COMS30115
Computer Graphics

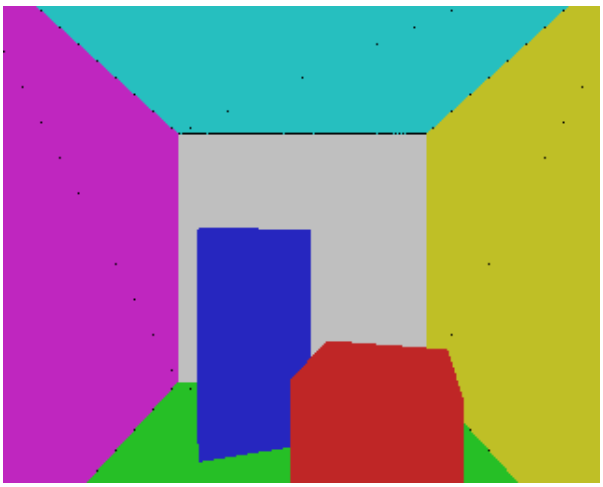
Luke Storry - LS14172

November 6, 2018

Raytracer

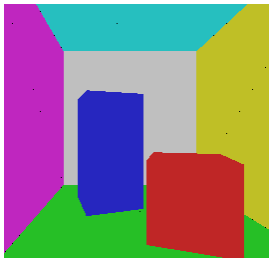
In the first part of the coursework I implemented a very basic Raytracer:

A set of triangle coordinates making up a coloured variation of the Cornell Box environment was loaded, then for each pixel on the screen, a ray was traced from the simulated camera, and the first detected intersection with a triangle determined that pixel's colour value.



Camera Movement

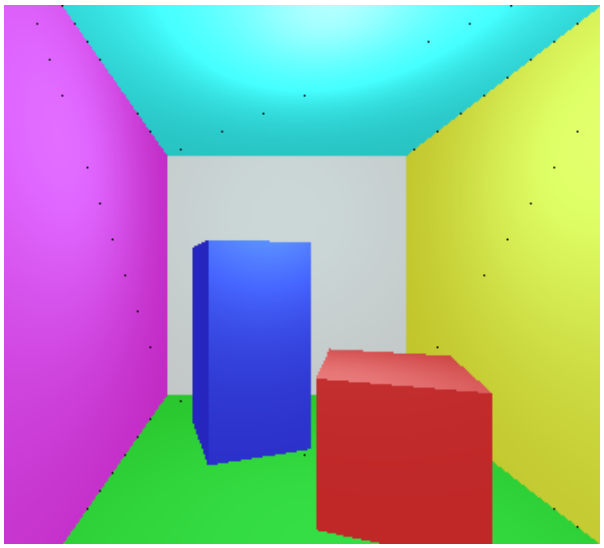
Next I added movement and rotation of the camera, by checking the keyboard for inputs each frame, and multiplying the camera's position coordinates by appropriate rotational and translational matrices.



To make real-time movement possible, I then incorporated OpenMP to parallelise the rendering. Only a few changes had to be made, as the act of looping through all pixels is a very nice loop to split between cores, as there are no critical sections other than the display, which can be collated at the end.

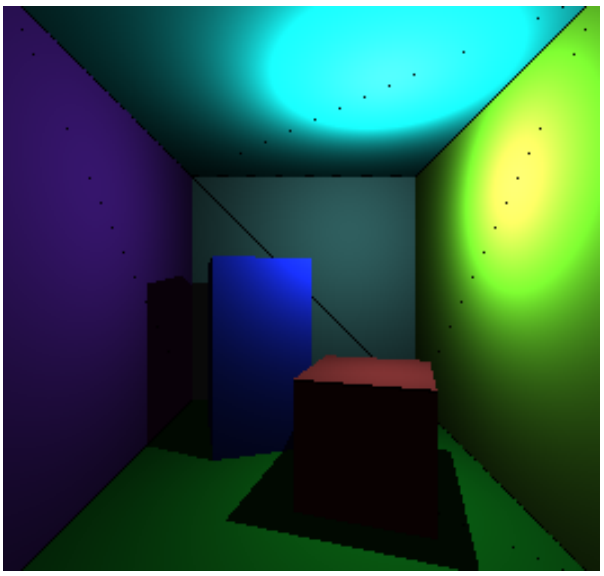
Direct Lighting

A Direct Lighting function was then added, which uses pre-calculated surface normals and a global light-position variable to determine the incident light upon each object, and multiplying this by the colour values from the previous section resulted in the below image:



Shadows

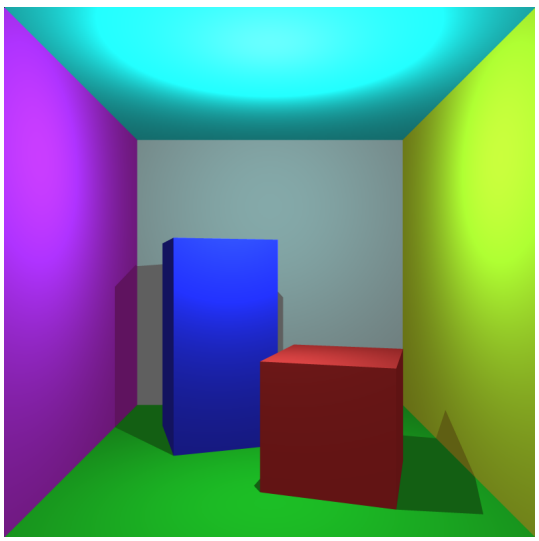
To make the lighting more realistic, this function was later extended to add shadows, by checking every light ray for the closest intersection, then comparing that value to the length of the light ray itself to determine whether other objects intersect it and cast a shadow:



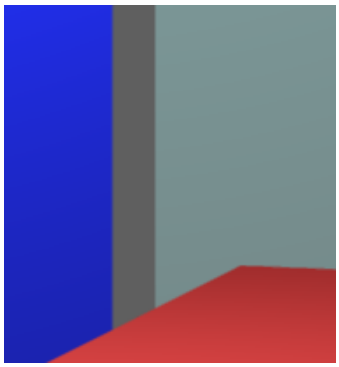
Anti Aliasing

Next was to add anti-alaising, which both smoothed all the edges and removed artifacts caused by the camera's rays penetrating gaps between triangles.

The method used was to simply add another loop inside the main per-pixel loop, which shoots off rays at a variety of offsets to the pixel's coordinates, then averages those values to give a much cleaner image:



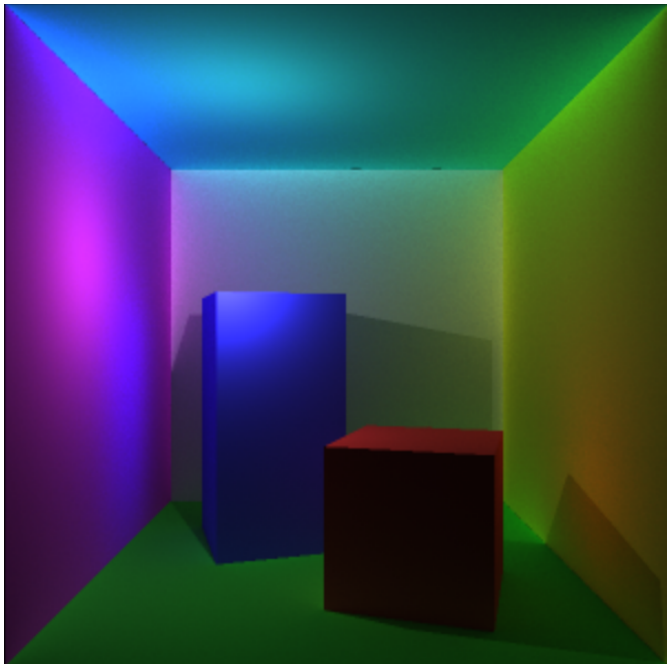
This effect can particularly be seen when zooming in on this section, which was previously very alaised.



Indirect Lighting

My final extension was to create a sort of global-illumination by extending the previous Direct-Lighting function into a recursive Indirect-Lighting function. This shoots 500 rays from every surface to determine the incoming lighting and colours using Monte-Carlo sampling.

Each of those 500 rays then "bounces" around the scene another two times, spawning 100 more rays per surface intercept, and each of these bounces collects the surface colour, the Direct-Light coming into that surface, and the angle of attack, meaning that every surface in the scene ends up with a very colour-mixed and diffuse lighting:



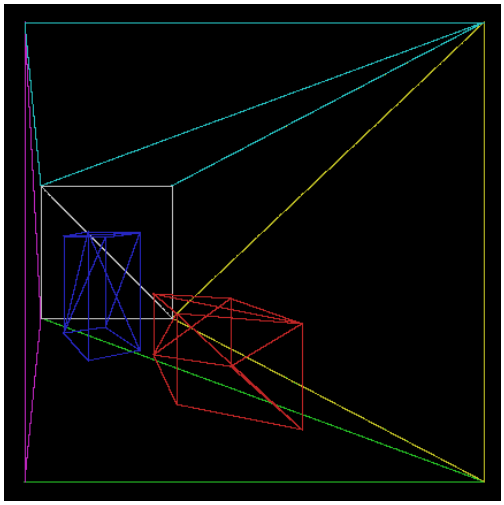
Rasterizer

Because the Raytracer is a very slow way of rendering it is not very useful for real-time usage, so next a rasterizer was developed.

Instead of looping through every pixel, this instead loops through every polygon in the scene, then projects its 3D coorrdinates onto a 2D image plane for display on the screen.

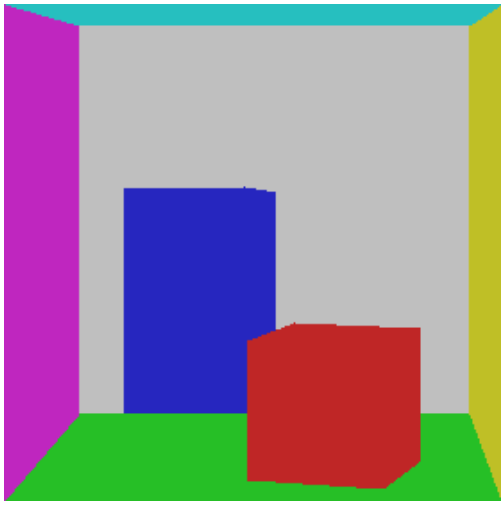
Vertex and Edge Projection

First was to mathematically calculate the projections of the vertices, then using linear interpolation to draw in the edges of each triangle between those points.



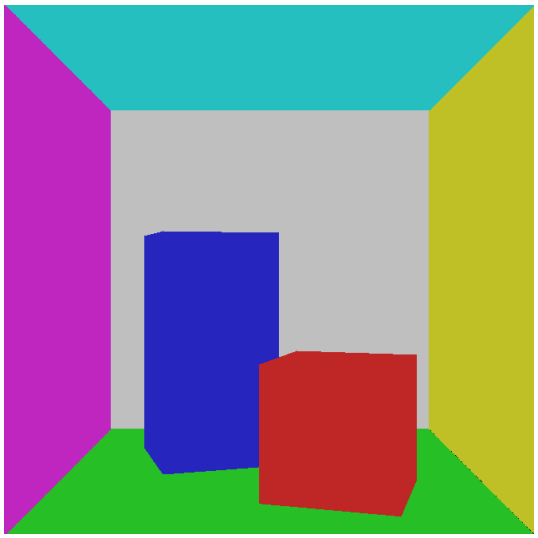
Interpolating for Fill

Next the interpolation was extended to also calculate the coordinates and colours of each pixel within the triangles.



Depth Buffer

The above images looked wrong because the program just drew the triangles in whichever order they were imported, so I next added a global depth buffer to store the greatest inverse-Z values per pixel, and only overwrite the pixel value if the new triangle had a higher inv-z value.

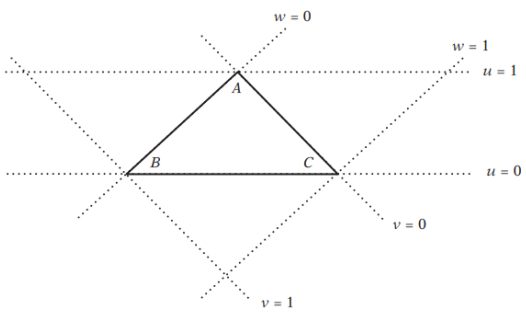


Barycentric Coordinates

I now had a simle and speedy rasteriser, however due to the way the edges of the triangles were calculated and stored in the interpolation function's implementation, there was a high chance of a segfault due to the size of an array tending to zero as you zoomed the camera in.

Thus a new method of interpolating the polygon's surface values, was needed, one that didn't store the coordinates of the edges.

In Christer Ericson's Real Time Collision Detection, I came across the concept of Barycentric coordinates, decribed by this diagram from page 50:



Instead of storing two edges of the triangles and interpolating each row, a bounding box is contructed around each triangle, and every pixel in that bounding box calls the pixel shader.

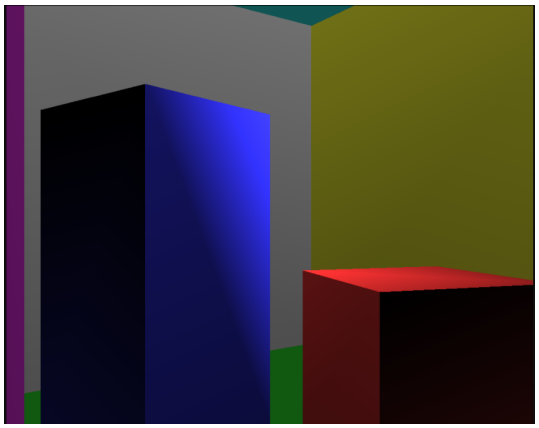
This shader then uses the vertex coordinates to convert the pixel's (x,y) coordinates into barycentric (u,v,w) coordinates, which are then used to interpolate the pixel's colour and depth values, as well as whether that pixel is within the triangle or not.

This implementation was not significantly slower, but did prevent the memory overload issues from the original implementation.

Per-Polygon Lighting

The vertex shader was then extended to work in a similar way to the DirectLight function of the raytracer, calculating the incident light and incorporate that into the vertex's colour.

These values were then interpolated across the surface to in the pixel shader, to give a smooth gradient between the different vertex colour-lighting values.



Per-Pixel Lighting

As can be seen on the blue rectangle in the above image, per-surface lighting calculations may be faster but can cause artifacts where one polygon is drastically different to a neighboring one.

Therefore the next thing added was per-pixel lighting, where instead the pixel shader calculated the lighting for each individual pixel instead of just interpolating it from the vertices' values.