Mini-Paper 1: Raytracer

Louis Asanaka
Ishaan Mathur

Abstract

To optimize this raytracer, we worked as a pair to iteratively test out our ideas after rounds of profiling. We proposed optimizations and worked on our individual branches to implement them. For testing, we relied on pprof's flamegraph and stat interface to identify bottlenecks. We continuously evaluated our optimizations on the Globe, Pianoroom, and Elephant scenes to confirm our hypotheses. We used a mix of approaches discussed in class such as hoisting, short-circuit evaluation, caching, parallelization, compiler flags, and building data-structures to improve performance. We noticed the most significant speedups came from passing the `-O3` aggressive optimization flag for `clang++` and using OpenMP to parallelize computation-heavy loops in the code. Overall, we reduced the runtime of Pianoroom by a factor of 17.5 and Globe by a factor of 9.3.

1 Optimizations

1.1 None

Figure 1. Flamegraph (pianoroom) before optimizations

The basic scene with little geometry, `pianoroom`, ran at around 2.22 s.

1.2 Clang & `-O3`

Figure 2. Flamegraph (pianoroom)

Not much to explain, `src/` was not compiled with optimizations. We are now at a 1.07 s for the same scene!

1.3 Pass-by-reference + const

Figure 3. Flamegraph (pianoroom)

Now at 0.89 s. Currently, the raytracer needlessly copies a lot of `Vectors` and `Rays` by value in arguments. We turn them into const references to avoid a copy. Not much performance is gained, likely due to the compiler eliding the unnecessary copies already.

1.4 `calcColor` sort → min

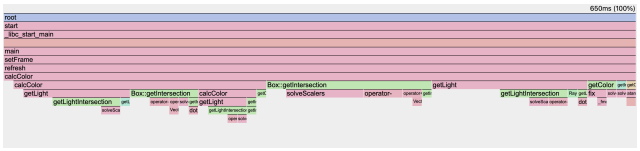

Figure 4. Flamegraph (pianoroom)

Now at 0.65 s. From the last flamegraph, we see that `calcColor` takes an unreasonable amount of time for itself outside of its calls. It tries to find the closest shape, yet fails miserably by performing repeated re-allocations in a loop before sorting the list of all shapes. We removed the allocations and turned the find-min into a linear scan. This optimization would aid a scene with many more shapes more than a simple one.

1.5 `solveScalers` caching

Figure 5. Flamegraph (pianoroom)

Now at 0.55 s. From the last flamegraph, we see that lots of time is spent in `solveScalers` when looking for an intersection. The code recomputes constants derived from 3 static vectors, `right`, `up`, and `vect`. We can cache these values and compute them on initialization.

1.6 Inlined Vector functions

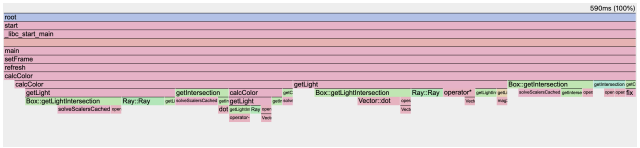

Figure 6. Flamegraph (pianoroom)

Now at 0.59 s. The flamegraphs suggested that the `Vector` functions were not being inlined, so we assumed that there is overhead when performing basic operations. However, profiling proved us wrong—it seems like the compiler is already doing that for us, and ignoring our hints. Digging

into the assembly would be good here, but we were short on time.

1.7 Use `std::vector` to store Shapes

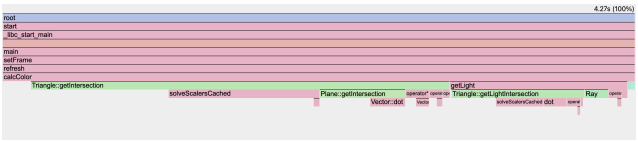

Figure 7. Flamegraph (elephant)

There is no advantage to using a linked list over a vector here, as there are only benefits to cache locality when we iterate over the vector of Shapes. We profiled this optimization for elephant.ray, since improvements are clearer with a larger list of shapes.

1.8 Improved Ray-Triangle Intersection

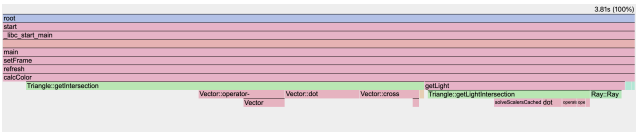

Figure 8. Flamegraph (elephant)

From 4.27 s to 3.81 s. With the elephant render, we are clearly spending all our time finding intersections now. This points to optimizing the ray-triangle algorithm as being low-hanging fruit. With some research, we replaced the current algorithm with the Möller–Trumbore intersection algorithm, which reduces the number of arithmetic operations needed.

1.9 Parallel with OpenMP

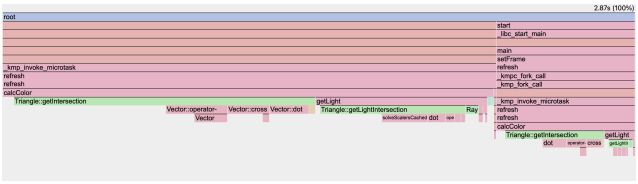

Figure 9. Flamegraph (elephant)

Now at 2.87 s. Another low-hanging fruit would be to use all our cores on the VM, since ray tracing is an embarassingly parallel problem, especially on the CPU with random access memory! Simply parallelizing the `refresh` significantly improves the performance.

1.10 Caching animation file data + minor changes

Up until this point every optimization we made did not take advantage of the fact that video data does not need to perform the same redundant computation on each frame. Namely, in `setFrame` the `animateFile` was being open and parsed every time. Because this logic was invariant to each frame and only needed to be computed once per animation, we created a `AnimationData` data structure to hold the animation data, and read from the data structure in `setFrame`.

1.11 `Triangle::getLightIntersection` early exit

This optimization did not really help, as we misinterpreted the function call source (there was not much of a need for early exiting).

1.12 Swap out `Vector` for `glm::dvec3`

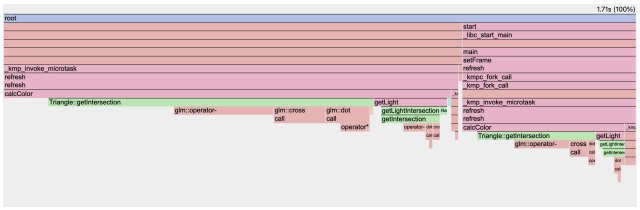

Figure 10. Flamegraph (elephant)

Why use our own `Vector` class when we can use a production-ready one from OpenGL Math, with SIMD intrinsics to optimize the basic operations? This assumes that we are bottlenecked by our vector operations, which is likely and supported by the flamegraphs.

1.13 Use `-O2` & `-march=native`

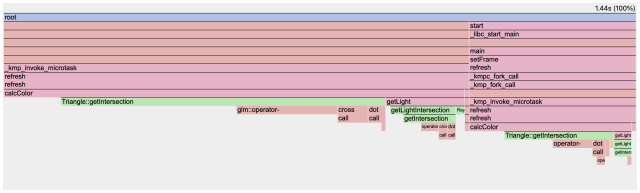

Figure 11. Flamegraph (elephant)

Finally, this is just to ensure our compiler knows which extensions (like AVX2) it can use to better optimize our code. Not too much of a difference.

1.14 Other

For animation, we noticed the entire scene is rerended every frame. We hyopthesized that for some frames, no objects changed, so we could cache the previous frame and avoid recomputation. However, after looking at the `setFrame` logic, we noticed that camera variables such as pitch, yaw, roll, and more were directly dependent on the frame number. This meant that each frame would alter the scene and our approach would not be caching any frames. We also noticed that each frame could be computed independently because there was no dependency between them, but we ran out of time to implement this approach.

2 Review

In summary, we were able to aggressively optimize our ray-tracer through a variety of approaches, each targeting different aspects of performance. We reduced the runtime of

Pianoroom by a factor of 17.5 and Globe by a factor of 9.3. Our work can be broadly categorized into several key areas:

- **Compiler Optimizations:** By compiling with `-O3` and later refining our flags with `-O2` & `-march=native`, we witnessed dramatic reductions in runtime. These changes highlight the importance of letting the compiler aggressively optimize the code.
- **Algorithmic Improvements:** Changing the implementation of the `calcColor` function from sorting to performing a simple linear scan for the minimum allowed us to eliminate unnecessary allocations. This small change resulted in significant speedups, emphasizing that even a minor algorithmic refinement can have large performance benefits.
- **Data Structure Modifications:** Switching from linked lists to `std::vector` for storing Shapes reduced overhead and improved data locality, contributing to improved cache performance.
- **Code-Level Enhancements:** Techniques such as passing arguments by reference (and marking them as `const` when appropriate) and inlining vector functions reduced redundant copies and function call overhead.
- **Parallelization:** The introduction of OpenMP allowed us to parallelize compute-intensive loops for scenes like `Globe` and `Pianoroom`.

2.1 Areas for Improvement

While our optimizations yielded impressive results, there are several areas for further optimization:

- **Further Parallelization:** Although OpenMP provided significant speedups, we did not use it to parallelize rendering multiple frames in an animation.
- **Memory Footprint and Cache Utilization:** Profiling the application with tools that specifically monitor cache performance could help us better understand the memory behavior and optimize data structures accordingly.
- **BVHs:** Instead of iterating over all shapes, storing them in a spatial data structure like a BVH could significantly improve performance, no need to loop through all the

Our code can be found at https://github.com/imathur1/598APE-HW1.

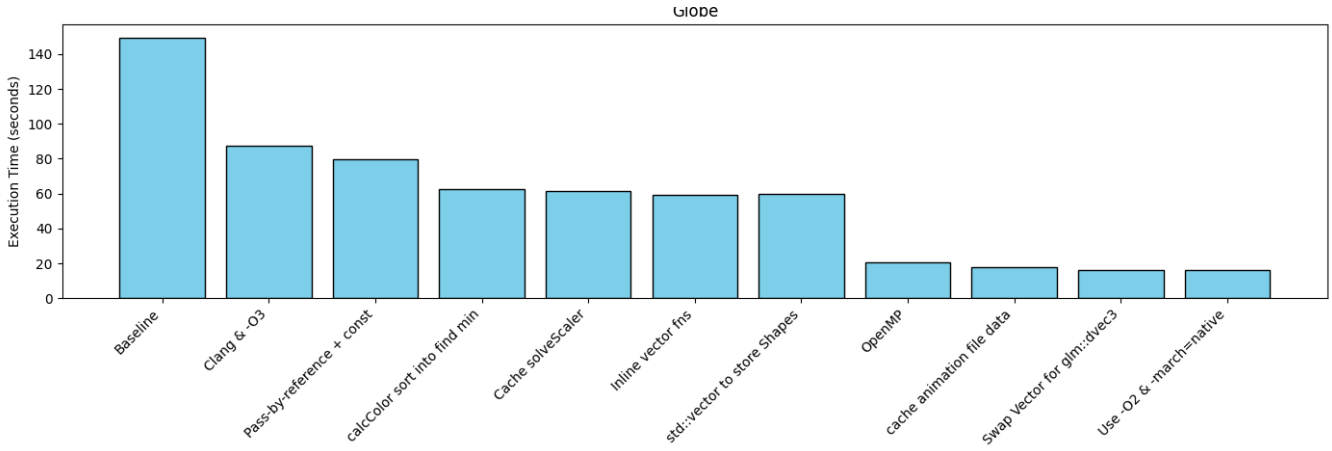

Figure 12. Execution time of Globe rendering with optimizations

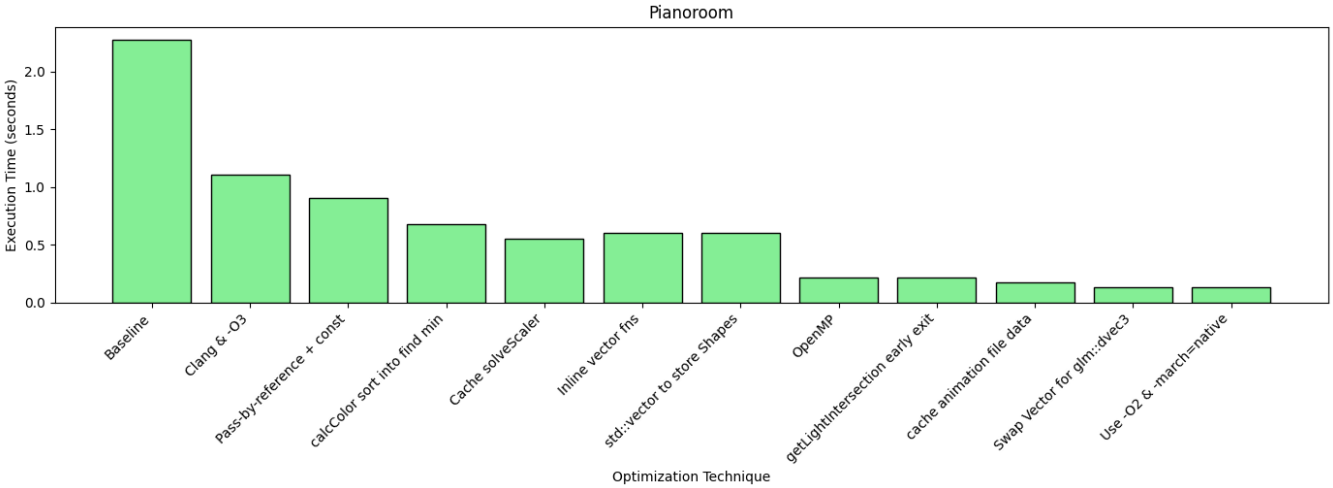

Figure 13. Execution time of Pianoroom rendering with optimizations

Table 1. Globe Optimizations

Optimization	Speed (s)	Rel Speedup	Hash
Baseline	149.401	–	d649112
Clang & -O3	87.353	1.710	2e154bc
Pass-by-ref + const	79.510	1.098	26828ab
calcColor find min	62.792	1.266	54b0047
Cache solveScaler	61.583	1.018	d30fa51
Inline vector fns	59.333	1.038	a497a1f
std::vector for Shapes	59.586	0.996	adcef25
OpenMP	20.617	2.895	12eb706
cache animation file	18.044	1.143	ddf5d3b
Vector -> glm::dvec3	15.979	1.129	890fac4
-O2 & -march=native	15.991	1.000	2d1de14

Table 2. Pianoroom Optimizations

Optimization	Speed (s)	Rel Speedup	Hash
Baseline	2.273	–	d649112
Clang & -O3	1.104	2.061	2e154bc
Pass-by-ref + const	0.908	1.214	26828ab
calcColor find min	0.678	1.341	54b0047
Cache solveScaler	0.555	1.220	d30fa51
Inline vector fns	0.603	0.920	a497a1f
std::vector for Shapes	0.600	1.005	adcef25
OpenMP	0.218	2.756	12eb706
getLightIntersection early exit	0.211	1.032	0c1b1bd
cache animation file	0.172	1.224	ddf5d3b
Vector -> glm::dvec3	0.133	1.298	890fac4
-O2 & -march=native	0.130	1.021	2d1de14