Final Report of Advanced Computer Graphics

Lu Jinfan
lujf22@mails.tsinghua.edu.cn

ACM Reference Format:
Lu Jinfan. 2024. Final Report of Advanced Computer Graphics. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In the Advanced Computer Graphics project, I developed a CPU-based software renderer. This project is centered on image rendering, with the architecture of the renderer taking inspiration from the PBRT (Physically Based Rendering: From Theory to Implementation, Version 4) framework.

The project entailed a thorough study and implementation of algorithms from PBRT. My objective was to emulate its architecture and craft a distinct, physically accurate rendering engine. I streamlined certain non-critical functions for efficiency and introduced additional features to enhance specific areas.

The code for this renderer is available in my GitHub repository at https://github.com/LJFYC007/Renderer.

2 METHOD

2.1 Model Loading

In the model loading, I utilized a third-party C++ library named `tiny_gltf`, which facilitated the loading of glTF2.0 files. Utilizing the structured recursive traversal provided by `tiny_gltf`, I processed all objects, constructed their materials, and integrated them into the scene. Following object processing, I invoked the BVH(bounding volume hierarchy) constructor to establish an efficient spatial acceleration structure, which significantly improved rendering performance.

To optimize memory usage in my renderer, I implemented a strategy to ensure that recurrent resources such as images, materials, and geometric shapes are loaded into memory only once, regardless of how many times they are used. This approach effectively reduces memory consumption by eliminating redundant storage of data. The source code for this process can be found in `model.h`.

2.2 Basic Path Tracing

In the rendering part, I implemented a comprehensive path tracing algorithm for global illumination based on the rendering equation:

$$L_o(p, \omega_o) = L_e(p, \omega_o) + \int_{\Omega^+} L_i(p, \omega_i) f_r(p, \omega_i, \omega_o)(\omega_i \cdot n) d\omega_i$$

Within my architecture, I first compute the $L_e(p, \omega_o)$ term, representing the emission from light sources. There are two scenarios considered: one where the light source is at infinity (which includes both infinite sunlight and HDR environmental textures in my architecture), and another where the object itself is an area light. The integral is evaluated using the Monte Carlo method, and the $f_r(p, \omega_i, \omega_o)$ term is computed using the principled BSDF (Bidirectional Scattering Distribution Function) model. The full implementation can be found in `integrator.h`.

2.3 Multiple Importance Sampling

In this section, I detail the multiple importance sampling (MIS) techniques employed in my rendering algorithm, particularly in the context of evaluating the following integral:

To approximate the integral of a function $f(x)$, we can use a probability density function $p(x)$ that approximates $f(x)$ to randomly choose n i.i.d. samples $x_1, \cdots, x_n$:

$$\mathbb{E}[f(x)] \approx \frac{1}{n} \sum_{i=1}^{n} \frac{f(x_i)}{p(x_i)}.$$

Extending this to the case where we have m different probability density functions p_i, from each of which we draw n_i samples $x_{i,j}$, and ensuring that the weight functions $w_i(x)$ satisfy the partition of unity $\sum_{i=1}^{m} w_i(x) = 1$ for all x where $f(x) \neq 0$, and $w_i(x) = 0$ whenever $p_i(x) = 0$, the MIS estimator can be formulated as:

$$\mathbb{E}[f(x)] \approx \sum_{i=1}^{m} \frac{1}{n_i} \sum_{j=1}^{n_i} w_i(x_{i,j}) \frac{f(x_{i,j})}{p_i(x_{i,j})}.$$

For the weighting functions $w_i(x)$, I utilize the power heuristic with $\beta = 2$, which is defined as:

$$w_i(x) = \frac{(n_i p_i(x))^2}{\sum_{k=1}^{m} (n_k p_k(x))^2}.$$

This choice of β has been shown to work well in practice, balancing variance reduction and robustness against the selection of suboptimal sampling strategies.

2.4 Acceleration Structure

In order to efficiently compute the rendering equation, it is crucial to rapidly determine the first intersection of a ray with objects in the scene. This intersection test is often the primary computational bottleneck in a path tracing renderer. To address this, I have implemented an optimized BVH(Bounding Volume Hierarchy) that leverages the SAH(Surface Area Heuristic) for spatial partitioning.

Empirically, after using SAH instead of the middle split method, the rendering speed is increased by 2 times. The complete implementation can be found in `bvh.h`.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2.5 Principled BSDF

The BSDF(Bidirectional Scattering Distribution Function) characterizes how light is reflected and transmitted at a surface, encompassing a variety of material properties. Following the approach outlined in PBRT, materials are commonly classified into three categories: diffuse, conductor, and dielectric, each with distinct optical characteristics.

For diffuse objects, we sample a random direction over the hemisphere oriented around the surface's positive normal, weighted by $\cos \omega$, where ω is the angle between the sampled direction and the surface normal. For conductor and dielectric objects, I adhere to the PBRT implementation that employs Microfacet Theory, utilizing the Torrance-Sparrow Model and Fresnel equations for specular reflection and transmission. The complete implementation is available in `bxdfs.h`

2.6 Texture

Textures play a crucial role in adding realism to rendered objects by simulating material details without the need for geometric complexity. Normal mapping is a technique in texture mapping that allows for the simulation of intricate surface details by altering the surface normals of an object. This method can create the illusion of depth and texture on otherwise flat surfaces.

The comparison images below showcase the difference normal mapping makes. To appreciate the finer details, it may be necessary to zoom in on the images. The effect of normal mapping is most apparent in the way light interacts with the surface. With normal mapping applied, the bumpy or textured parts of the object exhibit more dynamic and realistic reflections, as the perturbed normals affect the light's reflection angles.

The implementation of normal mapping and other texturing effects can be found in the source file `textures.h`. This file contains the code essential for applying texture maps to 3D models, which in turn significantly contributes to the visual complexity and realism of the rendered scenes.

2.7 Anti-aliasing

Since in path tracing, each pixel is sampled multiple times with slightly different ray origins or directions. This stochastic sampling inherently provides a form of anti-aliasing known as stochastic or distributed anti-aliasing.

As a result, the need for additional anti-aliasing algorithms, such as supersampling or multisample anti-aliasing (MSAA), is eliminated, simplifying the rendering pipeline while still achieving high-quality results.

2.8 Double Error Handling

When dealing with floating-point arithmetic, we need to estimate an upper bound on the error of a floating-point number, so that when a new ray is projected, we can use this upper bound on the error to get the new starting point of the ray, without having to set a hyperparameter to control it. Therefore, I have implemented a robust error handling tool, modeled after the implementation in PBRT, to estimate this error reasonably well during triangle intersection operations. The specific code can be found in `shape.h`.

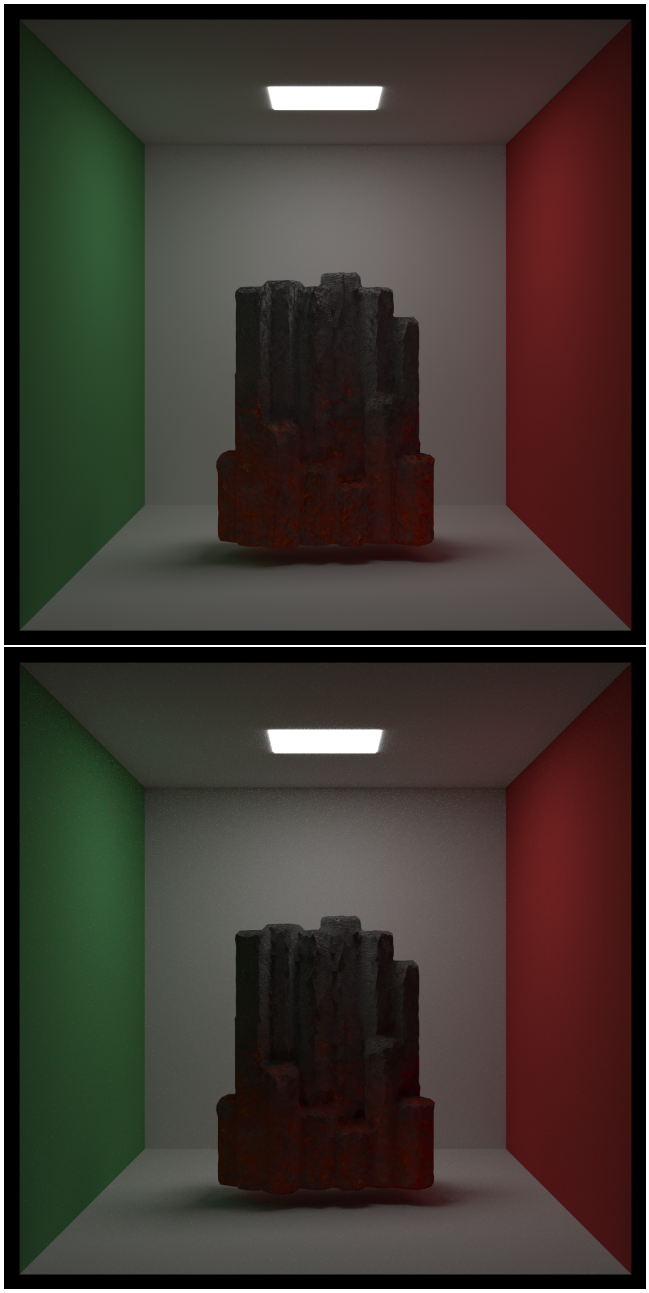


Figure 1: With or Without Normal Mapping(the image above contains normal mapping)

2.9 Depth of Field and Motion Blur

For Depth of Field, the camera is equipped with a customizable aperture that allows for control over the defocus angle. By varying the size of the aperture, the depth of field can be adjusted, altering the range of distances within which objects appear sharp. A larger aperture results in a shallower depth of field, creating a more pronounced bokeh effect. In contrast, a smaller aperture increases the depth of field, bringing more of the scene into focus.

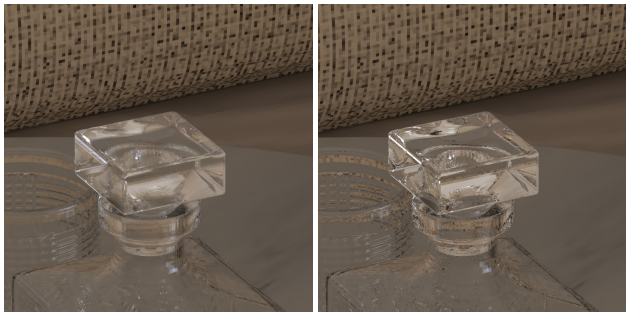
Figure 4: Final Scene

The implementation of Motion Blur simulates the camera itself moving, during the exposure period of each frame.

(a) Defocus 0.1 **(b) Defocus 0.2** **(c) Motion blur**

2.10 Color

I used spectral-based colors rather than RGB, and spectral rendering is important not only in that it provides us with a physically accurate representation of color, but also in that it simulates the effects of dispersion in optical phenomena. This is particularly evident in the enlarged image of the glass bottle cap. The dispersion effect can be seen in the image, where light passing through the glass bottle cap is refracted to different degrees due to different wavelengths, resulting in the visual phenomenon of color separation. This effect cannot be directly realized by the RGB model.



(a) With Spectral Rendering **(b) Without Spectral Rendering**

In order to accurately achieve this effect, my workflow uses the CIE-XYZ color space as an intermediate color gamut, which provides a linear wide color gamut. In the process of converting RGB to spectral, which requires some complex numerical integration, since I don't think this part is the focus of this big assignment, I used part of the PBRT code directly. This code can be found in `color.cpp`. Other code for the color transform is written by myself, which can be found in `color.h, colorspace.h, spectrum.h`.

3 RESULT

Here is the final scene.

All of the models were sourced from Taobao, but after many modifications and reconstructions by myself, the final scene's glTF file can be found in `final-scene.glb`.

4 EXTERNAL TOOL

In the project, I use two C++ third-party libraries, namely `tiny_gltf` and `stb_image`. Also, the file `color.cpp` directly comes from PBRT, which mentioned in the Color part. The other code is pure C++.

It's worth noting that since I don't like columns to be reversed in `glm`, I've implemented all the math classes myself.