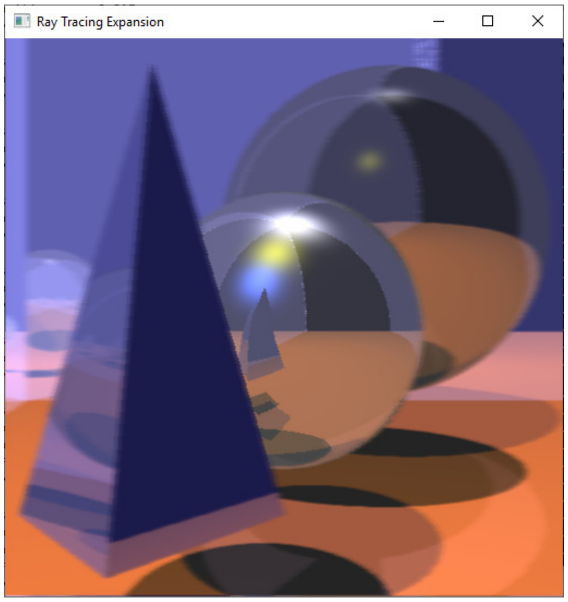
The Hunt for Real-Time Raytracing

A Report About Parallelization, and Failure in the Pursuit of

Anton Uklein
7739300

1. README



The following document assumes an introductory knowledge of the concept of raytracing.

Installation Guide

Extract all the contents of the **`*.zip`** into openGL solution's source files, in the directory with **`common.h`** and **`main.cpp`** and compile. This program *will* require them to be overwritten due to optimizations found during development, so ensure you have a backup before you overwrite them, as it may negatively affect the performance of other assignments.

Controls

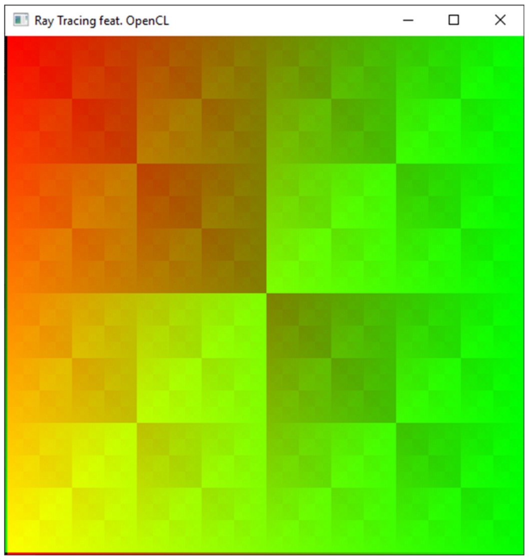
WASD – Allow you to move forward/backwards and strafe
Q/Z – Allows you to strafe in the vertical axis
E/R – Adjusts focal distance (E moves it closer to the camera, R moves it further away)
F/G – Adjusts camera aperture/intensity of depth of field (F moves it closer, R moves it away)
X/V – Adjusts camera field of view (X moves it closer, V moves it away)
C – Sets the camera's field of view to 75
Space – Toggles dithering effect on/off
Note that the dithering effect is automatically enabled every other frame

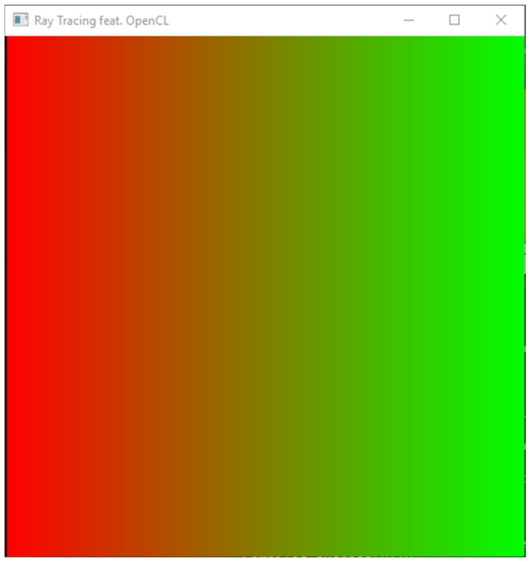
2. OpenCL

Raytracing is an embarrassingly parallel algorithm [1]. Each ray is an independent operation, and for every ray, we must test its intersection with every other object in the scene. This becomes very computationally expensive, very fast, as every pixel would need to test for every single object, increasing the performance costs with every additional pixel or object added to the scene. However, modern consumer-grade processors now come standard with 2 to 16 standalone processing cores, which could all be tasked to work on the same problem, leading to massive performance improvements, as more hardware becomes utilized.

This begs the interesting question, if a central processor is able to process this, why can't a graphics processing unit (GPU)? A high-end GPU from 2014, such as my Nvidia GTX 970 comes with 1664 cores standard [2], which given a 512x512 canvas, would theoretically allow it to render three rows of pixels are the same time. Granted, each GPU core is simpler than a CPU core, however, with historic demonstrations of real-time raytracing on consumer-grade hardware [3], which have become increasingly elaborate as technology marches on [4], the idea that a very parallelizable scene with a handful of primitive objects should be able to render at a sufficiently fast speed on hardware that can do over 1000 concurrent tasks doesn't seem insane.

However, as I don't know the hardware that my code may end up running on, any general-purpose GPU (GPGPU) implementations would have to be able to run on most hardware setups, or in a worst-case scenario, fall back on the CPU. This led me to two possible candidates for breaking up my task; *RadeonRays* and *OpenCL*. *RadeonRays* is specifically designed ray tracing software library capable of running on GPUs, which would also offer built-in acceleration structures such as more optimized BVHs, however, its poor documentation and lack of suitable examples on implementation turned me away from using it for this project. The alternative, is OpenCL, which is reasonably documented, runs very nicely with OpenGL, and recommended for heterogeneous computing regardless of user hardware.



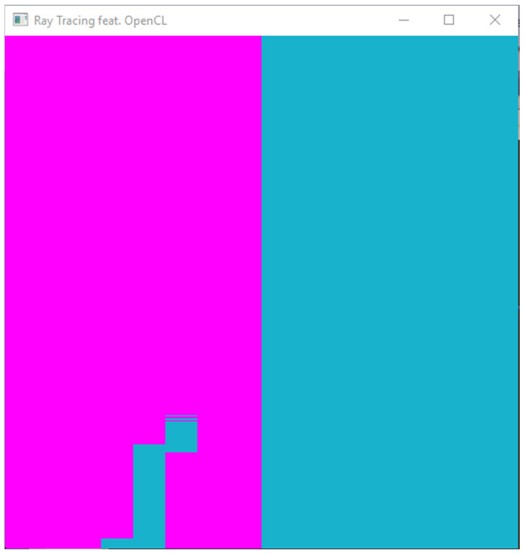


After experimenting with *RadeonRays*, *OpenCL*, by comparison, was relatively painless to install. This does not mean that it went without any trouble, as it requires references to specific locations on my hard drive, rather than as a subset of my solution, and due to some issue in **`cl.hpp`**, it is unable to run in debug mode, leaving me to have to solve any potential errors by myself by compiling only Release builds and running them standalone.

This wasn't enough of a deterrent. With a shockingly high time investment, I was able to get *OpenCL* working with OpenGL's buffers, being able to transmit any data back and forth as requested every frame. A simple test program was made that would have drawn two gradients based on the x/y position of the camera, which yielded… wrong results, see right. The fractal pattern was the result of a miscalculation on my behalf, however, even now I still struggle to understand how the top right corner has green pixels, as they were set by the red parameter, rather than the green.

After finding out the issue with the fractal pattern, which happened to be the result of type casting, a simple gradient where the red value is dictated by the X position, and the green value is set to 0.0f, I realized that OpenCL either requires a level of mastery in software engineering that I do not possess, or that my video card's VRAM is corrupt, both of which would be possible options. (This was later solved by changing how I interpret X, but I'm still not 100% sure what caused the original problem)

At this point, the amount of time I had was beginning to run critically low. I hoped that a quick and dirty port of the provided raytracer solution would have been a quick feat, until I realized that OpenCL is C, while our solution uses such "arcane features" of C++ such as polymorphism and placeholder typing. It's very nice code, but it wouldn't run without massive changes.



Literal hundreds of compiler errors later, and my implementation was, metaphorically, more embarrassing than the situation that resulted in the 10 frames-a-second 3DO port of Doom. While everything is sent properly, my mistaken belief in a forum post that "All entities through the cl API are pass-by-reference" [5] resulted in code that ended up being generally less than useless, as shown in the image to the right. For context, a magenta pixel means that it returned an invalid hit, while a deep cyan-line pixel means that nothing is rendered at this location. With only two days to spare before the due date, I scrapped this entire project, and returned back to the CPU raytracing example, with new ideas I had while working on this, and appreciation for anyone working with GPGPU solutions. A copy of my code has been provided, mostly as the base to actually connect with OpenCL is still in a functioning state, but the butchered OpenCL port of the code is less than useless, outside of the redesigned structs.

3. Optimization

With nothing to show, I began to look at the provided solution with two ideas in my mind to speed it up. Optimization can happen by either throwing hardware at a software problem, or by lowering the number of cycles required for the process.

The easier optimization was to lower the amount of rendering that needs to be done by half. If the vec3 array used for the scanline isn't cleared every row, it contains a copy of the last row obtained. By rendering every other pixel in the scanline, and then interpolating what would happen to the remaining pixels using a mix of their existing value and blending weighted towards the higher contrasted pixel nearby, we receive an image with a mix of antialiasing and slightly visible checkerboard aliasing, but with the benefit of nearly doubled performance.

However, meshes are still expensive to render, due to how many potential triangle intersections that exist. The easiest way to optimize this would be to cover the entire mesh in either a bounding box, or a bounding sphere, and checking if the ray intersects that before checking for all of the triangles within the mesh. The latter was the easier option to implement, as you could procedurally generate a sphere by finding the midpoint of a mesh, and taking the square root of each axes' lengths from the centre, to account for it being the radius. This wasn't something that was found online, this formula was made up by drawing a rectangle on a square and doing the Pythagorean theorem to try and figure out the radius, until I saw no need to square something we just found the square root of. This optimization led up to a 20% improvement in frametimes, depending on the scene; while a bounding box would outperform my code in basically all situations, my code was a one-line change and I was still pleased with the results I had.

Afterwards, the logic for each scanline was split apart and given to one of a number of threads, joining at the end of each scanline. While I was concerned about the overhead cost of implementing it in such a blunt and inelegant way, in my initial implementation, splitting up the logic from 1 thread to 8 was able to yield approximately 110-125% performance improvements in frametimes on my AMD Ryzen 5 3600, with 6 cores/12 threads on board.

Originally, using the most computationally expensive scene provided, the code took approximately 293.188 seconds to run, but with all the optimizations listed running at the same time, the same scene was successfully rendering in 57.114 seconds, *an improvement of 413%.*

Something still seemed off with the numbers, and after profiling yielded no results in what part of my code was performing inadequately, my attention focused on reducing overhead. The first option was to disable Visual Studio's debugger, and run the code in a Release configuration. This required the following three lines to be added to **`common.h`**:

```
#define FREEGLUT_STATIC
#define _LIB
#define FREEGLUT_LIB_PRAGMAS 0
```

With this change, the code could safely run as a Release solution, which, mixed with all the improvements, gave me extremely promising rendertimes, namely 18 seconds for the most computationally expensive scene when using 8 threads, and 17 and a half when using just one. This result was shocking, as it sped up my frametimes by thrice with just a simple change, but the lack of performance improvement worried me- if I was throwing hardware at a software problem, why is my software not running faster? Is there a bottleneck?

By running screen recording software, and playing it back frame by frame, I noticed that all the lines were being rendered smoothly one after the other, regardless of how many objects it had to intersect. This led to an idea- could OpenGL be the bottleneck causing performance issues? If I were to comment out **`glutSwapBuffers()`** each line, and add it at the moment when my console is updated with how long it took for a frame to render, the times improved even further to approximately 2-3 seconds per frame.

This was perplexing, there's no way that OpenGL had some sort of vertical synchronization enabled, so by changing my code back, and overclocking my monitor to 72Hz, I would theoretically have 20% improved performance compared to the existing 17-18 seconds, without changing a line of code. I ran this experiment and found that, yes, my code that took 18 seconds to run was taking shortly over 16 seconds consistently. The biggest bottlenecks to the code's performance at this point were OpenGL and Visual Studio themselves; OpenGL's double buffering caused the program to wait for 16ms every scanline, and dividing my resulting numbers by (height/refresh rate) yielded results that were very similar to my final findings.

As such, I ran the code in as many configurations as I could to document the performance differences, discovering that, in the end, my amateurish implementation of simultaneous threading was acting as a hinderance due to its overhead; a single thread that didn't have to create 8 threads and join them back together was running through the scanlines faster than all 8 threads combined. With all of my other performance improvements, the raytracer went from 293.188 seconds per frame initially to 2.156 seconds; ***an improvement of 13,499%***.

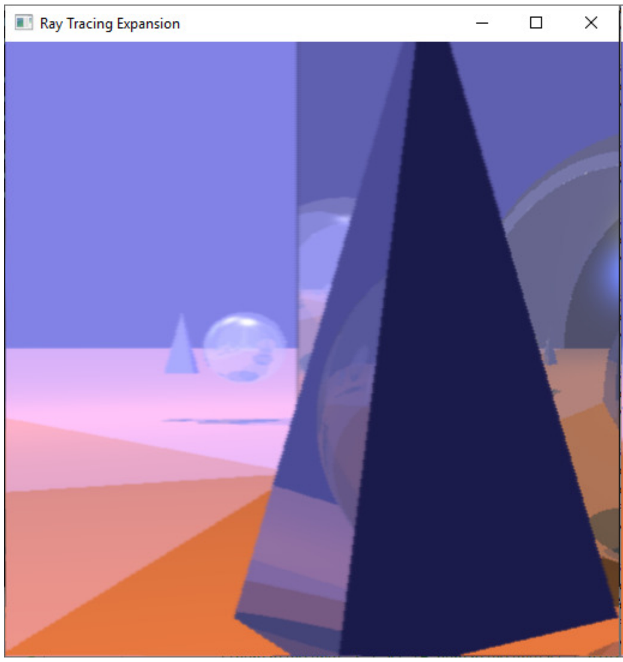
4. Performance Graph

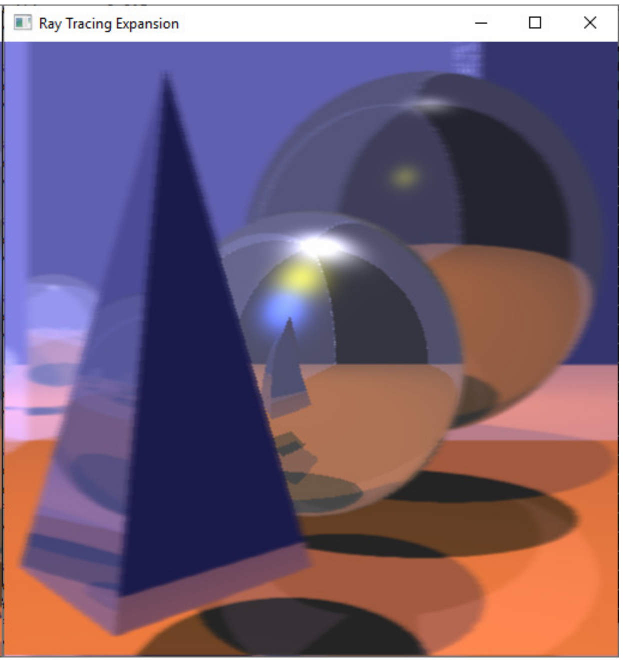
The code was tested on an AMD Ryzen 3600 6-core running at approximately 4.1GHz with SMT enabled, 32GB DDR4-3600 CL16 RAM, and a 3.5GB+0.5GB Nvidia GeForce GTX 970. When using multiple threads, a limit of 8 was chosen rather than 12 for more reproducible results.

Time	Threads	Checkerboard	Bounding Sphere	Debug Enabled	VSync Enabled	Relative Performance
293.188 s	1	No	No	Yes	Yes	100.00%
131.835 s	8	No	No	Yes	Yes	222.39%
239.496 s	1	No	Yes	Yes	Yes	122.42%
105.992 s	8	No	Yes	Yes	Yes	276.61%
150.203 s	1	Yes	No	Yes	Yes	195.19%
70.657 s	8	Yes	No	Yes	Yes	414.95%
122.122 s	1	Yes	Yes	Yes	Yes	240.08%
57.114 s	8	Yes	Yes	Yes	Yes	513.07%
17.473 s	1	Yes	Yes	No	Yes	1677.95%
18.168 s	8	Yes	Yes	No	Yes	1613.76%
4.616 s	1	No	No	No	No	6351.56%
3.868 s	8	No	No	No	No	7579.83%
3.840 s	1	No	Yes	No	No	7635.10%
3.308 s	8	No	Yes	No	No	8863.00%
2.748 s	1	Yes	No	No	No	10669.14%
3.118 s	8	Yes	No	No	No	9403.08%
2.156 s	***1***	***Yes***	***Yes***	***No***	***No***	***13598.70%***
2.936 s	8	Yes	Yes	No	No	9985.97%

5. Improving Image Quality

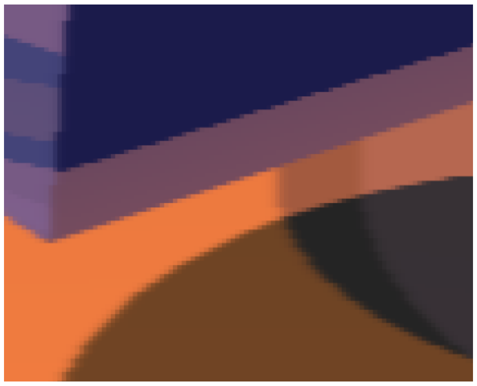
After spending literal days trying to squeeze out as much performance as is possible, very little time was actually left to adding in features. While I would have loved to have added a lookat towards the camera, I didn't understand how to manipulate the eye and s(x,y) transformations effectively, and ended up scrapping that aspect; however, camera movement does exist, and with it, depth of field via camera aperture and focal distance based off the location of the camera.



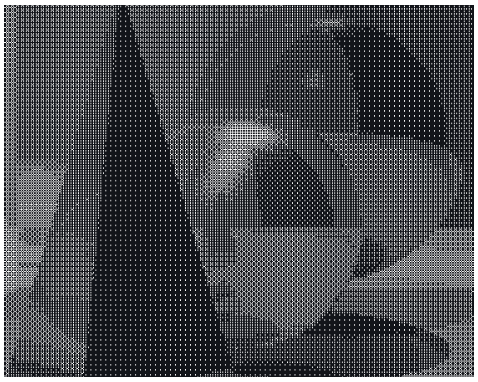


Top left: Camera translation and FOV manipulation.

Top right: Depth of field and camera aperture.



Additionally, due to how the checkerboard rendering works, it acts as a rough antialiasing algorithm, albeit only in the horizontal direction, and looks generally convincing mixed with the depth of field.



Lastly, there's also a pass every other frame to render out images as a Game Boy Printer would have, mostly for the fun of it. Theoretically, this could have been done on each independent object, as all the logic is still done on the CPU. As an idea, any ray that hits a mesh could have been forced to use the printer shader by double buffering a second texture consisting of nothing rendered but the mesh, however, due to the way I very roughly coded it using a bayer threshold map [6] mixed with greyscaling to the old BT.601 standard [7], a bug would exist in that it would only render the triangles and fill everything else as black, not render blacks properly, or not render anything at all.

6. Retrospective

I wasted so much time trying to get OpenCL working, if I were to have the project I have now and the time spent on that back, I'd rewrite the multithreading algorithm as a scheduler, so that I wouldn't have to deal with setting it up each scanline; giving each of them the next pixel to work on a double-buffered 2-dimensional texture and printing that out would have been the better result; I very likely would have breached the frames per second wall with this change, giving me an incentive to add onto the raytracing algorithm, making a final image based off a mix of raytracing, uniform random pathtracing, and cosine-distributed samples [8]. This isn't to say that I'm not pleased with finding optimizations to make the software given to me run nearly 150x faster, and already efficient software at that, however, there is more that I want to have been able to have done. To describe myself as in over my head for this project is a massive understatement.

7. Bibliography

As no standard was provided for the format, they are in the format of:
[reference number] – Author. "Title". URL to item.

[1] – Sorensen, Jordan. "Ray Tracing in Parallel". http://courses.csail.mit.edu/18.337/2009/projects/reports/jsorensefinalreport.pdf.

[2] – Nvidia, "GeForce 900 Series". https://www.nvidia.com/en-us/geforce/900-series/.

[3] – Pierce, Matt. "Realtime Raytracing". https://web.archive.org/web/20170114111504/http://mpierce.pie2k.com/pages/108.php.

[4] – NVIDIA GeForce YouTube Channel. "Battlefield V: Official GeForce RTX Real-Time Ray Tracing Demo". https://www.youtube.com/watch?v=WoQr0k2IA9A.

[5] – Howes, Lee, as AMD. "Re: Call-by-reference and const OpenGL objects". https://community.amd.com/t5/opencl/call-by-reference-and-const-opencl-objects/td-p/263582.

[6] – International Telecommunications Union. "Studio encoding parameters of digital television for standard 4:3 and wide-screen 16:9 aspect ratios". https://www.itu.int/dms_pubrec/itu-r/rec/bt/R-REC-BT.601-7-201103-I!!PDF-E.pdf.

[7] – Unknown, as Visgraph Laboratory. "Developed Algorithms - Ordered Dithering". https://www.visgraf.impa.br/Courses/ip00/proj/Dithering1/ordered_dithering.html.

[8] – Haines, Eric and Akenine-Mouller, Tomas and et al. "Ray-Tracing Gems". http://www.realtimerendering.com/raytracinggems/unofficial_RayTracingGems_v1.8.pdf.