DE HEER Karel S N

Low-Level Programming  Staffordshire University

Ray Tracer Optimization Documentation

Contents

[**Introduction** 2](#_Toc534843704)

[**Optimizations** 2](#_Toc534843705)

[**Use of C++ Standard Library – Chrono** 2](#_Toc534843706)

[**Memory Management** 2](#_Toc534843707)

[**Heap Manipulation** 2](#_Toc534843708)

[**Framework** 2](#_Toc534843709)

[**Ray Tracer Class** 2](#_Toc534843710)

[**Renderer Class** 2](#_Toc534843711)

[**Animation Classes** 2](#_Toc534843712)

[**Threading** 2](#_Toc534843713)

[**File Input & Output** 2](#_Toc534843714)

[**Data Structures** 3](#_Toc534843715)

[**Conclusion** 3](#_Toc534843716)

[**References** 3](#_Toc534843717)

**Ray Tracer Optimization**

# **Introduction**

For the assignment, the task was to optimize an inefficient ray tracer framework for not only faster execution speed. In addition to this, the structure of the framework has been changed to increase ease of use and readability for users. This report illustrates the alterations to the code made to reach these objectives and displays tabulated data comparing the speeds of the original and improved ray tracer.

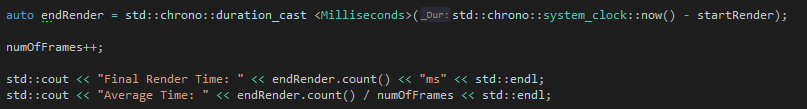
# **Optimizations**

All tables, statistics and

## **Use of C++ Standard Library – Chrono**

To record the speeds between the original ray tracer framework, the optimized and threaded versions, the standard C++ library feature, chrono, has been utilized. This gets the current the system clock at the has been used to record the start and end time of the framework during certain sections of processing. Namely the “SmoothScaling” function in which the program outputs to console the time taken each sphere file to be created/updated and the total time for the whole process with an average time taken as well. The program also outputs to file the total time and average time in milliseconds and seconds.





## **Memory Management**

### **Heap Manipulation**

To better monitor the use of memory within the application, I have created my own memory manager which uses its own new and delete functions which malloc and free memory stored data into random access memory (RAM). A heap class which stores class names and size, and a heap factory which contains all the heaps (default and class specific) created.

## **Framework**

To help ease of uses, the original code given in the framework has been refactored and split into the relevant classes. This allows for easier optimization of code.

### **Ray Tracer Class**

For the ray tracer, I have moved any code that is hardcoded inside of the “trace” function and moved them from the function scope to become class members so that they are only needed to be initialized once every time the instance of the class is called.

This class has been designed as a singleton class since only a single instance of the class is ever required to control throughout the execution.

I tested the original ray tracer with my optimized version in resolution dimensions of 1920x1080. Fig. 1 is the results found below. This tables compares the two version’s time taken to perform the trace method during the rendering the first 10 frames, recorded in micro seconds. Due to this method being called so many time within a short period of time, I have decided a sample size of 10 is fair enough time for both versions to illustrate their speeds.

|  |  |  |  |
| --- | --- | --- | --- |
| Render No. | Original (micro seconds) | | Optimized (micro seconds) |
| 1st | | 4.59135 | 0.0145862 |
| 2nd | | 4.60091 | 0.00309317 |
| 3rd | | 5.17816 | 0.00349055 |
| 4th | | 5.09855 | 0.00362703 |
| 5th | | 4.64418 | 0.00268277 |
| 6th | | 4.69802 | 0.0022526 |
| 7th | | 4.42941 | 0.00378424 |
| 8th | | 5.02573 | 0.00426939 |
| 9th | | 5.26033 | 0.00619647 |
| 10th | | 5.1504 | 0.00295235 |
| average | | 4.867704 | 0.004693477 |

Fig. 1: Comparing original with optimized trace method speed.

As Fig. 1 show, the average time of the original trace implementation is significantly longer than the optimized version’s time; In fact, the optimized version takes less than 10% of the time the original version does. This displays that the optimized version is much faster at processing this method than the original one did thus it can be claimed that the optimized version is superior in speed.

### **Renderer Class**

Inside this class, the “render”, “SmoothScaling”, “SimpleShrinking” and “BasicRender” methods have been moved to reside inside here.

Same with the previous class, all hardcoded values within the functions have been moved to class members to save time used instantiating variables.

An example of an optimization can be found in fig. 2. The table shows that the average time taken to run each “render” method is less for the optimized version than the original, showing that again the optimized version is more efficient at rendering and saving to file each scene.

If we calculate efficiency, the optimized version only takes 7 percent of the total time the original program takes, which is significantly more efficient. In fact, from the data in fig. 2 it can be assumed the optimized version renders and outputs an average of 14 frames for every 1 frame the original version renders and outputs.

This evidence

|  |  |
| --- | --- |
| Original (average time in milliseconds) | Optimized (average time in milliseconds) |
| 16055.2 | 1146.38 |

Fig. 2: Average times between original’s and optimized’s render method.

### **Animation Classes**

When setting up the sphere’s properties have been altered as well. Classes have been created to read a json file (exported from the editor, Keditor, created along side this project) with the properties of each sphere and store it into a “Frame” object that stores all the data. These are used to create the animations each scene. Fig. 3 presents the speed difference between using hardcoded values and values read in from file.

## **Threading**

## **File Input & Output**

The transform and colour properties of the sphere are no longer hardcoded values, but instead are variables within a “FrameCollection” that stores the values read in a json file.

The frames output the file are all .ppm files. To streamline the file writing, the information of all pixel on the image of the scene is only written to the file altogether instead of being written each file per pixel.

## **Data Structures**

# **Conclusion**

# **References**

Json for Modern C++

Nlohmann.github.io. (2019). *JSON for Modern C++: JSON for Modern C++*. [online] Available at: https://nlohmann.github.io/json/ [Accessed 6 Jan. 2019].