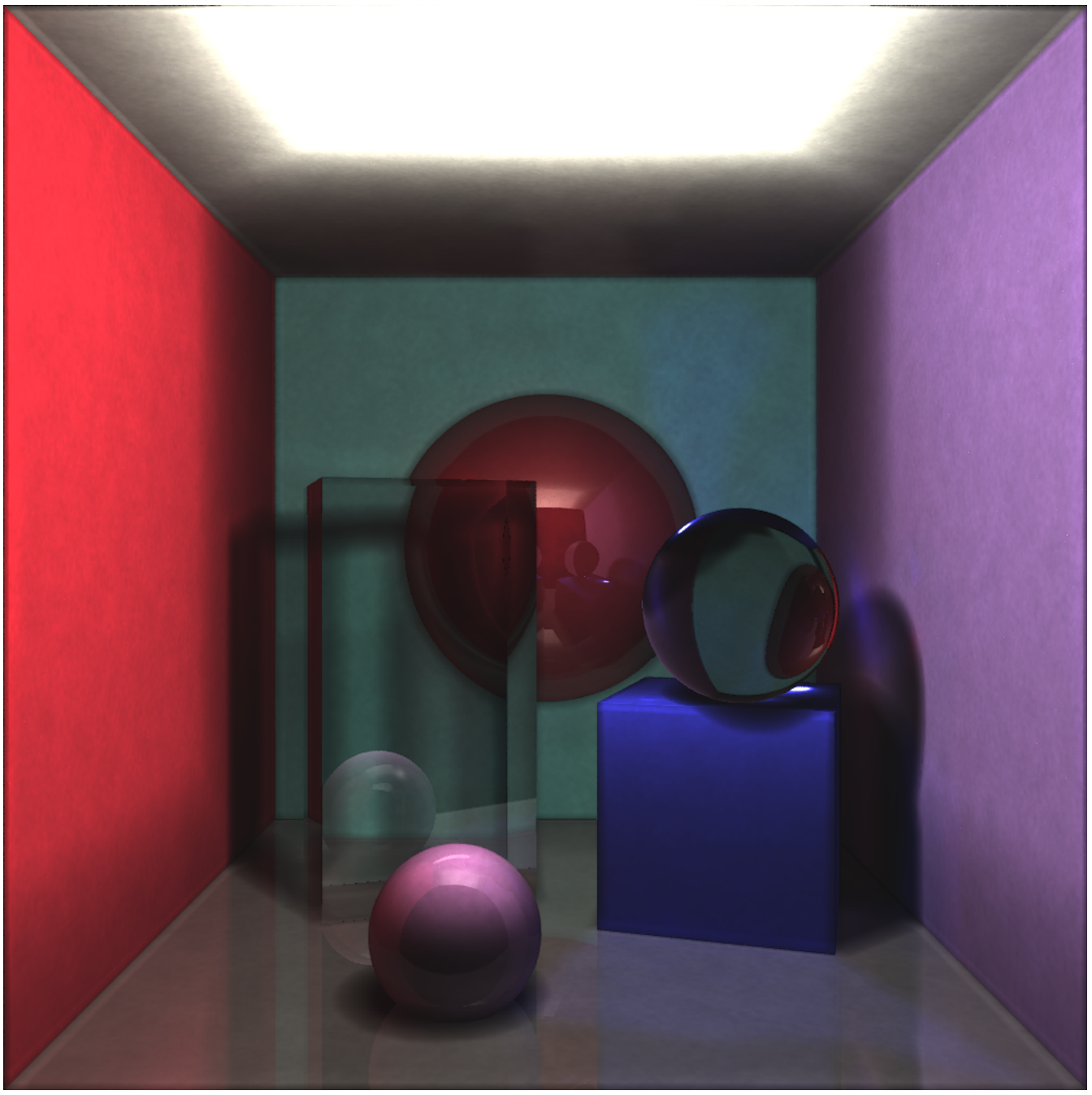
Raytracer Report - CP15571 & JT15435



Introduction

For the raytracer we wanted to dedicate our time to producing a spectacular image and optimise in areas where necessary, rather than focusing on just optimising. We spent most of our time implementing global illumination and optimising its process.

As specified by the unit director, this report aims to simply present the features we have added to our raytracer rather than explaining their theory and implementation. Below we have list all of the additional features we have added to our raytracer past the basic implementation.

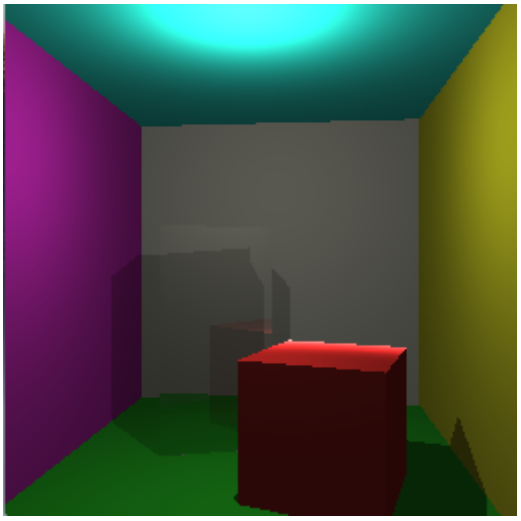
Raytracer Features

Specular Highlights

We used the Blinn shader to give specular highlights on certain materials. We originally implemented the Phong shader, however we wanted to be more efficient and not have to calculate the dot product for each intersection, so we decided to go with the Blinn-Phong model.

Reflections, Refractions and Fresnel

Adding realistic and correct reflections and refractions to the project took a long time as we had to perfect this both for the photon mapping case, as well as for general raytracing. We correctly combined our reflections and refractions to find a fresnel ratio meaning objects could be partially reflective and transmissive at the same time. We tested this heavily by checking how the light interacted with glass spheres until it matched the theoretical description of light reflection, refraction and fresnel. Below, an early picture of the fresnel effect is displayed for a transmissive block.



Spheres

Spheres were generated by calculating whether a ray intersects with the surface of a sphere in a similar way to the triangles. Our initial idea was to follow the icosphere approach, generating lots of triangles and using the same intersection formula. However this was extraordinarily expensive since the number of triangles grew by a huge number with each recursion step. On top of this, the sphere was never perfect and always had visible edges, following the mathematical approach gave us perfect spheres with little computational cost.

Parallelisation and Compiler Optimisations

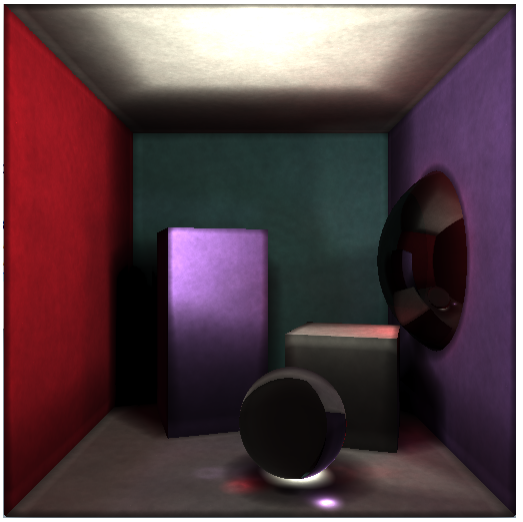
The raytracer was trivially parallelizable as it loops through each pixel one at a time to determine its colour, where no two pixels depend upon one another. Post processing was the same (anti-aliasing) to parallelize. We also parallelised, our photon tracing but had to ensure no race conditions appeared when adding the photons to the KDTree. As a further note, we found that a dynamic schedule worked best for our work load distribution since different iterations of the ray tracing loop would take a differing amount of time.

Well Structured Object Oriented Code Base

Although this may not affect the outputted image of the raytracer, a clean code base with correct C++ programming meant we could easily optimise our code in various ways. We correctly implemented polymorphism and inheritance for a general shape class making it easy to distinguish cases between spheres and triangles anywhere in the project. Furthermore, all the member variables were made private and accessible via mutator functions. Abstraction and Encapsulation was achieved, ensuring code was only present in relevant modules so that the general raytracing module is readable without having to access the individual modules. We modularised the code heavily so that the corresponding functionality was placed together, in fact our code base grew to have 16 modules due to the number of extensions we added! We believe our codebase should be able to be picked up by someone familiar with the concepts implemented and relatively easily be modified, we spent a long time trying to achieve this.

Light Sphere (Area Light for soft shadows)

In order to get more accurate shadows with our photon map we implemented an area light, but not as a square that you see in most raytracers. We wanted to avoid the unrealistic look the square glowing hole gave at the top of the room, therefore we simulated a spherical area light. This involved stochastically sampling positions in a sphere using rejection sampling for multiple point lights. We then simulated an equal number of photons from every single point light in the light sphere. This improved the accuracy of our light and shadows. The image to the right shows the appearance of soft shadows due to the light sphere.



Efficient (Edge Detection) Anti-Aliasing

The most common anti-aliasing scheme for the raytracer is to scale the image up and render it, then average each pixel value. However this technique can be extremely slow across the entire image, also anti-aliasing in a lot of places in the cornell box is in fact pointless. We instead performed anti-aliasing only on the edges of objects in the cornell box. This was our process:

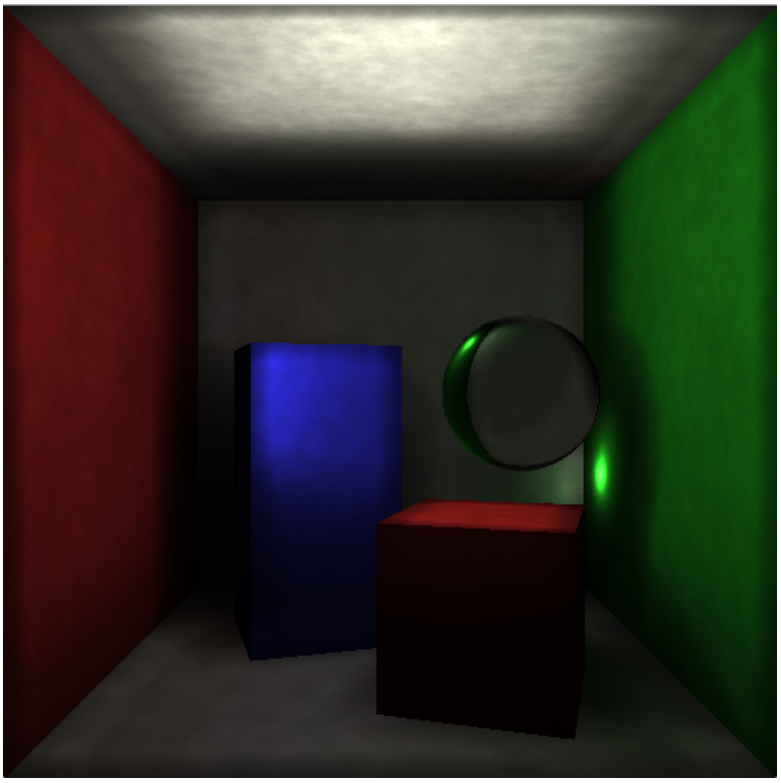
1. Find the gradient of the image by using the sobel operator to convolve the image
2. Threshold the gradient image to find all of the pixel locations of edges
3. Supersample each edge pixel with 12 stochastically chosen super samples per each pixel

4. Average the values of the super samples for each pixel to find the pixel value

Photon Mapping Global Illumination and the extensions that came with it

As mentioned in the introduction we spent most of our time implementing and improving our global illumination using the Photon Mapping technique. We have broken down the individual things we did for this process below:

- For generating and tracing photons we sampled from a light sphere. We used the Russian roulette approach to trace the photons until they were all fully ‘absorbed’ and stored all of their intersection positions. We modified the algorithm so that it correctly interacted with our glossy surfaces i.e. less photons stored on glossy surfaces. As well as adding more photons to the caustic map, so that caustics coming from transmissive spheres were more accurate and visible. The picture below clearly presents the effect of caustics from the photon map.

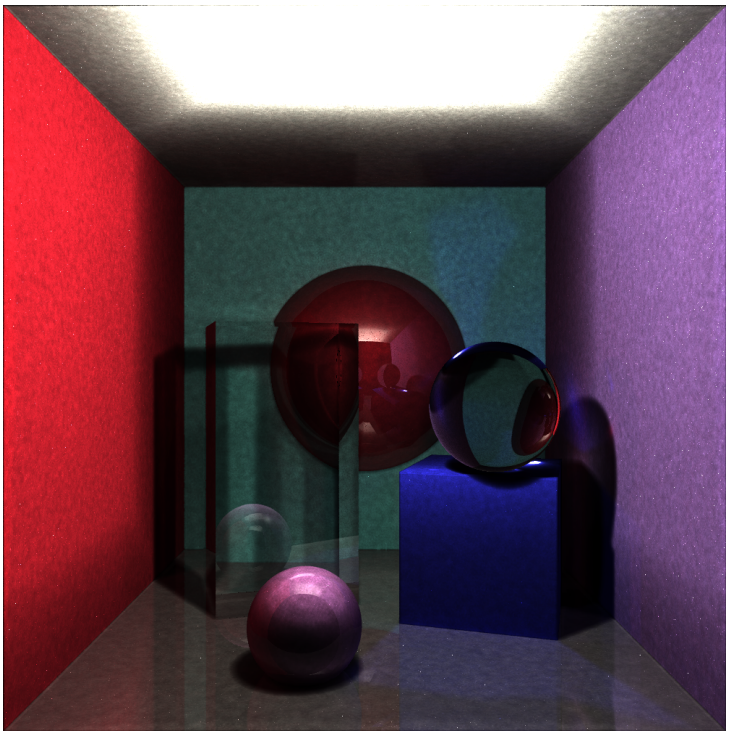


- Next we built a KDTree which held all of the photons intersection positions. This involved building a large recursive data structure which we optimised to work cleverly with pointer in order to minimise the amount of memory the KDTree required (this was a must as we shot up to 25 million photons during development). The KDTree also reduced the search time for a closest photon from O(n) to O(log n), therefore our nearest neighbour search was extremely efficient, even for millions of photons.

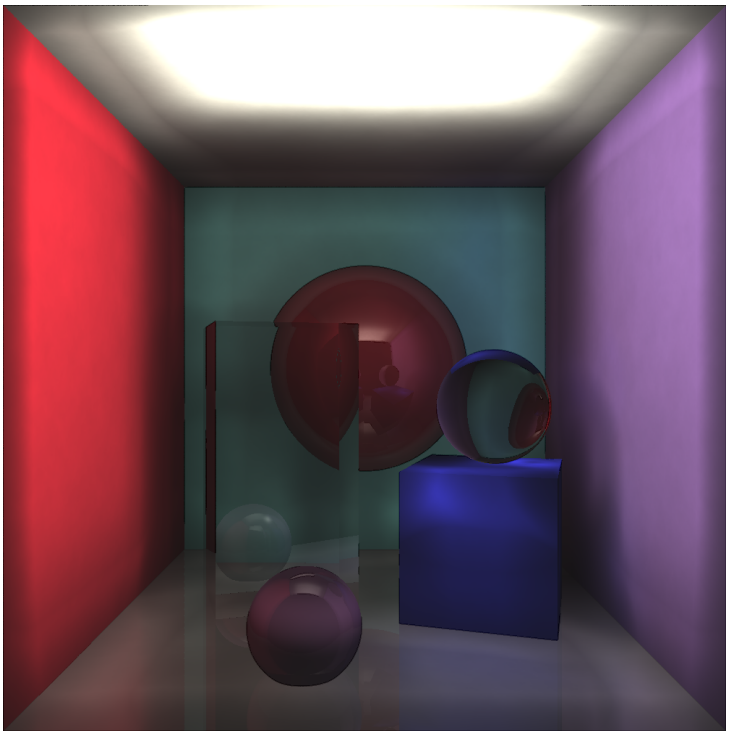
- Next the radiance estimate. This involved getting the estimated colour for a given pixel, therefore it had multiple cases for diffuse, specular, reflective and refractive surface. Where reflective and refractive surfaces had to correctly call call the diffuse and specular surface estimates. For a diffuse/specular surface estimate we had to gather the nearest n photon

intersections and use their power and colour to calculate the diffuse surface radiance for a pixel. We optimised this step by introducing a max search distance from a given point, effectively pruning the KDTree for positions too far away to consider. We also used a Gaussian Filter to improve the accuracy of our radiance estimate in the gathering stage, this massively reduced the effect of false colour bleeding.

We also spent a large amount of time trying to perfect our final render of the photon map. In order to get a very realistic image, we had to increase the number of photons and increase the gathering stage too. We found that these variables had to be at a specific ratio to one another to get the desired effect. So, our final render at the top of the report consisted of tracing 15 million photons and searching for the closest 2500 neighbours at a given point to calculate each pixel value. To give you a bit a more context, we have added two images below, both with the same image quality but with different parameters. As you can see our final render hits the sweet spot in the middle with high detail and low noise. The more photons you gather the lower the noise in the image, however this essentially blurs the image and leads to a large amount of false colour bleeding. However, if you have a large amount of photons and do not gather enough the room becomes more noisy.



25 Million Photons 1000 Gathered



1 Million Photons 10 000 Gathered

Finally we would like to point out that the devices we were rendering on could not render more than 25 million photons as this filled up our memory during rendering, causing the program to crash. Also note that as we increase the number of photons gathered the render time increase by a large amount, in fact the final render at the top of the report took 16 hours with 15 million photons and 2500 gathered! If we had more time and more powerful computers we could have rendered a higher quality image.