Monte Carlo Ray Tracing With Bounding Volume Hierarchy Tree Structure

Hantang Zheng, Chenxu Song, Hongqiang Cai, Jinqi Zhang
University of Southern California
hantangz@usc.edu, chenxuso@usc.edu, caihongq@usc.edu, jinqizha@usc.edu

Abstract

Ray tracing is a common method of image rendering other than rasterization. With ray tracing, we could produce images that can better demonstrate global illumination effects such as soft-edged shadows, refractions, and multiple reflections, effects that are hard to achieve via rasterization. However, the ray tracing algorithm also has its own problem, which is that the number of calculations needed for the test of light ray-object intersections could increase exponentially in a complex scene. We could improve on this using axis-aligned bounding boxes and a Bounding-Volume Hierarchy (BVH). A further improvement on the ray tracing algorithm is to remove the limit on the maximum number of light bounces and better simulate the real world physics of infinitely large number of light reflections, a variant of ray tracing known as path tracing. This paper will discuss our process of building a path tracing image renderer in three different stages: Whitted-style ray tracer, ray tracing with BVH acceleration, and Monte Carlo path tracing.

1 Introduction

Image rendering is a key step of the graphics pipeline, and a major topic in the field of computer graphics. As we hope to use computer graphics to create more photo-realistic reproductions of the real world for various purposes, such as modeling or gaming or special effects for films, more advanced rendering algorithms that can better simulate real life light transport processes are needed. Rasterization is a common rendering algorithm, and it can achieve good quality results in a computationally less expensive way. However, rasterization is limited because it could not simulate the effect of global illumination and natural phenomena such as light reflections and refractions. Shadows, reflections, and other qualities that we expect in a photo-realistic image could hardly be achieved via rasterization.

Hence, in order to achieve more realistic simulations of the world that we live in, a different rendering algorithm is in need. Our project investigates various ray tracing algorithms that render better quality images with more realistic global illumination effects compared to rasterization.

This paper will introduce three different stages of our quest for more photo-realistic images: the Whitted-style ray tracing; ray tracing with bounding volume hierarchy(BVH) for fast calculations; and lastly, the Monte Carlo path tracing algorithm.

2 Related Works

Ray tracing, as a method of building efficient global effects for rasterization, has been developed and used widely in many 3D renderings. As early as 1979, Whitted (1979) published the recursive ray tracing, calculate intensity by tracing the ray starting from each pixel recursively. The Whitted-style ray tracing enhances the quality of image by simulation of the real life light ray. However, the origin algorithm can be slow as its recursive nature produces enormous amount of calculations with increasing loop number.

Bounding volume hierarchy (BVH) is one of the accelerating algorithms that can be used to enhance the efficiency of rasterization methods. There has been considerable interest in the design of efficient, parallel algorithms for BVH for ray tracing(Gu et al., 2013). As the name suggests, BVH calculates bounding volume for each object and use it to decide whether a ray interacts with each object. By using axis-aligned bounding box, the calculation can be done fast and greatly shorten the time of ray tracing.

Monte Carlo method is a way of simulation by using random simulations to statistically estimate the result. In Monte Carlo path simulations, one is interested in the expected value of a quantity which is a functional of the solution to a stochastic differential equation (Giles, 2008). In ray tracing pro-

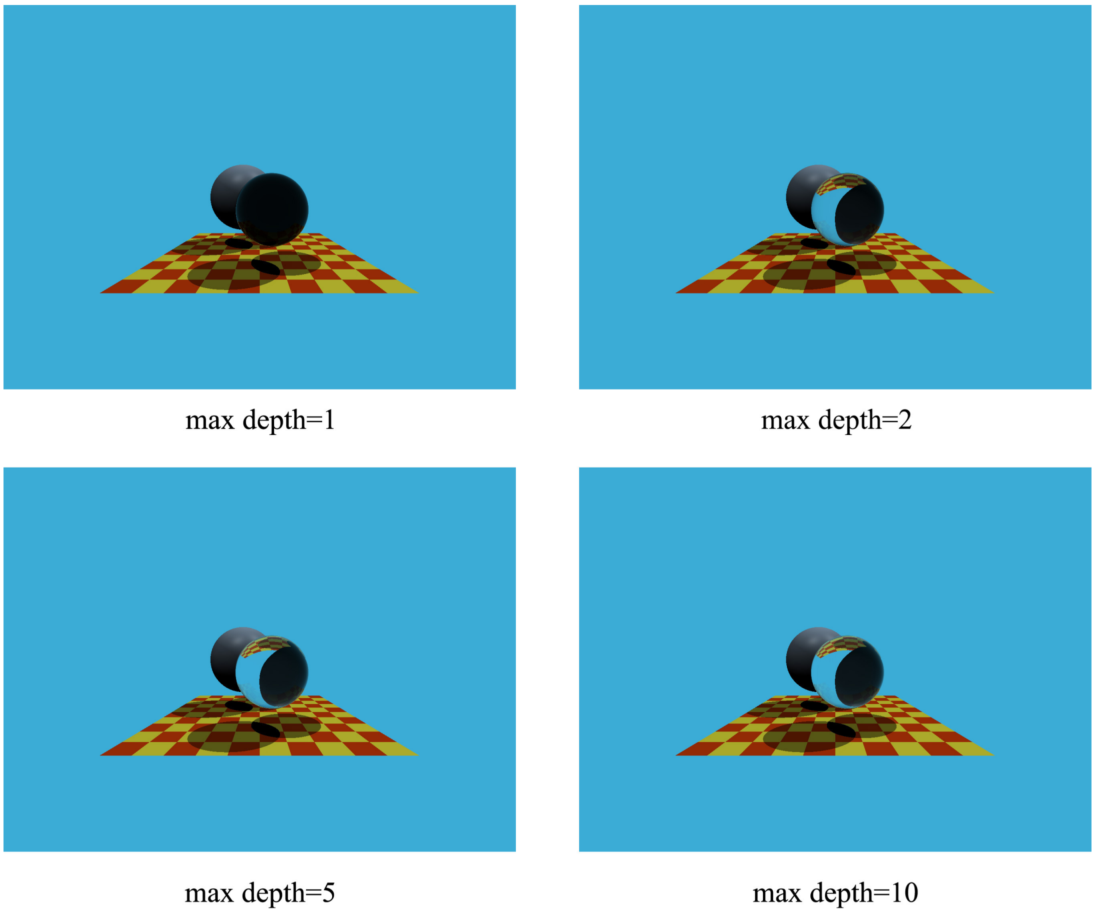

Figure 1: Result images using Whitted-style ray tracing with different max depth.

cess, the intensity of all the possible paths passing through a pixel can be modeled as a stochastic equation to use Monte Carlo simulation. This method can be applied to prevent exponential growth of possible rays of one pixel and make specific limit to the loops and the total time of a rasterization.

3 Our Works

We began our project by researching the underlying principles of ray tracing algorithms, as well as the various ray tracing algorithms that are commonly used today.

The resulting rendering program that we have developed for this project is built upon the skeleton code provided by professor Lingqi Yan of University of California, Santa Barbara(Lingqi, 2020). Based on this work, we added more functionalities that could simulate the physics of light rays such as reflection and refraction, and developed various ray tracing algorithms that we will discuss and compare later on.

Next, in the second stage of our project, we added the BVH structure to organize the triangle mesh of each object in the scene, reducing the number of calculations needed and thus accelerating the rendering time.

For the final stage of our project, since the Whitted-style ray tracing fail to handle global illumination correctly, we improved on the BVH-accelerated ray tracer to create a Monte Carlo path tracing program.

3.1 Whitted-Style Ray Tracing

Our first step of achieving more photo-realistic images compared to rasterization is to build a simple Whitted-style ray tracer as outlined in (Whitted, 1979). The Whitted-style ray tracing method differs from rasterization in that, instead of just calculating the color of each pixel once and for all based on the shading model, it takes global illumination effects such as reflections and refractions into account by generating light rays that simulate real life light physics. In a ray-tracing rendered image, the color of each pixel would be affected by reflection and refraction of light rays.

According to the real world physics of light, light rays are emitted from light sources, reflecting off the surfaces of objects and then traveling into human eyes. However, if we were to fully simulate real world physics via computer programs by tracing light rays from light sources to human eyes (i.e. the camera), an impractically large number of calculations are done while only a small fraction of these traced light rays would eventually enter the camera. This process known as forward ray tracing is thus disadvantaged compared to backward ray tracing, whereby the law of physics is reversed, and we trace light rays from the camera into the scene and eventually the light source instead. Backward ray tracing is a common method used in various ray tracing algorithms.

We trace a light ray from the camera into the scene via each pixel of the viewing plane, and the light ray would reflect or refract every time it hits the surface of an object in the scene. So each time a light ray hits an object surface, we compute the color of that surface under the light, then generate one or two new rays (reflection and refraction). We then recursively compute the color of new rays and sum all colors.

The reflection direction is determined by Snell‘s law and conservation of energy. Calculate the Fresnel term kr with index of refraction of object material and then combine reflection $kr \cdot L_i$ and refraction $(1 - kr) \cdot L_i$ to get result.

$$L_i = \underbrace{kr \cdot L_i}_{reflect} + \underbrace{(1 - kr) \cdot L_i}_{refract} \tag{1}$$

This process may potentially result in an arbitrarily large number of reflections and refractions, since a light path ends only when it hits the light source or an object with diffuse material. However, this also means that the end condition is very hard to reach, and the tracing process would take a huge amount of calculations and thus a long time. To make matters worse, for each ray generated, we have to test for intersection between each light ray and each triangle in the scene. Thus, the amount of calculations required will increase exponentially when the scene becomes more complex and the number of objects increases. In light of all these problems, we have to limit the maximum number of bounces that a light ray is allowed. We tested several maximum bounce number in a classic test scene using the Whitted-style ray tracer and found out 5 is the best.

In addition, to make rendering more realistic, a simple shadow mapping is added. Each time the light ray intersects with an object, a shadow ray is generated from the intersection towards the direction of the light sources. We then compare the distance from the intersection to the light source and the length of the shadow ray. If the length of the shadow ray is less than the intersection distance from the light source, it means that the shadow ray has encountered other objects along the way, which indicates that the shading point is in shadow as it is occluded from the light by other objects.

Max Depths	2 balls	100 balls
1	4	30
5	4	45
10	5	111
20	6	1665

Table 1: Running time(in seconds) of models with different depths and object numbers.

3.2 BVH Acceleration

The Whitted-style ray tracing can achieve excellent rendering effects for scenes that contain only simple geometric shapes. However, when trying to load a complex model with hundreds of triangular grids, the program fails to render the scene because of the vast amount of computation needed as Table 1 shows. Hence, for the second stage of the project, we want to accelerate the rendering process.

A sound and simple solution is to simplify and reduce the calculation needed for intersection tests via bounding boxes. Instead of testing intersection between a light ray and every triangle in an object's triangle-mesh, we can first test whether the light ray intersects the object's bounding box. We used Axis-Aligned Bounding Box (AABB) that are easy to implement and test intersection by a simple coordinate comparison.

In our program, we assign each triangle in an object's triangle-mesh a bounding volume, and we recursively process all triangles to find the minimum bounding box of the entire object. Then, we use the tree-like data structure bounding volume hierarchy (BVH) to recursively bi-partition the object's triangles until we get a single bounding box for each triangle. The partition strategy of building BVH tree is split the bounding box into two child boxes based on the longest axis of the parent bounding box. In detail, we first find the longest axis of the parent bounding box , sort all objects

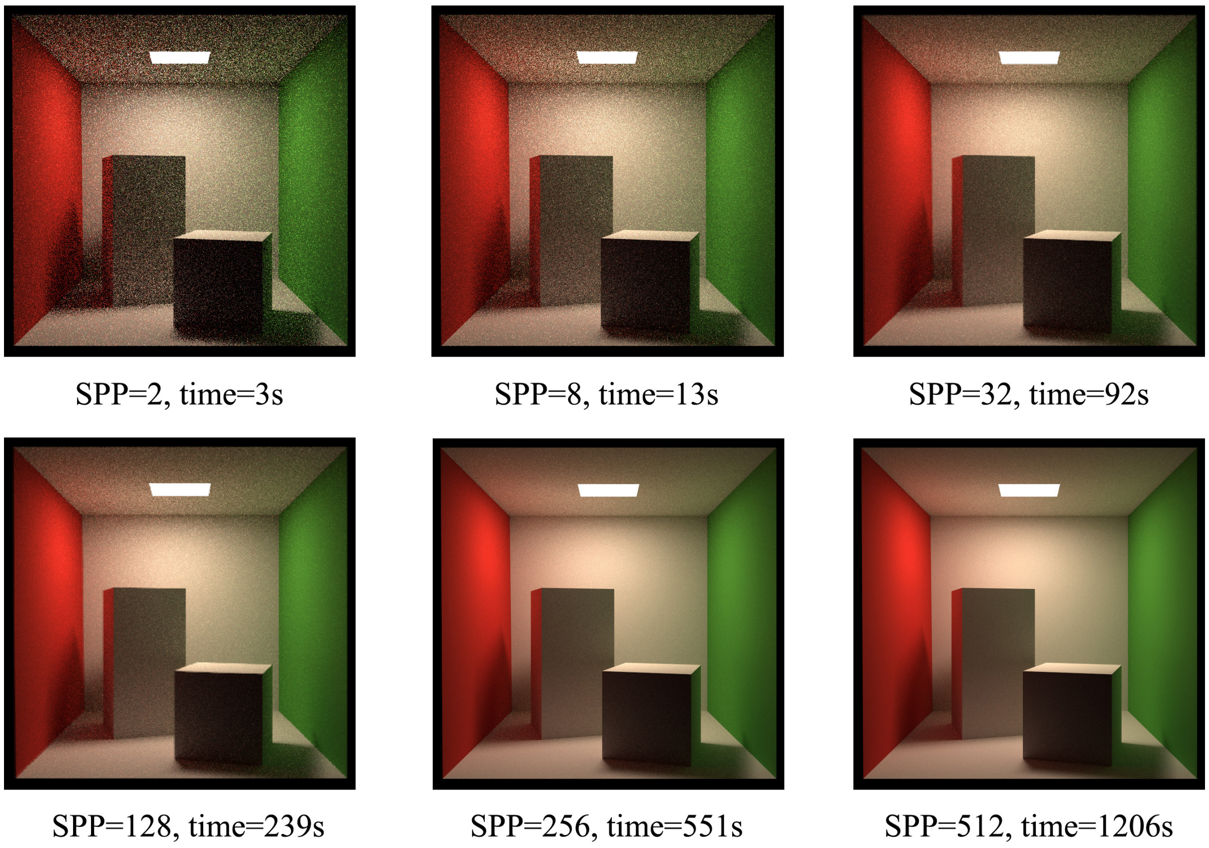

Figure 2: Image output using Monte Carlo path tracing, with different samples per pixel (spp).

with that axis, then split the set in the middle into two subsets.

The BVH data structure then accelerates the calculation for light ray-object intersection, as we now only have to test for the intersection with bounding box first. That can be easily done by traversing down the tree, testing intersection with children bounding boxes until reaching a leaf node with a single triangle. This binary-tree-like traversal speeds up the runtime from $O(N)$ to $O(\log N)$, with N being the number of triangles. BVH significantly simplifies the rendering process and reduces the number of calculations needed.

Same BVH tree is also applied to partition the all objects in the scene, thus no need for test intersection with every object.

In figure 3, an image output of 2 objects is tested for BVH acceleration. These two models are much more complex compared to the test scene for simple Whitted-style ray tracer without BVH, and the simple ray tracer could not handle them. With BVH acceleration, they can be rendered perfectly.

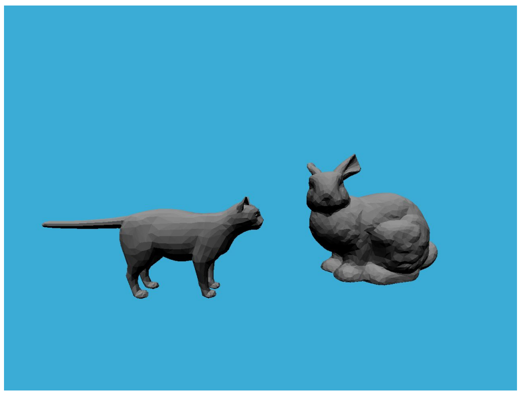

Figure 3: Image output using BVH acceleration.

3.3 Monte Carlo Path Tracing

So far, the Whitted-style Ray tracing still has flaws and there are three problems for simulating real world. First, the light source we used is a point light while in the real world there are area lights. Second, the limitation on the maximum light bounce time violates the conservation of light energy, since the light reflect and refract endlessly in the real world. Lastly, the reflection of diffuse material is ignored. To overcome these obstacles and further achieve photo-realism, we decided to improve our BVH-accelerated ray tracer by Monte Carlo path tracing.

In the render equation, the main difficulty for computation comes from the integral of radiance over the hemisphere. However, by introducing the Monte Carlo method which uses repeated random sampling to obtain numerical results, the integral can be solved by uniform sampling on the hemisphere(Matt Pharr, 2018). When ray hit an object, we sample N times on the hemisphere with each sample generating a ray. These rays are traced and averaged to get shading color.

After combining the Monte Carlo method, an ensuing problem is that each time a light bounces, N new rays are generated, thus the number of rays will increase exponentially with the number of light bounces. To solve the problem, instead of emitting one ray from a pixel and generating multiple sampling on each recursion, the program generates multiple rays from a pixel but only creates one sample for each recursion, preventing adding more rays.

Another issue we found is that the program treats direct illumination (light from source) and indirect illumination (light from reflection and refraction) equivalently, while direct illumination makes greater contribution to the shading point and need more weight in the shading result in general. To solve this problem, we calculate the direct illumination separately. Now when ray hit the object, the program will sample all light sources with Monte Carlo method individually and compute the color from light source before computing the indirect illumination.

The next task is sampling the light sources. As each single light source contains a triangle mesh, an algorithm is applied to select a random triangle from the BVH tree of the light source, then uniformly sample the selected triangle(Matt Pharr, 2018). For multiple light sources, we first sum the area of all light sources as $light_{area}$, generate a random number p within $[0, light_{area}]$, then traverse all the light sources in a loop, accumulate the area of light sources we visited. When the accumulated area is larger than P, we apply the sampling function to the current single light source.

The last step is to find an end condition for the recursion. Although the light bounce endless time in the real world, it is impractical to calculate the bounce endlessly. We then implemented a Russian Roulette algorithm to stop the light from bouncing endlessly. Before calculating the indirect illumination, the program randomly generates a number and compare it with a predefined Russian Roulette parameter P. Stop bouncing when the random number is larger than P; continue if it is smaller. The interesting facts is that in mathematics-wise, for a shading point color L_o in real, the expectation still L_o with Russian Roulette, meaning that no energy loss in this process. Thus, the Russian Roulette algorithm gives the recursion an end condition and simulate the real world ray transformation well.

$$E = P \times (L_o/P) + (1 - P) \times 0 = L_o \qquad (2)$$

4 Results & Conclusion

For testing the ray tracing simulation, we use a model of a non-transparent ball and a transparent ball. In figure 1, the result images using Whitted-Style ray tracing with max depth of 1, 2, 5 and 10 is shown. The comparison of result with max depth 1 and 2 shows the ray tracing through the transparent object. The reflection image in transparent ball also becomes brighter with max depth increase from 2 to 5 and 10, showing the effect of further ray reflection tracing.

With all the methods above combined, we use the classic Cornell Box model to evaluate Monte Carlo path tracing. The Cornell Box is an evaluative environment in which the Cornell University Program of Computer Graphics refined its radiosity rendering algorithms(Niedenthal, 2002). In figure 2, we make 6 image outputs with 2, 8, 32, 128, 256 and 512 samples per pixel. The improvement is obvious and show the effect of Monte Carlo path tracing.

Overall, our methods successfully create enhanced methods of raw ray tracing rasterization. However, there are still many places can be enhanced. Monte Carlo path tracing can only handle scene with area light, since sample on a point light is impossible. A potentially solution might be consider the point light as a tiny area light. Also Monte Carlo path tracing cannot be used in game development where real-time rendering is highly recommended, since takes more than 10 min to rendering a image with tolerable noisy. These are all possible topics for further work.

References

Michael B Giles. 2008. Multilevel monte carlo path simulation. *Operations research*, 56(3):607–617.

Yan Gu, Yong He, Kayvon Fatahalian, and Guy Blelloch. 2013. Efficient bvh construction via approximate agglomerative clustering. In *Proceedings of the 5th High-Performance Graphics Conference*, pages 81–88.

Yan Lingqi. 2020. Games 101: Introduction to computer graphics.

Greg Humphreys Matt Pharr, Wenzel Jakob. 2018. Physically based rendering.

Simon Niedenthal. 2002. Learning from the cornell box. *Leonardo*, 35:249–254.

Turner Whitted. 1979. An improved illumination model for shaded display. In *Proceedings of the 6th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '79, page 14, New York, NY, USA. Association for Computing Machinery.