Comparisons between Raytracing And Photon mapping

Author
Ishita Jain, 100985613
ishitajain@cmail.carleton.ca

Supervisor
Prof. Doron Nussbaum

December 17, 2018
Abstract

In this report, I explore the processes of raytracing and photon mapping. I explain both concepts, discuss the components and algorithms of each, and then display some of the images rendered using the algorithms. The results of various iterations and effects, such as motion blur, anti-aliasing etc. are shown. The two methods are also compared, and the conclusions state that despite the accuracy of image rendering through raytracing, it is a very time consuming process and that photon mapping helps in cutting down the rendering time, since the data structure computes the photons together.
I would like to thank Prof. Doron Nussbaum for giving me the opportunity to learn something new and explore a different area of Computer Science. I would also like to thank Pixar Animation Studios for providing a very detailed user manual about ray tracing and photon mapping, explaining these two concepts and their associated algorithms. I would also like to thank Henrik Wann Jensen, author of the book Realistic Image Synthesis using Photon Mapping, and Peter Shirley and R.Keith Morley, authors of Realistic Ray Tracing for providing some very insightful material on the topic.
Contents

1 Introduction.............................................. 1
   1.1 Motivation........................................ 1
   1.2 Ray tracing and Photon mapping.................. 2
      1.2.1 What is raytracing?......................... 2
      1.2.2 What is photon mapping?..................... 3
   1.3 Applications and uses in real life................ 4

2 Ray tracing.............................................. 5
   2.1 Components of Ray tracing....................... 5
   2.2 Algorithm Overview................................ 7
   2.3 Methods and effects............................. 9
      2.3.1 Anti aliasing............................... 9
      2.3.2 Motion Blur................................ 11
      2.3.3 Dielectric.................................. 11
      2.3.4 Emit lighting............................... 12
      2.3.5 Perlin Noise................................. 13
      2.3.6 Defocus Blur............................... 14
      2.3.7 Participating media/Volumes................ 15
      2.3.8 Diffuse Material............................ 16
      2.3.9 Solid Texture............................... 17
   2.4 Raytracers.......................................... 18
      2.4.1 Many Balls................................. 19
List of Figures

2.1 Code snippet showing how rays are passed through every pixel . . . . . . 8
2.2 Code snippet showing how rays are passed through every pixel . . . . . . 8
2.3 Code snippet showing rendering, with consideration of material . . . . . 9
2.4 Anti aliasing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.5 Motion blur . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
2.6 Dielectric . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
2.7 Light emission . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
2.8 Perlin Noise . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
2.9 Defocus Blur . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
2.10 Participating media/Volumes . . . . . . . . . . . . . . . . . . . . . . . . . 16
2.11 Diffuse Material . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
2.12 Solid Texture . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
2.13 ManyBalls Scene: First ray tracer . . . . . . . . . . . . . . . . . . . . . . 19
2.14 All effects final image: MotionBlur, texture, dielectrics, emit lighting etc. 20

3.1 Attempt at photon mapping . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
1 Introduction

Do you know what is common in between the movies Cars, Finding Nemo and Up, other than the fact that they are all from Pixar Studios? All these movies, along with many others, make use of the concept of ray tracing for generating the imagery and scenes we see in the movie.

In this report, I will be covering various topics to highlight what is ray tracing, what is photon mapping, how they are helpful in the real world, what I learnt about them. I will also be providing different sections explaining the procedure that went into producing the images and what I found the most interesting about them. I will also be providing a few code snippets in order to give a better understanding of the text. I have also given a section which highlights the different method of lighting and image producing that help to make the image look more real.

1.1 Motivation

In this world of increasing technology computer graphics is becoming more and more used to make images look better and more real. This project has helped me use the skills learnt in my undergraduate years of programming and also helped me learn a new area of Computer Science.
1 Introduction

The aim of this report is to learn to create visual effects and be able to understand how the images are made to look more realistic using different effects. Computer graphics is becoming more and more used since everything is working towards being digitalized as the world of technology increases. The topics learnt in the process of this report were: ray tracing, photon mapping, phong, and Gurrard lighting algorithms. All the images shown in the following report were generated using C++.

Global illumination, also known as indirect illumination, is a group of algorithms used in 3D computer graphics to add more realistic lighting to 3D images. Global illumination has two parts Point-sampling models and Integral based models. The process of ray tracing and photon mapping comes under the point-sampling models. Ray tracing algorithms used is mostly a class of algorithms which use repeated random sampling to compute results. Photon mapping is an alternative to ray tracing algorithms.

1.2 Ray tracing and Photon mapping

1.2.1 What is raytracing?

Ray tracing is a computer graphics rendering technique of generating images by tracing the path of light and computing the colour of a point in the scene. By following imaginary rays from the viewers eye to the objects in the scene, ray tracing can determine the visibility of objects within the scene. Scenes in ray tracing are generally described by the programmer.

Typically, each ray is tested for whether or not it intersects with any object in the scene. If a ray does, then the nearest object gets identified, and the algorithm estimates the incominglight at the point of intersection. Then, the material properties of the object are considered, and using this information, the final colour of the pixel is calculated.
1 Introduction

At the heart of all ray tracing algorithms is the computation of ray-object intersections. The basic ray tracing algorithm consists of two main calculations per pixel: finding the nearest surface point, and computing the colour at that point. If there is more than one intersection we usually want the nearest. Ray tracing is also useful for computing shadows i.e. beyond determining which objects are visible in which image pixels.

1.2.2 What is photon mapping?

Photon mapping is a global illumination technique, that approximately solves the rendering equation. The technique was developed by Henrik Wann Jensen [Pueyo and Schröder, 1996]. It is an extension of ray tracing i.e. photon mapping uses ray tracers as a base in order to render the image. The logic and technique used to create the image is different. The idea of photon mapping is to emit light from a light source. Light is represented as photons, which are given a specific power and are shot out from the light source.

Unlike ray tracing, photon mapping is a two-pass algorithm. In phase one, photon tracing, which is similar to ray tracing, a large number of photons are fired from the light sources into the scene that is to be created. Their path is traced and used to create two photon maps. This is done by emitting packets of energy (photons) from the light sources and storing these as they hit surfaces within the scene. Once again, the material properties are important - to determine if the photons are absorbed, reflected, refracted etc. Photons are only stored where they hit diffuse surfaces (or, more precisely, non-specular surfaces). The first map is a high resolution caustics photon map while the second is a low resolution map used during the rendering step, which is the second stage of the algorithm. The scene is rendered using a distribution ray tracing algorithm optimized by using the information in the photon maps. The second pass is similar to the pixel colouring from ray tracing i.e. the radiance of the pixel gets calculated. [Pueyo and Schröder, 1996]
1 Introduction

1.3 Applications and uses in real life

In today’s world, it seems that computer graphics are becoming more and more ubiquitous. We see the use of computer generated imagery (CGI) in so many different fields. Other than the Pixar movies, take for example hit movies within the Marvel Cinematic Universe, or TV shows such as Game of Thrones. In each of these, we see CGI being user to render scenes and imagery.

The concept was previously limited to film and television, since it required extensive resources to render the images. However, in the last 1-2 years, we have seen a massive leap in computing power offered by graphical processing units (GPUs), with initiatives from Microsoft, NVIDIA and AMD that will finally make ray tracing possible in real-time games, which means dazzling effects and much more immersive game worlds. [Hayward, 2018]
2 Ray tracing

This chapter builds upon the discussion from Section 1.2.1. It talks about the various components needed to build ray tracers. Subsequent to that, there follows an explanation of the ray tracing algorithm, which is followed by the various iterations of the ray tracers implemented.

2.1 Components of Ray tracing

the raytracer which shows many balls has the following components:

1. camera: is a class which has different positions of the vecColPos type

2. ray: this class is a simple depiction of what a ray is and shows position, point, direction and origin of the ray.

3. sphere: this class is simply creating the object that will be ray traced

4. surface and surfaceList: The surface list and surface classes are both abstract classes, just to keep track of the objects a ray will hit. I called it surface to signify the "surface" a ray hits.

5. materials: this class has many functions which are required in order to show shading, different materials like glass, metal, reflection, refraction.
6. vecColPos: this class is for color (r,g,b), vector and positions (x,y,z). This class is required for optimization of different functions. This class has every operation a vector does or can do like cross products, dot products, unit vector normalization, all the operations etc.

7. main class: Main class has all the rendering functions for images.

The next ray tracer has way more components since the image produced has a lot of other effects and a few different shapes as well, this ray tracer is extension of the previous one hence the components in the previous ray tracer remain the same. The components for the next ray tracer are as follows.

1. axisAligned: this class follows AABB that is axisAligned bounding box as it takes care of the overlapping of the objects in $A_{minX}, A_{maxX}$ and $B_{minX}, B_{maxX}$.

2. axisAlignedRect: this is same as the above AABB concept instead it is for rectangle so that we can use a rectangle to add light and other effects in our raytracer.

3. boundingVolumeHierarchy:

4. perlinNoise: for making the surface texture look more smoother. We generate random floats and then normalize the vectors.

5. box: in order to make the most famous cornell box, we need 5 faces of walls and this class helps us hold those classes in order to generate them.

6. texture: giving texture to our objects for example in the image shown on emit light the sphere has a texture, and the cube show in the last image has a texture and also this class helps with the perlin noise.

7. surfaceTexture: adds different simple texture by changing the colors at different coordinates to make it look like a texture.
8. constMed: helps us to provide with smoke/mist/fog effects like shown in the effect of Volumes below.

9. MotionObject: is to show some movement in the objects we create for example motion blur.

### 2.2 Algorithm Overview

The ray tracing algorithm is a simulation of the physical phenomenon of how we see. Rays are emitted from light sources, and bounce around, reflecting off various surfaces, until they hit the surface of our eye - where the object becomes visible (often also called the camera). To better understand the algorithm, it is essential to understand two terms:

1. Primary ray: The ray shot from the eye into the scene. Also known as camera rays.

2. Shadow ray: Ray from the object that is hit to the light source. Also called reflection or shadow rays.

The underlying logic of all raytracing algorithms is the concept of ray-object intersections. For each pixel in the image to be created, the algorithm shoots a primary ray into the scene. The direction of that primary ray is obtained by tracing a line from the eye to the center of that pixel. Then it is checked if the ray intersects with any objects in the scene. In some cases, the primary ray will intersect more than one object. When that happens, we select the object whose intersection point is the closest to the eye. We then shoot a shadow ray from the intersection point to the light. If this particular ray does not intersect an object on its way to the light, the initial hit point is illuminated. If it does intersect with another object, the second object casts a shadow on the first. [Anon, 2018] The code snippet below illustrates how the pixels in the image are looped over, and then intersections with objects are checked.
Based on the material properties of the object with which the rays interact, the rays will undergo physical phenomena such as reflection and refraction. These are important since they influence the colour and brightness/intensity at each pixel. The code snippet below shows how the reflection and refraction is accounted for.

Rendering is an essential part of the algorithm. Implemented as a function that loops
Ray tracing

over all the pixels in the image, the rendering generates and casts a primary ray into the scene per pixel. If the ray intersects with some object(s) in the scene, then the colour and intensity for each pixel is computed and made visible taking into account the properties of that material and how much light is incident on that particular pixel. Else, the colour and intensity of that pixel are set to default values (colour is generally black). In the end, the renderer is what generates the image by passing a ray through every pixel. This can be seen in the snippet below:

```java
surface *myShinyBallScene() {
    int n = 500;
    surface *list = new surface[n+1];
    for (int i = 0; i < n; i++) {
        for (int j = -1; j < 1; j++) {
            float choose_mat = drand48();
            vecColPos center = vecColPos(j, 0.2 + 0.8 * drand48());
            if ((center.vecColPos(0.2, 0).length() > 0) {
                if (choose_mat < 0.05) {// metal
                    list[i+1] = new sphere(center, 0.2, new lambertian(vecColPos(0.2, 0.2, 0.2), 0.54+1 + drand48(), 0.54+1 + drand48(), 0.54+1 + drand48(), 0.54+1 + drand48()));
                } else if (choose_mat < 0.95) {// metal
                    list[i+1] = new sphere(center, 0.2, new metal(vecColPos(0.2, 0.2, 0.2), 0.54+1 + drand48(), 0.54+1 + drand48(), 0.54+1 + drand48(), 0.54+1 + drand48()));
                } else { // glass
                    list[i+1] = new sphere(center, 0.2, new dielectric(1.5));
                }
            }
        }
    }
    list[n+1] = new sphere(vecColPos(0, 1, 0), 1.0, new dielectric(1.5));
    list[n+1] = new sphere(vecColPos(-3, 1, 0), 1.0, new lambertian(vecColPos(0.4, 0.2, 0.2), 0.54+1 + drand48(), 0.54+1 + drand48(), 0.54+1 + drand48(), 0.54+1 + drand48()));
    list[n+1] = new sphere(vecColPos(4, 1, 0), 1.0, new metal(vecColPos(0.7, 0.7, 0.7), 0.54+1 + drand48(), 0.54+1 + drand48(), 0.54+1 + drand48(), 0.54+1 + drand48()));
    return new surface_list(list, 11);
}
```

Figure 2.3: Code snippet showing rendering, with consideration of material

The section below talks about specific effects that are applicable to raytracing - and how they impact the rendering of the image.

### 2.3 Methods and effects

This subsection talks about various effects such as motion blur, anti aliasing etc., which are important considerations when trying to create scenes that better mimic real life.

#### 2.3.1 Anti aliasing
In order to understand the concept of anti-aliasing and noticing a difference between an anti-aliased image and a non anti-aliased image, two more spheres were created using different colours and shading them with different colors. The two images in Figure, show the differences in the edges the image to the left was created using the technique of anti-aliasing. Anti-aliasing is a technique used to add realism to the image by removing the jagged edges on curved lines and diagonals. Anti-Aliasing is taking the pixels next to the pixel around the curves which are jagged and to give those pixels the color or shade similar to the pixel that is giving the jagged look and start decreasing the shade of the color. Each pixel can be broken down into many samples and a ray is passed through each of these samples. The color of the rays that are passed is averaged in order to obtain the color for the next pixel or the pixel around it, in order to blur out the jagged lines around the curves.
2.3.2 Motion Blur

Motion blur is the apparent streaking of moving objects. Adding time element in order to show that image moves. The main change is to add the time in the for the rays to scatter and since the motion blur depicts movement of the sphere we need move the center from one point to another. Here the original centre, when is on the ground is center 0 and the centre1 is showing the movement of the sphere, which helps create the apparent streaking of moving objects. The basic idea is to apply stochastic sampling to time and space.

2.3.3 Dielectric
2 Ray tracing

Figure 2.6: Dielectric

Clear materials are known as dielectrics, a whole function is created in the materials in order to be able to create the water effect. When a light ray hits them, it splits into a reflected ray and a transmitted ray. The oval-shaped sphere on the left shows the dielectric of water. Even though it does not show the reflection of the ball next to it, it shows an inverted image of the surface it is on.

2.3.4 Emit lighting
Figure 2.7: Light emission

In the above image we can see light being emitted by a rectangle on the right side of the sphere and the area which is closest to the sphere is the brightest.

2.3.5 Perlin Noise
Perlin noise is technically an algorithm. It is a type of gradient noise, used to improve the realistic appearance of images in computer graphics. It is very useful when texturing.

2.3.6 Defocus Blur
For our code we could simulate the order: sensor, lens, aperture, and figure out where to send the rays and flip the image once computed (the image is projected upside down on the film).

### 2.3.7 Participating media/Volumes
Participating media is specified by scattering and absorption coefficients, saying how much light is absorbed and scattered to different directions within the volume. The image shows the effect of smoke.

2.3.8 Diffuse Material
Diffuse objects that don't emit light merely take on the color of their surroundings, but they modulate that with their own intrinsic color. Light that reflects off a diffuse surface has its direction randomized. So, if we send three rays into a crack between two diffuse surfaces they will each have different random behavior:

2.3.9 Solid Texture
Here we are making the colors of the surface procedural. In order to add the checker texture a function is created in the texture class, which creates the checker pattern by following odd and even variable names and then using angles and coordinate positions to determine the color of the squares in order to generate a square.

2.4 Raytracers

In this section, I will go over the various iterations of the raytracers I coded. Section 2.4.1 I will talk about the very first raytracer, discussing the iterations which lead to that scene and a few balls with mirror effects where created, in one of those balls the reflection of the rest of the scene is upside down. Lastly, the third iteration builds upon all the various effects discussed above in section 2.3. For each section, I have provided the image that was produced using that particular raytracer along with a few details which caught my interest and I found very interesting.
2.4.1 Many Balls

The image below took 6042.10s to produce which is about 1 hr 45 min. the height and width of the image given for this image was also 800,800 but only with 100 rays per pixel because of which it took way lesser time than the image produced in section 2.4.2. This image is rendered with the function called manyBallsScene which is of surface type. The main effect/material here is the mirror looking ball which shows the reflection of the whole scene, aslo the ball next to it which shows the whole scene inverted. A few glass balls are shown which look like they are floating because of the shadow effect underneath it. the glass balls also give a very transparent looking image, just like that of a glass and the object behind it can be scene. The positions and a few colours of the balls are produced randomly with the drand48() function.
2.4.2 All Effects

Figure 2.14: All effects final image: MotionBlur, texture, dielectrics, emit lighting etc.

This image took 82368.59s to generate, which is approx 23 hrs. The size of the image entered was 800,800 and the number of rays passed were per pixel were 10,000. this image shows all the effects described above. the main highlight would be the glass sphere even though the reflection of the light seems a little disoriented the shade at the bottom of the sphere is the same as that on top of it. The dielectric worked perfectly in rendering
2 Ray tracing

this image. the function that renders this image is called final which is of surface type. it takes in all the functions and classes we have created till now in order to produce the image. An array was created of 10,000 for the green cubes at the bottom surface. and the rest of the code is simply calling the different materials with appropriate parameter.
3 Photon mapping

This chapter builds upon the discussion from Section 1.2.2. It talks about the various components needed to implement photon mapping. Subsequent to that, there follows an explanation of the photon mapping algorithm, which is followed by my attempt at incorporating photon mapping into the existing raytracer.

3.1 Components of Photon Mapping

Photon mapping is an extension of raytracing. As a result, it uses the same components, although it does add a few of its own. One of the most important components in photon mapping is the kd-tree. This is explained more in the subsection below:

3.1.1 kd tree

A kd-tree is a k-dimensional tree, used for organizing points in a k-dimensional space, with particular use in applications that need to find the ‘n’ nearest neighbours. They are a special case of binary space partitioning trees.

It is very useful in the photon mapping algorithm owing to it’s speed and efficiency. The time it takes to locate one photon in a balanced kd-tree has a worst time performance of
3 Photon mapping

$O(\log N)$, where $N$ is the number of photons in the tree. [Jensen and Christensen, 2007]

However, since the tree is created only once and used many times during rendering, it is quite natural to consider balancing the tree. Another argument that is perhaps even more important is the fact that a balanced kd-tree can be represented using a heap-like data-structure which means that explicitly storing the pointers to the sub-trees at each node is no longer necessary. (Array element 1 is the tree root, and element $i$ has element $2i$ as left child and element $2i + 1$ as right child.) This can lead to considerable savings in memory when a large number of photons is used. [Jensen and Christensen, 2007]

Balancing a kd-tree is similar to balancing a binary tree. The main difference is the choice at each node of a splitting dimension. When a splitting dimension of a set is selected, the median of the points in that dimension is chosen as the root node of the tree representing the set and the left and right subtrees are constructed from the two sets separated by the median point. The choice of a splitting dimension is based on the distribution of points within the set. [Jensen and Christensen, 2007]

Efficiently locating the nearest photons is critical for good performance of the photon map algorithm. Fortunately, the simplicity of the kd-tree permits us to implement a simple but quite efficient search algorithm. A generic nearest neighbors search algorithm begins at the root of the kd-tree, and adds photons to a list if they are within a certain distance. For the $n$ nearest neighbors, the list is sorted such that the photon that is furthest away can be deleted if the list contains $n$ photons and a new closer photon is found. Instead of naive sorting of the full list, it is better to use a max-heap. A max-heap (also known as a priority queue) is a very efficient way of keeping track of the element that is furthest away from the point of interest. When the max-heap is full, we can use the distance $d$ to the root element (i.e. the photon that is furthest away) to adjust the range of the query. Thus we skip parts of the kd-tree that are further away than $d$. [Jensen and Christensen, 2007]
3 Photon mapping

3.2 Algorithm Overview

Photon mapping is an extension of raytracing, and is built on top of it - in the sense that in order to render an image with photon mapping, a raytracer is still used. The major consideration in photon mapping is that unlike raytracing, photon mapping is a two pass algorithm i.e. it is done in two separate phases. In the first pass, the photon map is generated, and in the second pass, the image is rendered. Light packets, also known as photons, are sent out from the light source and camera, and traced independently, until some termination criterion is met. Then, they are connected in a second step to produce a radiance value. [Jensen and Christensen, 2007]

It is important to know the definition of the term caustics - the patterns caused by the reflection and refraction of light, usually visible on nearby surfaces as patches of light.

Similar to raytracing, the intersection of photons and objects is of interest. Whenever a photon intersects with a surface, the intersection point and incoming direction are stored in a cache called the photon map. Generally, two or maybe even three different photon maps get used. One of them is limited to use only for caustics, while a second is used for global illumination purposes. Once again, the material properties are considered, and phenomena such as reflection, refraction and absorption are accounted for.

The photons get stored in a kd-tree, which is described in detail above.

In the second pass, the photon map from pass one is used to estimate the radiance of each pixel. In the second pass, the rendering pass, uses statistical techniques on the photon map to extract information about incoming flux and reflected radiance at any point in the scene. The photon map is decoupled from the geometric representation of the scene. This is a key feature of the algorithm. [Jensen and Christensen, 2007]
3 Photon mapping

3.3 Attempt to implement photon mapping

The image below is what I was able to produce through my attempts at creating a photon mapped image. I was unable to fix my code entirely. As can be seen in the image below, the intensity seems quite bright and uneven. Beyond that, it is also visible that the colouring of the pixels did not happen too well.

However, even in this limited attempt, it can be seen that the caustics of photon mapping seem to be better than that of raytracing. Additionally, this image got rendered much quicker when compared to raytracing for similar size images - this took around 20 minutes.

Figure 3.1: Attempt at photon mapping
4 Comparison of Ray Tracing and Photon Mapping

There are a few key differences between the ray tracing and photon mapping algorithms, both in terms of what can be achieved, and how they work:

1. One of the biggest advantages of photon mapping is that it is able to handle some aspects of the real world including diffuse inter reflections (bleeding of coloured light) and caustics (focused light) which ray tracing cannot. Both these issues are indirect illumination of diffuse surfaces

2. Photon mapping algorithm is a two pass algorithm when compared with the ray tracing algorithm

3. The photon mapping algorithm also requires a preprocessing step for storing the colour and illumination data in a data structure, which gets interpolated and looked up to estimate secondary lighting at render time

4. The error/noise in Photon mapping tends to be of low frequency, while ray tracing tends to have high frequency noise [Jensen and Christensen, 2007]

5. Photon mapping uses a data structure called the kd-tree, to store the photon map. Although this makes the algorithm more efficient, the side-effect is that it increases memory consumption (used for storing the photons) when compared with ray tracing
6. Computationally, photon mapping works faster than ray tracing when making the same scene [Jensen and Christensen, 2007]
5 Conclusions and Recommendations

5.1 Conclusion

According to the research and readings done I would like to come to the conclusion that ray tracing can be very inefficient and time consuming. Creating just one image which has 10,000 rays across 800,800 sheet takes approx 23hrs to produce. After studying the Photon mapping algorithm I understood that the data structure used and the functions performed would produce the image much faster and it could also produce images without any glitches.

There are a variety of things I learnt from this project a few being, the ability to understand what ray tracing and photon mapping is and how vastly it is used also understanding that how rapidly this area of computer science is growing.

After doing this project I do feel that I have learnt something very new and different, something which is not ordinary.

5.2 Recommendations

The recommendations I would like to give would be
5 Conclusions and Recommendations

1. Try to start with less pixels when doing trial and error because that would save up lots of time, while doing this project I noticed that I spent a lot of time waiting for the image to be produced in order to see what is wrong and what is right and how I can further improve it. Instead if I just perform the image with a few rays and a smaller height and width the image would be created faster and I would save a lot of time. When satisfied with the image being produced I could simply input the rays and the width and height later to produce the final image and then record the run time.

2. It would have helped to research a little more about Photon mapping in order to have been able to create an image for it in order to give a proper image comparison. I feel I got very engrossed in doing ray tracing that I didn’t give much weight to photon mapping.

5.3 Future work

I would like to increase the size of the images and pass more rays. Also I would like to add a few more shapes instead of just the basic square and circles. I would also like to integrate photon mapping into the raytracer, and have it working well.
6 References


Christensen, P. (2005) Global illumination for professional 3D animation, visualization, and special effects, Springer Verlag


Klawonn, F. (2012) Introduction to computer graphics, Springer


