Raytracer Optimizations

Nishk Patel
University of Illinois at Urbana Champaign
Champaign, United States
nishkdp2@illinois.edu

Sunskar Dhanker
University of Illinois at Urbana Champaign
Champaign, United States
dhanker2@illinois.edu

Abstract

Ray tracing is a computationally intensive technique for generating realistic images by simulating light behavior. This project optimizes an existing ray tracer to improve performance while preserving visual accuracy. Using profiling tools like perf and flame graphs, we identify bottlenecks and apply optimizations, including memory enhancements, parallel processing, and algorithmic improvements. Focusing on three test scenes (a piano room, a rotating globe, and a rotating sphere), we maximize rendering speed and resolution. Both the Flamegraph in figure 1 as well as most of the benchmarks in this paper will refer to the Piano scene: `./main.exe -i inputs/pianoroom.ray –ppm -o output/pianoroom.ppm -H 500 -W 500`
We document successful and ineffective optimizations, validate performance gains through benchmarking, and ensure reproducibility via Docker.

Github Repository

ACM Reference Format:
Nishk Patel and Sunskar Dhanker. 2018. Raytracer Optimizations. In *Proceedings of* . ACM, New York, NY, USA, 3 pages. https://doi.org/XXXXXXX.XXXXXXX

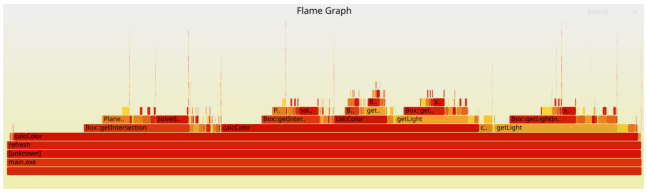

Figure 1. perf Flamegraph

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

In this project, we analyze and optimize an existing ray tracer to improve efficiency without sacrificing visual fidelity. Using profiling tools such as `perf` and flame graphs, we identify key bottlenecks and implement targeted optimizations to enhance memory management, parallel execution, and algorithmic efficiency.

Our optimizations focus on several core areas:

- **Memory Management**: The original implementation frequently allocated and copied memory inefficiently, leading to fragmentation and excessive cache misses. We optimize this by using dynamic resizing techniques with `realloc`, reducing overhead and improving performance.
- **Compiler Optimizations**: By leveraging compiler flags such as `-O3`, we enable aggressive optimizations like function inlining, loop unrolling, and SIMD vectorization, significantly improving execution speed.
- **Parallelization**: We introduce OpenMP to parallelize core computations, allowing multiple CPU cores to work concurrently, thereby accelerating the rendering process.
- **Bounding Volume Hierarchy (BVH)**: We implement a BVH to optimize ray-object intersection tests. This spatial acceleration structure reduces the complexity from $O(n)$ to $O(\log n)$, dramatically decreasing the number of intersection tests required per ray and significantly improving rendering performance.

2 Memory Modifications

Efficient memory management is critical in computationally intensive applications like ray tracing, where millions of intersections may need to be processed per frame. How memory is allocated and resized directly impacts performance, affecting execution speed and memory footprint.

Before the changes, the program allocated a new array every time an intersection is found:

```
malloc(sizeof(TimeAndShape) * (seen + 1));
```

Mallocations and deallocations result in fragmentation, and frequent memory allocation severely bottlenecks performance, reducing the number of rays processed per second. Also, excessive memory copying impacts L1/L2 cache usage, leading to slower rendering delays.

To account for this, we start with a fixed capacity and double its size only when necessary using `realloc`. The

time complexity becomes an amortized $O(1)$ for each new intersection storage operation. This process expands the existing memory block when possible, reducing unnecessary memory copying.

```
if (seen == capacity) {
    capacity *= 2;
    TimeAndShape* newTimes = (TimeAndShape*)
                             realloc(times,
                             sizeof(TimeAndShape)
                             * capacity);
    if (!newTimes) {
        free(times);
        return;
    }
    times = newTimes;
}
```

As a result of these optimizations, the execution time improved from 2.446816 seconds to 1.830028 seconds.

3 Compiler Optimization

When compiling a ray tracer with the `-O3` flag, the compiler applies aggressive optimizations that directly impact performance in several ways:

- **Function Inlining for Performance**
- **Loop Unrolling for Faster Traversals**
- **Vectorization for SIMD Execution**
- **Floating-Point Algebraic Simplifications**

To make this adjustment, we simply changed our makefile's (`./`, `./src`, `./src/textures`) to use the following flags:

```
FLAGS := -O3 -lm -g -Werror
```

4 Parallelization

The use of OpenMP to parallelize the `refresh()` function can significantly speed up the ray tracing algorithm by utilizing multiple CPU cores to process independent tasks concurrently.

```
#pragma omp parallel for
for(int n = 0; n < H * W; ++n) {
    Vector ra = c->camera.forward +
                ((double)(n % W) / W - .5) *
                ((c->camera.right))
              + (.5 - (double)(n / W) / H) *
              ((c->camera.up));
    calcColor(&DATA[3 * n], c,
    Ray(c->camera.focus, ra), 0);
}
```

We used OpenMP to parallelize the pixel computation in ray tracing, boosting rendering speed, especially for high-resolution images. This approach efficiently leverages multi-core processors and sets the stage for further optimizations in real-time ray tracing.

Given that the system has 4 CPU cores, the theoretical maximum speedup should be approximately $\frac{1}{4}$ of the original execution time. However, due to overhead from thread management, maximum computational depth, and other inefficiencies, the actual speedup observed was less than ideal. The execution time improved from 0.712313 seconds to 0.375353 seconds, which is a speedup factor of approximately 1.9 instead of the expected 4.

5 Bounding Volume Heiarchy

A Bounding Volume Hierarchy (BVH) is a spatial acceleration structure that optimizes ray-scene intersection tests by organizing scene geometry into a hierarchical tree structure.

5.1 How BVH Works

The BVH organizes scene objects into a binary tree structure. During construction, the algorithm recursively splits the scene objects into two groups, generating a tree. Each node in the tree contains a bounding box that encompasses all the geometry beneath it. Leaf nodes hold the actual scene objects (typically a small number, like 1-4), while internal nodes are used for efficient traversal.

5.2 Technical Advantages

The main advantage of using a BVH is improved performance. Without a BVH, testing a ray against the entire scene requires checking each object individually, resulting in $O(n)$ complexity. With a BVH, only objects along the traversal path need to be checked, which reduces the complexity to $O(\log n)$ on average. For a scene with 1000 objects, this means testing only 10 objects instead of all 1000.

Another key advantage is improved memory coherence. Objects that are spatially close in the scene are typically grouped together in the BVH, leading to better cache utilization during traversal. The structure adapts to scene geometry density—areas with more objects undergo more frequent subdivision.

5.3 Technical Disadvantages

However, there are trade-offs. Constructing the BVH takes $O(n \log n)$ time, and the structure itself requires additional memory, proportional to the number of objects. The quality of the BVH heavily depends on how objects are split during construction; poor splitting can lead to unbalanced trees, negatively impacting performance.

5.4 Performance Impact

The BVH improves performance by reducing intersection tests per ray from thousands to dozens, leading to rendering speedups of 10-50x. Although it increases memory usage (about 2x) and setup time, these trade-offs are valuable for non-trivial scenes. The BVH quickly eliminates irrelevant parts of the scene, allowing the renderer to focus on key

areas, enabling efficient rendering of complex scenes with many intersections.

5.5 Our Work: BVH Implementation

In our implementation, we created a BVH class that encapsulates the Axis-Aligned Bounding Box (AABB). This class facilitates the efficient organization of scene geometry into the BVH structure. To handle different types of shapes, we developed a ‘getBounds‘ method for each shape to calculate its bounding box, which is then used for constructing the BVH. Furthermore, we replaced the linear array search algorithm in the ‘calcColor‘ function with a more efficient search algorithm that traverses the BVH. This modification significantly speeds up ray-object intersection tests, contributing to a considerable performance enhancement in our rendering pipeline. In our implementation, there seems to be some BVH bugs we were unable to squash that leave artifacts of rays which miss some objects and cause minor visual differences.

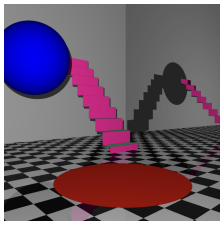

Figure 2. Original Piano

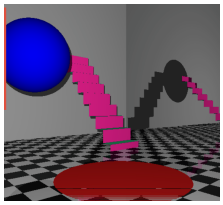

Figure 3. BVH Piano

6 Optional Speedups with Interleaved Ray Computation

An optional optimization to speed up ray tracing involves sending rays at specific intervals, which reduces computations and introduces motion blur. For example, odd-indexed rays can be computed in one frame and even-indexed rays in the next, or different interleaves, such as computing rays twice every three frames, can be used.

The following code snippet demonstrates how this can be implemented:

```
void refresh(Autonoma* c, BVH* bvh, int frame){
   ...
      // Check if the pixel is in the odd or
      even set based on the frame number
      if ((n % 2 == frame % 2)) {calculate ray}
   }
}
```

The key idea here is to process only every other pixel based on the current frame number.

- If `frame % 2 == 0`, only even-indexed pixels (0, 2, 4, ...) are processed.
- If `frame % 2 == 1`, only odd-indexed pixels (1, 3, 5, ...) are processed.

7 Results

The results show significant performance improvements for both the "Globe" and "Elephant" models. For the "Globe" model, optimizations in Memory Efficiencies and BVH reduce processing time from 144.65 seconds to 115.59 seconds and 8.56 seconds, respectively. Similarly, the "Elephant" model (actually the sphere model), which originally crashed, improves to 33.96 seconds with optimizations, and the BVH optimization reduces it further to 0.59 seconds. These results highlight the effectiveness of the optimizations, particularly the BVH and Memory Efficiencies, in improving computational efficiency and stability.

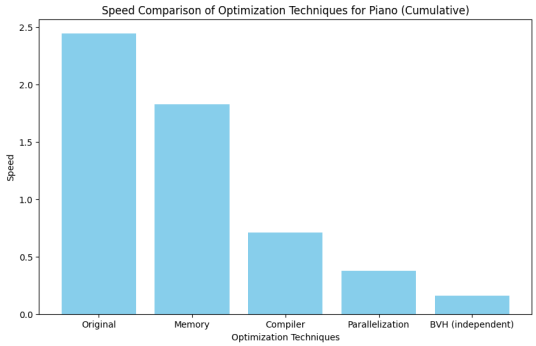

Figure 4. Performance Comparison for the Piano Model: Original vs Optimized (Memory Efficiencies and BVH).

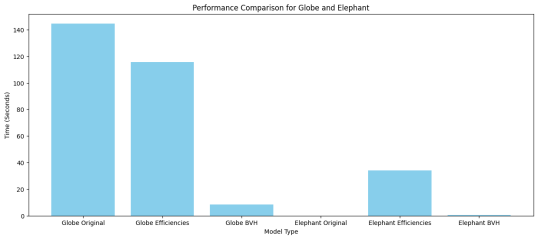

Figure 5. Performance Comparison for the Movie Models: Original vs Optimized (Memory Efficiencies and BVH).

8 Conclusion

In conclusion, we successfully optimized an existing ray tracer by focusing on memory management, compiler optimizations, parallelization, and the implementation of a Bounding Volume Hierarchy (BVH). These improvements led to significant performance gains, particularly in rendering speed and computational efficiency. While some minor visual artifacts remain due to BVH implementation issues, the overall optimizations resulted in faster rendering times, demonstrating the effectiveness of these approaches for large-scale, computationally intensive tasks.