Optimizing Ray Tracing

Jialu Xu
xuj14@rpi.edu
Rensselaer Polytechnic Institute
Troy, New York, USA

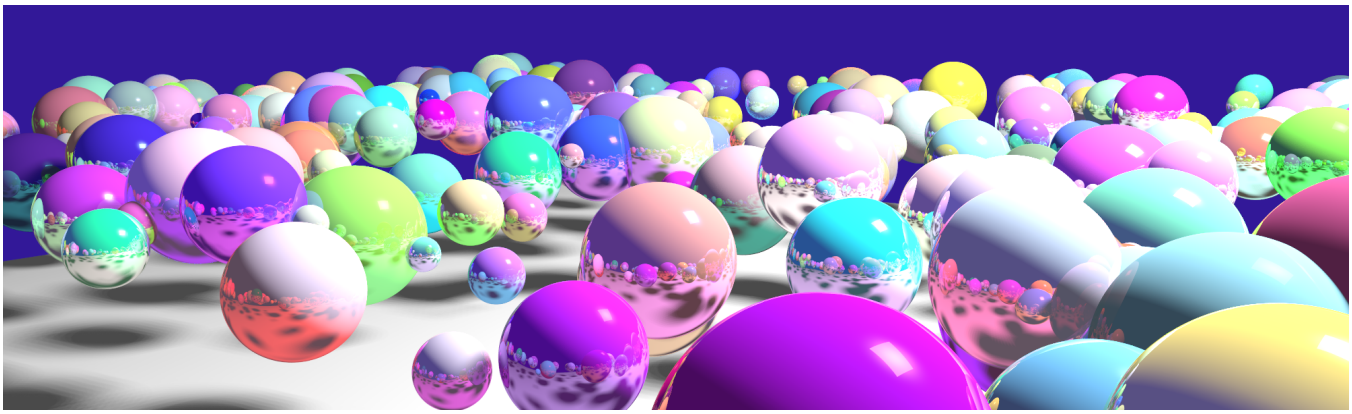


Figure 1: Ray Tracing of Randomly Generated Spheres.

ABSTRACT

This paper presents the optimization of a previously implemented naive recursive ray tracing algorithm by incorporating three distinct approaches: Russian Roulette, parallelization, and Bounding Volume Hierarchy (BVH). We discuss the implementation details and evaluate the performance benefits of each method, highlighting their effectiveness in reducing computational complexity and improving overall rendering times. By combining these optimization techniques, we demonstrate a significant enhancement in the efficiency of the ray tracing process, while maintaining the core visual fidelity and quality of the rendered images.

KEYWORDS

ray tracing, parallel, bvh

1 INTRODUCTION

Ray tracing has emerged as a popular method for rendering photo-realistic images in computer graphics. However, naive recursive ray tracing algorithms often suffer from high computational complexity, which leads to slow rendering times. This issue is especially noticeable on less powerful hardware, such as low-end laptops, where the performance limitations can severely impact the feasibility of ray tracing for real-time applications. Motivated by the need to optimize ray tracing performance on less capable devices, this paper presents three optimization techniques – Russian Roulette, parallelization, and Bounding Volume Hierarchy (BVH) – to improve the efficiency of a naive recursive ray tracing implementation.

By incorporating these methods, we aim to alleviate the performance constraints experienced on low-end hardware and broaden the range of devices capable of executing ray tracing algorithms efficiently.

2 RELATED WORK

In this paper, I implemented multiple techniques discussed in previous works used in my ray tracing algorithm.

2.1 Russian Roulette

Russian Roulette was introduced by Arvo and Kirk (1987)[1] as a method for reducing the computational complexity of global illumination algorithms. In their work, the technique was applied to a structure similar to photon mapping, rather than traditional ray tracing. My study aims to incorporate Russian Roulette into a naive recursive ray tracing algorithm and analyze its performance impact in terms of rendering times and image quality.

2.2 Parallelization

Parallelization has been a popular research topic in the realm of ray tracing optimization. OpenMP is a widely-used API that supports multi-platform shared-memory multiprocessing programming in C, C++, and Fortran. It has been instrumental in facilitating parallel processing for various applications, including ray tracing. In their work, Yviquel et al. (2023)[6] discussed the OpenMP cluster programming model, providing valuable insights into its use for parallelizing code. Building upon this foundation, my work focuses on parallelizing the loops and recursion in a naïve recursive ray tracing algorithm using OpenMP. I demonstrate the performance gains achieved through parallelization.

2.3 Bounding Volume Hierarchy

As early as 1980, Rubin and Whitted [2] proposed a hierarchy structure to accelerate the rendering process. Following this, Weghorst et al. [5] built the BVH using the modeling hierarchy. The Bounding Volume Hierarchy has since become one of the most common

acceleration structures in ray tracing. Wald et al. (2007) [3] introduced a fast and parallel construction of BVH using the Surface Area Heuristic (SAH). A decade later, Wald [4] revisited this topic in his blog, discussing potential improvements for modern-day applications and the use of BVH in real-time scenarios. I implement the basic BVH structure to the ray tracing algorithm and analyze its performance based on the number of primitives in the scene.

3 BACKGROUND AND PRIOR IMPLEMENTATION

Before discussing the optimization techniques, I briefly describe the key components of my prior implementation, which serves as the basis for the improvements presented in this paper.

Initially, I implemented a naive recursive ray tracing algorithm, which is a recursive function that loops through lights to trace shadows and recursively calls itself to handle bounced rays. The core of the ray tracing process involves casting rays through each pixel of the image and calculating the color contribution from the intersected objects and light sources.

To save the rendered images, I used a function that traces the rays for every pixel and saves the resulting image in the Portable Pixmap (PPM) format. The PPM format was chosen for its simplicity and ease of implementation.

To improve the quality of the rendered images and reduce aliasing artifacts, I employed a stratified jitter technique. This method generates a vector of uniformly distributed points before each sampling loop, which are then used to sample the scene.

Having provided this background information, I will now discuss the optimization techniques applied to enhance the performance of the ray tracing algorithm.

4 RUSSIAN ROULETTE

The Russian Roulette technique is an optimization method that aims to reduce the computational complexity of ray tracing by probabilistically terminating rays early in the tracing process. This technique involves the use of a probability p to determine whether a ray should continue being traced or be terminated early. When tracing a ray, if a random number generated within the range of [0, 1] is below p, ray tracing continues; otherwise, it terminates, effectively reducing the number of rays and intersections to be processed.

Algorithm 1 Russian Roulette

```
function TraceRay
    if rand(0, 1)< p then
        ans ←TraceRay/p
    else
        ans ← 0
    end if
end function
```

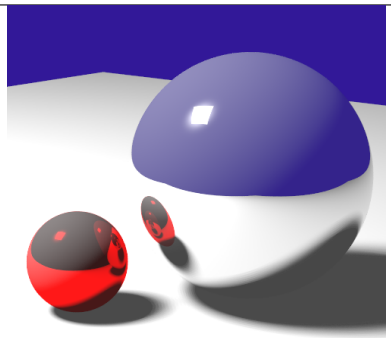
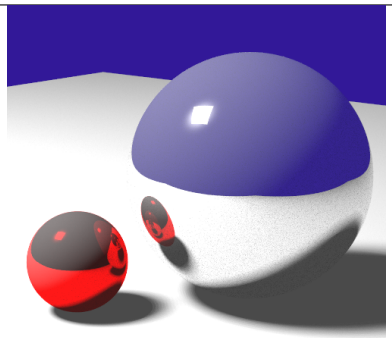
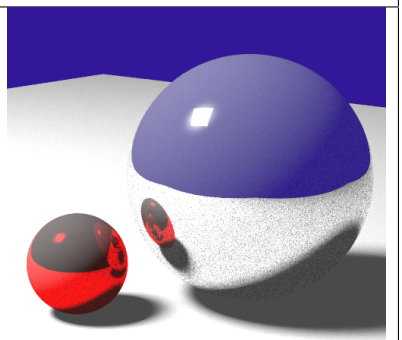
To ensure energy conservation when using Russian Roulette, the color contribution of each ray must be adjusted to account for the probabilistic termination. This is achieved by dividing the resulting color by the probability p. The energy conservation equation (Eq. 1) can be expressed as follows:

$$Color = p * (color/p) + (1 - p) * 0 \tag{1}$$

By applying this equation, we can maintain the overall illumination consistency in the scene even when rays are terminated early. However, as discussed in the next paragraph, this technique has certain limitations when applied to my ray tracing implementation.

Upon incorporating Russian Roulette into my ray tracing algorithm, I observed a reduction in rendering time, as shown in Table 1. As the probability p decreases, the rendering time is reduced. However, the image quality degrades significantly. This can be attributed to the fact that my ray tracing implementation only uses one ray per pixel. Consequently, if a ray is terminated early, there is no additional contribution to that pixel, resulting in the need for anti-aliasing to smooth out the image. In contrast, techniques such as photon mapping and path tracing that employ Russian Roulette typically have multiple contributions to a single pixel, so the energy conservation principle can still produce high-quality results.

Table 1: Russian Roulette Results

p	1.0	0.9	0.5
Time(sec)	309.451	314.264	234.966
Result			

Due to the significant impact on image quality, I decided not to use the Russian Roulette technique to optimize my ray tracing algorithm.

5 PARALLELIZATION

To further optimize my ray tracing algorithm, I employed parallelization using OpenMP, an API designed to support multi-platform shared-memory multiprocessing programming. By parallelizing the loops, I was able to achieve significant performance improvements. Additionally, I utilized the shared variable feature of OpenMP to prevent data races, ensuring the integrity of the parallel processing.

During the implementation of parallelization, I encountered a Segmentation Fault error. After researching the issue, I discovered that the problem stemmed from the use of `push_back()` and `rand()`, which can only be executed by a single thread at a time. To address this, I employed OpenMP's critical block feature, enabling these operations to run safely within the parallelized code.

The table below shows the rendering times for different numbers of threads:

Table 2: Parallelization Results

num thread	1	2	3	4
time (sec)	573.521	369.5	309.451	288.898

The results show a clear reduction in rendering time as the number of threads increases. In particular, when comparing the single-threaded execution to the 4-threaded execution, there is a substantial improvement in performance. However, the relative improvement in rendering time diminishes as the number of threads increases. This is due to the presence of shared blocks that can only be executed by one thread at a time, limiting the overall parallel efficiency.

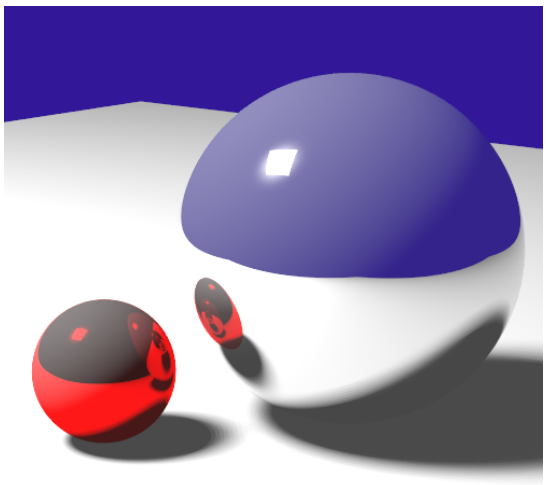


Figure 2: Parallel Rendering on Reflective Spheres

Additionally, the quality of the rendered images (Figure 2) remains consistent across all test cases, demonstrating that the parallelization effectively speeds up the rendering process without sacrificing image quality.

6 BOUNDING VOLUME HIERARCHY

The Bounding Volume Hierarchy (BVH) is a widely-used acceleration structure for ray tracing that aims to reduce the number of intersection tests required when rendering a scene. By organizing scene primitives into a tree-like hierarchy of bounding volumes, the algorithm can quickly cull portions of the scene that are not intersected by a ray, resulting in substantial performance improvements.

6.1 Construction

The construction of a BVH involves taking a vector of primitives and recursively building a tree-like structure with axis-aligned bounding boxes (AABBs) around these primitives. The algorithm starts by calculating the overall bounding box that encapsulates all the primitives in the scene. Once the initial bounding box is created, the algorithm proceeds to split the group of primitives into two smaller sets based on the largest dimension of the parent AABB. By sorting the primitives along the chosen dimension, the algorithm can efficiently divide the list into two equal parts.

The recursive splitting process continues, with each subsequent child node containing a smaller subset of primitives enclosed within its own bounding box. This process is repeated until each leaf node contains only one primitive.

The time complexity of constructing a BVH is $O(nlogn)$, primarily due to the sorting step involved in partitioning the primitives. Table 3 showcasing the construction times for different numbers of primitives demonstrates that the time required for BVH construction is quite minimal, even for large numbers of primitives. This low construction time, coupled with the substantial performance improvements provided during the ray tracing process, makes BVH an appealing choice for optimizing ray tracing algorithms.

During testing with 500×500 spheres, the program successfully loaded the meshes and constructed the BVH tree. However, when attempting to render the scene using OpenGL, a `Segmentation Fault` error occurred. To further investigate the issue, I tested the program with 100×100 spheres, which were successfully rendered, albeit with significant lag when moving the camera. Increasing the number of spheres to 200×200 resulted in a `GL_INVALID_VALUE` error, indicating that the maximum buffer size had been exceeded.

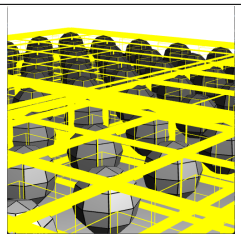
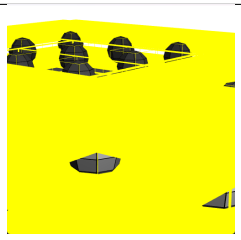
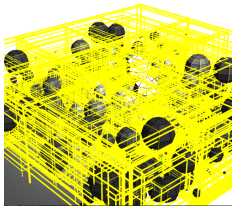
While it is possible that modifying the OpenGL connection scripts could resolve this issue, due to time constraints, I did not explore this solution further. Despite this limitation, the BVH construction process demonstrated significant performance improvements for the ray tracing algorithm.

6.2 Intersection Tests

The intersection test for BVH plays a crucial role in optimizing the ray tracing process by reducing the number of intersection tests required when rendering a scene. The test works by traversing the tree structure, starting from the root node and moving down to the

Table 3: BVH Construction Results

Sphere Count	20×20	50×50	500×500	20×20 (rand)
Time(sec)	0.0025	0.0226	6.1542	0.0027

Result			`[leaving updateVBOs]`
			`GL ERROR(0) GL_INVALID_VALUE`
			`...`
			`Segmentation Fault`
				

leaf nodes, checking the bounding boxes for intersection with the ray.

Initially, the algorithm checks whether the ray intersects the root node's bounding box. If there is an intersection, the algorithm proceeds to test the child nodes' bounding boxes. This traversal process is guided by a depth-first search strategy, which prioritizes visiting the closest bounding boxes to the ray origin. This ensures that the closest intersection point is found efficiently.

When the algorithm reaches a leaf node, it invokes the intersection function of the specific primitive stored in the node to determine if an actual intersection occurs. If an intersection is found, the algorithm updates the hit variable with the closest intersected primitive and the corresponding intersection point. This process allows the algorithm to discard irrelevant parts of the scene quickly and focus on the primitives that are most likely to contribute to the final image.

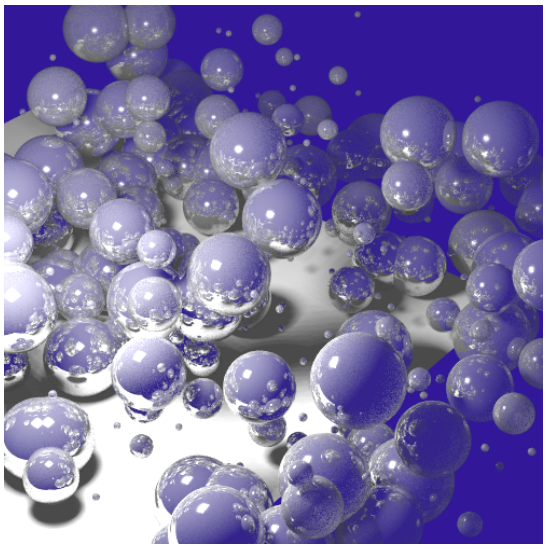


Figure 3: BVH Result

By exploiting the hierarchical structure of the BVH, the algorithm can significantly reduce the number of intersection tests needed, resulting in substantial performance improvements. I tested on a 20×20 randomly generated scene, the use of this acceleration structure can lead to dramatic reductions in rendering time. In the test case as shown in figure 3, the rendering time was reduced from 5.5 hours to just 2 hours, showcasing the effectiveness of BVH in optimizing ray tracing.

7 TEST ENVIRONMENT

The testing setup is as follows:

- Hardware: The experiments were conducted on a low-end laptop equipped with a Ryzen 3 3350U processor. This choice of hardware highlights the potential performance improvements achievable on modest systems.
- Ray tracing parameters: The ray tracing algorithm was configured with the following settings: 50 bounces, 50 shadow samples, and 16 anti-aliasing samples. These values were selected to provide a balance between image quality and computation time.
- Thread count: For the BVH and Russian Roulette testing, three threads were used to speed up the testing.
- Scene for Russian Roulette & Parallel testing: A simple scene consisting of two reflective spheres was used for testing the Russian Roulette and Parallel optimization techniques.
- Scene for BVH testing: A more complex scene containing 400 randomly positioned and sized spheres was employed to evaluate the BVH optimization technique. This scene was chosen to showcase the performance improvements provided by the BVH acceleration structure when dealing with larger numbers of primitives.

8 FUTURE WORKS

There are many I didn't implement for this paper/project:

(1) SAH-based BVH construction: One possible improvement to the BVH construction process is to utilize the Surface Area Heuristic (SAH) when splitting the primitives. SAH-based approaches have been shown to produce better tree structures by minimizing the expected cost of ray traversal through the hierarchy. This can lead to even faster intersection tests and, ultimately, shorter rendering times.
(2) OpenGL code optimization: To handle larger scenes, the OpenGL code could be modified to better manage the allocation and utilization of buffers. This may involve employing techniques such as dynamic buffer allocation, texture atlases, or instancing to reduce the memory footprint and

improve rendering performance for scenes with high primitive counts.

(3) GPU-based ray tracing: Another promising direction for future work is to leverage the power of the GPU for ray tracing computations. Modern GPUs offer immense parallel processing capabilities that can significantly accelerate ray tracing performance. By implementing a GPU-based ray tracing algorithm, we can exploit the inherent parallelism of the ray tracing process and achieve substantial performance improvements.

ACKNOWLEDGMENTS

To Professor Barb Cutler, for providing the code for connecting the ray tracing algorithm to OpenGL for rendering.

REFERENCES

[1] James Arvo and David Kirk. 1990. Particle Transport and Image Synthesis. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques* (Dallas, TX, USA) *(SIGGRAPH '90)*. Association for Computing Machinery, New York, NY, USA, 63–66. https://doi.org/10.1145/97879.97886

[2] Steven M. Rubin and Turner Whitted. 1980. A 3-Dimensional Representation for Fast Rendering of Complex Scenes. *SIGGRAPH Comput. Graph.* 14, 3 (jul 1980), 110–116. https://doi.org/10.1145/965105.807479

[3] Ingo Wald. 2007. On fast Construction of SAH-based Bounding Volume Hierarchies. In *2007 IEEE Symposium on Interactive Ray Tracing*. 33–40. https://doi.org/10.1109/RT.2007.4342588

[4] Ingo Wald. 2022. Parallel BVH Construction. https://ingowald.blog/2022/04/20/parallel-bvh-construction/. Accessed on 21 Mar. 2023.

[5] Hank Weghorst, Gary Hooper, and Donald P. Greenberg. 1984. Improved Computational Methods for Ray Tracing. *ACM Trans. Graph.* 3, 1 (jan 1984), 52–69. https://doi.org/10.1145/357332.357335

[6] Hervé Yviquel, Marcio Pereira, Emílio Francesquini, Guilherme Valarini, Gustavo Leite, Pedro Rosso, Rodrigo Ceccato, Carla Cusihualpa, Vitoria Dias, Sandro Rigo, Alan Souza, and Guido Araujo. 2022. The OpenMP Cluster Programming Model. In *Workshop Proceedings of the 51st International Conference on Parallel Processing*. ACM. https://doi.org/10.1145/3547276.3548444