APE Homework 1: Ray-tracer

Adam McNeil
adamwm4@illinois.com
UIUC
Urbana, Illinois, USA

Andy Zhou
andyz3@illinois.com
UIUC
Urbana, Illinois, USA

ACM Reference Format:
Adam McNeil and Andy Zhou. 2025. APE Homework 1: Ray-tracer. In . ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Link to the repo: https://github.com/adamMcneil/598APE-HW1

1 Introduction

We start optimizing the ray tracer by producing a flame graph to see what sections of the program are running the most. Figure 1 is this original flame graph. Figure 1 was created with the following command.

```
./main.exe -i inputs/pianoroom.ray --ppm -o \
    output/pianoroom.ppm -H 500 -W 500
```

This is the command that we used to record the runtime and collect data for all of the plots and flame graphs. We ran the program three times for each collected data point. We report the average and standard deviation of the three trials. Almost all of the time is spent inside the calcColor function in the Shape class. calcColor is a recursive function, and much of the time in calcColor is spent on the resulting calls to the same function. getLight is also a significant amount of work done inside the calcColor function. There are some low-hanging optimization that can be done in the getLight, so we will start there.

Conference'17, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

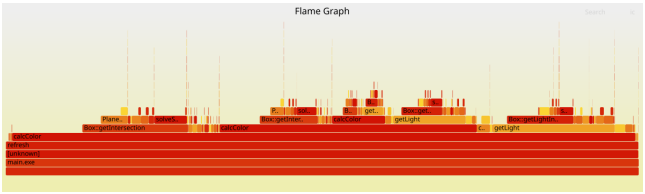

Figure 1. The flame graph of the original project with no optimizations.

2 getLight Optimizations

Inside the getLight function, we are able to do a couple of optimizations. The magnitude of the parameter norm was being recomputed every iteration of the loop. We can store that value in variable and use that instead.

```
double normMag = norm.mag();
```

The function also starts by initializing the first three memory locations of tColor to zero. This combined with the fact that the only other time they are updated, they are all updated with the same value we can get rid of the duplicate computations in getLight and move them to the end of the function. I also tested this assumption by comparing the output with and without the changes. We verified that they were the same. We added the following lines to the end of the function and removed the duplicate computations in the function body.

```
tColor[1] = tColor[0];
tColor[2] = tColor[0];
```

We also do not need to reallocate the lightColor variable each time through the loop. We can just move the definition out of the while loop.

```
double lightColor[3];
```

We did not see very big speed ups for these optimizations. Only about a 0.15 sec decrease in runtime which most likely came from the not recomputing the magnitude of the norm vector. It however made your code more readable, so it was worth it.

3 calcColor Optimizations

In the first part of the calcColor we create a pointer of TimeAndShape objects. In that section we call malloc and free in an inefficient way. We can introduce some extra logic to increase the length of array by more than 1 each time we exceed the length of the array. We determined that 25 was a good number to increase the capacity at each time as needed.

This is mainly because there are about 25 objects in the test scene we are using and each object needs to be tested for collision mean the size is never greater then 25. This is good because we only need to allocate the memory on the heap once. We used the following code to achieve this.

```
TimeAndShape *times = (TimeAndShape *)malloc(0);
size_t cap = 0;
int cap_increase = 25;
size_t seen = 0;
while (t != NULL) {
  double time = t->data->getIntersection(ray);
  if (seen == cap) {
    cap += cap_increase;
    TimeAndShape *times2 =
     (TimeAndShape *)malloc(sizeof(TimeAndShape)
          * (cap));
    for (int i = 0; i < seen; i++) {
      times2[i] = times[i];
    }
   times2[seen] = (TimeAndShape){time, t->data};
    free(times);
    times = times2;
   } else {
   times[seen] = (TimeAndShape) {time, t->data};
   }
   seen++;
   t = t->next;
 }
```

We then sort this array. We experimented with using other algorithms to sort the array with the below code. This however made execution time longer. We believe this is because the array was usually only about 20 objects long. Using the standard sort just added extra overhead. However we may have seen a speed up if the array was longer.

```
#include <algorithm>
void sortArray(TimeAndShape *arr, int n) {
 std::sort(arr, arr + n, [](const TimeAndShape &a,
                        const TimeAndShape &b) {
       return a.time < b.time;
 });
}
```

Overall, we reduced the execution time of the program by about 0.8 seconds by reducing the number of times we called malloc and free.

4 Box::getIntersection Optimizations

In the getIntersection we can move the call to solveScalers after the if statement to reduce the amount of work we have to do. if we see that the time is inf we will return and not have to compute solveScalers. This reduces the total computation that we have to do. This gave use about 0.5 sec of speed up.

```
double Box::getIntersection(Ray ray) {
  double time = Plane::getIntersection(ray);
  if (time == inf) {
    return time;
    }
  Vector dist = solveScalers(right, up, vect,
  ray.point + ray.vector * time - center);
 return (((dist.x >= 0) ? dist.x : -dist.x) > textureX / 2 ||
      ((dist.y >= 0) ? dist.y : -dist.y) > textureY / 2)
            ? inf : time;
}
```

5 solveScalers Optimizations

We added the following functions to reduce the amount of computation that needed to be done. Since x, y, and z can all be computed independently, we could compute x and possibly break from the loop. z was not being used when solveScalers was called. These functions break down the computation of solveScalers and just return the x, y, and z portions of the vector. It did not offer a very large speed up because it only reduced the number of additions and multiplications, which are relatively fast operations.

```
double computeDenom(Vector, Vector, Vector);
double computeX(Vector, Vector, Vector, double);
double computeY(Vector, Vector, Vector, double);
double computeZ(Vector, Vector, Vector, double);
```

6 calcColor Optimizations Part 2

The first part of calceColor is responsible for getting the first shape that the ray hits. The first object it hits will have smallest time value. This means that it only needs to store one object not the entire list. This also means that we do not need to sort a list. We can use the following code to find the first intersection.

```
  double curTime = inf;
  Shape *curShape = nullptr;
  while (t != NULL) {
    double time = t->data->getIntersection(ray);
    if (time < curTime) {
      curTime = time;
      curShape = t->data;
    }
    t = t->next;
  }
```

This gave us a speed up of about 0.25 sec because we did not need to sort the array after finding all of the collisions which reduced the amount of work we needed to do.

7 Macros

We can make macros for the compute functions that we made in the solveScalers optimization. This gives as a speed up because at compile time the compiler in-lines the macro, so we do not have to call a function. We see a speed up of about 0.2 sec by using the macros instead of the functions.

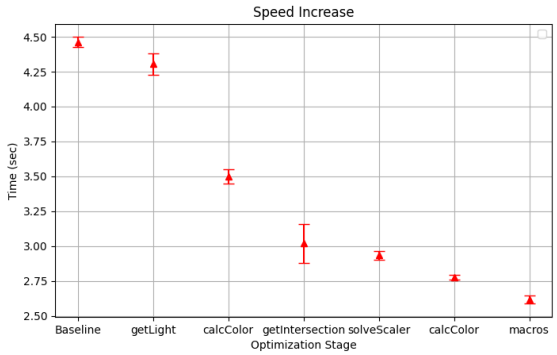

Figure 2. A graph of the average execution time of the program after specific optimization.

```
#define COMPUTE_DENOM(v1, v2, v3)
#define COMPUTE_X(v2, v3, C, denom)
#define COMPUTE_Y(v1, v3, C, denom)
#define COMPUTE_Z(v1, v2, C, denom)
```

8 Compiler Optimizations

We were also able to get a speed up by just passing the -O3 flag to the compiler. We made this change by modifying the Makefile in the src folder to use the -O3 flag instead of -O0.

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

We see the best speed up when we used the third level of optimization.

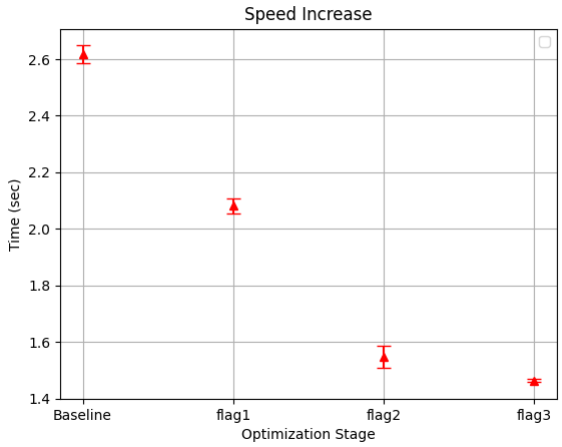

Figure 3. The speed increase from using the different optimization compiler flags.

9 Bounding-Sphere Rejection

In our raytracer's current design, each ray checks for intersections with *every* shape. We introduce a lightweight "bounding-sphere" test to skip shapes whose bounding spheres cannot possibly intersect a given ray, saving expensive intersection checks. If the ray is too far from a shape's bounding sphere (i.e. the quadratic discriminant is negative), we immediately skip the shape entirely.

We added a `boundingRadius` field in `Shape`, initialized in each shape's constructor. For example, `Sphere::Sphere(...)` sets `boundingRadius = rad`, `Disk` uses `max(textureX, textureY)/2`, etc. We then exposed a new virtual method:

```
virtual bool canSkipByBoundingSphere(const Ray &ray);
```

Each shape implements a quick discriminant check on `(ray.point - center)` to see if the bounding-sphere is missed entirely.

We apply this check for Sphere, Triangle, Box, and Disk. For instance, in `Sphere`:

```
bool Sphere::canSkipByBoundingSphere(const Ray &ray) {
    Vector dp = ray.point - center;
    double A = ray.vector.mag2();
    double B = 2.0 * dp.dot(ray.vector);
   double C = dp.mag2() - boundingRadius * boundingRadius;
    double disc = B*B - 4.0 * A * C;
    return (disc < 0.0);
}
```

Finally, in `calcColor`, we check `canSkipByBoundingSphere` *before* calling the more expensive `getIntersection`. If the result is `true`, we skip that shape immediately.

We found that this bounding-sphere skip reduced runtime by about 0.1-0.2 secs. The benefit was smaller than in our mesh-heavy scenes (e.g., the Elephant mesh had a higher speedup). We believe this optimization will pay off most in scenes with large numbers of shapes or meshes, where quickly pruning out-of-range objects prevents many unneeded intersection calculations.

10 Conclusion

We introduced a range of optimizations, from small-scale tweaks (e.g., reusing `norm.mag()` in `getLight`, chunk-based `mallocs` in `calcColor`, and macros for repeated math) to more structural strategies like bounding-sphere quick rejection. The largest overall gain is from combining the various optimizations we tested. Future improvements could include multi-threading, GPU acceleration, or hierarchical bounding volumes (BVH/KD-trees) for more efficient pruning.

These efforts highlight a recurring lesson: simple algorithmic changes (e.g., skipping unnecessary intersections or allocations) often exceed pure micro-optimizations. In the future, we could extend this to a BVH or KD-tree for even more robust pruning, or explore multi-threading to better utilize modern CPUs. Overall, these refinements illustrate how iteratively profiling, refining core data-flow, and eliminating wasted work can add up to significant performance gains without sacrificing correctness.