SDU
DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE

71B: Rendering 3D-Graphics with Ray-tracing

Jørn Guldberg : jogul16@student.sdu.dk
David A. H. Kaan : dakaa16@student.sdu.dk
Gabriel Howard Jadderson : gajad16@student.sdu.dk
Oliver Sønderskov Zarp : olzar16@student.sdu.dk
Frederik Wexø : frwex16@student.sdu.dk

University of Southern Denmark
Department of Mathematics and Computer Science

FF501
Supervisor: Simon Larsen
September 10, 2017

1 Abstract

Ray-tracing is a method for rendering 3D computer graphics. It renders a 3D world into a 2D image. A ray-tracer is sending a ray for every pixel in the image, and calculates the colour and light settings for the pixel. Calculations are based on intersections with objects in the 3D world.
We are going to implement a simple ray-tracer in Java and use Vector math to calculate the world scene. Our world contains Spheres, Triangles, Disks and Planes. These all derive from the same interface and implement their own method of intersecting. The world is split up into bounding volumes to achieve a faster time complexity. Additionally, we will investigate how ray-tracing is different to other methods of 3D graphic rendering, and the advantages of using bounding boxes.
We can render images with our ray-tracer and we have proved that it runs faster with bounding volumes implemented. We've also managed to implement simple concurrency techniques to achieve a significant decrease in run-time. Additionally, we've implemented a file parser that parses .obj-files and translates this text into objects and sets their respective size, colour and position in the world.

Contents

List of Figures

2 Introduction

Several methods are used for rendering 3D computer graphics. Some are good for live action, such as computer games, and some are good for high quality imagery.

We want to take a closer look at two of these methods. The first method is Rasterization, which is used for computer games and live graphics. We will compare Rasterization to the ray-tracer method. The ray-tracer method is a method for rendering high quality images.

We want to implement a simple ray-tracer, find the difference between it and Rasterization. We want to optimize the time complexity for our ray-tracer with bounding volumes.

Our problem formulation:

> We are going to implement a simple ray-tracer and look at time complexity, and how this can be improved by utilizing bounding volumes. In addition to this, we want to see how data structures can come in handy for this purpose. We will implement light and shading to gain an understanding of the calculations in a ray-tracer, and why a ray-tracer is time consuming and then compare it to the rasterization method.

Rasterization is a method for rendering 3D graphics in live action. It is used for computer games and have a fast calculation time. This comes with a cost in terms of image quality. Because a game has to run smooth, the images shown on screen has to be calculated very fast and in rapid succession to one another. This means that there is no time to calculate a high-quality image.

Ray-tracing is another method where a ray is sent into a 3D space for every pixel in the image [1]. For every ray sent, the ray-tracer calculates what its intersects, or if its intersects at all. Light sources and reflection can be added. All these calculations take time and therefore this technique is used for high-quality images.

Bounding volumes is a technique used to improve the time complexity. Imagine an animal made of 1,000 triangles. Instead of testing intersection for 1,000 triangles for every pixel, we can put it in a bounding volume. Now the ray-tracer only has to check if it hits the bounding volume. If it does hit the bounding volume, then it checks for intersections with the 1,000 triangles inside the volume. Even the objects of triangles can be split up into bounding volumes. This process can be repeated recursively to a bounding volume box with a fair amount of triangles in each.

In the first section, "Theory", we have described the basic theory for these two methods to gain a general understanding of how the theory behind these methods work. In the section "Implementation and materials" we have described the methods used in our implementation. We have used some obj-files. These files stores information for the geometry of complex objects.

We have conducted some tests on time complexity for our ray-tracer, and have discussed these in the "discussion" section, where we also discuss how we have applied the theory, and how we could improve our ray-tracer.

We follow up on our problem formulation and conclude on this project in the "Conclusion". Our source code is available on github: `https://github.com/Rhodez-x/rayTracer`

3 Theory

This section explores the theory behind ray-tracing and different techniques that improve the produced image in regard to realism. Lastly, ray-tracing is compared to rasterization.

Ray-tracing is a method for computing realistic 3D images, using vectors, rays and defined shapes. As the goal is to create 3D images, 3 dimensional vectors are used to describe a point in space. A ray, is in theory an infinite line defined by two vectors, which respectively determines the origin and the direction of the ray. In order to produce a 3D image, a ray is created for each pixel of the image. Each of the rays have the same origin, but their direction is based on the pixel in question.

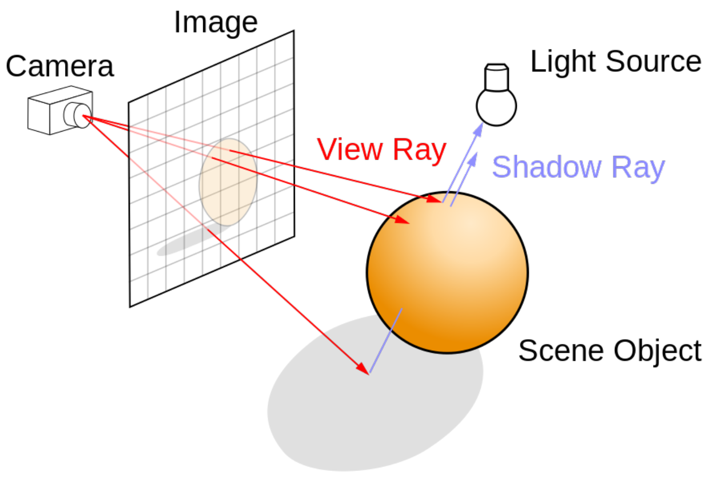

Figure 1: A visualization of the ray-tracing method [5]

As seen on figure 1, a good way to imagine ray-tracing is by seeing origin as a camera. Then in front of the camera is a grid, where each square in the grid represents a pixel. For each square on the grid, a ray is traced from the camera, through the square in question. If the ray intersects with a shape, the pixel that the square represents gets the colour of the shape. This is why ray-tracing is called image-centric - meaning that it starts from the image.
A shape, in this regard, is defined by one or more vectors, and sometimes a number, as to for example describe the radius of a sphere. As multiple shapes can appear, one needs to check for each shape, whether the current ray intersects with it.

The visibility problem is determining whether a part of a shape is visible from a specific viewpoint. This problem is solved by the method described above. This means, that we, in theory, can almost create a 3D-image. The only problem left, is if one ray intersects with more than one shape. Of course, the current pixel should get its colour from the shape that is closest to the camera. This means that the distance from the camera (origin) to the point of intersection with each shape, needs to be calculated.[10][1]

3.1 Camera

In computer graphics, a camera is typically composed of an eye position (our camera's position in the world), a direction, in which the eye is pointing, and two values determining the size of our view port, namely width & height. In contrary to how a real camera works, a ray tracer's camera work by shooting rays out, instead of taking them in. To understand how a camera works in ray tracing, we must first understand how they work in the real world. The concept 'pinhole camera' is typically used in the real world. A pinhole camera consists of a box, where there is a tiny hole in the middle of one side of the box, on the opposing side, we have a canvas. The tiny hole allows rays of light to enter the box, and capture the scene on the canvas. That is a very basic explanation of how a real camera works, and now to get a ray tracing camera, we can reverse this process and move the canvas out in front of the tiny hole, and call this our eye instead. Now, what we want to do is shoot a ray from our eye, through the canvas, and into the scene. If our Ray then intersects an object, we know that this point on the canvas represents a part of an object, and we can then proceed to apply the object's colour to this point. This process is repeated by iteratively looping through every pixel in the canvas, whose size is made up of width & height.[14]

3.2 Ray

We present a Ray in a 3D-space with two points. An origin vector denoting the position of the ray in space and a directional vector, denoting the direction of the ray. Together the origin and the directional vector can be represented as a line in space with a set of points defined as $P(t) = O + D * t$ where O is the origin vector for the Ray, D is the directional vector for the ray, and where t contains all the points along the ray line and $t > 0$. Note also that the directional vector is normalised in our purposes, this simplifies things for our shading and removes the "Shadow acne" problem. Normalising the directional vector will also simplify our intersection testing in terms of world coordinates. From further on we will reference the above equation as the ray's equation.

3.3 Shapes and intersections

This section will explain the advantages of defining different shapes and describe how to define spheres, planes, disks and triangles in 3D-space, as well as how to determine the intersection with a ray for each of these shapes.

A lot of shapes can be described in 3D space. All shapes can be represented by multiple triangles - thus triangles are the most important shape to have implemented into a ray-tracer if one's goal is to create arbitrary shapes. However, to create a sphere with triangles, one would need a lot of triangles and the sphere would still appear with edges. A lot of triangles means a lot of intersection tests, which leads to a high time complexity. An alternative to creating a sphere with triangles is to define a sphere within the 3D space. Not only would this make a round sphere with no edges, but there will also only be one intersection test to make. To further emphasise the advantage of implementing a sphere; a ray-sphere intersection is easier to calculate than a single ray-triangle intersection. A shape implemented like this is called an implicit shape.

3.3.1 Sphere intersection

A sphere in a 3D space is described with a point in the space and a radius.

A ray can be described by the parametric equation: $O + t \cdot D$, where O is the origin of the ray, t is a constant and D is the direction of the ray. This equation describes a vector, where the constant t, determines where along the ray, this vector should be. A sphere can be described with the following implicit function of a sphere equation: $|P - C|^2 - R^2 = 0$, where P is a point on the sphere's surface, C is the centre of the sphere and R is the radius. In order to check if the ray and the sphere intersects, we insert the parametric equation of the ray, instead of P in the equation of the sphere: $|O + t \cdot D - C|^2 - R^2 = 0$. When developed, this equation is a quadratic function, t being the unknown constant. By calculating the discriminant it can be determined if the ray intersects the sphere once, twice or not at all. If there are two points of intersection, the point of which t is lowest while also being positive would be the point closest to the ray's origin and thus the point considered the real intersection point. Finding the real intersection point is important when it comes to shading, as the normal of the intersection point is used (see Shading - section 3.3).[10]

3.3.2 Plane Intersection

A plane is defined with the following constants *A, B, C, and D* and the plane equation is:

$$A * x + B * y + C * z + D = 0$$

The normal vector for a plane is defined as

$$\vec{N} = (A, B, C)$$

Here x,y,z denotes A,B,C in the normal vector. The intersection distance t can be found by substituting the rays equation into the plane equation

$$A * (X_0 + X_d * t) + B * (Y_O + Y_d * t) + C * (Z_0 + Z_d * t) + D = 0$$

where $[X,Y,Z]_O$ is the ray origin and $[X,Y,Z]_d$ is the ray's direction. [1]
if we isolate t we get the following:

$$t = \frac{-(\vec{N} \cdot \vec{R_O} + D)}{\vec{N} \cdot \vec{R_d}}$$

above we've solved for t and used vectors instead of vector coordinates.
An eficient way to calculate t is to first calculate the normal vector and dot the normal vector with the ray direction. That is the denominator first in the above equation and if the denominator is greater than zero then further calculate the numerator. [1]

3.3.3 Disk intersection

A disk is defined by a position, a radius, and a plane in which it exists. For the most part, a disk is an abstraction of a plane, in that it utilises the functions and math previously defined in our plane object to determine whether or not it has been hit by a ray. When testing for intersections, what we really do is check if it hit the locally defined plane in the disk, and if it did, how far the hit was from the circle's origin, if this is greater than the disk's radius, the ray didn't hit our disk, if it is smaller or equal, then the hit is valid. In a way, the local plane serves as a bounding box for the disk.[13]

3.3.4 Triangle intersection

A triangle is described by three vectors. When performing a ray-triangle intersection test, the first thing to do is finding the triangle's normal, which is the same as finding a plane's normal, as the triangle is on the plane. Let A, B and C be the three vectors describing a triangle, then normal is found by calculating the cross product of AB and

AC.
Knowing the plane's normal, we can write out the equation of a plane: $N \bullet P + d = 0$, where N is the plane's normal, P is a point on the plane and d is the distance from the ray's origin to the plane, traced parallel to the to the plane's normal. Following the plane's equation, we can find d, by having the plane's normal and a point in the plane. The normal is calculated and we already have three points, that we know is on the plane; A, B and C. Thus we get: $d = N \bullet A$, (any of the points A, B and C could have been used). Knowing d, we can insert the ray's parametric equation, in the plane's equation instead of P: $N \bullet (O + t \cdot D) + d = 0$. Isolating t we get:

$$t = -\frac{N \bullet O + d}{N \bullet D}, where\ N \bullet D \neq 0$$

Note that $N \bullet D$ will only be 0, if the ray and the plane are parallel. If t is less than 0, the ray intersects the plane, behind the origin, which means, that the plane can't be seen. However, if t is positive, the plane is visible and there is an intersection. But this doesn't mean, that there is an intersection between the ray and the triangle.
Having t, we can calculate the intersection point between the ray and the plane, using the ray's parametric equation. In order to check whether the intersection point is inside the triangle, we will perform something called an "inside-outside test".This test checks for each vector defining the triangle, if the intersection point is on the left side. This needs to be true for all edges before we can conclude that the ray and the triangle do intersect. Let A, B and C be the three points of the triangle, then we find the edge AB by $B - A$. Let P be the intersection point, then we find AP by $P - A$. If the result of the equation; $N \bullet (AB \times AP)$, is positive, the intersection point, is to the left from the point A. If the intersection point is to the left from all the triangle's three points, it can be concluded, that the ray is intersecting the triangle. [12]

When implementing an intersection test, the order of tests is important. If a shape is behind the camera, we don't want to start calculating a bunch of numbers. We would rather conclude right in the beginning, that the ray does not intersect with the sphere, as the sphere is behind the camera, before performing the more complex calculations. A lot of optimisation is needed, as the intersection test is likely to be computed well over a million times.

3.3.5 Lighting & Shading

The point light implementation defines a singular position in the world from which the light will originate, and an ambience that determines the strength of the light. Lighting & shading work in cohesion, as the light is used to calculate to what degree the object's colour is lit. [6]
The shading used throughout the ray-tracer is Lambertian shading[7] which creates a diffuse or matte surface. Lambertian shading allows all the triangles to reflect light equally in all directions, this is achieved by taking the dot product of a surface normal vector and the light position in space. Note that the light position is normalised as well as the normal vector, this will address the Shadow acne problem described earlier and create a smoother shading.

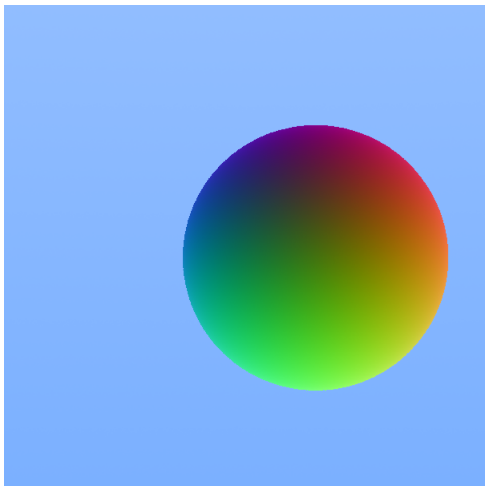

Figure 2: The following is a sphere shaded with it's calculated normal vector

Recall the equation of the ray given by $P(t) = O + D * t$ if intersection occurs then t denotes the lowest intersection distance. In other words, the smallest absolute root of the intersection. The Intersection method always returns the intersection point calculated from the rays equation, consecutively, the normal vector can easily be calculated for each shape once the intersection point has been found. Once the surface normal vector is known the calculation is done by taking the dot product of the normal vector on the light position. The dot product can be rewritten as:

$$L \cdot N = |N||L|cos\alpha$$

where α is the angle between the surface normal and the light position. As given the equation for the dot product we can observe that when the angle is 0 the dot product will be 1 and thus giving us a full shading directly, which means that if the light originates from our eye then the entire Lambertian surface will be lit, at a 90 degree angle.

3.4 Bounding Volumes

Bounding volumes are used for improving time complexity for ray-tracing. The idea is to encapsulate objects that consist of many shapes. If an object consist of 10,000 triangles, it will take a long time to calculate intersections for every triangle for every pixel. If we instead encapsulate the object in a bounding volume we only have one intersection test for every pixel. We only check triangle intersection if we intersect the bounding volume If we don't hit the volume we go on to the next, pixel.

The closer the volumes are to the object inside, the more optimal it is. Even objects can be in different bounding volumes.

If we intersect the volumes with 10,000 triangles inside, we have to make 10,000 intersection test every time. If the object was in four different bounding volumes inside the volume, we test which of these four we intersect, and inside here we "only" have to test for example 2,500 triangles.

In fact, the entire screen can be split up into bounding volumes. Imagine the screen split into four squares, and the squares were split up into another four. This way we could make a quad-tree, and bind objects to a leaf on the tree. [6]

Even though this technique improves the time complexity a lot, it can't be considered a "fast" algorithm like Rasterization.

3.5 Rasterization

Rasterization is another method for projecting 3D-scenes to a 2D-image. It differs from ray-tracing, by being object-centric instead of image-centric. The algorithm being object-centric means that it starts from the objects in the 3D-scene, and traces back to the image, which is the opposite of image-centric. This section will look at the differences between ray-tracing and rasterization and investigate if one of the methods could be considered obsolete.

All the geometry in a scene should be triangles when using rasterization, as it simplifies the process and makes for better time complexity. Each vertex of the triangles are projected onto the screen and for each pixel of the image, it is determined if it's part of a 2D-triangle, based on the projection. However, as with ray-tracing, some computation can be saved, by making the problem smaller. For rasterization, this is usually done by using 2D bounding volumes on the screen, where the triangle is projected. This limits the search of pixels that is within the triangle, to the pixels in the 2D bounding volume. Furthermore, some triangles can be ignored by the renderer, as triangles that can't be seen or are facing away from the camera does not need to be rendered at all.[11][1]

Rasterization is a lot faster than ray-tracing and it uses less memory. Because of this, rasterization is the algorithm used in modern GPUs, as it works better for real-time graphics. As opposed to the ray-tracing algorithm, rasterization does not need to check for intersections between rays and objects on the scene, which is a costly operation. Furthermore, as rasterization can disregard certain triangles, it gets an even greater advantage in time complexity, but also in saving memory. However, ray-tracing has its own advantage over rasterization, which means that none of them are obsolete. Rasterization is inefficient when it comes to determining the visibility between two points. This task is important for shading, and ray tracing is, as described in the *Shading* section, very good at this. Thus ray-tracing has an advantage when it comes to shading, but rasterization has an advantage at solving the visibility problem.[10]

4 Implementation and materials

This section will go over our implementation of a ray-tracer and the materials we have used.

4.1 Implementation

We have implemented a simple ray-tracer in Java. This ray-tracer supports spheres, planes, disks and triangles as shapes. A proper viewport has been implemented, as well as light and shading. Bounding volumes has also been implemented in the form of spheres. Our implementation is inspired by the C++ code from [10], and from [1].

4.1.1 Basic ray-tracer

All of our raytracer's calculations are rooted from a for loop, which is running as many times, as the output image is high. Inside this loop, is another for loop that runs as many times, as the output image is wide. This means that we iterate over every pixel in the image, while always knowing the x- and y-coordinate of the current pixel. Inside the inner for loop, a new ray is created. Its direction is calculated by the camera, based on the x- and y-coordinates. Then an intersection test between each bounding volume is performed. If the ray does not intersect with a bounding volume, the current pixel is set to be a calculated variant of a predetermined background colour. For each bounding volume, the ray intersects with, each of the shapes, contained within the bounding volume, is iterated over and tested for intersections. If no shapes are hit, the background pixel is set to the background colour, in the same way, described above. If one or more shapes are hit, information about the ray-shape intersection, closest to the ray origin, is returned, in order to calculate the colour to write to the pixel, based on lighting and shading.

4.1.2 Shapes and intersections

In order to implement shapes in our ray-tracer, we have implemented an interface with an intersect function. This function returns a *rayInfo*, which is an object holding the following information gathered from the intersection: Whether there was an intersection, the intersection point, the point's normal, the distance between origin and the point and the material of the shape.
The four different shapes that we have implemented (spheres, planes, disks and triangles), are each defined in their own class, implementing the shape interface.

The sphere class defines a sphere by a vector, a radius and a material. The way the ray-sphere intersection is calculated is not much different, from the method described in section 3.3. However, there are a few deviations, which is where the focus will be.
First, we define the constants of the quadratic equation, derived from the sphere's equation, with the ray's equation inserted as a point along the surface of the sphere. The discriminant is calculated, using these constants. With this value, we can determine, if there are none, one or two intersections points, between the ray and the sphere. If the discriminant is negative, there are no intersection points. Thus, if this is the case, a *rayInfo* is returned, saying that there were no intersection. If the discriminant is exactly zero or above, the quadratic equation is solved. By doing this, we get the distance from the intersection point/points. If there are two intersection points (discriminant is positive), the smallest - none-negative - distance is used to get the intersection point. Otherwise, we just get the one intersection point. Also; a *rayInfo* is returned, containing information

about the intersection test. All this is simply based on the theory of a ray-sphere intersection - however, one thing needs to be changed in order for the Java implementation to work every time. When solving the quadratic equation when there are two intersection points, the following equations are normally used, to find the intersection points:

$$\frac{-b + \sqrt{\Delta}}{2a} \quad and \quad \frac{-b - \sqrt{\Delta}}{2a}$$

However, these equations create a problem, as a computer won't be able to represent the decimal numbers accurately enough, which could rise problems with certain numbers when calculating the square root. Thus a workaround is created, where we compute a variable q:

$$q = -\frac{1}{2}(b + sign(b)\sqrt{b^2 - 4ac}), \quad where\ sign = \begin{cases} -1 & \text{if } b < 0 \\ 1 & \text{if } b \geq 0 \end{cases}$$

This ensures that there won't be a mistake, when calculating the square root. In order to get the intersection points, the following simple equations are needed:

$$Intersection_0 = \frac{q}{a} \quad and \quad Intersection_1 = \frac{c}{q}$$

Thus the ray-sphere intersection is computed correctly.

The plane class

The disk class defines a disk simply by a point and radius. It is similar to the Sphere shape in this regard, however, there are notable differences. The main difference being the lack of 3D 'feel' with the disk. Despite the lack of this, a disk is very much a 3D object, looking at it from different origins and shifting it around the world proves this, as it will stretch on certain axis' and assume a more oval representation visually if it's surface isn't exactly parallel with the camera.
A prerequisite for the Disk's intersection method to work is the Plane. Without the plane, our Disk implementation will not work. This is because the Disk defines a local plane that'll be invisible to the rendered image. It utilises this plane to check whether or not it has been hit. The process is relatively simple, once a ray intersects with the plane, the hit position and the disk's origin are subtracted with each other and is then checked against the disk's radius, if the difference is larger than this, no intersection with the disk has occurred, if it is equal or smaller, the disk has been successfully intersected.[13]

The triangle class defines a triangle by an array of three vectors and a material. The ray-triangle intersection, is exactly the same as the ray-plane intersection, in the beginning. This is because the triangle can describe a plane. By checking the ray-plane intersection - where the plane is described by the triangle - we can easily discard the triangle early on, by determining that the triangle is behind the ray or that it is parallel with the ray. If the intersection test gets past these checks, we can compute the intersection point of the ray and the plane.
When we have the ray-plane intersection point, we know that the ray intersects with the plane, described by the triangle. Now, all we need to figure out is if the intersection point is within the triangle. This is where this intersection test really differs from the ray-plane intersection. To determine whether the intersection point is within the triangle, we use the "inside-outside test", as described in section 3.3. This test is implemented exactly as described in the theory section. If the test returns false, the intersection returns a

rayInfo, stating that there were no intersection. Otherwise, a *rayInfo* is returned, stating that there was an intersection while returning the rest of the useful information gotten from the intersection test.

4.1.3 Lighting & Shading

Currently, our light implementation is a type of point light. Furthermore, our light consists only of a position in the world and a variable that defines its ambience, which is used to determine the strength of the light source. In addition to this, we're limited to a single light source as opposed to several points of light.
The equation is relatively simple and is only called once an intersection with an object has been confirmed. The ray that hit an object is passed to our shading function, along with a light source, we then dot the ray's normal with the light's position, and with the output, we can determine whether or not the position hit is visible to the light source, and to what extent it is visible. Followingly we calculate the ambience by taking the light's parameter 'ambience' and rounding it up so we won't be dividing by zero. We then multiply this value with the result from the previous calculation if it is greater than zero, otherwise, we multiply by zero and assume the light couldn't reach the point hit. Finally, we multiply the vector containing the albedo for the object's material, and the end result is a smooth looking Lambertian surface.

4.1.4 View port/camera

Our camera consists of a position in the world and a direction. By default, we've set these values to be (0, 0, 0) and (0, 0, -10) respectively. Given a normal right-handed coordinate system, X is left/right, Y is up/down and Z is back/forward. Because of the way this is set up, all objects in front of the camera (e.g. the direction our camera is facing) will be in -Z. Referring to what was stated in the theory about cameras, we have to define a rectangular canvas in front of our camera's origin (eye position), and can then through our cameras origin shoot out rays to each pixel of the canvas and check for intersections with objects.[14][1]

4.1.5 Bounding volume boxes

In our ray-tracer, we have created an object *boundingVol* which has a shape, and a list of shapes. The shape of a *boundingVol* can be any kind of shape in our ray-tracer. Since all of our shapes have an intersect method, our *boundingVol* can be checked for an intersection. The main loop calculates intersections over a list of *boundingVol's*, which contain all of our *boundingVol's* in the world.
The *boundingVol's* shape is defining the bounding volume. We have used our spheres as bounding volumes since these can encapsulate other shapes.
The *boundingVol's* shapes size is generally set manually in our ray-tracer, but if the object is loaded through our .obj-file loader function, it will automatically be set.
The loader remembers the outer marks for the object and calculates where the center of the object is. This will be the center of the bounding sphere. The loader then calculates the length of the vector from the center to the farthest point. The length of this vector is the radius of the bounding sphere. The radius is added 0.01 to be sure it does not touch the object, which could otherwise lead to the object being cut short on edges touching the border of the bounding sphere.

If two bounding volumes exists in one pixel, the loop will check which one contains the closest object. The bounding volume(s) with the closest object, will be the one in charge of rendering that pixel.

4.2 Materials

We have used some .obj-files to test rendering of more complex objects.[4] [2] [3]
To load these into our code we have implemented a simple .obj-file parser.
obj-files have three steps. Lines starting with "v" is a vertex, and have the points for its position in 3D-space. Lines starting with "vn" is normal vectors for all faces in the object. Lines starting with "f" is all the faces. It tells what vertices combined will give a face. These lines can have extra information about what texture the given face have, and what normal vector it has.
In our implementation, we have skipped all extra information and normal vectors. This means that we load all vertices into an array. We loop through all faces and create a new triangle for every "f"-line in the .obj-file.
All triangles are saved in a new array. The loader creates a bounding volume around the object as described in 4.1.5. The bounding volumes are added to the global list of bounding volumes and it's ready to be rendered.

5 Results

The results are generated on an Intel® Core$^{\text{TM}}$ i5-4590 CPU processor.
In testing time complexity with bounding volumes, two test scenarios are setup. The first with two grids containing 6X6 spheres is set up on a 1000X1000 pixels image. see figure 3 and 4.
The second scenario is a scene with three objects loaded from 3 different .obj-files. The bounding volumes for this test is calculated automatically by the .obj-file loader function in the ray-tracer. see figure 5 and 6.

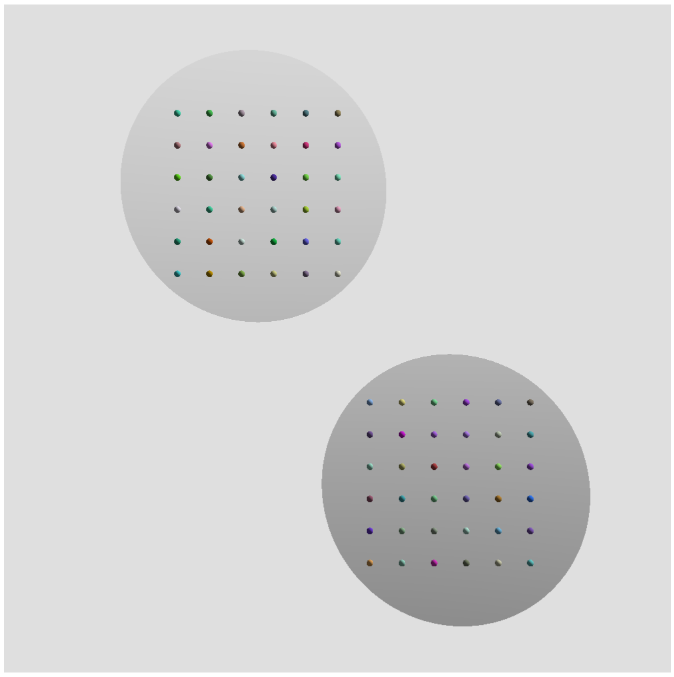

Figure 3: Illustrates what the bounding volumes encapsulates in test 1

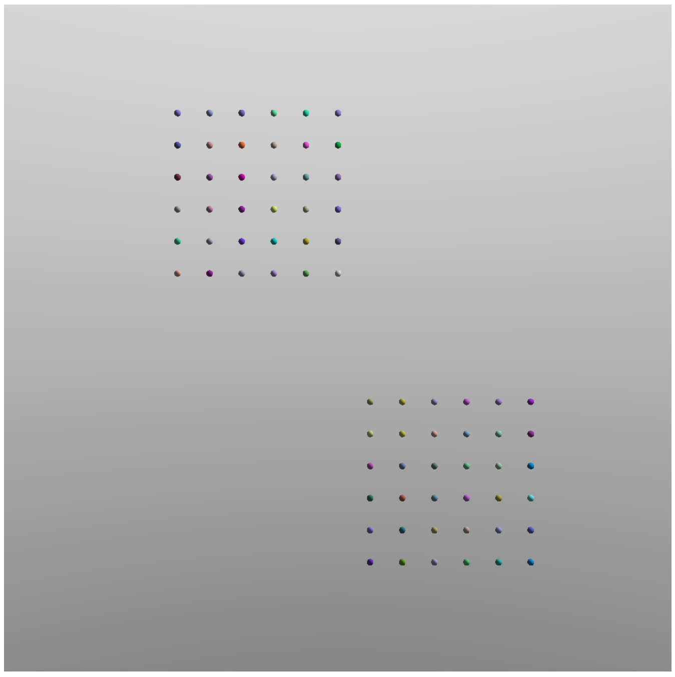

Figure 4: The test scenario where bounding volumes cannot be seen test 1

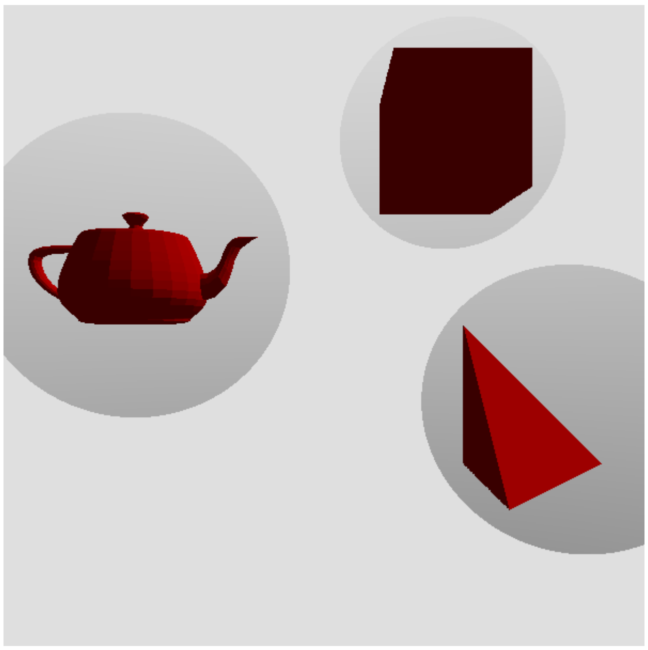
Figure 5: Illustrates what the bounding volumes encapsulates test 2

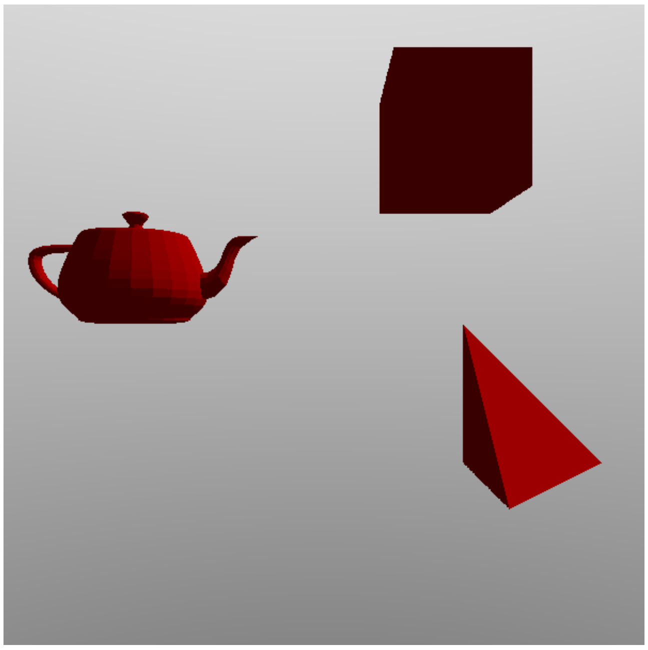
Figure 6: The test scenario where bounding volumes cannot be seen test 2

5.1 Time complexity

To ensure the program has a consistent runtime the first test is made 5 times. The results for the first test is in Table 1. Result for second test are in table 2.

Table 1: Results for time complexity for test 1

Time complexity (ms)		
test no	without bounding volumes	with bounding volumes
1	4822	920
2	4824	910
3	4850	912
4	4817	895
5	4804	899
avg	4823	907

Table 2: Results for time complexity test 2

Time complexity (s)		
test no	without bounding volumes	with bounding volumes
1	136.918	25.152

We have made a test for simple multi-threading for our ray-tracer. The test scenario was a 1000x1000 pixels image, with a wolf loaded from a .obj-file. Figure 12 is a 2000x2000 image of the wolf.

Table 3: Results for multi threading test

Time complexity (s)		
test no	without concurrency	concurrency
1	45.138	9.256

5.2 Numbers of intersects

In the test scenario we have 72 shapes to test for intersection.
The test scenario image has 1000x1000 pixels.
The overall numbers of intersects without bounding volumes is:

$$1000 * 1000 * 72 = 72,000,000 \tag{1}$$

With bounding volumes:
2 bounding volumes is tested for every pixel:

$$1000 * 1000 * 2 = 2,000,000 \tag{2}$$

A bounding volume is hit 256,242 times.
The total count for intersection test inside the bounding volumes is 9,224,712

the total number of intersection test is:

$$2,000,000 + 9,224,712 = 11,224,712 \tag{3}$$

5.3 Image quality progress

Figure 7: The very first "Hello world" example. It showed that our 3d world existed and in worked correctly because of intersection with the rays

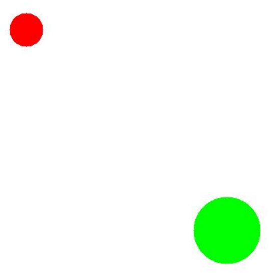

Figure 8: Second image is a bigger scale version of one. The image is 600x600 pixel, and confirmed the ray-tracer was working

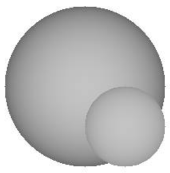

Figure 9: Colored by intersection distance

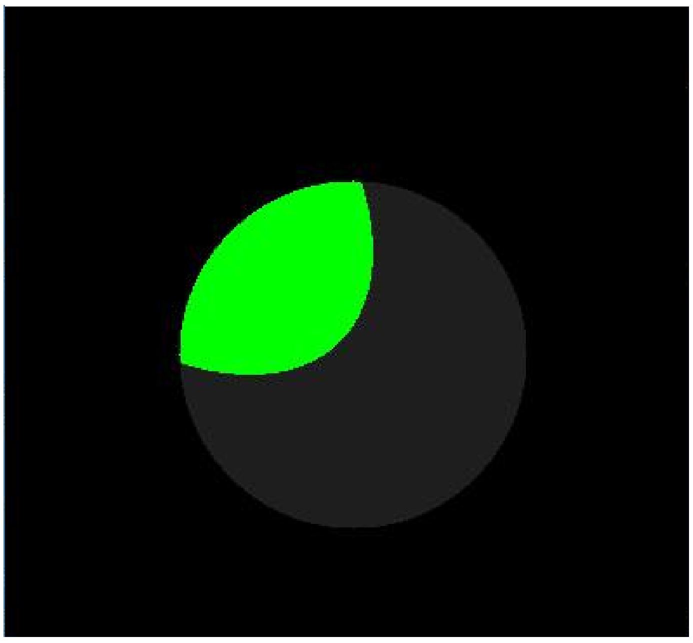

Figure 10: First implementation of light

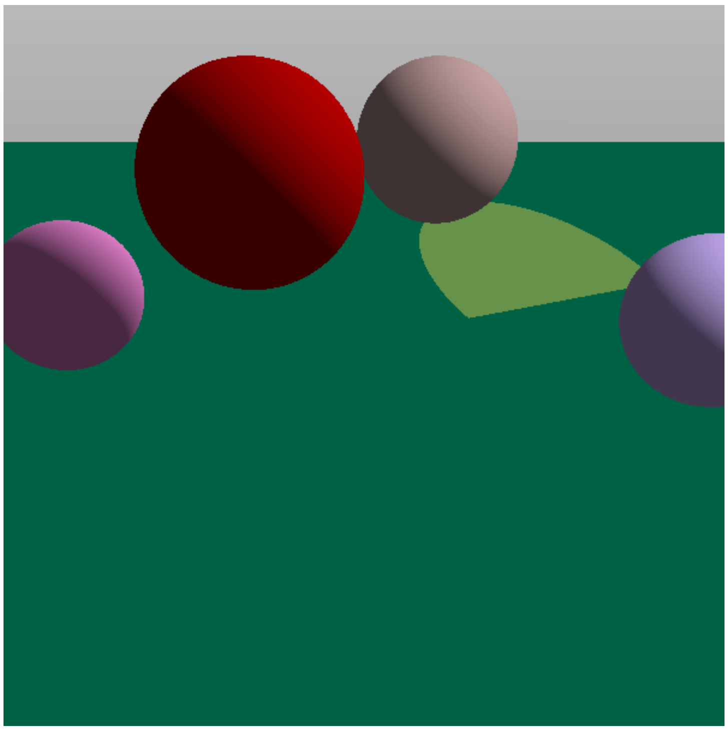

Figure 11: Plane, shading and disk object is added

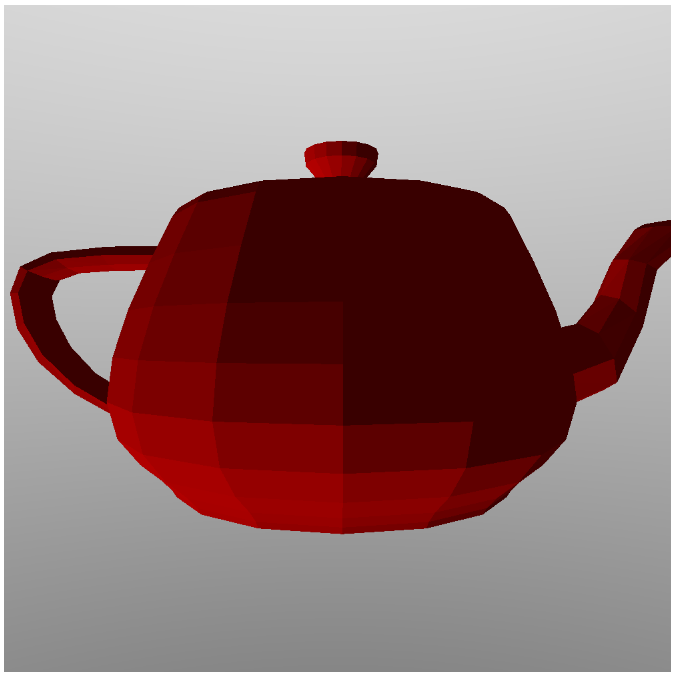

Figure 12: A load function for .obj-files is made and the ray-tracer can render the objects

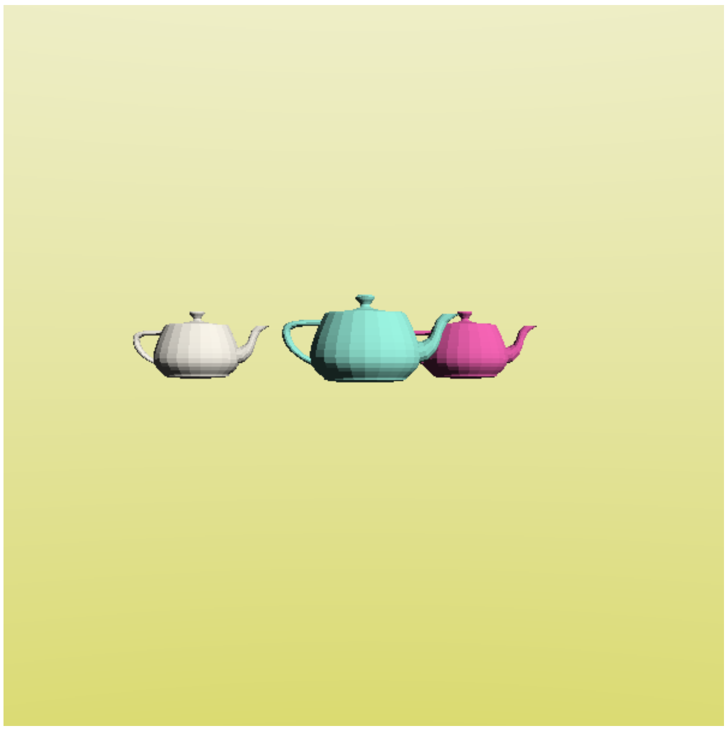

Figure 13: Bounding volumes painting closest object

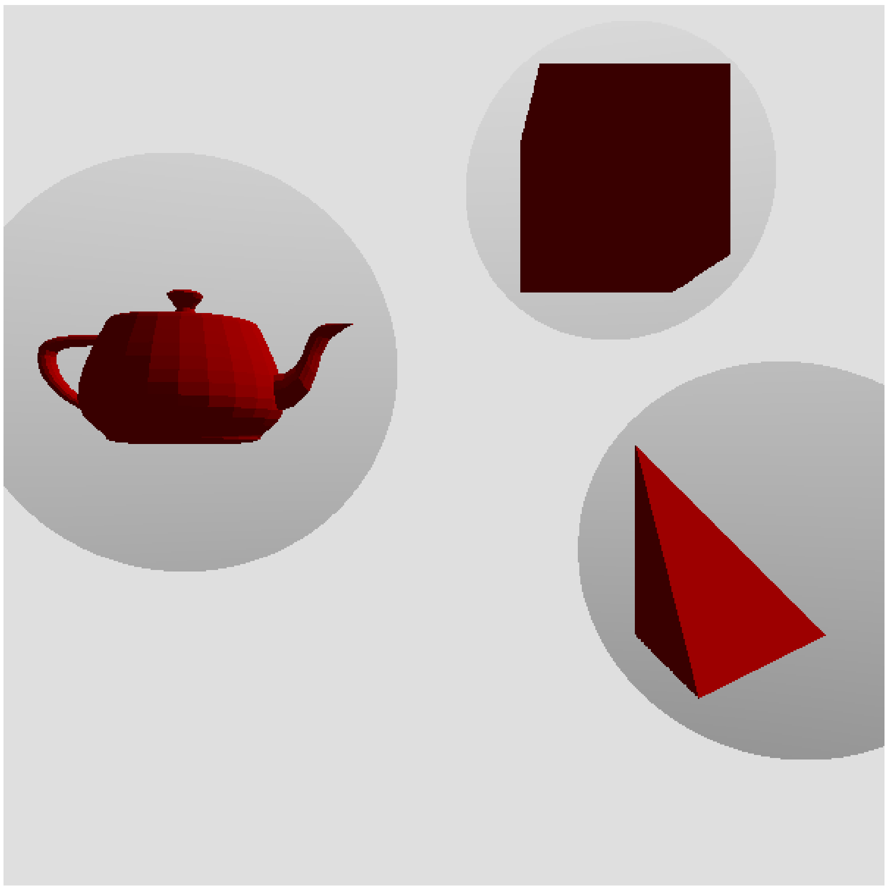

Figure 14: Automatic bounding volumes is set for every .obj object.

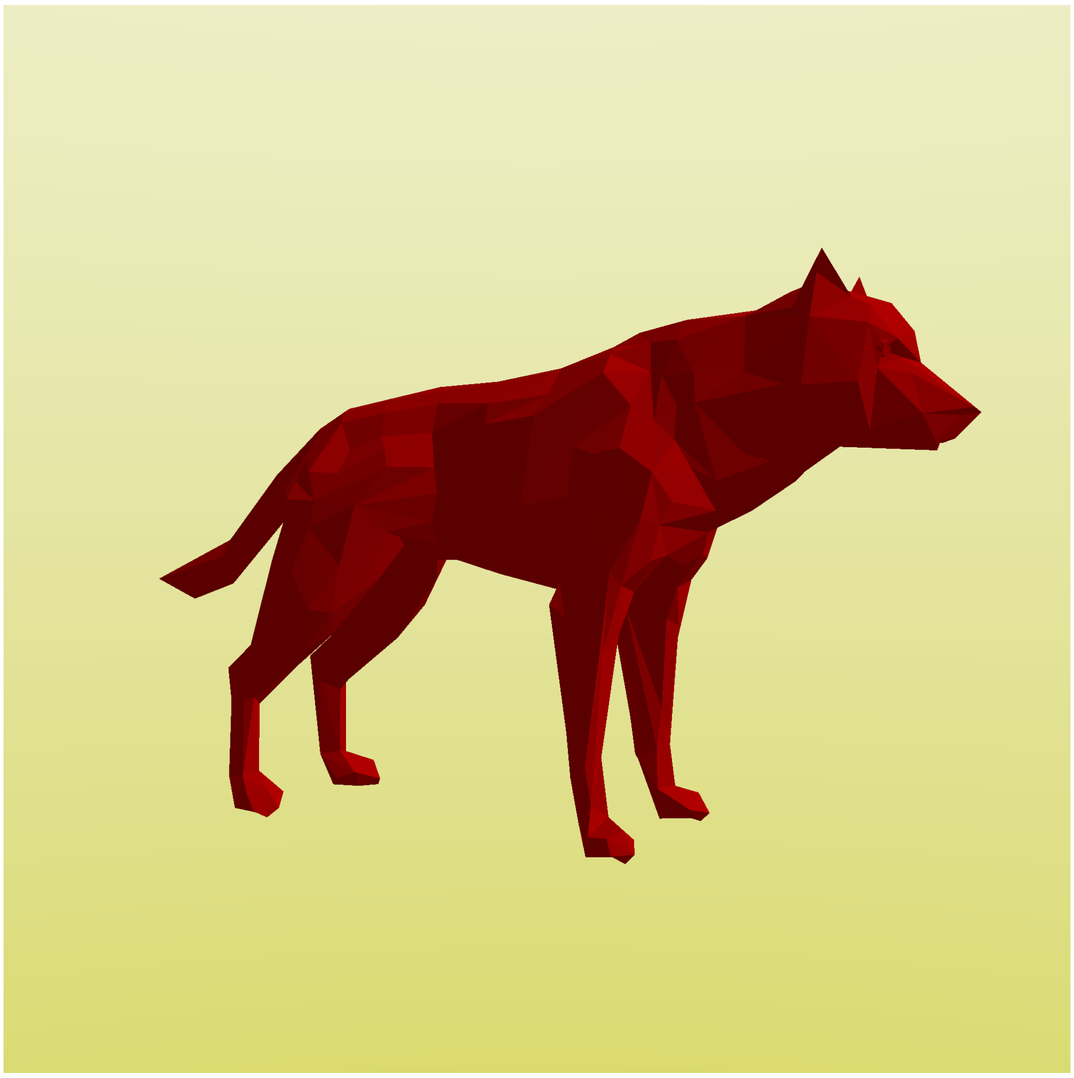

Figure 15: Latest improvement, a 2000x2000 pixels Wolf.

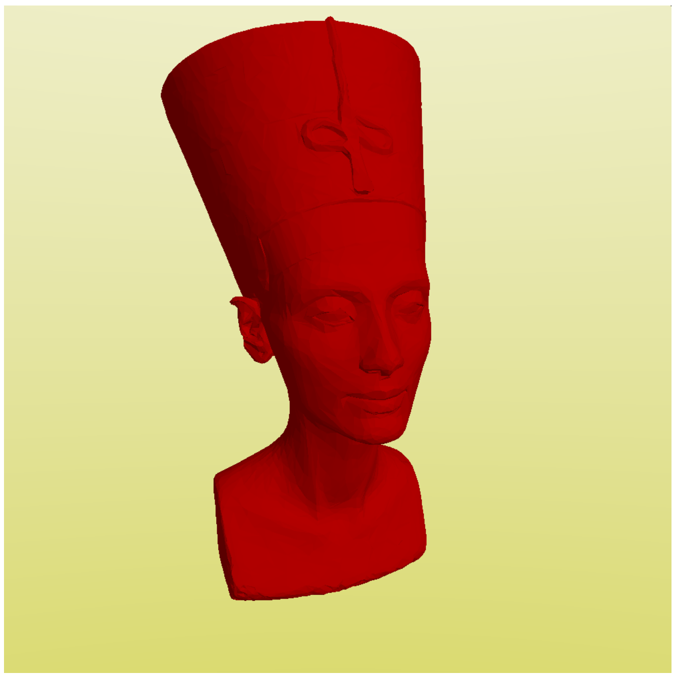

Figure 16: Latest improvement, object contains more than 10,000 triangles.

6 Discussion

We have implemented spheres, planes, disks and triangles as shapes. A triangle is a key object for graphics. Anything can be made of triangles. We have made a .obj-file loader for loading objects stored as triangles in a .obj-file.

The teapot in figure 13 is loaded from a .obj-file and consist of 2463 triangles. The loader sets bounding volumes around the object. The calculation for the bounding volumes are one point for improvement. TODO For the calculation to be general for all objects possible in the .obj-file, the bounding volume is often very big in relation to the object within.

We have used our spheres as bounding volumes. Since a sphere can encapsulate things, it was a fast way to implement bounding volumes. For an optimal bounding volume, the shape that encapsulates the object inside has to be as tight as possible. That means that in some cases a sphere would be a good choice and sometimes a bad choice.

Our way of a setting bounding volumes is made in a general way, for fitting the worst case scenario. This means it's not optimal for many shapes. However, it proves the point for the advantages of bounding volumes. We have tested our bounding volumes as shown in the result section.

In the first example, our ray-tracer was 5.31 times faster with bounding volumes and has 7.80 times fewer intersection tests. In our second example, we have used a much bigger object, the Utah teapot, with 2463 triangles. Also in this example is the ray-tracer 5 times faster with bounding volumes. Furthermore, the rendering time was also cut substantially with concurrency, we managed to render a 10813 triangle 3D-Object in under 9 seconds.

A way to make more efficient bounding volumes is to make them for an object itself. For large objects, there is no reason to check the triangles to the left, when we know we are on the right side of the object. If the object is split into serial bounding volumes, we only needed to check for the intersection for points we know are in the area.

This leads to another optimization model. The object can be split up into more bounding volumes. The entire screen could be split up into 4 boxes. These boxes could be split into 4 boxes and so on. This would create a quad three as described in [6].

We have furthermore improved the time complexity for the ray-tracer with multi-threading. We have implemented a simple form for concurrency and for our test example have we improved the time from 45 seconds to 9 seconds.

The light serves as a common world position that is referenced between all shapes and objects in the scene. It holds the responsibility of providing a position in the world to calculate the individual shapes visibility and colour.

Given the nature of programming and debugging, fixing some issues can lead to complications with other aspects otherwise previously perfectly functional. This was the case with our plane, and in extension, disks. Due to our triangle, probably the most significant and useful shape out of them, not wanting to work unless we flipped our camera. By fixing our triangles with a flip of our camera, the plane proved to contain bugs now, and the plane wouldn't work with flipped values in its intersection method. In extension to this, the disk that heavily relies on the plane to work broke as well. Adding onto the existing issues with our plane, another complication arises when mixing our implementation of bounding volumes with our rectangular planes. Given the nature of our bounding volumes being round, adding a rectangular object inside of it which exceeds the given bounds, will force it to become round instead of its intended shape. This is fixable by converting, or implementing, a box based bounding volume and using this as a form of parent bounding box.

7 Conclusion

We have succesfully implemented a simple ray-tracer. We have implemented certain shapes, namely a plane, disk, sphere and a triangle. By implementing bounding volumes we have shown that bounding volumes are improving time complexity. In our test scenarios the time complexity has been improved by a factor of five through the usage of our bounding volumes. Bounding volumes can be structured as trees for improvement. By splitting the screen into four boxes, and then recursively walking inside these boxes, a quad-tree structure can by made, and this can improve time complexity further. To improve our time complexity even further, we have implemented concurrency. In certain tests, this improved the time complexity from 45 seconds down to 9 seconds.
We have implemented light and shading, and to test this on a large scale, we made a simple obj-file loader. The ray-tracer is now able to load an obj-file and render the object defined by the vertices and faces stored in the file. Additionally, it is able to correctly sort and render the objects in the proper order, ensuring the closest is rendered instead of the farthest.
Ray-tracing calculates the color for every pixel in the screen through intersections with the objects that exist in the scene. This makes it slow compared to the rasterization that uses a object-centric approach instead of the image-centric one that ray-tracing utilizes.

8 Outlook

Ray-tracing is a popular method for rendering 3D-images. The method is known for creating realistic images, but also being slow.
Rasterization is also a popular method for rendering 3D-images, but for a whole other reason. Rasterization is fast and because of this, it is used in GPUs to render real-time 3D graphics. However; rasterization is not the only method, is widely used. Ray-tracing is often used commercially in movies, to render 3D-scenes or objects. When making a movie, the quality of the image is more important than the time it takes to render it.
Pixar Animation Studio is a studio, that creates animated movies and short films. For creating these movies, Pixar uses Renderman®. This software, contains a highly optimised version of a ray-tracer, with different visual improvements as well as optimised speed. Renderman®is released by Pixar but is used by many other companies for 3D-effects. [9]
However, the future of ray-tracing may be more than image- and movie-rendering. GPUs, using ray-tracing to render 3D-images, have been used, to render in real-time. With computers constantly getting more powerful, such GPUs might become normal in the future. [8]

References

[1] Andrew S. Glassner et. al. *An Introduction to Ray Tracing*. Acedemic Press, 1991.

[2] John Burkardt. *OBJ Files A 3D Object Format*. May 2017. URL: http://people.sc.fsu.edu/~jburkardt/data/obj/obj.html.

[3] Keenan Crane. *Keenan Crane*. May 2017. URL: https://www.cs.cmu.edu/~kmcrane/Projects/ModelRepository/.

[4] free3d. *Free obj objects*. May 2017. URL: https://free3d.com/3d-model/low-poly-wolf-71850.html.

[5] Henrik. *Ray trace diagram*. May 2017. URL: https://commons.wikimedia.org/wiki/File:Ray_trace_diagram.svg.

[6] Eric Lengyel. *Mathematics for 3D Game Programming and Computer Graphics Third Edition*. Course Technology PTR, 2012.

[7] Renfu Lu. *Light scattering technology for food property, quality and safety assessment*. CRC Press, 2016.

[8] Michael John Muuss and Maximo Lorenzo. *High-Resolution Interactive Multispectral Missile Sensor Simulation for ATR and DIS*. May 2017. URL: http://ftp.arl.army.mil/~mike/papers/95cadsymp/sensors.html.

[9] Pixar. *RenderMan Technical FAQ*. May 2017. URL: https://renderman.pixar.com/view/DP25847.

[10] Scratchapixel. *Introduction to Ray Tracing: a Simple Method for Creating 3D Images*. Apr. 2017. URL: https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-ray-tracing.

[11] Scratchapixel. *Rasterization: a Practical Implementation*. May 2017. URL: https://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation/overview-rasterization-algorithm.

[12] Scratchapixel. *Ray Tracing: Rendering a Triangle*. May 2017. URL: https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/ray-triangle-intersection-geometric-solution.

[13] Scratchapixel. *Ray-Plane and Ray-Disk Intersection*. May 2017. URL: https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-plane-and-ray-disk-intersection.

[14] Peter Shirley. *Ray Tracing in One Weekend (Ray Tracing Minibooks Book 1) eBook: Peter Shirley: Kindle Store*. URL: https://www.amazon.com/Ray-Tracing-Weekend-Minibooks-Book-ebook/dp/B01B5AODD8.

9 Process evaluation

To ensure that everyone in the group had a general understanding of the subject, we sat down as a group and wrote the most basic implementation of a ray tracer. Following this initial work, team members branched out and subsequently began to work on more specific aspects of the project to improve overall work efficiency and speed. Whereas those more experienced with programming sat out to do this, the rest began working on the report and acquired credible and valid material for the programmers to gain inspiration from. Generally, everyone has participated in every aspect of the project, some more in certain fields than others, but everyone has had something to do with more or less every aspect. There has been general equality in terms of work and what each individual has participated with.

During the process we have frequently updated our group journal, where our current process, and future tasks were noted. This served as a splendid way to gain oversight and properly manage the project in a sense that everyone were kept up to date throughout the development of the project.

Using our supervisor has proven to be extremely useful throughout the whole process, especially in certain situations where we may have gotten stuck with a certain issue, or if there has been a problem we weren't exactly sure how to tackle.
Throughout the weeks set aside for this project we've scheduled weekly meetings with our supervisor to ensure what were doing was correct and that we were on track. This served as a great way to ensure we never strayed far off of our path. Reaching our supervisor was easy, and allowed for some ease of mind during the somewhat stressful work at times.

The teamwork throughout the development of this project has been critical to the final outcome. Everyone has been able to work together in a mature and professional environment, and been understanding and flexible in regards to meetings and work hours. Without this teamwork, the product produced would have been of quite bad quality. Some issues associated with the group has been difference in programming experience and skill, some had more and were better, and therefor had to do more programming as a result. Granted, however, everyone in the group has been kept up to date throughout the development of the code, to ensure everyone understood what was going on and it wasn't just gibberish to them.
Few issues arose during development, namely the implementation of triangles, which is a crucial part in generating figures depicting various things, such as the tea pot. We waited too long to ask our supervisor for help in this regard, which lead to an unfortunate loss of time that could've otherwise been used for other things.

The overall process of the project has been a good experience, it has given us a lot of tools and experiences we can use in future projects of similar calibar, such as the group journal and the knowledge to ask for help in good time. Working on this project has given us an insight, and taught us, how projects are made at a academic level. Notable experiences include the need to do proper research for material to be used in the development, before the actual development of code begins. Due to lack of research and material, we became stuck on several issues that required us to take the time to find, and read, articles and books on how to solve the issue(s).
Prior to meetings with our supervisor, we made sure to prepare a series of questions that were necessary to have answered before we could progress further. In hindsight, further elaboration and clarification on both what kind of answer we needed from the

question, but also how to word it, would've been favorable. It would've allowed us to gain an answer we could work with straight off the bat, instead of having to interpret, and philosophize over it.