Ray Blazer - Optimization of an intentionally poorly written ray tracer

Gandalf Sellén Lindkvist

April 19 2023

Contents

Abstract

This report covers the steps that were taken to profile and optimize a ray tracer in C++ that was poorly written on purpose. By using Visual Studio's built-in profiler, along with the Concurency Visualizer plugin and uProf, the code was analyzed and then improved by increasing cache coherency, adding multi-threading, and removing unnecessary code structures, incrementally until the program reached 125 MRays/s on a 32 core system.

1 Introduction

Analyzing a program can involve any number of activities, but this report focuses on three main areas – creating and using a benchmark program, running static code analysis, and running different types of profilers to detect cache misses and hot functions. The process of optimizing also involves some programming intuition, which comes from experience. This report will therefore try to delineate the thought process behind each change and describe which tools were used to find the potential improvements to provide a more complete picture of what the process was actually like. Most of the time, it was not the literal values of the data from the analysis that was important for picking the next area of improvement, but rather just the hierarchy of the bottlenecks. Hence, large tables of data are avoided in this report and instead the method of discovery of the relevant improvement points are described.

The program in question is a poorly written C++ ray tracing (or path tracing) routine that runs on the CPU. The programmers of the original program have written intentionally bad code with unnecessary execution paths, memory leaks, incoherent cache and without multi-threading. In the following chapters, the evaluation, reparation and improvement of this code is presented.

2 Benchmark program

To evaluate the code of interest, a separate program was created where the ray tracing routine was isolated inside a loop of fixed iterations. The execution time could then be measured for this loop, whereupon the average time for all iterations could be calculated. The number of rays spawned could also be measured by incrementing a static (and atomic) integer inside the constructor of the Ray-class, which enabled the calculation of the average number of rays per frame and also the average number of rays per second (MRays/s). The number of iterations of the loop were increased throughout the optimization process as the program became faster. This was to ensure that the profiler tools got enough samples for their analysis. At first, 300 iterations was sufficient, but later 1200 iterations became more reasonable. Additionally, the resolution was set to 200x100 pixels, the max ray bounce count was set to 5, the number of rays per pixel was set to 1 and the number of spheres was set to 11.

An initial run of the benchmark program yielded the statistics that the next optimizations would try and improve upon. The information is presented below in Table 1.

Table 1: Initial benchmark results

ms/frame	rays/frame	MRays/s
96,67	31400	0,3248

Additionally, the benchmark program was analyzed with Visual Studio's

built-in profiler, which reported the top functions (functions that consume the most time). The top functions are presented in Table 2 below.

Table 2: Initial top functions

Function name	CPU%
Raytracer::Raycast	29
vec3::UpdateIsNormalizedVariable	14
vec3::UpdateIsZeroVariable	7,0

Using uProf (a profiling tool for AMD processors), one could also detect functions with a lot of cache misses, which gave (and would most of the time hence forth give) similar results as the top functions-analysis. The results can be found in Table 3, where IBS-DC-MISS-LAT is the total latency of all data cache misses expressed in clock cycles. One thing to note about the results is that vec3::UpdateIsZeroVariable did not show up amongst the top functions for cache misses. That is probably because the vec3::len-function (which computes the length of the vector) is present in both vec3::UpdateIsNormalizedVariable and in vec3::UpdateIsZeroVariable, and since these two functions were always called back to back after each other in that order, the vec3::len-function would produce a cache miss in the first and still be in cache for the latter.

Table 3: Initial top functions for cache misses

Function name	IBS-DC-MISS-LAT
Raytracer::Raycast	416
vec3::Raytrace	106
vec3::UpdateIsNormalizedVariable	56

Lastly, a static code analysis was performed with the built-in tool in Visual Studio. The results presented a wall of warnings, which will not be included in this report for readability reasons but can be found in a text-file in the testing-folder in the project directory. Moving on with optimizations, the output from the benchmark program, together with the information from the Visual Studio profiler and uProf, was used when relevant to find the next bottleneck and to confirm that the changes produced better results.

3 Improvements on vec3

Since the vec3-class was responsible for two of the top contenders in the previous analysis, it was the top priority. One could immediately see that the two functions that checked the length of the vector upon creation were totally unnecessary and they could easily be removed. The data type of the elements of the vector were also changed from double to float since double precision is rarely

needed in graphics programming and offers faster arithmetics. The len-function and the normalize-function could also be simplified, since there were a lot of unnecessary variables created. The initializer-list constructor could also be removed since objects can be initialized with curly braces by default in modern C++. Lastly, some operators could me marked as const as they did not alter the internal state of the class. All together this resulted in the benchmark program producing the results found in Table 4.

Table 4: Benchmark results after vec3 improvements

ms/frame	rays/frame	MRays/s
65,45	31400	0,4797

Checking the top functions again showed that the vec3-class was no longer causing trouble and that a new set of functions had risen to the top (Table 5).

Table 5: Top functions after vec3 improvements

Function name	CPU%
Raytracer::Raycast	34
std::string::$_C onstruct$	7,6

When checking the cache misses in uProf, the Raytracer::Raytrace-function stood out particullarly, and since the fix was so simple, that is what happened next.

4 Improvements on Raytrace

Initially, when looping through each pixel to perform the ray tracing, the outer loop iterated over the x-coordinate while the inner loop iterated over the y-coordinate. Then when adding the resulting color to the framebuffer (an std::vector of colors), the index of the color was calculated as the y-coordinate times the width plus the x-coordinate, and it was at that line that the cache misses occurred. The reason for this is the ordering of the nested loops – when the inner loop iterates, the y-coordinate is increased by one, which will produce a huge jump in the framebuffer-index. That in turn will cause incoherent memory access, which will cause cache misses. To fix this, the outer and inner loops were swapped and the cache misses decreased dramatically. Some cache misses still occurred, and that is probably because the entire framebuffer cannot fit into L1 cache and different chunks of the vector must be fetched throughout the loops. Checking the benchmark program again showed a slight improvement, but an improvement nonetheless. The data is found in Table 6 below.

Table 6: Benchmark results after Raytracer::Raytrace improvements

ms/frame	rays/frame	MRays/s
63,38	31401	0,4954

5 Improvements on Raycast and TracePath

From observing the top functions again, one could see that the Raytracer::Raycast-function still ruled supreme. When digging deeper into what was actually causing the workload of the function it was discovered that the Raytracer::TracePath-function was a huge part of the problem. By changing the Raytracer::TracePath-function from a recursive function into an iterative function and removing unnecessary sorting and adding to an std::vector inside the Raytracer::TracePath-function, huge improvements were seen. Additionally, a bunch of function arguments in the Raytracer-class were changed from pass-by-value to pass-by-reference to decrease the copying required. There were also some instances where objects were created on the heap where there was no need to, so changing those objects to stack objects also improved performance by lessening the number of allocations. Lastly, some unused variables inside the Object-class (which the Sphere-class inherited from) were removed. All in all, this resulted in the biggest improvement yet, visible in Table 7.

Table 7: Benchmark results after Raytracer::Raycast and Raytracer::TracePath improvements

ms/frame	rays/frame	MRays/s
4,417	31402	7,109

The improvements also resulted in a new set of top functions (Table 8), which were starting to look more reasonable from a ray tracing algorithm perspective.

Table 8: Top functions after vec3 improvements

Function name	CPU%
Sphere::Intersect	46
Raytracer::Raycast	7,5
Raytracer::Raytrace	6,6
Raytracer::TracePath	4.5
(vec3) normalize	4,5

The cache misses were also dramatically decreased when checking uProf again, although in retrospect, they seem to be alarmingly few, considering the cache miss discoveries that are yet to be covered.

6 Improvements on Material and Pbr

Since the top functions were looking quite in order at this point, the programmer intuition came in to play. When looking through the source code files, the Material-class and the Pbr-functions stood out particularly. The Material-class had a variable that described its type with an std::string and in the Material::BSDF-function that string was compared to check which BSDF-procedure to use. Replacing this type variable with an enumerator and putting the differend BSDF-procedures into separate functions allowed for a cleaner and more efficient Material::BSDF-function, where the enumerator type was checked in a switch-statement whereupon the corresponding BSDF-function was called. Additionally, some function arguments were changed from pass-by-value to pass-by-reference. These changes resulted in a slight improvement presented in Table 9.

Table 9: Benchmark results after Material improvements

ms/frame	rays/frame	MRays/s
4,201	31402	7,473

7 Implementing a memory pool

At this point in the optimization process, it was getting more difficult to find plain errors in the code, and the need for new ideas and innovation was increased. One idea was the MemoryPool-class, which put objects next to each other in memory instead of storing pointers to heap allocated objects inside an array. This would theoretically increase cache coherency, since there would be no need to look up fragmented objects in the heap. The memory pool was used to store spheres, so that the Raytracer::RayCast-function would access them more quickly when testing intersections. Since storing the spheres as pointers to heap allocated objects was no longer an options, there was no point in having a base class (the Object-class), so it was scrapped completely and the Sphere-class was made into a stand-alone class.

The MemoryPool-class is a very minimalistic memory manager. It is a template class that allocates a fixed-size array of the given type upon creation. It then keeps track of how many of the objects are instantiated, and when the user wants a new object, it simply returns a pointer to the next available object in the array, essentially mimicking a stack. Since objects did not have to be removed after creation, no such function was implemented, and it would have needed to shuffle around a bunch of memory if the objects were not removed in the reverse order that they were allocated, so it was not worth the effort of implementing for this application. The MemoryPool-class also offers an index-operator and a method for getting the number of objects instantiated so that the array can be traversed.

After replacing the std::vector of Object-pointers with a MemoryPool of Sphere objects in the Raytracer-class, the performance was increased again (shown in Table 10). Interestingly enough, the number of rays generated seemed to increase. In retrospect this seems odd, and it might have been caused by some ignorant change to some random seed value that was not documented.

Table 10: Benchmark results after implementation of a memory pool

ms/frame	rays/frame	MRays/s
3,926	31626	8,056

8 Replacing the RandomFloat-function

Continnuing with the programmer intuition, the next piece of code that stood out was the RandomFloat-function. It seemed unnecessarily complex with one function producing a random unsigned integer which another function would then use to create a random float, so it was replaced by a simpler xorshift-function. As it turns out, this was not the last time that the RandomFloat-function would be altered, but its first improvement yielded the results found in Table 11 below.

Table 11: Benchmark results after replacing the RandomFloat-function

ms/frame	rays/frame	MRays/s
3,774	31628	8,381

9 Implementing a thread pool

The largest undertaking of this whole optimization process was the multi-threading. It took a long time to get right, but it allowed the results to sky-rocket. Using the Concurrency Visualizer plugin for Visual Studio, the initial CPU-utilization could be observed. It was (as expected from a single threaded program) very poor, and it can be viewed in Figure 1. One could also observe how the threads (one being the main thread an the other being some background thread) were spread across the 32 cores in Figure 2.

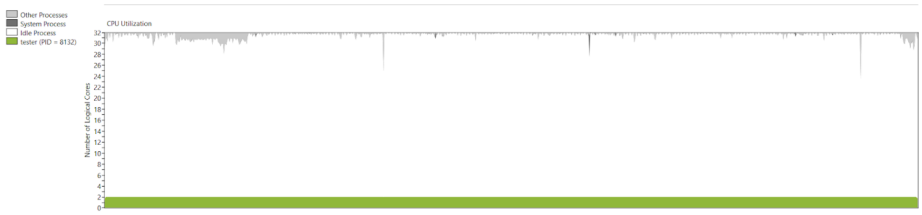

Figure 1: CPU utilization before multi-threading

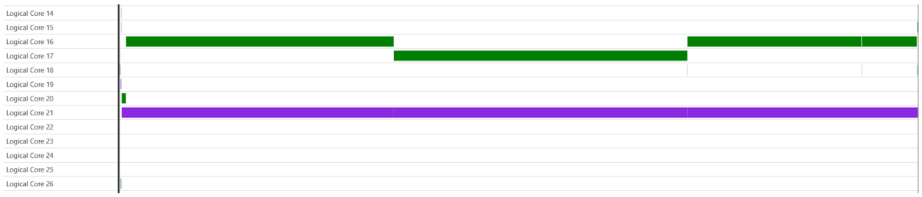

Figure 2: Core utilization before multi-threading

The ThreadPool-class was implemented to store an std::vector of Workers, where each Worker is a struct of an std::thread pointer and a boolean that flags if the thread is done with its work or not. The ThreadPool-class also contains an atomic integer that allows the threads to increment it safely when they are done with their work. To use the class, one first creates an instance of it, where the number of threads needed is submitted. It is then up to the user to initialize each Worker through the ThreadPool::InitThread-function, where the work that the Worker is going to performed is passed as a function pointer along with the arguments to pass into that work-function. A wrapper function is then created that will loop infinitely until the ThreadPool flags that it should shut down.

Inside the loop the Worker's flag is checked to see if it should perform some work or not. If it should perform some work, then the submitted work-function is called with the submitted arguments and the Worker's flag is changed so that it does not repeat the work again. If it should not perform any work, then it yields its time so that the OS can reschedule. This wrapper-function is then submitted to the thread upon creation along with the submitted work-function arguments. The ThreadPool-class then provides an ExecuteAndWait-function that loops through each Worker and sets their flag to false (meaning that they should do some work) and then it waits for the atomic integer to reach the same number as the number of Workers (which means all Workers are done). The counter is then reset to zero and the function is exited. In the destructor of the ThreadPool-class, the flag for the Worker's infinite loop is set to false, whereupon each thread is joined and deleted. This assures that each thread exits gracefully.

When putting the ThreadPool-class to use in the Raytracer, it was initialized to use as many threads as there were core available on the machine. Each Worker was then initialized to perform an altered version of the Raytracer::Raytrace-function that allowed the function to start at a certain pixel coordinate and compute a certain amount of pixels. In the original Raytracer::Raytrace-function everything could then be replaced by a single line of code where the ThreadPool's ExecuteAndWait-function was called.

This change did not bring immediate huge improvements, but further down the light we will discover its true potential. Below are the CPU utilization (Figure 3), the core utilization (Figure 4), and the benchmark results (Table 12) which show great improvements.

Figure 3: CPU utilization after multi-threading

Table 12: Benchmark results after implementing the ThreadPool-class

ms/frame	rays/frame	MRays/s
1,855	31627	17,05

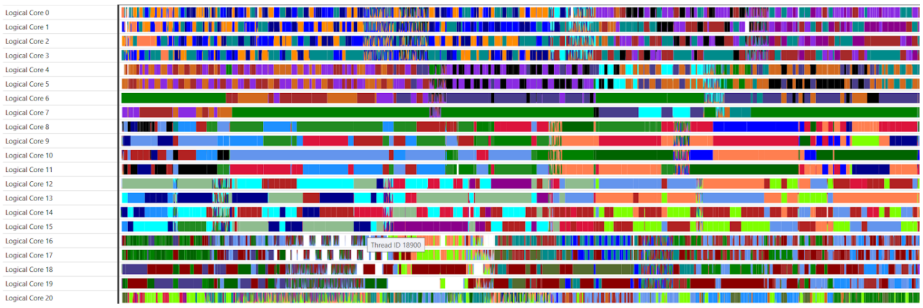

Figure 4: Core utilization after multi-threading

10 Improving RandomFloat further

When analyzing the cache misses after implementing the multi-threading, one could observe that the RandomFloat-function was the largest bottleneck, more specifically its static unsigned integer seed variable that was used to change the output after each call. To solve this, the seed was changed to be sent as a value to the function, so that the caller of the function would be responsible for changing the seed. This removed the cache misses for the RandomFloat-function and hence improved the performance (as seen in Table 13).

Table 13: Benchmark results after removing static variable from the RandomFloat-function

ms/frame	rays/frame	MRays/s
1,533	29768	19,42

11 Moving additional work to the threads

At the end of each frame, the result from the framebuffer must be copied into a result-framebuffer and then averaged over the number of frames that are accumulated. By moving this into the Raytracer::RaytraceGroup (which each thread runs), the performance could be increased even more (Table 14).

Table 14: Benchmark results after parallelizing the framebuffer copying

ms/frame	rays/frame	MRays/s
1,450	29767	20,52

12 The last static variables

After the realization that static variables cause cache misses, the hunt for the last static variables was on. One was found in the Raytracer::RaytraceGroup-function, where it was a seed for the RandomFloat-calls within that function. It could simply be replaced by some xor's and or's based on the current frame index, which produced a good enough randomness. The second one was found inside the Material-class, where it (once again) was a seed variable for the RandomFloat-calls within that function. To solve this, a seed was sent to those functions from the caller, all the way back up the the Raytracer::RaytraceGroup-function, which generates the seed. This simple improvement unleashed the long awaited potential in the multi threading (shown in Table 14), and when the ray-counter (which also was a static variable) was implemented without being static and atomic the results got even better (shown in Table 15).

Table 15: Benchmark results after removing two static variables

ms/frame	rays/frame	MRays/s
0,743	31437	42,32

Table 16: Benchmark results after removing the static ray counter variable

ms/frame	rays/frame	MRays/s
0,25184	31437	124,8

13 Implementing bounding spheres

The final change to the code was implementing bounding spheres. The idea with bounding spheres is that multiple spheres (or potentially other geometry) can be grouped inside a bounding sphere, so that when doing intersection tests, one can first test intersection against the bounding sphere and if there is no intersection, one can ignore all the objects inside it. Trying to make the bounding spheres as light weight as possible, they simply store a position, a radius, a fixed-size array of indices to spheres and an integer for how many indices are used inside the array. Then, before starting the render loop, we iterate through each sphere and put it into a bounding sphere that it will fit into (also considering the max radius of the bounding sphere). If there is a bounding sphere that can include it, it will make sure its radius is updated to surround it completely. If there are no bounding spheres that can include the sphere, a new one is added to the the center of the sphere with its radius plus some small error margin. Finally, when casting rays in Raytracer::Raycast, one iterates through the bounding spheres in an outer loop, testing intersection and checking if inside it, then (if the ray

overlaps the bounding sphere) a second loop iterates over the indices of spheres that the bounding sphere contains and intersection tests are run as usual for them.

When testing the performance, it was obvious that there would need to be more spheres in the scene than 11 to see the full potential of this solution, so the first test was done with 100 spheres, then 500, then 1000 and finally 10000 (results in Table 17). All bounding sphere tests were performed with the bounding spheres' max radius set to 10 and their max capacity set to 20 (a combination of values that seemed to yield good results in a simple series of tests).

Table 17: Benchmark results with and without bounding spheres (100, 500, 1000 and 10000 spheres)

spheres	bounding spheres	ms/frame	rays/frame	MRays/s
100	0	1,659	35340	21,30
100	10	0,8019	35498	44,27
500	0	13,32	59913	4,498
500	30	5,968	60024	10,05
1000	0	34,09	78068	2,290
1000	55	15,00	78068	5,204
10000	0	297,8	93979	0,3156
10000	504	107,5	93980	0,8742

A final test was also performed with 1000x500 resolution, 20 rays per pixel, 10000 spheres (from (-50, 0, -100) to (50, 50, 20)), 5 maximum bounces and with the camera at (0, 10, 0) looking down the negative z-axis. The ranges for where the spheres could spawn meant that most spheres would be located outside the camera's view frustum. From the results in Table 18 one can see that the bounding spheres provide a 10 times performance increase in this specific scene, which is expected.

Table 18: Benchmark results with and without bounding spheres with greater resolution and more rays per pixel

spheres	bounding spheres	ms/frame	rays/frame	MRays/s
10000	0	144100	46720999	0,3242
10000	798	14618	46720999	3,196

14 Cleaning up remaining warnings

As previously mentioned (ch.2), there were initially a lot of warnings coming from the static code analysis. Throughout the optimization process, most

of these had been fixed automatically. There were only a handful of type conversion-warnings left to fix.

15 Discussion

It is difficult to tell how much faster the final version of the program runs than the original, since the original probably would run out of memory before it even finished (if it ever finished), but the final version (with original configurations) at least ran about 390% faster than the original version, which is beyond what was expected. Here should be noted again that these results are generated on a 32-core machine. On a 12-core machine, the speed of the final program scaled down close to linearly with the number of cores available. The percentage difference between the single-threaded original version and the final optimized version is therefore expected to differ across different machines.

The largest discovery of this exercise was the impact that static variables have on performance due to their cache-unfriendly nature. The impact that cache misses have on multi-threading is also astoundingly large. Running multiple threads with the original RandomFloat-functions (which had a lot of static variables) yielded worse results than running single threaded, which was an eye opener for the importance of cache coherency when utilizing multiple threads.

Regarding the final state of the code, it is far from pretty due to a lack of time. One could spend hours, refactoring and making the style consistent. Further optimizations could also be made, like implementing SIMD-supported operations for the matrix math. An attempt at adding SIMD to the mat4's transform-function did work, but it yielded no better performance (at least on the AMD machine that was used), so it was deprioritized.

Lastly, the project files are reorganized into an engine-folder (for common code) and a projects-folder containing a main-folder (for the graphical program), and a tester-folder (for the benchmark program). To build the visual studio solution, simply use CMake.