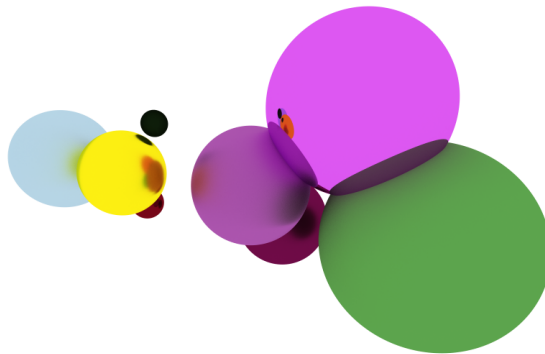


Optimization

Fixing a Slow Raytracer



Spelkonsoller och System, S0017D

Author:

Fredrik Lind (efilj-7@student.ltu.se)

Supervisor:

Fredrik Lindahl



Institutionen för system- och rymdteknik
October 2, 2020

Abstract

An experiment was carried out in order to gain understanding of how to profile and optimize a program written in C++. A raytracing engine – used to render images with high realism and visual fidelity by simulating the flow of light – was provided, intentionally poorly written. There were severe inefficiencies, as well as memory leaks. Various linters and profilers were used in order to investigate hotspots and memory usage. The data was used to fix any issues in the original source code. Next, the entire program flow was restructured to allow for more efficient execution, as well as facilitate multi-threading.

In the end, an approximate 200 times runtime improvement could be achieved.

Contents

| | | |
|----------|--------------------------------------|-----------|
| 1 | Introduction | 1 |
| 2 | Theory | 2 |
| 2.1 | Raytracing | 2 |
| 2.2 | Optimization and Profiling | 2 |
| 2.2.1 | Static Code Analysis | 3 |
| 2.2.2 | Dynamic Code Analysis | 3 |
| 3 | Method | 4 |
| 3.1 | Code Analysis | 4 |
| 3.1.1 | Static | 4 |
| 3.1.2 | Dynamic | 4 |
| 3.2 | Program Flow | 5 |
| 3.3 | Parallelization | 8 |
| 3.4 | Polymorphism | 8 |
| 3.5 | Render Passes | 8 |
| 3.6 | Miscellaneous Changes | 10 |
| 4 | Result | 11 |
| 4.1 | Code Analysis | 11 |
| 4.1.1 | Logic Issues | 11 |
| 4.1.2 | Performance Issues | 12 |
| 4.2 | Performance Gain | 13 |
| 4.2.1 | Fixes to the Original Code | 13 |
| 4.2.2 | Changes to Program Flow | 13 |
| 4.2.3 | Improved Multi-threading | 14 |
| 4.2.4 | Vector Precision | 14 |

| | | |
|----------|------------------------------|-----------|
| 4.2.5 | PRNG Bottlenecking | 15 |
| 4.2.6 | Final Result | 15 |
| 5 | Discussion | 16 |

1 Introduction

For the experiment, a raytracing program was provided, complete with an OpenGL render window. At very low resolutions, the program ran at near real-time. At higher resolutions, however, it became apparent performance was an issue. The task was to optimize this program as best as possible.

A benchmarking program was written as a command line wrapper for the raytracer, allowing a user to request images with various parameters. It was important to ensure consistent and reproducible results. A custom pseudorandom floating-point number generator was implemented for this reason, which allowed for seeding.

Various methods were then used to improve the quality and execution time of the render.

A simple test script was written to assist in running the program multiple times to get an average running time, as well as log the result to a file.

This report documents the progress and results of the optimization process.

2 Theory

2.1 Raytracing

A raytracing engine creates an image by calculating the paths of individual rays of light, travelling through a scene, and estimating their colors based on the material properties of the objects they collide with.

As opposed to a rasterizer, which produces decent results quickly by drawing models made up of polygons, shaded with approximated lighting, a raytracer can produce highly realistic results with significantly higher render times.

For this reason, rasterizers are typically used in games and real-time media, while raytracers are used in movies and CGI.

In order to get a smooth and anti-aliased result, each pixel in a raytraced image is given a fair number of rays, the color of which are then averaged together, in order to produce a smooth and pleasant transition between colors.

2.2 Optimization and Profiling

Getting a program to run quicker is a multi-faceted problem. No-one writes perfect code (it can be argued that almost no-one writes good code), and it is easy to make mistakes that causes degradation of your program performance. Copying values around incessantly, unnecessary loops, and having variables be mutable when not needed are examples of small performance drains that are easy to miss. If you identify enough of these small issues you can gain a decent runtime improvement. However, most modern compilers can do this for us, resolving these issues at compile time.

2.2.1 Static Code Analysis

Before the program is compiled, you can use a code checking tool - commonly referred to as a "linter" - to analyze the quality of your code. In modern design environments, these are often integrated and running continuously, allowing one to fix issues as they arise, rather than at a later stage.

The linter can for example help you discover unused variables, variables that could be assigned as constant, or unnecessary copy operations. While the compiler could probably fix a lot of these issues more or less automatically, it's always better to not take any chances.

Popular choices include *cppcheck* or *clang-tidy* for C++, or *Resharper*.

2.2.2 Dynamic Code Analysis

In order to identify the larger issues, the ones actually responsible for the poor performance of a program, profiling tools can be especially helpful.

There are many to choose from, where popular alternatives include *valgrind* and *perf*. These programs can be used to analyze the memory usage or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls.

Different profilers work in different ways, with different levels of accuracy. Statistical profilers typically sample the call stack at regular intervals. This approach is theoretically less accurate than, for instance, an instrumenting profiler, which modifies the program itself to gain accurate results. However, this modification - depending on the method used - can introduce ephemeral bugs, unintended behaviour, and inaccurate result. Since a sampling profiler allows the program to run at close to full speed, unmodified, they avoid these issues, and typically attain very accurate results.

With the data from the profiler, it is then possible to investigate the biggest bottlenecks, and spend time optimizing where it actually matters. With repeated profiling, it is possible to make significant improvements in running time. For this reason, a profiler is an essential tool for any developer concerned with speed.

3 Method

3.1 Code Analysis

3.1.1 Static

The original source code was put through an array of linter programs, namely *PVS Studio*, *cppcheck*, and *clang-tidy*. Issues were noted and later fixed.

3.1.2 Dynamic

The compiled program was put through various profilers.

The free open-source program *Valgrind* was used with the *Memcheck* tool enabled, in order to track leaks and allocation errors.

From the same suite, the instrumenting profiler *Callgrind* was utilized to find causes for slow execution.

The statistical profiler *AMD μ Prof* provided complementary data, which, when cross-referenced with the Callgrind output, provided a very good insight in the worst offenders with regards to performance.

3.2 Program Flow

After improving the original code, the next logical step was to work on the program flow itself, which was far from ideal. Rays were produced with nested (x, y, ray) loops, and all bounces were calculated to completion by means of recursion. This produces issues with memory alignment – the CPU is rendered unable to predict future instructions, which results in sub-optimal cache usage.

```
1 Color Raytracer::TracePath(Ray ray, unsigned n)
2 {
3     vec3 hitPoint;
4     vec3 hitNormal;
5     Object* hitObject = nullptr;
6     float distance = FLT_MAX;
7
8     if (Raycast(ray, hitPoint, hitNormal,
9               hitObject, distance, this->objects))
10    {
11        Ray* scatteredRay = new Ray(
12            hitObject->ScatterRay(ray, hitPoint, hitNormal));
13        if (n < this->bounces)
14        {
15            return hitObject->GetColor()
16                * this->TracePath(*scatteredRay, n + 1);
17        }
18        delete scatteredRay;
19
20        if (n == this->bounces)
21        {
22            return {0,0,0};
23        }
24    }
25
26    return this->Skybox(ray.m);
27 }
```

Listing 1: The original path tracing function. Note the use of recursion on lines 15-16.

In order to align the data more efficiently, the recursion must be removed. The idea is to ensure each batch of rays per pixel lies sequentially in memory, which will let us average the ray color easily. This data-oriented approach will also make multi-threading the application much easier.

Table 1: Demonstration of desired ray memory alignment.

| | First pixel | | | Second pixel | | | Third pixel | | |
|-------|-------------|--------|--------|--------------|--------|--------|-------------|--------|--------|
| X, Y | (0, 0) | (0, 0) | (0, 0) | (1, 0) | (1, 0) | (1, 0) | (2, 0) | (2, 0) | (2, 0) |
| Ray # | Ray 0 | Ray 1 | Ray 2 | Ray 0 | Ray 1 | Ray 2 | Ray 0 | Ray 1 | Ray 2 |

The main raytracer flow was rewritten almost in its entirety to facilitate the more efficient data alignment, as well as implement multi-threading. At this point all unnecessary dynamic memory allocations were also purged from the application, for instance the creation of spheres and rays – simple structures that do not require dynamic allocation.

The three nested loops (x, y, ray-per-pixel) that were used to create rays were replaced with a single loop, where pixel coordinates were instead calculated from the ray index.

```
1 // Setup ray buffer for a new trace pass.
2 for (size_t i = 0; i < ray_count; i++)
3 {
4     int x = ((i / rpp) % pass_width) + x_offset;
5     int y = ((i / rpp) / pass_width) + y_offset;
6
7     float u = ((float)(x + rng.fnext())
8               * (1.0f / owner->width)) * 2.0f) - 1.0f;
9
10    float v = aspect * (((float)(y + rng.fnext())
11                       * (1.0f / owner->height)) * 2.0f) - 1.0f);
12
13    // ...
14 }
```

Listing 2: Modulus and division is used to calculate the coordinates for each ray.

The ray objects themselves also required changing. Since the previous variant used recursion to calculate the color of each complete traced ray, the color could simply be passed from the intersection functions, up through the recursion stack, and immediately printed to the frame buffer. The new system was designed to "march" all rays forward sequentially, one bounce at a time – and so the rays themselves needed to carry color. An easy fix: all rays were given an initial color of (1, 1, 1), white light. As they marched through the scene, their color was multiplied with that of the material of the hit object.

```

1  for (size_t b = 0; b < bounces; b++)
2  {
3      // ...
4      for (size_t r = 0; r < ray_count; r++)
5      {
6          // ...
7          if (finished[ray_index])
8              continue;
9
10         owner->raycast_spheres(
11             origin[ray_index], direction[ray_index], res);
12
13         // Add raycast functions for additional shapes here
14
15         switch (res.shape)
16         {
17             case Shapes::None:
18             {
19                 finished[ray_index] = true;
20                 color[ray_index] *=
21                     owner->skybox(direction[ray_index]);
22                 break;
23             }
24             case Shapes::Sphere:
25             {
26                 // ...
27             }
28             default:
29                 fprintf(stderr,
30                     "Undefined shape type %i hit!\n", res.shape);
31                 break;
32             }
33         }
34     }
35 }
36 }

```

Listing 3: The new trace function utilized iteration rather than recursion.

Cutting out the recursion meant significant changes to the ray tracing method itself. The goal was to store all ray data in a array (vector), then repeatedly iterate over it, marching each ray forwards one bounce, until the bounce limit is reached. An early exit was added to ensure no calculations were performed on rays that had already exited the scene.

3.3 Parallelization

With the flow simplified, adding multi-threading was a lot easier. A user-defined number of threads were started in the main trace function. Each thread was allotted a section of the prepared ray data. The threads would iterate over the rays, march them forward, and exit once all rays are finished or the bounce limit was reached.

After all rays were finalized, a number of render helper threads would be spawned, averaging all rays-per-pixel together and adding the final color value to the framebuffer.

3.4 Polymorphism

The raytracer in its original form could only render spheres, which were inherited from a base class. Object oriented programming is known to be a trade-off; trading "programmer performance" for program performance. In other words, the program can become less performant, but easier to comprehend. It was therefore decided to remove the polymorphism, while – at least to some extent – retaining the possibility of adding additional shape renders in the future.

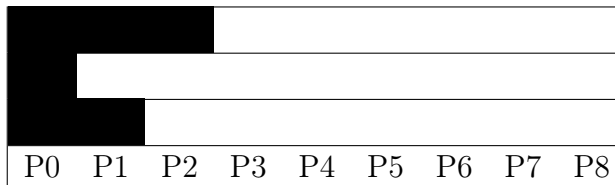
The object class was removed, all relevant data being moved into the sphere class itself. The array holding object pointers for intersection tests was replaced with an array holding spheres, as stack allocated data. The intersection function was rewritten in order to solely facilitate spheres. Additional intersection tests can be added after all sphere tests have concluded.

3.5 Render Passes

Allocating space for all the rays used for the scene in advance is efficient in regards to memory alignment. However, as resolution and rays-per-pixel increases, the memory usage becomes significant. While this wouldn't be a problem for the intended test parameters of the program, it would definitely hinder any attempt at rendering a high-res image of high quality. In fact, during testing the program rendered the lab computer unresponsive a number of times.

A system for rendering in passes was devised: the requested image resolution is divided vertically between threads (meaning each thread gets assigned y/T rows of pixels, where y is vertical resolution, and T is the number of threads). Each thread then renders its assigned block of pixels in a number of passes, moving horizontally along the image. The number of passes is decided based on the amount of RAM dedicated to the process.

Table 2: *Demonstration of ray passes, with threads on the Y axis and passes on the X axis.*

| | | | | | | | | | |
|----|--|----|----|----|----|----|----|----|----|
| T0 |  | | | | | | | | |
| T1 | | | | | | | | | |
| T3 | | | | | | | | | |
| | P0 | P1 | P2 | P3 | P4 | P5 | P6 | P7 | P8 |

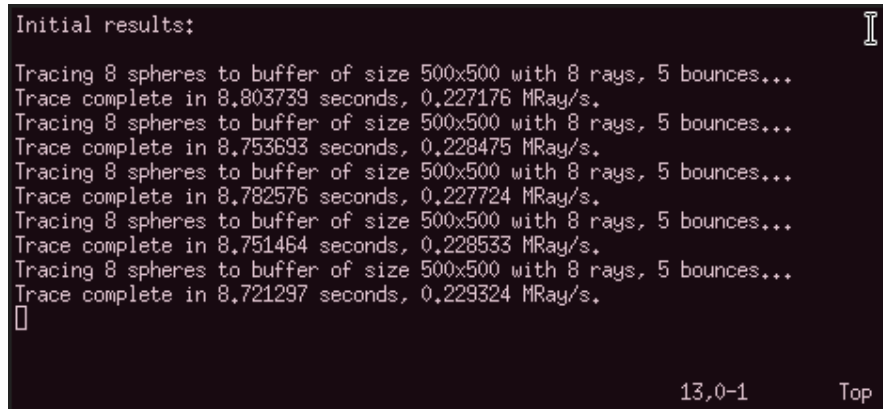
In conjunction with this change, the multi-threading flow was altered as well. Spawning threads can be a costly operation. Rather than letting the trace helper threads exit before spawning new threads for rendering, the jobs were combined into one. After tracing, the thread would perform the per-pixel color averaging to the framebuffer, before finally exiting.

In order to minimize the number of memory reservation calls for the ray vector, it was decided all threads should share a single vector, with offsets dictating where they were allowed to read and write. At this point, the vector of ray structs that had been used originally was broken into several vectors, each holding a single data type that had originally been stored in a struct. The reason for this was to – potentially – increase memory read efficiency. Another benefit was that a boolean field contained within a struct was padded to a single byte, while a vector of bools works as a bit field, compressing data eightfold.

3.6 Miscellaneous Changes

- The double precision `vec3` was shortened to 32-bit floats, as it resulted in faster trace passes with no noticable lack of image fidelity.
- The `Color` struct was replaced with `vec3:s`, as they are functionally identical, reducing casts and allowing optimizations in `vec3` to be used for colors as well.
- Each thread was given its own instance of a pseudorandom number generator. Previously they had shared, leading to huge memory bottlenecks.
- The `vec3` class was given a move constructor, allowing for more efficient data reuse.
- Function calls in iterator loops, such as `vector.size()`, was replaced with constant variables.
- C-style casts were replaced with modernized versions from the C++ standard.

4 Result



```
Initial results:
Tracing 8 spheres to buffer of size 500x500 with 8 rays, 5 bounces...
Trace complete in 8.803739 seconds, 0.227176 MRay/s.
Tracing 8 spheres to buffer of size 500x500 with 8 rays, 5 bounces...
Trace complete in 8.753693 seconds, 0.228475 MRay/s.
Tracing 8 spheres to buffer of size 500x500 with 8 rays, 5 bounces...
Trace complete in 8.782576 seconds, 0.227724 MRay/s.
Tracing 8 spheres to buffer of size 500x500 with 8 rays, 5 bounces...
Trace complete in 8.751464 seconds, 0.228533 MRay/s.
Tracing 8 spheres to buffer of size 500x500 with 8 rays, 5 bounces...
Trace complete in 8.721297 seconds, 0.229324 MRay/s.
□
13,0-1 Top
```

Figure 1: Measured successive running times of the original program.

The raytracer originally had an output of approximately .2 MRays/s when run with the test parameters (8 spheres to a canvas of 500x500 pixels, using 8 rays per pixel and 5 bounces), as evidenced by the figure. While the exact running time is, of course, hardware dependent, the figures are provided for rough comparison purposes.

4.1 Code Analysis

Static code analysis revealed a number of issues in the existing code. Most of these were logical errors, such as variables not being initialized upon construction of an object, which can cause unexpected behaviour. A good few were performance hindering issues, such as objects being copied rather than referenced.

4.1.1 Logic Issues

- Class members must be initialized. Using uninitialized memory causes undefined behaviour, and introduces bugs that can be hard to track down.
- Class members are initialized in the constructor body, rather than an initializer list, which can lead to reduced performance.
- Classes lacking copy and assignment operators is bad form when used with dynamic memory allocation.

- Constructors with single or defaulted parameters can be used for implicit conversion when calling functions or methods with incorrect types. If this is not intended, the constructor should be marked as *explicit*.
- Child classes must declare overridden parent methods with the *override* specifier.
- Dynamically allocated arrays of data *must* be freed with the *delete[]* keyword. Failure to do so causes leaked memory.
- Several objects are created by dynamic allocation, but never deleted, causing severe memory leaks.
- A child class overrides the virtual destructor of its parent, causing a memory leak. The parent virtual destructor should never be overridden, as it is run when deleting a derived object to remove parent data.
- Several values are passed by copy, rather than immutable (const) references, leading to significant performance degradation.

4.1.2 Performance Issues

Next, a series of dynamic code analyses were issued using the profilers *AMD μ Prof* and *Callgrind*. It became evident that the vector math library provided caused severe slowdown, for several reasons:

- A number of methods for checking whether a vector was zero or normalized were called in the constructor, resulting in several thousand calls to the vector length function – a fairly costly operation on account of its square root calculations. These checks were completely redundant, as the results were never consulted, and the methods themselves failed to account for floating point errors, and would have given incorrect results regardless.
- An initializer list constructor was provided in the vector class. While convenient for large object initialization, the implementation was significantly slower than a regular constructor.
- The constructor was called exceedingly often, as vectors were very rarely passed by reference. Constructing new objects are slow operations and should be avoided for rapid execution.

4.2 Performance Gain

4.2.1 Fixes to the Original Code

After fixing all reported linter errors/warnings, as well as removing some of the redundant code causing slowdown, the performance was immediately improved.

The most significant changes are listed below:

| Change | Avg. MRay/s |
|--|-----------------|
| Remove unnecessary calls to 'len' in vec3 | +0.33 |
| Remove initializer list loop in vec3 | +0.15 |
| Ensure objects are passed as const ref | +0.33 |
| Removed <i>volatile</i> ¹ keyword | +X ² |

Avg. runtime before/after changes: 8.80s / 1.93s

The changes made to the original program so far were minor, leaving the basic program flow untouched. Already, however, the average runtime had decreased by more than five times!

4.2.2 Changes to Program Flow

After redoing the program flow to allow for multi-threading, running time was – as expected – significantly improved. Note: All multi-threaded benchmarks are performed with the default setting, equal to the number of CPU hyperthreads.

| Change | Avg. MRay/s |
|---|-------------|
| Removed recursive calls and added multi-threading | +13.5 |
| Removed polymorphism from render objects | +0.15 |

¹The *volatile* keyword tells the compiler a variable may change frequently between accesses, even without influence from nearby code. It is a low-level keyword made primarily for hardware interfacing. Its performance impact can be significant, as it renders the compiler unable to perform basic optimizations.

²This change only has an effect on runs made with compiler optimized source code. These measurements were taken with optimizations disabled, and cannot be accurately measured.

4.2.3 Improved Multi-threading

After adding render passes and reducing the number of threads spawned and killed, a clear improvement could be noted. Sharing memory between threads in order to avoid memory reservation calls did not make a noticeable difference.

| Change | Avg. MRay/s |
|---|-------------|
| Minimized thread spawning and added render passes | +8.8 |
| Shared memory between threads | +0 |
| Ray constructor takes const vec3 ref | |
| Vec3 assignment operator returns const ref | +5.4 |

4.2.4 Vector Precision

It was decided vector calculations using 32-bit floating point precision should be enough for the purposes. Double precision requires double the memory bandwidth, and increases allocation times. The vector library was changed to use floats, with no noticeable drop in image quality. With this change, the Color struct was functionally identical to a vec3, and was therefore removed and replaced.

| Change | Avg. MRay/s |
|------------------------------|-------------|
| Halved vector data precision | +2.4 |
| Replaced Color with vec3 | +0 |

4.2.5 PRNG Bottlenecking

After additional profiling, it was discovered that pseudorandom number generator made up a large memory inefficiently. Since all threads shared a single instance of the generator, a lot of time was spent idle, waiting for memory access.

Giving all threads its own instance produced great results.

| Change | Avg. MRay/s |
|---|-------------|
| Removed shared RNG | |
| Added move operator to vec3 | +20.0 |
| Removed calls to vector.size() in iterators | +1.6 |

4.2.6 Final Result

Pre-optimization results:

| Parameters | Avg. Runtime (s) | Avg. MRay/s |
|--|------------------|-------------|
| 500 x 500, 8 rays per pixel, 5 bounces | 8.80 | 0.22 |

Post-optimization results:

| Parameters | Avg. Runtime (s) | Avg. MRay/s |
|---|------------------|-------------|
| 500 x 500, 8 rays per pixel, 5 bounces | 0.04 | 44.9 |
| 1000 x 1000, 8 rays per pixel, 5 bounces | 0.15 | 51.2 |
| 1000 x 1000, 16 rays per pixel, 5 bounces | 0.30 | 53.2 |
| 2000 x 2000, 1024 rays per pixel, 5 bounces | 73.1 | 56.0 |

The final, optimized product is approximately 200 times faster than the original version, when run with the standard test parameters. Unfortunately, the runtime data using different parameters for the original program was lost as a file was accidentally overwritten.

5 Discussion

The experiment revealed several points of importance to keep in mind when optimizing programs in the future:

- Use a profiler, or better yet - several. It saves you micro-optimizing with little to no result.
- Ensure constness whenever possible. It allows the compiler much more freedom in optimizing your code.
- Pass by reference whenever possible. It really makes a difference.
- Keep data in mind when writing your program – saving you a headache and huge rewrite when adding multi-threading.
- Don't ignore warnings about missing copy, move, and assignment operators.
- Consider your compiler settings – you can gain some performance just by setting a flag or two.
- Really think about what you're doing when recursion looks tempting.
- Write your code to minimize syscalls, like starting new threads or allocating memory.
- You can often get away with using the stack solely, without having to dynamically allocate anything – especially for simple programs.
- Multi-threading is easy in theory, and gives huge performance boosts, but can be a nightmare to debug.
- Trust the linter and compiler, and don't ignore warnings – they can have bigger performance impacts than you may believe.
- Consider whether you really need double precision or not.
- Be very careful when sharing data between processes. Bottlenecks arise easily, voiding a lot of the benefit given by the parallelization in the first place.