Implementation of Monte Carlo path tracing algorithm based on OpenGL

A Dissertation

Presented to the University of Dublin, Trinity College

in partial fulfilment of the requirements for the degree of

Master of Science in Computer Science (Augmented and Virtual Reality)

Supervisor: Michael Manzke

August 2022

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

August 19, 2022

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

August 19, 2022

Acknowledgments

I would like to thank my mom, my dad. They has been supporting me all the time, without their help I would not have the opportunity to come here to study.

I would also like to thank Professor Michael Manzke for inspiring me with his Realtime Rendering course, which laid the foundation for my subsequent research. He is also thanked for his guidance and supervision of the entire project. I would also like to thank Professor Fergal Shevlin. His mathematical guidance was very important to me.

Finally, I would like to thank my friends from within the college as well as outside. Thank you for the laughter, guidance, and lessons they have given me throughout the year.

University of Dublin, Trinity College
August 2022

Implementation of Monte Carlo path tracing algorithm based on OpenGL

, Master of Science in Computer Science

University of Dublin, Trinity College, 2022

Supervisor: Michael Manzke

As the performance of graphics cards continues to improve, Ray tracing algorithms are getting faster to compute. It has been widely used in movies and games for its more realistic experience than rasterization. Using publicly available tools, this paper implements rendering of virtual scenes based on Monte Carlo path-tracing algorithm. Make it close to the lighting effect of the real scene. This paper also studies ray-tracing acceleration algorithm and OpenGL data transmission between CPU and GPU.

Summary

The paper is laid out as follows: Firstly, The research background is discussed. Topics will include applications of ray tracing, technical advantages and disadvantages of ray tracing, existing algorithmic research on ray tracing, and a literature review of algorithms involved in ray tracing. This is followed by a design and introduction of the flow of the algorithm used in this paper, followed by a detailed explanation of the actual implementation. The actual operation results are then evaluated and discussed. Finally, the conclusion is given, and the limitations of this method and the need for further work are pointed out.

Contents

List of Figures

Chapter 1

Introduction

Ray tracing was used in the early days of computer graphics for movies and television shows. But ray tracing requires a lot of graphical computing power, so it requires the power of an entire server farm (or cloud computing). Since NVIDIA released the RTX20 series graphics card, it is also the first time that ray tracing technology appears on the graphics card, so that ordinary users, especially game users, can feel the changes brought by ray tracing technology for the image.

Ray tracing technology, using algorithms to simulate the physical characteristics of light in the real world. It can achieve physical accurate shadow, reflection, refraction and global illumination.[1] In other words, in a virtual scene, it can make the objects in the scene more realistic. It gives games movie-quality graphics, such as the fire, smoke and explosions of war movies, and makes them feel like they're there. General ray tracing algorithms have two major shortcomings. One is that the surface attributes are relatively single, so it is difficult to enrich the various optical effects that occur after light touches the surface of the object. The other is that diffuse reflection is ignored.Monte Carlo path tracing is an improvement of the traditional ray tracing algorithm.

The goal of this project is to use OpenGL and GLSL to build a basic Monte Carlo path tracing model from the bottom layer, so that it can correctly simulate the optical effect in the closed scene, and make the rendering result close to the real scene. The project also studied the OpenGL data transfer method directly between CPU and GPU, BVH

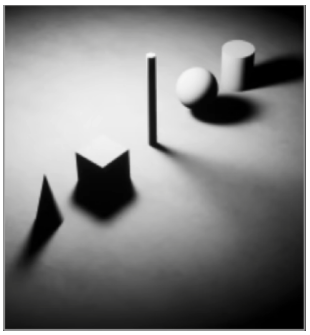
acceleration algorithm and the de-noising of monte Carlo path tracking algorithm results. The results of this project will build a model for pre-processing data in the CPU and performing ray-tracing calculations in the GPU. There are many supported ray tracing rendering tools that can achieve realistic rendering effects such as V-Ray, OctaneRender etc. but using them directly cannot understand the whole process and essence of ray tracing. This project is helpful for the in-depth understanding of Monte Carlo path tracing algorithm and the subsequent improvement of this type of path tracing model.In the future We can take this offline ray tracing rendering model as the basis, combine rasterization technology and simplify the ray tracing algorithm, and further implement real-time ray tracing technology, which is a very popular technology in the game industry.

Chapter 2

Background

2.1 Global Illumination

The real world light is in a linear space, and the lighting effects can be superimposed. The final lighting result equal to direct illumination plus indirect illumination, and the result is also known as global illumination. This is contrasted with local illumination (only direct illumination is considered). Reflection, refraction, and shadow belong to the category of global illumination because they must be simulated not only by considering the direct effect of the light source on the object but also by considering the interaction between objects. However, specular reflection and refraction generally do not need to solve the complex illumination equation, and do not need to carry out iterative calculation. As a result, these parts of the algorithm are already very efficient, even real-time[2]. Different from the specular reflection, the direction of the diffuse reflection surface is approximately "random", so the reflection result cannot be obtained by simple ray tracing, and often needs to be iterated by multiple methods until the distribution of light energy reaches a basic equilibrium state.



(a) soft shadows

(b) reflection and specular

(c) global illumination

Figure 2.1: Some advanced ray traced effects(From LingQi Yan, UC Santa Barbara)

Depending on the hardware features, there are two ways to achieve global illumination:Ray tracing and rasterization. After decades of development, global illumination(GI) has been implemented in many directions, the book "Advanced Global Illumination" by Dutre et al.[3] Offers analysis for most of the offline global illumination techniques such as Ray tracing, Path tracing, Photon mapping, point based Global illumination, Ambient occlusion, Metropolis light transport, light Propagation Volumes Global illumination etc. Each of these techniques can be divided into different kinds of improved and derived algorithms. The path tracing algorithm used in this project is a faction based on ray tracing combined with Monte Carlo algorithm. Unlike biased calculation methods such as ambient light masking and photon mapping, Monticaro path tracing can converge to the correct result by averaging over infinitely many renderings of the same scene.

2.2 Development of GI(Based on ray tracing and path tracing)

The origin of ray-tracing algorithm can be traced back to the concept of ray casting proposed by Arthur Appel in 1968. In his paper[4], ray casting means from our eyes or camera to each pixel of the projection imaging screen out a light (here we only consider our eyes or cameras is a point), this light will be hit at a certain position in the scene. If the light hits an object in the scene, then draw a line between the intersection point and the light source to determine whether the point is visible to the light source.We can construct an efficient light path in this way, and then calculate the energy of that light path to figure out the final color of the pixel.

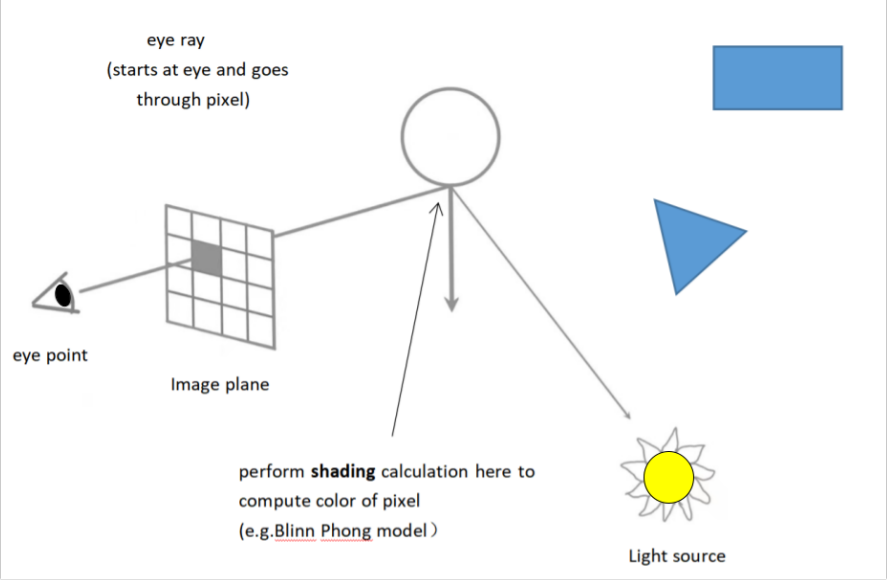


Figure 2.2: ray cast

Appel's algorithm uses View Ray and Shadow Ray, which is actually the direct lighting part of the lighting equation[5]. A significant advantage of ray casting over traditional scan line rendering algorithms is that it can deal with uneven surfaces and solids. Most of the animations for the Tron Series (1982) were rendered using ray casting[6].

In 1979, Turner Whitted added the interaction between light and surfaces to ray casting, extending the process by introducing reflection, refraction and shadow. This algorithm is called Recursive Ray Tracing (or Whitted-style Ray Tracing)[7], so that there is no longer a single Ray, but a path of light transmission.Whitted-Style Ray Tracing said when light hits an object, both reflection and refraction occur at the intersection point (assuming Specular reflection occurs on a smooth surface), and both refracted and reflected light continue to travel. Due to the many times of light bouncing, the algorithm calculates the color of the light source at each intersection of refraction or reflection, and finally obtains the final color recursively.Whitted ray tracing mainly solves the problem that there is no indirect light in the scene, but in Whitted's model, all indirect light only comes from perfect specular reflection or refraction. It assumes that the surface of the object is absolutely smooth, which is obviously inconsistent with the surface properties of most substances in nature.

Figure 2.3: Spheres and checkboard, T.Whitted, 1979

Robert L. Cook proposed Distributed Ray Tracing[8] in 1984, which turned a single beam of reflected light into an integral calculation around the range of diffuse or highlight reflection in a space. Cook's method is also known as stochastic ray tracing because the Monte Carlo method was introduced to compute integral equations. Cook's model is computationally expensive. Each ray from the camera is reflected in several

different directions at the surface point, dispersing into multiple rays, and recursively, each ray eventually forms a tree of rays, especially for indirect diffuse light, which reflects almost the entire visible space.

James T. Kajiya unified rendering equation in 1986[9], and deduced the formula of light path expression form, makes the rendering equation by a recursive structure, becomes a path function integral, each of the Monte Carlo random number just produce a path. The path does not need to be recursive, each path can be randomly generated as a result. The value of each Path is then used as a random number to calculate the final rendering result, in a new form called Path tracing.

Bi-directional path tracing was proposed by Eric P. Lafortune and Yves D. Willems in 1993[10]. It starts from the two directions of light source and camera respectively, and after passing a certain path respectively, connects the ends of the two paths to form a complete path. This greatly increases the effective contribution of the light source. Veach describes bidirectional path tracing in detail[11].

Although ray-tracing has been explored as a research topic since the 1980s, its progress has been slow due to the lack of hardware computing power at the time(Rendering figure2.4 cost 74 minutes at that time). Until 2018, it was considered the first year of an era of ray-tracing. Ray-tracing was introduced to the public in GDC 2018. NVIDIA, ILMxLAB, and UE4 have released a Star Wars short film based on real-time ray-tracing with movie-quality visual effects. NVIDIA released the RTX Tehnology Demo and Project Sol Cinematic Demo Part 1. EA SEED team brought PICA real-time ray tracing Demo; Remedy's Northlight engine brings the Ray Tracing in North Light Demo; The Futuremark team released the DirectX Ray tracing Tech Demo.

Figure 2.4: NVIDIA RTX Ray-tracing technology, Star Wars Demo

2.3 Ray Tracing VS Rasterization

As the most traditional rendering method, rasterization has always been the most important and also the most original rendering method. Now it is still the main rendering method for 3D games due to its fast rendering speed.[12] Although many algorithms optimize the overall effect of rendering on the basis of rasterization such as using geometry shader render shadow[13], using CUDA Rasterizer to do Multi-resolution shadow mapping[14]. However, the principle of rasterization rendering is to project vector graphics onto the screen through various transformations and then pixelate (sample), which can not reflect the physical properties of the rendered objects. Therefore, rasterization rendering method can not accurately describe the shadow of objects, light reflection, refraction and other phenomena, and the realism of the rendered image is still unsatisfactory in some aspects. In particular, some global effects, such as soft Shadows, Glossy Reflection, and Indirect illumination, are not well handled, especially when the light is bounced more than once in the scene.

(a) Ray tracing on (b) Ray tracing off

Figure 2.5: "Shadow of the Tomb Raider". Shadow contrast after ray tracing mode is turned on

As a special rendering algorithm in 3D computer graphics, Raytracing is used to simulate the way of light in the real world in physics-based rendering. At present, it has been widely used in animation, games and other fields, e.g. early offline game graphic rendering Myst and Riven[15], ray traced version of animated Quake 4[16]. As a physically-based rendering method, The result of raytracing is very close to the real world. It handles global effects perfectly, not just the global shading part, but also all possible shadows. Because it is a good simulation of light in the process of reflection, refraction, scattering and other phenomena, so that the material properties of the object get a good performance. However, due to the recursive algorithm required to simulate the light propagation process, the rendering speed is far less than rasterization.In recent years, with the joint efforts of industry and academia, real-time ray tracing has been realized through the continuous approximation and simplification of ray algorithm, as well as the improvement of computer hardware level, and has been well applied in 3D games. Jacco Bikker said[17] "Ray tracing promises an elegant and fascinating alternative to z-buffer approaches, as well as more intuitive graphics and games development."

2.4 Uses of Global Illumination

More and more commercial game engines offer specialized global illumination technology implementations to support different levels of real-time photorealism at different performance costs. With scene precomputation, Unity 3D game engine provide offline implementations for different use cases.[18] Unreal Engine 4 implements a monolithic illumination system based on its own surface caching technology "Lightmass Global Illumination".[19] CryEngine provide precomputing system based on user requirements, what is more, it has full support for real time global illumination.[20]

Since Nvidia released its RTX2000 series graphic card in 2018, the use of ray-tracing in games has grown. Many 3A games supported ray-tracing now, such as Cyberpunk 2077, Dead Stranded, Red Dead Redemption 2, etc[21]. Even more impressive than the growing number of games joining the ray-tracing camp is the adoption of ray-tracing hardware. According to Steam data, 30 percents of graphics cards supported ray-tracing as of July 2022. That's more than double from 14 percent in January 2021.

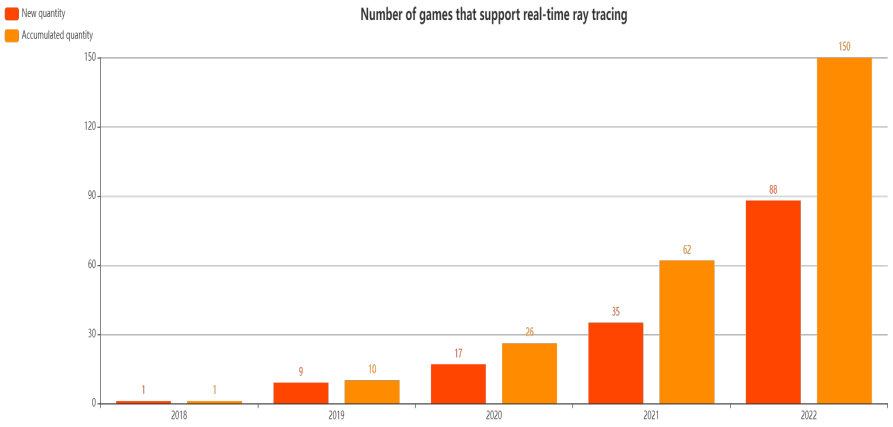


Figure 2.6: So far in 2018, according to Steam data and NVIDIA Game News Stats

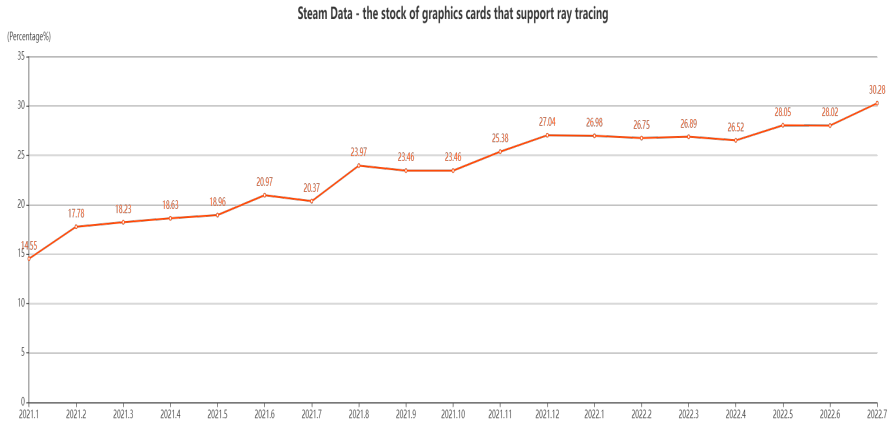
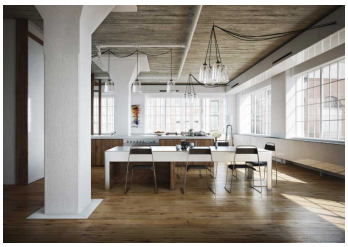


Figure 2.7: Until July 2022, the stock of graphics cards that support ray tracing, according to Steam data.

Global illumination has also been used in a number of films. The first movie to use global lighting was “Shrek 2”. It uses two-dimensional texture mapping to store the direct illumination on the surface of the object or person, and then calculates the global illumination after a single bounce by distributed ray tracing[22]. Pixar’s famous film “Cars” is another example.They use multiresolution geometry and texture caches, and use ray differentials to determine the appropriate resolution in order to use ray tracing in highly complex scenes[23].

It is also used for medical purposes, where the use of global illumination technology can provide a more realistic view of the human anatomy, help to understand the internal structure and interaction of the human body, and provide better illustrations for medical training and teaching.[24]



(a) interior design

(b) Porsche rendered by ray tracing technology

Figure 2.8: Ray tracing in art design products

On the other hand, ray tracing is also widely used in industrial design and interior design. It is relatively straightforward and simple to simulate optical effects, and it is possible to programmatically calculate the visual impact of changing the light source or object on the scene[25]. This means that it can not only save the designer a lot of time, but also make the customer feel the potential of the model more intuitively, saving the customer's time.

2.5 Main Algorithm

We define ray in graphics like this[26]: Light travels in straight lines(Although light is actually a wave). Light rays do not collide with each other if they cross. Rays have color and intensity.

The ray is emitted from the light source and then reflected and refracted throughout the scene until it enters our eyes. The physics is invariant under path reversal, if the light source can somehow see the camera, then the ray from the camera must somehow reach the light source (reciprocity).

2.5.1 Ray-Surface Intersection Algorithm

In mathematics, ray is defined by its origin and a direction vector[27].

In the process of path tracing, we need to detect the intersection of ray and triangle mesh, By calculating the intersection point, we can determine whether the intersection point is visible to the light source. And also can determine whether the light source is inside the closed object[27] (an odd number of intersection points means the light source is inside the object, and an even number means the light source is outside the object).

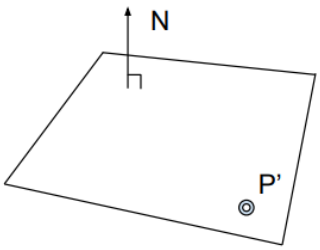


Figure 2.9: Define a plane

In order to determine whether the ray intersects the triangle, we can first determine whether the ray intersects the plane of the triangle, and then determine whether the intersecting point is inside the edges[28]. With a normal vector and a point we can define a plane. So as long as a

point p and a point p' in a given plane satisfy $(p - p') \bullet N = 0$, then we know that point is in the plane.

On this basis, a simpler algorithm is used in this paper, Moller Trumbore Algorithm[28].

A ray with origin O and normalized direction D can be defined as

$$R(t) = O + tD \tag{2.1}$$

We can writing the points on the triangle in terms of barycentric coordinates

$$T(b_1, b_2) = (1 - b_1 - b_2)P_0 + b_1P_1 + b_2P_2 \tag{2.2}$$

Where (b_1, b_2) are the barycentric coordinates, which fufill $b_1 \geq 0$,$b_2 \geq 0$ and $1 - b_1 - b_2 \geq 0$. This (b_1, b_2) can also be used for texture mapping, normal interpolation etc. Computing the intersection between the ray $R(t)$ and the triangle$T(b_1, b_2)$ we can get

$$O + tD = (1 - b_1 - b_2)P_0 + b_1P_1 + b_2P_2 \tag{2.3}$$

The barycentric coordinates (b_1, b_2) and the distance from the ray origin to the intersection point can be found by solving the linear system of equations

$$\begin{bmatrix} t \\ b_1 \\ b_2 \end{bmatrix} = \frac{1}{S_1 \bullet E_1} \begin{bmatrix} S_2 \bullet E_2 \\ S_1 \bullet S \\ S_2 \bullet D \end{bmatrix} \tag{2.4}$$

Where $E_1 = P_1 - P_0$,$E_2 = P_2 - P_0$,$S = O - P_0$,$S_1 = D \times E_2$,$S_2 = S_1 \times E_1$.

2.5.2 Accelerate Algorithm

Since there are usually a large number of triangles in a scene, it would take an unacceptable amount of time to traverse them from every pixel of the screen to check for collisions. We need to use some acceleration algorithms to speed up the detection process.

Bounding Volumes In this paper, the method is to surround a complex object with a simple bounding volume. The object is completely

surrounded by the bounding volume. If the ray does not intersect the volume, it will not intersect the object inside the volume. Therefore, collision detection can be performed on the volume before collision detection. We often use an Axis-Aligned Bounding Box(AABB)[29]. We use axis-aligned boxes because it's easier for the rays to intersect with these horizontal planes, which reduces the amount of computation.

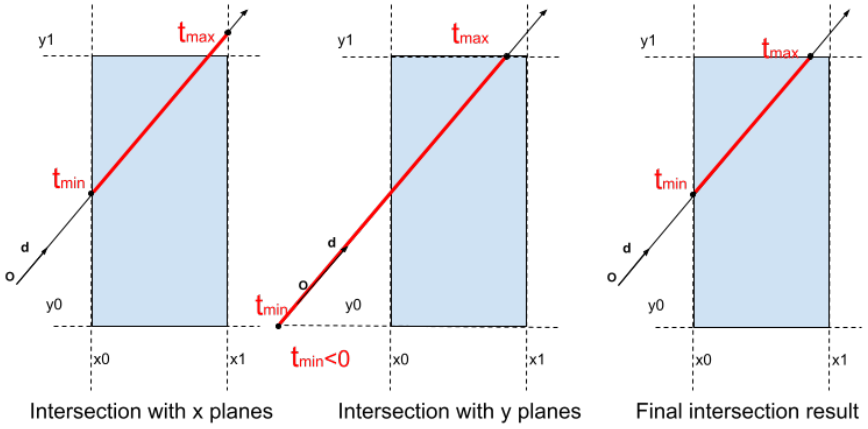


Figure 2.10: Intersection judgment of ray and bounding box in 2D case.

In the two-dimensional case, we find the time T when the ray intersects the two groups of parallel lines that form the cuboid respectively, and take the intersection of these two groups of times to get the time when the ray actually enters and moves out of the cuboid. Generalize to the three dimensional case, the ray enters the box only when it enters all pairs of slabsand the ray exits the box as long as it exits any pair of slabs. So for each pair, we calculate the t_min and t_max, for the 3D box[30].

$$t_{enter} = max\{t_{min}\}, t_{exit} = min\{t_{max}\} \tag{2.5}$$

Ray is not a line, it has direction. When $t_{exit} < 0$ it means the box must behind the ray, they have no intersection. When $t_{exit} \geq 0$ and $t_{enter} < 0$, it means the light source is inside the box, and they must have intersection. In summary, If $t_{enter} < t_{exit} t_{exit} \geq 0$, we know the ray stays a while in the box, in other word, they have intersection.

BVH(Bounding Volume Hierarchy)[31] After the scene model was determined, we first completed the pre-processing of the scene, the acceleration structure, and then considered how to find the intersection point with the ray. Establishing kd-tree[32] is one of the methods. In order to ensure that the size of the enclosing box is balanced, we divide it along the three axes of X, Y and Z successively. We don't store the actual triangle data on the intermediate node, we only store it on the leaf node. The problem of using kd-tree is that the same object may belong to multiple bounding volume. Different leaf nodes may store the same triangle data, and the establishment of kd-tree needs to consider the intersection of triangle and bounding volume, which is quite complex. In this experiment, we choose BVH method to divide the bounding volume based on object rather than space.

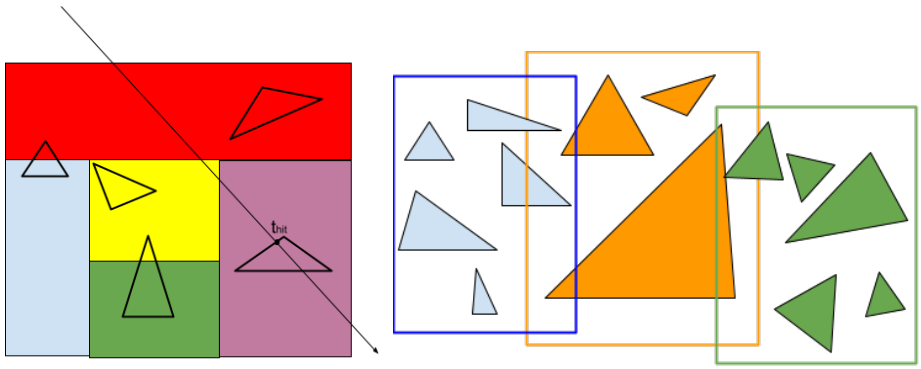


Figure 2.11: BVH method.

We find bounding volume first. Then recursively split set of objects in two subsets, and then recompute the bounding box if the subsets. Stop when only one triangle in the box. We store objects in each leaf node. All the other parts are used to determine the accelerated structure. The method of kd-tree can be learned in the actual partitioning process. One dimension is selected for each division to ensure that the size of the enclosing box is uniform. When dividing objects, we always take the object in the middle for division, so as to ensure that the number of objects in the enclosing box on both sides is more balanced (Ensure the balance of the tree, the smaller the depth of the tree can reduce the number of searches[33]). Use the quick selection algorithm[34] to select

the triangle in the middle of the enclosing box. Finally, the triangle data is stored in the leaf node.

2.5.3 Monte Carlo Integration

PDF(Probability Distribution Function) A random variable X that can take any of a continuous set of values, where the relative probability of a particular value is given by a continuous probability density function $p(x)$

$$\begin{aligned} Conditionsonp(x) : p(x) \geq 0 and (x)dx = 1 \\ Expectedvalueof X : E[X] = \int xp(x)dx \end{aligned} \tag{2.6}$$

Monte Carlo integration is a technique that uses random numbers for numerical integration. It is a special Monte Carlo method that can numerically compute definite integrals. For any function $f(x)$, we sample a probability density function in the integral domain to obtain the corresponding function $f(X_i)$ and probability density $p(X_i)$ of the sampled samples, and divide them to average. The more samples we have, the less error we're going to get. (Sampling on a sample means integrating on the current sample). We estimate the integral of a function by averaging random samples of the function's value.

$$\int f(x)dx = \frac{1}{N} \sum_{i=1}^{N} \frac{f(X_i)}{p(X_i)} (X_i \sim p(x)) \tag{2.7}$$

2.6 Radiometry

Light transmission is the theoretical foundation of computer graphics, formalizing mathematical models and driving modern rendering and image synthesis, as well as other related research, such as inverse rendering and computer vision[35]. For many years, optical transport has been formulated in the context of classical radiometry. It perform lighting calculations in a physically correct manner.The core quantity of radiometry are values of intensity and radiance.

In Radiometry, radiant energy is the energy of electromagnetic radiation. It is measured in units of joules, and denoted by the symbol $Q = [J = Joule]$. Radiant flux (ϕ) is the energy emitted, reflected, transmitted or received, per unit time. This value indicates how bright a light source is, denoted by the symbol $\phi = \frac{dQ}{dt} = [W = Watt]$. The Radiant Intensity is power per unit solid angle, denoted by the symbol $I(\omega) = \frac{d\phi}{d\omega}$. Solid Angle is ratio of subtended area on sphere to radius squared, denoted by the symbol $\Omega = \frac{A}{r^2}$.

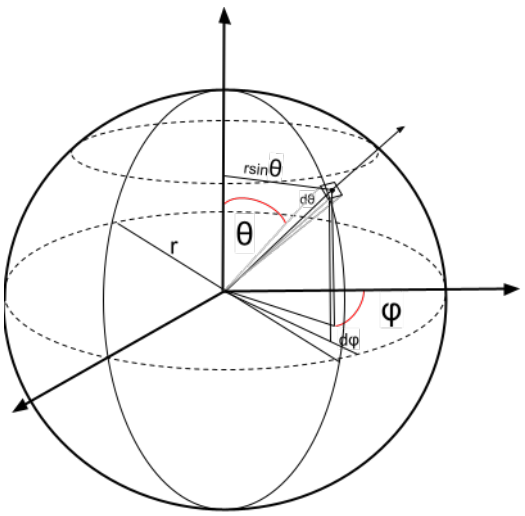


Figure 2.12

From the spherical coordinate system in the figure above, φ and θ can uniquely determine one direction in space. The unit solid Angle can be calculated as

$$
\begin{aligned}
dA &= (rd\theta)(rsin\theta d\varphi) \\
&= r^2 sin\theta d\theta d\varphi \\
d\varphi &= \frac{dA}{r^2} = sin\theta d\theta d\varphi
\end{aligned}
\tag{2.8}
$$

The Irradiance($E(x) = \frac{d\varphi(x)}{dA}[\frac{W}{m^2}]$) is the power per unit area incident on a surface point, according to Lambert's Cosine Law[36], irradiance at surface is proportional to cosine of angle between light direction and surface normal. The Radiance($L(p,\omega)) = \frac{d^2\varphi(p,\omega)}{d\omega dAcos\theta}$,$cos\theta$ account for projected surface area)is the power emitted, reflected, transmitted or received by a surface, per unit solid angle, per projected unit area, it is the fundamental field quantity that describes the distribution of light in an environment, it is the quantity associated with a ray, our rendering is all about computing radiance. Incident Radiance is the irradiance per unit solid angle arriving at the surface. Exiting surface radiance is the intensity per unit projected are leaving the surface. (Unit Hemisphere:H^2)

$$
\begin{aligned}
dE(p,\omega) &= L_i(p,\omega)cos\theta d\omega \\
E(p) &= \int_{H^2} L_i(p,\omega)cos\theta d\omega
\end{aligned}
\tag{2.9}
$$

2.7 BRDF(Bidirectional Reflectance Distribution Function)

The BRDF [37]describes how the irradiance received by one unit area from one unit solid angle is distributed to other different angle. In other word, how much light is reflected into each outgoing direction from each incoming direction. In the case of specular reflection, all the incident radiance is distributed in the direction of outgoing reflection, while there is no energy in the non-specular direction. In the case of diffuse reflection, the incident radiance is evenly distributed in different reflected directions.

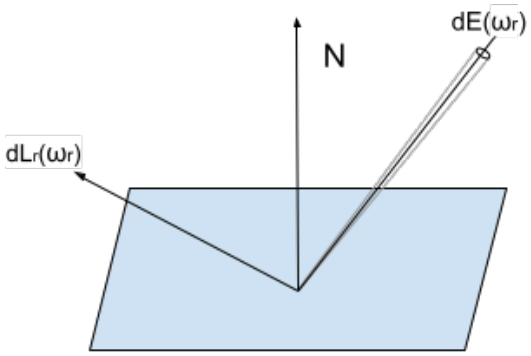


Figure 2.13

Given the energy and the angle of the incident ray, when ray reach an object's surface, it will absorb the incoming energy and radiates in different directions. We can use the BRDF to find the amount of energy radiated in a particular direction.

$$\begin{aligned} f_r(\omega_i \rightarrow \omega_r) &= \frac{dL_r(\omega_r)}{dE_i(\omega_i)} \\ &= \frac{dL_r(\omega_r)}{L_i(\omega_i cos\theta_i d\omega_i)} \end{aligned} \tag{2.10}$$

2.8 The Rendering Equation

Reflected radiance depends on incoming radiance, and incoming radiance also depends on reflected radiance(at another point in the scene). So we can think of rendering equation as a recursive process.According to Kajiya's theory[9], light can be described as an electromagnetic radiation using the following equation(rendering equation)

$$L(p, \omega_r) = L_e(p, \omega_r) + \int_{\Omega^+} f_r(p, \omega_i \rightarrow \omega_r) L_i(p, \omega_i) cos\theta_i d\omega_i \qquad (2.11)$$

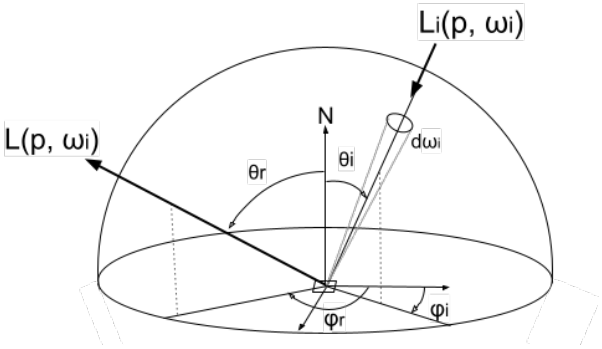


Figure 2.14

Where $L(p, \omega_r)$ is the radiance from point p in the direction of ω_r. $L_e(p, \omega_r)$ is the emittance term, which represents the radiance directly emitted from point p in direction ω_r . $f_r(p, \omega_i \rightarrow \omega_r)$ is the BRDF of the surface at point x, and this is the radiance from the direction of ω_i reflected to the direction of ω_r , it represents what percentage of the radiance is going to be radiated. $L_i(p, \omega_i)$ represents the radiance which comes from direction ω_i to p. Ω^+ is the upper hemisphere oriented around the normal vector N, and θ_i is the angle made by the direction with normal vector N.[38]

The only thing we don't know about the above equation is how much radiance each object reflects. We can think of this equation as saying that the energy radiated from a location is equal to the light emitted from that location plus the energy reflected to that location through BRDF

from other objects or surfaces of light sources.This can be discretized to a simple matrix equation (L,E are vectors,K is the light transport matrix).:

$$\begin{aligned} L &= E + KL \\ IL - KL &= E \\ (I - K)L &= E \\ L &= (I - K)^{-1}E \end{aligned} \tag{2.12}$$

According to Binomial theorem, we can get approximate set of all paths of light in scene:

$$\begin{aligned} L &= (I + K + K^2 + K^3 + ...)E \\ L &= E + KE + K^2E + K^3E + ... \end{aligned} \tag{2.13}$$

Now we can define the entire rendering equation. The final result is equal to what we see without reflection(light source), plus what we see with one reflection(direct illumination), plus two, three, or more results after reflection(indirect illumination).

Chapter 3

Design

The hardware information used in this experiment is GPU 12th Gen Intel(R) Core(TM) I7-12700KF, 3.61ghz, 32GB running memory size, and NVIDIA GeForce RTX 3060 12GB graphics card. The resolution of out put image is 800x600.

3.1 CPU and GPU

Compared with GPU,CPU has more RAM, and is more suitable for dealing with complex geometry computation, physical simulation and particle system. Examples include subdividing millions of polygons into surfaces, expanding geometry to generate hair, reading larger texture assets, or generating clouds and explosion effects. These tasks can easily crash on GPU, because it doesn't have enough RAM. At the same time, for some complex operations, it is difficult for GPU to keep all the contents running synchronously, while CPU has better stability in this aspect. The advantage of GPU over CPU is mainly its processing speed. GPU have thousands of small but efficient cores in a single graphics card, which can process a large number of tasks in parallel, focusing all computing power on a specific task, while CPU have only a few or a dozen cores. GPU is optimized for processing huge tons of data by performing the same operation quickly and repeatedly, and the transformation of object vertex coordinates and the coloring of screen pixels during rendering are a lot of repetitive operations. In the rendering of some complex scenes, the

CPU may take hours to process, while the GPU can output the correct result in just a few minutes.

The whole process of the rendering model is mainly divided into two parts. The first part of the process is the preprocessing of the data, the packaging of the scene data and the construction of the acceleration structure of the scene data, which is carried out in CUP. The second part of the flow is the main algorithm of Monte Carlo path tracing, which is carried out in GPU. Such a design ensures the maximum utilization of the processing data characteristics of the CPU and GPU. Rational use of their characteristics to improve the running efficiency and enhance the stability of the program. similar to the mainstream ray tracing methods.

3.2 CPU Part

In the CPU processing stage, we first read the 3D model using the Assimp library, and then construct the BVH acceleration structure based on the triangle data of the model. Store the triangles in the leaf nodes of the octree. Then we store the information of the Bounding Volume(Boundary location) and the triangle information (position, normal, color, texture coordinates, light source) in separate arrays. In general, the number of triangles in our scene is quite large, and it is easy to exceed the maximum length of the general data format, so we use TBO (Texture Buffer Objects) for data transfer. TBO is a special form of texture that allows direct access to the contents of a cached object from a shader, which can be thought of as a giant 1D texture. After packing this data as a TextureBuffer, we pass the resulting TextureBuffer into GLSL, where we read it using the texelFetch () function.

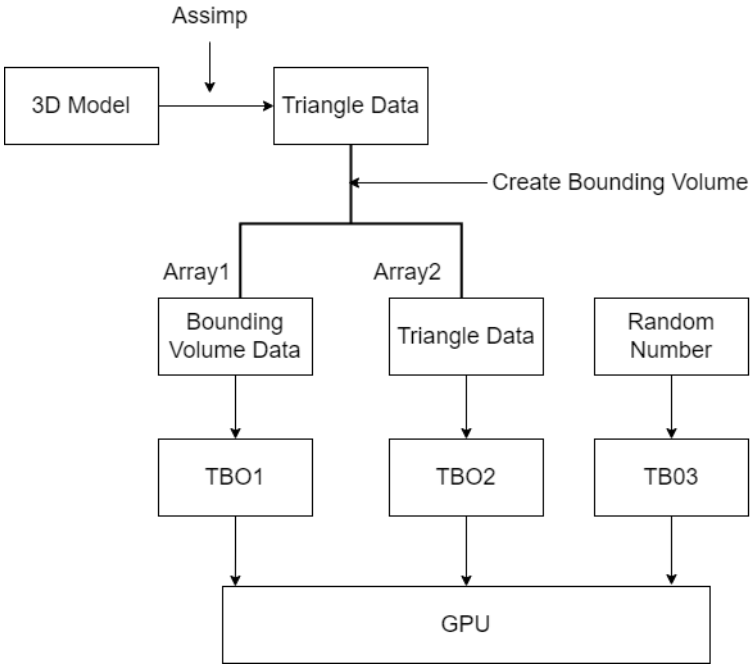


Figure 3.1: Processes in the CPU.

Since we always use a lot of random numbers in the path tracing algorithm, and GLSL cannot generate random numbers, we also use TBO to store and pass our random numbers to GLSL. Finally we use the OpencV library, when we get the result of a single rendering we store the color value of each pixel, and finally after multiple rendering, we calculate the average value of each pixel to achieve the purpose of reducing the noise, and obtain the final pixel color for saving.

3.3 GPU Part

The main part of the Monte Carlo path tracing is implemented using GLSL. With the rendering equation we have mentioned before and the Monte Carlo integration method, we can now try to solve our rendering equation using Monte Carlo integration method. The direct illumination for a point in the scene is equal to the integral of the light from the source in the scene reflected to the hemisphere surface in our viewing direction after passing through the BRDF. After applying the Monte Carlo integral we can sample in a random direction over the hemisphere,

$pdf(w) = \frac{1}{2\pi}$because we sample uniformly across hemispheres.

$$L_o(p, \omega_0) \approx \frac{1}{N} \sum_{i=1}^{N} \frac{L_i(p, \omega_i) f_r(p, \omega_i, \omega_o)(n \bullet \omega_i)}{pdf(\omega_i)} \tag{3.1}$$

If the light hits an object, since we know how to calculate the direct illumination, as shown in the figure, if the light generated at point P hits point Q, then the radiance reflected from point Q to point P can be regarded as the calculated direct illumination at point Q. We only shoot one ray in one direction at any given point, because if we generate multiple rays, each ray will continue to generate multiple rays after it hits the object, and the number of rays will increase exponentially, causing the program to crash.

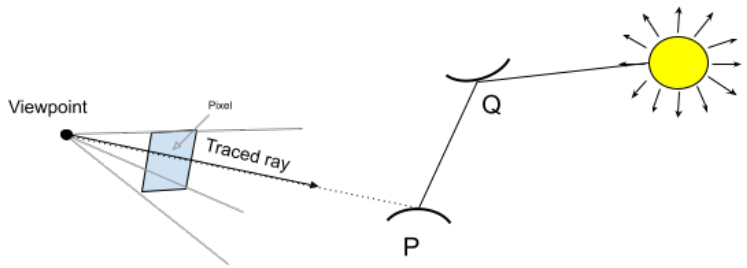


Figure 3.2

The pseudocode of the above procedure is given here:

Algorithm 1 Calculate color for Single light bounce

Randomly choose One direction $\omega_i \sim PDF(\omega)$
$L_0 = 0$
for each ω_i **do**
  Trace a ray $r(p, \omega_1)$
  if ray r hit the light **then**
    $L_0 += L_i * f_r \frac{cos\theta}{pdf(\omega_i)}$ ▷ This is a comment
  else if ray r hit an object at q **then**
    $L_0 += Shade(q, -\omega_i) * f_r \frac{cos\theta}{pdf(\omega_i)}$ **return** L0

But the result in this case will be a lot of noise (because we only

generate one ray, and Monte Carlo integration requires a lot of sampling to reduce the noise). We can solve this problem by using a large number of paths. We can emit a ray from each pixel multiple times in a random direction, and finally average their radianceThen we need to determine when the light stops bouncing, we can't let the whole process go on indefinitely. Previously, we always shoot a ray at a shading point and get the shading result Lo, we want to stop the ray propagation at some probability, but also we do not want to change the final expectation Lo. Here we can use Russian Roulette(RR) to solve this problem. Suppose we manually set a probability $P(0 < P < 1)$. With probability P, shoot a ray and return the shading result divided by P. With probability $1-P$, we stop shoot ray and return 0.

There is no recursive structure in GLSL, so although the above steps appear to be a recursive pattern, we can take advantage of the stack structure to expand the recursion into an iterative form. We define an array as long as possible.Each time we put the color data on the stack, we exit the stack one by one to calculate the final color.

How to calculate the random direction on the hemisphere is also a problem. This project is based on Marsaglia's method [39] sampling in the special case of z-axis as normal, and uses Ortho-normal Bases to generalize it to the case of the general normal. This way we can generate reflected rays of random direction by random numbers obtained from the CPU.

Another problem that needs to be solved is that if the light source is small and we sample around the collision point, there may be a problem that a lot of rays will not hit the light source, which makes the sampling process not really efficient. If we can find a suitable PDF, it can make our algorithm more efficient, and the method used in this paper is to sample directly on the light source. In this way, according to the properties of Monte Carlo integration, we need to redefine the rendering equation on

the light source. Then we can rewrite the rendering equation as:

$$L_o(x,\omega_0) = \int_{\Omega^+} L_i(x,\omega_i) f_r(x,\omega_i,\omega_o) cos\theta d\omega_i \\ \int_A L_i(x,\omega_i) f_r(x,\omega_i,\omega_o) \frac{cos\theta cos\theta'}{\| x' = x \|^2} dA \tag{3.2}$$

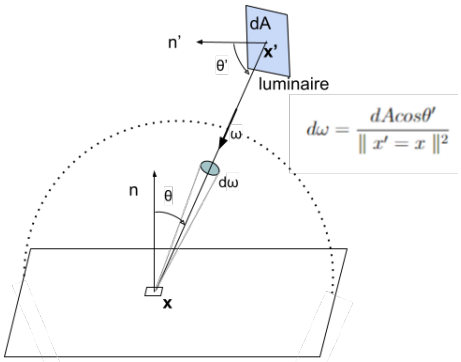


Figure 3.3

Now we can consider the radiance coming from two parts, from light source is direct illumination, no need to do RR, from other reflectors is indirect, need RR. When we sample the light source, we also need to shoot a ray at the light source first. If it hits another object before hitting the light source, we can determine that the point is occluded from the light source.

Algorithm 2 Recursively get the final color

$shade(p, \omega_0)$

Uniformly sample the light at x' (pdf(light)= $\frac{1}{A}$)

$$\mathrm{L}_{dir} = L_i * f_r * cos\theta * cos\theta' * \frac{\parallel x' = x \parallel^2}{pdf(light)}$$

Test Russian Roulette

Uniformly sample the hemisphere toward $\omega_i (pdf(hemisphere) = \frac{1}{2\pi})$

Trace this ray $r(p, \omega_i)$

if ray hit a non-emitting object at q **then**

$$\mathrm{L}_{indir} = shade(q, -\omega_i) * f_r * cos\theta * \frac{RR}{pdf(hemisphere)}$$

return $\mathrm{L}_{dir} + L_{indir}$

(f_r is BRDF fuction, RR is Russian Roulette probability.)

3.4 External Liberaries and Tools

OpenGL OpenGL(Open Graphics Library)[37] It is a cross-language and cross-platform application programming interface (API) for rendering 2D and 3D vector graphics. This interface consists of nearly 350 different function calls for drawing anything from simple graphic bits to complex three-dimensional views. OpenGL is commonly used in CAD, virtual reality, scientific visualization programs, video game development and interact with graphics processing units(GPU) for hardware accelerated rendering. Today, OpenGL is the most widely accepted API for processing 2D/3D graphics in the video industry. Based on this, application functions on various computer platforms and many applications on devices have been spawned for the study of computer vision technology. It is independent of the Windows operating system and operating system platform, can undertake a variety of different neighborhood development and content creation, in short, it helps developers to achieve PC, workstations, super computers and all kinds of hardware, such as industrial control, high performance, high for visual demands high visual graphics processing software development.

GLSL OpenGL Shading Language (OpenGL Shading Language) is a short custom program written by developers, which is used for Shading programming in OpenGL. It is executed on the Graphic Processor Unit and replaces part of the fixed rendering pipeline to make the different levels in the rendering pipeline programmable. GLSL Shader code is divided into two parts: Vertex Shader and Fragment, and sometimes Geometry Shader. GLSL uses C language as the basic high order coloring language to avoid the complexity of using assembly language or hardware specification language. There are many benefits to using GLSL, such as its cross-platform compatibility with multiple operating systems, including Linux, macOS, and Windows. It has the ability to write shaders that can be used on the graphics cards of any hardware vendor that supports OpenGL coloring language. Each hardware vendor includes a GLSL compiler in its drivers, allowing each vendor to create code optimized for its specific graphics card architecture.

GLFW GLFW(Graphics Library Framework) is an Open Source, lightweight, multi-platform library for OpenGL, OpenGL ES and Vulkan development on the desktop. It provides a simple API for creating windows, contexts and surfaces, it also provides the functions of processing gamepad, keyboard and mouse input. GLFW is written in C and supports Windows, macOS, X11 and Wayland.

GLEW The OpenGL Extension Wrangler Library (GLEW) is a cross-platform open-source C/C++ extension loading library. GLEW provides efficient run-time mechanisms for determining which OpenGL extensions are supported on the target platform. Different graphics card companies release extension functions that only their own graphics card supports, and if you want to use them, you have to find the latest header file. With the GLEW extension library, GLEW automatically identifies all OpenGL advanced extension functions supported by your platform. You don't have to manually find the interface to the function.

GLM OpenGL Mathematics (GLM) is a C++ mathematics library for graphics software based on the OpenGL Shading Language (GLSL) specification. To make it easier for programmers to learn and use, it provides classes and functions with the same names and functions as GLSL. GLM is not limited by GLSL features. It is a GLSL-based extension that provides functions such as matrix transformations, quaternions, semi-base types, random numbers, and so on. In ray tracing, rasterization, image processing, and physical simulation, we often need simple and convenient mathematical libraries, and GLM can provide most of the mathematical functions we need.

Assimp Open Asset Import Library (Assimp) is a cross-platform 3D model import library. It written in C++, and offers interfaces for both C and C++. When rendering in OpenGL, sometimes need to use model materials download from the internet. However, due to the large number of model formats in the network, we need to parse models in different formats. As an open source project, ASSIMP has designed a set of extensible architecture, which provide a common application programming

interface (API) for different 3D asset file formats. Assimp currently supports 57 different file formats for reading, including COLLADA(.dae), 3DS, DirectX X, Wavefront OBJ(.obj) and Blender 3D(.blend).

Chapter 4

Evaluation

The acceleration effect of the BVH acceleration structure mentioned in Section 2.4 is tested, and evaluated the final rendering effect of the algorithm, including comparison with real photos and comparison with images generated by raster transformation. For the results of ray-tracing algorithms, the image should ideally be close to the real result, with obvious optical effects such as soft shadows and color bleeding, and an ideal stable program should be able to continue rendering computations for long periods of time without crashing.

For these tests, I used 3Dmax to process the model, select the appropriate size and position for placement, and adjust the camera to the appropriate position to observe the overall rendering effect. The models used in the program are from Stanford University's 3D models rabbit and Dragon, and the Utah teapot, and other 3D objects are from the website free3D.com.

To calculate the rendering speed, I timed the program using the c++11 library function clock() and timed the CPU processing time separately from the gpu processing time in order to compare the actual performance of the speedup algorithm. First, the program with acceleration structure is tested for the difference of running speed corresponding to the different number of triangles in the scene. (The number of iterations sampled during testing is 8, and the number of sample is 100) The test results are shown in the following figure:

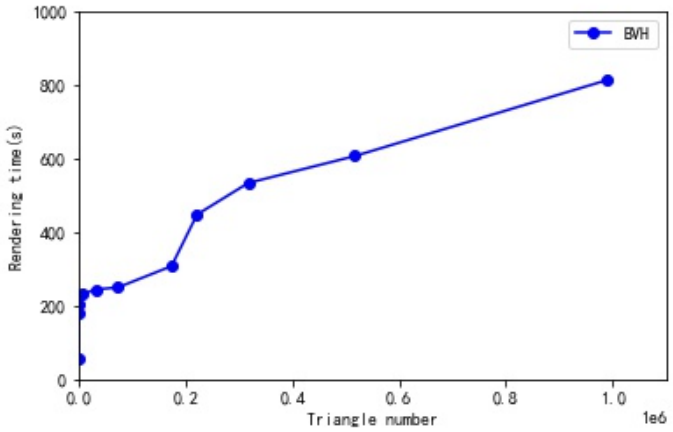


Figure 4.1: Rendering time variation for different number of triangles(With accelerate structure)

Clearly as you can see from the graph, the triangle number less than 100, rendering time as triangle populations increased very fast. This is due to the small number of triangles in the scene, and the construction of BVH structure leads to the need to detect the bounding volume first when detecting the collision. For the scene with only a few triangles, the acceleration effect is not obvious. And when the number of triangles is small, the curve has a certain fluctuation, which is caused by the location concentration or scattered emission of the model in the scene. A scene with the same number of triangles takes longer to render when the model is scattered across the scene(figure). As the number of triangles increases, the increase in rendering time flattens out and increases logarithmically, which is a nice advantage when dealing with complex scenes with a large number of triangles.

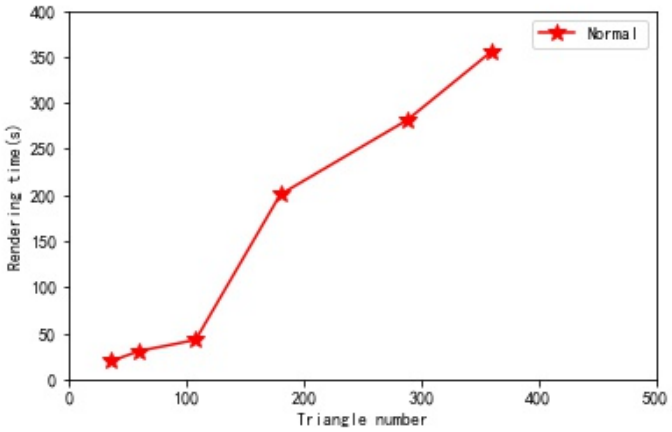


Figure 4.2: Rendering time variation for different number of triangles(Without accelerate structure)

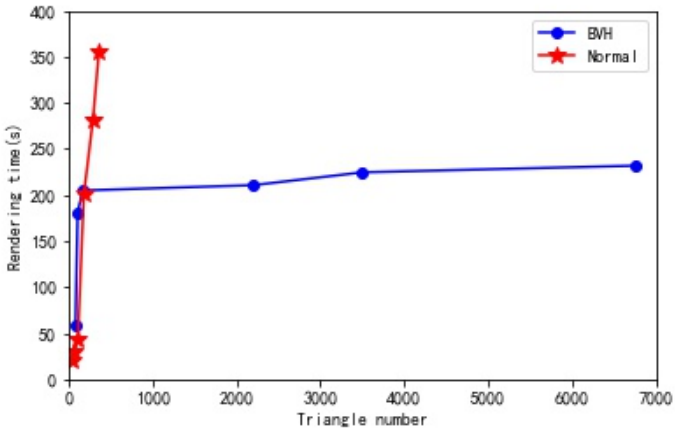
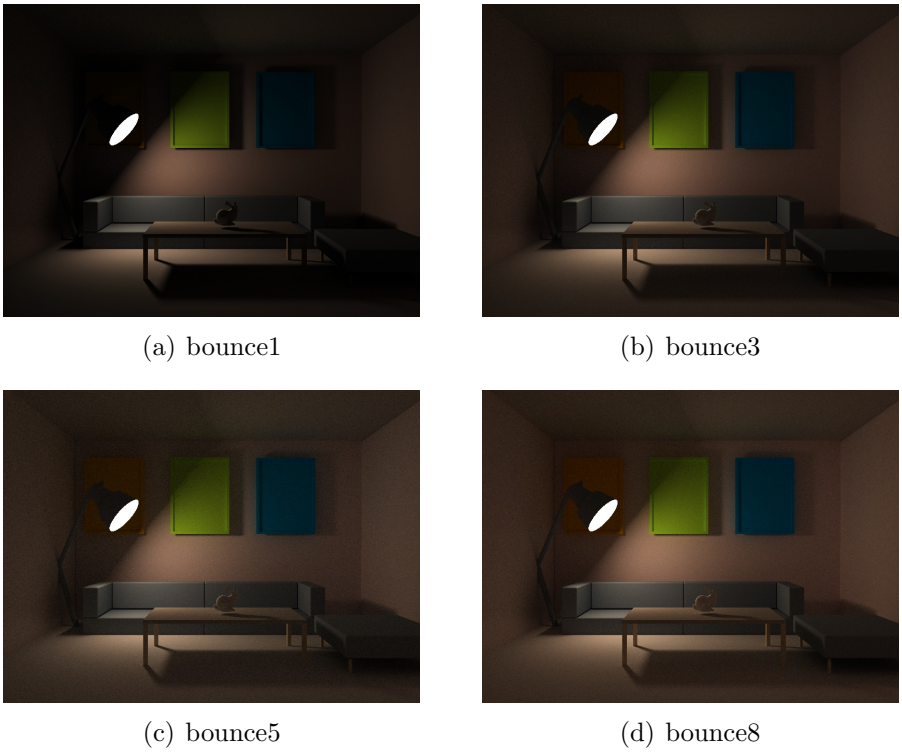


Figure 4.3: Compare the two methods

Secondly, the same number of iterations and rendering times are used in the same scene to compare the rendering time between using the accelerated results and not using the accelerated structure. According to the test results, it is obvious that although the algorithm without accelerating structure has a slight advantage in speed when the number of triangles is small, the algorithm without accelerating structure has an almost exponential increase in operation time when the number of tri-

angles gradually increases. In fact, during testing, when the number of triangles exceeds 3000, renderers without acceleration structures often crash.The stability of the algorithm with acceleration structure is much more than that of the algorithm without acceleration structure.

Using path-tracing algorithms inevitably consumes a lot of time for rendering, and the calculation time is affected not only by the number of triangles in the scene but also by the number of times each light bounced in the scene, that is, the number of times each pixel is iterated to calculate the result of the rendering equation.



(a) bounce1 (b) bounce3

(c) bounce5 (d) bounce8

Figure 4.4: Rendering results after light bounces different times

In order to obtain better image quality in a relatively short time, under the premise of the same scene and the same sampling times, using different maximum ejection times under the premise of using BVH structure acceleration, counting the rendering time, and comparing the quality of the final rendering result. Here are the final results rendered for different number of ejections. As the number of ejections continues to

increase, the scene gradually brightens. When the number of light ejection is less than three times, the image information on the back of the light source is almost invisible, and when the number of light ejection is five times, the orange picture frame in the left shadow can be clearly seen. When the time of light bounce increase, the whole process of rendering time increase gradually, but the final brightness convergence gradually. You can see that the difference in brightness between 8 and 15 ejections is no longer significant, but the difference in render time is 1000 seconds. In practice, we can use the appropriate number of ejections as needed. (Appendix2 figure 6 gives a comparison at detail.)

(a) bounce8

(b) bounce15

Figure 4.5: Compare the rendering results with light bounces 8 and 15 times

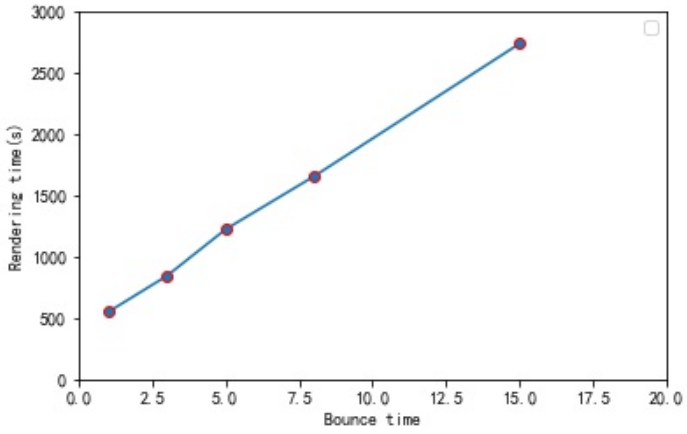
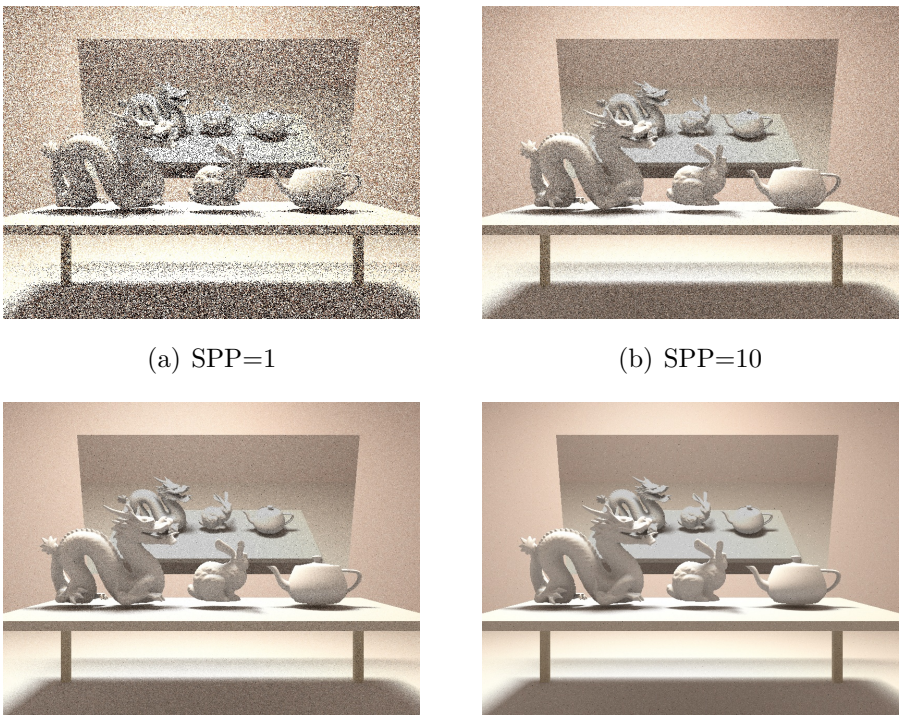


Figure 4.6: Rendering time variation for different number of bouncing

In this project, random light is emitted to each pixel for many times, and finally the average method is used to denoise the image. The number of Spp is the key to the effect of image denoising. The images generated by different Spp times are compared, and the final denoising effect of this method is also evaluated. It can be seen that the number of noise points in the image has been significantly reduced when SPP =200, but the existence of noise points can still be felt on the whole, and the picture quality is not good. When SPP =1000, the noise of the image is almost invisible, and a few black noise points can be seen when zooming in.



(a) SPP=1 (b) SPP=10

(c) SPP=50 (d) SPP=200

Figure 4.7: Different output image based on different SPP

When Spp=1500, the number of black noise points continues to decrease or fade, and the image quality is close to the photo quality. The sampling method based on Monte Carlo integration can approximate the real results on the basis of a large number of samples. However, we can see that there are aliases at the edge of the object, which affects the quality of the image to a certain extent. Perhaps multi-sampling the pixels on the edge of the object based on the Monte Carlo method can achieve anti-aliasing effect and further improve the image quality.

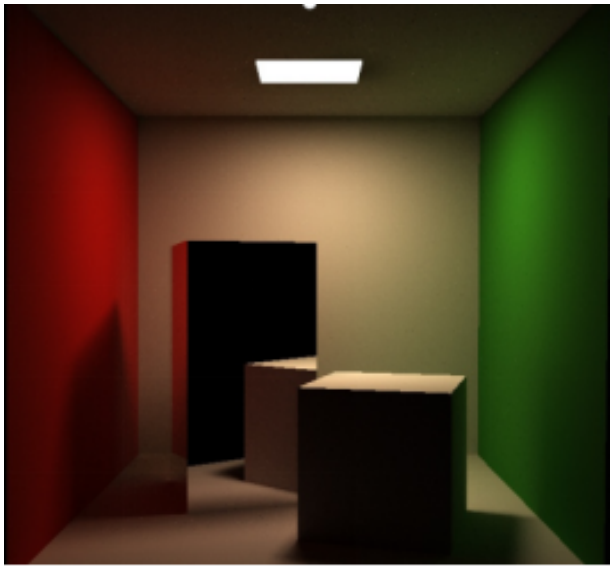
(a) SPP=1000

(b) SPP=1500

Figure 4.8: Compare the output image quality from SPP=1000 and SPP=1500

To evaluate the image quality of the rendered results, I used Cornell Box, a model from Cornell University, and compared the rendered results with real photos given by Cornell University. You can see that except for the jagged edges of the objects, the overall effect is very similar to the picture, which is a good imitation of the real lighting conditions. At the same time, we can clearly see the effect of color bleeding and soft shadow, which is difficult to be achieved by rasterization.

(a) Real Photo



(b) Rendering result

Figure 4.9: The first image is real cornell box photo provided by Cornell University, the second one is rendering image using renderer build by this project.

Chapter 5

Conclusions

This paper has shown that it is possible to use OpenGL to create a stable ray-tracing renderer from the underlying layer that can render complex scenes. And it is stable and feasible to use TextureBufferObject(TBO) to transfer a large amount of data from CPU to GPU in OpenGL. In this paper, some important technical difficulties that need to be implemented in the idea of path tracing algorithms are investigated, including ray collision detection, bounding volume generation, random ray generation, Monte Carlo integral method, and Russian Roulette algorithm.They need to be used in combination to implement a full path tracing algorithm.The final rendering results prove that the path tracing algorithm based on Monte Carlo method can simulate the physical properties of light well, and the final image has a quality comparable to that of real photos. On this basis, this paper proves that the acceleration algorithm is very efficient in ray tracing, which can greatly improve the rendering speed and enhance the stability of the rendering program. At the same time, the image brightness tends to converge with the increase of light ejection times.

Chapter 6

Further Research

Visual operation interface At present, this project is to set up the scene in 3Dmax in advance, and then export the model for OpengL to use. If we need to adjust the size, position, angle of the model, then we should first adjust in 3Dmax, and then export it to use. The operation is tedious, which greatly reduces the efficiency of program debugging.At the same time, due to the slow running speed of the ray tracing algorithm, the problem caused by the camera position or the line of sight Angle needs to be manually adjusted, which will consume a lot of time. The following plan is run in rasterized form first, with ImGUI library directly visualized in OpengL to adjust the position of the model and other attributes, and control the camera to the appropriate position. After the scene is determined, we can package the data to glsl and run the ray tracing algorithm for rendering.

Textures and other material At present, only a single color and brightness light source is used in this experiment, no texture of the object is added, and only a single color is used for rendering. The model is also seen as having only diffuse, specular, or refracted reflections. In reality, objects often have diffuse reflection, refraction and specular reflection at the same time. At present, the objects with mixed materials are not processed. Adding these items later will make the scene look more realistic and beautiful.

Image anti-aliasing Although the final image effect after a large number of samples is close to the level of real photos, there are still clearly visible jagged edges at the edges of the model, which has a certain impact on the performance of the overall image. In particular, this effect is more pronounced for objects with edge details. It can be solved by supersampling the image edge pixels.

Hardware The project has only been tested on a single computer, and ray tracing has relatively high hardware requirements. Different CPUs, GPUs, and RAMS may exhibit large rendering time gaps, as well as other hardware-specific bugs. Testing on different devices can help improve program stability and find more appropriate data processing methods.

Real-time ray tracing In this project, the time required to render a single image at an acceptable level of quality was in the order of 15 minutes or more, which is far from the speed required for real-time ray tracing (If the ray tracing can achieve 30 frames per second, it can be called real-time ray tracing)The introduction of a hybrid rendering pipeline, which combines the advantages of rasterization and ray tracing, may improve rendering speed.

Bibliography

[1] Cristian Lambru, Anca Morar, Florica Moldoveanu, Victor Asavei, and Alin Moldoveanu. Comparative analysis of real-time global illumination techniques in current game engines. *IEEE Access*, 9:125158–125183, 2021.

[2] Masanori KAKIMOTO, Tomoaki TATSUKAWA, Geng CHUN, and Tomoyuki NISHITA. Real-time reflection and refraction on a per-vertex basis. *The Journal of the Institute of Image Electronics Engineers of Japan*, 37(3):196–205, 2008.

[3] Bekaert P. Bala K. Dutre, P. Advanced global illumination (2nd ed.). 2006.

[4] Appel and Arthur. Some techniques for shading machine renderings of solids. page 37–45, 1968.

[5] Tom McReynolds and David Blythe. *Advanced Graphics Programming Using OpenGL (The Morgan Kaufmann Series in Computer Graphics)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[6] B. Livny. *mental ray for Maya, 3ds Max, and XSI: A 3D Artist's Guide to Rendering*. Serious skills. Wiley, 2008.

[7] Turner Whitted. An improved illumination model for shaded display. *SIGGRAPH Comput. Graph.*, 13(2):14, aug 1979.

[8] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. *SIGGRAPH Comput. Graph.*, 18(3):137–145, jan 1984.

[9] James T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150, aug 1986.

[10] Eric P. Lafortune and Yves D. Willems. Bi-directional path tracing. In *Proceedings of Third International Conference on Computational Graphics and Visualization Techniques (Compugraphics '93)*, pages 145–153, Alvor, Portugal, December 1993.

[11] Eric Veach and Leonidas Guibas. Bidirectional estimators for light transport. In *In EGWR '94*, pages 147–162, 1994.

[12] Chun-Fa Chang, Kuan-Wei Chen, and Chin-Chien Chuang. Performance comparison of rasterization-based graphics pipeline and ray tracing on gpu shaders. In *2015 IEEE International Conference on Digital Signal Processing (DSP)*, pages 120–123, 2015.

[13] Lingchun Li, Xubo Yang, and Shuangjiu Xiao. Efficient visibility projection on spherical polar coordinates for shadow rendering using geometry shader. In *2008 IEEE International Conference on Multimedia and Expo*, pages 1005–1008, 2008.

[14] Markus Giegl and Michael Wimmer. Queried virtual shadow maps. pages 65–72, 01 2007.

[15] Myst PC Game. 1993.

[16] D. Pohl. Quake 4 ray traced. 2007.

[17] Jacco Bikker. Real-time ray tracing through the eyes of a game developer. In *2007 IEEE Symposium on Interactive Ray Tracing*, pages 1–1, 2007.

[18] Michelle Menard and Bryan Wagstaff. *Game Development with Unity*. Australia ; Boston, MA : Cengage Learning,, 2015.

[19] P . V . Satheesh. *Unreal Engine 4 Game Development Essentials*. 2016.

[20] A. Kaplanyan. Real-time diffuse global illumination in cryengine 3. 2010.

[21] James Archer. Here are all the confirmed ray tracing and dlss games so far. July 22, 2022.

[22] Eric Tabellion and Arnauld Lamorlette. An approximate global illumination system for computer generated films. *ACM Trans. Graph.*, 23(3):469–476, aug 2004.

[23] Per H. Christensen, Julian Fong, David M. Laur, and Dana Batali. Ray tracing for the movie 'cars'. In *2006 IEEE Symposium on Interactive Ray Tracing*, pages 1–6, 2006.

[24] Philip Voglreiter, Jürgen Wallner, Knut Reinbacher, Katja Schwenzer-Zimmerer, Dieter Schmalstieg, and Jan Egger. Global illumination rendering for high-quality volume visualization in the medical domain. 2015. face 2 face - science meets art ; Conference date: 02-10-2015 Through 03-10-2015.

[25] Easy Render. 3d interior design.

[26] Philip Dutré, Henrik Wann Jensen, Jim Arvo, Kavita Bala, Philippe Bekaert, Steve Marschner, and Matt Pharr. State of the art in monte carlo global illumination. In *ACM SIGGRAPH 2004 Course Notes*, SIGGRAPH '04, page 5–es, New York, NY, USA, 2004. Association for Computing Machinery.

[27] Volume information. *Mathematics of Computation*, 47(176):763–64, 1986.

[28] Tomas Möller and Ben Trumbore. Fast, minimum storage ray/triangle intersection. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, page 7–es, New York, NY, USA, 2005. Association for Computing Machinery.

[29] René Weller, Jan Klein, and Gabriel Zachmann. A model for the expected running time of collision detection using aabbs trees. In *Proceedings of the 12th Eurographics Conference on Virtual Environments*, EGVE'06, page 11–17, Goslar, DEU, 2006. Eurographics Association.

[30] You-Dong Liang and B. A. Barsky. A new concept and method for line clipping. *ACM Trans. Graph.*, 3(1):1–22, jan 1984.

[31] Olmedo Arcila, Simena Dinas, and Jose Bañón. Collision detection model based on bounding and containing boxes. pages 1–10, 10 2012.

[32] Alexander Reshetov. Omnidirectional ray tracing traversal algorithm for kd-trees. In *2006 IEEE Symposium on Interactive Ray Tracing*, pages 57–60, 2006.

[33] Andrew T. Cable and Domenick J. Pinto. Dynamically balancing a binary tree (abstract). In *Proceedings of the 1990 ACM Annual Conference on Cooperation*, CSC '90, page 438, New York, NY, USA, 1990. Association for Computing Machinery.

[34] Sebastiano Battiato, Domenico Cantone, Dario Catalano, Gianluca Cincotti, and Micha Hofri. An efficient algorithm for the approximate median selection problem. In *Proceedings of the 4th Italian Conference on Algorithms and Complexity*, CIAC '00, page 226–238, Berlin, Heidelberg, 2000. Springer-Verlag.

[35] Shlomi Steinberg and Ling-Qi Yan. A generic framework for physical light transport. *ACM Trans. Graph.*, 40(4), jul 2021.

[36] Martin H. Weik. *Lambert's cosine law*, pages 868–868. Springer US, Boston, MA, 2001.

[37] Fred E. Nicodemus. Directional reflectance and emissivity of an opaque surface. *Appl. Opt.*, 4(7):767–775, Jul 1965.

[38] Cristian Lambru, Anca Morar, Florica Moldoveanu, Victor Asavei, and Alin Moldoveanu. Comparative analysis of real-time global illumination techniques in current game engines. *IEEE Access*, 9:125158–125183, 2021.

[39] G. Marsaglia. Generating discrete random variables in a computer. *Commun. ACM*, 6(1):37–38, jan 1963.

Appendix1 - Project Link

GitHub Link: https://github.com/LingLINKfeng/Monte-Carlo-path-tracing-algorithm-based-on-OpenGL

Appendix2 - Some Other Rendering Result

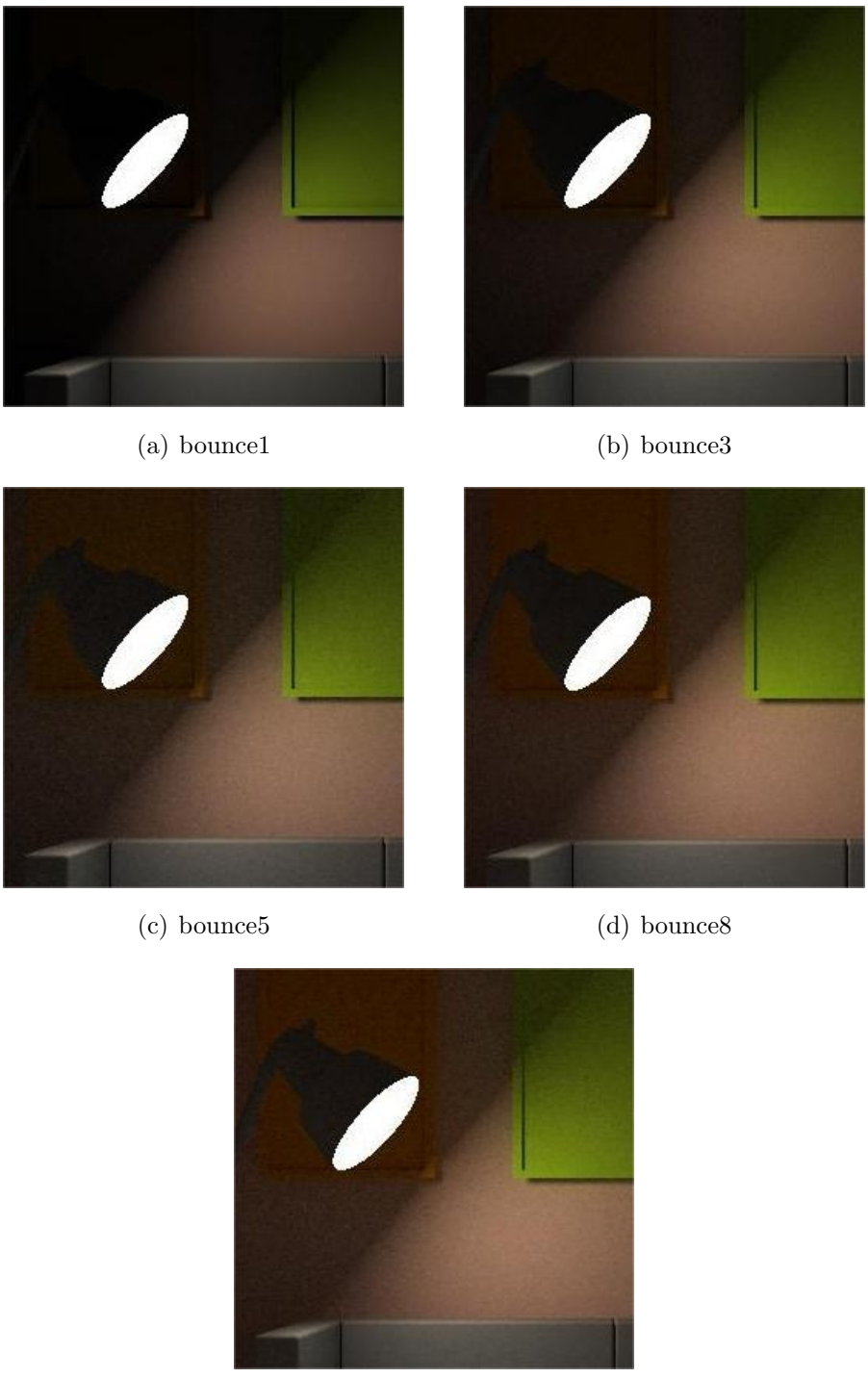
Figure 1: Mirror material model, SPP=1500,bounce 8 times, cost 100minutes

Figure 2: SPP=1000, bounce 8 times, cost 50minutes

Figure 3: SPP=1000, bounce 20 times, cost 120minutes

Figure 4: glass material model, SPP=500,bounce 8 times, cost 31minutes

Figure 5: SPP=200, bounce 8 times, cost 12minutes



(a) bounce1

(b) bounce3

(c) bounce5

(d) bounce8

(e) bounce15

Figure 6: Comparison of details under different light bouncing times