Rasteriser Overview

By Jamie Willis, jw14896 and Harry Mumford-Turner, hm16679.

Introduction

We implemented the base features from the coursework specification, then we added the below list of extensions to make the rasteriser look awesome! See `README` for setup instructions.

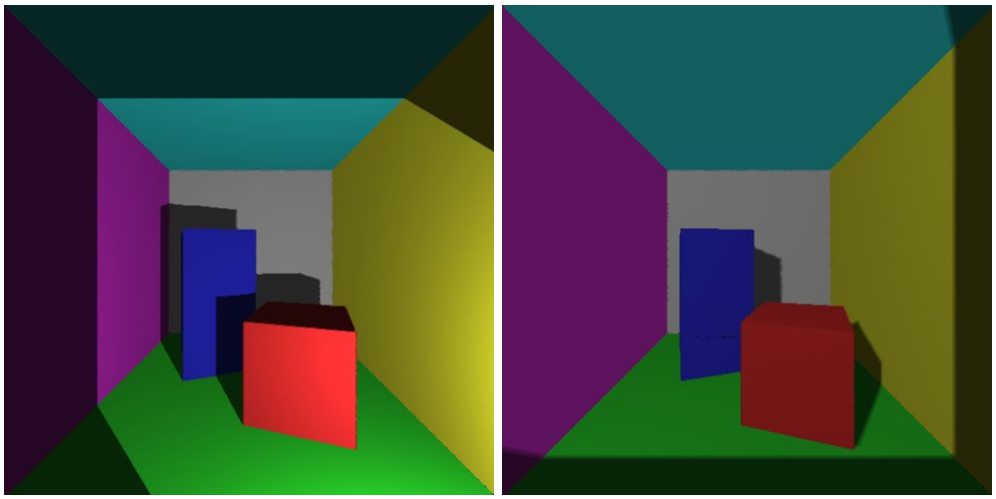

Fig. 1. Our final render without soft shadows (left) and with (right).

Contents of Extensions

Interpolation Changes & Basic Clipping 1

Frame buffer, Deferred Rendering and SoA 2

Directional Lights - Light as a Camera 2

Shadows - Shadow Maps 3

Soft Shadows 5

Anti-Aliasing - FXAA 5

Parallelisation - GPU 7

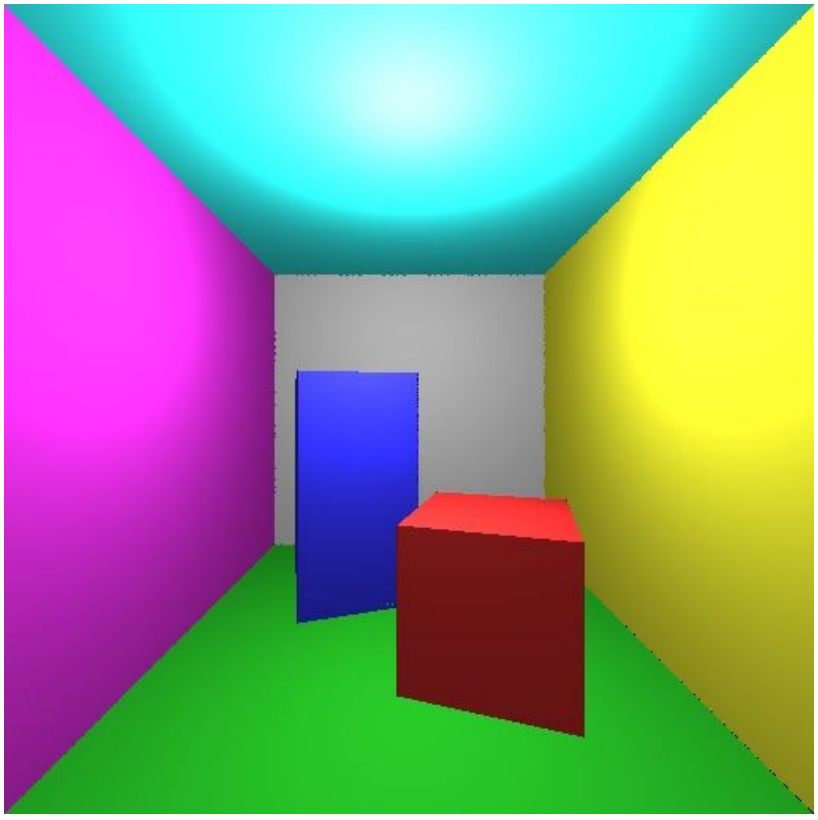

Fig. 2. Base render.

Interpolation Changes & Basic Clipping

We started off by moving to barycentric coordinates during the raster step, which massively simplifies interpolation and allows us to perform basic clipping very easily. It turns out that barycentric coordinate interpolation was actually faster than the original strategy in the first 50% and removed a lot of the problems with memory allocation for the pixel vectors.

The idea behind barycentric interpolation is that each coordinate inside a triangle is between (0, 0, 0) and (1, 1, 1) where the vertices are at (0, 0, 1), (0, 1, 0) and (1, 0, 0). What this means is that every coordinate in the triangle is a weighted contribution of all the properties at the vertices. That makes it very easy to interpolate; just multiply each component of the coordinate with the corresponding property at the vertex then combine them all.

The strategy for rasterising the triangles is now to iterate through the triangle's bounding box, find the pixels inside the triangle, and perform the calculations on it. Therefore, to perform basic clipping you just cap the iteration to the screen boundaries. To remove issues with triangles behind the camera, we simply clip by not rendering triangles with at least one point falling behind the camera. It's not perfect, but it doesn't suffer from any performance issues.

Frame buffer, Deferred Rendering and SoA

To make things much easier later on, we spent some time restructuring how the rendering is done. Firstly, if we want to have any chance of moving the pixel shader onto the GPU, we need to move it into an independent step. Namely, it should run after all the `drawPolygon` calls are done. This is similar to deferred rendering, but doesn't quite go to the same extremes.

To achieve this, we expanded on the idea of the depth buffer by adding all the information the pixel shader needs for a point (x, y) into an augmented buffer called a frame buffer. We store information such as normals, colours, 3d positions and depth. This now allows us to perform a separate loop over the depth buffer after the raster stage to execute the pixel shader.

To simplify things for the GPU down the line and improve performance (GPU's love coalesced memory access, i.e. it likes to pull down large chunks of data at a time: see diagram below), instead of having a single array where each element has all the data we need, we will store many arrays in a single struct so all the data elements are right next to each other in memory. This is called Struct of Arrays vs Array of Structs style.

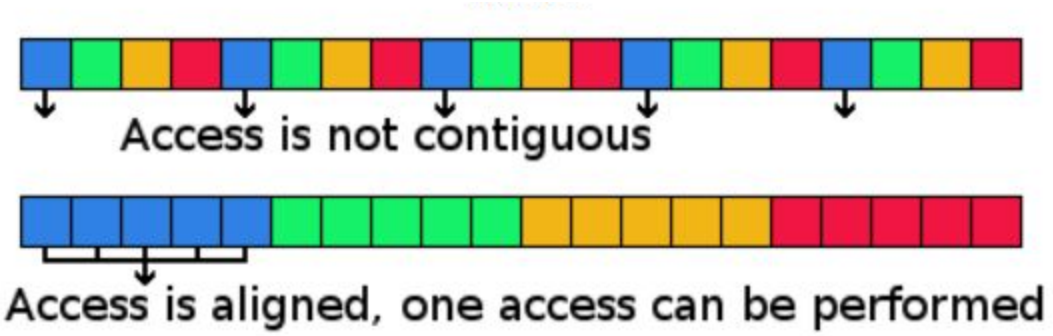

Fig. 3. Improved data alignment using SoA, GPU can pull all the data at once for each different data element.

Directional Lights - Light as a Camera

In order to make our way towards shadows, we needed to take a quick pitstop at directional lights. Though shadows for point lights are perfectly possible, the implementation for a directional light is much simpler and better highlights the premise.

The general idea is that the light source becomes a camera, and pixels are projected onto the surface of the light. If they are within the bounds of the light's resolution after projection, then they are considered to be lit up. This currently ignores occlusion, but we'll get to that! The following image shows the light pointing forward positioned somewhere in front of the camera.

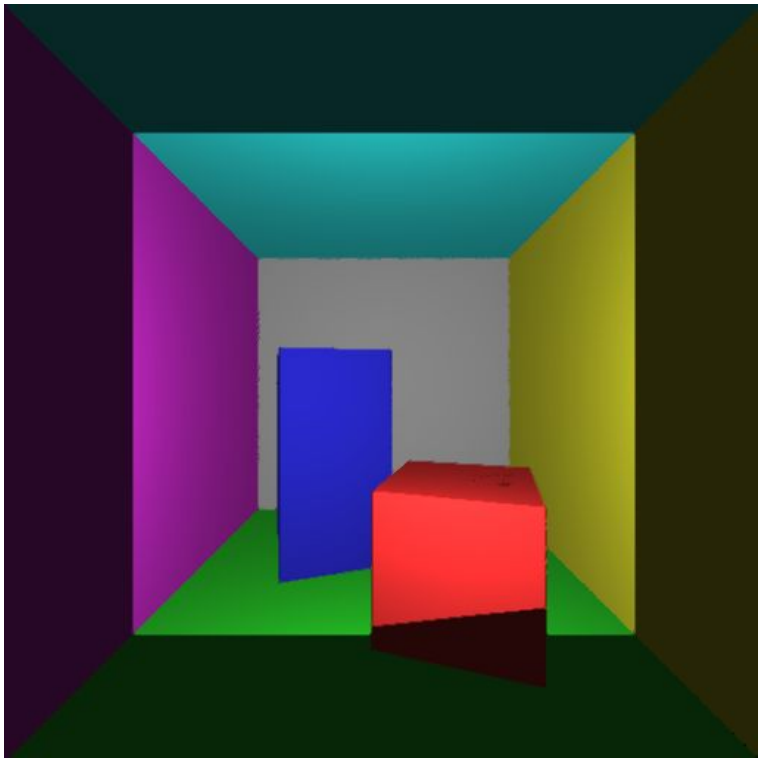

Fig. 4. Point light in front of the camera pointing forwards to show side shadows.

Though it is perfectly possible to position it on the ceiling and rotate it to point down, we've chosen not to do this as it's a lot harder to make it look nice from the ceiling, especially when there is no visible indication there is a directional light there.

Shadows - Shadow Maps

Now that we have directional shadows, it's very easy to get our hands on shadows. In the previous section we actually started out with some simple shadows that result from pixels not projecting onto the light's "screen". We can take this one step further; in the raster stage we will also compute the depth buffer with respect to the light source and store it separately. This is referred to as a shadow map. The idea is that this shadow map stores the positions that are closest to the light, and if a pixel that we sample is not the shallowest thing in the buffer, it must be occluded by something and therefore in shadow.

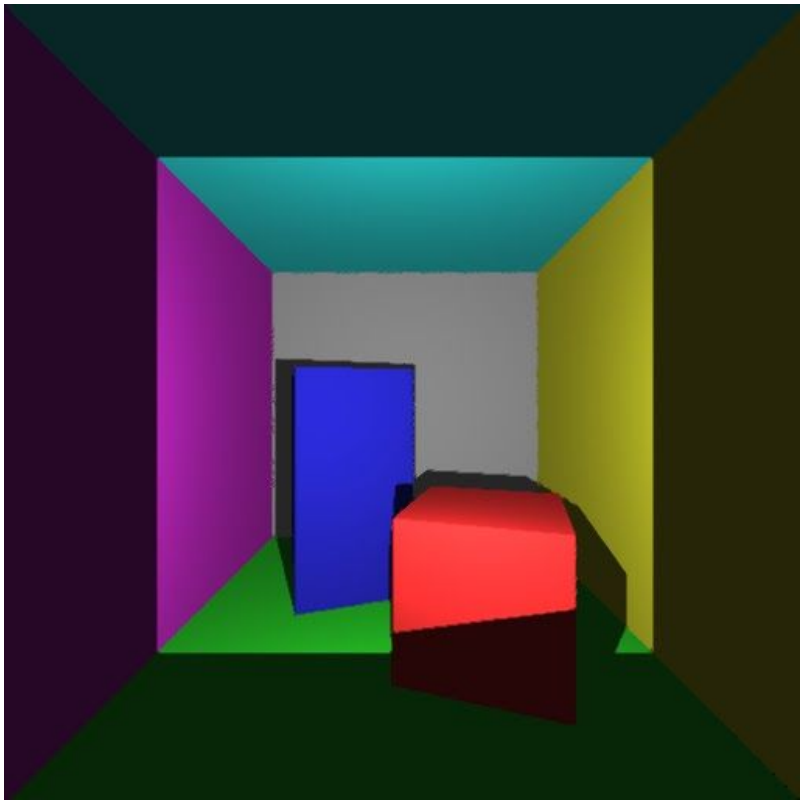

Fig. 5. Using the shadow map.

Once the shadow map has been generated, it's a simple one line change to get those shadows working in the pixel shader, which produces reasonably good shadows. However, it does suffer from resolution problems. The shadow map's resolution is fixed, so when we move the camera towards the shadows, the shadows will start to appear more pixelated, unless we move the light too. The fix is simple; increase the resolution of the shadow map. Here is a comparison from approximately the same camera position of what shadows look like close up on the back wall at both 512x512 and 2048x2048 shadow maps.

Fig. 6. A low resolution shadow map

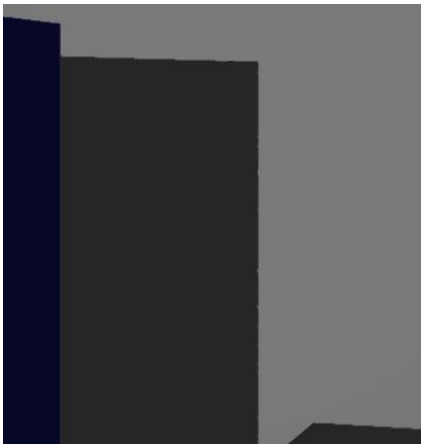

Fig. 7. A high resolution shadow map

Soft Shadows

After having implemented shadows, soft-shadows are very simple to implement. The premise is very similar to the soft-shadows in the raytracer; sample points in the light camera with slight offsets applied around the actual shadow sampling position then average the result. This looks fairly decent, but it's very difficult to get balanced, and runs much slower on the CPU (but fine on a GPU, as we'll see). Soft-shadows done in this manner suffer from a few artifacts that are difficult to get rid of. As a result you can revert to normal shadows by setting soft-shadow sample to 1.

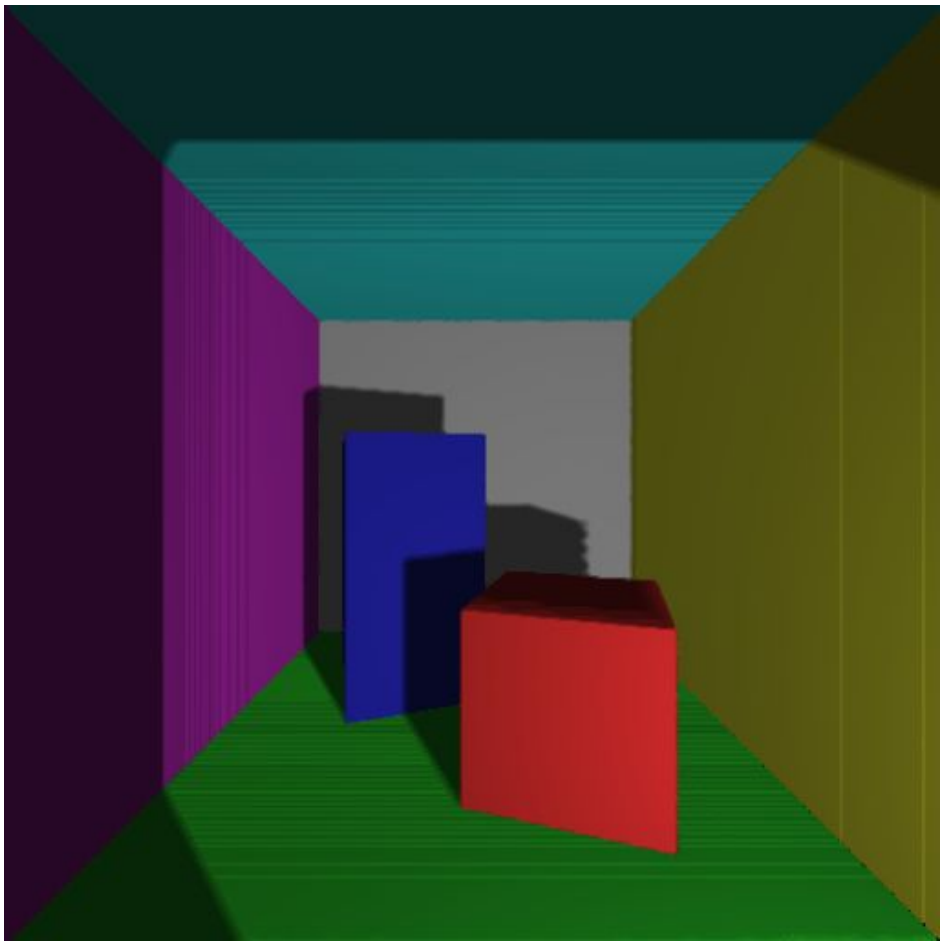

Fig. 8. Soft shadows with artifacts

Anti-Aliasing - FXAA

In contrast to the raytracer, speed matters here. As such, SSAA is infeasible. There are two real options we can take here; MSAA (Multi-Sample Anti-Aliasing), which runs during the rasterisation step, it's very taxing on performance and is quite difficult to implement and FXAA (Fast Approximate Anti-Aliasing) which is our choice. FXAA is a post-processing solution, which means it doesn't need any information about the scene, just the final pixels. It's also incredibly fast, especially when ran on the GPU, there is no framerate difference between having it on and having it off!

The general idea is to find edges in the scene using edge-detection kernels similar in premise to Sobel filters, and then blur along those edges if there is a strong contrast change. The result can therefore be a bit blurry and less accurate than other solutions, but still provides decent visual improvements.

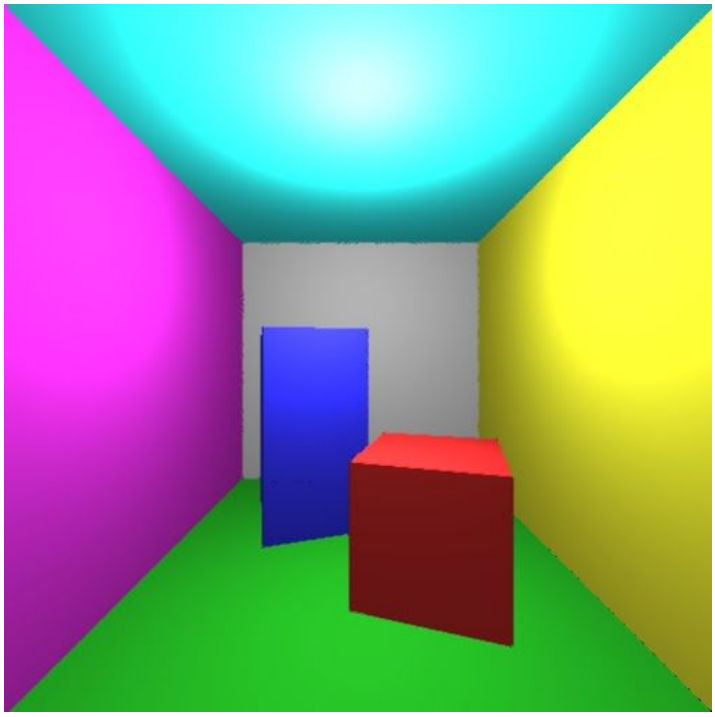

Fig. 9. Anti-Aliased render with FXAA

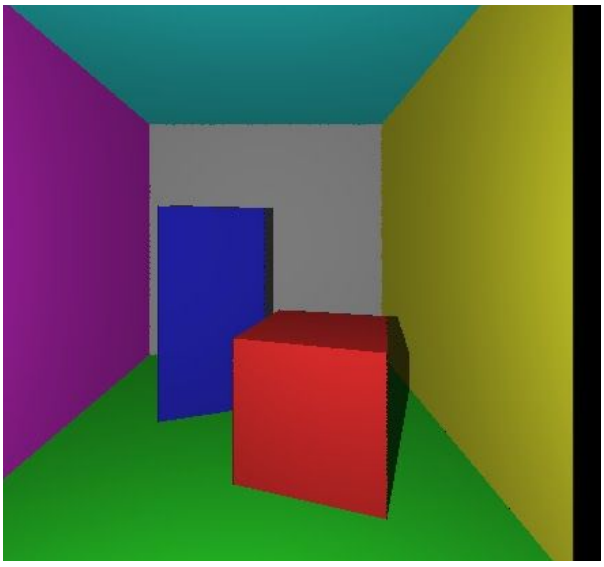

Fig. 10. Aliased

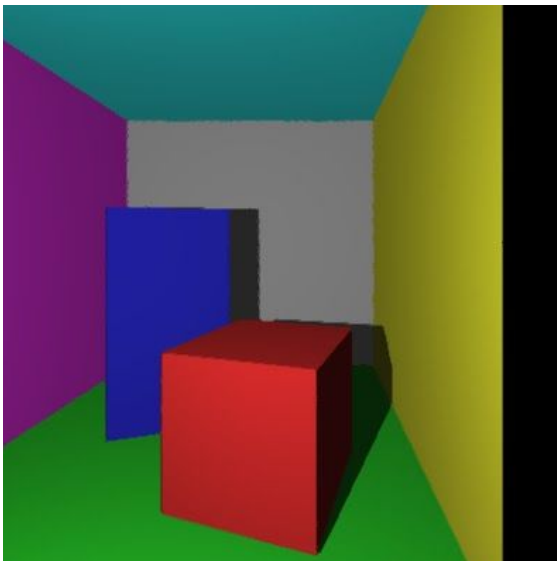

Fig. 11. Anti-Aliased render with FXAA

Parallelisation - GPU

With a rasteriser, the first and last of the three stages of rendering can be easy put onto a GPU (the second can be done in hardware on the GPU, but this isn't in spirit of the coursework). With such a low poly-count however, the first stage (vertex shading) is not really worth porting to a GPU, especially since we lose access to convenient matrix operations. This leaves us with the pixel shader and the post processing shaders.

In order to move our computation to the GPU, we used OpenCL which allows us to write GPU kernels, and load them in at runtime, sending data to the GPU from stage 2, running the shaders on all pixels in the frame buffer and then sending it back to the CPU to be put onto the screen.

The ports of the pixel shader and the fxaa shader are relatively straightforward and they are well suited to being ran on the GPU. Perhaps the execution of soft-shadows is a little inefficient on the GPU, because GPUs inherently dislike branching instructions and soft-shadows introduces a loop, but this could have easily been manually unrolled to aid the GPU.

The results of running on the GPU range from unimpressive to staggering. For instance, with just the simple pixel shader and fxaa, the GPU performs very slightly slower than a pure CPU implementation running OpenMP and 8 threads. This is because there is overhead in moving the data to the GPU and enqueuing the kernels. Since in this scenario the bottleneck is the raster process, both perform equally well. Where the GPU shines, however, is when the pixel shader has a lot of added complexity. When soft shadows were implemented, the GPU's higher levels of parallelism and vector operators really made the difference; for a 512x512 image running with 50 soft-shadow samples, the GPU was able to maintain 30fps, whilst the CPU was spending a colossal 1.7 seconds per frame! This is due to the GTX 970's 1664 compute cores, which dwarf the 8 cores of the CPU. The following youtube videos showcase our rasteriser running on the GPU.

50 shadow samples CPU vs GPU:
https://youtu.be/nlUu7aPz-H4

2048x2048 images on both CPU and GPU, comparable performance:
https://youtu.be/siGBA8brz9E

512x512 with 3 samples on GPU, 60fps:
https://youtu.be/HQDrsd6H4bY