Master Project
Computer Graphics: Rendering Track

Alhajras Algdairy
Advisor: Prof. Dr.-Ing. Matthias Teschner

September 4, 2021

Acknowledgment

I want to express my special thanks of gratitude to Prof. Dr.-Ing. Matthias Teschner, who guided me through my master's program in general and in this master's project in specific. This project gave me the golden opportunity to dig into an exciting topic in computer graphics that I am keen on. The flexibility in research made me learn more about the related topics I am interested in, in rendering. Reading scientific topics and reviewing codes from different repositories expanded my expertise, where I had a full responsibility to control my time and resources, to learn more about the research process.

Abstract

This report investigates four different acceleration data-structure methods for implementing a simple raytracer on CPU. These methods are KD-tree, Uniformgrid, BVH, and LBVH. The report aims to compare the different approaches and their impact on the raytracer rendering time. The raytracer used in this project builds upon a previous lab project where the foundation of the raytracer is already implemented; however, without using a sophisticated data-structure to enhance the rending performance; hence the focus of this report will be how to improve the raytracer performance by using data-structures. ***Keywords: Ray Tracing; kD-Tree; BVH; LBVH; Uniformgird; Data structure;***

1 Introduction

This report summarises my journey to implement a simple raytracer focusing on data structure level, where the performance of rending a scene will be the lion's share of the report. Theory, implementation, and results will be discussed in depth in the report, where an introduction and motivation of what is raytracer and how to implement it will be briefly discussed; because data structures are more interesting for us, I will only explain the topics that are used in this raytracer.

2 What is Raytrcer and how it works

Computer graphics has three main pillars: *Modelling*, *Rendering*, and *Simulation*. Scientists are interested in simulating a real-life phenomenon, such as Gravity Fraction, Rain, Snow, and the exciting part for us, light. Simulating light is arguably the most challenging part because light has always been difficult to characterize as it can behave as particles and waves, which

makes it spread into the whole scene based on probabilities. This makes it difficult to compute as it involves complex computation and infinite simulations to execute to have a perfect result without an error, this is what is known as Rendering.

Why do we need to simulate light? Adding light into a scene will generate shadows, reflection, and refraction consequently will illuminate the scene, making the scene look like a reality, which can be helpful for some applications. For the internal designers, it is vital to simulate the final result of the lightning inside a room, such as sunlight coming from the windows, the light of lamps, and fireplace, these with react together and create shadows, before constructing the building and payloads of money, simulating the right angle and place of the room is helpful to imagine the final result. Solar engineers use 3D tools to build solar panel farms where the angle between the sunlight and the panel is essential to gather as much light energy as possible. In gaming, different companies compete to design engines that can produce natural scenes; this includes lighting and shadowing.

Figure 2, shows an example of rending a scene that can not be distinguished from reality, the details it catches as glossy materials, the reflection of surfaces, and the shadow. Capturing these details requires rendering techniques. Two popular methods are Rasterization and Raytracing. Raytracing outperforms rasterization in capturing more details; however this leads to performance issues, that is why most application uses Raytracing is offline rendering and not real-time rendering like games, where speed is vital to render each frame with compromising details.

Figure 1: https://www.pcgamer.com/unreal-engine-5-tech-demo-pc-performance/ The raytracing algorithm builds an image by extending rays into a scene and bouncing them off surfaces and towards sources of light to approximate the color value of pixels [Piotr Dubla, "Interactive Global Illumination on the CPU."]

2.1 Raytracing definition

So what is Raytracing? Raytracing is a rendering technique that provides highly lifelike lighting effects. In other words, an algorithm can track the source of light and then mimic how

the light interacts with the virtual objects it eventually encounters in the computer-generated environment. Raytracing produces far more lifelike shadows and reflections, as well as significantly enhanced translucence and dispersion. The algorithm considers where the light falls and calculates the interaction and interplay in the same way as the human eye does with actual light, shadows, and reflections.

2.2 Raytracing mechanism

Raytracing follows three main steps: *Casting Rays*: This is similar to how the eye works, however rather than eyes works as a sensor, we have a camera, we shoot rays from the camera to the scene, and we calculate which is the closest object it hits, its color and brightness. This is done for each pixel. *Path tracing*: Casting rays can solve the visibility issue; this means which object appears on the camera and at what position and color. However, if we want to simulate the following effects: soft shadows, depth of field, motion blur, caustics, ambient occlusion, and indirect lighting, then we need to trace the rays from or to the light source, this will accumulate the brightness of an object or model in a scene, also will give the depth of different objects. Note that the more rays and depth we need, the more quality we get but the slow the simulation becomes due to the complexity. *Shading*: In addition to tracing rays, we need a model to simulate different materials like Transparent, Glossy, and Diffuse. This is where the Shading phase is needed.

Raytracing has two main methods:

- *Forward Raytracing*: The light particles (photons) are tracked from the light source to the object via Forward Raytracing. While forward ray tracing is the most exact method for determining the color of each object, it is also the most inefficient.
- *Backward Raytracing*: An eye ray is formed at the eye in backward ray tracing, and it travels through the viewplane and out into the scene. If it hits an object, it will return it to the viewplane immediately. This method is more efficient than Forward Raytracing but less accurate due to reducing the rays used. In this implementation this method is used.

3 Raytracer Implementation

In this chapter, raytracer terms and definitions are introduced, illustrating the implementation and decisions that have been made. We need to set up our implementation to work for the raytracer as shown in Figure 2; the basic raytracer requires a camera; its responsibility is to shoot rays that travel through the scene and return a value or color. How many rays should it shoot? This depends on the width and height of the image we want to generate, the bigger the image, the more details we capture, but the more expensive computation gets. The rays idea is to find an intersection with the scene objects and try to return a color of the object to represent a corresponding pixel value. For object visibilities, we do not need a light source; however, to make the scene more realistic, we need to add shadows and other effects as reflections and refractions; hence we need a source light and tools, as shown in Figure 2.

The algorithm works as follows, initiating the camera position and the orientation direction or where it looks at. Secondly, we subdivide the scene into pixels; this is based on the settings of the raytracer, usually by setting a height and width, reading a scene from usually an XML file that has a description of the number of objects in the scene, their color, and type. Afterward, generating rays for each pixel, this ray will be shooted toward the scene, and for each ray, we go

through all the objects and test intersection tests; if the test returns true, we save its position, and we save the object, we do this for all objects and if more than two objects are intersected we compare between their position to detect which one is closer to the camera. In order to create shadows, we track the intersection point from the camera toward the light; if there is an object between the light source the hitting point, then we detect shadow. The corresponding value of the pixel is saved in a buffer and after the ray tracing is done, the pixels value are saved in a file of PPM extention.

This is the basic idea, and to improve it, other topics need to be covered, such as Data structures, Objects Materials, Anti aliasing, Soft shadow. In this chapter, the basic blocks and components that make up the raytracer will be explained; only the methods are used will be explained, because as mentioned before, the performance of the raytracer is the main scope for this project.

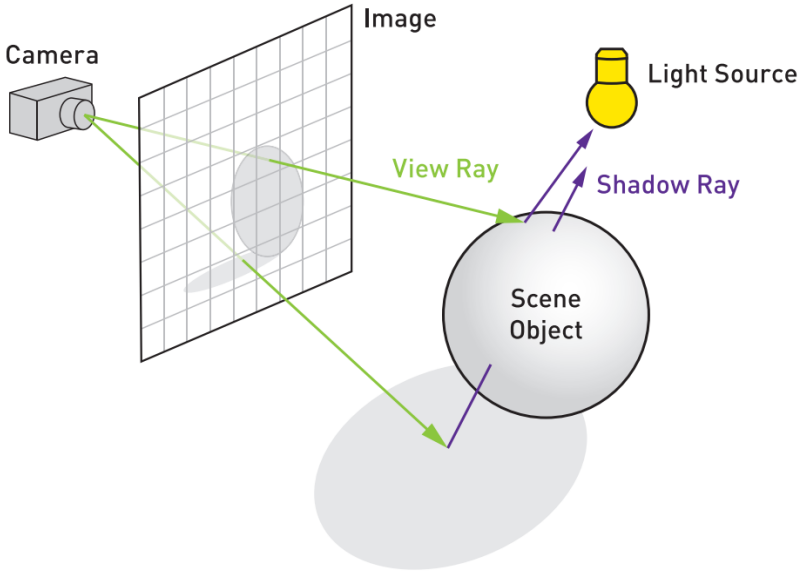


Figure 2: An image showing how a ray is casted from the camera and how objects are illuminated and shadows are computed in ray tracing .

3.1 Software architecture

Configuration

We start with the configuration of the Raytracer by introducing a **struct Settings**, here we specify the following properties of the raytracer such as the **width** of the image, **height** of the image, the **background default color** of the scene, **maximum depth of path tracing**, **anti-aliasing samples** and **acceleration data Structure type**. Some of the previous terms will be explained later.

Scene generation

To create a scene, spheres will be used to create meshes; mesh is a collection of objects at a position and orientation to build a much more complex object. For example, the Stanford bunny shown in Figure 3 is composed of approximately 35947 spheres. Usually, meshes are made of triangles because they are more efficient and easier to build a mesh; however, I want

to try to use something else and test the result and the quality of spheres. **createScene()** is implemented to load .obj files the next models will be used for testing in this report: **Stanford Bunny**, **Igea** and **Armadillo**, Figure 4 shows the different complex model that will be used for testing.

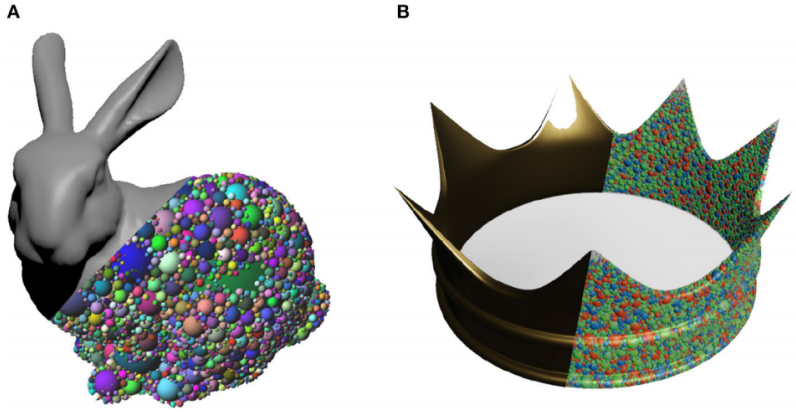
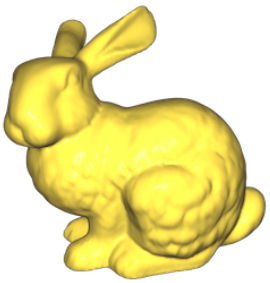


Figure 3: Stanford bunny and a crown made of a collection od spheres .

(a) Armadillo model

(b) Igea model



(c) Stanford-bunny model

Figure 4: Rendering different shapes in the scene

The **Sphere** is a class with the next attributes: **Center**: Posiyion of the sphere, **MaterialType**: *diffuse*, *glossy*, *reflection* and *refraction* ,**Radius**: Sphere radius usually 1, **surfaceColor**: Color of the sphere in rgb format, and **emissionColor**: This is for the light. Lights are just vector of spheres where the **emissionColor** is set to one.

4 Object subdivision

5 Spatial Subdivision

This lab was a great way to gain practical experience of the concepts we learned in our computer graphics course. I would like to thank **Prof. Dr.-Ing. Matthias Teschner** again for advising me throughout this lab. As a final remark, I would like to mention that I really enjoyed working on area lights. This includes debugging issues as I was working on them as well as how they were fixed. At the end, the results were worth the effort.

References