Physically Based Rendering

The art of the science of Light

$$L_o(p, \omega_0) = L_e(p, \omega_0) + \int_{S^2} f(p, \omega_0, \omega_i) L_i(p, \omega_i) |cos\theta_i| d\omega_i$$

Sohaib Alam

A thesis presented for an Extended Project Qualification

St Bartholomew's School
United Kingdom
November 3, 2024

Contents

1 Introduction **3**
- 1.1 Motivation . . . 3
  - 1.1.1 Success Criteria . . . 3
- 1.2 The Ray-tracing Algorithm . . . 3
- 1.3 Alternatives . . . 4
  - 1.3.1 3D Gaussian Splatting . . . 4
  - 1.3.2 Traditional Rasterisation . . . 4
- 1.4 Implementation Detail . . . 5
- 1.5 History and Current State of the Field . . . 5
  - 1.5.1 Relevant Literature . . . 5

2 Design Overview **6**
- 2.1 Phases of Execution . . . 6
- 2.2 Ray-Sphere Intersection . . . 6
- 2.3 Camera Model . . . 7
- 2.4 Ray Propagation . . . 7
  - 2.4.1 Acceleration Structures . . . 8
  - 2.4.2 Bounding Volume Hierarchies . . . 8
- 2.5 Materials . . . 8
- 2.6 Final Image . . . 10

3 Development **10**
- 3.1 Scene Description . . . 10
- 3.2 Camera . . . 10
- 3.3 Ray Propagation . . . 10
  - 3.3.1 Bounding Volume Hierarchy . . . 10
- 3.4 Ray-Object Intersection . . . 11
- 3.5 Materials . . . 12

4 Conclusion and Evaluation **12**
- 4.1 Project Evaluation . . . 13
- 4.2 Presentation Evaluation . . . 13

References **14**

A Code Snippets **16**

B Images **22**

Abstract

This thesis covers a method of Photorealistic Rendering called Physically Based Ray-Tracing (or just Ray-Tracing). Within the introduction, exists a brief history of the field, as well as the current state of the field. The research review covers the literature that was used to create this project. The design overview covers the design of the ray-tracer. The development section covers the implementation of the ray-tracer. Lastly, the conclusion and evaluation section covers the evaluation of the project, including the presentation.

1 Introduction

Since the inception of computing technology, there has been an concerted effort to replicate observable phenomena from the natural world as a means of solving problems that once couldn't be solved without computers. For example, Craig Reynolds in his seminal paper [1] is an excellent case study, of such an example. Reynolds defined three very simple rules pertaining to the behaviours of birds in a flock. These rules are:

- Alignment: the birds will steer towards the average heading of their peers.
- Cohesion: the birds will steer towards the center of the mass of its peers.
- Separation the birds will steer away from colliding with their peers.

These simple laws, though primitive, produce a "realistic" approximation of how birds behave in actuality, so much so, that the US Army uses this algorithm, for their UAV-UGV programs [2].

In essence, computer simulation provides a "good enough" approximation for the theories encompassing the real world, crafted by mathematicians and physicists alike.

1.1 Motivation

Rendering is the process of producing an image from a description of a 3D scene. This daunting task can be easily understood by asking: Given a set configuration of a room, what would a camera see, in a set location within the room. If left pondering, one can easily come to a reasonable algorithm, befit the style of rendering they want. For example, a very simple approach could be to check if a light ray enters the camera from light sources (if any) within the room. If so, then the camera could record a colour based on some product of the colours the light ray hit before it entered the camera.

However, I intend to implement the "Physically Based" ray-tracing algorithm. As the name suggests, this algorithm tries to stay true to the physics of the world, imitating its behaviour as matter and light mesh together. It differentiates different materials dependent on their reaction with light incident upon them, as well as how light itself reacts through mediums not necessarily vacuums, like fog.

I intend to implement this algorithm not only because it is a excellent opportunity to mix the three subjects I do for A-Level together (Physics, Maths, Computing), but also as an challenging extension in the field, in which I will be considering a future in.

1.1.1 Success Criteria

The objective of this project is to create a piece of software, that given a scene description will produce an image, following the rules of physically based rendering. I aim to achieve the following:

- Correct Ray-Object Intersection
- Correct Ray-Object Reflection/Refraction
- Correct Light Source Distribution
- Correct Indirect Light Transport
- Interesting Objects
- Correct Materials

I understand that knowing what the success criteria is vague, I hope to make these more concrete in the design overview.

1.2 The Ray-tracing Algorithm

PBRT (Physically Based Ray-tracing), can be easily split up into several distinct componets in a "pipeline" or "story" of sorts. This can be done, primarily because PBRT is an algorithm, a series of steps, but also because elements of pipeline or story can easily

be implemented independently, and understood independently, as a function with an input and an output.

Going back to the idea of thinking PBRT as imagining a camera in a room, we can easily demarcate different functions of the algorithm. The following steps are derivative of the steps outlined in [3].

- Camera model: The camera model given its initial location and orientation within 3D space, must provide a method of generating rays in a scene (Reference place ahead). It must also do pre and post-processing if necessary (.i.e. different film types).
- Ray propagation: The ray generated from the camera will traverse the scene differently, depending on which medium it is in. For example, in a vacuum, all light energy is conserved, but that is a rare case on Earth, therefore extra calculations must occur to account for different mediums (like through a fog).
- Ray-object Intersection: One can only "trace rays", if said ray hits (and bounces off) an object. The program must detect when there is a collision between an object and a ray, and appropriately "cause" the ray to bounce of the object in the appropriate direction. The algorithm must also distinguish between different objects (usually returning the closest one). Also, the ray tracer can only generate images as sophisticated as the objects it can detect, so the program must also provide intersection tests with different kinds objects (triangles, circles, etc). Due to this, finding intersections is the most expensive process of the entire algorithm, so the program must also provide some method of culling objects not near the ray.
- Light sources: Without a light in a room, it would be pointless to render a scene. The ray tracer must accurately describe a light distribution within a scene. As a part of this, the tracer must also distinguish whether a point in a scene has any light shining on it or not.
- Indirect Light Transport: Because light can arrive at a surface after bouncing off or passing through other surfaces, it is usually necessary to trace additional rays originating at the surface to fully capture this effect [3].

1.3 Alternatives

Most algorithms usually have alternatives, and Physically Based Rendering also has alternatives

1.3.1 3D Gaussian Splatting

3D Gaussian Splatting is a brand-new method in real-time rendering.[4] provides a nascent method that rasterises 3D Gaussian, rather than raytracing (which is what we use). Gaussian Splatting is an interesting topic on its own, however, I will not be using this method because, while this method is good for real-time rendering, I will be conducting "physically-based" rendering. The former, while faster than the latter, but the latter is more accurate than the former. Since I would like to focus on accuracy, I will not be implementing this method.

1.3.2 Traditional Rasterisation

Traditional Rasterisation, using basic rendering techniques available, using the GPU, is similar to Gaussian Splatting, and while it is fast, by using specialist equipment (GPU), provides speed but not accuracy[1].

[1]This was a larger problem, when GPUs did not necessarily conform with IEEE 754, but most GPUs are accurate enough, the issue of time also comes into the question. Also the architecture of GPUs prevent efficient raytracing.

1.4 Implementation Detail

This project will be implemented with C++. While there are "easier" and more readable languages, C++ provides efficiency unparalleled to any other language. Its granular memory controls and threading support make it invaluable to a project of such caliber of complexity. I talk more about this in the design overview.

1.5 History and Current State of the Field

Unsurprisingly, Physically Based Rendering (PBR), is a relatively nascent field (only being studied since the 70s), and as the field has advanced to cleverer solution to increasingly difficult problems. For example, in the 70s, the biggest problem to solve was the lack of memory available to computers (1 MB at its rare), where physical accuracy was not biggest focus, but to speed up how long an image took to render [3]. This was achieved by taking a subset of the entire scene representation into memory, however this caused problems with global illumination algorithms, as this (the name might suggest) required a larger (or the entire) scene to be loaded into memory. So, with speed, creators of such graphics decided that geometric accuracy was more important then realism of light. Because of this, many people though at the time that realism was not as important as artistic merit, where the computer graphics was used as an aid, rather than a tooling which is essential from the ground up.

The best example of this is the rise of Computer Generated Imagery (CGI) within the film industry, where some of the biggest blockbuster releases, leveraged the power of computers to render backgrounds and add complex elements to a scene, or augment a character's physical appearance (e.g. Terminator 2) [5].

When computers did become advanced enough to render scenes in full, many film studios adopted the physically based rendering pipeline into their film-making process. A good case study is Pixar, who have been propellers of this field into the mainstream. Their *Renderman* renderer has used the photorealistic algorithm, and have won several awards for their films. They also contribute to the SIGGRAPH community, which is another essential source in this field.

However, as Jim Blinn states: "as technology advances, rendering time remains constant". This observes that as technology advances, people's demand for "better" and "more realistic" images increases, rather than being content with current standards, and rendering those scenes faster [3].

1.5.1 Relevant Literature

Physically Based Rendering: From Theory to Implementation 3rd edition [3] is perhaps the best book on this topic. It was written by ex-NVdia and ex-Google employees, who specialise in image rendering. It is the only consolidatory source in this field and provides excellent resources for further research (which proved useful for certain topics within my EPQ), as well as provide exercises to get a better understanding of the topic. If I have to mention any drawbacks about this literature, would be its age. Written in 2016, this book lacks to mention any alternatives, or cover use of specialist equipment, like GPUs. However, do note this problem is remedied within the 4th edition of the book.

Glassner's introduction to the ray-tracing algorithm provides an excellent insight into the basics of the algorithm. Though from the 80s, it still is a excellent foundation for more complex and modern algorithms [6]

Highner's book on numerical algorithms, isn't related to ray-tracing or PBR, but provides an excellent bank of algorithms and "tricks" to solving complex problems, such as the Monte-Carlo Integration algorithm [7].

Glassner's book on "Graphics Gems" is also a very useful resource and similar Highner's book, provides insightful knowledge into algorithms referenced in Physically based Rendering 3rd ed [8].

Woop, Benthin and Wald's paper on ray-triangle intersections is a essential resource, as provides a novel algorithm in solving the problem of ray-triangle intersections. Without it, it would be near impossible to render complex objects which aren't quadrics [9].

Shirley's 3-part series on ray-tracing are an excellent method, from which I dipped my feet into the world of PBR. While the premise of the series might be considered by most as simple, it still is an important resource on solving problems simply rather than robustly [10].

Cohen and Kappauf's paper on colour theory was also useful when making the camera of the ray-tracer, it broke down the complicated Commission Internationale de l'Éclairage (CIE) standards on colour [11].

2 Design Overview

In this chapter, I will outline the artefact's inner workings; showing which components are used where, and how they interact. Much of the system has been informed by how [3] designed their system, albeit modified to fit our use case.

2.1 Phases of Execution

Initially, the program creates its own representation of the scene being rendered. This is done by creating scene objects and materials and then adding them to the scene.

Each object is treated as a *hittable* object, which means that it can be hit by a ray. Note this is also how [10] designed and called their system. A hittable object is defined as an abstract class, meaning that it can be inherited by other classes, and custom functions can be implemented for each object.

For example, the *hittable* object offers a function *hit* which returns a *hit_record*

And so with a sphere, we can implement a custom *hit* function, that takes into account the parameters of the sphere. For demonstration purposes, we will derive this function in order to illustrate how the ray-hit algorithm works.

2.2 Ray-Sphere Intersection

Note that we have a ray $R(t) = A+tb$ where A is the origin of the ray, b is the direction of the ray, and t is the distance from the origin (acting as a parameter) [8]. We also have a sphere with a center C and a radius r. We can write the equation of the sphere, centered at the origin, with radius r as

$$x^2 + y^2 + z^2 = r^2$$

The equation can be thought of as also asking whether a given point is inside, outside, and when combined with the ray: whether the ray is a tangent [to the sphere], or only touches the sphere at a point, or if it intersects the sphere at two points. To move towards this ideal goal, we can represent as the centre of the sphere as $C = (C_x, C_y, C_z)$, and move towards writing the equation of the sphere in terms of vectors. Let P represent an arbitrary point in 3D space:

$$(C - P) \cdot (C - P) = r^2$$

We can represent the point P, as a point on our ray ($P = A + tb$), and thus we get:

$$(C - (A + tb)) \cdot (C - (A + tb)) = r^2$$

Writing in terms of t:

$$t^2 b \cdot b + 2tb \cdot (A - C) + (A - C) \cdot (A - C) - r^2 = 0$$

This is in the form of the quadratic equation in terms of t, and can be thought of asking if there is a parameter t for some ray (a point along the ray), which touches or intersects

the sphere. Using the quadratic formula, we can solve for t:

$$a = b \cdot b$$
$$b = 2b \cdot (A - C)$$
$$c = (A - C) \cdot (A - C) - r^2$$

and t as

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

If the discriminant $b^2 - 4ac$ is less than 0, then the ray does not intersect the sphere, if it is equal to 0, then the ray is tangent to the sphere, and if it is greater than 0, then the ray intersects the sphere at two points.

By allowing child classes to implement their own *hit* function, we can easily add new objects to the scene, and the ray tracer will automatically detect them, so long as the hit function has been implemented correctly.

We allow all objects to have a material associated with them, which is a class that defines how light interacts with the object, further information on this will be provided in the appropriate section.

Once all the objects have been added to the scene, the program will begin the ray tracing process. In order to begin ray tracing, once must obtain these rays from the camera.

Since the ray tracing process acts as a reverse image, by tracing these rays back to a light source, we can obtain the colour of the pixel. Done for all rays, we can obtain the final image.

2.3 Camera Model

For this project, we will use a very simple camera model, a pinhole camera. Pinhole cameras are the simplest form of camera, and are often used in ray tracing algorithms as a beginner camera model. The camera is defined by a position, a look-at point, and a field of view. The camera generates rays by shooting rays from the camera's position to the point on the screen, and then normalizing the vector to get a unit vector as the direction of the ray. The camera generates rays in a grid pattern, and the number of rays generated is determined by the resolution of the image [10].

The pinhole camera does not have a lens, and thus does not have any distortion. This is a limitation of the pinhole camera, and is not representative of real-world cameras. However, for the purposes of this project, the pinhole camera is sufficient.

2.4 Ray Propagation

Once the rays have been generated by the camera, the rays are propagated through the scene. The use of the term "propogate" is a misnomer in the context of this system, since we are not transporting these rays through a medium, like fog or water [3] but rather tracing these rays (thus the name), back to a light source.

There are three cases we will need to consider when propagating rays:

- The ray intersects a light source
- The ray does not intersect anything, and is in the void
- The ray intersects an object

If the ray intersects a light source, then the job is simple, we return the colour of the light source. Similarly, if the ray does not intersect anything, then we either return black as the colour of the void, or the background colour if specified within the scene.

The most complicated case is when the ray intersects an object, because here the system must calculate two things:

- The colour of the object, which is the colour of the object at the point of intersection
- The outgoing ray, which is the ray that is reflected or refracted off the object

In order to calculate the colour of the object, we must know the lighting conditions within the scene, however this can only be determined by the calculating the outgoing ray within the scene, to find a light source. One can immediately notice that this problem can be solved recursively.

By the process of recursion, the system treats the ray out of the camera and the ray out from the object as the same general 3D ray against a general scene, treating as some geometric problem. This is where the ray-tracing algorithm spends the most of its execution time, as to get a good approximation of the lighting within the scene, the ray must bounce > 50 times [3].

2.4.1 Acceleration Structures

As established, the ray-tracing algorithm will spend most of its time finding intersections between rays and objects. The most naive method would be to check a ray's intersection with every single object within the scene. One method of speeding up this process of finding intersections is to use an acceleration structure [3]. These structures allow the ray-tracer to effectively cull objects not near the ray, and focus more computation on objects with a higher probability of being hit.

There exist a few varieties of acceleration structures, such as bounding volume hierarchies, and kd-trees [3, 12]. And this system uses the former to speed up the ray-tracing process. There is no compulsion to choose one over the other, since both provide equivalent performance improvements, so the choice to the use the former is completely arbitrary.

Another acceleration structure used within this system is the "OctTree", however the use of such was more relevant when implementing the system, and thus will not be discussed here.

2.4.2 Bounding Volume Hierarchies

A BVH is a tree structure where each node represents a bounding volume containing a subset of objects in the scene [13]. At the root, the bounding volume encloses all objects within the scene. Moving down the tree, each child node represents a smaller bounding volume that encloses a subset of objects from its parent node. The tree terminates at the leaves, where the bounding volumes typically contain individual objects or small groups of objects. The bounding volume is generally chosen to be a simple shape, like an axis-aligned bounding box (AABB) or a sphere, making intersection tests computationally inexpensive.

The efficiency of a BVH stems from its hierarchical structure. When a ray traverses the scene, it first checks for intersection with the bounding volume at the root. If there is no intersection with the root volume, the ray can discard all objects within the root's subtree, effectively "culling" a large portion of the scene. If an intersection is found, the ray recursively traverses the tree, testing intersections with each child node. This process narrows down the set of objects the ray intersects, focusing computational resources on the areas with higher likelihood of an intersection [13].

2.5 Materials

Materials are an important part of the ray-tracing process, as they define how light interacts with objects in the scene, and in turn determine how the viewer percieves the scene.

The system will provide three materials for the scene to use:

- Lambertian
- Metal
- Dielectric

One important note about this system, is that is lacks the usage of BxDFs to describe

materials, which are a family of probability distributions that describe how light interacts with surfaces [3]. This is an intentional limitation introduced to simplify the system, as the goal for this project was to build a simple ray-tracer, with all of its foundations rather than a full-featured ray-tracer.

The Lambertian material is the simplest of the three materials provided, and is used to represent an ideal diffuse surface. This entails that the reflected ray is scattered proportional to the cosine of the angle between the incoming ray, and the normal of the surface [10]

The Metal material is used to describe a surface that reflects light. Though called a metal, it can also be used to represent mirrors, or any other surface that reflects light. The metal material is defined by a colour, and a roughness parameter, which dictates how much the reflected ray is scattered from its true reflection. For example, a roughness of 0 would represent a perfect mirror, where the angle of incidence is equal to the angle of reflection, and a roughness of 1 would represent a diffuse surface, where the reflected ray is likely to be scattered in any direction [3, 10].

Lastly, the most complicated material that will be implemented is the Dielectric material. This material represents transparent surfaces, such as glass or water. Such materials, instead of reflecting light refract light, which means that the light is bent as it passes through the surface. Such refraction is described by Snell's Law [14]:

$$\eta_i \sin\theta_i = \eta_t \sin\theta_t$$

In order to know the direction of the refracted ray, we must know the angle of refraction, which is written as:

$$\sin\theta_t = \frac{\eta_i}{\eta_t}\sin\theta_i$$

If we treat the refracted ray as a vector R', then we can describe the refracted ray as the sum of the vector $R'_\perp$ which is perpendicular to the surface normal n and the vector $R'_\parallel$ which is parallel to the surface normal n.

$$R' = R'_\perp + R'_\parallel$$

The vector $R'_\parallel$ is given by:

$$R'_\parallel = -\sqrt{1 - |R'_\perp|^2}n$$

and the vector $R'_\perp$ is given by:

$$R'_\perp = \frac{\eta_i}{\eta_t}(R + \cos\theta n)$$

where R is the direction of the incoming ray.

In the above equations, we know of all terms except the cosine component. Recall that the cosine component is also present in the dot product of two vectors, thus our $R'_\perp$ can be written as:

$$R'_\perp = \frac{\eta_i}{\eta_t}(R + (-R \cdot n)n)$$

The proof and derivation of the above equations is left as an exercise to the reader.

Going back to Snell's Law:

$$\sin\theta' = \frac{\eta}{\eta'}\sin\theta$$

If the ray is incident from glass to air ($\eta = 1.5$ and $\eta' = 1.0$):

$$\sin\theta' = \frac{1.5}{1.0}\sin\theta$$

But the value of $sin\theta'$ cannot be greater than 1, So if:

$$\frac{1.5}{1.0}\sin\theta > 1$$

Then the inequality is not satisfied, and our Snell's Law is broken. In this case we must reflect the ray. This is known as Total Internal Reflection [14], and must be something the system must account for if it is to be accurate.

2.6 Final Image

Once all the rays have been propagated through the scene, the system will have a list of colours for each pixel in the image. The system will then write these colours to a file, which can be viewed by the user.

3 Development

In this chapter I will discuss the implementation details of the project, as outlined in the design overview. I will outline some challenges faced and how they were solved. Please refer to the appendix for the full code listings.

3.1 Scene Description

Before starting the ray-tracing algorithm, the system must know the parameters of the scene. This includes the parameters about the pinhole camera and the objects in the scene.

Initially, all objects are stored in a large list. As described in the design overview, all objects are derived from the base class *hittable* which has a virtual function *hit* which is overridden by all derived classes. This allows for a single list to store all objects as shared pointers to the base class, but allows us to call the overridden *hit* function of the derived class, due to the polymorphic nature of C++ pointers [15].

Each object must also provide a material, which within the *hittable* object is defined as a reference[2] to a *Material* object.

As with the *hittable* object, the *Material* object is a base class, which is overridden by derived classes. Note that we describe a light source using a material, unlike [3], where light sources are a separate entity. This is because we can describe a light source as a material, and this allows for a more consistent interface, as opposed to having a separate entity for light sources, which is primarily used for BxDFs.

There also exist some special 3D objects, such as triangles, while derived from the *hittable* object, require another argument to be present, namely the *Mesh* and its constituent *Triangle* objects. The reason will become apparent in the relevant section.

3.2 Camera

The camera is defined by the position of the camera, the point it is looking at, and the up vector, using these parameters, the system can calculate the camera's orientation.

The camera class describes methods to generate rays, given a pixel on the screen. The camera generates rays by first generating a ray from the camera to the point on the screen, then jittering the ray to account for the depth of field. The jittering is done by generating a random point on the lens of the camera, and calculating the ray from the camera to the point on the screen, through the point on the lens.

3.3 Ray Propagation

As said in the design overview, the propagation of rays is the longest part of the algorithm. Therefore, when the ray is propagated through the scene, the system must check for the intersection of the ray with objects within the scene. This system used a bounding volume hierarchy (BVH) as an acceleration medium.

3.3.1 Bounding Volume Hierarchy

The BVH is a tree structure, where each node has two children, and each child has two children, and so on. Therefore, the implementation of the BVH is much akin to an implementation of a binary tree, where the main structure defined is a BVH node, which has two children (also BVH nodes),

[2] C++ makes a distinction between references and pointers, and also within the field of Computer Science. However, in this paper the term reference and pointers are used interchangeably.

and a bounding box. The bounding box is defined as the smallest box that can contain all the object(s) in the BVH node. Listing 3 shows the structure of the BVH node.

When actually constructing the final BVH, the process first creates a bounding box that contains all the objects in the scene, then splits the bounding box into two smaller bounding boxes, such that the objects are evenly distributed between the two bounding boxes. This process is repeated until the bounding box contains a single object, or some objects.

3.4 Ray-Object Intersection

In this section, to serve as an example, I will divulge into the process of implementing the intersection of complex objects using triangles and Meshes.

To serve as context, it is important to note that all 3D objects can be represented as a collection of triangles. The more triangles used to describe an object, the more accurate the representation of the object.

[9, 16] are the sources used to implement the Möller-Trumbore algorithm, which is a fast algorithm to calculate the intersection of a ray with a triangle. The algorithm is as follows:

1. Calculate the normal of the triangle.
2. Calculate the determinant of the matrix formed by the ray direction and the edges of the triangle.
3. If the determinant is zero, the ray is parallel to the triangle, and there is no intersection.
4. Calculate the barycentric coordinates of the intersection point.
5. If the barycentric coordinates are within the triangle, there is an intersection.
6. Calculate the intersection point in this case.

While for a singular triangle, this process is trivial, this problem is more complicated when dealing with a mesh, which could possibly consist of more than a million triangles. This problem is similar to the performance bottleneck experienced in the naive ray object intersection, where we brute force check every object in the scene, but in this case we brute force check every triangle in the mesh with the ray.

To solve this problem, like the BVH, we can use another acceleration structure, which in this case is an Oct-Tree. An Oct-Tree is another acceleration structure used to optimize the intersection-finding process within ray tracing systems [17]. While a Bounding Volume Hierarchy (BVH) organizes objects into hierarchically nested bounding volumes of arbitrary size, the Oct-Tree divides the entire 3D space into fixed, equally-sized regions known as “octants.” Each subdivision of space in the Oct-Tree creates eight child regions, hence the name ”Oct-Tree.”

The Oct-Tree is constructed by recursively subdividing each parent node into eight smaller octants, each representing an equal partition of its parent volume. This process continues until the tree reaches a certain level of granularity, often defined by the number of objects in each region. Unlike BVHs, where bounding volumes are based on object positions, Oct-Trees create fixed spatial partitions that do not adjust to object locations. This can make Oct-Trees more effective for triangles meshes, since the fixed regions can provide a more uniform distribution of objects across the tree.

Using this we can reduce the number of triangles we need to check for intersection with the ray, and therefore reduce the time taken. This also allows us to render more complex objects. Figure B.1 shows the St Lucy model, which is an example of a complex object that can be rendered using this method; with the OctTree took appropriately 10 minutes to render the model, with

2 million triangles.

3.5 Materials

Materials are defined by the *Material* class, which is a base class that is overridden by derived classes.

Whilst most materials consist of a single color, this project also allows textures to be used as the base of the materials. Figure B.2 shows a Perlin Noise texture applied to the floor and the sphere. In order to apply a texture to the sphere, it must be noted that on a unit sphere (a sphere with a radius of 1), each point on the sphere can be represented using spherical coordinates (θ, ϕ), where θ is the angle from the bottom pole $(-y)$, and ϕ is the angle from the $-x$-axis to $+z$-axis (so around the y axis)[3]. We can translate between Cartesian form and spherical form using the following equations:

$$\begin{aligned} y &= -\cos(\theta) \\ x &= -\sin(\theta)\cos(\phi) \\ z &= \sin(\theta)\sin(\phi) \end{aligned}$$

From this we can use our θ and ϕ to sample the texture, and apply it to the sphere, by creating two new variables to normalise our θ and ϕ to the range $[0, 1]$, and then sample our texture at the point (u, v), where u is the normalised ϕ, and v is the normalised θ.

4 Conclusion and Evaluation

Overall, I think this project has been very successful in meeting its criteria defined. As a reminder, the success criteria were:

- Correct Ray-Object Intersection
- Correct Ray-Object Reflection/Refraction
- Correct Light Source Distribution
- Correct Indirect Light Transport
- Interesting Objects
- Correct Materials

Figure B.3 showcases the best of what this project can produce. The reflections of the chess pieces (and the reflections of said reflections) can be seen in high quality, accurately modelling light transport, and distribution of light from sources. In addition, the Lambertian chess pieces also "absorb" the colour of the walls that surround them, and the floor utilises a checkerboard texture, which also reflective, thus showing an accurate model of the Lambertian materials.

As shown before, Figure B.1 shows the St Lucy model, which is an example highlighting the best of the ray-object intersection algorithm implemented within this project. The St Lucy model is a complex object, with over 2 million triangles, and the Oct-Tree acceleration structure allowed the model to be rendered in 10 minutes. The Oct-Tree structure also works alongside the Möller-Trumbore algorithm, which allows for interesting objects to be rendered.

Figure B.4 showcases the Chinese Dragon with a metallic material applied to it. The dragon here isn't that reflective, however still reacts to the strong light source at the top of the screen, as well as the walls to red and green walls to its side. One can note the distinct lack of shows in this image, especially on the floor. The reason behind this is that this ray-tracer does not include shadow rays, whose purpose is to determine if a point in the scene, however by not including any shadow rays, we get caustics and subsurface scattering for free [10].

Figure B.5 shows the standard Cornell Box. The Cornell Box is a standard test scene in the field of ray-tracing, and is used to test the accuracy of the ray-tracer. The Cornell Box is a simple scene, with a light

[3]This is opposite to the convention used in most physics textbooks

source, a few walls, and a few objects. This image shows the biggest limitation of this ray-tracer, the lack of global illumination. Global illumination is the process of light bouncing off surfaces and illuminating other surfaces. This is a very complex process, and is usually done by using BxDFs as well as Monte-Carlo Integration, which is a very complex algorithm [7]. Without using BxDFs, the image produced is very "grainy" or "noisy". This is because not all rays out of the camera, end up hitting a light source, and therefore the image is incomplete. Using BxDFs allow the rays to be biased towards the light source, so more rays end up hitting the light source. This process was not implemented in this project, primarily due to time constraints because of the complexity of the algorithm.

4.1 Project Evaluation

Time constraints were a huge issue during the development of this project, as I had to balance this project with my A-Level studies. This meant that I had to cut some features, such as BxDFs. For example, I could not work on the project during my practice exams which meant I was behind schedule, however I managed to catch up during the summer holidays.

Another issue was that I felt that I was overtly ambitious with the project, as I wanted this to be a serious ray-tracer, however I did not realise the true complexity of setting up the code infrastructure to even begin ray-tracing. What I had not realised was that my primary source for this project was not a recipe book on how to make a ray-tracer, but rather an encyclopaedia on the theory of ray-tracing. This meant that I had to spend a lot of time translating abstract math written on paper into concrete code. This was a very difficult process, as sometimes I had to teach myself the math[4], before I could even begin to implement it. As a result, my project timeline was completely wrong at the beginning, where I had intended to finish my project by June. However, our EPQ deadline was extended before I planned my project timeline, therefore I will evaluate myself too harshly on this aspect. Nonetheless, I found it a very rewarding experience, as I moulded my own ray-tracer from scratch, rather than following a cookie-cutter recipe on the Internet[5].

4.2 Presentation Evaluation

My presentation of this project, however, went extremely well. I used Manim [19] to create a presentation of my project, that was animated, and therefore engaging. I managed to brief the audience as to the reasoning behind why one would want to use a ray-tracer, as well as an apologia for some decisions and limitations of my project. However, I felt as if my presentation could have benefited from me slowing down. On the other hand, in the 10-minute limit we were given, I think I could not have done much better explaining the fundamentals of ray-tracing, while also giving enough respect to the evaluations of my project.

In conclusion, I think this EPQ was a major success for me, as I managed to create a ray-tracer from scratch, and present it in an engaging manner. Whilst I did not manage to implement all the features I wanted to, by not implementing them, I have learned a lesson in being realistically ambitious. I have also learned a lot about ray-tracing, and the theory behind it, which will be beneficial in my university studies in Computer Science.

[4] A lot of which was and is not covered in my A-Level course

[5] See [18] for context

References

1. Reynolds, C. W. *Flocks, herds and schools: a distributed behavioral model* in *Proceedings of the 14th annual conference on Computer graphics and interactive techniques* (Association for Computing Machinery, New York, NY, USA, Aug. 1987), 25–34. ISBN: 978-0-89791-227-3. https://dl.acm.org/doi/10.1145/37401.37406 (2024).
2. Saska, M., Vonásek, V., Krajník, T. & Přeučil, L. Coordination and navigation of heterogeneous MAV–UGV formations localized by a 'hawk-eye'-like approach under a model predictive control scheme. en. *The International Journal of Robotics Research* **33.** Publisher: SAGE Publications Ltd STM, 1393–1412. ISSN: 0278-3649. https://doi.org/10.1177/0278364914530482 (2024) (Sept. 2014).
3. Pharr, M., Jakob, W. & Humphreys, G. *Physically Based Rendering: From Theory to Implementation (3rd ed.)* 3rd. ISBN: 978-0-12-800645-0 (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, Nov. 2016).
4. Kerbl, B., Kopanas, G., Leimkuehler, T. & Drettakis, G. 3D Gaussian Splatting for Real-Time Radiance Field Rendering. en. *ACM Transactions on Graphics* **42,** 1–14. ISSN: 0730-0301, 1557-7368. https://dl.acm.org/doi/10.1145/3592433 (2024) (Aug. 2023).
5. Bramesco, C. Terminator 2 at 30: groundbreaking sequel that led to CGI laziness. en-GB. *The Guardian.* ISSN: 0261-3077. https://www.theguardian.com/film/2021/jul/03/terminator-2-at-30-groundbreaking-sequel-that-led-to-cgi-laziness (2024) (July 2021).
6. Glassner, A. S. *An Introduction to Ray Tracing* en. Google-Books-ID: YPblYyLqBM4C. ISBN: 978-0-12-286160-4 (Morgan Kaufmann, Jan. 1989).
7. Higham, N. J. *Accuracy and stability of numerical algorithms* 2nd ed. en. ISBN: 978-0-89871-521-7 (Society for Industrial and Applied Mathematics, Philadelphia, 2002).
8. *The AP Professional graphics CD-ROM* en (eds (Firm), A. P. & (Firm), E. E. I.) Medium: computer file. ISBN: 978-0-12-059756-7 (AP Professional, Chestnut Hill, Mass., 1995).
9. Woop, S., Benthin, C. & Wald, I. Watertight Ray/Triangle Intersection. en. **2** (2013).
10. Peter Shirley, Trevor David Black, Steve Hollasch. *Ray Tracing in One Weekend* https://raytracing.github.io/books/RayTracingInOneWeekend.html#citingthisbook/basicdata (2023).
11. Cohen, J. B. & Kappauf, W. E. Color Mixture and Fundamental Metamers: Theory, Algebra, Geometry, Application. *The American Journal of Psychology* **98.** Publisher: University of Illinois Press, 171–259. ISSN: 0002-9556. https://www.jstor.org/stable/1422442 (2024) (1985).
12. Arvo, J. in *Graphics Gems* (ed Glassner, A. S.) 548–550 (Morgan Kaufmann, San Diego, Jan. 1990). ISBN: 978-0-08-050753-8. https://www.sciencedirect.com/science/article/pii/B9780080507538501194 (2023).
13. Wald, I. *On fast Construction of SAH-based Bounding Volume Hierarchies* en. in *2007 IEEE Symposium on Interactive Ray Tracing* (IEEE, Ulm, Sept. 2007), 33–40. ISBN: 978-1-4244-1629-5. https://ieeexplore.ieee.org/document/4342588/ (2024).
14. Bone, G. *A Level Physics for OCR A Student Book* en. Google-Books-ID: 2eQ2EAAAQBAJ. ISBN: 978-0-19-837864-8 (Oxford University Press - Children, May 2016).

15. *C++17 - cppreference.com* https://en.cppreference.com/w/cpp/17 (2024).
16. Möller, T. & Trumbore, B. Fast, Minimum Storage Ray-Triangle Intersection. *Journal of Graphics Tools* **2.** Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/10867651.1997.10487468, 21–28. ISSN: 1086-7651. https://doi.org/10.1080/10867651.1997.10487468 (2024) (Jan. 1997).
17. Meagher, D. *Octree Encoding: A New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer* (Oct. 1980).
18. *How to Escape Tutorial Hell* en. https://www.linkedin.com/pulse/how-escape-tutorial-hell-ikechukwu-vincent (2024).
19. *Manim Community* en. https://www.manim.community/ (2024).

A Code Snippets

Listing 1: "Camera settings"

```
camera cam;

cam.aspectratio = 16.0f / 9.0f;
cam.image_width = 1200;
cam.samples_per_pixel = 10;
cam.max_depth = 20;

cam.fov = 20;
cam.lookfrom = point3(13, 2, 3);
cam.lookat = point3(0, 0, 0);
cam.vup = Vec3(0, 1, 0);

cam.background = color(0.7, 0.8, 1.0);
```

Listing 2: "Generation of Rays"

```
Ray sendRay(int i, int j, int s_i, int s_j) {
    auto offset = sample_square_stratified(s_i, s_j);
    auto pixel_sample =
        pixel00_loc + ((float) (i) * pixel_delta_u) + ((
           float) (j) * pixel_delta_v);

    glm::vec3 ray_origin =
        (defocus_angle <= 0) ? center : defocus_disk_sample
           ();
    glm::vec3 ray_direction = pixel_sample - ray_origin;

    return Ray(ray_origin, ray_direction);
  }
```

Listing 3: "BVH Node Structure"

```
class bvh_node : public Hittable {

public:
  bvh_node(hittable_list &list)
      : bvh_node(list.objects, 0, list.objects.size()) {}

  bool hit(const Ray &r, float t_min, float t_max,
           hitrecord &rec) const override {
    if (!bbox.hit(r, interval(t_min, t_max)))
      return false;

    bool hit_left = left->hit(r, t_min, t_max, rec);
    bool hit_right = right->hit(r, t_min, hit_left ? rec.t :
       t_max, rec);
```

```
        return hit_left || hit_right;
    }

    aabb bounding_box() const { return bbox; }

    bvh_node(std::vector<shared_ptr<Hittable>> &objects,
        size_t start,
              size_t end);

private:
    std::shared_ptr<Hittable> left;
    std::shared_ptr<Hittable> right;
    aabb bbox;

    static bool box_compare(const shared_ptr<Hittable> a,
                            const shared_ptr<Hittable> b, int
                              axis) {
      auto a_axis_interval = a->bounding_box().axis_interval(
          axis);
      auto b_axis_interval = b->bounding_box().axis_interval(
          axis);
      return a_axis_interval.min < b_axis_interval.min;
    }

    static bool box_x_compare(const shared_ptr<Hittable> a,
                              const shared_ptr<Hittable> b) {
      return box_compare(a, b, 0);
    }
    static bool box_y_compare(const shared_ptr<Hittable> a,
                              const shared_ptr<Hittable> b) {
      return box_compare(a, b, 1);
    }
    static bool box_z_compare(const shared_ptr<Hittable> a,
                              const shared_ptr<Hittable> b) {
      return box_compare(a, b, 2);
    }
};
```

Listing 4: "Oct-Tree definition and implementation"

```
class Octree
{
public:
    Octree(const aabb& boundingBox, int maxDepth = 15, int
        maxTrianglesPerNode = 100)
        : bbox(boundingBox), depth(0), maxDepth(maxDepth),
          maxTrianglesPerNode(maxTrianglesPerNode)
```

```
    {
    }

    void build(const std::vector<std::shared_ptr<Triangle>>&
        triangles, int depth = 0);

    bool hit(const Ray& r, float t_min, float t_max, hitrecord&
        rec) const;

private:
    void subdivide();

private:
    aabb bbox;
    int depth;
    int maxDepth;
    int maxTrianglesPerNode;

    std::vector<std::shared_ptr<Triangle>> triangles;
    std::unique_ptr<Octree> children[8];
};
void Octree::build(const std::vector<std::shared_ptr<Triangle> >
    &triangles, int depth) {
    this->depth = depth;

    if (triangles.size() <= maxTrianglesPerNode || depth >=
        maxDepth) {
        this->triangles = triangles;
        return;
    }

    subdivide();

    std::vector<std::shared_ptr<Triangle> > childrenTriangles
        [8];
    for (const auto &triangle: triangles) {
        aabb triangleBoundingBox = triangle->bounding_box();
        for (int i = 0; i < 8; i++) {
            if (children[i]->bbox.intersect(triangleBoundingBox)
                ) {
                childrenTriangles[i].push_back(triangle);
            }
        }
    }

    // now recursively build the children
    for (int i = 0; i < 8; i++) {
        children[i]->build(childrenTriangles[i], depth + 1);
```

```
    }
}

void Octree::subdivide() {
    glm::vec3 center = bbox.center();
    glm::vec3 halfSize = bbox.half_size();

    children[0] = std::make_unique<Octree>(aabb(bbox.min(),
        center)); // Bottom-left-front
    children[1] = std::make_unique<Octree>(aabb(glm::vec3(center
        .x, bbox.min().y, bbox.min().z),
                                                glm::vec3(bbox.
                                                    max().x,
                                                    center.y,
                                                    center.z)));
                                                    // Bottom-
                                                    right-front
    children[2] = std::make_unique<Octree>(aabb(glm::vec3(center
        .x, bbox.min().y, center.z),
                                                glm::vec3(bbox.
                                                    max().x,
                                                    center.y,
                                                    bbox.max().z)
                                                    )); // Bottom
                                                    -right-back
    children[3] = std::make_unique<Octree>(aabb(glm::vec3(bbox.
        min().x, bbox.min().y, center.z),
                                                glm::vec3(center
                                                    .x, center.y,
                                                     bbox.max().z
                                                    ))); //
                                                    Bottom-left-
                                                    back
    children[4] = std::make_unique<Octree>(aabb(glm::vec3(bbox.
        min().x, center.y, bbox.min().z),
                                                glm::vec3(center
                                                    .x, bbox.max
                                                    ().y, center.
                                                    z))); // Top-
                                                    left-front
    children[5] = std::make_unique<Octree>(aabb(glm::vec3(center
        .x, center.y, bbox.min().z),
                                                glm::vec3(bbox.
                                                    max().x, bbox
                                                    .max().y,
                                                    center.z)));
                                                    // Top-right-
                                                    front
```

```
    children[6] = std::make_unique<Octree>(aabb(center, bbox.max
        ())); // Top-right-back
    children[7] = std::make_unique<Octree>(aabb(glm::vec3(bbox.
        min().x, center.y, center.z),
                                               glm::vec3(center
                                                   .x, bbox.max
                                                   ().y, bbox.
                                                   max().z)));
                                                   // Top-left-
                                                   back
}

bool Octree::hit(const Ray &r, float t_min, float t_max,
    hitrecord &rec) const {
    if (!bbox.hit(r, interval(t_min, t_max))) {
        return false;
    }

    bool hit_anything = false;
    float closest_so_far = t_max;

    // leaf node
    if (!children[0]) {
        for (const auto &triangle: triangles) {
            if (triangle->hit(r, t_min, closest_so_far, rec)) {
                hit_anything = true;
                closest_so_far = rec.t;
            }
        }
    } else {
        // internal node
        for (int i = 0; i < 8; i++) {
            if (children[i]->hit(r, t_min, closest_so_far, rec))
                {
                hit_anything = true;
                closest_so_far = rec.t;
            }
        }
    }

    return hit_anything;
}


static point2 get_sphere_uv(const point3 &p) {
    auto theta = acos(-p.y);
    auto phi = atan2(-p.z, p.x) + pi;

    return {phi / (2 * pi), theta / pi};
```

```
}
```

B Images

Figure B.1: St Lucy statue, rendered with appropriately 2 million triangles

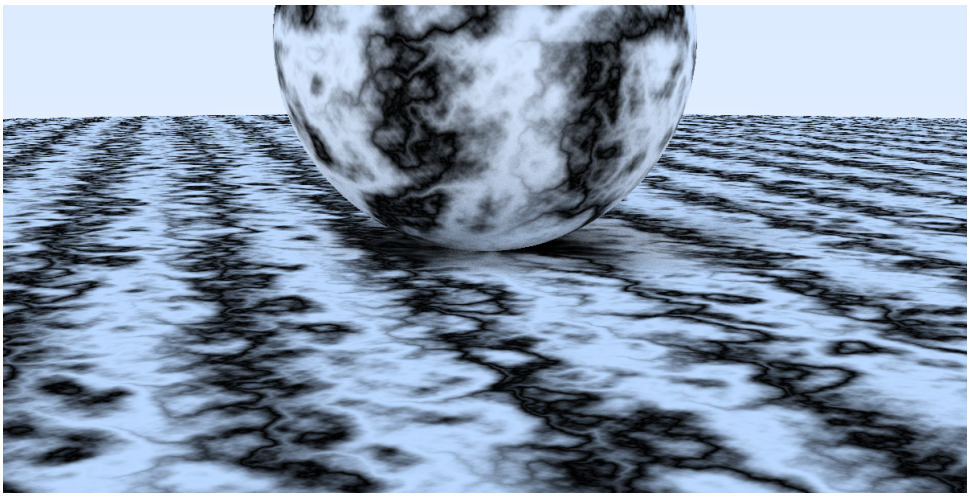

Figure B.2: Perlin Noise texture applied to the floor and the sphere

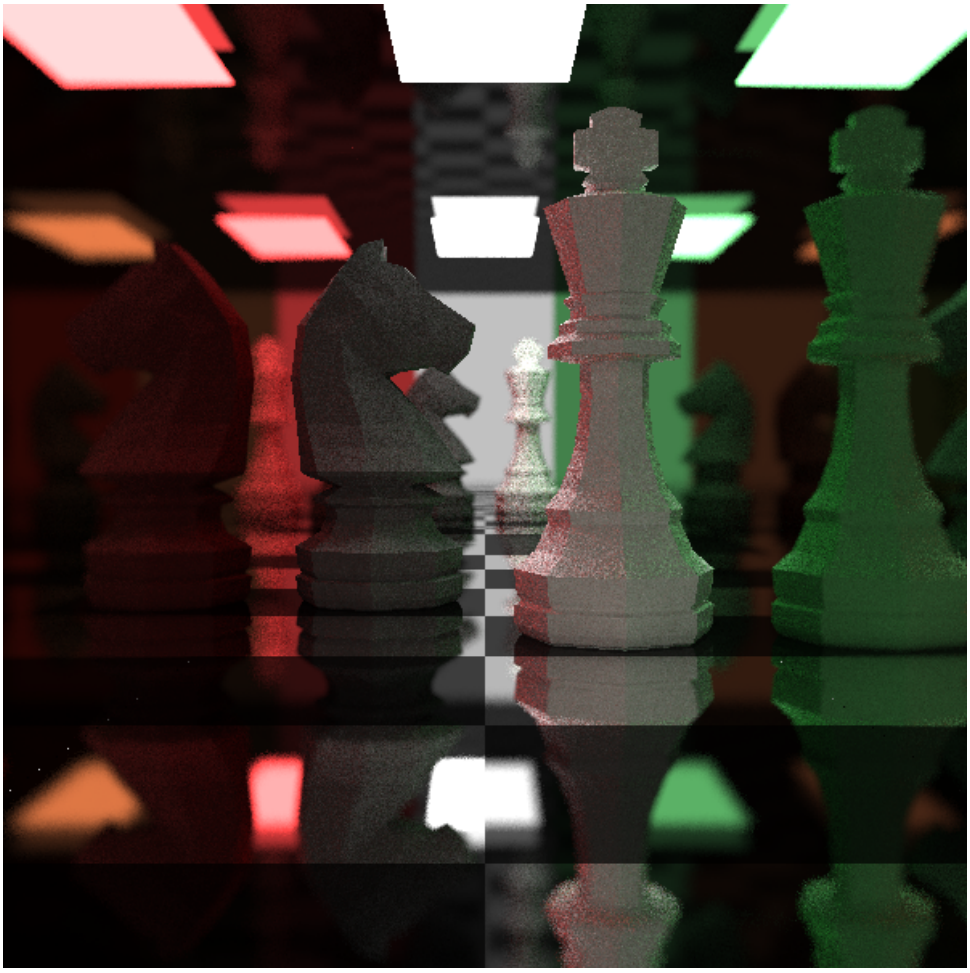

Figure B.3: Two chess pieces with reflective walls and mirrors

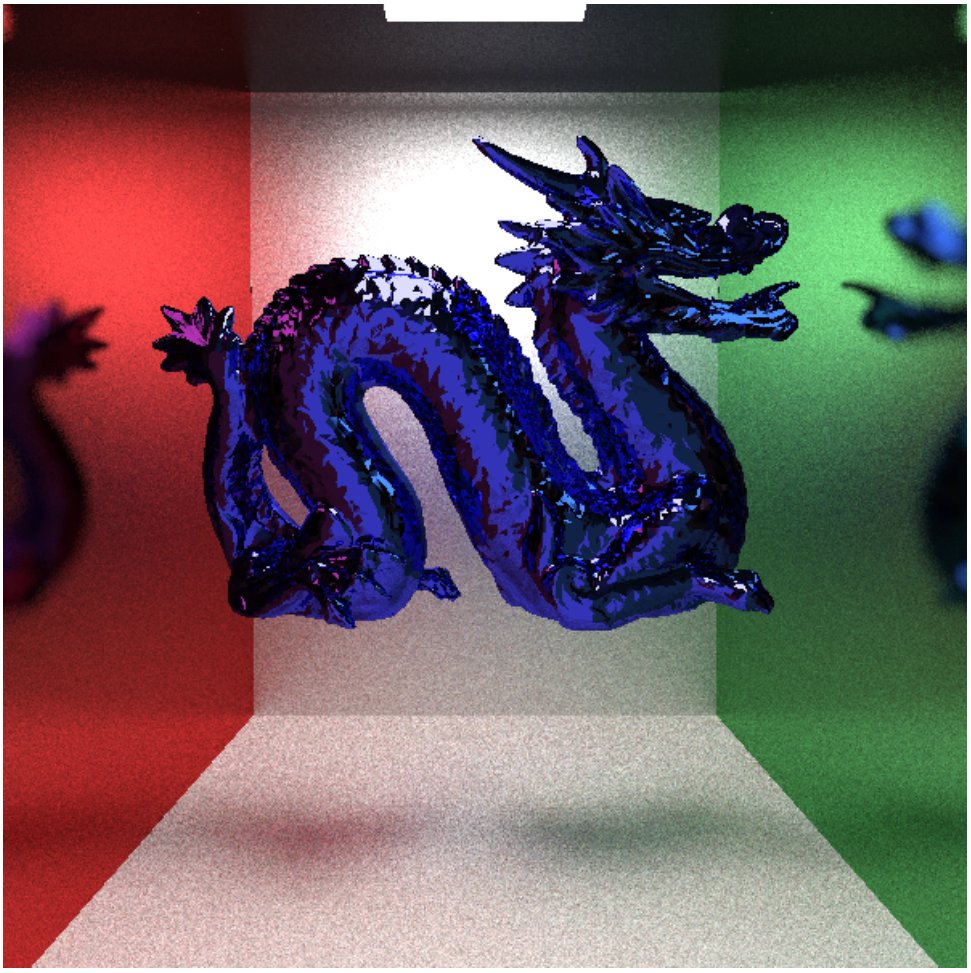

Figure B.4: A metallic dragon

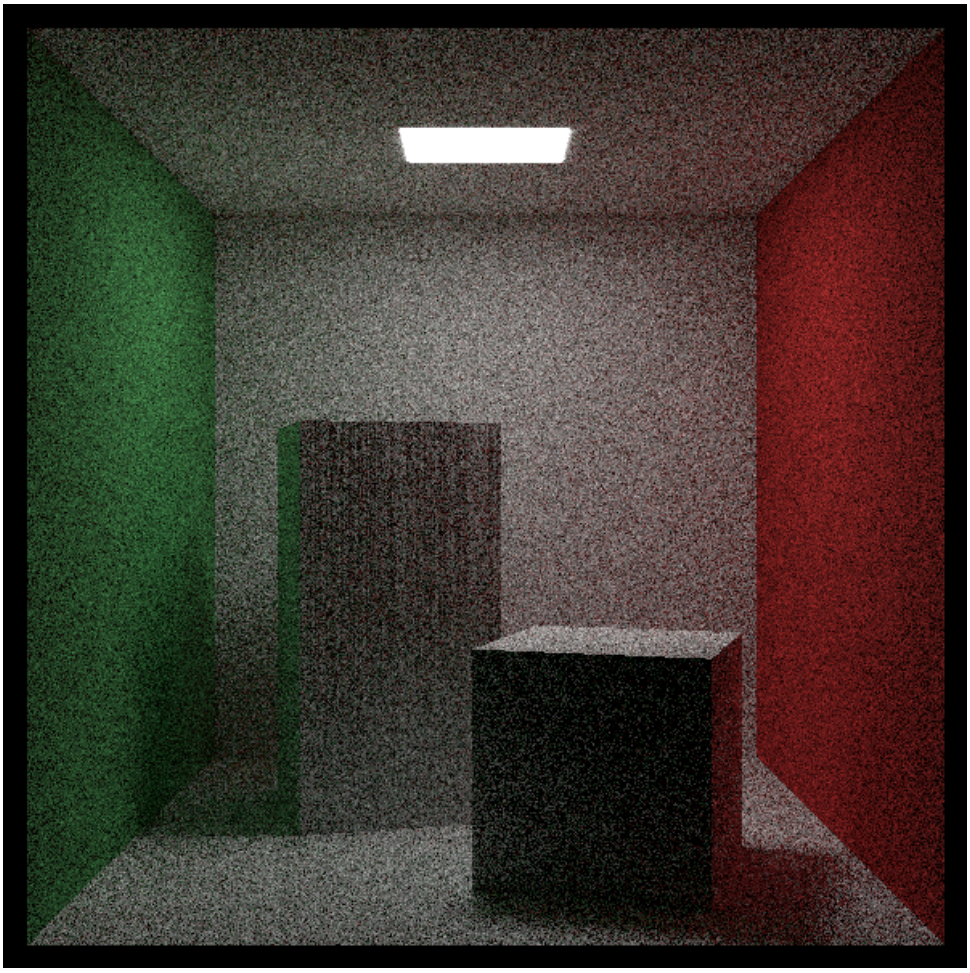

Figure B.5: A standard Cornell Box