REPORT

General – Raytracing

To begin the Raytracing assignment, we downloaded the skeleton and began following the instructions given on the sheet for Coursework 1. We understood the implementation of setting the screen parameters and made sense of the while loop's use to call the functions Update() and Draw(). The Update function contains the tools to translate and rotate the camera and light positions. To translate both the camera and light, we updated the camera/light position variable with vectors that we would essentially add on to the initial one so one axis would be incremented slightly to mimic movement. As for rotation, it was a little bit more complicated. We initially made the float variable 'yaw' global, but for some reason, the updates did not stick with the variable and the value returned to 0 every time. To overcome this, we moved the keystate implementation for rotation to within the while loop within the main; on the same level as Update and Draw. By doing this, we declared 'yaw' within main and updated it correspondingly within the while loop. In order to project the rays in a different direction to imitate a rotating camera, we passed the updated variable 'yaw' to the Draw function and multiplied its' direction variable by the matrix R (containing the updated yaw). To conclude, instead of testing for the rotation keys within the Update function, we tested for it within the main function and rotated the raytracing by multiplying the direction in Draw by R. Though this might not be the most straightforward route, it does not add much extra rendering time to the project and avoids our problem of 'yaw' not updating. Moving on, we followed the instructions given to implement a function to test for Intersection between the ray being casted and the triangle itself. After adding in the program's ability to perceive and compare the depth of triangles, we successfully produced the general, coloured, outline of the Cornell Box. Though most details were provided in the instructions, we came across some problems along the way. For instance, the direction was backwards and the parameters for testing intersection was unknowingly a bit off. To work through this problem, it was key to look at chunks of code at a time and work through the details with the both of us.

Now, we will move on to our procedures in Illumination. It was quite straightforward to achieve the visual image of direct light, as color and mathematical equations were given in the coursework. What was slightly hard in this part was to fully comprehend the algorithms that were being implemented and understand the roles they take in every part of the image. Because the shadows part did not have much instructions to be given, we had a few bugs to begin with but worked them out in the end. Indirect Illumination was also rather straightforward and we noticed that the image got a lot less harsh after the mathematic equations were successfully put into the existing program.

Extensions – Raytracing

The first extension we implemented for the raytracer was a simple parallelization of the main loop over each pixel in the image using OpenMP; all the variables required in the loop were moved inside the parallel for loop, so as to not cause any data dependency issues, and then a simple #pragma omp parallel for was placed at the top of the loop, and the two for loops, one for

the width of the screen, one for the height, were squashed into one a single for loop. Furthermore, schedule(dynamic) was used to squeeze out slightly better performance. This gave a good boost to the render time. The X11 library was required for OpenMP parallelization, to initalise multiple threads. Also we compiled with the –O3 flag which optimizes the compilation and gave a little extra boost.

The next extension we implemented was to add more lights to the scene, adding 2 more lights on the left and right walls, and in tandem with this, we implemented soft shadows, which was achieved by sending multiple slightly perturbed rays from each light source, and then summing and averaging the contributions. We settled upon shooting 25 rays for each light source, which gave nice looking shadows, without too much added computational cost.

A similar extension we then implemented was anti-aliasing, which was again a case of shooting multiple rays and averaging the results, except this time we shoot multiple rays from each pixel when computing the ClosestIntersection function, and averaging the results of this. This has the effect of softening some of the jagged edges in the image, by combining the colour found by each ray so as to have a blurring effect on each pixel, making it less rough.

The chief extension we implemented for the raytracer was adding a specular component to the colour computation. We firstly implemented simple Phong reflection, which computes the amount of reflection via the equation R = lightDir - 2 * dot(lightDir, normal) * normal, and then computes the specular component with spec_color += pow(max(dot(viewDir, R), 0), 50)) * B, where B is the light intensity. This Phong reflection gave some shininess to surfaces, but didn't look very pleasing compared to other BRDFs we saw online. So we then went on to compute the Cook-Torrance BRDF, which gives a nicer shiny look to objects in the scene. The Cook-Torrance BRDF computes the specular component via the general BRDF equation spec_color += (F * D * G) / (Pi * dot(normal,lightDir) * dot(normal,viewDir)), where F is the Fresnel equations for the ratios of reflected/refracted light, and then D and G are the associated specific equations for the microfacet distribution and the geometric attenuation of Cook-Torrance respectively. The result of using this Cook-Torrance BRDF were some nice, smooth colour transitions and bleeding between colours, as can be seen in the figures; figure 2 in particular shows the colour bleed on the now white

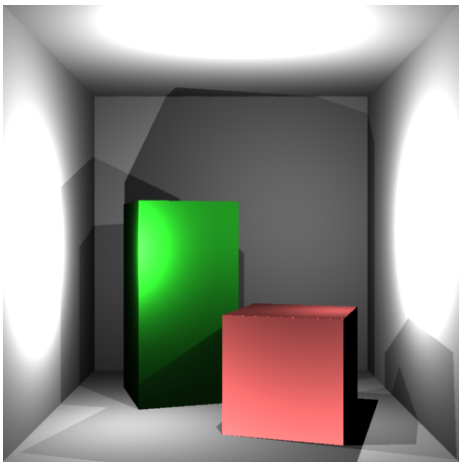

Figure 1 (left) – white lights

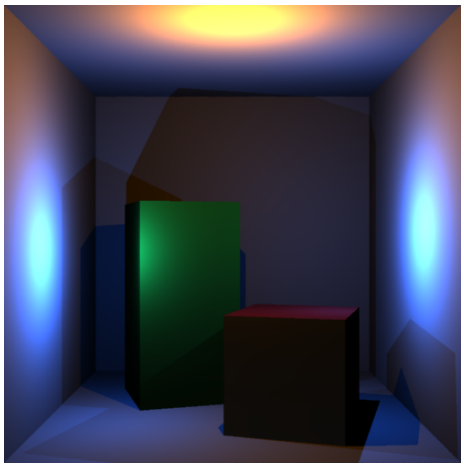

Figure 2 (right) – blue and orange lights

walls from the scene’s lights.