Ray Tracing and Accelerating Structures

Roberto Carrillo Granados

969197

Submitted to Swansea University in fulfilment
of the requirements for the Degree of Bachelor of Science

Swansea University
Prifysgol Abertawe

Department of Computer Science
Swansea University

May 8, 2020

Declaration

This work has not been previously accepted in substance for any degree and is not being con- currently submitted in candidature for any degree.

Signed Roberto Carrillo Granados (candidate)

Date May 8, 2020

Statement 1

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed Roberto Carrillo Granados (candidate)

Date May 8, 2020

Statement 2

I hereby give my consent for my thesis, if accepted, to be made available for photocopying and inter-library loan, and for the title and summary to be made available to outside organisations.

Signed Roberto Carrillo Granados (candidate)

Date May 8, 2020

Abstract

This dissertation aims to give an insight into ray tracing, by first illustrating its origin and principles, then briefly comparing it to rasterization which is the predominant rendering algorithm in the industry of computer graphics.

The following chapter gives a more technical explanation of ray tracing, going from the most basic algorithm to the more advanced recursive ray tracing, while also describing the implementation of the Phong reflection model and the different objects which will be later used in scenes.

In the second part, different accelerating structures are discussed with the intention of confronting the weak spot of ray tracing, which historically has been its render times when required to deal with more complex scenes with a significant number of primitives. Several renders and data are displayed to show a relation between bounding volume hierarchies and a significant speedup of render times, to serve as the conclusion of the dissertation.

Acknowledgements

I would like thank all the people who I have met during these three years and have made Swansea such a special place to live in, especially my prodigal housemate Jose Ignacio, and my classmate Cavan who made group assignments an enjoyable thing to do. Mark, my supervisor, for delivering one of the most interesting modules throughout the degree, Computer Graphics, which got me interested in ray tracing. And of course, my parents Sebastián and Margarita and my sister Macarena for their help and support, without which I could have not made it here.

Table of Contents

Chapter 1

Introduction

This dissertation will address the subject of ray tracing, which is a technique used in computer graphics to generate realistic-looking 3D images. It represents a solid alternative to other rendering methods such as rasterization, having both advantages over it but at the same time disadvantages due to the intrinsic nature of ray tracing; even though the algorithm per se relies on basic principles, the complexity of ray tracing stems from the computational power required to produce ray-traced scenes. Ray tracing has been around for quite some time; pioneer Appel introduced some of the ideas that laid the foundations for ray casting back in 1969, in *Some techniques for shading machine rendering of solids*, where he proposed a method to shoot rays into a scene to get each pixel colour. The difference between early ray casting and ray tracing is that the former only shoots single rays into the scene, finding the closest intersection with an object to get the object's shading using its properties and the lights in the scene. Meanwhile, the ray tracing algorithm extends ray casting in a recursive manner, spawning new rays to simulate phenomena such as refraction, reflections and shadows, as first proposed by Whitted in 1979. The problem of ray tracing, however, is the associated computational cost of tracing a high number of rays and finding their intersection, thus making rasterization the predominant option for image rendering especially when it needs to be done in real time.

Meanwhile, rasterization uses triangular meshes to represent three-dimensional models in a scene. In these triangular meshes, the vertices of a triangle intersect with other triangles. Each vertex stores useful information for image processing, such as its colour, texture, or surface normal. [1] Then, the 3D models are projected onto the image plane and represented as bi-dimensional objects, computing which pixels are overlapped by the objects through the use of different techniques, such as the scanline algorithm or the edge function; a method first presented by Pineda in 1988 which classifies points on a 2D plane subdividing them into three regions.

The resulting value of the function can be used to determine if a pixel is contained within the polygon [2]. Rasterization can deliver image outputs in a short space of time, sacrificing image quality for speed.

Until very recently, with the introduction of the new generation GPUs that have made it possible to render complex scenes using real time ray tracing, it was not contemplated as an alternative to rasterization for real time graphics, while now we can see how video games developers are starting to introduce ray tracing for image processing. Personally, I have always been fascinated by how realistic ray traced images look, I used to wonder if we someday would have the technology to make real time ray tracing a reality. In the past, however, it has been used for animation that did not require real time processing by companies such as Pixar, which own huge render farms for this purpose. In the case of Pixar, the company added ray tracing-based features to their proprietary RenderMan engine to allow for more advanced effects such as ambient occlusion and detailed shadows, resulting in a hybrid of ray tracing and a scanline Reyes algorithm [3]. Good examples of this are films like Cars, which have been praised by critics for their lighting and artistic style that has been archived through ray tracing.

1.1 Objectives

The aim of this project is ultimately to have a fully functional ray tracer in Java that can render complex scenes, and more specifically:

- Understand the principles of ray tracing through the analysis of the main algorithms and specific details.
- Produce a working implementation of a ray tracer, to archive the goal of producing photorealistic images, by adding shading, reflection and refraction.
- Exhaustive testing of the software and fixing any possible bugs. Learn how to manage time in order to fulfil the project management goals.
- Research methods to optimize and speed up the process of rendering to the maximum extent possible.

- Provide a comparison between the results archived using naïve ray-tracing and those archived by using bounding volume hierarchies, analysing and discussing the efficacy of the implementation.

Chapter 2

Ray Tracing: an overview

In real life, we are able to see light because the objects that surround us absorb certain frequencies of light which are emitted from a light source and then, light rays of different frequencies bounce off of them. Those rays then travel into our eyes and there they get to the retina, where an upside-down image is formed, which is then interpreted by the brain. In other words, we could say our eyes act as sensors by capturing the photons which make up light rays. Digital cameras do follow a similar process for image capturing through sampling and quantization.

Ray tracing intends to mimic this process except it does it the other way around: to recreate a scene rays are traced from a pre-defined point in space (which acts as the camera), until they intersect with an object in the image plane, and the algorithm works out what the colour of a pixel should be given the object colour and the light contribution at the hit point.

2.1 Eye-based and light-based Ray Tracing

It should be noted that the type of ray tracing discussed in this dissertation is also known as eye-based ray tracing. Casting rays from the light source to simulate the photons' real path would take a considerable amount of time and considering that many of the rays would not even hit the camera and hence would be of no use this approach does not give any real benefit. [4]

Some authors have proposed a combination of this method and light-based ray tracing to improve the simulation of indirect light effects, separating paths from a light source and an image plane, and then connecting them with a deterministic ray, ensuring light will reach the eye. [5]

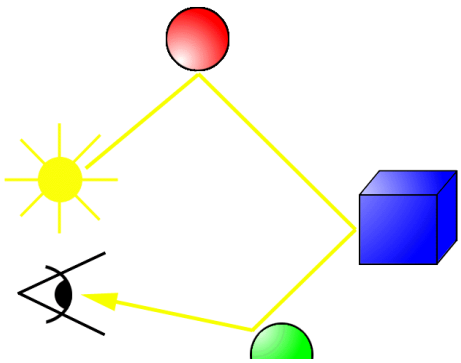


Figure 1. Light-based ray tracing [6, fig 2]

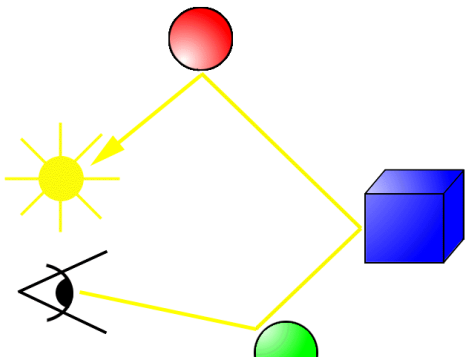


Figure 2. Eye-based raytracing [6, fig 1]

As aforementioned, to render a scene in a eye-based ray tracer, we need to shoot rays from every pixel and test those against the objects present to see if they are hit by the rays. Those objects are defined by their position in the X, Y and Z-axis, thus making it possible to calculate the ray-object intersection, as rays are contained in the 3D space too, and defined by their origin and a direction vector. Objects include a range of geometrical shapes such spheres, triangles, cylinders, cubes, etc. In this project, we will be looking at spheres, triangles and cuboids. Triangles will provide the ability to render more complex shapes as they can be used in triangle meshes, while cuboids will be used for axis-aligned bounding boxes (AABB), which are crucial in the construction of bounding volume hierarchies.

In a eye-based ray-tracer a few important concepts can be differentiated:

- Camera
    - This will act as the “eye” of the ray-tracer, which will be set at a certain point in space and look at a point in the scene. A description on how to set a movable camera is given in this chapter.
- Rays
    - Traced from the camera origin to the scene, rays will let us know what objects are present and at what distance. They are also used for shading.
- Lights
    - Lights illuminate the scene with a pre-defined light intensity. We will implement multiple light sources that can be added to a scene.

- Shapes
    - They are the primitives that will make up the objects contained in the scene.

2.2 Setting up the camera and rays

From the beginning of this section until the end of the dissertation, we will make heavy use of vectors. To handle vectors correctly, instead of using the default vector class from Java, a custom vector class was created, which will allow us to carry out operations such as the dot product, the cross product or finding the length of a vector, and so we can implement any other functions if needed. This vector class is also used to store colours, given the convenience of storing all RGB components together in a vector.

Having said this, the first step to set up a scene and be able to cast rays, is to specify the camera position and have all objects defined within world coordinates. A movable camera will be defined by its location in space, the target point, a view up vector, the camera's field of view, and the image's aspect ratio.

With these three components, we can calculate the other vectors that are needed for ray casting, as to construct a ray we will need its origin and direction. A ray is expressed by the parametric line equation $y = o + td$, where 'o' is the origin of the ray, 'd' is the direction vector and 't' is a constant $t > 0$. Those values are stored in the Ray class, where the origin and direction are passed in to the constructor.

As we are working with a perspective projection, all rays will have the same origin, that is, the camera origin, but their direction will be different for each pixel, and this must be taken into account. The normalised direction vector of a ray can calculated if we know the coordinates of the pixel in the virtual screen and the camera origin:

$$dir = \frac{P1 - P0}{||P1 - P0||} \quad (1)$$

where ‘P0’ are the camera origin coordinates and ‘P1’ are the target coordinates in the virtual screen. In the virtual screen (figure 3), the pixel at (0,0) would have world coordinates (W/2, H2), with ‘W’ and ‘H’ being the width and height of the screen respectively in world coordinates, and the pixel at the upper right corner would have world coordinates (W/2, H/2)

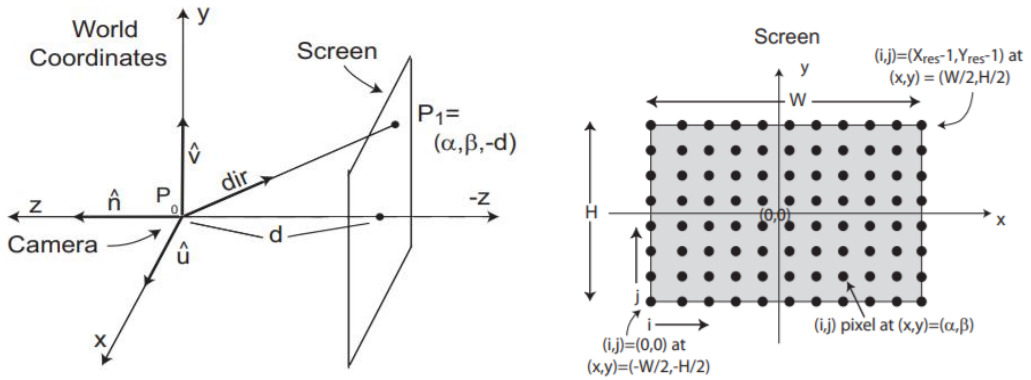


Figure 3 [7, fig. 4]

If we generalise this, a pixel located in the position (i, j) in the virtual screen has the following world coordinates:

$$P = (x, y, z) = (x, y, -d) = \left(\frac{-W}{2} + \frac{W*i}{X_{res}-1}, \frac{-H}{2} + \frac{H*j}{Y_{res}-1}, -d\right) \text{ [7]}$$

where X_{res} is the number of pixels in a row, Y_{res} is the number of pixels in a column, and d is the distance between the camera and the view screen plane.

In Figure 3, the camera was placed in the origin, and since it was aligned with the XYZ axes thus being an orthonormal basis we could easily calculate the world coordinates of a pixel in the virtual screen using the expression above, but we cannot assume that the camera will always be static in the origin, as we aim to implement a camera model that can be positioned anywhere in the scene.

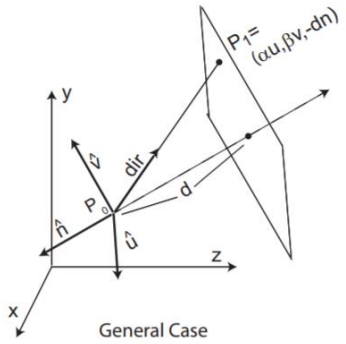


Figure 4 [7, fig. 2]

Figure 4 depicts the case where the camera is not aligned with the axes. In this case, if we want to derive the world coordinate of a point in the virtual screen, we will need a new orthonormal basis ($\hat{u}$, $\hat{v}$, $\hat{n}$) to express the pixel's coordinates. This means we need a set of $\hat{u}$, $\hat{v}$, $\hat{n}$ vectors which are orthogonal between each other and are also normalised.

The view plane normal can be calculated by subtracting the target point to the camera's origin and normalising the vector, the '$\hat{n}$' component of the new basis will then be equal to the normalised view plane normal. The cross product of the view up vector and the view plane normal will give us the '$\hat{u}$' component, and '$\hat{v}$' is the result of the cross product of '$\hat{u}$' and '$\hat{n}$'.

Now we can express any point P as a linear combination $P = \alpha\hat{u} + \beta\hat{v} - d\hat{n}$, where 'd' is the distance and both α and β can be calculated as

$$(\alpha,\ \beta) = (\frac{-W}{2} + \frac{W*i}{X_{res}-1}, \frac{-H}{2} + \frac{H*j}{Y_{res}-1})$$ [7]

In the implementation, as we are using a vertical field of view parameter to set the camera's aperture angle, and if we have previously calculated the aspect ratio, the width term can be expressed as:

$$Width\ = aspectRatio * Height$$

And in this case, the height is also given by:

$$Height = \tan\frac{\theta}{2}$$

where θ is the field of view value of the camera, expressed in radians. [8]

Now we are able to cast rays from a camera into the scene, as the two share the same origin, while the direction of the ray can be calculated with (1) and a point in the scene in world coordinates.

2.3 Shapes and intersections

Intersections play an important role in a ray-tracer, as it is vital to find if a ray hits an object in the scene -or more than one- and which is hit first. All of the shapes -with the exception of AABB- that are going to be added to our ray tracer inherit from a parent Shape class, which implements an abstract intersect method to be overridden in the derived classes. Hence, if we have a list of Shapes and we want to find the closest intersection, we can do so in the following Java method:

```
public boolean intersect (Ray r) {
    double distance = Double.MAX_VALUE;
    boolean intersect = false;
    for (int i = 0; i < shapes.size(); i++) {
        Shape shape = shapes.get(i);
        if (shape.intersect(r)) {
            double shapeDistance = shape.getDistance(r);
            intersect = true;
            if (shapeDistance < distance) {
                distance = shapeDistance;
                intersectShape = shape;
            }
        }
    }
    return intersect;
}
```

Figure 5

This method loops through the list of shapes, finding the closest hit and saving the intersected shape in a class variable, which later will be of use to find the surface normal.

Apart from the intersect method, the parent Shape class also implements the following abstract methods:

- AABB getBBox();
    - Returns the pre-computed bounding box -an AABB object- of a given shape.
- Vector getNormal();
    - Returns the surface normal of an object.
- double getDistance(Ray r);
    - Returns the distance of the ray passed as an argument to the shape.

In the following section, we will look from a geometrical point of view at how these abstract methods are implemented for each shape in our ray tracer.

2.3.1 Spheres

The equation of a sphere with centre c and radius r can be stated as $(\vec{p} - \vec{c})^2 = r^2$ where p are all points of the sphere. If we substitute 'p' by $o + td$, which is the equation of a ray, the points that belong both to the ray and the sphere can be found, and it can be expressed as:

$$(o + td - c)^2 = r^2$$

expanding and rearranging terms gives:

$$d^2t^2 - 2td(o - c) + (o - c)^2 - r^2 = 0$$

Solving for t, as it is a quadratic equation, it has roots

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

where

$$a = d^2 \quad b = 2t(o - c) \quad c = (o - c)^2\, r^2$$

which all are known values. In the Sphere class, we store the vectors containing the radius and centre as a class variable.

Depending on the discriminant $\Delta = b^2 - 4ac$, the equation will have two, one or no real solutions:

- In the case where $\Delta < 0$ the equation doesn't have a real root, meaning the ray does not intersect the sphere.
- If $\Delta > 0$ it will have two solutions and the sphere intersects the ray in two different points.
- And the last case, if $\Delta = 0$, there is only one point of intersection.

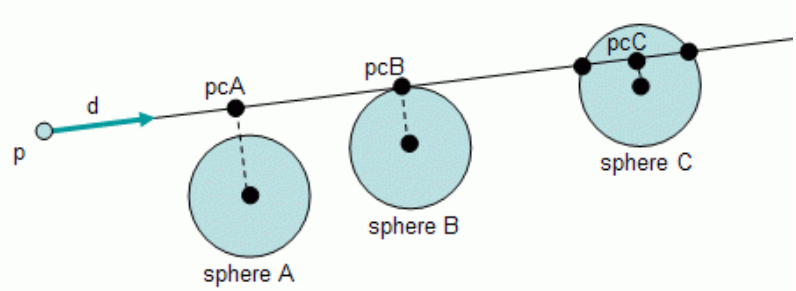


Figure 6. The three cases described above, no intersection, only one point of intersection, and two points [9]

The surface normal of a sphere is computed at the same time as the intersection; if the equation described above returns a real root t > 0 (as we would not need it if it does not intersect the ray) and since we already have 't', which is the distance between the ray origin and the sphere, we can simply plug its value into the ray equation which will then return the surface hit point P.

$$\vec{P} = \vec{o} + t\vec{d}$$

In the Java implementation, we can do this by calling the method pointAt in the Ray class. Once we get the value of P, we then subtract the sphere centre C, which returns a vector N:

$$\overrightarrow{(P - C)} = \vec{N}$$

Which is the surface normal that is then normalised for further shading processing.

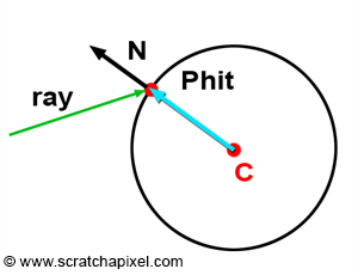


Figure 7. A sphere shape hit by a ray at point Phit.

The distance getter method is implemented in a similar fashion, as we just return t, the value we previously got from the ray-sphere intersection equation and stored in a class variable.

Spheres will also need an implementation of bounding boxes we can return through the getBBox method; for this we will need two minV and maxV vectors to create an instance of a bounding box. The minV and maxV vectors represent the minimum and maximum coordinates of the sphere in the XYZ plane, and in this case are:

$$\overrightarrow{(C+R)} = \overrightarrow{maxV} \qquad \overrightarrow{(C-R)} = \overrightarrow{minV}$$

where C is the centre of the sphere and R is a vector R = (radius, radius, radius), where the radius is stored for convenience in order to perform subtraction [RT next week]

2.3.2 Triangles

Having implemented spheres, the next milestone in the project was enabling our ray tracer to render triangles. More complex objects such as the Stanford Bunny are imported from .txt files that store triangular meshes, defined by a list of vertexes and triangle faces linked to them, then read in by the program in the class ReadShapeFile, which constructs an ArrayList of Shape containing all the objects in the scene.

A triangle primitive will belong to the Triangle class, which as mentioned before, will be derived from the parent Shape class. This Triangle class stores a set of vectors v0, v1 and v2, which are the triangle's vertexes in the plane, a double t which is the distance between a ray and the triangle hit point (and is returned by the getDistance method), and the triangle's bounding box.

The vectors defining the triangle's edges are v0v1 and v0v2, which are calculated by subtracting v0 to v1 and v0 to v2 respectively. The getNormal method will return a triangle's surface normal, which is the result of the normalised cross product of v0v1 and v0v2.

For the ray-triangle intersection, we are presented with two main algorithms to choose from:

- Barycentric coordinates

Any point inside a triangle can be expressed in barycentric coordinates as a linear combination of its edges, such that $P = uA + vB + wC$, in this expression A, B and C are the vertexes of the triangle, and u, v, and w are three scalars $u + v + w = 1$. Consequently, if we know two of the scalar values we can establish the third one as: $w = 1 - v - u$. This also implies $1 \geq v + u$ and $v \geq 0, u \geq 0$

A point P expressed as a combination of these scalars h, is inside the triangle, if and only if $0 < (w,\ v,\ u) < 1$, meanwhile if $0 \leq (w,\ v,\ u) \leq 1$, P might lie on the edge of the triangle.

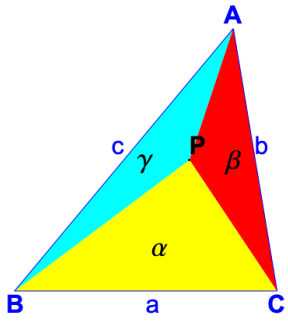


Figure 8. Barycentric coordinates divide triangle ABC in sub triangles APB, APC and BCP.

In other words, barycentric coordinates divide the triangle in three sub-triangles, and then express a point as a combination of the areas of the sub-triangles, as their areas must be proportional to the area of the original triangle.

Therefore, to check for a possible ray-triangle intersection, we will need to find a point $P = uA + vB + wC$, that is in the same plane as the triangle and is perpendicular to the ray direction. Then we can test for intersection using the criteria presented above, and if the point meets the requirements to be inside the triangle the ray intersects the triangle. We will also need to check if the plane is parallel to the ray, or if the triangle is behind the ray, in which cases the ray does not intersect the triangle.

- Möller-Trumbore algorithm

Despite its simplicity, if we are using barycentric coordinates we will need to compute the plane containing the triangle and then test for the intersection. The Möller-Trumbore algorithm addresses this issue and provides a faster triangle intersection algorithm with minimum storage, eliminating the need to find the triangle's plane; it only stores the vertices of the triangle and presents an up to 50% memory save for triangle meshes [12]

The original algorithm proposes two different routines, a culling option for one-sided triangle intersections, which discards all triangles that are back facing the ray, and another for two-sided triangles. During testing, the culling branch was shown to be up to 70% faster. However, due to the fact that it gets rid of the back-facing triangles, it may lead to non-accurate results when transparent glass-like objects are rendered, and therefore the algorithm of choice was the non-culling branch.

It makes use of the fact that was mentioned before; the possibility of expressing any point P in barycentric coordinates as $P = (1 - u - v)\, A + uB + vC$, which then intersected with a ray R$= O + tD$, yields:

$$O + tD = (1 - u - v)\, A + uB + vC$$

After rearranging terms, we get the following system of linear equations:

$$[-D, B - A, C - A] \begin{bmatrix} t \\ u \\ v \end{bmatrix} = [O - A] \qquad \textit{[12, fig. 4]}$$

The Möller-Trumbore algorithm, effectively solves these equations by finding the value for u, v, t by using Cramer's rule, which we will be implementing in Java following the original implementation:

```
if (det > -EPSILON && det < EPSILON) return false;
double invDet = 1.0 / det;
Vector tvec = r.getOrigin().sub(v0);
double u = tvec.dot(pvec) * invDet;
if (u < 0.0 || u > 1.0) return false;
Vector qvec = tvec.Cross(v0v1);
double v = r.getDirection().dot(qvec) * invDet;
if (v < 0.0 || (u + v) > 1.0) return false;
t = v0v2.dot(qvec) * invDet;
```

Figure 9. Java implementation of the Möller-Trumbore algorithm [12]

This method will return false, meaning it did not find an intersection, if u or v do not fulfil the requirements for the point to be inside the triangle, otherwise it will return true, and 't' -the distance- will be stored in the class variable.

When the triangle constructor is called and a new instance is created, the triangle's bounding box is set on the fly. We must then set the maximum and minimum coordinates of the triangle so they can be passed as an argument to the AABB constructor. This is done in a line of code for each axis using Java's min and max functions:

```
double minX = Math.min(v0.getX(), Math.min(v1.getX(),v2.getX()));
...
double maxX = Math.max(v0.getX(), Math.max(v1.getX(),v2.getX()));
...
Vector max = new Vector(maxX, maxY, maxZ);
return bBox = new AABB(min, max);
```

Figure 10.

2.3.3 Infinite planes

Any infinite plane can be defined by three points contained in the plane and a normal vector N. This is basically how planes are constructed in our implementation, three vectors v1, v2 and v3 are passed into the class constructor which and N is computed as the cross product:

$$\vec{N} = \overrightarrow{V2V3} \times \overrightarrow{V2V1}$$

The dot product of two vectors a and be can defined as:

$$a \cdot b = ||\vec{a}|| \left\|\vec{b}\right\| cos\theta$$

We will make use of this property to derive the ray-plane intersection. Given a point in the plane q, another point P will be in the plane if it fulfils the following property:

$$\vec{N} \cdot (\vec{P} - \vec{q}) = 0$$

Since the normal vector is perpendicular to the surface, the dot product between the surface normal and any vector contained in the plane must be 0.

To calculate the ray-plane intersection, we substitute P with a point contained in the ray:

$$\vec{N} \cdot ((\vec{O} + t\vec{d}) - \vec{q}) = 0$$

After expanding and rearranging terms to find t, the distance from the ray origin to the plane, this expression results in:

$$t = \frac{\vec{N} \cdot (\vec{q} - \vec{P})}{\vec{N} \cdot \vec{d}}$$

If $t \geq 0$, then we will have found an intersection, and we store t in the class variable which is returned when the distance getter is called.

2.3.4 Axis-aligned bounding boxes (AABB)

For our acceleration structure of choice, a bounding volume hierarchy, we will need a set of bounding volumes we will use to enclose the other primitives in the scene. As the name suggests, those bounding volumes will be boxes ordered along an axis.

A bounding box is defined in the implementation by two vectors minV and maxV that store a box minimum and maximum values in the space. These two vectors are passed in at the time of construction. Bounding boxes are an exception to the other shapes; they are the only shape to not be inherit from the parent Shape class, as since they will not be rendered and shown in the scene and we do not need to know their surface normal, centre or hit point.

We will, however, need to find out if a bounding box is intersected by a given ray. To do so, Kay and Kajiya slab method was adapted to our needs as follows:

```
public boolean intersect(Ray r) {
    double invX = 1 / r.getDirection().getX();
    double invY = 1 / r.getDirection().getY();
    double invZ = 1 / r.getDirection().getZ();

    double t1 = (minV.getX() - r.getOrigin().getX())*invX;
    ...
    double t6 = (maxV.getZ() - r.getOrigin().getZ())*invZ;

    double tmin = Math.max(Math.max(Math.min(t1, t2), Math.min(t3, t4)
    ), Math.min(t5, t6));
    double tmax = Math.min(Math.min(Math.max(t1, t2), Math.max(t3, t4)
    ), Math.max(t5, t6));
    if (tmin > tmax || tmax < 0) {
        return false;
    }
    t = tmin;
    return true;
}
```

Figure 11. Implementation of the slab method [13]

In figure 12, a slab is the region contained between the planes y1, y2 and x1, x2 that are parallel to the y and x axis, although in the implementation we have an additional slab as we are working with 3D coordinates, making a total of six planes. Those planes are be defined by the XYZ components of the min and max vectors of the bounding box.

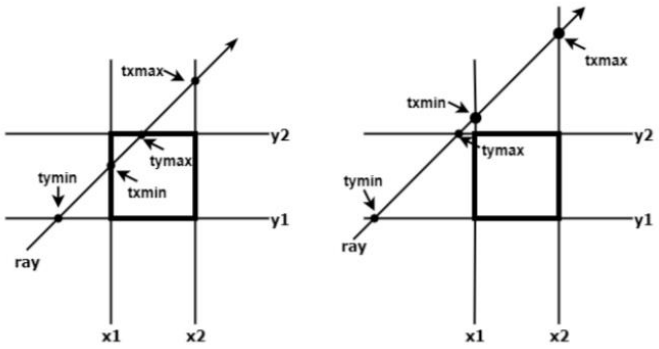


Figure 12. Graphic representation of slabs. [13, fig. 2]

As we have seen previously, the intersection between a ray and a plane is given by:

$$t = \frac{\vec{N} \cdot (\vec{q} - \vec{P})}{\vec{N} \cdot \vec{d}}$$

In this case, we can simplify this expression by eliminating N, as the normal vector of a plane that is parallel to an axis is a unit vector in the direction of that axis. This operation will have to be carried out a total of six times, one for each bounding plane of the box. Once this is done, the intersection points of the planes are compared, saving the minimum intersection point component as tmin and the maximum as tmax. It should be noted those two variables are not vectors but scalars

Then, their values are compared again: the ray does not intersect the box if tmin > tmax (the ray does not intersect the box) or if tmax < 0 (the box is behind the ray), in the other cases the box is hit by the ray. If so, the variable class t, storing the distance from the ray origin to the box, is then set as tmin.

2.4 Lights and shading

In the previous section 2.3, we have seen what objects were added to our ray tracer and learned about their properties. We are now able to render a range of shapes, but another important matter is how these will be shaded.

Local or direct illumination is the most basic form of illumination, it refers to the interaction of light with a single object and does not consider the other objects that may be present. Global or indirect illumination is a wide topic by itself which encompasses several techniques, being the most costly form of illumination as it computes the light contribution from all objects in the scene.

With the intention of providing a realistic simulation of local illumination, Phong's model was implemented which is an empirical reflection model consisting of three main components; ambient, diffuse and specular lighting. Being an empirical model means that the results this model yields are not rigorous from a physical point of view: the model was derived after studying the effects of different light conditions on objects, and for example, it does not account for energy conservation [15]. Despite this, the model is realistic looking enough to be used for our shading implementation.

To add light to our scenes, point light sources were implemented in a class Light, where a light source is defined by its intensity and a vector that sets the light location. At the same time, a given scene can have multiple light point sources, all of which will contribute to the scene's global illumination.

2.4.1 Ambient reflection

The ambient reflection value at a certain point can be defined as:

$$I_{out} = I_a k_a$$

where $I_a$is the ambient light intensity, $k_a$is an ambient reflection coefficient, which is surface dependant and ranges from 0 to 1. This is a simplistic way of calculating ambient reflection, as it assumes the ambient light is uniform throughout the scene,

when instead the contribution of all objects should be accounted for, but this approach avoids the cost of global illumination.

2.4.2 Diffuse reflection

Diffuse reflection occurs when the incoming light that hits an object is reflected and this reflection is scattered. This usually happens on rough surfaces, where the irregularities spread the reflected rays at a variety of angles. An ideal diffuse surface that reflects equal luminance in every direction is called a Lambertian surface. [30]

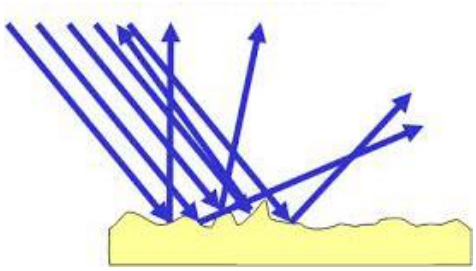


Figure 13. Representation of diffuse reflection.. [17, fig. 1]

The diffuse reflection intensity at a point I_{out} can be calculated as the product of the light intensity, the cosine of the angle formed by the incoming light direction and the surface normal at that point, and a material dependant constant K_d also known as the albedo, a unitless number that ranges from 0 to 1 which defines how much of the incoming light is reflected [17]:

$$I_{out} = k_d I_a \, cos\theta$$

Given the cosine law and its relation to the dot product, we can express $cos\theta$ as:

$$\frac{\vec{N} \cdot \vec{l}}{\left|\left|\vec{l}\right|\right| \left|\left|\vec{l}\right|\right|} = cos\theta$$

hence if we assume that both $\vec{l}$ and $\vec{N}$ are normalised:

$$I_{out} = k_d I_a \, (\vec{l} \cdot \vec{N})$$

2.4.3 Specular reflection

Specular reflection takes place when the incoming light is reflected in a single direction. Phong's equation for simulating specular reflection is:

$$I_{out} = k_s I_a (\cos\theta)^e$$

where θ is the angle formed between the reflected view vector and the reflected ray $\vec{R}$. In the case of pure specular reflection, this angle θ is 0 and gives the impression of a mirror-like surface, and 'e' is the shininess value of the object. [18]

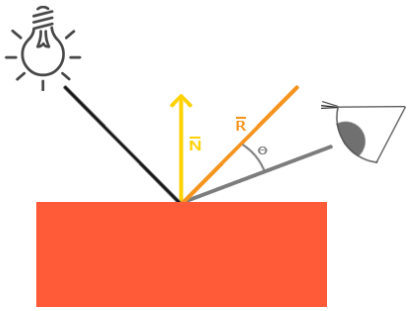


Figure 14. Representation of specular reflection.. [18, fig. 4]

For this we will need to calculate the reflected vector $\vec{R}$ which is done in a method reflect within the Scene class, which takes in two vectors (the incident ray vector $\vec{I}$ and the surface normal $\vec{N}$) and returns:

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

Following the cosine law, the formula can be expressed as:

$$I_{out} = k_s I_a (\vec{V} \cdot \vec{N})^e$$

at the same time,

$$\vec{V} = -\vec{r}$$

where $\vec{r}$ is the ray direction from the camera to the pixel.

Once we have calculated the values of the ambient, diffuse and specular components the new intensity of the point is given by the following formula which is a combination of the three components.

$$I = I_a + k_d \sum_{j=1}^{j=ls} (\vec{N} \cdot \overrightarrow{L_J}) + k_s S \qquad [19, \text{eq } 2]$$

Where:

- I = the total reflected intensity
- I_a = reflection due to ambient light,
- k_d = diffuse reflection constant
- $\vec{N}$ = unit surface normal
- $\overrightarrow{L_J}$ = the vector in the direction of the jth light source
- k_s = the specular reflection coefficient

- $S =$ the intensity of light incident from the $\vec{R}$ direction

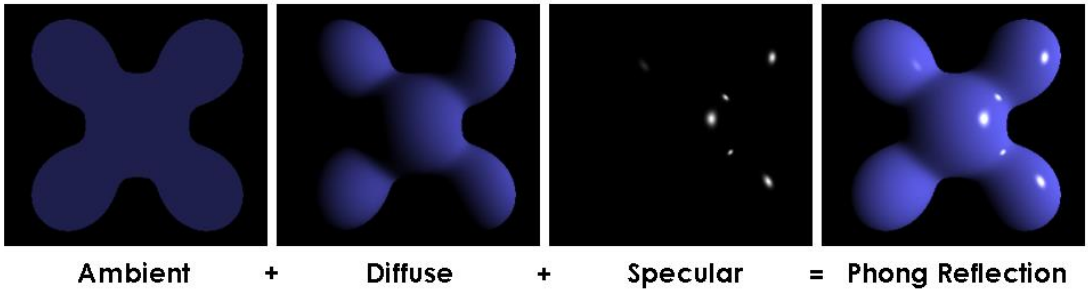


Figure 15. Representation of the Phong model components. [20]

Then, the pixel colour is set as the vector containing the RGB values of the colour resulting from the dot product of the intensity and the object colour.

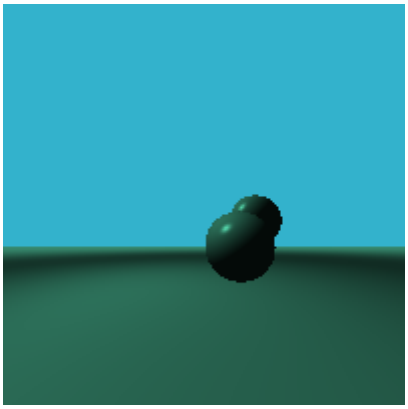


Figure 16. An early implementation of the ray tracer showing Phong illumination

2.4.4 Shadows

To render shadows, we need to cast shadow rays from the original point of intersection towards the light source and test for intersection: if an intersection is found it means there is at least one object blocking the light path meaning the object is in shadow; in this case the only contribution towards the object illumination is the ambient reflection, and we do not compute the object's specular and diffuse reflection. As mentioned, shadow rays have their origin at the hit point, although this is slightly moved in the direction of the object's surface normal to avoid the object self-intersecting and resulting in false positive intersections.

2.4.5 Recursive Ray-Tracing

Recursive Ray-Tracing was introduced by Whitted in 1980. Based on previous techniques such as Phong's illumination model, it allows for more realistic light effects. Appel's first approach used primary rays to determine the colour of the pixel if it intersects an object and analyses the object's properties, with a ray stopping after it first hits an object.

Recursive ray tracing extends this idea by making each visible intersection of a ray with a surface spawn more rays in the R direction, the P direction, and in the direction of each light source, as seen in figure 17; and when these secondary rays collide into another object, they spawn more rays, and so on.
Eventually this process stops, or a maximum amount of collisions might be set for it to stop after a threshold, to avoid, for example, an infinite loop of rays colliding between two mirrors.

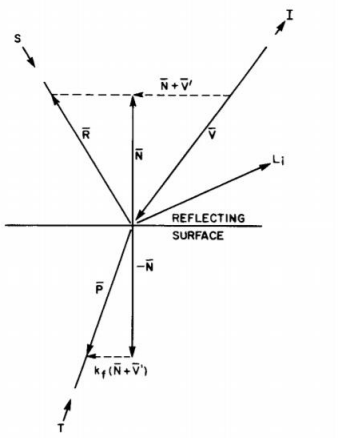


Figure 17. A ray hits a surface [19, fig 1]

It is worth mentioning that recursive ray tracing was later extended by Cook, who introduced distributed ray tracing. This approach introduces soft shadows, motion blur, depth of field and translucency, by taking multiple samples of a pixel and averaging the result [21]. More recent developments in the field of global illumination include photon mapping by Jensen in 1996, a two pass method based on the concept of photon maps; created by emitting packets of energy from light sources and storing these as they hit surfaces within the scenes. [22] Path tracing by Kajiya is another advanced global illumination method based on the original Whitted ray tracer algorithm although instead of spawning new rays in the direction of refraction and reflection when an intersection is found, the direction of these is chosen randomly, solving the rendering equation by Monte Carlo integration. [23]

In our case, we followed the Whitted technique to make our ray tracer recursive in a fairly simple way; first of all, the reflection and refraction rays are calculated and the scene is recursively tested for intersections using the new rays until the desired depth is reached, halting the recursion.

Reflection rays make use of the reflect method already implemented for specular reflection, which will return the direction of a reflected ray originating in the object's hit point, whereas for refraction we will need a new refract method that will set the refraction rays direction:

```
private static Vector refract(Vector I, Vector N, double refractStrength) {
        double cosi = Math.max(-1, Math.min(1, I.dot(N)));
        double etai = 1;
        double etat = refractStrength;
        Vector n = N;
        if (cosi < 0) {
            cosi = -cosi;
        } else {
            etai = etat;
            n = N.dot(-1);
        }
        double eta = etai / etat;
        double k = 1 - eta * eta * (1 - cosi * cosi);
        return k < 0 ? new Vector(0,0,0) : I.dot(eta).add(n.dot((eta * cosi
- Math.sqrt(k))));
    }
```

Figure 18. Our Java implementation of refraction [25]

Chapter 3

Accelerating Structures

A naïve approach to ray tracing casts a ray for every pixel in the scene, then the ray is tested for intersection against every primitive. This means that for a 1000x1000px scene with 1000 primitives -and this is a low number of primitives-, 1 billion intersections will have to be carried out. If we desire to add effects such as shadowing or if reflections are implemented, more rays need to be spawned depending on the level of recursion that is to be archived. In a time where speed is a must, we have to find a way to accelerate the process of ray tracing. For this purpose, there exist a series of proposed methods: reducing the number of test intersections to be done, reducing the average cost of intersecting a ray, and replacing individual rays with a more general entity. [24]

We will be examining the first approach and analysing how the number of intersections can be reduced thus effectively reducing render times.

3.1 K-d trees

Space partitioning data structures such as kd-trees can be used to reduce the number of ray intersections. Kd-trees, or binary space partitioning (BSP) algorithms build a binary tree by dividing the space in a recursive manner.

To build a kd-tree, we begin creating a root node in the binary search tree and then proceeding with the construction; during this first step the bounds of the general scene are set by looping through the list of objects contained in it and getting the minimum and maximum XYZ coordinates. Next, the algorithm splits the axis that contains the greatest number of objects. Depending on the split plane, the objects that were previously contained within the world bounds are placed in the left or the right node. It could also be the case that an object is contained in both nodes if the split plane is within the object bounds. This process is carried out recursively and only stops when the termination criteria is met, and that is when a pre-defined depth

of the tree is reached, or the number of primitives within the node are below a certain threshold. In this case, the left and right children of the current node are set null, creating a leaf node.

```
stack.push(root,sceneMin,sceneMax)
tHit=infinity
while (not stack.empty()):
    (node,tMin,tMax)=stack.pop()
    while (not node.isLeaf()):
        a = node.axis
        tSplit = (node.value - ray.origin[a]) / ray.direction[a]
        (first, sec) = order(ray.direction[a], node.left, node.righ
t)
        if (tSplit ≥ tMax or tSplit < 0)
            node=first
        else if( tSplit ≤ tMin)
            node=second
        else
            stack.push(sec, tSplit, tMax)
            node=first
            tMax=tSplit
    for tri in node.triangles():
        tHit=min(tHit,tri.Intersect(ray))
        if tHit<tMax:
            return tHit //early exit
return tHit
```

Figure 19. Pseudocode for a kd-tree traversal algorithm [26]

Once the kd-tree is built, checking for intersections requires a traversal that will visit the tree nodes until it finds a leaf, in which case it will go through the list of triangles stored in the node testing them for intersection. Instead of running recursively, an optimized implementation of a traversal makes use of a stack, such as the algorithm in figure 19. One of the advantages of kd-trees over hierarchical bounding volumes is that it gets rid of the need to choose what volume is best for every object. [27]

3.2 Bounding volume hierarchies (BVH)

Another way of doing this is by using bounding volumes that surround the more complex objects in a scene made up of a high number of primitives with a box that is simpler to intersect. This way, the number of intersections that need to be calculated are drastically decreased, as the original primitives will only have to be tested for intersection if the ray hits the bounding volume.

When defining the bounding volume the difference between the area of the bounding volume and the original shape should be considered; this difference, also called the void area should be minimal in order to reduce the number of intersections. Although the choice for an object bounding volume could seem obvious, such as in figure 20, where one could think the volume that provides the tightest fit would be a sphere, picking the right bounding volume is in reality ray-dependent; it depends on whether the rays are going to be shot transversely or longitudinally in relation to the object, thus the best volume choice for a circular object as seen in the figure might be a cube instead. [28]

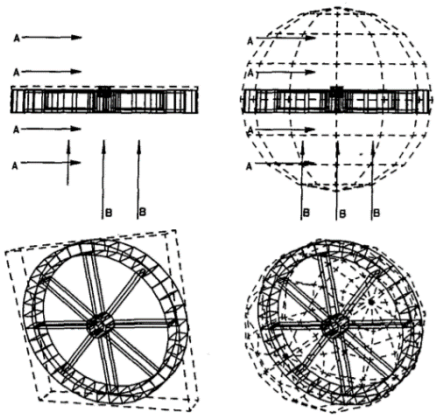


Figure 20 [27]

This goes however beyond the scope of this dissertation and we will be looking at axis-aligned bounding boxes instead, which will be the bounding volumes of choice that surround the more complex polygon meshes in the scene.

Bounding volume hierarchies are built recursively in a similar way to kd-trees. Our implementation builds upon Peter Shirley's bounding volume hierarchy as shown in *Ray Tracing: the next week*, with several differences. The bounding volume tree is built when the class constructor is called, which takes in an ArrayList containing the objects in the scene, and two numbers representing the start and end index of the list. Then we sort the list by the objects coordinates, and for this we have to pick an axis: as opposed to the book's implementation which randomizes the axis choice we will compute which of the axis contains the greatest number of shapes by ordering them by their centroid -the geometric centre of the objects- which is calculated using the each object bounding box. The BVH is then split in two nodes, left and right, which are built recursively by calling the BVH class constructor again, passing in half of the list of objects which also gets split by the list mid-point. The recursion process ends when there are only one or two shapes in the list; and in that case the remaining shapes are set as the left and right nodes, and thus those will be leaf nodes with no children. We then retrieve the bounding box value of the left and right nodes, and combine their bounding boxes so it results in a bigger bounding box, which will eventually hold the entire scene geometry.

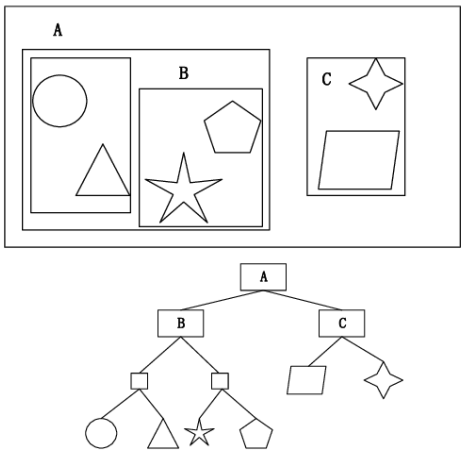


Figure 21. Visualization of a bounding volume hierarchy. [31]

One important detail is that the BVH class inherits from Shape: to override the intersect method we have to traverse the hierarchy; we test for intersections of the bounding boxes until we get to a leaf node which contains a primitive such as a triangle. If a primitive is intersected, this method returns true and the intersected shape is stored in a class variable.

Chapter 4

Results

4.1 Testing

A series of different scenes were rendered in order to carry several tests whose results were used to compare different aspects of the implementation.

The first control scene contains a triangle mesh of the Stanford Bunny, two spheres and a plane. This scene was rendered at different recursion depths and resolutions using the non-accelerated algorithm.

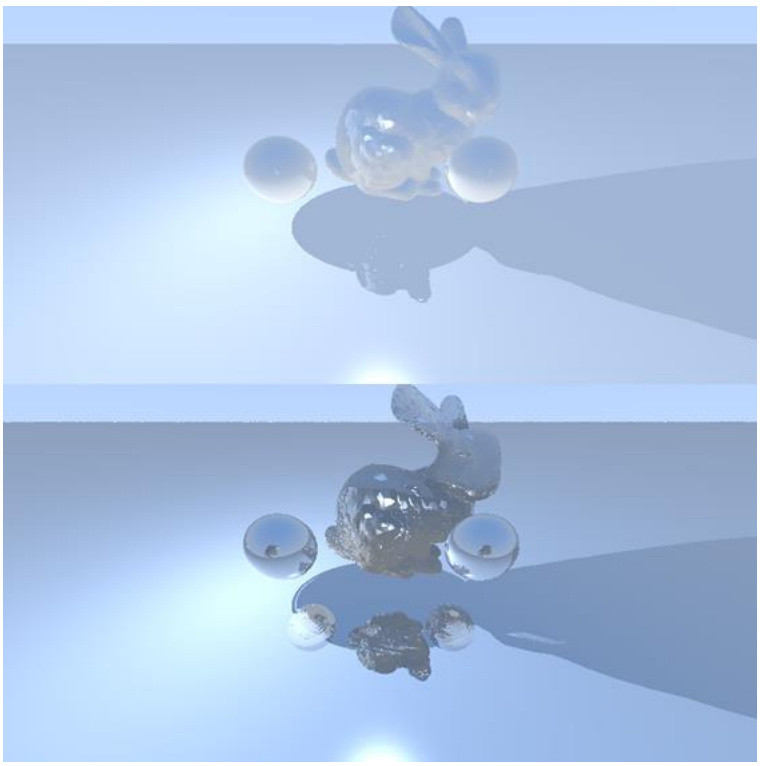
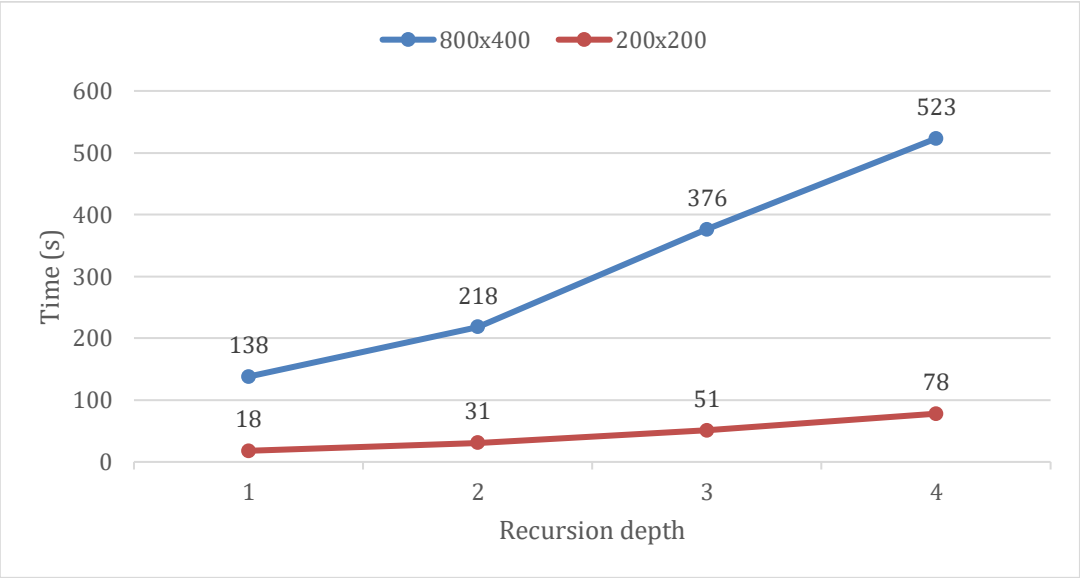


Figure 22. Side by side of scene 1, rendered with depth 1 and 5.

The resulting data is shown in the following graph:



As expected, a higher recursion level provides better visual results but at the same time can lead to an increase of 60% in render times. The scene was tested using two different resolutions: 800x400, containing a total number of 320,000 pixels, and 200x200 with brings this number down to 40,000, making a huge difference when the recursion depth is augmented.

Another render of this scene was produced in the meantime, in order to showcase the ray tracer capability of dealing with multiple light point sources, as shown in the following figure:

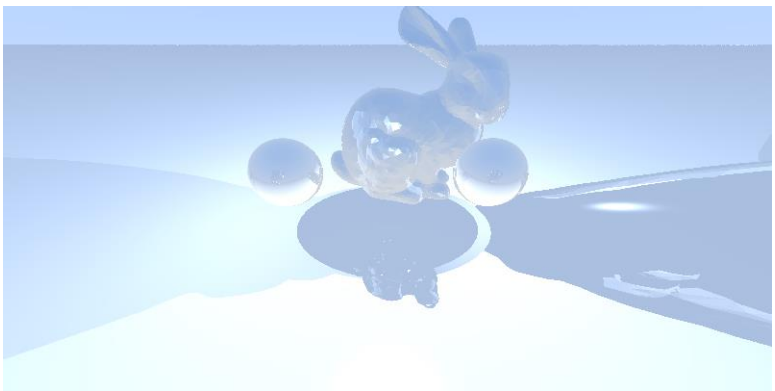
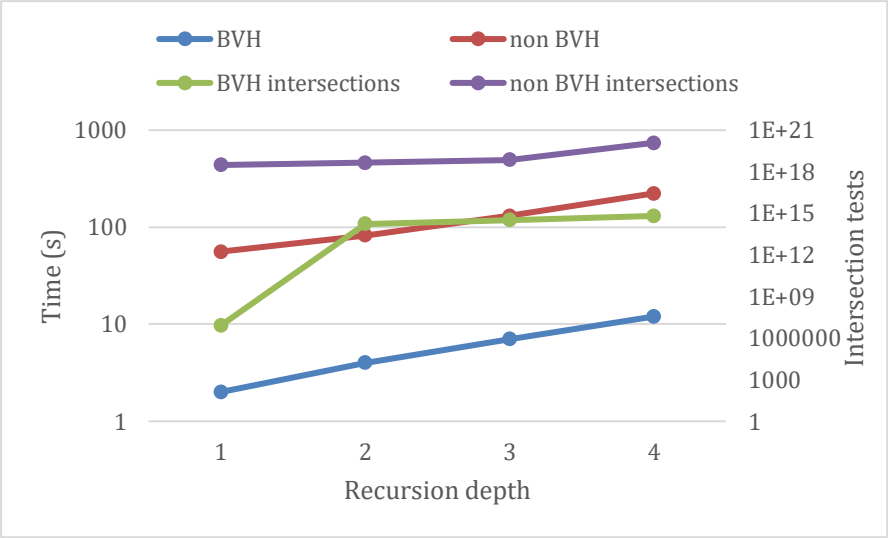


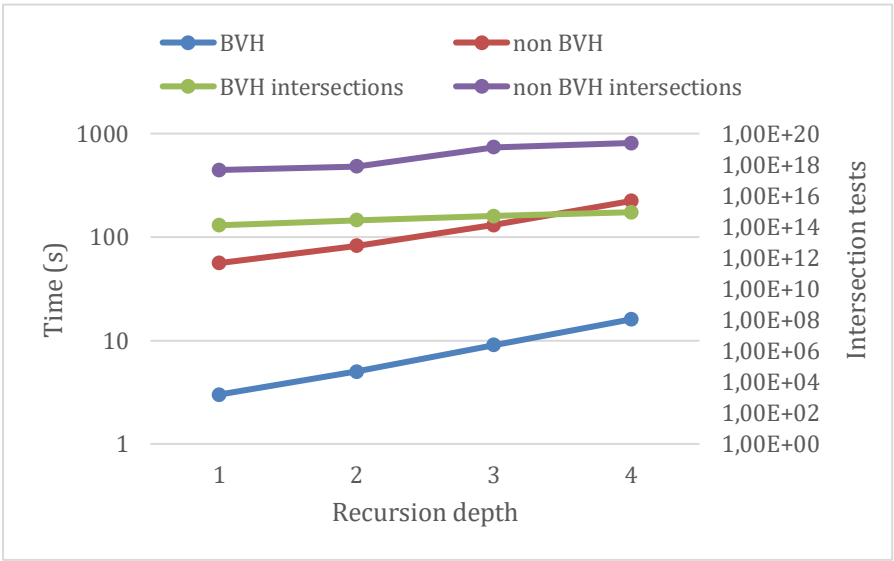
Figure 23. The Stanford Bunny rendered with two light sources.

The remaining tests were then used to check the efficacy of our BVH implementation: for the second scene we used a triangle mesh of the Stanford

bunny containing a total of 4968 triangles. It was rendered at a resolution of 800x600 and different recursion depths, using a single light source. The render times for the BVH and non-BVH versions of the ray tracer were measured, as well as the number of triangle intersection tests carried out.



In the third control scene we rendered a 5,000-triangle mesh containing the Utah teapot. The test was conducted under the same conditions as the previous scene, and the results were again inserted into a graph for visualization.



Due to the number of intersection tests being several orders of magnitude greater in the non BVH renderer, both graphs use a base-10 logarithm scale. The blue and red lines represent the render time in seconds.

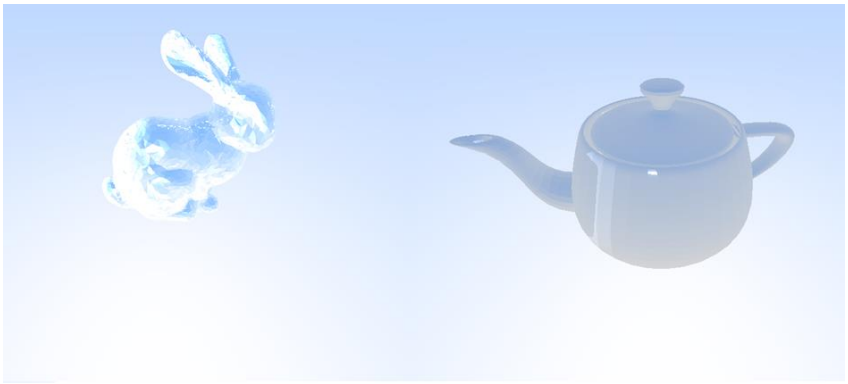


Figure 24. Scene 2 and 3 renders.

The last test scene contained the Stanford Armadillo, a collection of 345,994 triangles, rendered with a recursion depth of 1 and a resolution of 800x600 with a single light source. This test showed the capabilities of our BVH implementation: it took the non-BVH version 6930s, or nearly 2 hours, to produce an image output, while the BVH-based one archived this in 62 seconds with a total of 99,250,516 triangle intersection tests.

Figure 25. Stanford Armadillo render

4.2 Conclusion and future work

Results-wise the project was successful, given that the core features were correctly implemented, therefore meeting the expectations that were set at the beginning of the project. Texture mapping as well as the addition of a GUI were discarded due to lack of time, as they were not essential features. Initially an eventual translation of the source code from Java to C++ was considered, as this would have allowed things like operator overloading or passing parameters by reference which may have made things easier on the programming side. Using C++ could have slightly improved performance, but not substantially, and in any case it would not have been a replacement for an accelerating structure.

Another design decision taken during the development of the project was switching from kd-trees to BVH. Both of these structures have been shown to perform similarly in previous studies [29]. Bounding volume hierarchies were effectively implemented, considerably reducing render times by an average of 90% during the tests. Perhaps the most significant example of this was that the Stanford Armadillo scene took nearly 2 hours to render whereas when BVH were not used, and when used it rendered within one minute.

Generally, the development of the project was heavily delayed, resulting in two main stages of development which took place in the months of November and April. A basic render of the Stanford Bunny along with a sphere was ready by the date of the Gregynog presentation, but the Stanford Armadillo was not rendered until the implementation of BVH in late April.

After Gregynog, large parts of the code had to be debugged and rewritten. Until March no new features were added, and a big effort was then needed to deliver a solid piece of software that would meet the project aims. Build backups were periodically done throughout these stages to avoid data loss, which did not happen at any time.

Despite debugging playing a part in the development delay, there were other reasons that led to delays, namely:

- Time management
    - The course workload during the second semester was underestimated, and the initial deadlines were pushed back in a general basis.
- Coronavirus pandemic
    - In March, the World Health Organisation declared a global coronavirus pandemic which was still ongoing by the time this document was composed. It was not one of the risks assessed in the initial document as such a situation could not be expected. It created an atmosphere of uncertainty which affected productivity.

As for future work, it would be more than possible to extend the features of this ray tracer and head the project towards other techniques such as path tracing, or use it as the base for a real time ray tracer. Personally, I have found the topic of this dissertation to be very interesting and would like to keep doing research on possible ways to extend this project.

Bibliography

[1] The Official NVIDIA Blog, 2019. What's the Difference Between Ray Tracing, Rasterization? | NVIDIA Blog. [Online] Available at: https://blogs.nvidia.com/blog/2018/03/19/whats-difference-between-ray-tracing-rasterization/

[2] Pineda J., 1988. "A Parallel Algorithm for Polygon Rasterization", Apollo Computer, Inc. Chelmsfort, MA.

[3] Christensen P., 2006. “Ray Tracing for the Movie ‘Car,’” IEEE Symposium on Interactive Ray Tracing.

[4] Arvo J., 1986. “Backward Ray Tracing”, Apollo Computer, Inc. Chelmsfort, MA.

[5] Bonner M., “Bidirectional Ray Tracer Implementation”, University of British Columbia

[6] Stanford University, “Types of Ray Tracing” [Online]. Available at: https://cs.stanford.edu/people/eroberts/courses/soco/projects/1997-98/ray-tracing/types.html

[7] Orr J, 2012. “Ray Tracing Notes” Willamette University

[8] Shirley, P.'Positionable Camera' Ray Tracing in One Weekend. Amazon Digital Services LLC https://github.com/petershirley/raytracinginoneweekend

[9] Lightouse3D, “Ray-Sphere Intersection” [Online]. Available at: https://www.lighthouse3d.com/tutorials/maths/ray-sphere-intersection/

[10] Scratchapixel, "Ray-Sphere Intersection" [Online]. Available at: https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-sphere-intersection

[11] Knott R., "Triangle Convertor for Cartesian, Trilinear and Barycentric Coordinates" [Online]. Available at: http://www.maths.surrey.ac.uk/hosted-sites/R.Knott/Triangle/tricoords.html

[12] Möller, T and Trumbore, B., 1997. "Fast, Minimum Storage Ray-Triangle Intersection". Journal of Graphics Tools.

[13] Kay, T. L., and Kajiya, J. T. 1986, Ray tracing complex scenes. In Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques, ACM, New York, NY, USA, SIGGRAPH

[14] Georgios Chatzianastasiou and George A. Constantinides, 2018. "An Efficient FPGA-based Axis-Aligned Box Tool for Embedded Computer Graphics", Imperial College London, UK.

[15] Pharr M., Jakob W., and Humphreys G., 2016. Physically Based Rendering: From Theory to Implementation (3rd. ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[16] Rob Bruce, "CS-116A: Introduction to Computer Graphics" [Online]. Available at: http://www.cs.sjsu.edu/~bruce/fall_2016_cs_116a_lecture_light_and_color_part_1_of_2.html

[17] Umbaugh. S.E., 2010. Digital Image Processing and Analysis: Human and Computer Vision Applications with CVIPtools, Second Edition (2nd. ed.). CRC Press, Inc., USA.
[18] LearnOpenGL, "Basic Lighting" [Online]. Available at: https://learnopengl.com/Lighting/Basic-Lighting
[19] Whitted T., 1979. “An improved illumination model for shaded display,” Proceedings of the 6th annual conference on Computer graphics and interactive techniques - SIGGRAPH 79.
[20] Nick Evanson, "How 3D Game Rendering Works: Lighting and Shadows", The Math of Lighting [Online]. Available at: https://www.techspot.com/article/1998-how-to-3d-rendering-lighting-shadows/
[21] R. L. Cook, T. Porter, and L. Carpenter, 1984. “Distributed ray tracing,” Proceedings of the 11th annual conference on Computer graphics and interactive techniques - SIGGRAPH 84.
[22] Jensen H.W., 1996. Global Illumination using Photon Maps. In: Pueyo X., Schröder P. (eds) Rendering Techniques ’96. EGSR 1996. Eurographics. Springer, Vienna
[23] Stanford University, "Monte Carlo Path Tracing" [Online]. Available at: https://graphics.stanford.edu/courses/cs348b-01/course29.hanrahan.pdf
[24] Andrew S. Glassner (Ed.), 1989. An introduction to ray tracing. Academic Press Ltd., GBR.
[25] Scratchapixel, “Introduction to Shading” [Online]. Available at: https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-shading/reflection-refraction-fresnel
[26] Reiter Horn D., Sugerman J., Houston M, and Hanrahan P., 2007. Interactive k-d tree GPU raytracing. In Proceedings of the 2007 symposium on Interactive 3D graphics and games (I3D ’07). Association for Computing Machinery, New York, NY, USA.
[27] T. Foley and J. Sugerman, 2005. “KD-tree acceleration structures for a GPU raytracer,” Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware - HWWS 05.
[28] H. Weghorst and D. P. Greenberg, “Improved Computational Methods for Ray Tracing,” ACM Transactions on Graphics, vol. 3, no. 1.
[29] Marek Vinkler, Vlastimil Havran, and Jiří Bittner. 2014. Bounding volume hierarchies versus kd-trees on contemporary many-core architectures. In Proceedings of the 30th Spring Conference on Computer Graphics (SCCG ’14). Association for Computing Machinery, New York, NY, USA.
[30] Birchfield S., Image Processing and Analysis.
[31] Parallel BVH Construction Using Locally-Density Clustering - Scientific Figure on ResearchGate. [Online] Availabl: https://www.researchgate.net/figure/An-Example-of-The-Bounding-Volume-Hierarchy-in-2DThe-root-nodes-in-this-figure-represent_fig1_334816976