Assignment 1: Raytracing framework

Arian van Putten - 4133935

December 6, 2017

1 Implemented features

1.1 Basic features

All basic features have been implemented.

1.2 Extra features

1.2.1 Vulkan and Rust

I opted to implement my ray tracer in the Programming Language Rust. A modern replacement for C++ that outlaws many of C++'s flaws. For example, use-after-free errors are caught at compile time. You get the safety of a GC-based language like C, with the performance of C++ or C.

Rendering is performed by Vulkan. Vulkan is the successor of both OpenGL and OpenCL. It gives very low level access to the GPU, both to the rasterizer and to the compute pipeline. Currently, the rasterizer is only used to blit an image on the screen.

At compile time, you can either enable or disable the compute pipeline. When the compute pipeline is enabled, a compute shader is loaded, instead of using the CPU ray tracer. For now, the compute shader just calculates a mandelbrot, to demonstrate that the Compute component of Vulkan actually works. In Assignment 3, I am planning to port the ray tracer to a Vulkan Compute shader.

Vulkan drivers don't speak GLSL, but speak SPIR-V, a custom portable bytecode for GPUs. Shaders can be written in any language, and can then be compiled to SPIR-V using LLVM and the Vulkan toolchain. In theory, this means I could write my compute shaders in Rust, but I haven't gotten it working yet. So now I currently use GLSL-Compute.

The most interesting feature of Vulkan is that there is no concept of a rendering thread, and all operations are non-blocking and return so called "futures" that allow us to asynchronously subscribe to when the GPU is done with your command buffer. This allows us to easily batch operations that are sent to the GPU, but also do operations in parallel to the GPU with ease without using a dedicated worker thread. I will use this in the future to rebuild the BVH in parallel during rendering.

1.2.2 Realistic refractions and Reflections

Realistic refraction has been implemented using Schlicks approximation of the Frensel equations and Beer's law for absorbance.

I decided to use Schlick's approxmiation for reflective surfaces as well. This gives a bit more realism to our reflective surfaces.

1.2.3 Triangle meshes and OBJ loader

Triangle meshes are supported. I also support loading in triangle meshes from OBJ files.

Currently, there are some limitations:

- Only one mesh per OBJ is supported, though the format ofifically supports multiple meshes per OBJ
- Material properties are not loaded from the OBJ, but set globally per mesh. However, changing this should not be too much work.
- Normal interpolation is currently not yet performed for the triangles, but could be added easily
- Texture mapping is not implemented

1.2.4 Multiple light sources and Area lights

Multiple light sources are supported. Furthermore, area lights are simulated by taking four random samples on a square. This creates nice diffuse shadows.

1.2.5 Multithreading

On the CPU, work is divided evenly over all cores. For this I use the `scoped_threadpool` library.

1.2.6 Performance tuning

```
83.82%  testit   testit                          [.] nearest_intersection
 6.71%  testit   testit                          [.] testit::tracer::trace
 2.38%  testit   libm-2.26.so                    [.] __expf_finite
 2.33%  testit   testit                          [.] <F as scoped_threadpool::FnBox>::c
 1.71%  testit   testit                          [.] half::convert::f32_to_f16
 0.81%  testit   libm-2.26.so                    [.] expf
 0.61%  testit   [kernel.vmlinux]                [k] entry_SYSCALL_64_fastpath
 0.42%  testit   [kernel.vmlinux]                [k] native_irq_return_iret
 0.21%  rustc    ld-2.26.so                      [.] do_lookup_x
 0.20%  testit   testit                          [.] <vulkano::buffer::cpu_pool::CpuBuf
 0.15%  testit   testit                          [.] expf@plt
 0.09%  rustc    ld-2.26.so                      [.] strcmp
 0.08%  rustc    ld-2.26.so                      [.] check_match
```

Performance tuning was done before implementing area lights.

Using the `prof` sampling profiling tool, it was found that almost all time of the program (82 percent) was spent in the intersection code, Further inspecting the assembly, I found out that most of that time is spent in the inlined version of the triangle intersection code so I decided to tackle only only that code.

First of all, we start with a classic high level optimisation. Early out. I enabled *backface culling* for diffuse materials. This improves the performance a lot! Without it, most viewing angles give us a seconds per frame of around 1.60 seconds. But with back-face culling enabled, seconds per frame drops down to 0.7 seconds at certain viewing angles.

Second of all, I have rewritten the triangle intersect code using SIMD intrinsics. (See `src/tracer/primitive/triangle.rs, src/vec/mod.rs`)

This brings down the average seconds per frame to 1.02 seconds, whilst the best case is still around 0.7 seconds. However, some small artifacts appear in the triangles. Either this is due to a bug, or due to the fact that the *Reciprocal* instruction is only an approximation. I have not yet had time to debug this, but would be interesting to look at in asssignment two.

A final profiling round shows that the handwritten SIMD code is now dominated by converting between SIMD `f32x4` and Rust's `Vector3`. I use `Vector3` in all of the other code, so I only convert to `f32x4` when doing the triangle intersect. But apparently, this has quite some overhead. However, due to time constraints, I was not yet able to move fully to `f32x4` for all vector arithmetic so I will leave this as a promising next step for Assignment two.

1.3 Build instructions

So this is a bit complicated. I'm working with a lot of new technologies, so building this can be non-trivial. The following things are needed, at least:

- A Driver that supports Vulkan. Intel and NVIDIA have good support
- Vulkan SDK
- a GLSL to SPIR-V Compiler (glslValidator has one built in)
- CMake
- Rust nightly (The unstable version of rust) for SIMD instructions
- Probably Linux (I have not tested this code on Windows at all. It probably doesn't build on Windows)

Finally, we build the binary but make sure to enable SIMD intrinsics `RUSTFLAGS="-C target-cpu=native" cargo run --release`

Once all these tools are installed, a simple `cargo run --release` in the project directory should build all the dependencies and fire up the ray tracer.

2 Sources

As I already told Jacco Bikker, last year I worked together with Renier Maas on his path tracer that he built for Advanced Graphics. I worked on parallelisation and safe multi-threading for the Conepts of Program Design course. Source code of that tracer can be found here: `https://github.com/arianvp/PathTracer`.

The free camera is based off the free camera I have had lying around since Bachelor year two, which was then used in a GPU Path Tracer assignment in the Concurrency Course, and is almost the same as the one used in Reinier's Path Tracer (As I implemented it there). It has all kinds of advanced movement, but also support for depth of field (if used in a path tracer), and changing the field of view.

Multi-threading is very similar as that previous path tracer (as I also wrote that). Chunks of memory are simply submitted to a threadpool using the `scoped_threadpool` library.

Finally I'd like to note that `https://www.scratchapixel.com/` is a *great* resource for learning about computer graphics. I referenced it a lot to first refresh my knowledge from the bachelor graphics course on basic shading. But also for building sphere intersections, and basic refractions.

Furthemore the following libraries have been used:

- *tobj* A port of tinyobjectloader for Rust
- *vulkano* Vulkan API for rust
- *cgmath* Library implementing vector operations
- *scoped_threadpool* Multi-threading library for rust