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 Extented Project Qualification

St Bartholomew's School
United Kingdom
2024-03-10

Contents

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" rendering 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

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. It's 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

prevent efficient raytracing.

"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 the 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.

2.1.1 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. 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

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

This can be thought of as also asking, whether a given point is inside, outside, or within the sphere. For us, there are only two possiblities we care about. Whether the ray is a tangent, and 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 arbitary point in 3D space, then we get:

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

We can represent the point P, as a point on our ray, 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, which touches or intersects the sphere. Using the quadratic formula, we can solve for t:

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

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 appropiate section.

Once all of 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.

2.1.2 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].

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. 1, 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. *The International Journal of Robotics Research* **33,** 1393–1412. ISSN: 0278-3649. https://doi.org/10.1177/0278364914530482 (2024) (Sept. 1, 2014).
3. Pharr, M., Jakob, W. & Humphreys, G. *Physically Based Rendering: From Theory to Implementation (3rd Ed.)* 3rd. 1266 pp. 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. *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. *The Guardian. Film.* ISSN: 0261-3077. https://www.theguardian.com/film/2021/jul/03/terminator-2-at-30-groundbreaking-sequel-that-led-to-cgi-laziness (2024) (July 3, 2021).
6. Glassner, A. S. *An Introduction to Ray Tracing* 364 pp. ISBN: 978-0-12-286160-4. Google Books: YPblYyLqBM4C (Morgan Kaufmann, Jan. 28, 1989).
7. Higham, N. J. *Accuracy and Stability of Numerical Algorithms* 2nd ed. 680 pp. ISBN: 978-0-89871-521-7 (Society for Industrial and Applied Mathematics, Philadelphia, 2002).
8. *The AP Professional Graphics CD-ROM* (eds (Firm), A. P. & (Firm), E. E. I.) 1 p. ISBN: 978-0-12-059756-7 (AP Professional, Chestnut Hill, Mass., 1995).
9. Woop, S., Benthin, C. & Wald, I. Watertight Ray/Triangle Intersection. **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,** 171–259. ISSN: 0002-9556. JSTOR: 1422442. https://www.jstor.org/stable/1422442 (2024) (1985).