Advanced Computer Graphics Report

dh651

December 2019

1 Introduction

This report summarizes the work that has been done for the Advanced Computer Graphics Stage 2 submission. Section 2 briefly introduces the concepts present throughout the report. Sections 3 and 4 cover the further work and therefore make up the remainder of the report.

2 Ray Tracing

The concept of ray tracing is the basis of all the work done throughout the rest of the report. It is a method of rendering based on shooting rays from the camera into the scene. The main alternative to ray tracing is rasterization, which is the process of converting vector based images into pixels, primitive by primitive.

Given an adequate definition of a scene, ray tracing can define some[1] visual phenomena quite easily. A scene with a light source L, a set of objects O, and a camera C can be rendered via the usage of ray tracing as shown with pseudocode in listing 1.

Listing 1: Ray tracing pseudocode

```
1 For each pixel in the image
2 {
3     Calculate a vector R from C to pixel
4     If R intersects an object from O
5     {
6         Compute light intensity I on O from L
7         Set pixel to I
8     }
9 }
```

Line 6 of listing 1 is where lighting properties of an object in O are calculated. Any lighting model can be apllied at this stage, the choice of model for this coursework is the Phong lighting model, l. It splits the lighting of an object into three components:

[1] Not all, as is discussed in section 4

1. The ambient component l_a, constant illumination irrespective of the direction and presence of the light L
2. The diffuse component l_d, illumination based on the random scattering of incoming light
3. The specular component l_s, illumination based on perfect reflection of the incoming light by the object

The three components added together create an illumination of the object. The equations for l are defined in the `calculate_colours` function in the `image_generator.cpp` file.

Another important lighting feature made simple using ray tracing is shadows. A ray is traced from the object that has been intersected to the light source. If there are any objects between the light and the initial object hit then the initial hit is considered to be in shadow and therefore not illuminated.

2.1 Phong shading

Alongside Phong lighting there is Phong shading. It has been implemented using two guides online[2][3]. It is a method of interpolating normals in order to produce smooth surfaces from polygonal surfaces. The `polymesh.cpp` file computes the vertex normals using the `compute_vertex_normals` function, which are then used in the intersection function in `triangle.cpp` to compute the interpolated normal. The normal interpolation uses Barycentric coordinates of the hit to compute the weights of each vertex normal.

2.2 Ray Traced Image

Figure 1 shows the scene rendered using only conventional ray tracing. Phong lighting illuminates the spheres and teapot, and shadows are cast on the box surfaces by the objects inside the box. The teapot has been shaded smoothly using methods described in section 2.1.

3 Reflection and Refraction

Reflection and refraction are implemented using the guide on the website scratchapixel.com[4]. Two functions are implemented to take care of that, `get_reflected_ray` and `get_refracted_ray`. Given an incoming ray and a hit on an object O they return a reflected/refracted ray.

[2] `https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/barycentric-coordinates`

[3] `http://blackpawn.com/texts/pointinpoly/default.html?fbclid=IwAR3rHgXfmHYjLTgWIBFUQ4lhmWAMN-tTrHJNn2VcggD2tInAC7DOzMKhSAg`

[4] `https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-shading/reflection-refraction-fresnel`

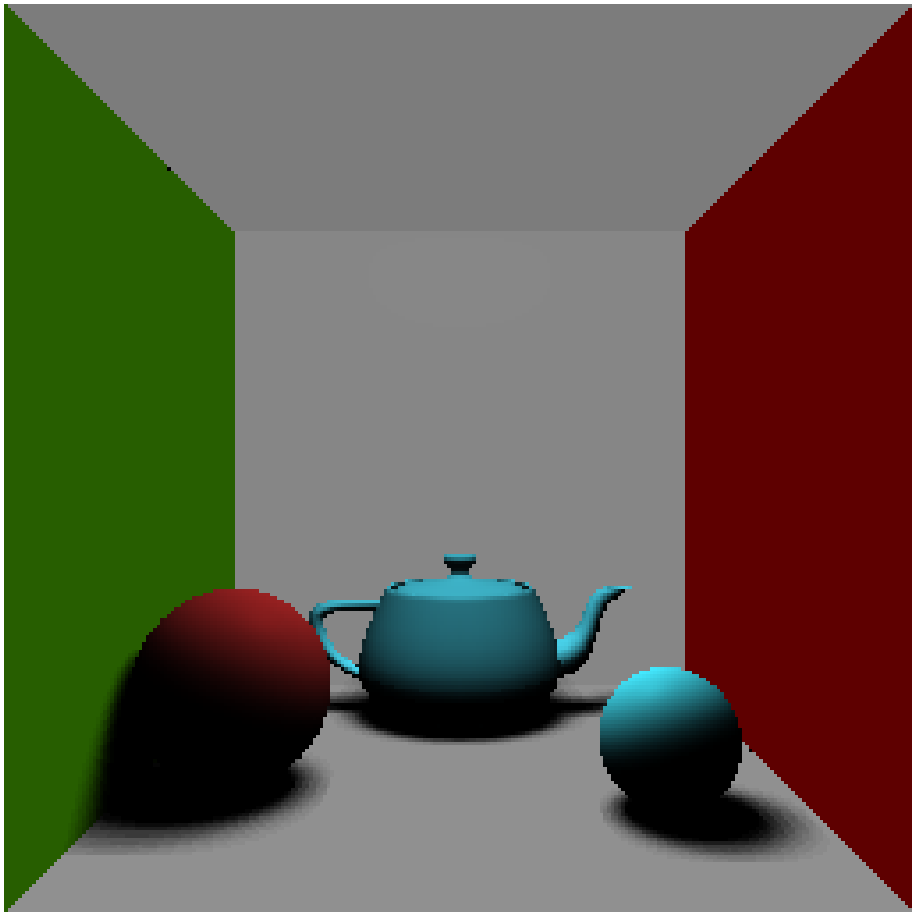


Figure 1: Ray traced image

The introduction of these two effects also requires the introduction of ray depth. Given that a ray always reflects/refracts off of a reflective/refractive surface, the ray could potentially continue travelling forever in a scene with many reflective/refractive surfaces. This is not a viable solution in a computer system therefore depth is introduced, which is simply the maximum number of times that the ray can be reflected/refracted. The `cast_ray_PM` function is called recursively until a depth counter reaches zero, at which point the recursion stops.

3.1 Reflection

Each object O in the scene has a predefined reflection coefficient C_{rl}. It determines whether or not the object should reflect an incoming ray, and how much of the reflected light should appear on the object. The `get_reflected_ray` function is simply an implementation of equation 1.

$$R = I - 2(N \cdot I)N \tag{1}$$

Given an incident ray I and a normal to the object hit position N the function returns the reflected ray R.

3.2 Refraction

Each object also has a refractive index C_{rf} and a refraction boolean. The two put together denote whether or not a ray should be refracted, and if so how much should it refract. A transmitted ray T is computed by the `get_refracted_ray` function using equations 2, 3, 4, and 5.

$$\eta = \frac{\eta_1}{\eta_2} \tag{2}$$

$$c_1 = N \cdot I \tag{3}$$

$$c_2 = \sqrt{1 - \eta^2(1 - {c_1}^2)} \tag{4}$$

$$T = \eta I + (\eta c_1 - c_2)N \tag{5}$$

Where: η_1 is the refractive index of the medium the ray is in, η_2 is the refractive index of the medium the ray will transmit into, N is the normal to the hit, and I is the incident ray.

There are a few special cases that are handled by the `get_refracted_ray` function, the first one being the fact that the ray might be entering or exiting the object that it hits. There is a simple solution to this and that is to test whether c_1 is positive or negative. If it is negative, the ray is entering the object, and if positive it is exiting the object. In order for the calculations to be correct in the former case c_1 needs to be negated, and in the latter N needs to be negated.

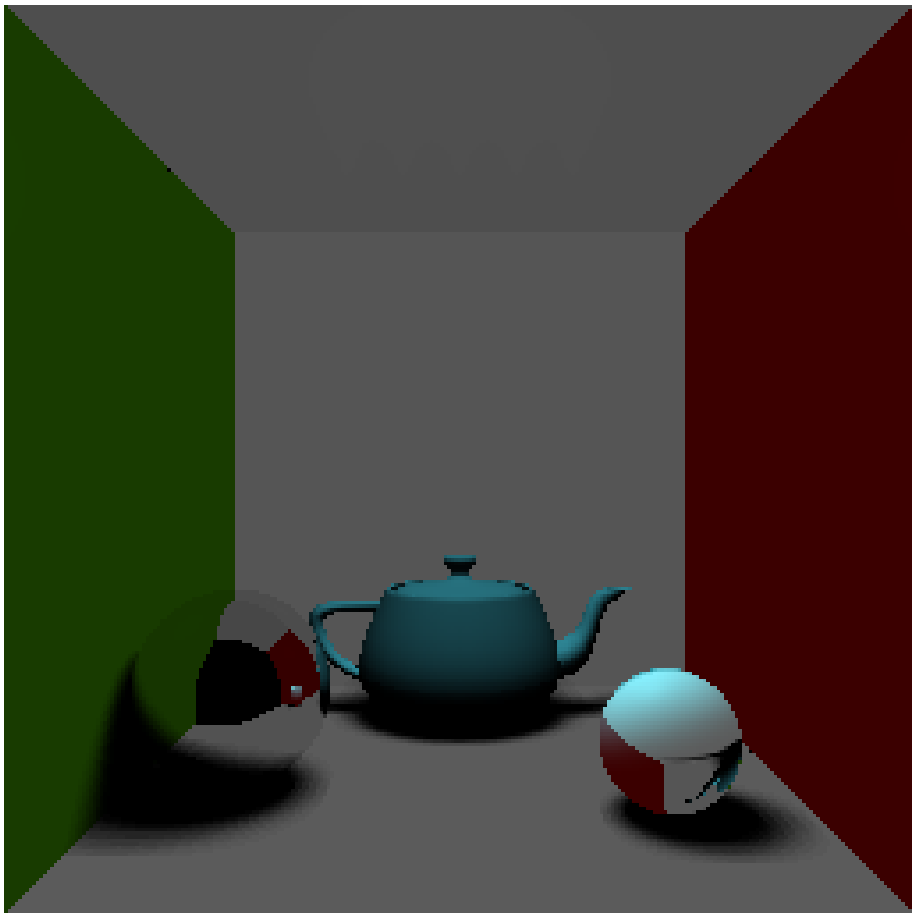


Figure 2: Image with reflection and refraction

Another special case is tested by checking the value of c_2. If that value is negative then the ray undergoes total internal reflection. In this case, the reflected ray is computed and returned. Otherwise T is computed using equation 5 and returned.

3.3 Image with reflection and refraction

Figure 2 shows the scene rendered using ray tracing with reflections and refractions. The sphere on the right has a refractive index of 1.517, the approximate refractive index of glass.

4 Photon Mapping

Ray tracing on its own is quite limited in what lighting effects it can produce. Despite the high quality of the effects it produces the image generated still looks very artificial. Jensen (1996) introduced the concept of photon maps to be used alongside standard ray tracing. They help generate effects such as caustics and colour bleeding, which using conventional ray tracing would be either computationally very expensive or impossible. Photon mapping has been implemented based on the guide of Jensen (2004).

The general principle of photon mapping is to shoot packets of energy, called photons, from all the light sources into the scene. As the photons interact with the scene they get absorbed/reflected/refracted and pick up the colour of the objects they interact with, creating the colour bleed effect. The density of photons in a certain area defines the illumination of that area, areas of the scene where more photons have been absorbed will become brighter in the final render of the image.

4.1 Creating photon maps

The previous ray traced image only required one stage, the rendering stage. Photon mapping introduces another stage, a photon map creation stage. The photon maps are generated using the `generate_photon_maps` function in the `image_generator.cpp` file.

The process of creating a photon map is presented in pseudocode in listing 2. All the lights in the scene generated are near the ceiling of the box, therefore the y direction has been limited to values between 0 and -1. Otherwise the direction of x and z is a random value between 1 and -1.

Listing 2: Photon map generation pseudocode

```
1 Set a number of photons to be generated
2 while (not enough photons){
3     For each light source {
4         Shoot ray in random direction into scene
5         Decide if the ray should survive
6         If survived, diffuse the ray back into scene
7         If hit, store photon
8     }
9 }
```

Listing 2 describes the creation of the global photon map (GPM), which is a map that stores photons that have been reflected diffusely at least once. The restriction on at least one reflection is so that the GPM does not contribute to direct illumination of a point in the scene. This is explained more in section 4.3.

Alongside the GPM a caustic photon map (CPM) is also created. Caustics require a lot of sharpness and detail in order to be rendered correctly, therefore

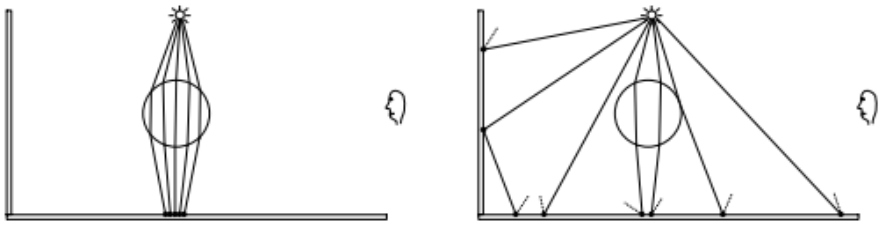


Figure 3: Generating two photon maps, caustic on the left and global on the right, taken from Jensen (2004)

a separate photon map is created that provides extra photons in ares where they are required, that is when interacting with refractive objects. The CPM is generated in the same way except that the rays from the light sources aren't shot in a random direction. They are shot on a random point within the refractive sphere present in the scene. Therefore they are guaranteed to hit a refractive object and thus contribute to the caustic illumination. Figure 3 shows a visual representation of generating both photon maps.

4.1.1 Storing the photon maps

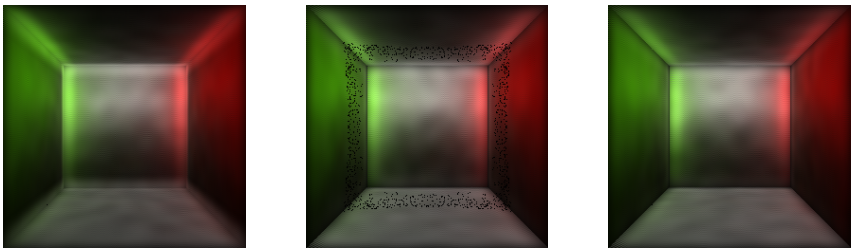
Both photon maps have been stored in the scene using KD-trees, using the Alglib[5] library. More specifically, the KD-tree stores the position of the photon in the scene, and a tag that refers that point to a member of a photon array corresponding to the photon map being queried. The KD-tree is generated at the end, after all the photons have been created, therefore it is balanced upon creation.

4.2 Sampling from the photon map

Retrieving photons from the photon map is made easy thanks to the Alglib library. When getting information from the photon map at a certain point P in the scene the `kdtreetsquerynn` Alglib function is called which retrieves all photons within a certain radius R around the point. The `ts` within the function name stands for *thread safe* which allows for parallel queries to the KD-tree, which is useful as the `main` function within `image_generator.cpp` uses simple multi threading to increase rendering speed. Once the KD-tree is queried, the tags from that query are extracted. As mentioned above the tags map onto indexes of a photon array where the colour and incident direction of the photon are stored.

The flaw with the function above is that it finds all photons in a sphere around the hit point. This becomes problematic around edges of objects as unwanted photons are being retrieved for the sample which can be seen in figure 4a. This is fixed by simply testing if the photon lies on the plane that is being

[5] `https://www.alglib.net/other/nearestneighbors.php`



(a) All samples (b) Samples lie on plane (c) Samples in a boundary

Figure 4: How sample selection affects the final image quality, only displaying the global photon map

sampled from. A vector V from the position of the hit point to the position of the photon is created, and a dot product of the hit point normal and V is computed. If that is equal to zero then the point lies on the plane. This can be seen in figure 4b, however the image has produced some noise. A discussion of this is made in section 5 however a simple fix for that is to give the dot product comparison a small boundary around 0. Figure 4c shows the render of samples that fit the following boundary: $-0.000001 > V \cdot N < 0.000001$, where N is the normal to the hit position.

4.3 Rendering using the photon map

The rendering stage when using photon mapping is essentially the same as when using only conventional ray tracing: rays are shot from the camera into each pixel and lighting is calculated for the object that is hit by the ray, although in this case the lighting calculation becomes more complex. Equation 6 shows an adapted version of equation 19 from Jensen (2004). The components of light are split into a direct lighting component L_d, lighting component from the global photon map L_g referred to as indirect illumination, and the lighting component from the caustic photon map L_c. The Phong lighting model used in the implementation is made up of two components: a diffuse component P_d and a specular component P_s. Adapting equation 19 from Jensen (2004) using the lighting components and Phong components yields equation 6.

$$I = L_d + \sum^{n}(P_d L_g + P_c L_g) + \sum^{m}(P_d L_c + P_c L_c) \tag{6}$$

The illumination I at a point is a sum of the direct illumination, the diffuse and specular illumination for n points found from the global photon map, and the diffuse and specular illumination for m points found from the caustic photon map. The `cast_ray_PM` function implements this equation. L_d uses the Phong lighting model to compute direct illumination from the hit to the light just as the two sums. However the P_d and P_c components within the sums use the incident direction of the photon as the direction that the light is coming from.

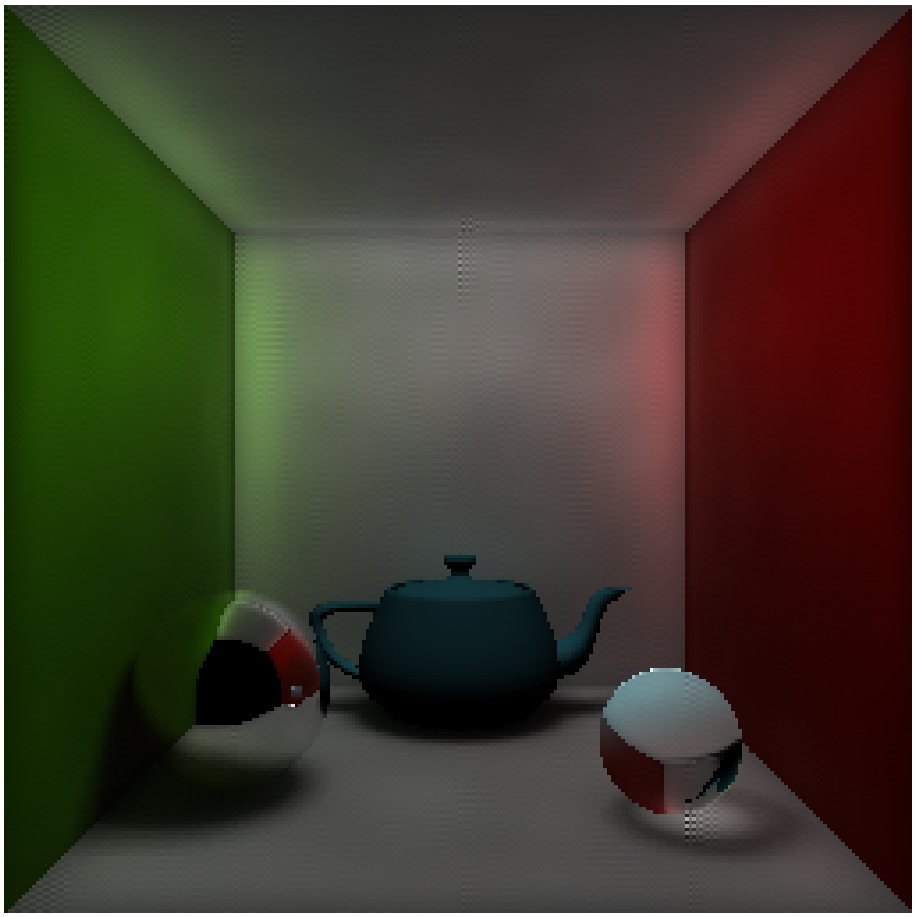


Figure 5: Image with photon mapping

The lighting contribution from that photon is calculated as if a light source was where the photon came from.

4.4 Image with photon mapping

Figure 5 shows a render with photon mapping used. Both the global and caustic photon map are used. The caustics are visible in the shadow area of the sphere on the right. Colour bleeding can mostly be visible on the back and top of the box from the sides, but also note the slight colour bleed from the teapot on the back wall just above it. Please note: this is not the final image render, the scene, colour bleeding, and caustic effects are subject to change based on bug fixes.

5 Bugs and workarounds

5.1 Hit imprecision

The sampling method described in section 4.2 exposes a complication with the hit precision. After some debugging it has been found that the implementation of ray tracing sometimes does not cause hits to be precise e.g. if the ray records a hit on an object where $x = 5$ sometimes the hit will store $x = 4.99999$ or any other permutation that is minimally offset from the correct value. This does not pose any issues until the photon map sampling is checked. For minimal offset values the dot product of the vector V and normal to hit N as described in section 4.2 will be minimally offset from 0 and therefore discarded from the sample. The workaround presented in the aforementioned section solves that issue, however it might introduce some unwanted colour bleed.

5.2 Sample validity testing

Another issue with the sampling method described in section 4.2 is that it does not generalize to curved surfaces. The dot product test method only checks the presence of photons on a flat surface, and therefore the teapot is not shaded in using the photon map information.

5.3 Exposure coefficients

Due to the concentration of samples in the caustic and global photon maps, what has been found is that some parts of the render are much brighter than other parts of it. This has been fixed by introducing exposure coefficients to each component of equation 6. They are set such that all the components look realistic and that their contribution to the final render can be visible.

6 Conclusion

Ray tracing and photon mapping prove to render high quality images when used in conjunction. Although the methods are computationally very expensive certain workarounds and optimizations can make rendering much faster. Current developments in ray tracing hardware such as Nvidia RTX cores[6] will make this technology more available and more accessible, to researchers and consumers alike.

[6] `https://www.nvidia.com/en-gb/geforce/20-series/rtx/`

References

Jensen, H.W., 1996. Global illumination using photon maps. In: X. Pueyo and P. Schröder, eds. *Rendering techniques '96*. Vienna: Springer Vienna, pp.21–30.

Jensen, H.W., 2004. A Practical Guide to Global Illumination Using Ray Tracing and Photon Mapping. *ACM SIGGRAPH 2004 Course Notes*. New York, NY, USA: ACM, SIGGRAPH '04. Available from: `https://doi.org/10.1145/1103900.1103920`.