Monte Carlo Ray Tracing renderer

Olle Grahn
Jonathan Bosson

October 30, 2015

Abstract

This report describes methods for rendering images with global illumination. It also gives a detailed explanation of the renderer that was developed in this specific project. It presents some images that have been rendered in this project. The report concludes with a discussion about the quality of these images and the efficiency of the renderer.

Contents

List of Figures

Chapter 1

Introduction

Rendering photo realistic images has always been something to strive towards in 3D computer graphics but they are yet out of reach. In order to achieve a good representation of a physical scene, a lot of considerations are needed when the light is calculated. Previous shading techniques in 3D computer graphics have utilised a local illumination model which calculates how light from a lightsource gets reflected on an object. The shortcoming of this model is that the only part that affects the light in the scene is the direct emission from the light sources, but such limitations were required to render realistic results with a acceptable computation times.

A global illumination model considers not only reflected light from objects with emission, but also reflected and refracted light from objects in the scene. Algorithms to render a global illuminated image have been around for some time, the two most prominent methods are Radiosity and Ray tracing or a combination of the two. The ray tracing method's final result is directly dependant on the density of rays that is sent through the scene. The algorithm needs multiple rays per pixel in order to keep the rendered image from being noisy, which meant that its use previously was practically impossible due to the long computation time. However, with recent major improvements in computers CPU and GPU as well as extending the ray tracing with Monte Carlo and photon mapping techniques, a high quality image is within reach using a Monte Carlo Ray Tracing method.

In this project a Monte Carlo ray tracer has been implemented. It renders polygonal objects with Lambertian reflectance models.

The report will first describe different methods used when rendering images with global illumination in the Related work chapter. It will then go on to explaining how the project was carried out and which tools were used. Next the renderer will be explained. This explanation is divided in two chapters; the first talks about the data structures of the renderer and the second will explain the rendering algorithm and related algorithms. Finally the results will be show in chapter six followed by an analysis and discussion of these results as well as the conclusions drawn from the project. Pseudo code will be presented in the report and will be based on the actual code of the project. It will be influenced by the C++ language.

Chapter 2

Related Work

2.1 Radiosity method

Radiosity is a global illumination method for scenes with isotropic surfaces. Compared to local illumination methods, the radiosity algorithm allows soft shadows and subtle lighting effects to become visible as well as some soft color bleeding making the image look more realistic. The radiosity equation is used to evaluate the radoisity at a given surface.

$$M(x) = M_e(x) + pE(x) \tag{2.1}$$

Where $M(x)$ is the total radiosity emitted, $M_e(x)$ the emitted radiosity if the surface is a lightsource, p the fraction of the irradiance that is re-emitted and $E(x)$ the irradiance that falls upon the surface x[1].

The radiosity method consists of a number of passes. The first pass illuminates the reflection of surfaces directly hit by a lightsource. The second pass will further carry a fraction of the previously illuminated light to visible surfaces around its source. For each pass, the rendered image normalises more and becomes more realistic.

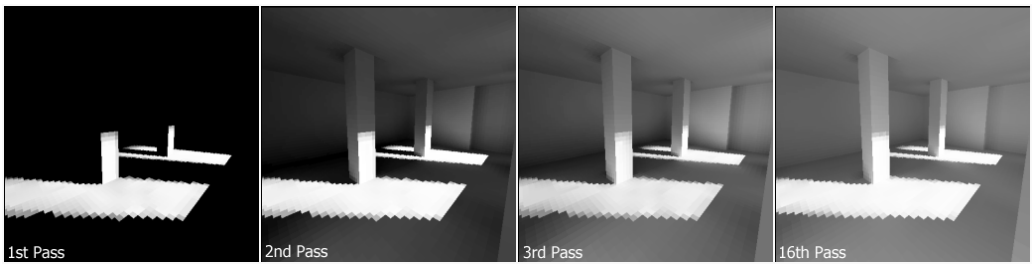

Figure 2.1: Radiosity progress

2.2 Whitted Ray tracing method

The ray tracing technique generates an image by tracing the path of light in the virtual scene. Starting from an imaginary eye with a direction each ray is traced through a pixel in the image plane into the scene, creating a ray path for every time it intersects with an object[2]. The method is able to render very photo realistic images at the cost of a greater computation time. Because of this ray tracing is well suited for applications where the rendering can be done ahead of time, such as in movies and

still images and visual effects. For real time rendering applications will either limitations to quality or optimisations be needed to be made in order to meet the required render speed.

A new ray has a chance to be created each time the initial ray intersects with a surface. This new ray's direction is dependant on the met object's bidirectional reflectance distribution function (BRDF), which determines how well an object reflects different colors as well as in what direction. By linking each new ray with its parent a ray path is created that is later used to evaluate the pixel's color value. The ray path ends when the ray gets terminated instead of reflected on an intersection.

At each intersection point, an additional `shadow ray` is created facing each lightsource. If the ray intersects with any opaque object, the surface is in shadow and should not be illuminated by the light. This gives the image more realism by simulating shadows, a task which is difficult with other methods. Another advantage with the ray tracing method is due to the computational independance between each ray, it is possible to parallel computing. A disadvantage with ray tracing is even though it handles reflection and refraction accurately the result is not necessarily photo realistic. An image is photo realistic when the approximation is close to the analytical rendering equation. To better this approximation, the Whitted Ray tracing can be extended with Monte Carlo.

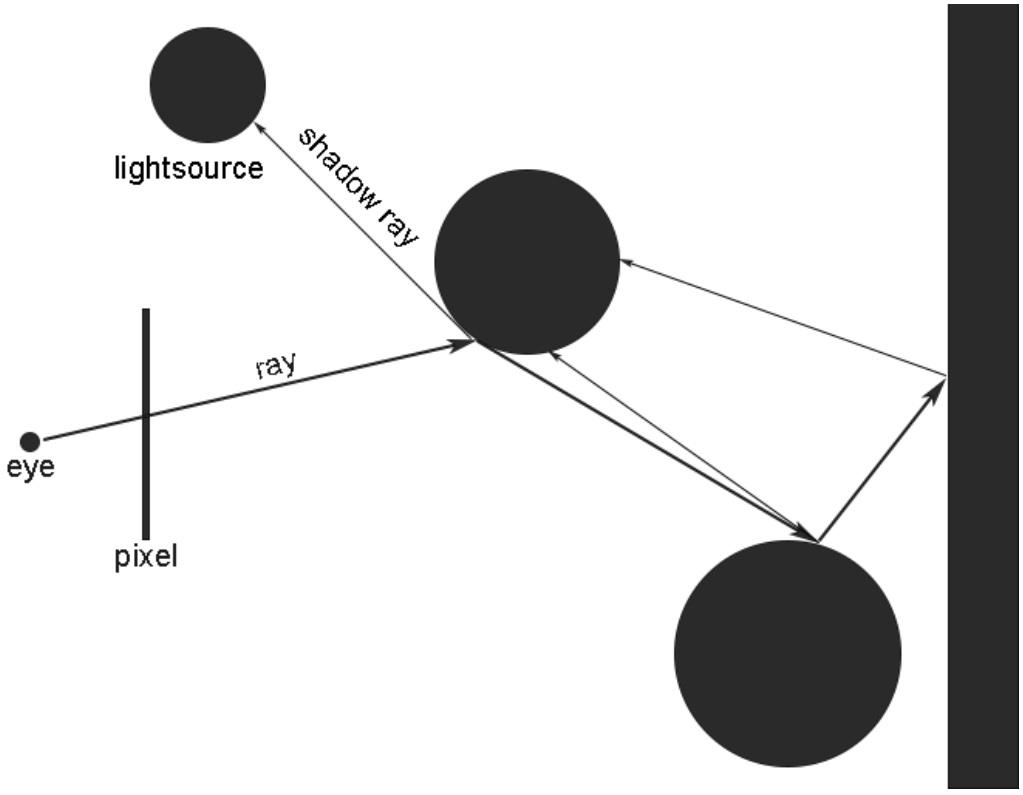

Figure 2.2: Ray path

Apart from initial position, direction and links to parent and childs, the ray also carries a vector variable called importance. Initially the value is set to 1. When a new ray is generated through an intersection, its importance value is calculated by using its parents importance multiplied with the hit surface's BRDF and π. The importance is calculated to later be used in the rendering equation on the second pass.

2.3 Monte Carlo Ray tracing

Monte Carlo is a method that relies on repeated random sampling to obtain numerical results and works as an extension to previously described Whitted Ray tracing. In order to convert the analytical

rendering equation shown in equation 2.3 to a numerical solution, the integral needs to be approximated.

When a ray hits a surface, it has a probability to either reflect or terminate. Which direction the reflected ray is traced is also returned through a probability ditribution depending on the BRDF. Monte Carlo method offers the equivalent conversion from the analytical integral to an expected value[3].

Let x be a random value distributed by the PDF p(x). The probability that x will take on some value in the inverval [$-\infty$, x] is given by the cumulative distribution function.

$$cdf(x) = \int_{-\infty}^{x} p(x)dx \tag{2.2}$$

The chance that x will be drawn between $[-\infty, \infty]$ is 1, representing the maximum of the $cdf(x)$. By inverting the $y = cdf(x)$ into $x = cdf^{-1}(x)$ and drawing random values y_i using a uniform PDF, y_i can be mapped onto random numbers with the $p(x)$ PDF.

2.4 Photon Mapping

In the construction of a photon map, light packets called `photons` are traced from the light sources into the scene. When a photon hits a surface, the intersection point and incoming direction are stored in a kd-tree data structure. Usually two separate maps are created for caustics as well as global light[4]. At an intersection the same Monte Carlo method is used to determine if the photon terminates or toward what direction it is traced.

The photon map is later used in the second pass of the rendering in order to more efficiently determine the light sources contribution to the reflected light at a surface than previous method with only shadow rays.

2.5 Rendering algorithm

The rendering equation is used to evaulate the color of each pixel.

$$L_o(x, \omega) = L_e(x, \omega) + \int_{\Omega} f_r(x, \omega_i, \omega,)L_i(x, \omega_i)d\omega_i \tag{2.3}$$

It describes that the color that is reflected from a surface is the sum of the surface's emission and the integral over the BRDF and incoming light. x is the surface position, ω is the direction the light gets reflected towards and ω_i is the incoming light direction.

To accurately calculate the color value of a pixel the rendering equation has to be computed recursively along the ray path. By stepping backwards along the ray path, the contribution of light from objects around the surface are taken into consideration. That is why the renderer has two passes. In the first pass ray is traced through each pixel into the scene, creating a ray path and calculating the importance along each reflected ray. In the second pass the pixel color is evaluated recursively by stepping backwards in the ray path calculating each intersection points reflected. Due to the fact that in the creation of the ray path the importance was calculated, the relation between the previous importance value divided by the current can be used instead of the BRDF.

As described in the Monte Carlo section, when the ray intersects with a surface it has a chance to terminate. Because of this we need to rescale the reflected value with the probability of reflecting.

The rendering equation is thus changed to as shown below. W is the importance of the current ray and W_c is the importance of the previous ray. L_i is the incoming light to the surface x, which conists of both contribution from the lightsources through shadow rays as well as contribution from previously visited surfaces in the ray path.

$$L_o(x, \omega) = L_e(x, \omega) + ((W_c/W)/P)L_i(x, \omega_i) \tag{2.4}$$

Chapter 3

Method

The project was implemented over two weeks by two developers, programming the majority of time in pairs. Git and github were used as version control system to keep the code structured and minimise problems in the development. All code has been written in the C++ language.

Visual Studio 2013 was used as development environment. The only dependency that is used is the OpenGL mathemathics API `glm` for linear algebra computations, making the project cross platform.

Chapter 4

Data Structure

4.1 Geometry

All objects in the scene are polygonal objects that are defined by a vertex list and a triangle list. A vertex consists of a 3D vector that defines the position of that vertex in model coordinates. Support for vertex normals and texture coordinates has also been implemented. Why this support has been implemented will be discussed later in the report. A triangle consists of three integer values that define the vertices of the triangle in counter clockwise order. They all have a position and an orientation in world coordinates. BRDF and emission properties are defined as two 3D vectors per object. As mentioned there is also support for textures which enables variable BRDF and emission properties on the object surface.

Listing 4.1: `pseudo code` of a scene object

```
struct vertex {
    float xyz[3];
    //float nxyz[3];
};

struct triangle {
    uint index[3];
};

struct quad {
    uint index[4];
};

struct texST{
    float st[2];
};

class Mesh
{
public:
    //different functions for getting and setting relevant variables
    get...();
    set...();

protected:

    vertex* vertexArray;
    triangle* indexArray

    glm::dvec3 position;
    glm::dmat4 orientation;

    glm::dvec3 lightEmission;
    glm::dvec3 brdf;
    double P;
```

```
}
```

The reason all objects are explicit polygonal objects is that it allows for arbitrary shapes and enables general algorithm handling. Algorithms like searching for ray intersections are the same for all objects, i.e. they do not change if the shape of the objects change. Descriptions of the used algorithms can be found in section 5.2. This makes the code cleaner and more readable and allows for more diverse scenes. Using implicit geometry could improve performance, but when using both explicit and implicit geometry many algorithms have to change depending on the object.

The intended structure of the geometry in this project includes bounding boxes per object as well as some geometric sorting of the vertex and triangle lists. For example algorithms like octrees or k-d trees. They are not needed to render but would drastically decrease the amount of computations when rendering detailed objects. These two things have not yet been implemented.

4.2 Camera and Image

The camera is a separate object from the other scene objects. This is because it is an important object and it will never be rendered.

The camera has a position, direction and orientation like the other objects but because it will never be rendered it does not have a vertex or triangle list.

Listing 4.2: `pseudo code` of a scene object

```
class Camera
{
public:
    Camera(glm::vec3 origin, glm::vec3 end);

    set...();
    get...();

private:

    glm::mat4 orientation;
    glm::vec3 direction;
    glm::vec3 position;
};
```

The direction and position of the camera is chosen and from these the cameras basis transform is created by the direction and an arbitrary orthogonal vector to the direction

The perspective of the camera does not explicitly exist in the camera class but is defined by an implicit view plane in the main rendering loop. The dimensions of this view plane is calculated from the chosen resolution of the image.

Listing 4.3: `pseudo code` of a scene object

```
    Image img(1920, 1080);

    double xMax = img.x / img.y;
    double yMax = 1.0;
    double xMin = -xMax;
    double yMin = -yMax;
```

An Image is a very small class that defines an image resolution, allocates and stores the pixel data of the image and implements functions that saves the image in a certain format. As of the writing of this report the implemented file output format is the bitmap format.

The pixel data is stored as a three dimensional matrix where the dimensions are the resolution of the image multiplicated with the dimensions of the color space. For this project only the red, blue and green channels are used but there could be some use for a fourth channel storing an alpha value.

Listing 4.4: pseudo code of a scene object

```
class Image
{
public:
    Image(int _x, int _y);

    void saveAsBMP();

    int x; int y;

    //3 dimensional matrix where dimensions = x*y*3
    glm::dvec3** imgData;
};
```

4.3 Light and Importance Rays

The Ray class is the class that handles all importance and light transportation in the scene. It has access to all scene data and a global random number generator. Rays have a pointer to the parent ray, i.e. the previous ray that spawned the current ray. This pointer is null if the parent is the camera. They also have two pointers to their possible ray children. One child for reflection and one child for transmission. These pointers are null if a reflection or transmission did not happen. Rays also consist of three vectors that define the origin of the ray, the direction and a possable hit location. These three vectors are always assumed to be in world coordinates. The class also includes two index variables that points to the object that it hit and the triangle it hit in that object. The final thing stored in a ray is its importance, defined as a three dimensional vector.

There are also two very important functions in the ray class; the transportation function and the evaluation function.

The transportation function is called once per ray, it calculates where the ray hits an object, the direction of possable reflections and transmissions and how much importance the reflection and transmission rays will have.

The evaluation function is a recursive function that is also called once per ray. Here direct light on the rays hit location is calculated with shadow rays. This direct light, the incoming light from the rays children and the BRDF of the hit object is entered into the rendering equation and the result is returned to the parent ray.

More details on the transportation function and evaluation functions will be given in the next section.

Listing 4.5: pseudo code of a scene object

```
class Ray
{
public:

    Ray(vec3 _origin, vec3 _direction, Ray* _parent, _sceneData, RNG _rng, vec3 _W);

    void Transport(vec3 _origin, vec3 _direction);

    vec3 evaluate();

    vec3 W;

private:

    vec3 direction;
```

```
    vec3 origin;
    vec3 hit;
    vec3 hitNormal;

    int objectIndex;
    int triangleIndex;

    Ray* parent;
    Ray* tChild;
    Ray* rChild;

    std::vector<Mesh*>* sceneObjects;

    RNG* rng;

};
```

4.4 Photon Map

Chapter 5

Rendering and related algorithms

5.1 Image and ray creation

For each pixel a number of ray paths are created. The number of ray paths per pixel can be changed but higher amounts of ray paths per pixel results in better quality. These first rays are created as a vector between the cameras position in the camera coordinate system and a pixel in the implicit view plane. The directions also have a random offset within the pixel to aquire a range of different samples. The vectors are then transformed to the world coordinate system.

Listing 5.1: `pseudo code` of a scene object

```
Ray* rIt;
double x = (double) img.x / (double) img.y;
double y = 1.0;//(float) img.y / (float) img.x;
double xCo = -x;
double yCo = -y;
double rX; double rY;

RNG rng;

double xStep = (2* x) / (double)img.x;
double yStep = (2* y) / (double)img.y;
vec3 tempRGB;

for (int i = 0; i < img.y; i++)
{
    yCo += yStep;
    xCo = -x;
    for (int j = 0; j < img.x; j++)
    {
        xCo += xStep;
        tempRGB = vec3(0.0, 0.0, 0.0);
        for (int p = 0; p < rPP; p++)
        {
            rX = rng.dist(rng.mt) * xStep;
            rY = rng.dist(rng.mt) * yStep;

            vec3 rDirection = mat3(cam.getCTransform()) * (vec3(xCo+rX, yCo+rY, -1.0));
            vec3 rPos = vec3(cam.getCTransform() * vec4(0.0, 0.0, 0.0, 1.0));

            rIt = new Ray(rPos, rDirection, nullptr, scene, rng, vec3(1.0));
            tempRGB = tempRGB + rIt->evaluate();

            delete rIt;
        }
        img.imgData[i][j] = tempRGB / (double)rPP;
    }
}
```

5.2 Raypath and importance calculations

As soon as the outer ray is created it starts recursively calling the transportation function. The first thing that is done in this function is finding the triangles and objects that are intersected by the ray. When these are found the closest of these intersection is stored. This is done with a slightly modified version of the Möller–Trumbore intersection algorithm [5].

When the object that was hit has been found, two random numbers between 0 and 1 are generated. The first of these is compared to the probability variable of the object. This variable defines the likelyhood that a ray hitting the surface will be reflected or absorbed. It is chosen by the programmer and will affect the quality of the final image. If the random number is larger that the probability variable the ray path is terminated. If it is smaller a reflection and possible transmission is calculated.

As of the writing of this report the scene used in this project only contains Lambertian reflectors. The integral of the rendering equation is computed with a Monte Carlo scheme and the chosen probability function is defined in equation 5.1 .

$$pdf = \frac{cos(\theta)}{\pi} \tag{5.1}$$

The two random numbers are used as arguments in the resulting inverse cumulative distribution function and two random angles are aquired, see equation 5.2.

$$u_1, u_2 \in [0, 1] \quad \phi = 2\pi u_1 \quad \theta = cos^{-1}(\sqrt{u_2}) \tag{5.2}$$

These two angles are used to calculate the direction of the new reflected ray. This is done by rotating one of the vectors in the hit triangle around the surface normal in the hit location, which is orthogonal to the triangle edges, by the angle ϕ. Then the surface normal is rotated around this new vector by the angle θ. The resulting vector is the direction of the reflected ray.

Listing 5.2: `pseudo code` of a scene object

```
//the rotate() function is a function that rotates a vector by an angle around another vector
rotationVector = rotate(triangleVector, Phi, surfaceNormal);
rayDirection = rotate(Normal, Theta, rotationVector);
rChild = new Ray(hit, rayDirection, this, sceneObjects, _rng, M_PI*sceneObjects->at(objectIndex)->BRDF()*W);
```

Finally an new ray is created and set as the reflection ray of the current ray. This new ray's origin is the hit of the current ray, its direction is the previously calculated direction, its parent is the current ray and its importance is the current ray's importance multiplied with π and the BRDF of the object that was hit by the current ray. When created, this new ray also calls the transportation function.

Listing 5.3: `pseudo code` of a scene object

```
                    //origin    //direction    //parent     //importance
reflectedChild = new Ray(hit, reflectedDirection, this, M_PI*sceneObjects->at(objectIndex)->BRDF()*W);
```

This process repeats until a ray is absorbed and there are no more reflections or transmissions. When this happens the ray path is complete and has ended on a diffuse reflector.

5.3 Evaluating the raypath

Evaluating the raypath is a matter of traversing it from its end back to the camera. This is done with the evaluate function. It is a recursive function that called the first time from the ray connected to the

camera. This starts a traversal along the ray path since the evaluate function of a ray also calls the evaluate function its ray children as long as it has any.

The first thing that is always done in the evaluate function is the shadow ray calculations. These rays are created as vectors between the hit location of the current ray and a random point on one of the light sources in the scene. A certain number of these rays are created for each light source in the scene, more rays means more calculations but higher image quality. If something is between the hit location and the light source that shadow ray does not contribute to the direct light at the hit location, otherwise it recieves emission from the light source. This intersection check is performed with the Möller-Trumbore as before. The contributions from all shadow rays of a certain light source are summed and then normalized as seen in listing 5.3.

This direct light contribution is then added to the rendering function along with the light from the reflected and transmitted children of the current ray as well as the emission contribution if the object that was hit emits any light. The light contribution from the children is found by calling the evaluate function for the children as well. This is done from within the evualate function of the current ray thus making the function recursive. This call is only made if the ray has children otherwise only the direct light and emitted light is returned. If it has no children and did not hit an object a null vector is returned.

Listing 5.4: `pseudo code` of a scene object

```
for(int i = 0; i < nrOfShadowRays; i++)
{

...

    shadowLight += lightSource->getArea()*lightSource->getLightEmission()*object->getBRDF()*
    (dot(shadowNormal, (-1.0)*shadowDir)*dot(hitNormal,shadowDir)/(max(shadowLength*shadowLength, 0.01)));

}
shadowLight = shadowLight / nrOfShadowRays;

if (rChild && !tChild)
{
    return sceneObjects->at(objectIndex)->getLightEmission() +
    ((rChild->W / W)/sceneObjects->at(objectIndex)->getP()) * rChild->evaluate() + shadowLight;
}
else if (rChild && tChild)
    return sceneObjects->at(objectIndex)->getLightEmission() +
    (sceneObjects->at(objectIndex)->getP())*((rChild->W + tChild->W) / W)*(rChild->evaluate() +
    tChild->evaluate()) + shadowLight;
else if (tChild)
    return sceneObjects->at(objectIndex)->getLightEmission() +
    ((tChild->W / W)/sceneObjects->at(objectIndex)->getP()) * tChild->evaluate() + shadowLight;
else if (objectIndex != -1) // Ray did hit an object
    return sceneObjects->at(objectIndex)->getLightEmission() + (shadowLight);
else // no hit
    return glm::dvec3(0.0, 0.0, 0.0);
```

When the ray path have been traversed the total light contribution along that ray path is added to the total value of that pixel. When all ray paths for a pixel has been calculated the total value is divided by the number of rays per pixel.

Finally when all pixel values have been calculated they are normalized by dividing all pixel values by the largest pixel value in the image and then taking the root of this division.

Listing 5.5: `pseudo code` of a scene object

```
for(int i = 0; i < img.y; i++)
        for(int j = 0; j < img.x; j++)
        {
            img.imgData[i][j].x = sqrt(img.imgData[i][j].x/maxIntensity);
            img.imgData[i][j].y = sqrt(img.imgData[i][j].y/maxIntensity);
            img.imgData[i][j].z = sqrt(img.imgData[i][j].z/maxIntensity);
```

```
}
```

Chapter 6

Results

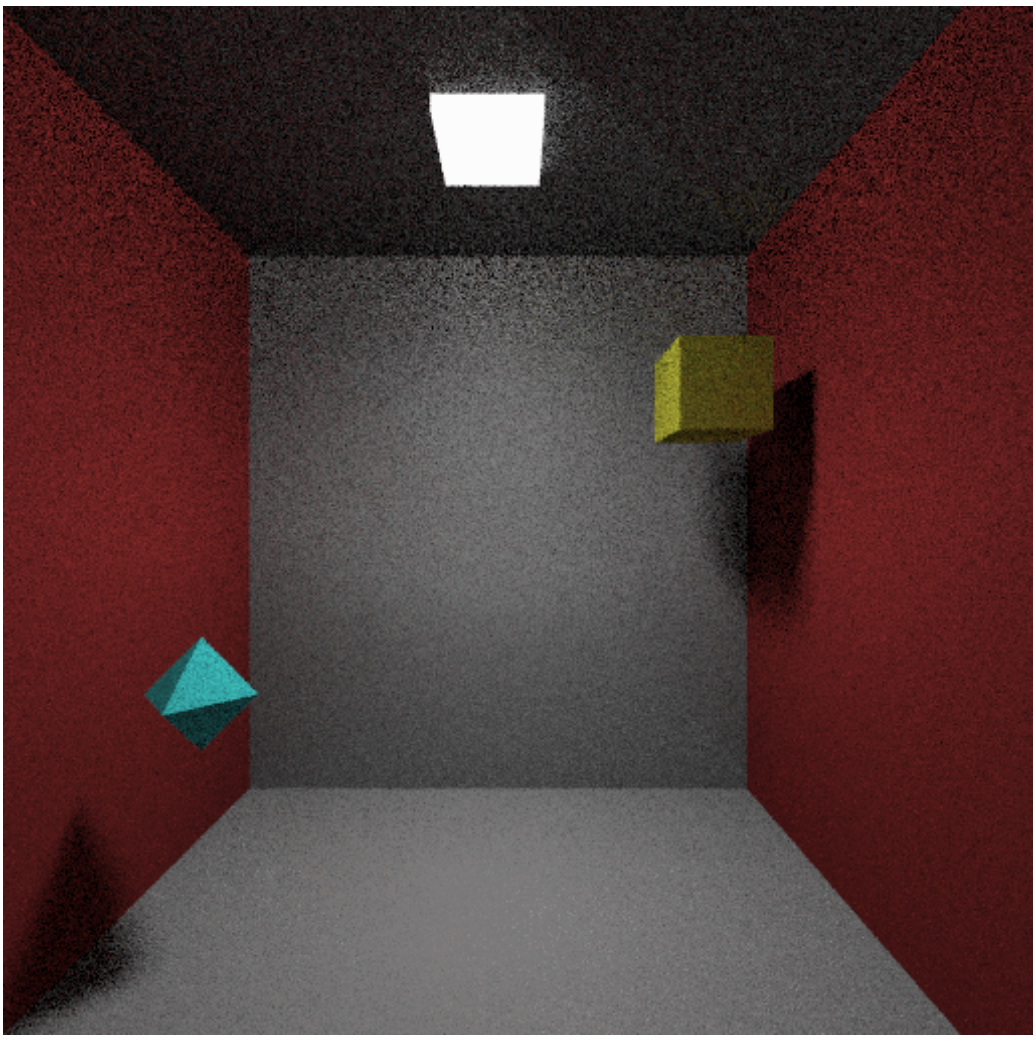

Figure 6.1: Render time: 299 seconds with i5 2500k @ 3.7GHz, 512x512, 10 rays/pixel, 1 shadow ray

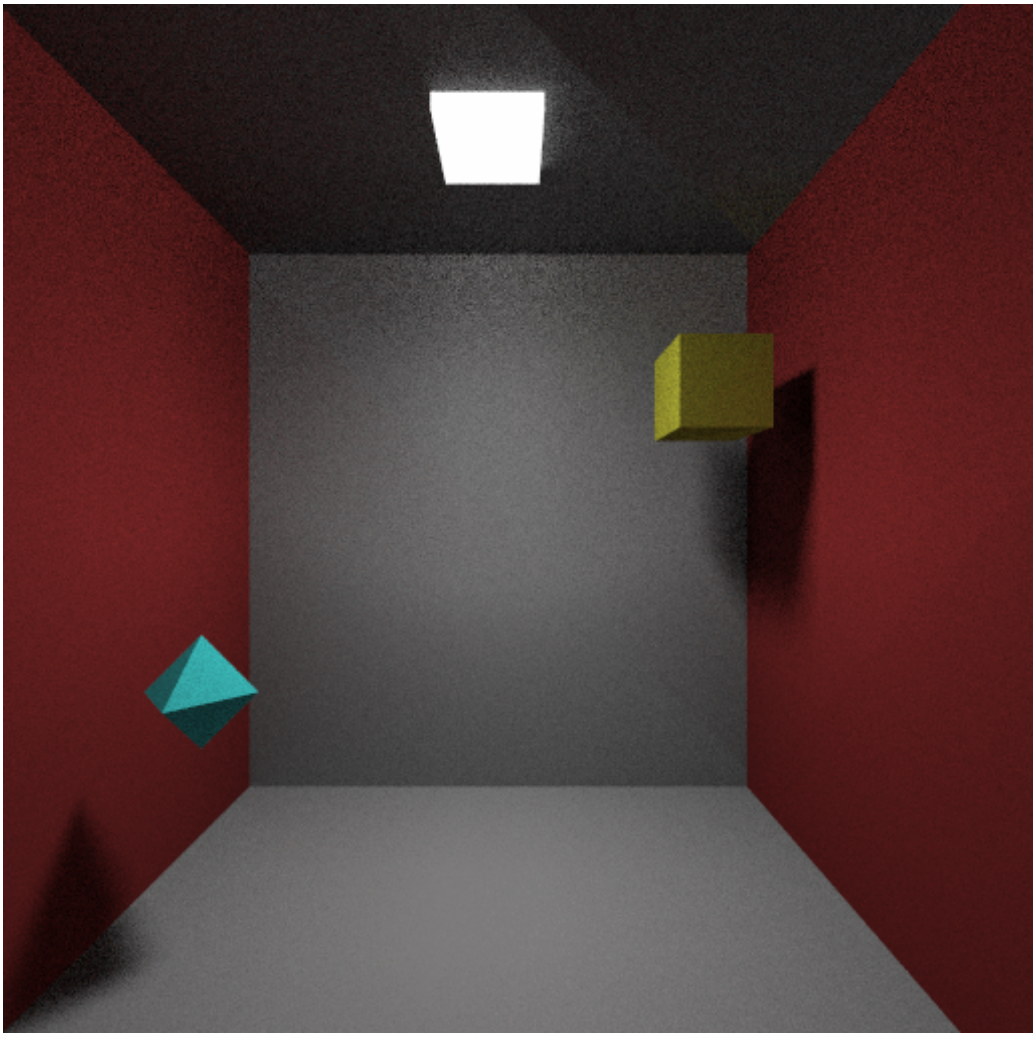

Figure 6.2: Render time: 26 minutes and 24 seconds with i5 2500k @ 3.7 GHz, 512x512, 50 rays/pixel, 1 shadow ray

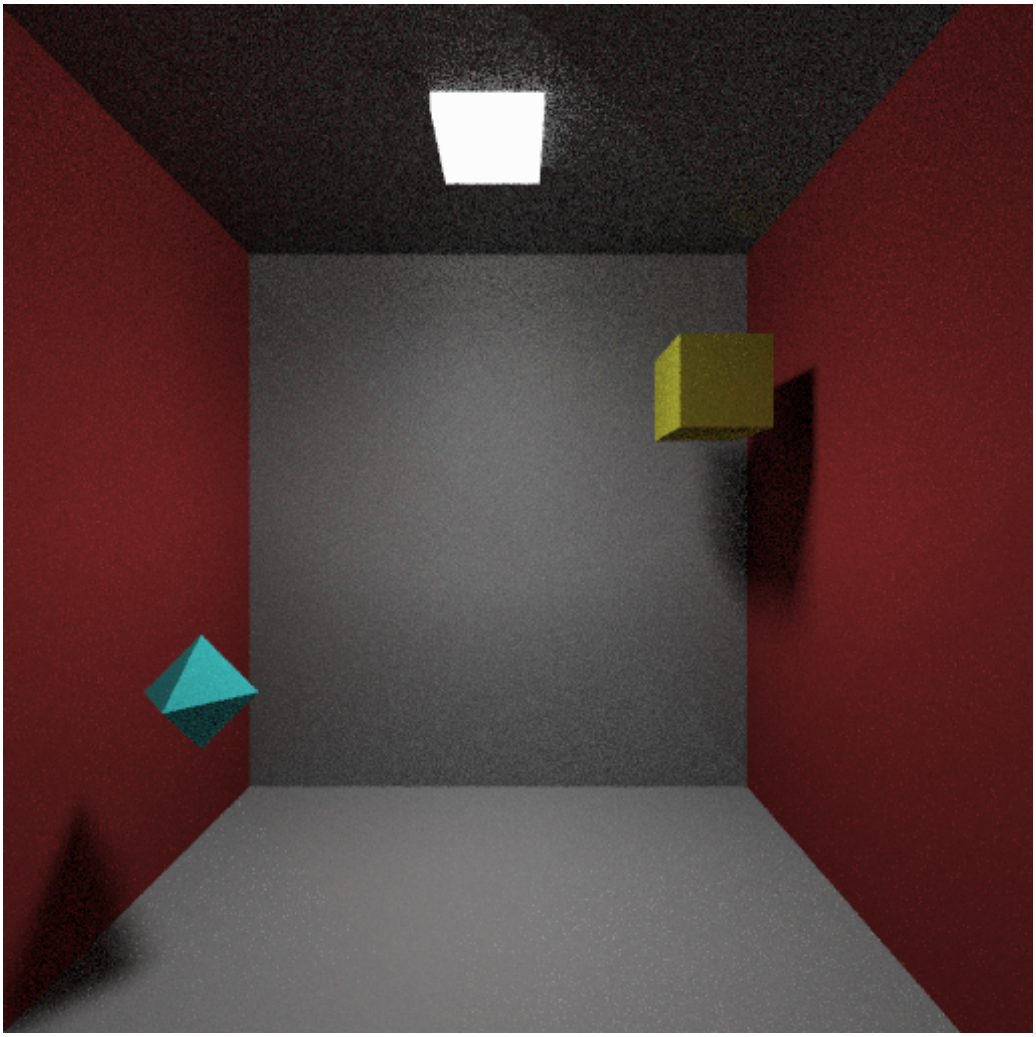

Figure 6.3: Render time: 22 minutes and 47 seconds with i5 2500k @ 3.7 GHz, 512x512, 10 rays/pixel, 10 shadow rays

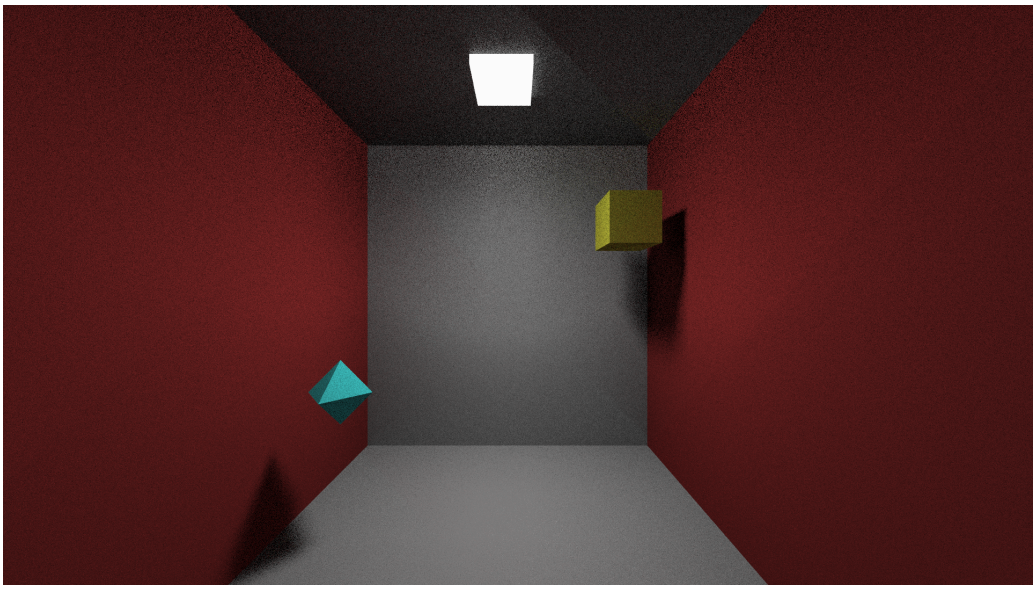

Figure 6.4: Render time: 47 minutes and 48 seconds with i5 2500k @ 3.7 GHz, 1920x1080, 10 rays/pixel, 1 shadow ray

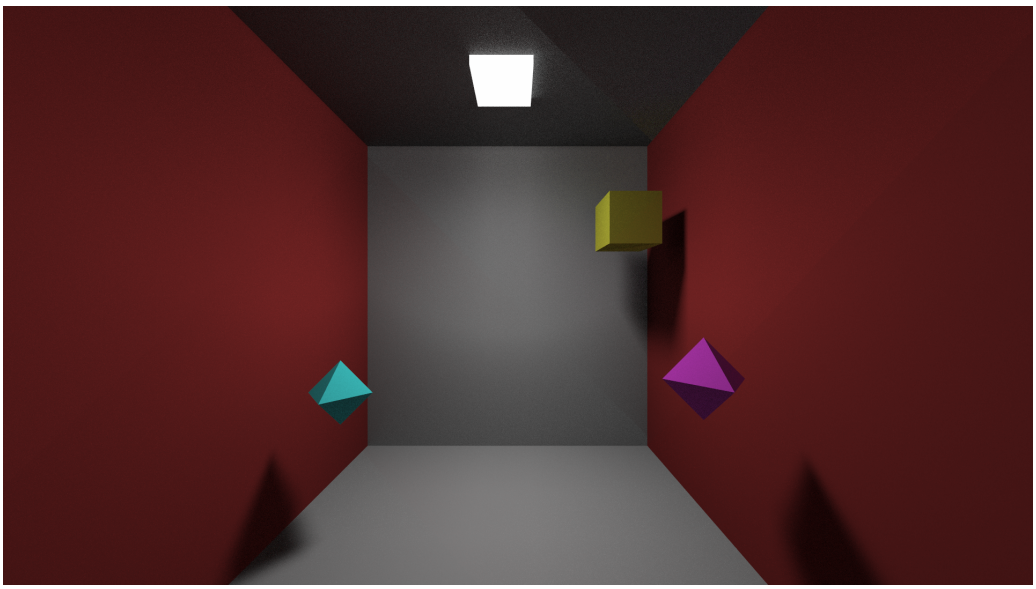

Figure 6.5: Render time: 13 hours and 24 minutes with Intel Xeon CPU E5-1620 @ 3.6 GHz, 1920x1080, 40 rays/pixel, 10 shadow rays

Chapter 7

Analysis and Discussion

The resulting renderer is a good basis for global illumination rendering. It supports general polygonal models, it has texture support and the code has an understandable structure.

The choice to use exclusively polygonal objects and the Möller-Trumbore algorithm has made the code more elegant and usable. As previously mentioned these kind of models and algorithms are also necessary if a diverse and life-like scene is wanted. It could however be argued that algorithms for ray intersections with analytical functions for spheres and other shapes that are easily described by analytical functions should be implemented. Some performance could be gained by using analytical functions instead of polygonal models for these shapes. To render a polygonal sphere with the projects current code that looks as good as an analytical sphere would require a very large amount if triangles. This amount could be decreased by interpolating the normals over the triangle surface since it is mainly the static normals that would make the sphere look faceted. This is why support for vertex normals has been implemented in the project. Even with a reduced amount of triangles some performance would be gained. This would however make the code less readable and would require some if and else statements on several locations in the code.

In regards to geometry and rendering times there are some major changes to the structure that would greatly decrease rendering times. The first of these are bounding boxes. Bounding boxes would allow detailed objects to be dissmissed by a simple check instead of calculating intersections against all triangles in the objects even if they ray is far away from that object. The current code always checks all triangles of all objects in the scene. The next step would be to use some kind of geometric sorting algorithm so that even if the check against the bounding box was not enough a large amount of the triangles in the model could still be dismissed. The sorting algorithm that will be used for this project is the Octree [6] algorithm. These two things have yet to be implemented.

Currently there is also some kind of directional bias in the image. The triangles of the objects can sometimes be seen clearly in how the diffuse light scatters. This must somehow stem from how the direction of the reflected rays are calculated. We have tried different methods for generating this direction. One method was described in section 5.2, another method was similar to this method but instead of using a vector from the triangle an arbitrary vector that was orthogonal to the surface normal was generated. A third method that was tried was to use the two random angles and transform these to cartesian coordinates. This cartesian coordinate vector was then transformed to world coordinates by a basis transform created by the surface normal and its two orthogonal vectors. All methods had similar results. The current hypothesis is that there is something lacking in the random number generator method that is used. Either in its seed or its distribution.

Several key features are planned to be implemented into this project. The first thing being transparent and perfectly reflecting objects as well as the oren nayar reflectance model. These wont require any conceptual changes in the used rendering algorithm or even much work but could still greatly improve

the visual result.

The second planned feature is rendering objects with textures. This would allow more interesting objects and be handy in some computational tasks. It would also allow the BRDF of an object and its emissive properties to change over the surface. Features such as normal and parallax mapping are also interesting prospects.

The third feature is photon mapping. This is perhaps the feature that will require the largest conceptual change of the rendering algorithm. Still it is more like added functionality than it is a replacement of the raytracing model.

The fourth feature has already been mentioned and it is that of the bounding boxes and sorting algorithms.

The fifth and final feature is CPU threading. Most CPUs these days have several cores. Currently the project renders the entire image on a single thread of the CPU. If different parts of the image was rendered by different CPU threads the rendering time would decrease by a factor proportional to the amount of threads that are available on the CPU.

Chapter 8

Conclusion

The result of this project is a good foundation for an advanced and diverse global illumination renderer. The renderer currently render objects exclusively with the lambertian model but all attributes and parameters needed for more advanced lighting models are implemented. It supports rendering of textured and arbitrary shapes. The rendering algorithms are not dependant on the objects being rendered. Some sorting algorithms and functions could greatly improve the performance of the renderer.

Bibliography

[1] Mark Eric Dieckmann. *Radiosity method*. 2015.

http://staffwww.itn.liu.se/~mardi/WebPages/Courses/TNCG15/Lecture3New.pdf

[2] Thomas Nikodym. *Ray tracing algorithm*. 2010.

https://dip.felk.cvut.cz/browse/pdfcache/nikodtom_2010bach.pdf

[3] Mark Eric Dieckmann. *Monte Carlo integration*. 2015.

http://staffwww.itn.liu.se/~mardi/WebPages/Courses/TNCG15/Lecture6New.pdf

[4] Henrik Wann Jensen. *Global Illumination using Photon Maps*. 1996.

http://graphics.ucsd.edu/~henrik/papers/photon_map/global_illumination_using_photon_maps_egwr96.pdf

[5] Thomas Möller. *Minimum Storage Ray/Triangle Intersection*. 2003.

http://www.cs.virginia.edu/~gfx/Courses/2003/ImageSynthesis/papers/Acceleration/Fast%20MinimumStorage%20RayTriangle%20Intersection.pdf

[6] Meagher, Donald (October 1980). *Octree Encoding: A New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer*. Rensselaer Polytechnic Institute