Raytracer Overview

By Jaime 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 ray tracer look awesome! See `README` for setup instructions.

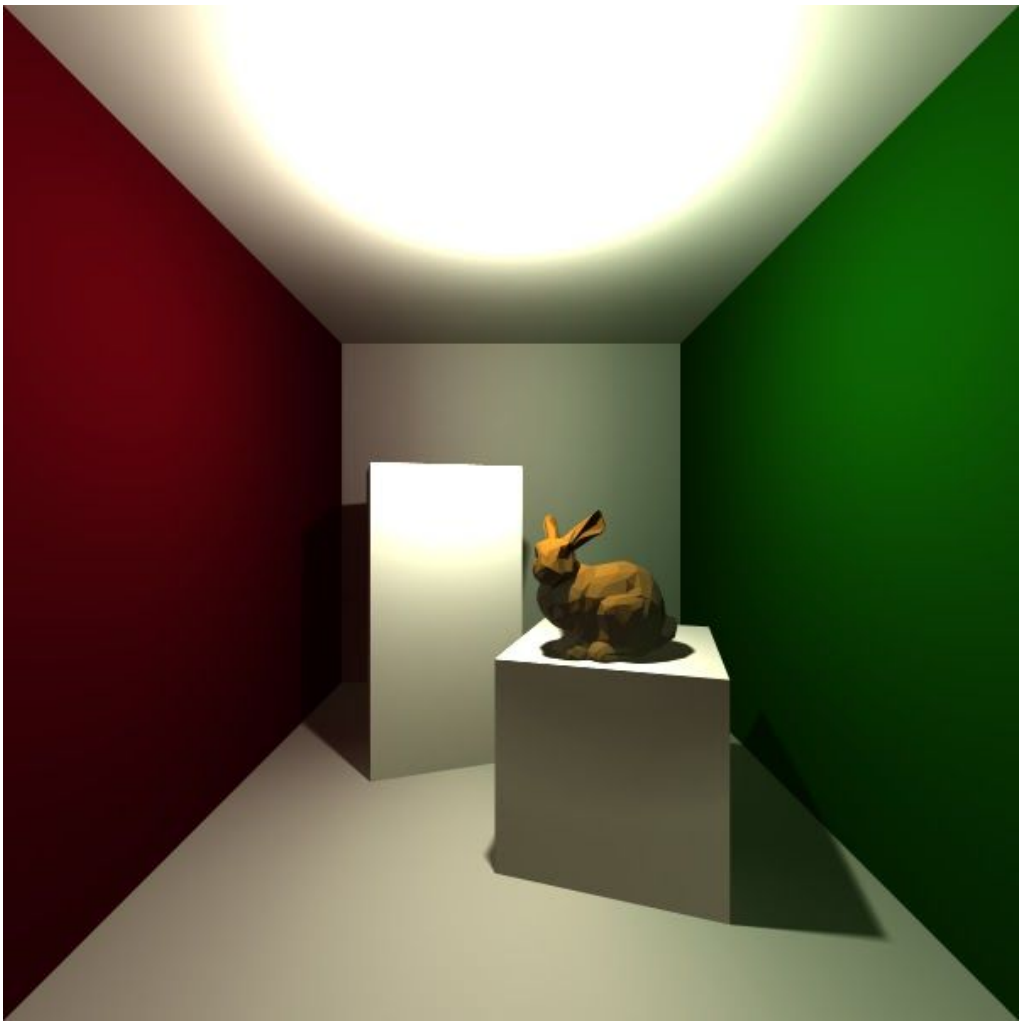

Fig. 1. Our final render.

Contents of Extensions

Small Improvements

In the raytracer we started off by optimising the closest intersection code to use Cramer's rule, providing speed improvements of around 17%. We also improved the quality of our shadows by making them subtract from the colour of the underlying pixel instead of being solid black. This strategy was later discarded after we implemented photon mapping.

Anti-Aliasing - SSAA

In order to remove so-called "jaggies" from our render we needed to use anti-aliasing, a technique that smooths out jagged edges in scenes. As ray-tracers are typically very computationally heavy programs, we opted to use a very expensive, but very effective form of anti-aliasing called SSAA (Super-Sampling Anti-Aliasing). The idea is to render the image at a much higher resolution and then downscale it back to the desired size, averaging out pixels as we go.

The result is very good even with relatively small sampling. Notably, the small out of place pixels on the edges of the enclosing box completely disappear, which removes a big distraction in the image. Though the lighting in the two images changed between the following screenshots, we can still see the effect of SSAA in the highlighted regions of the screenshots (top (fig.2) is no AA, bottom (fig.3) is 4x scaling on each edge, i.e. 16 samples per pixel). Note these screenshots were taken before the 50% mark was reached.

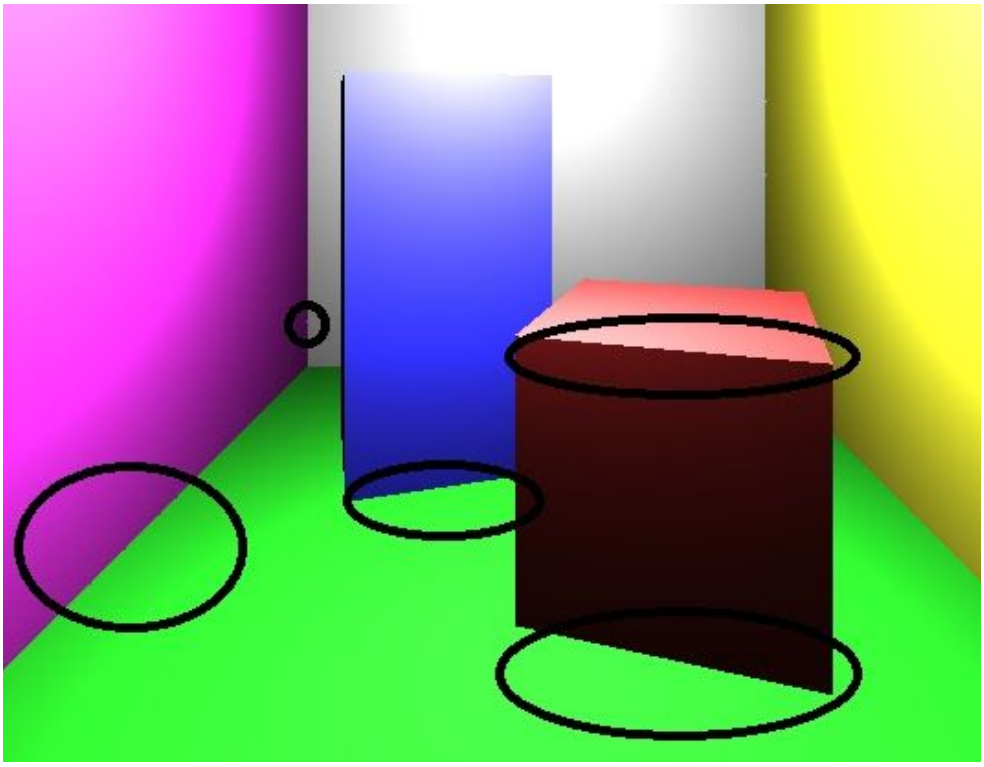

Fig. 2. Without SSAA.

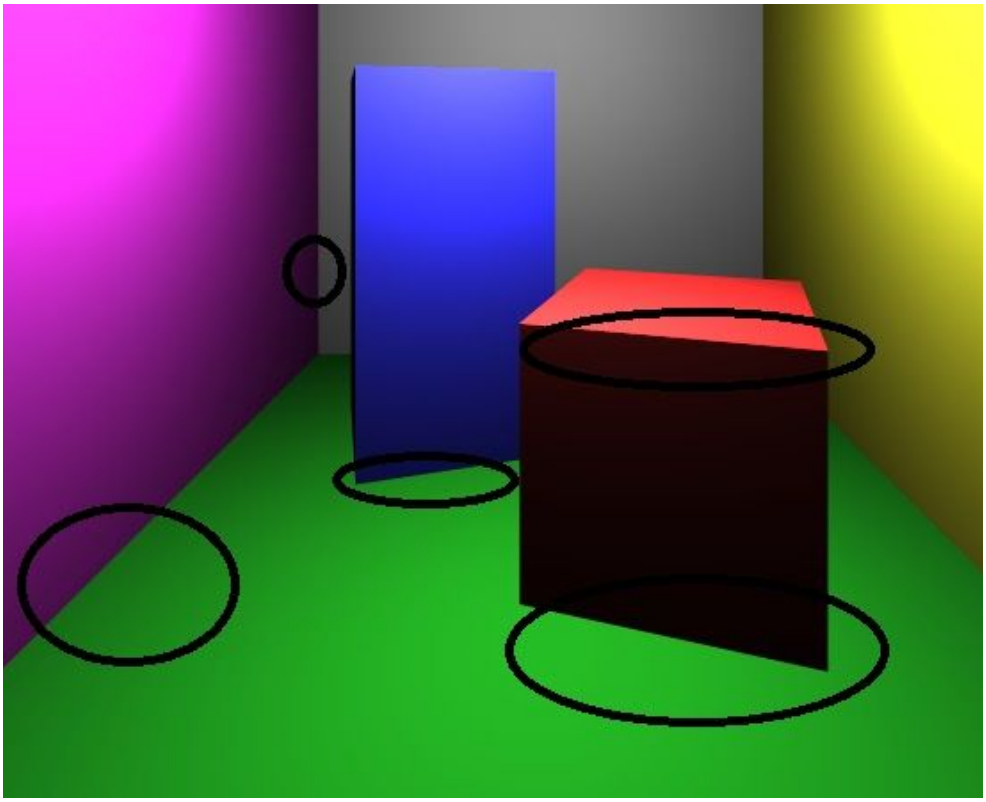

Fig. 3. With SSAA.

Soft-Shadows

Before soft-shadows were implemented, our shadows were very sharp, making them look unnatural for the level of light in the scene. They were smooth from anti-aliasing, but didn't have the nice feathering effect that exists for real shadows.

In order to achieve this we take our point-light and randomly sample points around it within a certain radius. Then each ray cast will check if it is in shadow to any of those points, the average is used to determine whether it is in shadow and by how much. Since we compute the light samples only once at the beginning, we have to do less work during each ray, but we require a higher number of shadow samples to reduce banding. The two images below show the difference before and after soft-shadows were implemented (the colours were changed to make the scene look prettier). The final render contains 400 soft-shadow light samples.

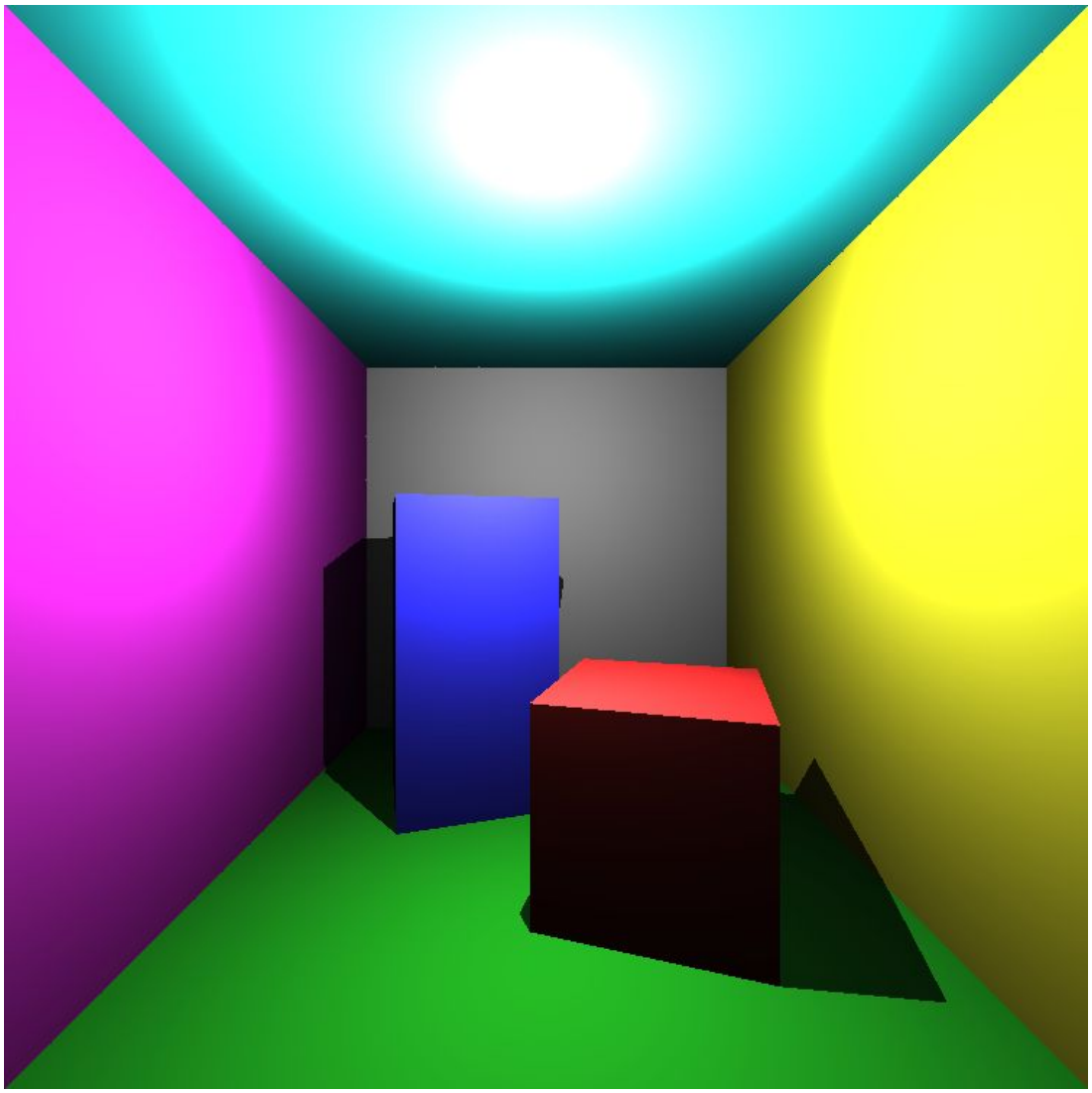

Fig. 4. Without soft shadows.

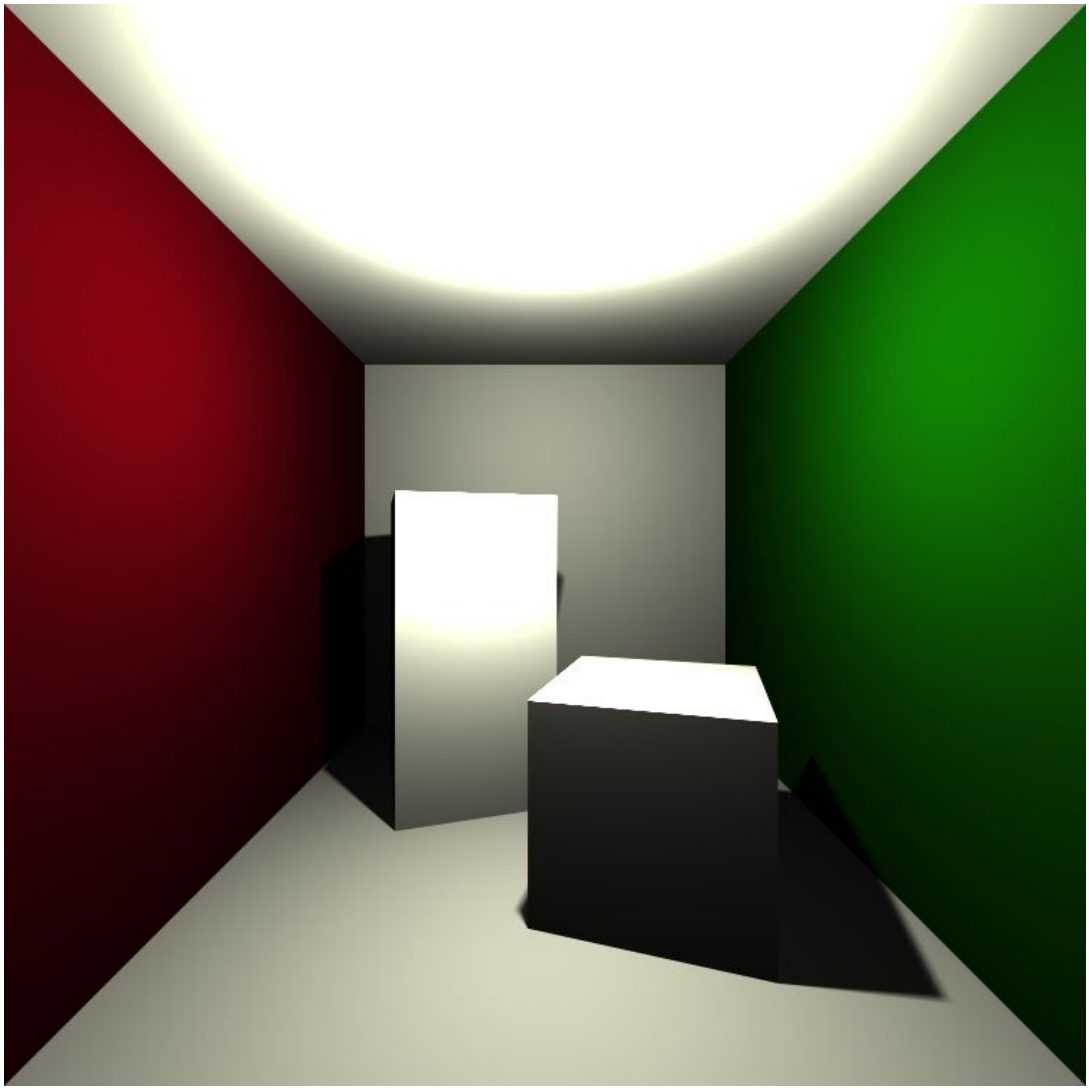

Fig. 5. With soft shadows.

Parallelisation - CPU

The raytracer is a very parallelisable program, however it suffers from one drawback: recursion. Due to the recursion present in the lighting calculations (and, though we haven't implemented them, reflections) with intersections and later the photon mapping, the raytracer does not lend itself well to the GPU. Instead of adapting the system to work with the GPU we instead opted to utilise the full potential of our CPUs.

The final render for this report was ran on an intel i7 4790k which has 8 cores. This CPU supports the intel SSE4.2 vectorised instruction set which is turned on in our makefile. This makes it possible for the compiler to, providing data is aligned well, perform several optimisations that result in multiple computations at once on the same core. However, more importantly, we make use of all 8 cores to split up the image and process it up to 8x faster.

There are several ways of going about this; either we could use the threading libraries in C++ to manually allocate tasks to each thread, we could use OpenMP to allow the compiler to sort out all our threading for us, or use MPI to break up the data more explicitly. The simplest, but not the most scalable is OpenMP. This makes it very easy to parallelise the very outer for loop in the drawing process, so that each thread takes a chunk of the image to render independently. In order to best distribute the load, we make use of OpenMP's dynamic thread scheduling; instead of giving each core a fixed chunk of the image, they are provided a smaller chunk and given a new one when they are finished. This means that slower cores don't limit the render time as much as they might have done.

Though not particularly computationally expensive, we also parallelise the soft-shadow sample generation. Note we can't parallelise the generation of the photon map (even though it would speed things up quite a bit) because there is no thread-safe reduction for C++ STL Vectors.

Model Loading

We added in a model loader for a simplified subset of the .obj file format. This allowed us to add in a Stanford Bunny into the scene and any other models we might have liked. The rest of the cornell box was moved out into .obj files.

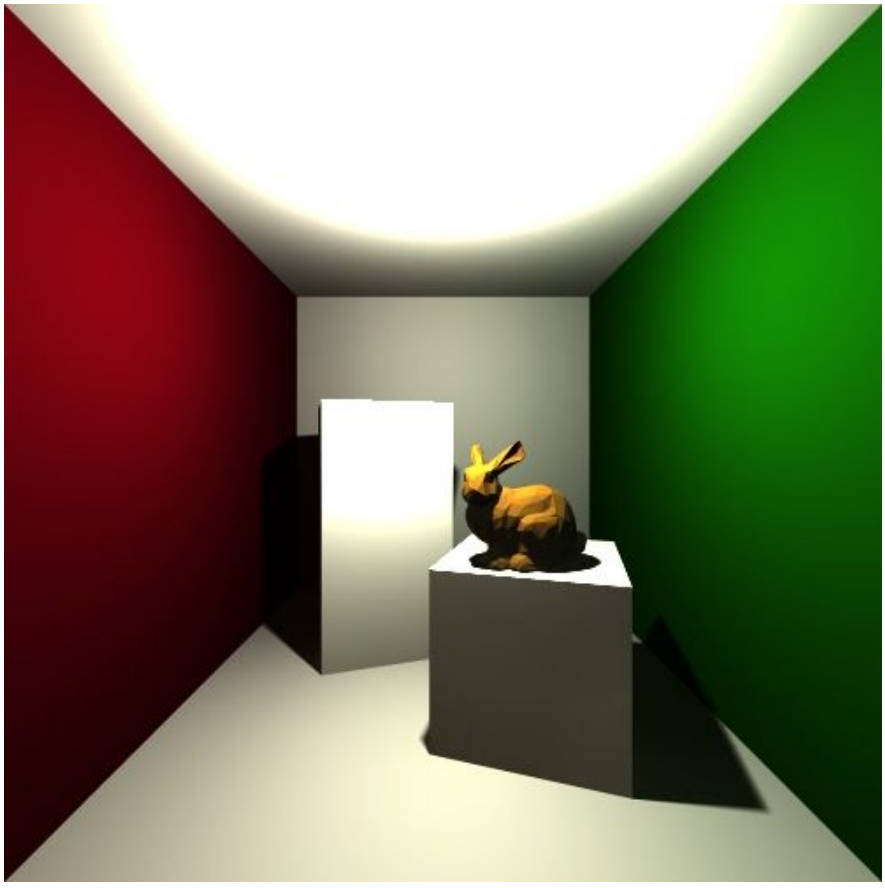

Fig. 6. Loading the bunny from an .obj file.

Photon Mapping

In order to make our way towards global illumination we first need to perform photon mapping, where we cast out many photons out from a light source and bounce them around the scene, recording all the places they bounce. This allows us to approximate where light can access in a scene.

Once before the rendering of the scene starts, we generate a photon map, or a list of all collisions the photons made. We allow photons to bounce a maximum of 5 times unless they leave the scene. We also discard all the first bounces of the photons, as we will not need to use them; direct lighting calculations from the raytracer are better quality than those generated by the direct photon map. The following screenshot shows the quality of the direct lighting that would have been generated from the photon map alone.

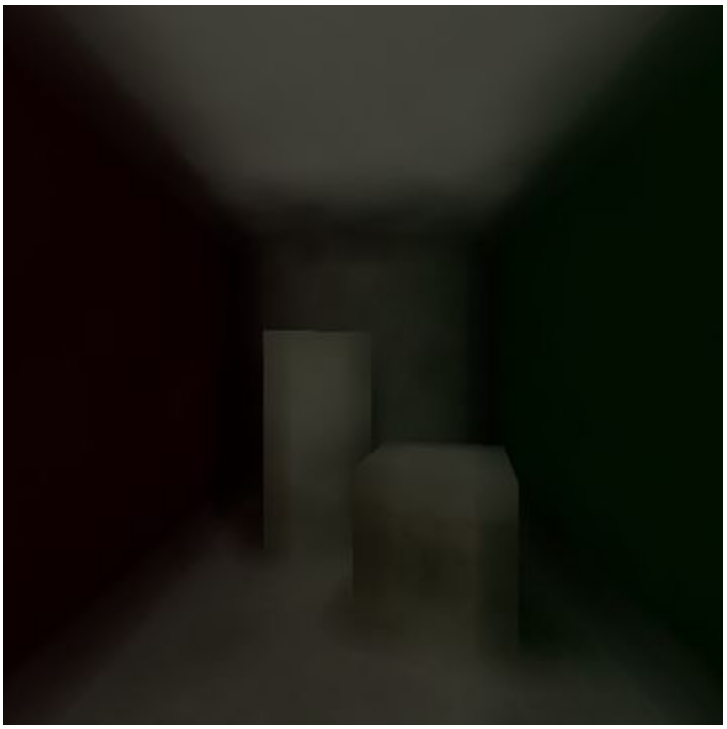

Fig. 7. Direct lighting generated just from the photon map.

Approximating Global Illumination (Indirect Illumination and Colour Bleeding

In order to approximate global illumination at a pixel we will sample nearby photon bounces for the ray's intersection point. There are two methods of doing this; K nearest neighbour and a radius search. We chose to use a radius search, since, if there are insufficient photons nearby to a point in space, there will be a lot of colour blending in from relatively unrelated pixels using knn. Using radius search was more straightforward to implement and produced better results. Namely the time complexity of a naive knn was O(n^2) in photons and a range search is O(n).

The result of adding the colour of the nearby photons to the direct illumination was very good, the shadows no longer had to be approximated by darkening the colour and, since photons change colour when they bounce on a coloured surface, we get the added benefit of colour bleeding, which is especially evident in the shadows. Here is an example of colour bleeding and shadows at this point;

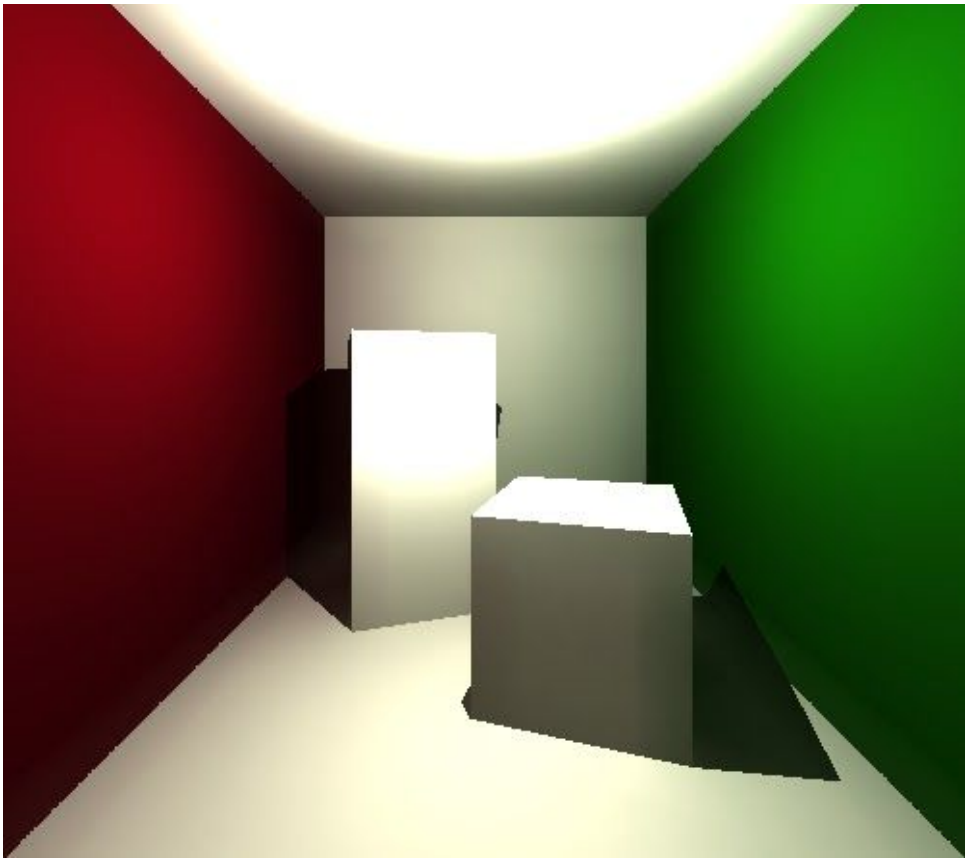

Fig. 8. Colour bleeding, notice the green in the shadow from the cube and single colour shadows are present

Range Trees and KD-Trees

In order to improve on knn search, we can make use of a KD-Tree, which simplifies a knn search to O(log(n)), however it only simplifies a range search to O(k*n^(1-1/k)), which isn't much of an improvement over linear search, given the additional memory overhead from a tree.

A more attractive option is a range tree, which can guarantee O(log(n)^k) time for a radius search. However, after integrating an implementation of a range tree I found online, we found that was not only much slower than the linear search, but consumed a colossal amount of memory (200MB became 7GB). It's likely this was not a very efficient implementation, and would have benefited from specialisation to three dimensions. However we were unable to implement a range tree ourselves, so we left it out of the raytracer.

Photon Mapping Optimisations

In order to improve the runtime of the linear search we moved as much of the computation to the generation stage. This means that the photons store the length of their normal vectors. In addition, we were able to remove expensive square roots by comparing to the search radius squared. This provided small speedups for low numbers of photons, but this likely made a big impact during the final render.

Improving Photon Mapping Quality

The previous image in the report shows unfortunate blending in the shadows on the floor that leaves them almost a single colour. We can also see banding on the edges of the walls. These are both undesirable.

Our first attempt to fix this was by ensuring that photons can only be gathered if they are on the triangle that we intersected with. This immediately solves the issue of banding and creates a distinct colour separation in the shadows on the floor.

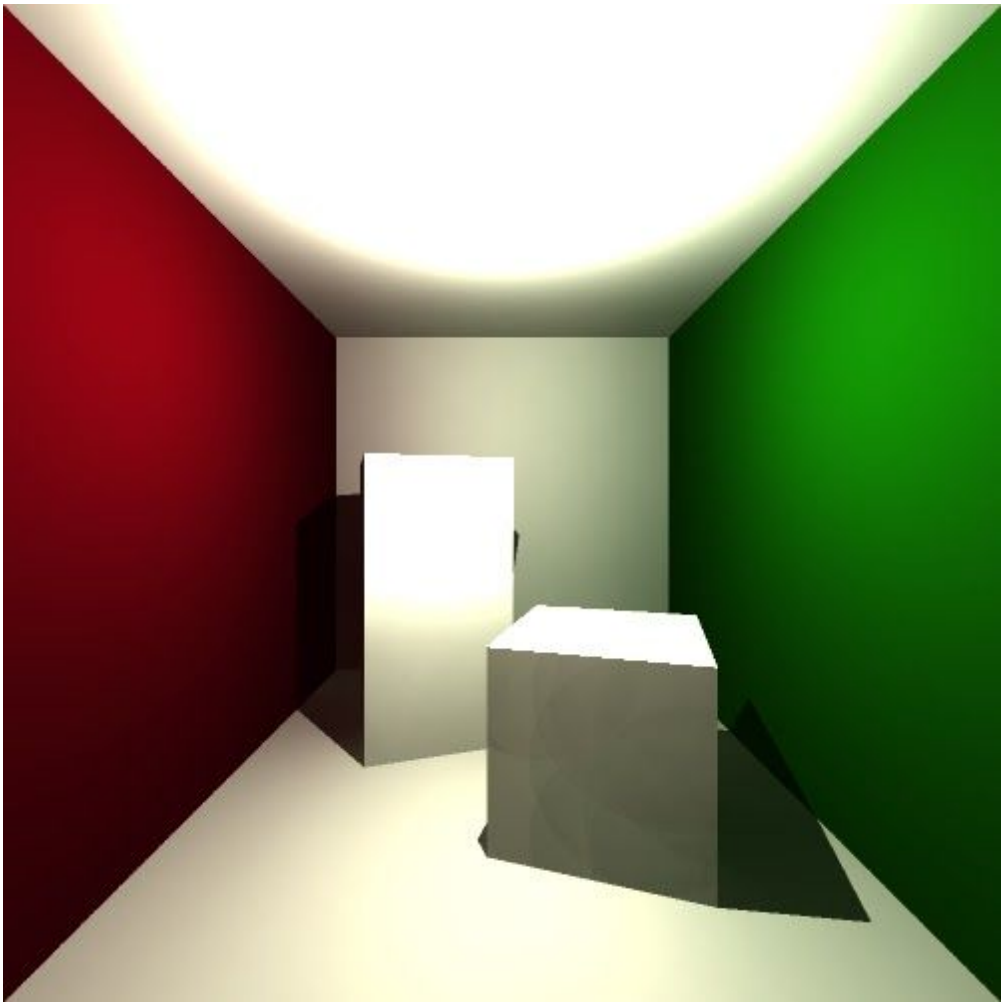

Fig. 9. Solving banding and shadow colour separation.

However, when we added the bunny, things got worse again. There were too few photons landing on the bunny's underside to produce anything other than near black shadows. What we actually wanted was the effect we were trying to avoid in the first place! To combine the best of both worlds, if two triangles are at an angle to each other and not a right angle (the actual threshold on this is any angle such that the cosine of that angle is greater than 0.25) then the photons can be gathered from both surfaces, else they are local to the triangle. This produced a much nicer effect which can be seen here in the final render (8x8 SSAA, 400 Soft Shadow Samples, 750000 photons);

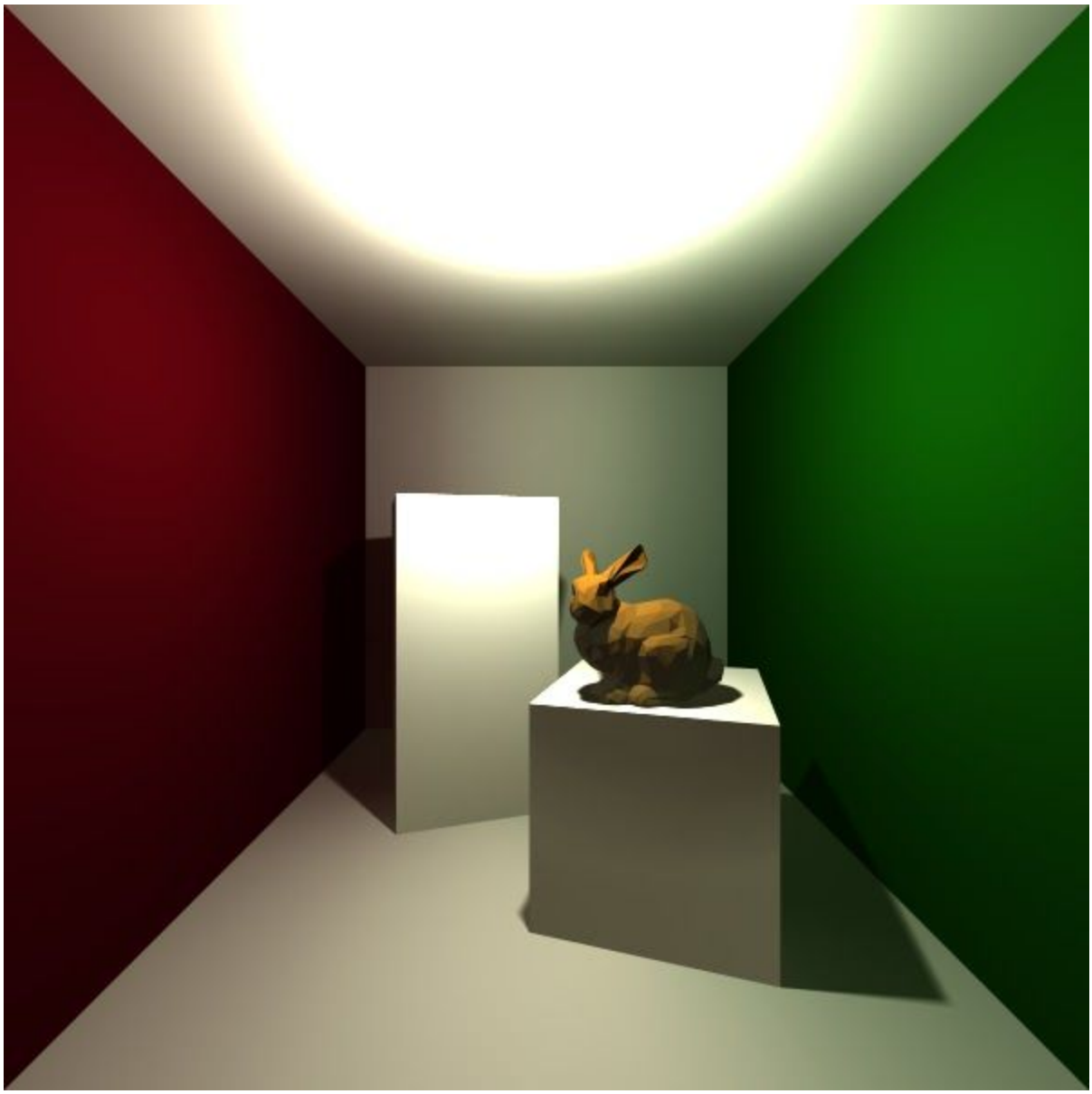

Fig. 10. Our final render with 8x8 SSAA, 400 soft shadow samples and 750k photons. Bunny's underside has blended shadows