

APE Homework 1: Ray-tracer

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

ACM Reference Format:

Adam McNeil. 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.

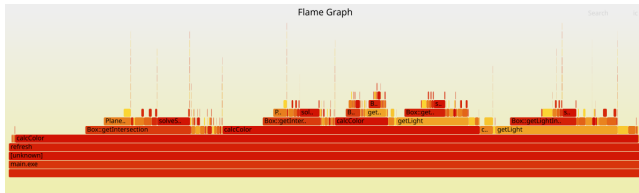


Figure 1. The flame graph of the original project with no optimizations.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org. Conference'17, Washington, DC, USA

© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.

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

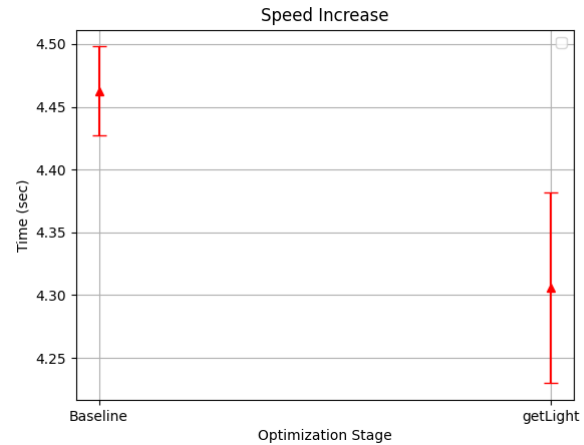


Figure 2. Speed increase after the getLight optimizations

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.

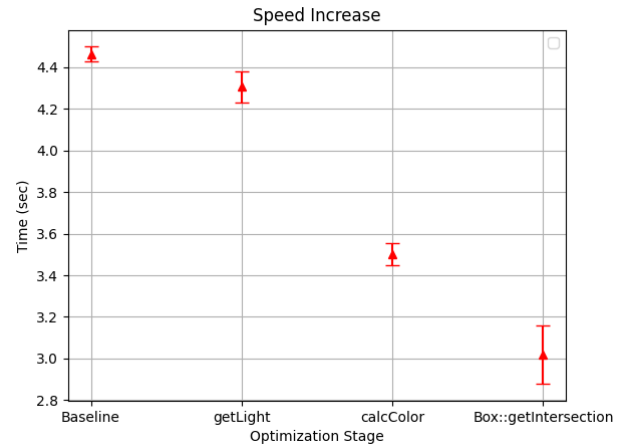


Figure 3. Enter Caption

4 Box::getIntersection Optimizations

In the getIntersection we can move the call to solveScalars 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 solveScalars. 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 = solveScalars(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 solveScalars 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 solveScalars was called. These functions break down the computation of solveScalars 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 calcColor 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 following functions so we do not need to call a new function when we call them. We see a speed up of about 0.2 sec by using the macros instead of the functions.

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

#define COMPUTE_DENOM(v1, v2, v3) \
((v1).z * (v2).y * (v3).x - (v1).y * (v2).z * (v3).x - \
(v1).z * (v2).x * (v3).y + (v1).x * (v2).z * (v3).y + \
(v1).y * (v2).x * (v3).z - (v1).x * (v2).y * (v3).z)
#define COMPUTE_X(v2, v3, C, denom) \
(((C).z * (v2).y * (v3).x - (C).y * (v2).z * (v3).x - \
(C).z * (v2).x * (v3).y + (C).x * (v2).z * (v3).y + \
(C).y * (v2).x * (v3).z - (C).x * (v2).y * (v3).z) / \
(denom))
#define COMPUTE_Y(v1, v3, C, denom) \
(((C).z * (v1).y * (v3).x + (C).y * (v1).z * (v3).x - \
(C).z * (v1).x * (v3).y - (C).x * (v1).z * (v3).y - \
(C).y * (v1).x * (v3).z + (C).x * (v1).y * (v3).z) / \
(denom))
#define COMPUTE_Z(v1, v2, C, denom) \
(((C).z * (v1).y * (v2).x - (C).y * (v1).z * (v2).x - \
(C).z * (v1).x * (v2).y + (C).x * (v1).z * (v2).y + \
(C).y * (v1).x * (v2).z - (C).x * (v1).y * (v2).z) / \
(denom))
```

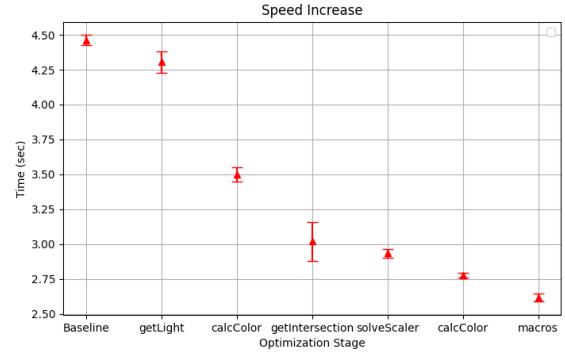


Figure 4. A graph of the average execution time of the program after specific optimization.

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.

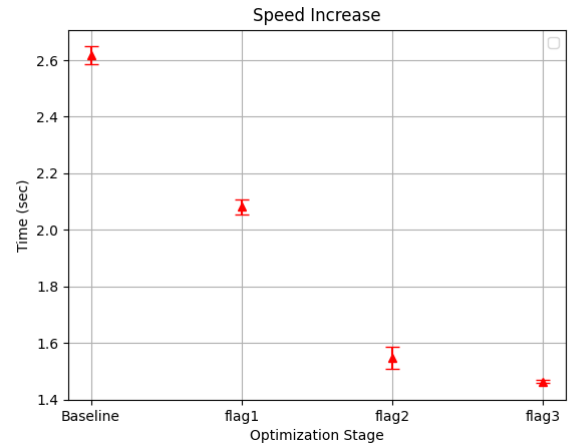


Figure 5. The speed increase from using the different optimization compiler flags.