Raytracer Optimizations

Nishk Patel
nishkdp2@illinois.edu
University of Illinois, Urbana Champaign
Urbana, Illinois, USA

Sunskar Dhanker
dhanker2@illinois.edu
University of Illinois, Urbana Champaign
Urbana, Illinois, USA

I. Introduction

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.

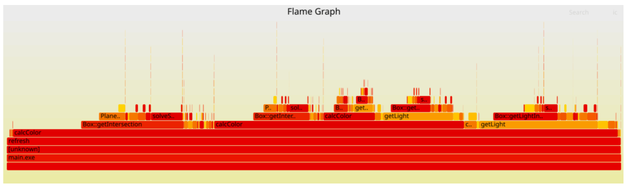


Figure 1: perf Flamegraph

II. Memory Efficiencies

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; }
```

III. Compiler Optimizations

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

1. Function Inlining for Performance
2. Loop Unrolling for Faster Traversals
3. Vectorization for SIMD Execution
4. Floating-Point Algebraic Simplifications

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

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

IV. 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.

V. Bounding Volume Hierarchy

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

How BVH Works:

The BVH organizes scene objects into a hierarchical tree structure. During construction, it starts with all scene objects and recursively splits them into two groups, creating a binary tree. Each node in the tree contains a bounding box that encompasses all geometry beneath it. Leaf nodes contain the actual scene objects (usually a small number like 1-4), while internal nodes are used for traversal.

Technical Advantages:

The primary advantage is performance. Without a BVH, testing a ray against a scene requires checking every object (O(n) complexity). With a BVH, you only need to check objects along the traversal path (O(log n) complexity on average). For a scene with 1000 objects, this might mean testing only 10 objects instead of all 1000.

The BVH also provides good memory coherence. Objects that are spatially close in the scene tend to be grouped in the tree, which leads to better cache utilization during traversal. The structure adapts to scene geometry density - areas with more objects get subdivided more times.

Technical Disadvantages:

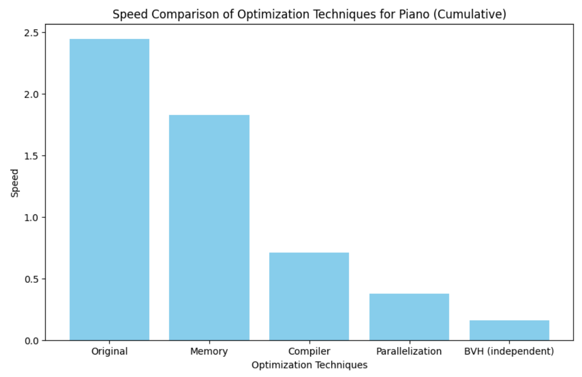
There is overhead in both memory and construction time. Building the BVH takes O(n log n) time, and the structure requires additional memory (roughly proportional to the number of objects). The quality of the BVH depends heavily on how objects are split during construction - poor splitting can lead to unbalanced trees and worse performance.

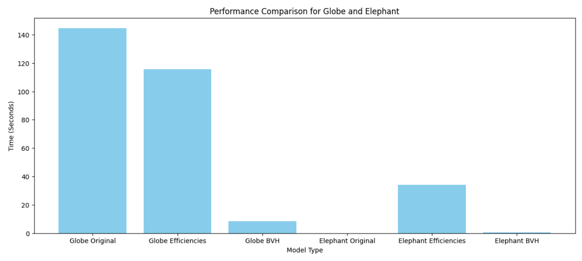
Performance Impact:

The BVH can dramatically improve performance by reducing the number of intersection tests per ray from thousands to dozens, resulting in rendering speedups of 10-50x or more. Though it increases memory usage (roughly 2x) and setup time, these trade-offs are usually worthwhile for non-trivial scenes. The BVH is especially effective for ray tracing, as it quickly eliminates large portions of the scene, allowing the renderer to focus only on relevant areas. This makes it possible to render complex scenes with thousands or millions of objects in a reasonable time.

VI. Results

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





VII. Conclusion

While we have made significant strides in optimizing our ray tracer, there are still avenues for further improvement:

1. Explore more advanced BVH construction algorithms to optimize tree balance and traversal efficiency.

2. Investigate GPU acceleration techniques to leverage parallel processing capabilities of modern graphics hardware.

3. Implement adaptive sampling techniques to focus computational resources on areas of the image with high detail or complexity.

In conclusion, this project has successfully enhanced the performance of our ray tracer through a combination of low-level optimizations and high-level algorithmic improvements. The resulting optimized ray tracer not only renders scenes faster but also scales better with scene complexity, paving the way for more realistic and intricate 3D visualizations.