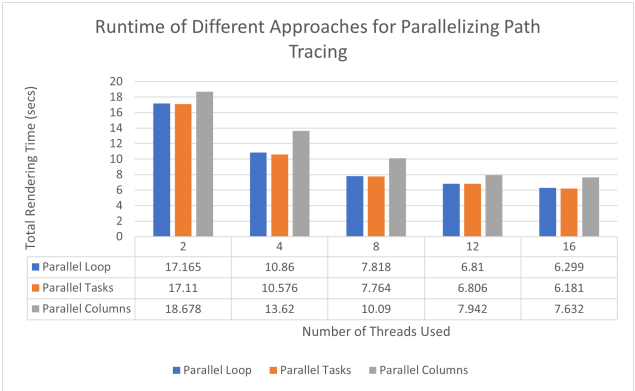
Looking at the plot in Figure 6.6b, we observe that speedup increases as we increase the number of threads,

[1]This is quite fast because we have optimized our scene with all the approaches that we suggested thus far. Had we not used a BVH, for example, the scene would have easily taken more than a couple of hours to render

[2]The authors found that "with the real-world application use cases ... tasking implementations may outperform the comparable worksharing implementations [i.e., #pragma omp for, which is a worksharing-loop construct]. This is particularly true for situations in which the load is not evenly balanced and a dynamic scheduling scheme is employed, in which case tasks may offer an even finer load balancing..." [Terboven et al., 2012] The last sentence is especially true in rendering, as some pixels are way more expensive to render than others.



(a) *Results of Parallelizing Scene* 1.



(b) *Results of Parallelizing Scene 2.*

Figure 6.7: *Total Runtime of Parallelization in Different Scenes.* We can observe that the total runtime after parallelization follows a trend similar to that shown in Figure 6.6.

but the rate at which it increases diminishes as we add more threads. That is, the speedup curve (of any of the three parallelization strategies) exhibits a sub-linear growth. This is common in practice; in fact, most parallel programs exhibits such phenomena [Navarro et al., 2014] (we discussed this in §2.5.2).

Looking at the plot in figure 6.6c, we observe that the *parallel tasks* version is the most efficient. When designing parallel algorithms, we care more about efficiency than speed, especially when using a low number of threads, because efficiency is an indicator of scalability. A key observation is that there are diminishing returns: Adding more parallel units decreases the runtime, but the ratio of the improvement decreases as the number of units increases. This is also common among parallel algorithms.

We repeated the tests on the two scenes shown in Figure 6.4; the results are shown in figure 6.7. Notice that we observe similar patterns for different scenes.

Let us compare our parallelization strategies to the one given by He et al. [He et al., 2021], which we discussed in §2.5.2. We rendered an almost identical scene[3] (see Figure 6.9). Albeit it is unfair to compare our version with theirs (we don't know whether they used importance sampling or not, whether they implemented other acceleration techniques, etc...); however, we see that our results are very close. The lower speedup we achieved could be due to the overhead caused by the serial part of our program, which could limit the parallel speedup attained. This is implied by *Amdahl's law* [Navarro et al., 2014].

It is worth noting that in this rectangular scene, the *parallel columns* strategy performs as good as the *parallel tasks* strategy, as shown in Table 6.9. This confirms our expectation that the *parallel columns* strategy performs well when the image width is larger than the image height.

[3]He et al.'s scene uses a checker texture as a ground material. A checker texture is nothing but a procedural function that alternates between odd and even numbers, generating a different color for each; hence, it shouldn't affect the runtime by a huge factor.

Threads	Real Time (seconds)	Speedup	Efficiency
16	18.085	5.44	0.34
8	24.34	4.04	0.51
4	35.38	2.78	0.70
2	55.136	1.78	0.89
1	98.373	1	1

Table 6.7: Speedup: *Parallel Tasks Parallelization.*

Threads	Real Time (seconds)	Speedup	Efficiency
16	227.899	6.333	0.396
8	298.451	4.836	0.604
4	452.150	3.192	0.798
2	772.112	1.869	0.935
1	1443.194	1	1

Table 6.8: *Parallel Speedup* [He et al., 2021]. Copied directly from He et al.'s research paper.

6.5.2 Summary

Our CPU parallel implementations has led to a speedup of 5∼7 times, depending on the scene rendered. These speedup results are generally a good indicator that our parallel implementation is good[4] [Pacheco, 2011]. The *parallel task* strategy implementation performs better than the two other parallel strategies, and the *parallel columns* strategy produces comparable results when the rendered image is rectangular.

6.6 GPU

In this section, we will analyze the efficiency of the GPU implementation of our rendering engine.

6.6.1 BVH: The CUDA Version

Our GPU implementation is similar to that given by Cioli et al. [Cioli et al., 2010] (see §2.6.2); except that instead of using a grid as an acceleration structure, we are using a BVH. Our BVH implementation, just like Cioli et al.'s grid implementation, minimizes the number of ray intersection calculations.

We will use Figure 6.8 to compare rendering performance on the GPU with and without a BVH. The results are shown in Table 6.10. They show that without a BVH, we perform 263 times more ray/triangle intersection tests [5]. The speedup from our CUDA BVH implementation is 32 times.

[4] The author states that "In many cases, obtaining a speedup of 5 or 10 is more than adequate" [Pacheco, 2011].

[5] We note that this doesn't account for ray/rectangles intersection tests, so the total number of ray/primitives intersection tests is higher.

Threads	Real Time (seconds)	Speedup	Efficiency
16	19.154	5.13	0.32
8	24.75	3.97	0.50
4	36.557	2.69	0.67
2	57.878	1.70	0.85
1	98.373	1	1

Table 6.9: Speedup: *Parallel Columns Parallelization.*

BVH Used	Number of Ray/Triangle Intersection Tests	Time (secs)
No	1,155,603,792	2.88
Yes	4,392,285	0.09

Table 6.10: *BVH Usage and Performance.* These results were obtained from rendering the scene with only one sample-per-pixel. Hence, the number of intersection tests significantly increases when rendering it at a higher samples-per-pixel count.

6.6.2 Kernel Launch Parameters

We tested the scene shown in Figure 6.8 using different kernel launch parameters (number of blocks and number of threads per block). The results (Table 6.11) show that the rendering time decreases until it reaches the minimum (88.735 seconds) when we reduce the number of blocks. Decreasing the number of blocks further increases the time. This process is empirical, and is usually different from one scene to another. The total number of threads is calculated by the following equation:

$$\text{Total number of threads} = (\text{Number of blocks}) \times (\text{Number of threads per block}) \tag{6.1}$$

Number of Blocks	Number of Threads per Block	Total Number of Threads	Time (secs)
160,801	4	643,204	163.663
80,601	8	644,808	107.848
40,401	8	323,208	107.848
40,401	16	646,416	88.735
26,867	24	644,808	90.893
20,301	32	649,632	90.054
10,201	64	652,864	118.203
5,151	128	659,328	118.799
2,601	256	665,856	112.698
1,326	512	678,912	107.851

Table 6.11: *Performance of Different Launch Parameters.* The scene (Figure 6.8) was rendered with 1,000 samples-per-pixel.

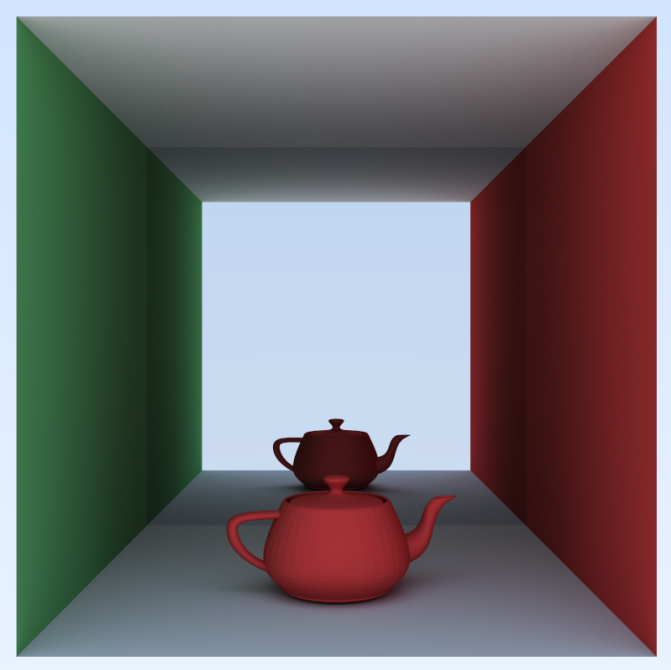


Figure 6.8: *A Teapot inside a Cornell Box.* This scene was rendered on the GPU and was used for both §6.6.1 & §6.6.2.

6.6.3 Summary

The GPU implementation greatly accelerates the speed of our engine. The implementation of a BVH proves to be vital, even when relaying on the GPU. Finally, some empirical experiments can lead to good launch parameters for our `render()` kernel function, but more sophisticated methods exists, such as using CUDA's occupancy calculator to determine more efficient launch parameters.

6.7 A Final Test

In this chapter, we discussed the results of the following approaches to accelerate rendering:

1. Importance sampling.
2. Faster ray intersection tests.
3. BVH.
4. CPU multithreading.
5. GPU multithreading.

For a final test, we will measure the speedup improvements achieved by implementing these techniques on a single scene[6]. This will give us a general idea of how combining these acceleration techniques can

[6]We will not show the speedup from importance sampling because as we mentioned in §6.2, it is hard to give an exact

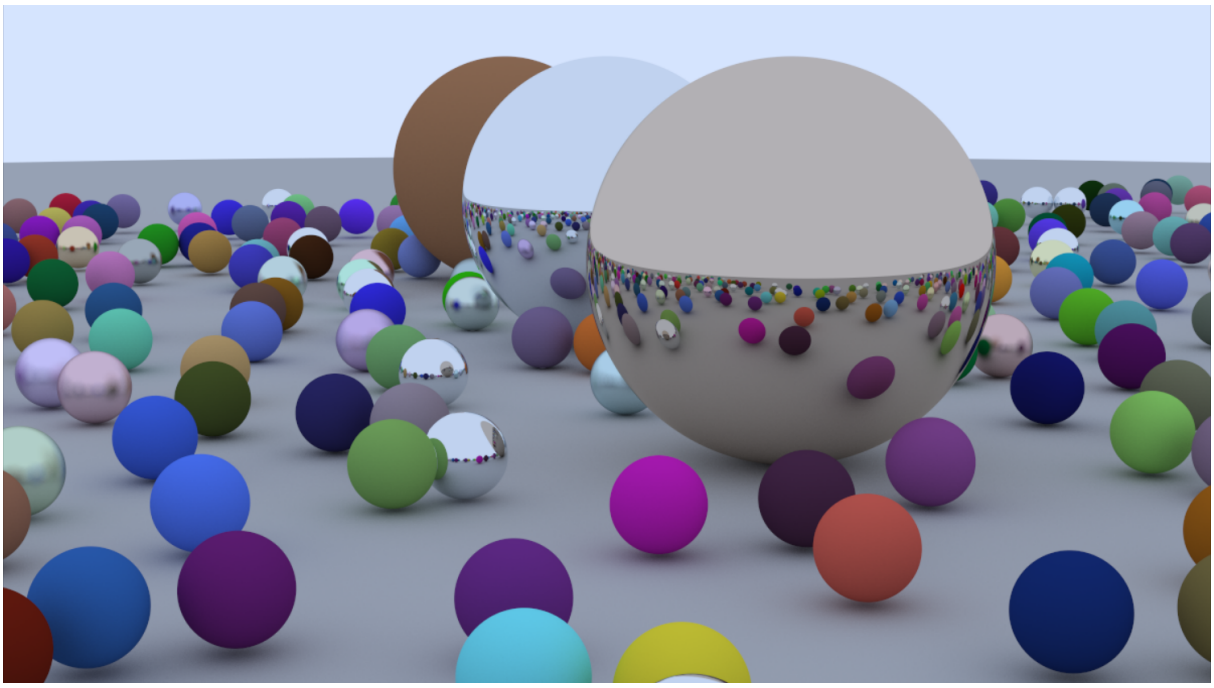


Figure 6.9: This scene was used in §6.5 and §6.7. We rendered this scene based on the one in Shirley's book [Peter Shirley, 2023a].

enhance rendering speeds. Our final test will be conducted on the scene shown in Figure 6.9 (the same as the one we used in §6.5). The results are shown in Table 6.12.

Importance Sampling	BVH	Faster Ray Intersection	CPU Multi-threading	GPU Multi-threading	Rendering Time (s)	Speedup
Used	-	-	-	-	1738.75	-
Used	Used	-	-	-	338.965	~ 5
Used	Used	Used	-	-	121.037	~ 14
Used	Used	Used	Used	-	22.246	~ 78
Used	Used	Used	-	Used	3.387	~ 513

Table 6.12: Rendering acceleration resulting from all the techniques we have implemented. The speedup is measured relative to the speed of the first row.

measurement of it.

Chapter 7

Conclusion

7.1 Summary of Our Work

Through the design and analysis of a path tracer's performance, this dissertations has successfully demonstrated the significant speedup (+500X) achieved by the implementation of various rendering acceleration techniques. The implemented techniques include:

1. **Importance Sampling** (§5.5)
2. **Faster Ray Intersection Tests** (§5.6)
3. **Bounding Volume Hierarchy (BVH) Implementation** (§5.7)
4. **CPU Parallelism** (§5.8)
5. **GPU Parallelism** (§5.9)

We have successfully achieved our aim of identifying and implementing techniques to accelerate rendering speeds (§1 & §1.1). Furthermore, we accomplished our two main goals, which we previously outlined in §1.2:

1. **Goal 1**: Implement an engine capable of rendering primitives (spheres, boxes, triangles, etc...) with lighting and materials.
2. **Goal 2**: Enhance the engine's performance, primarily utilizing parallelism and acceleration data structures to achieve speed improvements.

Our implementation and results were supported by the findings of our literature survey, where we discussed several rendering algorithms (§2.2.1), shading models (§2.2.2), and acceleration structures (§2.3), focusing on spatial data structures (§2.3.1). We explored the concept of bounding volume hierarchies (§2.3.2), examining the heuristics employed in their creation (§2.3.3). Moreover, we discussed how ray/object intersection tests can be optimized to accelerate rendering (§2.4), and we surveyed different approaches to parallelizing rendering and the challenges of load balancing in this context (§2.5.2). Next, we considered GPU parallelism through CUDA, discussing the implementation of rendering algorithms, and the construction of acceleration structures (§2.6.3). We then discussed heterogenous computing as an alternative method for accelerating rendering (§2.6.4).

We listed the details of the functional and non-functional requirements of our research, and we explained the strategy we adopted to evaluate our work (§3.3). We also discussed the implementation of our rendering engine, and examined many of the optimization techniques that we employed (§5).

The findings of this dissertation can be summarized as follows:

1. **Importance Sampling**: The allocation of computational resources can be optimized by favoring paths that contribute more significantly to the final image, which can reduce noise and improve rendering speeds (§6.2).
2. **Faster Intersection Tests**.
   (a) For the ray/sphere intersection, the *algebraic* solution was shown to be faster than the *geometric* solution (§6.3.1).
      - The *algebraic solution* seems to be the standard ray/sphere intersection routine as it is used in many rendering engines (e.g., PBRT and Mitsuba 3).

   (b) For the ray/triangle intersection, both *Möller–Trumbore* and *Snyder & Barr* algorithms showed similar speed results, but the *Möller–Trumbore* algorithm performed slightly better (§6.3.2).
      - The most commonly used ray/triangle intersection algorithm is *Möller–Trumbore*.

   (c) For the ray/Axis-Aligned Bounding Box (AABB) intersection, both *Kensler's method* and the *optimized slab method* resulted in doubling the rendering speed (§6.3.3).
      - PRBT uses the *optimized slab method*[1].
3. **BVH**: For the CPU and GPU implementations, minimizing the size of the list passed at each recursive level yielded a significant decrease in the BVH build time, resulting in faster rendering speeds. Speed improvements were also evident when choosing a different split axis at each recursion call and grouping the objects by the centroids or maximum coordinates of their bounding boxes instead of the minimum coordinates. Lastly, for the CPU implementation, it is better to rely on a partial sort via `nth_element()` instead of a full sort (§6.4).
4. **CPU Parallelism**: Among the three parallel strategies used, task-based parallelism yielded the best speedup results (§6.5).
5. **GPU**: Our GPU implementation, when combined with all the preceding acceleration techniques except CPU parallelism, achieves

   a speedup of more than 6X over a 16-threaded CPU implementation (and more than 500X over a single-threaded CPU implementation),

   without any loss in image quality (§6.6). This demonstrates that path tracing is an embarrassingly parallel algorithm and highlights the cumulative impact of the optimizations we have implemented.

7.2 Our Contributions

Writing a rendering engine from scratch is no simple task. To make this goal achievable, we drew inspiration from the *Ray Tracing in One Weekend* book series written by Peter Shirley, which is widely regarded as a standard and valuable resource for learning computer graphics due to its hands-on approach to implementing a rendering engine. However, the code of *Ray Tracing in One Weekend* is neither optimized

[1] We note that their code is more robust as they guard against precision problems.

for fast rendering, nor does it render any complex objects aside from spheres and boxes. For this reason, we completely rewrote many parts of the engine, changing many features and adding new ones.

Three additional resources that we utilized to learn about the theory of computer graphics, which greatly aided our implementation and were frequently referenced throughout this text, are

1. *Fundamentals of Computer Graphics* [Shirley et al., 2021].
2. *Real-Time Rendering* [Akenine-Moller et al., 2019].
3. *An Introduction to Ray Tracing* [Glassner, 1989].

The following list summarizes our contributions, with links to the respective features in our GitHub project repository:

1. **New ray/sphere intersection test.**
   - Shirley's code implements the *algebraic solution*. Our code adds the *geometric solution*.
2. **New ray/AABB intersection tests.**
   - Shirley's code implements the *slab method*. In our code, we implement other algorithms:
     (a) Branchless method.
     (b) Optimized slab method.
     (c) A method that we wrote (see §5.6.2).
3. **New `Triangle` class.**
   - Shirley's code doesn't have a triangle class; however, to render meshes, we needed to implement one. It has the following features:

   (a) We implemented three ray/triangle intersection algorithms:
      i. *Möller–Trumbore algorithm.*
      ii. *Snyder & Barr algorithm.*

   (b) We implemented a method to sample points on the triangle. This was needed to render less-noisy triangles when light is used.

   (c) We implemented a method to calculate the PDF value of sampling a random direction on the triangle.

   (d) We implemented a method to allow constructing an AABB for a triangle, and thus render it faster using the BVH class.

   (e) We implemented functions to read .obj files and facilitate loading meshes in order to render them. The functions allow the programmer to scale, rotate, and translate the loaded mesh.
4. **Meshes.**
   - Shirley's code has no meshes. Using the `Triangle` class that we implemented, we were able to read .obj files and build meshes using triangles in the process. This allowed us to render all the meshes that were shown in the research.
5. **New class to support 2D vectors**.
   - The `Vec2D` class helped us in implementing the `Triangle` class.
6. **New materials.**
   - Shirley's code has three material types: Lambertian (i.e., perfect diffuse), metal, and dielectric. Our engine expands these materials and implements new ones:
     (a) A `Phong` class for Phong material.
     (b) A diffuse material based on Disney diffuse model [Burley and Studios, 2012] [Boksansky, 2021] (`Disney_Diffuse`) (see Figure A.3).

(c) Another diffuse material based on uniformly sampling the hemisphere (`Uniform_Hemispherical_Diffuse`) (see Figure A.4).

7. **Importance Sampling**
   - We extended Shirley's code to enable importance sampling for:
     (a) specular materials (`Specular_PDF`), which we use for Phong materials (see Figure A.5), and
     (b) uniformly sampling the hemisphere (`Uniform_Hemispherical_PDF`), which we use for a variant of diffuse materials (discussed above).
8. **New Transformations.**
   - We implemented new classes to allow for rotations along the X and Z axes (Rotate_X and Rotate_Z).
9. **BVH Implementation.**
   - Shirley's code uses `std::sort` to group the objects by the minimum coordinates of their bounding boxes along the partition axis, which is chosen at random. This is inefficient, so we used different heuristics to construct the BVH.
     (a) We experimented using other sorting functions (e.g., `std::nth_element()`), and found that they can accelerate rendering (§6.4).
     (b) We extended the BVH implementation to allow sorting objects by the maximum coordinates and centroids of their bounding boxes, which proved to yield an appreciable speedup (§6.4).
     (c) Instead of choosing the split axis at random, we allowed for choosing axis rotationally (first X, then Y, then Z). This has also shown to accelerate rendering (§6.4).
     (d) Finally, we optimized the implementation of the BVH by changing the way the lists of objects are passed through each recursive calls, which significantly decreased the build time of the hierarchy and increases rendering speeds (§6.4).
     (e) We used OpenMP to enable constructing the BVH in parallel, thus saving a lot of time.
       – These new changes can be found in the following classes: `BVH_Max_Coordinate`, `BVH_Centroid_Coordinate`, `BVH_Fast`, `BVH_Parallel`, and `Primitive`.
10. **CPU Multithreading.**
    - We used OpenMP to enable CPU parallelism. We previously discussed how we implemented several strategies (§5.8) and showed that they accelerated rendering (§6.5). These strategies can be found in the `Parallel_Rendering_Functions` file in our code.
11. **GPU Multithreading.**
    - We wrote a version of our code that supports GPU parallelism using CUDA. We previously discussed our implementation and some of the challenges we faced (§5.9), and we also showed that we were able to accelerate rendering speeds using the GPU (§6.6) (GPU Version).
12. **Textures.**
    - We implemented different textures (see this folder) than the ones presented in Shirley's code (see Figure A.7 for an example).

Graphics studios tend to be surprisingly open about their research, which is indicated by the many research papers that we referenced throughout this text, many of which were publications made by the likes of Pixar, Disney, and NVIDIA. Nonetheless, a huge part of the implementation remains propriety, and in many instances it is hard to know what the exact method an engine implements to solve a specific problem (say, what the multi-threading model an engine uses)[2]. For this reason, it was hard to determine whether our approach to solve some problems (like implementing multithreading) is used by other engines. Nonetheless,

[2]For instance, to gain access to Unity's engine source code, one would need to contact the enterprise (link).

as we discussed in §6.5, we were able to compare our multithreading speedup results with other research papers.

7.3 The Future

The purpose of our research was to take a naïve ray tracer and turn it into a fast path tracer. This was done through the implementation of importance sampling, fast intersection tests, efficient BVH construction, and CPU and GPU multithreading. We have basically equipped ourselves with some sophisticated tools to explore other research areas in the field of computer graphics.

As a starting point, we can explore other Monte Carlo-based rendering methods, and an excellent entry point for this exploration is the well-regarded thesis of Veach [Veach, 1998]. Almost every established production graphics studio incorporates concepts from it, particularly the utilization of *bidirectional light transport* and the application of *Metropolis sampling* to rendering. In the latter, our sampling space will change from solid angles to a random walk through path space. Building on this foundation, we can progress to implementing *multiple importance sampling*, which can eliminate large variance (i.e., noise) by drawing samples from multiple sampling distributions. This increases the efficiency of our Monte Carlo renderer. We already have different sampling distributions, so all we need to do is instrument our code to draw samples from each and somehow combine them correctly.

Subsequently, we can extend our BVH implementation on the CPU by incorporating a surface area heuristic (SAH), which usually yields superior tree qualities, as it can lower the number of nodes traversed and the number of intersection tests. We already have a way of sorting objects in the scene based on their centroids. What remains is adding a mechanism to traverse each potential split and compute their resulting surface areas. We keep repeating this process and minimizing the cost of traversals for ray intersection tests. This can further accelerate our engine's rendering speeds.

For the GPU implementation, we can implement a more sophisticated hierarchy: The LBVH, which is easily parallelizable. We discussed it in §2.6.3. Moreover, instead of transferring the meshes from host memory to device memory, as we currently do, we can explore storing triangle data in the GPU's texture memory. By processing data in chunks, the CPU can read the current chunk while the GPU simultaneously processes the previous chunk, constructing the mesh in parallel. Attention must be paid to potential bottlenecks caused by caching, and these can be mitigated by invoking the GPU kernel from a sepparate CPU thread [Possemiers and Lee, 2015].

We can also explore the applications of deep learning in rendering. In terms of accelerating rendering, we can, for example, use deep learning models for denoising. There are many publications from NVIDIA and Disney that discuss this interesting intersection of deep learning and computer graphics. We already have a functional engine at our disposal; all we need is to use it to render images and create a custom dataset that we can train the desired deep learning model on.

Appendix A

A.1 Project Planning

Project planning is an essential part of our research for several reasons, some of which we list below:

1. **Time Management:** We have several tasks to do as part of our research, such as literature review, thesis writing, and engine implementation. Planning helps us allocate time effectively and ensures sufficient progress.
2. **Task Sequencing:** Breaking down our research into smaller tasks and sequencing them helps us understand in which order to execute them to ensure a smooth workflow.
3. **Adaptability:** Writing a good project plan helps us mitigate unforeseen circumstances.

The work on our research spans across two semesters (September semester, 2023 and January semester, 2024). Throughout this semester, we have maintained a weekly log of all the tasks completed for that particular week. This weekly log is maintained in our GitHub repository (see Figure A.1).

A.1.1 September Semester - 2023

During the September semester, we focused on writing the Research Report and planning our project. We surveyed the literature to gather information that may be relevant to our topic. The aim was to collect as many relevant resources as possible, eliminate the ones that we will not consider, and focus on papers that are highly relevant to our research topic. To set the stage for our research, a review of background concepts, including various rendering algorithms and shading models, was conducted. Furthermore, we started implementing some parts of our rendering engine. Finally, we focused on working on other chapters, including project planning and evaluation strategies.

The project plan of this semester is summarized in the Gantt charts below:

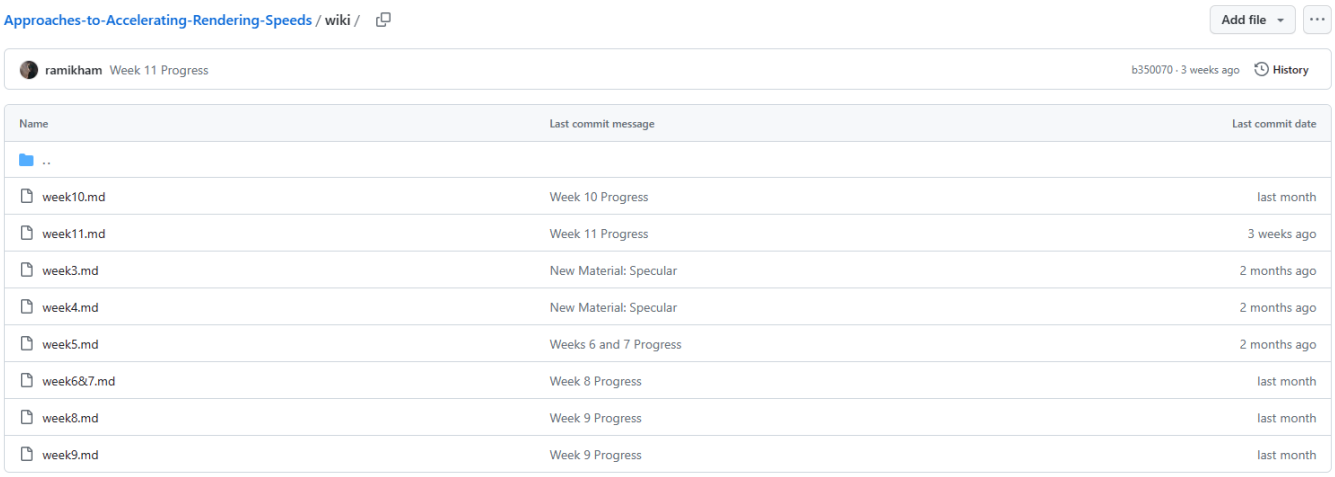
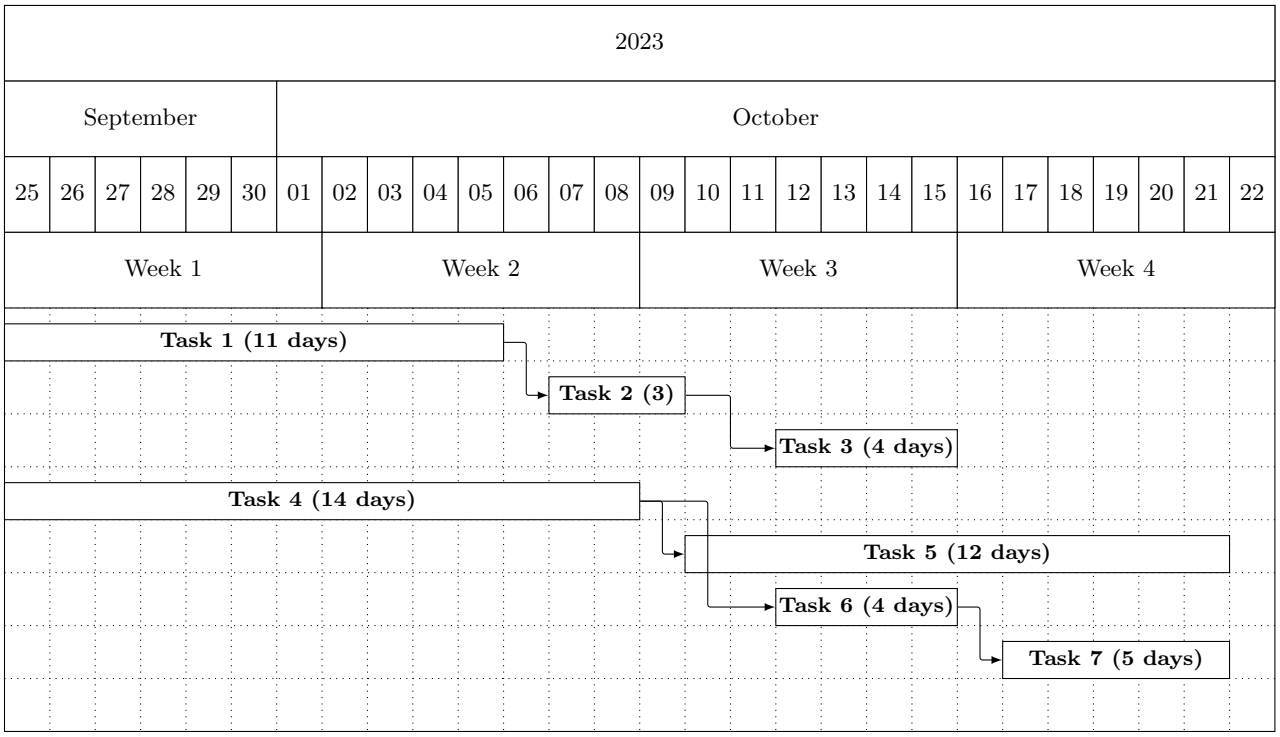


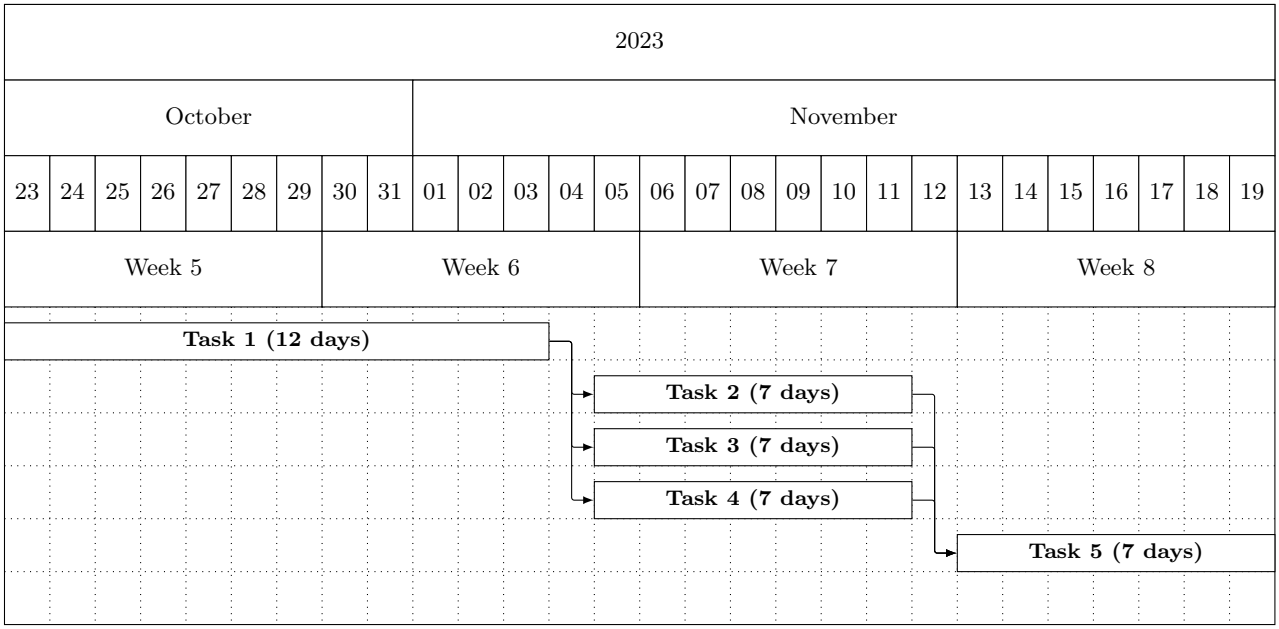
Figure A.1: The weekly log maintained in our GitHub repository.



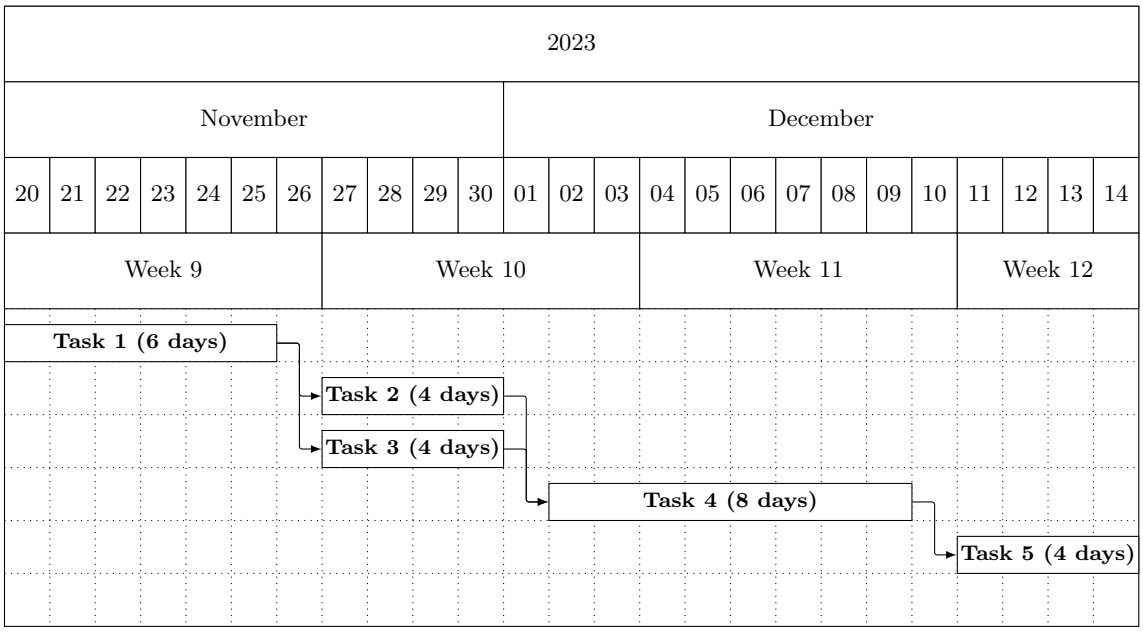
- **Task 1:** Start the implementation of the rendering engine (features include simple shading, a camera class, ambient light, spheres, and ray-tracing to render images).
- **Task 2:** Update the *Appendix* to discuss the math behind the implementation of the rendering engine.
- **Task 3:** Review the code and refactor if necessary (i.e., investigate if some functions can be simplified and/or if adding/deleting some classes can be useful).
- **Task 4:** Conduct a literature review with a focus on background topics and acceleration data

structures.

- **Task 5:** Write the *Literature Review* chapter.
- **Task 6:** Write the first draft of the *Abstract* motivated by the literature review (**Task 4**).
- **Task 7:** Write the *Introduction* chapter. This includes formulating the aims and the objectives of our research. The *Introduction* should be connected to the *Abstract* (**Task 6**), but it should be more detailed.



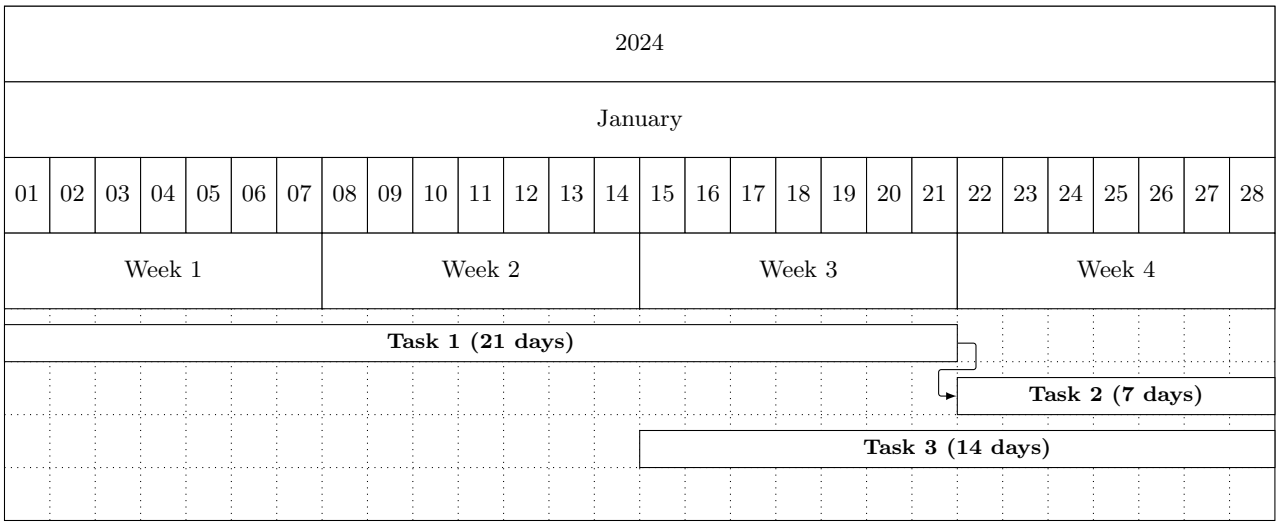
- **Task 1:** Conduct a literature review, focusing on CPU/GPU accelerated rendering and exploring deep learning techniques related to rendering.
- **Task 2:** Write the *Literature Review* chapter.
- **Task 3:** Write the *Methodology, Evaluation & Requirements* chapter, focusing on specifying the functional and non-functional requirements of our project.
- **Task 4:** Write the *Project Management* chapter, focusing on specifying the tools used for managing the project and discuss any social and ethical issues.
- **Task 5:** Finish the first draft of the *Research Report*.



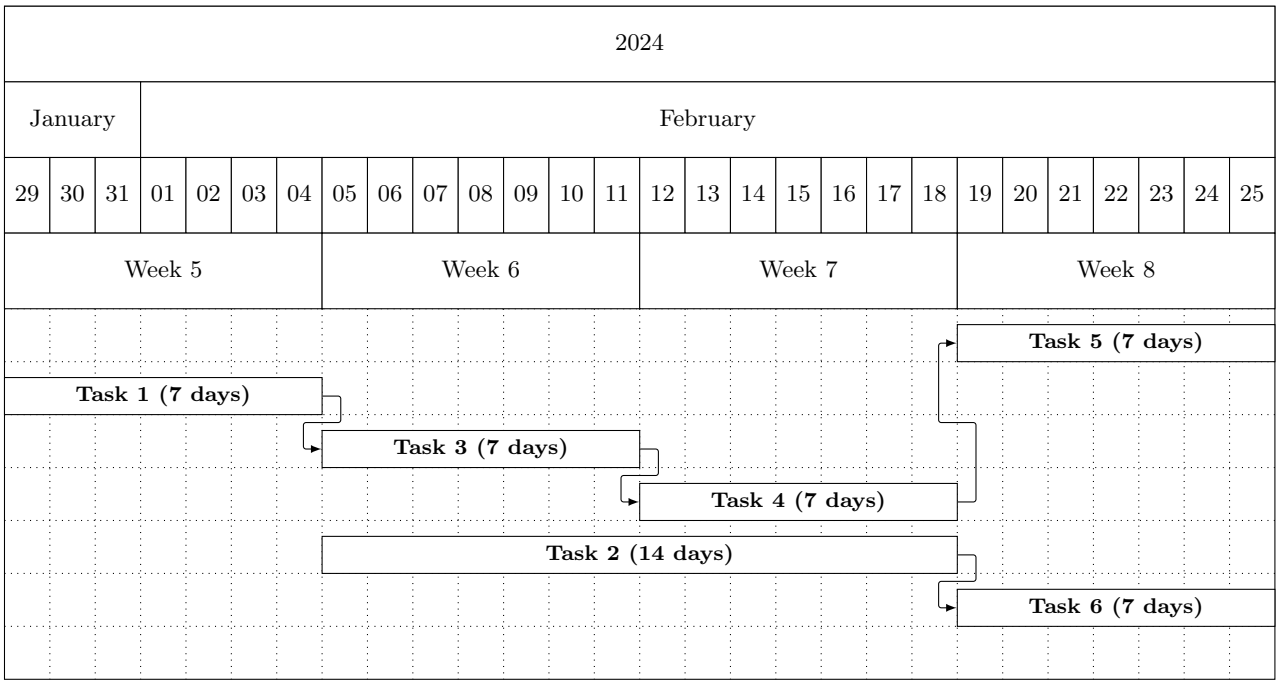
- **Task 1:** Write the second draft of the Research Report, focusing on completing the *Abstract* and the *Introduction* chapter.
- **Task 2:** Revise the *Methodology, Evaluation & Requirements* chapter. This includes specifying the metrics and writing an evaluation strategy.
- **Task 3:** Revise the *Project Management* chapter. Add a detailed project plan and draw Gantt charts to illustrate the timeline.
- **Task 4:** Write the third draft of the Research Report, including a short *Conclusion* chapter.
- **Task 5:** Prepare the final version of the research plan for submission. It should be ready to be submitted on 14^{th} of December.

A.1.2 January Semester - 2024

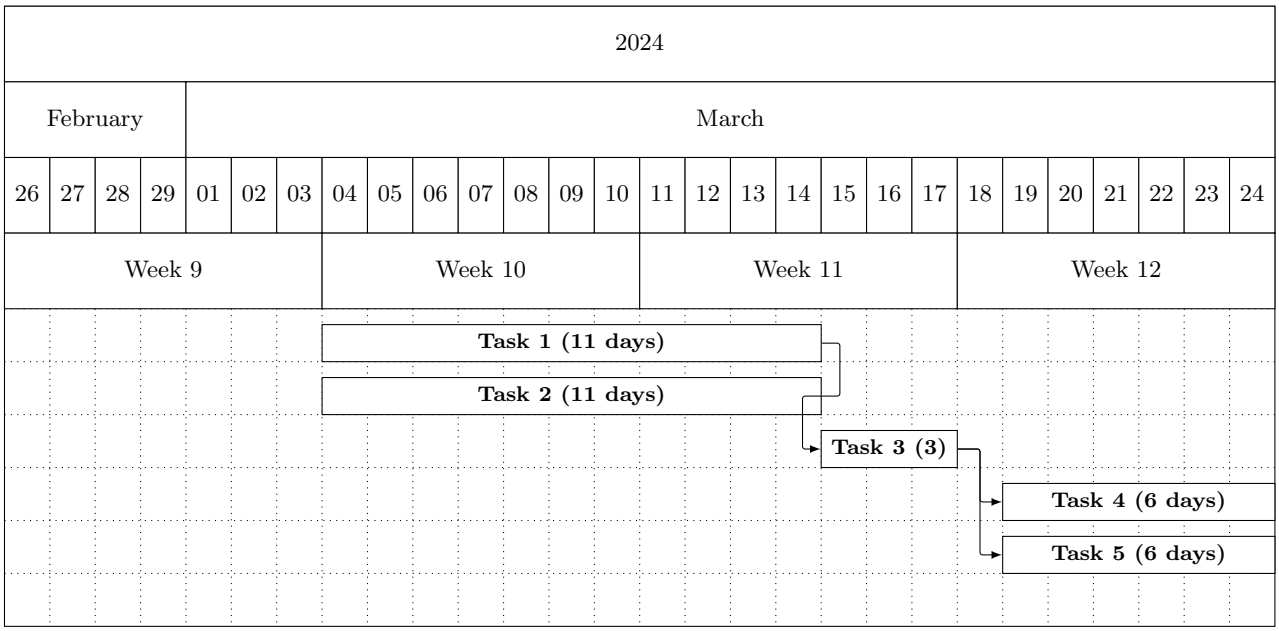
Below are the Gantt charts that detail our plan for the next semester.



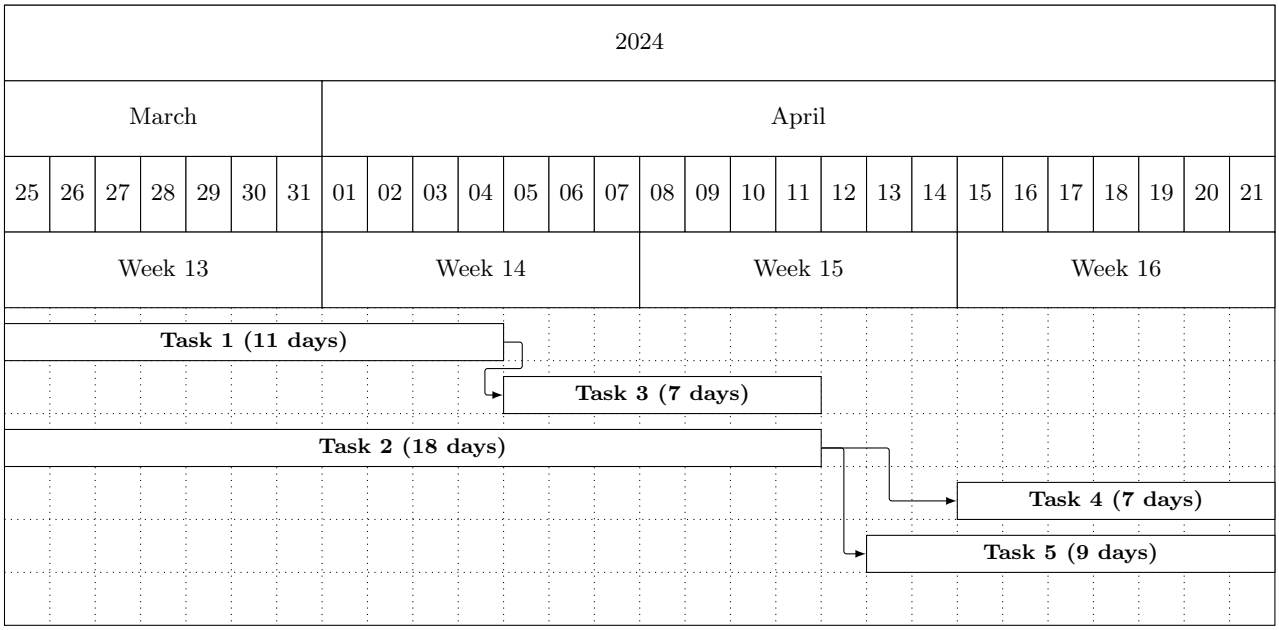
- **Task 1:** Continue the implementation of the rendering engine. Work in the first semester focused only on finishing a minimal viable produce prototype. We will add the following features:
  - More materials (e.g. dielectrics).
  - A BVH with AABB as the bounding volumes.
  - Textures.
  - New primitives (boxes, triangles, and meshes).
  - Explicit lights (in the September semester, we only implement ambient (i.e., environment) light).
  - Implement path tracing. The code written in the September semester only support ray tracing. This means we need to implement importance sampling.
- **Task 2:** Start implementing multithreaded rendering. The aim is to go beyond simply parallelizing the render loop.
- **Task 3:** Start writing the dissertation. This iteration entails carrying over the work we did in the Research Plan and put it in the dissertation in the appropriate chapters.



- **Task 1:** Start benchmarking the performance of the rendering engine, including calculating the appropriate metric measures that we previously discussed (see §3.3). Here, we should be able to understand how much did AABB and OpenMP help in accelerating the engine. Additionally, It is normal that during experimentation, some errors will reveal themselves in the code. This is a chance to debug and fix the errors appropriately (**Task 3** will be allocated to debugging). Additionally, it is worthwhile to note any improvements that can be made to optimize the code (**Task 4** will be dedicated to this goal).
- **Task 2:** Start CUDA integration. Incorporating CUDA is a notorious task, so giving it enough time is necessary.
- **Task 3:** Debug any implementation errors that show during the experimentation carried out in **Task 1** (if any bugs show).
- **Task 4:** Optimize the code (if any optimization is viable).
  - **Note:** Ideally, work on **Task 3** and **Task 4** should not take two weeks; therefore, we will dedicate any remaining free days to work on **Task 2**, which has high priority for our research.
- **Task 5:** Draft the findings obtained from the experiments. Present them in charts and graphs so that they will be included in the dissertation.
- **Task 6:** Benchmark the results obtained from integrating CUDA.



- **Week 9:** This week will have no major task; it will be a catch-up period to allow for completion of any outstanding tasks from previous weeks.
- **Task 1:** Explore other techniques to accelerate our rendering engine. We have already examined some in our literature review, so this task presents an opportunity to implement, modify, or add to the techniques we have considered, or if not feasible, to explore alternative approaches.
- **Task 2:** Dissertation writing.
- **Task 3:** Draft methods to evaluate our newly implemented techniques.
- **Task 4:** Run experiments and record the results.
- **Task 5:** Write and interpret the results of the experiments in the dissertation.



- **Task 1:** Investigate further acceleration techniques that can be implemented (e.g.: can we parallelize the code more efficiently? Can we change the structure of the code to reduce its running time? etc.)
- **Task 2:** Prepare the first draft of the dissertation.
- **Task 3:** Include any results observed from **Task 1** in the dissertation.
- **Task 4:** Prepare the poster.
- **Task 5:** Write the second draft of the dissertation.

- **Task 1:** Write the third and final draft of the dissertation.
- **Task 2:** Prepare the dissertation for submittion.

A.2 Risk Analysis

Refer to table A.1 for a detailed risk assessment.

A.3 Implementing the Rendering Engine

In the following subsections, we will dive deeper into few aspects of our implementation, particularly focusing on the mathematical derivation of some of the features that we implemented.

A.3.1 Importance Sampling: Continued

This section is a continuation of the description of the implementation of generating random numbers in our rendering engine that we discussed in §5.5.1. We will expand the explanation here by including more details.

Risk	**Likelihood of Occurrence**	**Impact**	**Mitigation**
Difficulty in integrating CUDA	Medium	High	Use a different computer or consult with the supervisor to find alternative solutions.
Difficulties in implementing some of the algorithms	Medium	High	Consider using different algorithms or consulting with the supervisor to explore alternative approaches.
Difficulties in meeting deadlines	Medium	High	Prioritize tasks based on importance and set realistic and achievable goals. Remain flexible and open to adjusting goals if necessary.
Data loss	Medium	High	Regularly back up data using version control tools as discussed in section 4.1.
Limited scalability of the engine's code	Medium	High	Ensure the code-documentation is used throughout the development process, and include a how-to-use guide if possible.
Limited cross-platform performance	Medium	High	The rendering engine may not perform consistently across different hardware configurations. This is dangerous because the results will not be reproducible (or verifiable). The mitigation would be to regularly test the rendering engine on various configurations and address in the dissertation any platform-specific compatibility issues if they arise.
Bugs in the rendering engine	Medium	Medium	Utilize software testing practices, such as writing unit tests, to identify and address bugs.
Inaccurate lighting and shadow calculations	Medium	Low	Debug the code or use simpler scenes and address the limitations in the dissertation.
Software limitations	Low	Low	Render scenes with lower complexity and address the limitations of the rendering engine in the dissertation.
Hardware limitations	Low	Low	Render scenes with lower complexity and address the limitations of the hardware used in the dissertation.
Student Illness	Low	Low	Focus on recovery and utilize the extra week allocated in the project plan if needed.

Table A.1: Risk Mitigation Plan

A.3.1.1 Sampling the Hemisphere

Let us see how to generate directions on the unit hemisphere[3]. Consider a point ω on the unit hemisphere Ω. By the definition of a valid PDF, we have: $\int\int_{\Omega} p(\omega)\, dA = 1$. Since we are generating points uniformly on a unit hemisphere, we have: $\int\int_{\Omega} dA = 2\pi$. This means that we can obtain a uniform distribution over the unit hemisphere if we respect the following condition:

$$p(\omega)\, dA = \frac{1}{2\pi}\, dA = p(\theta, \phi)\, d\theta\, d\phi \tag{A.1}$$

By noting that w can be parameterized by (θ,ϕ), and by the definition of the solid angle: $dA = \sin(\theta)\, d\theta\, d\phi$ [Weisstein, 2002b], we can rewrite A.1 by noticing that: $\frac{1}{2\pi}\, dA = p(\theta, \phi)\, d\phi\, d\theta \Rightarrow \frac{1}{2\pi} \sin(\theta)\, d\theta\, d\phi = p(\theta, \phi)\, d\phi\, d\theta$, and canceling the d's we get:

$$p(\theta, \phi) = \frac{1}{2\pi} \sin(\theta) \tag{A.2}$$

Really, the whole point of (A.2) is that it allows us to invoke the definition of the marginal PDF to get:

$$\begin{aligned} p(\theta) &= \int_0^{2\pi} p(\phi, \theta)\, d\theta \\ p(\phi) &= \int_0^{\pi} p(\phi|\theta)\, d\phi \end{aligned} \tag{A.3}$$

Two observations can be made here. First, $p(\phi)$ is uniform, because the unit hemisphere is symmetrical along the Z-axis. Second, all we need to do to generate random directions (on a sphere or weighted) with a specific PDF p(w) is to parameterize them with θ and ϕ and plug it in p(θ) (p(ϕ) will always remain the same: uniform), thus A.3 becomes:

$$\begin{aligned} p(\phi|\theta) &= \frac{1}{2\pi} \\ p(\theta) &= \int_0^{\pi} p(\phi, \theta)\, d\phi \quad = \int_0^{2\pi} \frac{1}{2\pi} sin(\theta)\ d\phi = \sin(\theta) \end{aligned} \tag{A.4}$$

Now, using inverse transform sampling, we can sample each PDF. Let us compute the CDFs, which represent the two random numbers that we need, r_1 and r_2:

$$r_1 = P(\phi|\theta) = \int_0^{\phi} \frac{1}{2\pi}\, d\phi' = \frac{\phi}{2\pi} \tag{A.5}$$

$$r_2 = P(\theta) = \int_0^{\theta} \sin(\theta')\, d\theta' = 1 - \cos(\theta) \tag{A.6}$$

[3]We use a similar method to sampling the sphere, which is explained in many references [Weisstein, 2002b] [Simon, 2015].

Solving for ϕ in A.5 and for $cos(\theta)$ in (A.6)[4], we get:

$$\phi = 2\pi \cdot r_1 \tag{A.7}$$

$$\cos(\theta) = 1 - r_2 \tag{A.8}$$

Let us convert these parameters back to Cartesian. We also note that we can simplify calculations by using the Pythagorean identity: $\sin^2(\theta) + \cos^2(\theta) = 1 \Rightarrow \sin^2(\theta) = 1 - \cos^2(\theta) \Rightarrow$ (plugging Equation A.8) $\Rightarrow \sin^2(\theta) = 1 - (1 - r_2)^2 \Rightarrow$ because r_2 is uniformly distributed $\in$ [0,1], we can safely reflect it $\Rightarrow \sin^2(\theta) = 1 - r_2^2 \Rightarrow \sin(\theta) = \sqrt{1 - r_2^2}$. The Cartesian coordinates are thus:

$$\begin{aligned} x &= \sin(\theta)\cos(\phi) = \sqrt{1-(r_2)^2}\cos(2\pi r_1) \\ y &= \sin(\theta)\sin(\phi) = \sqrt{1-(r_2)^2}\sin(2\pi r_1) \\ z &= \cos(\theta) = 1 - r_2 \end{aligned} \tag{A.9}$$

A.3.1.2 Cosine-weighted Hemisphere Sampling

Similar to what we did in §A.3.1.1, we can sample the hemisphere with a cosine weighted PDF. As stated in §5.5.2, the PDF is:

$$p(\omega) = \frac{\cos(\theta)}{\pi}$$

Our transformation will be:

$$p(\theta, \phi) = \frac{\cos(\theta)}{\pi} \cdot \sin(\theta)$$

The marginal density functions can be obtained as such:

$$\begin{aligned} p(\theta) = \int_0^{2\pi} p(\phi, \theta)\, d\phi \quad &= \int_0^{2\pi} \frac{\cos(\theta)}{\pi} \cdot \sin(\theta)\, d\phi \quad &= 2\cos(\theta)\sin(\theta) \\ p(\phi) \Rightarrow p(\phi|\theta) = \frac{p(\theta,\phi)}{p(\theta)} \quad &= \frac{\frac{\cos(\theta)\sin(\theta)}{\pi}}{2\cos(\theta)\sin(\theta)} \quad &= \frac{1}{2\pi} \end{aligned} \tag{A.8}$$

Now, we can get the two random numbers by setting them equal to the CDFs:

$$r_1 = P(\phi|\theta) = \int_0^{\phi} \frac{1}{2\pi}\, d\phi' = \frac{\phi}{2\pi}$$

$$r_2 = P(\theta) = \int_0^{\theta} 2\cos(\theta')\sin(\theta')\, d\theta' = \sin^2(\theta) = 1 - \cos^2(\theta)$$

Now, using the following observation:

$$\phi = 2\pi \cdot r_1$$

[4]We don't need to solve for θ, which is good because `acos()` is slow.

$$\cos(\theta) = \sqrt{1 - r_2}$$

We can solve for the Cartesian coordinates:

$$\begin{aligned} x &= \cos(\phi)\sin(\theta) = \cos(2\pi r_1)\sqrt{r_2} \\ y &= \sin(\phi)\sin(\theta) = \sin(2\pi r_1)\sqrt{r_2} \\ z &= \cos(\theta) = \sqrt{1 - r_2} \end{aligned}$$

A.3.1.3 Power Cosine-weighted Hemisphere Sampling

We can generalize §A.3.1.2 to use importance sampling when rendering Phong materials. The PDF[5] is:

$$p(\omega) = \frac{(n+1)cos^n\theta}{2\pi}$$

Our transformation will be:

$$p(\theta, \phi) = \frac{(n+1)cos^n(\theta)}{2\pi} \cdot \sin(\theta)$$

The marginal density functions can be obtained as such:

$$\begin{aligned} p(\theta) = \int_0^{2\pi} p(\phi, \theta)\, d\phi \quad &= \int_0^{2\pi} \frac{(n+1)\cos^n(\theta)}{2\pi} \cdot \sin(\theta)\, d\phi \quad &= (n+1)\cos^n(\theta)\sin(\theta) \\ p(\phi) \Rightarrow p(\phi|\theta) = \frac{p(\theta,\phi)}{p(\theta)} \quad &= \frac{\frac{(n+1)\cos^n(\theta)\sin(\theta)}{2\pi}}{(n+1)\cos^n(\theta)\sin(\theta)} \quad &= \frac{1}{2\pi} \end{aligned} \tag{A.2}$$

Now, we can get the two random numbers by setting them equal to the CDFs:

$$r_1 = P(\phi|\theta) = \int_0^{\phi} \frac{1}{2\pi}\, d\phi' = \frac{\phi}{2\pi}$$

$$r_2 = P(\theta) = \int_0^{\theta} (n+1)\cos^n(\theta')\sin(\theta')\, d\theta' = 1 - \cos^{n+1}(\theta)$$

To get r_2 above, we used *u-substitution*. Now, using the following observation:

$$\phi = 2\pi \cdot r_1$$

$$\cos(\theta) = r_2^{\frac{1}{n+1}}$$

[5] Lawrence [Model,] provides a details of why the PDF has this form and why it can be used in Phong importance sampling. He also shows the final Cartesian coordinates. However, the details of the derivation are assumed and not shown, so we included them here.

Where we derived $\cos(\theta)$ as such:

$$\begin{aligned} r_2 = 1 - \cos^{n+1}(\theta) \Rightarrow \cos^{n+1}(\theta) = 1 - r_2 \\ \Rightarrow \cos(\theta) = \sqrt[n+1]{\cos^{n+1}(\theta)} = \sqrt[n+1]{1 - r_2} \\ \Rightarrow \cos(\theta) = r_2^{\frac{1}{n+1}} \end{aligned}$$

In the last step, we used the symmetry of a uniform random variable.

We can solve for the Cartesian coordinates:

$$\begin{aligned} x &= \cos(\phi)\sin(\theta) = \cos(2\pi r_1)\sqrt{1 - r_2^{(\frac{2}{n+1})}} \\ y &= \sin(\phi)\sin(\theta) = \sin(2\pi r_1)\sqrt{1 - r_2^{(\frac{2}{n+1})}} \\ z &= \cos(\theta) = r_2^{n+1} \end{aligned}$$

A.3.1.4 Sampling the Sphere

The PDF of sampling the unit sphere is:

$$p(\omega) = \frac{1}{4\pi}$$

Our transformation will be:

$$p(\theta, \phi) = \frac{1}{4\pi} \cdot \sin(\theta)$$

The marginal density functions can be obtained as such:

$$\begin{aligned} p(\theta) = \int_0^{2\pi} p(\phi, \theta)\, d\phi \quad &= \int_0^{2\pi} \frac{1}{4\pi} \cdot \sin(\theta)\, d\phi \quad &= \frac{\sin(\theta)}{2} \\ p(\phi) \Rightarrow p(\phi|\theta) = \frac{p(\theta, \phi)}{p(\theta)} \quad &= \frac{\frac{1}{4\pi} \cdot \sin(\theta)}{\frac{\sin(\theta)}{2}} \quad &= \frac{1}{2\pi} \end{aligned} \tag{A.-5}$$

Now, we can get the two random numbers by setting them equal to the CDFs:

$$r_1 = P(\phi|\theta) = \int_0^{\phi} \frac{1}{2\pi}\, d\phi' = \frac{\phi}{2\pi}$$

$$r_2 = P(\theta) = \int_0^{\theta} \frac{\sin(\theta')}{2}\, d\theta' = \sin^2(\theta) = \frac{1}{2}(1 - cos(\theta))$$

Now, using the following observation:

$$\phi = 2\pi \cdot r_1$$

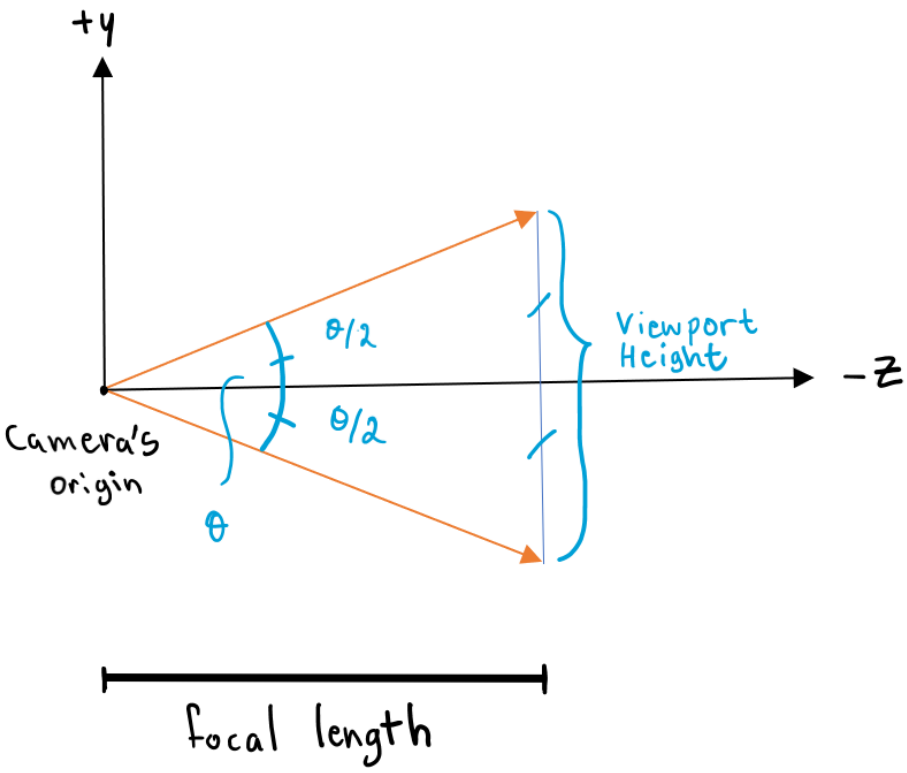


Figure A.2: Camera's geometry

$$\cos(\theta) = 1 - 2r_2$$

We can solve for the Cartesian coordinates:

$$\begin{aligned} x &= \cos(\phi)\sin(\theta) = \cos(2\pi r_1)2\sqrt{r_2(1-r_2)} \\ y &= \sin(\phi)\sin(\theta) = \sin(2\pi r_1)2\sqrt{r_2(1-r_2)} \\ z &= \cos(\theta) = \sqrt{1-r_2} \end{aligned}$$

A.3.2 The Camera Model

The angular range of the observable space seen from the position of an observer is called the *field-of-view* (FOV).

For a camera to be positioned in a 3D space, we need to to be capable of controlling its FOV. A FOV, by definition, has a horizontal extent and a vertical extent. The part of the FOV we wish to control is the *vertical field-of-view* (VFOV).

The geometry of the camera can be seen in Figure A.2. We can use it to derive the important quantity of the camera's model.

By the definition of the tangent of an angle, we can write:

$$\tan\left(\frac{\theta}{2}\right) = \left(\frac{half\ height}{focal\ length}\right) \tag{A.-9}$$

Using (A.1), we can write:

$$half\ height = focal\ length * \tan\left(\frac{\theta}{2}\right) \tag{A.-9}$$

This, alongside the calculation of the viewport height and viewport width, produces a camera that has an adjustable angle: we can increase the VFOV to get a wide-angle view. To make the camera's angle rotatable and the camera's origin arbitrarily positionable in space, thus adding more flexibility to the model, we need to use the idea of an *orthonormal basis*. An orthonormal basis is a basis whose vectors are *orthonormal*: they are all unit vectors and orthogonal to each other. If we can calculate the following:

- a unit vector originating from the camera's origin pointing toward the **opposite** direction of the point in the scene the camera is looking at (opposite to maintain the convention of the right-hand rule),
- a unit vector originating from the camera's origin pointing toward the camera's rightward direction, and
- a unit vector originating from the camera's origin pointing to the camera's upward direction,

then we have a coordinate system, and we refer to it as the *camera's coordinate system.* A view rotation is simply the rotation of the camera around its coordinate system.

The first unit vector can be provided explicitly, and the two other unit vectors can be calculated by the use of cross products [Weisstein, 2002a].

A.3.3 Intersection Tests

A.3.3.1 Ray/Sphere Intersection

Given a ray defined as:

$$P(t) = O + tD \tag{A.-9}$$

and a sphere centered at (X_0, Y_0, C_0) defined as:

$$(x - Cx)^2 + (y - Cy)^2 + (z - Zz)^2 = r^2 \tag{A.-9}$$

which can be rewritten in vector form as:

$$(\vec{P} - \vec{C}) \cdot (\vec{P} - \vec{C}) = r^2 \tag{A.-9}$$

by noting that $\vec{P} =< x, y, z >$ and $\vec{C} =< Cx, Cy, Cz >$, then our task is to find t for which *P(t)* satisfies the sphere's equation. To do this we write:

$$(\vec{P}(t) - \vec{C}) \cdot (\vec{P}(t) - \vec{C}) = r^2 \tag{A.-9}$$

and solve for t.

Simple algebraic substitutions and simplifications lead us to the following form:

$$t^2\vec{b}\cdot\vec{b} + 2t\vec{b}\cdot(\vec{A}-\vec{C}) + (\vec{A}-\vec{C})\cdot(\vec{A}-\vec{C}) - r^2 = 0 \tag{A.-9}$$

Noting that t is the only unknown variable, we can write the following:

$$\begin{aligned} A &= \mathbf{b}\cdot\mathbf{b} \\ B &= 2\mathbf{b}\cdot(\mathbf{A}-\mathbf{C}) \\ C &= (\mathbf{A}-\mathbf{C})\cdot(\mathbf{A}-\mathbf{C}) - r^2 \end{aligned}$$

These are ready to be plugged in the quadratic formula, and easily solved in code:

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{A.-9}$$

More information about the ray/sphere intersection subroutine can be found in *An Introduction to Ray Tracing* [Glassner, 1989].

A.3.3.2 Ray/Triangle Intersection

Snyder & Barr and *Möller-Trumbore* intersection algorithms are both used in rendering to determine if a ray intersects with a triangle. Both algorithms leverage barycentric coordinates to calculate the intersection point between the ray and the triangle, and both algorithms only require a triangle to be represented via its three coordinates.

The essence of *Snyder & Barr* is that using the ray's equation and the three vertices, we can write a linear system of equations in which we equate the ray's equation with the parametric representation of the triangle (which uses its three coordinates and represents them via barycentric coordinates). This amounts to having three equations and three unknowns (t, β, and γ). To solve for these unknowns, we simply write the three equations as a standard linear system and solve it via *Cramer's rule*.

A.3.3.3 Ray/Box Intersection

For brevity, we shall discuss the 2D case here. The 3D case is just a natural extension. Again, we are given a ray defined as:

$$P(t) = A + tb \tag{A.-9}$$

and a 2D bounding box defined by four lines:

$$\begin{aligned} x &= x_{\min} \\ x &= x_{\max} \\ y &= y_{\min} \\ y &= y_{\max} \end{aligned}$$

The region bounded by these lines contains a primitive(s). Hence, our goal is to test if the ray passes through this region in space.

Let us first consider the line $x = x_{\text{min}}$. We want to find the t at which the ray and the line intersect. Call this specific time t_{xmin}. We can think of the ray as:

$$\begin{bmatrix} x(y) \\ y(t) \end{bmatrix} = \begin{bmatrix} A_x \\ A_y \end{bmatrix} + t \begin{bmatrix} b_x \\ b_y \end{bmatrix} \tag{A.-9}$$

At t_{xmin}, we have two equations:

$$\begin{aligned} x(t_{\text{xmin}}) &= A_x + t_{\text{xmin}} b_x \\ y(t_{\text{xmin}}) &= A_y + t_{\text{xmin}} b_y \end{aligned} \tag{A.-9}$$

Thus, if we plug A.3.3.3 into A.3.3.3, we get:

$$\begin{bmatrix} x(t_{\text{xmin}}) \\ y(t_{\text{xmin}}) \end{bmatrix} = \begin{bmatrix} A_x \\ A_y \end{bmatrix} + t_{\text{xmin}} \begin{bmatrix} b_x \\ b_y \end{bmatrix} \tag{A.-9}$$

Since we need to solve for one variable, $x = x_{\text{min}}$, we only need to use one of the two equations. We will go with the first row, because we are given the line $x = x_{\text{min}}$. Thus we have:

$$t_{\text{xmin}} = \frac{x_{\text{min}} - A_x}{b_x} \tag{A.-9}$$

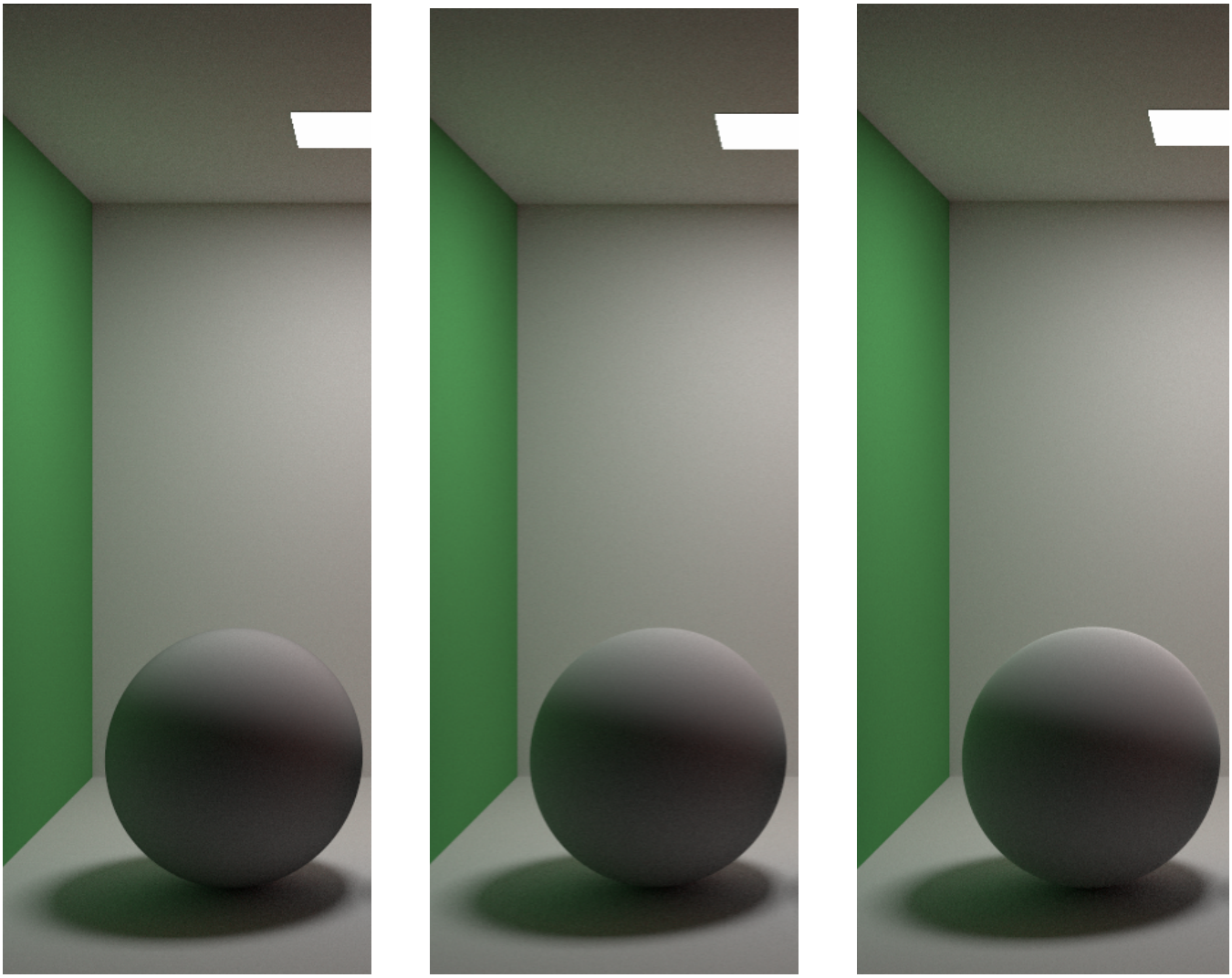
Similar computations can be made to derive t_{ymin}, t_{xmax}, and t_{ymax}. We should also account for the potential division by zero in our code. We can mitigate this by using the *min* and *max* operations in C++. Any other optimizations that we mentioned in §A.3.3.3 build upon this algorithm.

A.4 3D Models Used

The 3D models used in this project were taken from the following websites:

- https://graphics.stanford.edu/data/3Dscanrep/
- https://graphics.cs.utah.edu/teapot/
- https://www.graphics.cornell.edu/online/box/
- https://casual-effects.com/data/index.html

A.5 Gallery



(a) *Roughness = 0.0* (b) *Roughness = 0.5* (c) *Roughness = 1.0*

Figure A.3: Spheres rendered using the Disney diffuse model with different roughness parameters. Notice the reflection of light at the top of each sphere.

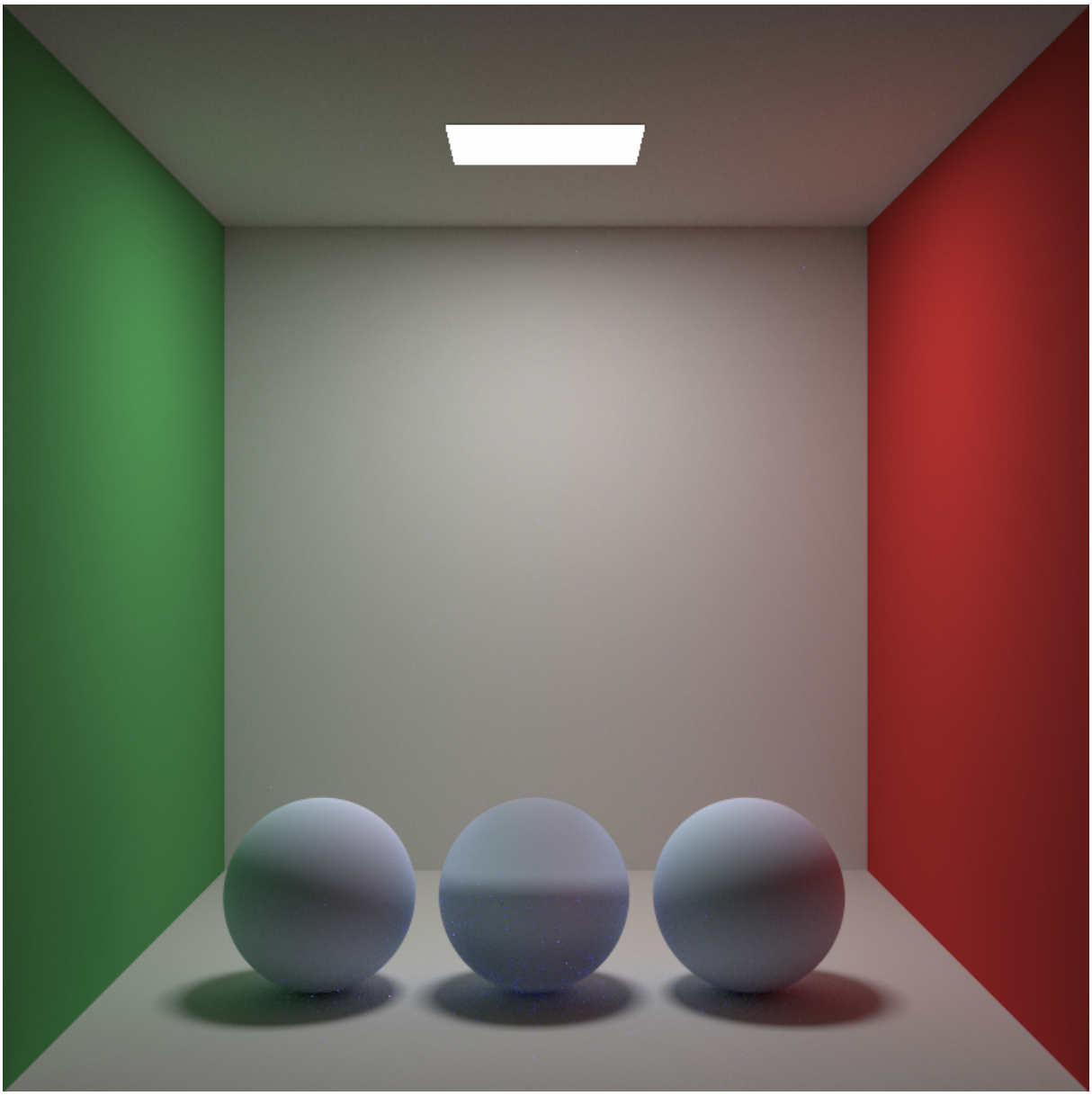


Figure A.4: Three diffuse spheres rendered with different models: the Disney model, uniform hemisphere sampling, and cosine-weighted hemisphere sampling, from left to right. Notice that the diffuse sphere rendered via uniform hemisphere sampling (middle) is noisy, despite rendering the scene with a very high samples-per-pixel count (5,000). This is why it is important to implement importance sampling (see §5.5.2).

Figure A.5: The Utah Teapot on the left is rendered with a Phong material that doesn't support importance sampling. The image on the right shows the effect of importance sampling in reducing noise. Both images were rendered at a low samples-per-pixel.

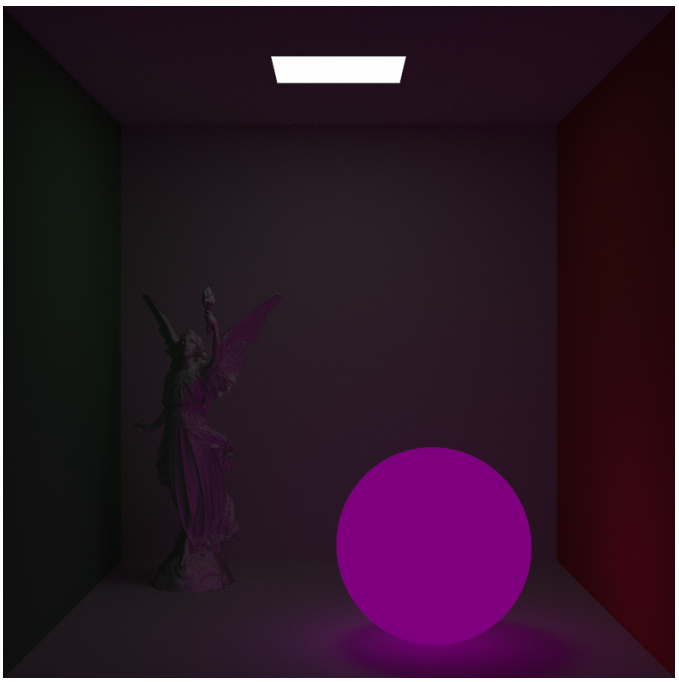


Figure A.6: *Sampling a Spherical Luminaire.*

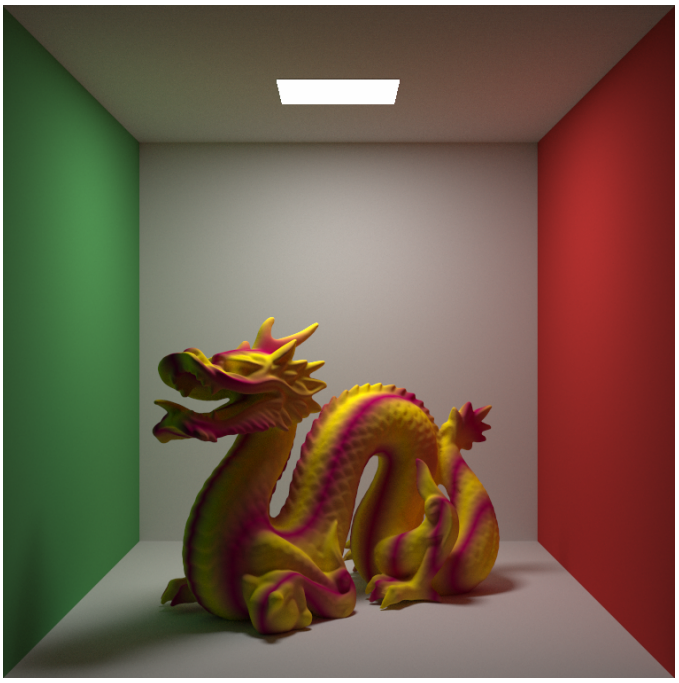


Figure A.7: *Stripes Texture.*

Student Declaration of Authorship

Declaration of Authorship

HERIOT WATT UNIVERSITY
UK | DUBAI | MALAYSIA

Course code and name:	
Type of assessment:	**Group / Individual** *(delete as appropriate)*
Coursework Title:	
Student Name:	
Student ID Number:	

Declaration of authorship. By signing this form:

- **I declare** that the work I have submitted for individual assessment OR the work I have contributed to a group assessment, is entirely my own. I have NOT taken the ideas, writings or inventions of another person and used these as if they were my own. My submission or my contribution to a group submission is expressed in my own words. Any uses made within this work of the ideas, writings or inventions of others, or of any existing sources of information (books, journals, websites, etc.) are properly acknowledged and listed in the references and/or acknowledgements section.

- I confirm that I have read, understood and followed the University's Regulations on plagiarism as published on the University's website, and that I am aware of the penalties that I will face should I not adhere to the University Regulations.

- I confirm that I have read, understood and avoided the different types of plagiarism explained in the University guidance on Academic Integrity and Plagiarism

Student Signature *(type your name):*

Date:

Copy this page and insert it into your coursework file in front of your title page.
For group assessment each group member must sign a separate form and all forms must be included with the group submission.

Your work will not be marked if a signed copy of this form is not included with your submission.

Bibliography

[Aila and Laine, 2009] Aila, T. and Laine, S. (2009). Understanding the efficiency of ray traversal on gpus. In *Proceedings of the conference on high performance graphics 2009*, pages 145–149. 18

[Akenine-Moller et al., 2019] Akenine-Moller, T., Haines, E., and Hoffman, N. (2019). *Real-time rendering*. AK Peters/crc Press. 1, 63

[Appel, 1968] Appel, A. (1968). Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 37–45. 5

[Barnes, 2019] Barnes, T. (2019). Fast, branchless ray/bounding box intersections, part 2: Nans. *URL https://tavianator. com/fast-branchless-raybounding-box-intersections-part-2-nans/. Accessed: January*. 15, 36

[Barringer et al., 2017] Barringer, R., Andersson, M., and Akenine-Möller, T. (2017). Ray accelerator: Efficient and flexible ray tracing on a heterogeneous architecture. In *Computer Graphics Forum*, volume 36, pages 166–177. Wiley Online Library. 21

[Bittner et al., 2015] Bittner, J., Hapala, M., and Havran, V. (2015). Incremental bvh construction for ray tracing. *Computers & Graphics*, 47:135–144. 14

[Blinn, 1977] Blinn, J. F. (1977). Models of light reflection for computer synthesized pictures. In *Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, pages 192–198. 10

[Boksansky, 2021] Boksansky, J. (2021). Crash course in brdf implementation. 63

[Bui-Tuong, 1975] Bui-Tuong, P. (1975). Illumination for computer generated pictures. *CACM*. 10, 12

[Burley et al., 2018] Burley, B., Adler, D., Chiang, M. J.-Y., Driskill, H., Habel, R., Kelly, P., Kutz, P., Li, Y. K., and Teece, D. (2018). The design and evolution of disney's hyperion renderer. *ACM Transactions on Graphics (TOG)*, 37(3):1–22. vi, 2

[Burley and Studios, 2012] Burley, B. and Studios, W. D. A. (2012). Physically-based shading at disney. In *Acm Siggraph*, volume 2012, pages 1–7. vol. 2012. 63

[Caulfield, 2022] Caulfield, B. (2022). What is path tracing? 5

[Chitalu et al., 2020] Chitalu, F. M., Dubach, C., and Komura, T. (2020). Binary ostensibly-implicit trees for fast collision detection. In *Computer Graphics Forum*, volume 39, pages 509–521. Wiley Online Library. 21

[Cioli et al., 2010] Cioli, S., Ordeix, G., Fernández, E., Pedemonte, M., and Ezzatti, P. (2010). Improving the performance of a ray tracing algorithm using a gpu. In *2010 XXIX International Conference of the Chilean Computer Science Society*, pages 11–20. IEEE. 17, 19, 57

[Cook et al., 1984] Cook, R. L., Porter, T., and Carpenter, L. (1984). Distributed ray tracing. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 137–145. 6

[Crockett, 1997] Crockett, T. W. (1997). An introduction to parallel rendering. *Parallel Computing*, 23(7):819–843. 15

[Eijkhout, 2017] Eijkhout, V. (2017). *Parallel programming in MPI and OpenMP*. Lulu. com. 41, 42

[Erickson, 2017] Erickson, M. (2017). Virtual reality system helps surgeons, reassures patients. 28

[Glassner, 1989] Glassner, A. S. (1989). *An introduction to ray tracing*. Morgan Kaufmann. 15, 63, 81

[He et al., 2021] He, H., Song, W., and Zong, Y. (2021). Parallel ray tracing with openmp. viii, 16, 56, 57

[Hery et al., 2013] Hery, C., Villemin, R., and Studios, P. A. (2013). Physically based lighting at pixar. *Part of "Physically Based Shading in Theory and Practice," SIGGRAPH*. vi, 9

[Jennings, 2019] Jennings, W. (2019). The limitations of transistor scaling. 17

[Johannes et al., 2021] Johannes, N., Vuorre, M., and Przybylski, A. K. (2021). Video game play is positively correlated with well-being. *Royal Society open science*, 8(2):202049. 28

[Kadir and Khan, 2008] Kadir, S. A. and Khan, T. (2008). Parallel ray tracing using mpi and openmp. *Project Report, Introduction to High Performance Computing, Royal Institute of Technology, Stockholm, Sweden*. 16

[Kajiya, 1986] Kajiya, J. T. (1986). The rendering equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 143–150. 8

[Karlsson and Ljungstedt, 2004] Karlsson, F. and Ljungstedt, C. J. (2004). *Ray tracing on programmable graphics hardware*. PhD thesis, Master's thesis, Chalmers University of Technology, Göteborg, Sweden. vi, 18

[Kay and Kajiya, 1986] Kay, T. L. and Kajiya, J. T. (1986). Ray tracing complex scenes. *ACM SIGGRAPH computer graphics*, 20(4):269–278. 15, 36

[Krogh et al., 1995] Krogh, M., Hansen, C., Painter, J., and de Verdiere, G. (1995). Parallel sphere rendering. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States). 16

[Lafortune et al., 1997] Lafortune, E. P., Foo, S.-C., Torrance, K. E., and Greenberg, D. P. (1997). Non-linear approximation of reflectance functions. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 117–126. 12

[Lafortune and Willems, 1994] Lafortune, E. P. and Willems, Y. D. (1994). *Using the modified phong reflectance model for physically based rendering.* Katholieke Universiteit Leuven. Departement Computerwetenschappen. viii, 35

[Lauterbach et al., 2009] Lauterbach, C., Garland, M., Sengupta, S., Luebke, D., and Manocha, D. (2009). Fast bvh construction on gpus. In *Computer Graphics Forum*, volume 28, pages 375–384. Wiley Online Library. 20, 52

[Lemmens et al., 2011] Lemmens, J. S., Valkenburg, P. M., and Peter, J. (2011). Psychosocial causes and consequences of pathological gaming. *Computers in human behavior*, 27(1):144–152. 28

[MacDonald and Booth, 1990] MacDonald, J. D. and Booth, K. S. (1990). Heuristics for ray tracing using space subdivision. *The Visual Computer*, 6:153–166. 14

[Marrs et al., 2021] Marrs, A., Shirley, P., and Wald, I. (2021). *Ray tracing Gems II: next generation real-time rendering with DXR, Vulkan, and OptiX.* Springer Nature. 36

[Meister et al., 2021] Meister, D., Ogaki, S., Benthin, C., Doyle, M. J., Guthe, M., and Bittner, J. (2021). A survey on bounding volume hierarchies for ray tracing. In *Computer Graphics Forum*, volume 40, pages 683–712. Wiley Online Library. 13

[Model,] Model, P. R. Importance sampling of the phong reflectance model. 77

[Möller and Trumbore, 2005] Möller, T. and Trumbore, B. (2005). Fast, minimum storage ray/triangle intersection. In *ACM SIGGRAPH 2005 Courses*, pages 7–es. 14, 49

[Navarro et al., 2014] Navarro, C. A., Hitschfeld-Kahler, N., and Mateu, L. (2014). A survey on parallel computing and its applications in data-parallel problems using gpu architectures. *Communications in Computational Physics*, 15(2):285–329. 56

[Pacheco, 2011] Pacheco, P. (2011). *An introduction to parallel programming.* Elsevier. 57

[Pallavicini et al., 2022] Pallavicini, F., Pepe, A., and Mantovani, F. (2022). The effects of playing video games on stress, anxiety, depression, loneliness, and gaming disorder during the early stages of the covid-19 pandemic: Prisma systematic review. *Cyberpsychology, Behavior, and Social Networking*, 25(6):334–354. 28

[Pantaleoni and Luebke, 2010] Pantaleoni, J. and Luebke, D. (2010). Hlbvh: Hierarchical lbvh construction for real-time ray tracing of dynamic geometry. In *Proceedings of the Conference on High Performance Graphics*, pages 87–95. 21

[Parker et al., 2005] Parker, S., Martin, W., Sloan, P.-P. J., Shirley, P., Smits, B., and Hansen, C. (2005). Interactive ray tracing. In *ACM SIGGRAPH 2005 Courses*, pages 12–es. 16, 55

[Peter Shirley, 2023a] Peter Shirley, Trevor David Black, S. H. (2023a). Ray tracing in one weekend. vii, 60

[Peter Shirley, 2023b] Peter Shirley, Trevor David Black, S. H. (2023b). Ray tracing: The next week. `https://raytracing.github.io/books/RayTracingTheNextWeek.html`. 38, 54

[Peterson and Silberschatz, 1985] Peterson, J. L. and Silberschatz, A. (1985). *Operating system concepts*. Addison-Wesley Longman Publishing Co., Inc. 15

[Pharr and Fernando, 2005] Pharr, M. and Fernando, R. (2005). *GPU Gems 2: Programming techniques for high-performance graphics and general-purpose computation (gpu gems)*. Addison-Wesley Professional. 1

[Pharr et al., 2023] Pharr, M., Jakob, W., and Humphreys, G. (2023). *Physically based rendering: From theory to implementation*. MIT Press. 38, 52, 54

[Possemiers and Lee, 2015] Possemiers, A. L. and Lee, I. (2015). Parallel importing of obj meshes in cuda. 65

[Purcell et al., 2005] Purcell, T. J., Buck, I., Mark, W. R., and Hanrahan, P. (2005). Ray tracing on programmable graphics hardware. In *ACM SIGGRAPH 2005 Courses*, pages 268–es. 17, 18

[Reinhard and Jansen, 1997] Reinhard, E. and Jansen, F. W. (1997). Rendering large scenes using parallel ray tracing. *Parallel Computing*, 23(7):873–885. 16

[Sanders and Kandrot, 2010] Sanders, J. and Kandrot, E. (2010). *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional. 17

[Schütz et al., 2022] Schütz, M., Kerbl, B., and Wimmer, M. (2022). Software rasterization of 2 billion points in real time. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 5(3):1–17. 5

[Shirley, 2016] Shirley, P. (2016). New simple ray-box test from Andrew Kensler. https://psgraphics.blogspot.com/2016/02/new-simple-ray-box-test-from-andrew.html. [Online; accessed 07-March-2024]. 15, 36

[Shirley et al., 2021] Shirley, P., Ashikhmin, M., and Marschner, S. (2021). *Fundamentals of computer graphics*. AK Peters/CRC Press. viii, 5, 7, 13, 14, 35, 38, 52, 54, 63

[Shumskiy, 2013] Shumskiy, V. (2013). Gpu ray tracing–comparative study on ray-triangle intersection algorithms. In *Transactions on Computational Science XIX: Special Issue on Computer Graphics*, pages 78–91. Springer. 49

[Simon, 2015] Simon, C. (2015). Generating uniformly distributed numbers on a sphere. viii, 35, 75

[Snyder and Barr, 1987] Snyder, J. M. and Barr, A. H. (1987). Ray tracing complex models containing surface tessellations. *ACM SIGGRAPH Computer Graphics*, 21(4):119–128. 14, 15

[Terboven et al., 2012] Terboven, C., Schmidl, D., Cramer, T., and an Mey, D. (2012). Task-parallel programming on numa architectures. In *European Conference on Parallel Processing*, pages 638–649. Springer. 55

[Trobec et al., 2018] Trobec, R., Slivnik, B., Bulić, P., and Robič, B. (2018). *Introduction to parallel computing: from algorithms to programming on state-of-the-art platforms*. Springer. 42

[Veach, 1998] Veach, E. (1998). *Robust Monte Carlo methods for light transport simulation*. Stanford University. 65

[Weisstein, 2002a] Weisstein, E. W. (2002a). Cross product. *https://mathworld. wolfram. com/*. 80

[Weisstein, 2002b] Weisstein, E. W. (2002b). Sphere point picking. *https://mathworld. wolfram. com/*. 75

[Whitted, 1979] Whitted, T. (1979). An improved illumination model for shaded display. In *Proceedings of the 6th annual conference on Computer graphics and interactive techniques*, page 14. 6, 13

[Williams et al., 2005] Williams, A., Barrus, S., Morley, R. K., and Shirley, P. (2005). An efficient and robust ray-box intersection algorithm. In *ACM SIGGRAPH 2005 Courses*, pages 9–es. 15, 36

[Woop et al., 2013] Woop, S., Benthin, C., and Wald, I. (2013). Watertight ray/triangle intersection. *Journal of Computer Graphics Techniques (JCGT)*, 2(1):65–82. 49