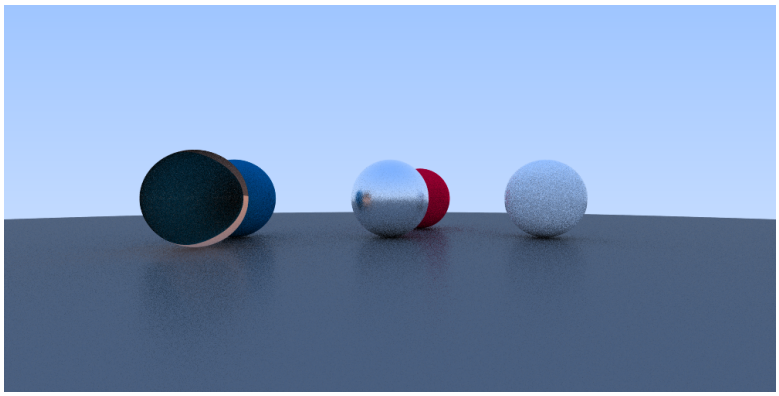
Optimizing a ray-tracing algorithm in C++



Author
Pontus Andersson
ponane-6@student.ltu.se

Lecturer
Fredrik Lindhal

Department of Computer Science, Electrical and Space Engineering

LULEÅ TEKNISKA UNIVERSITET

December 18, 2020

Contents

1 Inledning

The assignment presented in this paper consisted of running various analysis tools e.g. static code analysis and profiling on a ray-tracing program given by the lecturer. Any errors, warnings and memory leaks were to be fixed and optimization in areas such memory management and mathematics were to be carried out. Further, multi-threading and a data-oriented approach was to the implemented in appropriate areas.

2 Method

Various approaches were used in the process of optimizing the raytracer program. Depending on the situation and the area of appliance, some were used more frequent than others. For example, static code analysis was performed after a certain implementation criteria had been met, whilst profiling could be used after each code change. After each implementation the running time of the raytracing algorithm as well as the number of rays per second were recorded in a log file.

2.1 Static code analysis

For the process of static code analysis, PVS-Studio was used with an academic licence. With this tool a trace could be generated from the program code, which could then be analyzed and displayed with the help of a visualizer tool. From this information code bugs and potential memory issues could be identified and resolved.

2.2 Profiling

For profiling, Valgrind's tool Callgrind was used alongside KCachegrind for analysing and visualization. Callgrind logged various run-time information about the program, e.g. the number of method calls and their accumulated processing time. From this information it became possible to pin-point areas of the code which were heavy on performance.

2.3 Manual code analysis

One of the more frequently used methods were to manually analyze the code. This proved useful as an initial method to quickly identify errors and potential improvements throughout the code. One example when this was used were in the identification of appropriate function calls to use constant reference (pass-by-reference) instead of pass-by-value.

2.4 Iterative approach

Another method was to simply try different solutions to a problem and measure their respective performance. When a solution was found without any distinct possibility for further improvements, the code was analyzed to identify which areas improved on the previous implementation and why.

2.5 Key implementations

2.5.1 Memory optimization

One of the key implementations throughout the program were to decrease the number of constructed variables and methods calls. The reasoning behind this were to reduce the overhead needed to allocate memory if it could be avoided. This were done in one of three way; restructure the code to invoke less constructor calls, redesign the code to re-use already utilized memory, or remove redundant code completely.

One great example of the first one were in a function which performed multiplication between a 3 item vector and a 4-by-4 matrices. Initially the function created 10 vector variables (not counting ones constructed in internal method calls), and afterwards only 1.

Of the second one the best example is the one already mentioned, change the passing of variables by reference instead of value where possible. Pretty much the every function in the program had one or more parameter which could be altered in this sense.

An example of the last one was the cutting of two complete functions in the vector class where the end result never was utilized, thus rendering them completely unnecessarily and safe to remove.

2.5.2 Multi-threading

Another key implementation was the utilization of multiple threads. To make this work, the function which handled the ray-tracing had to be re-designed, including the removal of its recursive calls. The new functionality included splitting the work of rendering a scene into jobs, where each pixel of the scene resolution represented a job for a thread to work on. The total amount of jobs (sum of pixels) was divided between the total amount of threads the program were prompted to use. The last thread were tasked with any trailing jobs should the distribution be uneven.

2.5.3 Memory buffers

The implementation of buffers which stored object data were yet another implementation. The buffers stored the material, transform and id of objects in the scene and could provide direct pointer access to the data. The primary use of this were in specific ray-trace functions where pointers to the different data attributes could referenced, used, and incremented.

Splitting the objects and storing them in buffers also had the benefit of reducing the redundancy of loading entire objects, when really only one of the attributes was needed.

3 Results

The results is displayed in the same order as various code improvements were made. Each section represent a code commit with its associated changes and the performance of the program after the commit was made. Each section is also labeled with a number used as an index in this summarized table at the end of the chapter. The tests where run with the pre-defined settings of a 1000 by 500 resolution, 20 rays per pixel, and a maximum of 5 bounces.

(0). The performance of the base application without any changes to the original code.

$Time(t)$	$Megarays/sec(Mr/s)$
4.134768	3.866590

(1). Commit include the following improvements: Removing redundant method functions which performs heavy computations without ever being utilized; Changing the precision throughout the project from double to float; Turned all possible function parameters into constant references throughout the project.

$Time(t)$	$Megarays/sec(Mr/s)$
1.301902	12.263152

(2). Commit include: The class used as material now utilized an Enum to distinguish from different material types instead of a String.

$Time(t)$	$Megarays/sec(Mr/s)$
1.242929	12.837567

(3). Commit include: Corrected some method parameters to use constant references which were missed from previous commits.

$Time(t)$	$Megarays/sec(Mr/s)$
1.140755	13.951686

(4). Commit include: Reduced the number of vectors and rays created in the main loop which handles ray-tracing; Improved the intersection test function for the sphere objects to use more constant references.

$Time(t)$	$Megarays/sec(Mr/s)$
1.025232	15.547829

(5). Commit include: Implemented a data-oriented approach of storing geometry, materials and objects. This was done by creating buffers and providing pointers for said data.

$Time(t)$	$Megarays/sec(Mr/s)$
1.012415	15.793352

(6). Commit include: Reduced the number of randomly generated floats and flow-control branches inside a material function.

$Time(t)$	$Megarays/sec(Mr/s)$
0.996511	16.045407

(7). Commit include: Removed the recursive loop used for tracing a ray in the scene; Utilized the data oriented storage inside the main ray-trace loop.

$Time(t)$	$Megarays/sec(Mr/s)$
0.896141	17.796066

(8). Commit include: The ray-trace loop now re-uses rays instead of creating new ones each time it collides or a new trace is to be made.

$Time(t)$	$Megarays/sec(Mr/s)$
0.860484	18.533519

(9). Commit include: Implemented multi-threading with 8 threads after some testing with how many thread yielded the best result; The value in the framebuffer is modified directly instead of first using a separate color buffer for calculation and then copying over the data.

$Time(t)$	$Megarays/sec(Mr/s)$
0.425345	34.026432

Table 1: Summary of the computation time t and the number of megarays Mr/s corresponding to each code commit c and the difference d in megarays from its parent.

$Commit$ (c)	$Time$ (t)	$Megarays/sec$ (Mr/s)	$Diff$ (d)
0	4.134768	3.866590	0
1	1.301902	12.263152	+8,3966
2	1.242929	12.837567	+0,574415
3	1.140755	13.951686	+1,114119
4	1.025232	15.547829	+1,596143
5	1.012415	15.793352	+0,245523
6	0.996511	16.045407	+0,252055
7	0.896141	17.796066	+1,750659
8	0.860484	18.533519	+0,737453
9	0.425345	34.026432	+15,492913

4 Diskussion och slutsatser

From the results one can clearly see an improvement throughout the optimization process. Looking at the data displayed in table 1, the two commits which contributed the most is the 1st and 9th commit. The first commit contains the cutting of redundant functions, changing the precision, as well as turning most function parameters into constant references. One mistake is that these three wasn't tested individually, rather than merged together as they are now, since it would have been interesting to know which contributed the most. While the second change surely made an impact, it is possible safe to assume that the other two out-weights it. One way of getting an idea is to look at commit number 3, when a few function parameters has been missed and those alone increased the number of megarays per second by 1. Taking this into perspective, if the whole ray-tracer was overhauled with these changes, one can believe that a majority of the improvement from the first commit were from this alone. This said, it is clear how effective the constant reference qualifiers can be as a code optimization if one has the opportunity to use it.

The other big performance boost were from the 9th commit which consisted of the multi-threading implementation. Not surprisingly, this had a huge impact in the number of rays calculated per second as it roughly doubled the performance of the ray-tracer program with 8 cores running in parallel. One could argue however, that one could expect that number to be higher since the jump from 1 to 8 cores should result in an 8-times increase, right? While in a perfect environment and better optimization the result would come much closer to that, in reality the number of threads doesn't necessarily equivalent a performance multiplier. One possible improvement to the current implementation could be to change the way the threads write to the framebuffer. Currently they all share a common pointer to the start of the same framebuffer, and the idea is that this might cause a race condition of which thread will have the access of writing its data first. The solution to this might then be to create a separate buffer for each thread to write to during computation, and ultimately link this together as the final framebuffer.

Another implementation that improved the running time less than one might anticipate is the data-oriented approach contained in commit number 5 and 7. First it is worth mentioning that in the latter commit there is also the change to the recursive ray-trace loop. This did however, drawing from memory, not affect the running time in any noteworthy way, which means that the performance gain comes solely from the utilization of the data-oriented approach. Now one can also see comparing these two commits that the implementation of the data-oriented storage did almost nothing on its own performance-wise, as the key was how to actually utilize it. Granted, the 7th commit ranked in the top three of of commits which did the most in increasing performance, it's the notion that these results could have been better given more time and thought about the data-oriented implementation for the current ray-tracing scenario. This of course is also true for the other implementations, and much could have been gained if more research had been done on existing solutions.