# COMS30115 - Computer Graphics

Gavin Parker(gp14958), Samuel Hicks(sh14820)

May 1, 2017

https://github.com/sammhicks/computer-graphics

## 1 Introduction

Over the course of 10 weeks we produced 2 functional renderers - a software Raytracer and Rasterizer. Our Raytracer supports multiple lighting methods including flat lighting, Phong/Diffuse lighting and Global Illumination. Our Rasterizer supports fewer lighting techniques but has support for realtime shadows using a shadow map. Both renderers also support texture mapping.

## 2 Raytracer

### 2.1 Basic Rendering

Our initial implementation of the raytracer used a flat lighting model, rendering each surface with a single colour.

### 2.2 Texture Mapping

The next feature we added was texture mapping so that surface colours were determined by an image rather than a single colour. Our implementation worked by taking the UV coordinate of the collision position of the incoming camera ray, and using it to index a stored image. The image was loaded from a file using a png loading library called 'LodePNG'. We only used the texture mapping for diffuse color mapping, though the same technique could be used for normal/bump mapping.

### 2.3 Phong/Diffuse shading

To improve the quality of the rendered scene we introduced higher quality lighting using the Phong model. We started by implementing diffuse lighting which involved calculating the brightness of a surface from the distance from and intensity of a light source. We were also easily able to add shadows by finding intersections between each surface point being rendered and the light source. The Phong model improves upon this by taking the view direction into account to calculate a specularity value. This value is applied to the surface to add white reflections, making metallic surfaces look more realistic.

### 2.4 Anti-Aliasing and Soft Shadows

Two more improvements to visual quality we made were the introduction of soft shadows and Anti-Aliasing. Anti-aliasing was implemented by simply calculating multiple rays per pixel and taking an average. This meant that colors on edges were rendered as an average of the two surfaces, making the edge look softer. We also introduced soft lighting which worked similarly by taking multiple rays from a surface position to a light source and taking an average. Here the light source was given a radius so that the rays could be slightly different. This means that at shadow edges, some rays collide with an obstacle while some reach the light, so the shadow color is not completely dark.

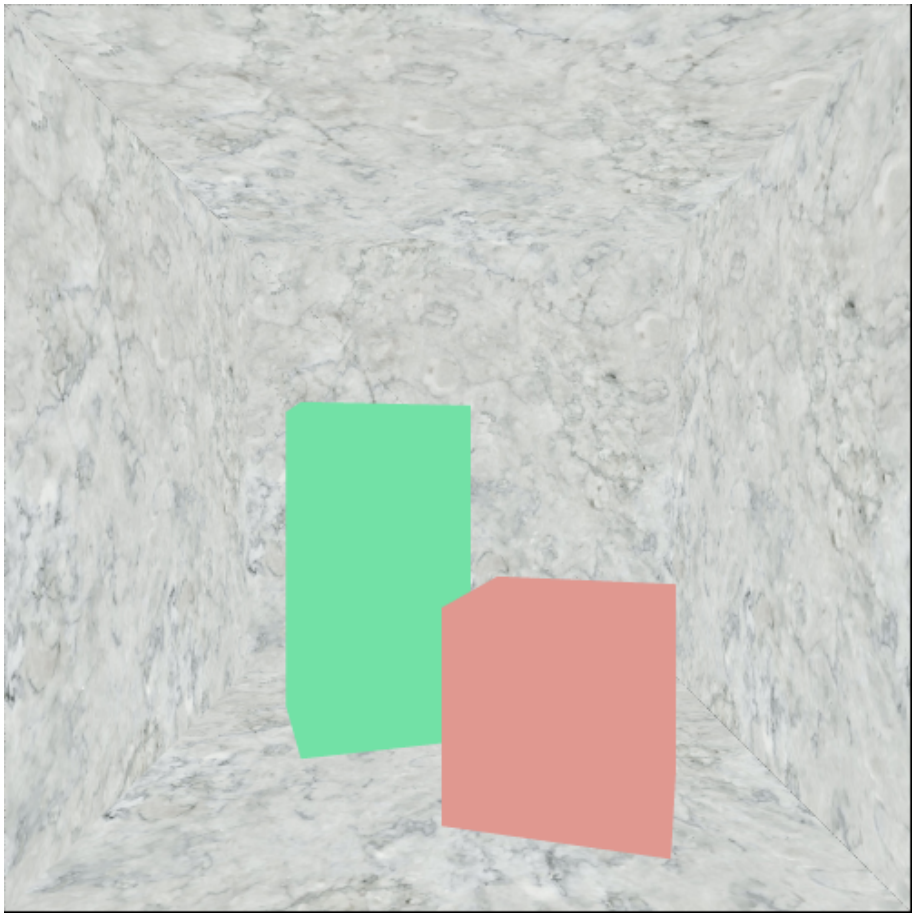

Figure 1: The cornell box scene rendered with textures and flat lighting.

### 2.5 Global Illumination

We implemented a type of global illumination called path tracing, which simulates the bouncing of light rays in the scene by tracing back from the camera ray. For each camera ray collision we calculate a random new ray and trace that further into the scene. This means that the renderer can accumulate indirect lighting from different surfaces in the scene, resulting in higher quality shadows as well as effects like colour bleeding. To produce a fast result we use a convergent approach where we only produce a single ray per bounce, and then take an average of the rendered frames.

### 2.6 Software Optimization

The Biggest issue with the Raytracer model, and especially path tracing, is that it is very slow. We have made various improvements to our renderer to reduce render times without sacrificing visual quality. The most obvious improvement was to make use of multi-threading a the raytracer is easily parallelisable. The raytracer is slow as many collisions need to be calculated for each pixel, for the camera ray as well as any light rays. This usually involves iterating through every piece of geometry in the scene, though some improvements were made, most notably the use of a 'Bounding Volume Hierarchy'. This technique involves calculating bounding boxes around sections of geometry, so that we can discard them very quickly when tracing rays. We can also finish tracing early when calculating shadow rays as we are only interested in the possibility of a collision, rather than which collision is closest. We also make use of Cramer's Rule to calculate intersections using fewer expensive operations.

### 2.7 Hardware Acceleration

Even with all the software optimizations we introduced, the raytracer was slow - far from the real time rendering possible with the Rasterizer. We were able to vastly improve the speed of our

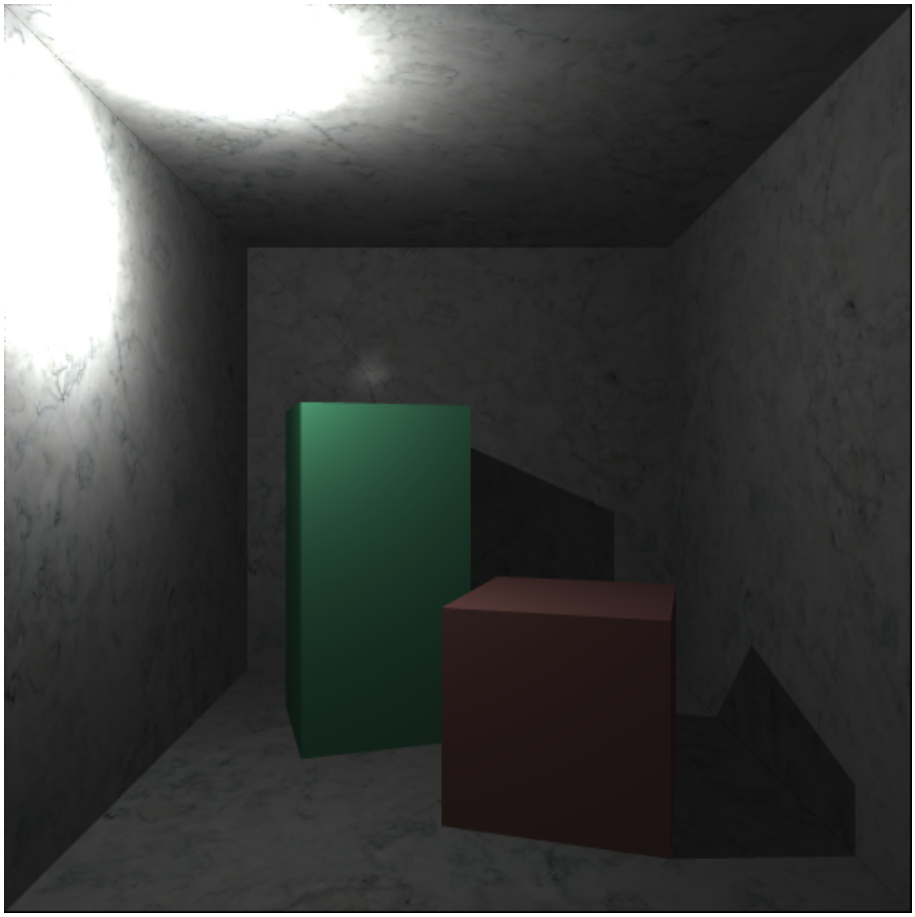

Figure 2: The cornell box scene rendered with textures and Phong lighting.

raytracer by implementing it in OpenCL, making full use of the parallelism available on a modern GPU. This was difficult as GPU architecture is not particularly suited to ray tracing due to limited local memory, no recursion and the cost of branching. Our implementation uses very few branches to trace rays and uses a similar convergent technique so as not to run out of local memory. This implementation resulted in much faster rendering times when using global illumination.

# 3 Rasteriser

## 3.1 Basic Rasteriser

The basic raytracer would render the cornell box using perspective projection with flat colours and lighting. We used a depth buffer to ensure that the boxes were correctly rendered in front of the back wall.

## 3.2 Texture Mapping

We also implemented texture mapping in the rasterizer, making use of the same code to load in PNG image files. The difference here being that the texture coordinate is calculated using barycentric coordinates so that the images do not appear warped on the surface.

## 3.3 Phong/Diffuse shading

We also implemented phong/diffuse shading as a pixel shader in our rasterizer. This is slower than a vertex shader, which would interpolate the light values, but produces much better images. We also used a shadow map for our point light. The shadow map works by projecting each camera space pixel into the camera space of the light, imagining the light as having 6 view directions. We

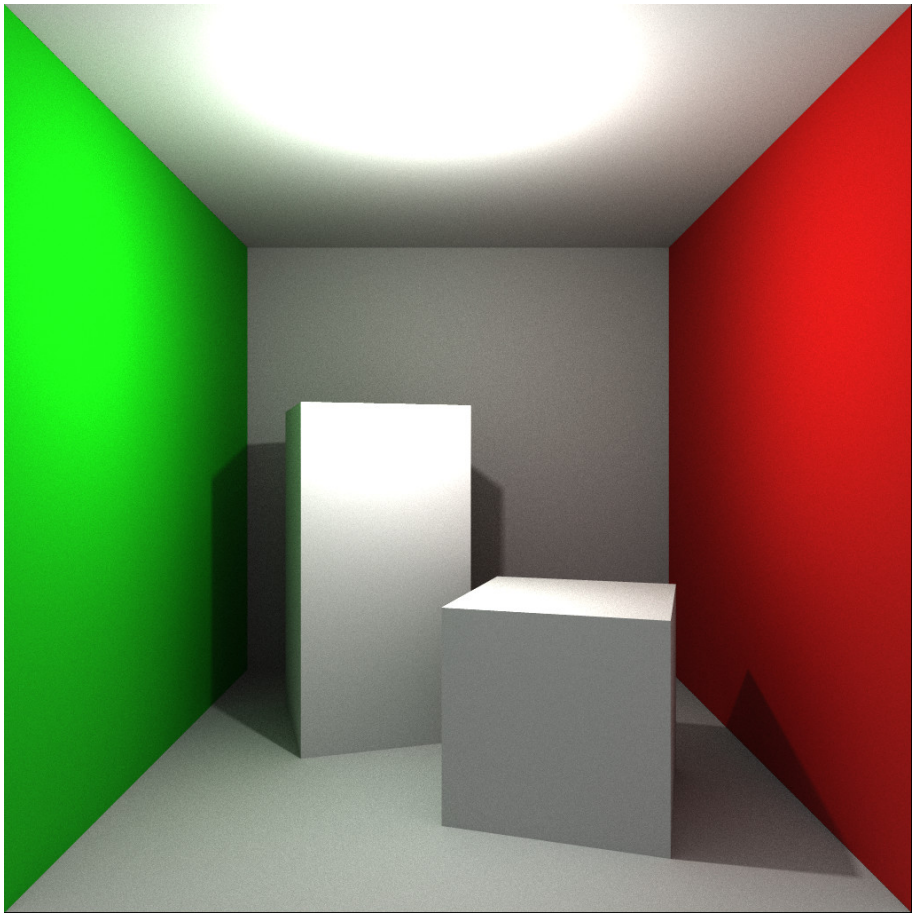

Figure 3: The cornell box scene rendered with path tracing, making use of soft shadows and AA.

can then use a depth buffer for each view to determine if something is between the light and a given point. This lets us add shadows to our scene with minimal performance loss. However our implementation still produces artefacts and is not as visually impressive as raycasting for shadows or using shadow volumes.

### 3.4 Basic clipping

We have a limited implementation of clipping that removes triangles that are completely outside the view frutsum. This prevents errors in some cases, but still results in artefacts as the camera approaches the scene.

## 4 Miscellaneous

### 4.1 Wavefront obj/mtl parser

All our rendering modes support different scenes, loaded in as 'sobj' files. This is an intermediate format produced by our prolog based .obj file parser. The parser also allows for the conversion of .mtl files such that our renderer can use materials extracted from the object file.

### 4.2 Automatic scaling adjustments

To support our general object file loading we adjust our camera and light parameters automatically such that the scene is always fully contained within the camera view and lit as to be perfectly visible.