Real Time Ray Tracing
Abstract

Ray tracing has for a long time been used to create photo realistic images, but due to complex calculations done per pixel and slow hardware, the time to render a frame has been counted in hours or even days and this can be drawback if a change of a scene cannot be seen instantly. When ray tracing a frame takes less than a second to render we call it “real time ray tracing” or “interactive ray tracing” and many solutions have been developed and some involves distributing the computation to different computers interconnected in a very fast network (100 Mbit or higher). There are some drawbacks with this approach because most people do not have more than one computer and if they have, the computers are most likely not connected to each other.

Since the hardware of today is fast enough to render a pretty complex image within minutes it should be possible to achieve real time ray tracing by combining many different methods that has been developed and reduce the render time.

This work will examine what has to be sacrificed in image quality and complexity of static scenes, in order to achieve real time frame rate with ray tracing on a single computer. Some of the methods that will be covered in this work are frame optimizations, secondary rays optimization, hierarchies, culling, shadow caching, and sub sampling.

Keywords: Real time, Ray tracing, Optimization, Rendering, Sub sampling
# Table of Contents

1 Introduction......................................................................................................................... 4  
  1.1 Problem.................................................................................................................. 4  
  1.2 Goals and Criteria............................................................................................. 5  
  1.3 Motivation........................................................................................................... 5  
  1.4 Outline.............................................................................................................. 5  
2 Ray Casting and Ray Tracing...................................................................................... 7  
  2.1 Background.......................................................................................................... 7  
  2.2 Ray Casting.......................................................................................................... 7  
  2.3 Ray Tracing......................................................................................................... 8  
  2.4 Intersections....................................................................................................... 9  
  2.5 Camera............................................................................................................... 12  
3 Frame Optimizations..................................................................................................... 15  
  3.1 Entities.............................................................................................................. 15  
  3.2 Camera............................................................................................................... 15  
  3.3 Lights.................................................................................................................. 16  
  3.4 Results............................................................................................................... 17  
4 Secondary Ray Optimizations.................................................................................... 19  
  4.1 Shadows............................................................................................................ 19  
  4.2 Reflections.......................................................................................................... 21  
  4.3 Results............................................................................................................... 23  
5 Hierarchies and Culling.............................................................................................. 25  
  5.1 Bounding Volume Hierarchy............................................................................ 25  
  5.2 Octree.............................................................................................................. 26  
  5.3 Frustum View Culling....................................................................................... 28  
  5.4 Results............................................................................................................... 29  
6 Sampling........................................................................................................................ 32  
  6.1 Projective Extents............................................................................................. 32  
  6.2 Adaptive Sub Sampling................................................................................... 33  
  6.3 Results............................................................................................................... 34  
7 Conclusion...................................................................................................................... 36  
  7.1 Conclusions....................................................................................................... 36  
  7.2 Results............................................................................................................... 36  
Appendix A Scene Descriptions...................................................................................... 38  
Appendix B Rayman.......................................................................................................... 39  
Appendix C Using All Optimizations........................................................................... 40  
Bibliography....................................................................................................................... 41
1 Introduction

The main goal of ray tracing has always been to produce images that are equal or somewhat close to be as realistic as a photograph. By using more computer power, distributing the computation over networks, and by creating better ray tracing techniques we have also made it possible to create whole movies entirely by using ray tracing with a result that is almost as realistic as the world we live in.

Today ray tracers are mostly used in movie productions (Ice Age [23]) or when doing static images that should have great realism like POV-Ray [22] and this is due to the long render times and the fact that there is no cheap hardware that supports it. This is of course something that is about to change since there is a lot of research going on in real time ray tracing, both in hardware and software. Wald et al [14] has suggested a hardware architecture for real time ray tracing called SaarCOR. A very interesting project is the Avalon Project [19], which is a hardware-based ray tracing chip, although the last update on the homepage is dated to September 2002.

When it comes to software based real time ray tracers there exist some really nice projects of which Federation Against Nature’s [20] Realstorm benchmark engine looks very nice and they have also released a couple of demos of which all of them uses real time ray tracing. Lev et al [21] has made a real time ray tracing sphere engine, that is, the ray tracer is optimized to only handle spheres, which they also use to build up worlds with. They have also made a 3D-shooter game called AntiPlanet of which the game uses their ray-tracing engine.

Wald et al [13] did a ray tracer that took better advantage of the underlying hardware like caches and SIMD instructions and they even shows that their software implementation outperforms high-end graphics hardware in complex environments. They did use BSP (Binary Space Partition) for this project, which divides the scene by different splitting planes and makes the polygons to be in front of the plane or behind the plane. For more information about BSP, take a look at Ranta-Eskola [12] and Simmons [15].

In this work the term “real time ray tracing” is used to describe a frame that is ray traced under a second to match the “frames/second”, although no action is taken if the particular rendering of the frame takes more time than a second.

1.1 Problem

Due to the many calculations done per pixel, the render time has always been counted in frames-per-hour rather then frames-per-second and that is of course a drawback, especially if you want to see the result of a change instantly. When ray tracing a frame takes less than a second to render we call it “real time ray tracing” or “interactive ray tracing” and many solutions have been developed and some involves distributing the computation to different computers interconnected in a very fast network (100 Mbit or higher). There are some drawbacks with this approach because most people do not have more than one computer and if they have, the computers are most likely not connected to each other.

This thesis will address the problem:

To examine what has to be sacrificed in image quality and complexity of static scenes, in order to achieve real time rate with ray tracing on a single computer.
1.2 Goals and Criteria
To be able to solve the problem stated in the previous chapter some goals have to be set;

- To look at different parts of the ray tracing process in order to see were certain speedups can be done. This involves both image quality and scene complexity.
- The methods to be used should be simple to understand and easy to implement, since this work covers a big area and the time spend explaining in this work and implementing the routines shouldn’t take up too much.
- An own ray tracer has to be implemented since there is much easier to determine how the ray tracer should work and how the data should be handled in order to combine different optimizations together than recode an already existing ray tracer.
- To make it easy to alter scenes and configurations without the need to recompile the ray tracer in order to see if a certain optimization is better than the other.

1.2.1 Limitations
Since ray tracing involves many different parts and some of them takes too long to execute in an interactive environment and some topics are left out due to time limits, the following will not be discussed:

- Texturing.
- Different light schemes.
- Different spatial subdividing schemes like BSP\(^1\) and 3D-DDA\(^2\), since each of them is big area of research subjects.
- Specific CPU-optimization (MMX™\(^3\), SSE\(^4\), etc).
- Refraction due to complex and time-consuming calculations.
- Antialiasing.
- Dynamic environment such as moving objects and lights.

1.3 Motivation
When doing ray tracing there has always been a desire to decrease the render time, since the time it takes to render a frame of a scene can take up to a day, depending on scene complexity and computer power. This is of course not desirable if some of the objects material are wrong and needs to be updated. When the texture is correct the slow rendering process must be done again. So this work will look at different parts of the pipeline of a ray tracer to see were speed-ups can be done.

1.4 Outline
Chapter 2 describes how ray casting and ray tracing works. Intersections and the camera are explained here to.

Chapter 3 looks at frame optimizations, that is, what calculations can be moved to just to be done once per frame. The chapter ends with test results for this chapter.

\(^1\) Binary Space Partitioning
\(^2\) 3D-Digital Differential Analyzer
\(^3\) Multimedia Extension is trademark by Intel®
\(^4\) Streaming SIMD Extensions
Chapter 4 goes a bit deeper and looks at secondary rays optimizations. Shadows and reflections are covered here. The chapter ends with test results for this chapter.

Chapter 5 covers hierarchies and culling. Bounding volumes, octrees and frustum view culling are parts of this area. Ends with test results.

Chapter 6 covers samplings like projective extents and adaptive sub-sampling. As with the previous chapters, the results of the tests are presented at the end of this chapter.

Chapter 7 finishes this work with conclusions and results.

Appendix A describes all the scenes used.

Appendix B describes a little bit of the ray tracer I created.

Appendix C shows the results when all optimizations are combined.
2 Ray Casting and Ray Tracing

"Ray tracing is an algorithm that can be used to produce photo-realistic pictures of three-dimensional virtual worlds on a computer. It simulates the propagation of light through an environment by tracing rays of light in a scene to determine which objects they interact with. It also models physical properties of lights, objects and the interaction between them." - Viale [11]

2.1 Background
The process of ray tracing is very time-consuming, since many calculations are done per-pixel and the calculations is done completely by the CPU. Different methods has been used during the last two decades to speed up the render-time but nothing has come near the speed of the triangle-based rasterization methods that many 3D-cards uses today. These cards are used to visualize objects very fast so different methods and cheats has to be done to increase the realism and decrease the render time. The output of these cards have not been anything near the result of a ray tracer and this is something every game company is trying to improve but it usually results in a higher polygon count and larger detailed textures which ends up in increasing render times, with the exception of the newer cards with per-vertex shaders and even per-pixel shaders.

Ray tracing can be described as if a ray is cast from your eye, and through a pixel on the screen (assuming that you are in front of your computer screen) and out on the back of your screen to see if the ray has intersected with an object behind the screen. If one or more intersections has been found then the closest intersection is used to determine how lights and/or shadows will be on that pixel. When the shading of that pixel is done, then the process starts over with the next pixel and the whole procedure is repeated until the whole picture in done. This is of course a very simplistic description since it does not takes into account that the user usually wants to use a camera and different materials in the scene etc.

The basic philosophy of a ray tracer is very simple as can be seen in figure 2.3 (see further down) but due too the many calculations done per pixel it takes to long time to actually use it interactively. To use it interactively a common solution is to let other computers calculate different parts of the image and then send it back to a computer that puts the image together. This is of course an elegant solution although very few users have more than one computer at home and if they do, they usually don't have dual-CPUs as universities and companies have when they use this solution. When many computers are connected together there is a bandwidth problem that needs to be kept in mind, especially if the network is of 10Mbit (~1.2 Mb/sec transfer). Even if the network bandwidth is raised to 100Mbit, only about 12 uncompressed images at 640x480x24 bits can be transferred each second.

2.2 Ray Casting
In real life, photons are emitted from the sources of light (light bulbs, the sun etc) in the world. If a solid object is hit by a photon, the photon either bounces away or is absorbed, and if it for instance hits a glass or other transparent or semi-transparent medium the photon both refracts and reflects depending on what the medium is made of. To simulate this behavior, it would take an enormous amount of time to trace all the rays from the light source and only a small portion of rays would actually hit the camera (our eye in the 3D world). The solution is to reverse the process and cast rays (hence the word ray casting) from the eye out to the world and only
calculate those rays that hit an object (fig 2.2.1).

When an intersection with an object has occurred, the light and the surface material properties are used when calculating the color at that intersection point. However, when using this method there is no way to know how to calculate reflectance or know if an object or just a piece of it is in shadow since we don’t know anything about the environment. In my work I will use the term ray casting when I mean primary rays (see further down),

2.3 Ray Tracing
To know if an object is in shadow or know how to refract or reflect rays, more information about the surrounding environment is needed and this can be done by continue tracing the reflected and the refracted rays from the surface hit point until a new intersection with a surface has occurred as shown in fig 2.3.2. If the object has a material that is both reflective and refractive (like glass) there would be one ray that is reflected and another one that propagates through the object. In order to calculate the right intensity on that intersection point we would have to trace or follow both rays when they are continuing in the scene. Of course there would be more rays created when a ray intersects another object with both reflective and refractive properties. So in the end we would have many rays for each pixel to trace, although we start with only one ray from the eye.

Ray tracing is very time-consuming due to the many calculations required to get a good image result and this will increase when the resolution and/or the scene complexity is increased. To get a better understanding of the ray tracing process, the following pseudo code shows a simplified view of the kernel process;

```
Figure 2.3.3
1  func trace()
2  {
3     for every_object
4         {
5             test_intersection();
6         }
```
Due to the nature of ray tracing, the easiest way to structure the code, is to do the code recursive as can be seen in the code above since it will do the same calculation on the first ray that was cast as it will do on the reflective and/or the refractive ray. The above code is done for each pixel and is not very efficient because it will generate \[\text{numPixels} \times \text{numObjects}\] intersections per frame. The time it would take to ray trace a complete scene with this when no reflectance and refraction is used is \(O(M \times N)\), were \(M\) is the number of pixels and \(N\) is the number of objects, and therefore the time complexity is linear if \(M\) is constant and thus not very good for scenes with large object count or high resolution.

### 2.4 Intersections

To do ray tracing we need to define a ray and some object (in this case a sphere) to test for an intersection. A ray consists of a start point, a normalized direction vector and a scalar \(t\) like in figure 2.4. The \(t\) is treated as a hit point, since it describes were an intersection with an object has occurred in the direction of the ray.

![Figure 2.4.1](image)

**Fig. 2.4.1** A ray. “s” is origin of ray and “d” is the direction of the ray.

Kessler [1] define ray as;

Figure 2.4.2

1) \[ R(t) = R_0 + R_d \times t \text{ where } t \geq 0 \]

\(R_0\) = Origin of ray at \((x_0, y_0, z_0)\)
\(R_d\) = Direction of ray \([x_d, y_d, z_d]\) (unit vector).

This formula can be used when testing for intersections with a sphere surface (Kessler [1]). The surface is defined by a set of points \{(\(x_s, y_s, z_s\)\}) satisfying the equation:
2) \[(x_s - x_c)^2 + (y_s - y_c)^2 + (z_s - z_c)^2 - r^2 = 0.\]

Center of sphere: \((x_c, y_c, z_c)\)
Radius of sphere: \(r\)

Inserting the ray equation 1 into the sphere surface equation 2 gives:

3) \[(x_0 + x_d t - x_c)^2 + (y_0 + y_d t - y_c)^2 + (z_0 + z_d t - z_c)^2 - r^2 = 0.\]

⇔

\[(s + d* t - p)^2 = r^2.\]

\(p = \) position of the sphere,
\(s = R_0\)
\(d = R_d.\)

We have to solve \(t\) to know were in the direction of the ray an intersection has occurred:

LHS:

\[(s+d* t-p)(s+d* t-p)\]

⇔

\[s^2 + sdt - sp + sdt + (dt)^2 - dtp - sp + p^2\]

⇔

\[s^2 + 2sdt + p^2 - 2dtp + t^2.\]

When the equation is rewritten\(^5\)

\[t^2 + s^2 + p^2 + 2sdt - 2sp - 2dtp = r^2\]

Extracting \(t\)

\[t^2 + 2t(d(s-p)) + s^2 + p^2 + 2sp = r^2\]

⇔

\[t^2 + 2t(d(s-p)) + (s-p)(s-p) = r^2\]

Which is a second-degree equation:

\[x^2 + ax + b = 0;\]

By substituting

\[x=t\]
\[a=2(d(s-p))\]
\[b=(s-p)(s-p)-r^2\]

Solve \(x\)

\(^5\) \(d^2\) equals 1.
$$x = \frac{a}{2} \pm \sqrt{(\frac{a}{2})^2 - b}.$$ 

If the inner part of the square root argument is positive, there is an intersection otherwise the ray misses the sphere.

If $x \geq 0$ intersection with the sphere
if $x < 0$ no intersection

A second-degree equation gives two values and this corresponds to a ray that intersects in two points, when it enters and when it leaves the sphere as can be seen in fig 2.4.3. To know which of the two values to be used in the ray equation (figure 2.4.2, equation 1) compare the two values with each other and we will use the smallest one. The coordinate we get is the point of intersection on the surface of the sphere.

Fig. 2.4.3) Ray intersects the sphere where the black dots are. E is eye point.

A pseudo code for a ray/sphere intersection function:

```plaintext
Figure 2.4.4
1 float sphereIntersection( Vector s, Vector d, Vector p, float r )
2 {
3     float t, a, b, m;
4     m = subtract( s, p );
5     a = 2*dotProduct( d, m );
6     b = dotProduct( m, m ) - r*r;
7     if b <= 0
8         return -1.0;
9     float t1 = a/2 – squareRoot( b );
10    float t2 = a/2 + squareRoot( b );
11    float t = min( t1, t2 );
12    if t<=0
13        return -1.0;
14    return t;
15 }
```

This function has the start vector of the ray, the normalized direction vector of the ray, the sphere’s position vector and the sphere’s radius as parameters and returns the hit value, which is 0 if there is an intersection or -1 if there is no intersection.

As can be seen in the code, if the value of $b \leq 0$ then the ray misses the sphere and this is the same for t (see the argument above). The code won’t work when the camera is inside the sphere since t would then always be less than zero and therefore there would be no intersection.

6 If x1 and x2 is equal, the ray has intersected the tangent to the sphere.
Intersection with other primitives like planes, cones, tube, boxes etc can be done in the same way as the ray/sphere intersection although the ray intersections with other objects are slower then the ray/sphere intersection. A ray tracer has a nice advantage over scan-line rasterizers when using a primitive like a sphere, since no matter how close the camera is to the sphere, the sphere is always perfectly round and the intersection code can be done very efficient. This will of course be the same for all mathematically described objects.

2.5 Camera

A camera is used in the scene to make the interactivity easier to handle and everything the camera can see through it’s lens is what will be rendered to the pixel buffer (and shown on the screen). To know at what direction the rays should be cast, some parameters of how the camera is oriented are needed. We need to know where the camera is located in the world and at what direction the camera should be pointing at, so we need a position of the camera (called eye point) and a target of what the camera is pointing to. We also need a vector to be able to know how the camera is rolled (called the up vector), that is, what is up and down in the world. This vector is not an exact direction but it is just used as a hint and we will extract a better vector later on. The figure 2.5.1 shows how this would look like.

![Fig 2.5.1. The camera.](image)

These values are not enough to use in our environments, since we need to know more precisely how the camera is oriented to know in which direction we want to cast rays in the world. To do this we create a new coordinate system, called the viewing system, and attach it to the camera. The first thing to do is to create a direction vector and for this we use the target point and the eye point and create the unit vector \( n \) that points in the opposite direction of what the camera is targeted at (the camera is by default looking down at the negative \( n \)-vector). Using the cross product between the up vector and the \( n \) vector we create a new unit vector \( u \). This vector is pointing to the right of the camera and finally a more accurate up unit vector is created and this time we use the cross product between the \( n \) vector and the \( u \) vector and gets a vector called \( v \). The pseudo-code to create these unit vectors is then:

```c
void createUVN( Vector eye, Vector target, Vector up )
{
    n = _eye - _target;
    u = crossProduct( up, n );
    v = crossProduct( n, u );
    normalize( u );
    normalize( v );
    normalize( n );
}
```

Figure 2.5.2
The final result of how the vectors are oriented can be seen in figure 2.5.3. For a more complete guide of how to create these vector, see [7].

![Figure 2.5.3](image)

*Fig. 2.5.3* The u, v and n unit vector.

Now we have the vectors to know at what direction we should cast the rays, but we don’t know how each ray should be cast in order to simulate a real camera, that is, we need to know the limits of how far to the left, right, up and down we should cast the rays. These limits are called a camera’s view frustum, and this frustum is build up by six planes; near, far, left, right, up and down. A frustum has the shape of a pyramid with the exception that the top is shopped off. The near plane is of course the plane that is closest to the camera and the distance to the plane and the view angle (from now called field of view, FOV) is all we need right now. The frustum will be used further down when doing culling.

The FOV can be arbitrary chosen but to avoid fish-eye view a FOV of 45 degree works well. Since we know what FOV we want and the distance to the near plane (a near plane distance value of 1 works well) we can use the tan equation to get the width units (see figure 2.5.4);

![Figure 2.5.4](image)

*Fig.2.5.4* The tan equation. In the left part there is the eye point (e), the width of the screen (w) and the distance to the plane (d). The right part of the figure shows the relation between the tan equation and the left part.

To extract y;

\[ y = (\tan \alpha) \times x. \]

Since we know \( \alpha \) and x, it’s easy to calculate y.

Now that we have the w we have to get the h. To get the h we use the fact that

\[ \frac{w}{h} = \frac{\text{width}}{\text{height}}. \]

The ratio \( \frac{\text{width}}{\text{height}} \) is called the pixel ratio and it’s a value of how high a pixel should be in order to be squared. A normal value of this ratio is 4:3 (1.33, normal TV/screen resolution) or 16:9 (1.78, wide-screen).
We get
\[ h = \frac{w \times height}{width}. \]

We now have all the values to do proper steps from left to right and from top to bottom when we cast rays, and each ray that is cast, corresponds to a pixel in the pixel buffer.

The pseudo code to cast width*height rays;

Figure 2.5.5

```python
1 func castRay( int width, int height, float fov, float near, Camera camera )
2 {
3     float w = tan(fov/2) * 2 * near;
4     float h = w * height / width;
5     Ray ray;
6     ray.origin = camera.eyePoint;
7     for y = 0 to height
8         {
9             for x = 0 to width
10                {
11                 float fx = -w + x * 2 * w / width;
12                 float fy = -h + y * 2 * h / height;
13                 ray.dir = -near * camera.n + camera.u * fx + camera.v * fy;
14                 normalize( ray.dir );
15                 trace( ray );
16             }
17         }
18 }
19 }
```

On line 14, the value of the near plane distance is multiplied by \(-1\) since the vector is pointing in the opposite direction of the direction we want (see figure 2.5.2).
3 Frame Optimizations

Many calculations are done for each and every ray cast and traced. To reduce some of the calculations we can reuse values that won’t change, for example, until next frame is calculated.

When casting rays out in the scene from the eye point, all the rays (called primary rays) origins from the same point (eye) and this is were the first hit optimization would be done but these optimizations will only work for primary rays, not for secondary rays so a function for doing intersections with secondary rays must also exist.

3.1 Entities

In the pseudo code above (figure 2.4.4) the multiplication with two on line 6 and the division by two on line 11 and 12 are not necessary and are removed. The line 5 and 7 are computed every time the function is called although the ray's origin and the sphere's position never changes between each pixel. The only time a change happens is when the ray's origin moves (a camera movement) or the sphere's position changes (a translation). These calculations should be in a different function that is called once every frame just to update them. Line 11 is not necessary since t1 will always have the smallest value and therefore line 13 can also be ignored. The new pseudo code will look like the following;

```
float b, m, r2;
void sphereFrameUpdate( Vector s, Vector p, float r )
{
    r2 = r*r;
    m = subtract( s, p );
    b = dotProduct( m, m ) – r2;
}

float spherePrimaryIntersection( Vector d, float b, float m )
{
    float t, a;
    if b <= 0
        return -1.0;
    a = dotProduct( d, m );
    float t = a – squareRoot( b );
    if t<=0
        return -1.0;
    return t;
}
```

Calculations like view frustum culling (see further down) are not depending of these pre-calculations, so these variables only needs to be updated with objects that are in the frustum. Of course these optimizations can be used on other primitives as well.

3.2 Camera

Primary rays are cast with constant steps from left to right and from top to bottom. This is something that can be taken advantage of, especially since these are calculations, which is done for every pixel and can be pre-calculated in each frame. One way to solve this (Ludwig [5]), is to create a unit vector that points to the upper left corner and use that vector with two other; a vector for each scan-line step and one vector for each pixel step to the right. The following pseudo code should then be called once per frame;

```
func calculateSteps( Vector u, Vector v, Vector n, int width, int height )
```
Figure 3.2.2

1  func raytraceScene( Camera camera, int width, int height )
2  {
3      Vector scanline;
4      Ray ray;
5      ray.origin = camera.position;
6      for y = 0 to height
7        {
8          scanline = topLeft + downStep*y;
9          for x = 0 to width
10             {
11              ray.dir = normalize( scanline );
12              scanline += rightStep;
13              tracePrimary( ray );
14          }
15      }
16  }

3.3 Lights
When using light sources in ray tracing, different algorithms has been developed to more accurately simulate the physical behavior of light, and I used Phong illumination shading model for my light equation since it produce a acceptable result and does not involve too complex calculations that slow down the process. I will not discuss the light shading equation and how it works, but Johansson [8] has simple information about implementation and Hill [6] has more deep information about different light models.

To reduce the needed shading calculations in the scene, a far range can be used with the light and this is also the maximum range the light rays will be able to reach and beyond that value the light is not involved in further light calculations. When we use light sources with a far range, the distance between the object and the light is compared with the lights sources’ far range (figure 3.3.1, line 7), and if the distance between the object and the light is greater than the far range, no light is hitting the object and the object can be ignored if there is no other lights reaching it. If the distance is less than the far range it should be shaded and to decide if the intersection point on the surface is within the range the distance between the light and the intersection point is compared with the far range. When modeling the world an artist can specify manually by hand or use the modeling program (if supported) to decide if the light reaches the object by using a flag with (figure 3.3.1, line 5) the object that is later used to decide if the object really is needed in the light-object distance test.

Figure 3.3.1

1  for each light
2  {
3      for each object
4          {
5              if object.receiveLightFlag == currentLight
6                  {
7                  if obj_light_distance<=currentLight.farRange
8                      {
9                          add_to_scene
10                  }
11              }
12          }
13      }

16
This approach is very efficient especially if the scene has some bounding volumes\(^7\). If there are bounding volumes involved two things can be done. The first one is if the bounding volume is inside the far range, the whole volume is used when ray tracing, although some of the sub object are outside the far range. The second method is to further test each sub objects if they are within the far range and if they are, they are added to the list of objects that would be ray traced.

### 3.4 Results

This chapter is presenting the test results after the suggested optimizations have been done. I will discuss and compare the different results that have been achieved. I used a resolution of 320x240 and a color depth of 16 bits and with no reflections or shadows used in the scenes. Every test is run 15 times in a sequence and the time is summed for each frame and then divided by 15 to get an average time.

#### 3.4.1 Entities

In this test I will try both the un-optimized version of my code as it was in chapter 2.4 and code 2 and compare it with the optimized version I suggested in this chapter.

The scenes I use as tests are scene 1, scene 2 and scene 3 since they have different amount of spheres and one can easily see how much speed is gain. The values in the tables are in seconds and the percentage is calculated as the formula 1 – optimization/no_optimization.

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Scene 1</th>
<th>Scene 2</th>
<th>Scene 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>No optimizations</td>
<td>0,33</td>
<td>0,29</td>
<td>1,09</td>
</tr>
<tr>
<td>Optimizations</td>
<td>0,24</td>
<td>0,27</td>
<td>0,80</td>
</tr>
<tr>
<td>%</td>
<td>27,3%</td>
<td>6,9%</td>
<td>26,6%</td>
</tr>
</tbody>
</table>

Table 3.4.1

As can be seen in table 3.4.1, the performance did increase a bit more than ¼ in scenes were the sphere numbers is high. Since both the results of Scene 1 and 3 have almost the same values I guess that the optimization is about 26-27% faster than the slow version.

#### 3.4.2 Cameras

When testing the camera code it doesn’t matter which scene that is used as a reference or if the code uses the optimized entity intersection code or not. Instead of looking at how much performance is gained when using the optimized version of the camera code I will use four different resolutions; 320x240, 640x480, 800x600 and 1024x768. As in the entity result I will use the same calculation to get the gained speed in percent.

<table>
<thead>
<tr>
<th>Optimization</th>
<th>320x240</th>
<th>640x480</th>
<th>800x600</th>
<th>1024x768</th>
</tr>
</thead>
<tbody>
<tr>
<td>No optimizations</td>
<td>0,28</td>
<td>1,11</td>
<td>1,74</td>
<td>2,83</td>
</tr>
<tr>
<td>Optimizations</td>
<td>0,27</td>
<td>1,07</td>
<td>1,68</td>
<td>2,73</td>
</tr>
<tr>
<td>%</td>
<td>3,6%</td>
<td>3,6%</td>
<td>3,4%</td>
<td>3,5%</td>
</tr>
</tbody>
</table>

Table 3.4.2

As can be seen, the results when comparing the un-optimized code with the optimized, the speed-up aren’t that high and is almost equal overall the resolutions.

---

\(^7\) See chapter 6
3.4.3 Lights
In this test I will only demonstrate light range checking and only use one scene, since scene 3 is the only usable scene to test with this behavior of light. Although it’s not so useful to measure speed increase with only one scene I did it anyway since it can be good to see how much speed increase one can get when removing some scene geometry.

<table>
<thead>
<tr>
<th>Far range optimizations</th>
<th>Scene 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>No optimizations</td>
<td>1.09</td>
</tr>
<tr>
<td>Far range checking is used</td>
<td>0.79</td>
</tr>
<tr>
<td>%</td>
<td>27.5%</td>
</tr>
</tbody>
</table>

Table 3.4.3

In this scene I had three lights with different far range values, which can be seen in the left figure 3.4.1. The right figure shows the amount of the objects that was removed when far range checking was on. From start there were 122 objects (121 spheres and one plane). This was later reduced to 96 objects as can be seen in the right figure.

*Fig 3.4.1). Left, With light range checking on. Right, The same scene with no range checking.*
4 Secondary Ray Optimizations

When doing shadow tests, reflections or refraction, the rays that are thrown are called secondary rays and unfortunate the rays cannot be pre-calculated as the primary rays. The behaviors of the secondary rays are too unpredictable, so the optimization has to be done on the scene data or to alter the behavior of the rays.

4.1 Shadows

When a ray is intersecting a surface, the color at that point is determined by the amount of light from the scene and the surface properties. If the eye is looking at a solid object and the only source of light is positioned behind the object, there would be no light reaching the surface. If the angle between the hit point of the surface normal and the light point are less or equal to 90 degree, the light is involved in the shading process of the hit point on the surface. When looking at Fig. 4.1.1, the light source is obviously placed somewhere to the right of the picture and when looking at the sphere’s backsides, very little, or no light is reaching it. This picture looks good but it lacks realism since it would at least be some shadows on the plane beneath the spheres and on the spheres themselves.

![Fig. 4.1.1) One light source but no shadows.](image)

When using a single light and enabling shadows the above scene would look like this:

![Fig. 4.1.2) One light source and shadows.](image)

To know if the point of intersection is in shadow or not, we have to create a ray from the point where the ray has intersected the object to the point of the light source and do an intersection test for each object in the world to see if any object is obscuring the light. Figure 4.1.3 illustrates how two objects are obscuring the light.
The pseudo code for a shadow test with multiple lights would look like this:

```c
1 func shade( Vector hitPoint, Vector hitPointNormal )
2 {
3     Color color = 0;
4     Vector rayToLight = createReflectionVector( hitPoint, hitPointNormal );
5     for each_light
6         for each_object
7             if object.intersecting( ray )
8                 color = ..... //
9         }
10    }
11    do light calculations shading
12 }
```

Since we only want to know if an intersection has occurred or not, it is quite unnecessary to do further intersection tests with other objects and we can therefore stop doing the intersection tests. The time spend doing the shadow tests is in the worst case O(n*p) were n is number of objects and p is numbers of lights, since the routine will do intersection test with each object in the scene for each light source that exists in the scene.

### 4.1.1 Object flags
To skip not needed intersection tests, every object gets two flags, one to know if the object can receive shadows, and one to know if the object cast shadows. When doing the intersection test for an object to see if there is a shadow at the point of intersection, the time spent on doing the intersection tests against every object in the scene is time consuming. By using a flag for objects that shouldn’t receive shadows, the time doing intersections can be reduce by O(n) (best case), since we can skip all the intersection tests against the light sources. When the object that is tested for shadow intersection doesn’t cast shadows the intersection test with that object can be ignored and the next in line can be tested instead. This can further be extended to be two flags per light connected to the object and thus making it more flexible since then we can skip many object-in-shadow-tests against a particular light source and ignore shadow tests on an object if it cannot receive shadows from other objects.

### 4.1.2 Shadow Caching
When doing these tests it will still do shadow tests on each object if the object both cast and receive shadows, so to further reduce these tests we can eliminate some of the tests and still keep the quality of the image. If a point is in shadow then the point next to it is probably in shadow too, since these two points have probably the same object that is obscuring the light. By saving the object that last obscured the light(s) many intersection tests can be avoided, since we can first
test intersections against that object to know if the object is still shadowing. When the object is not obscuring the light anymore, i.e. the ray from the point of intersection to the light point is at a somewhat different place, we have to go through the object list again until there is a new object that is casting shadows or there are no more objects left. If there is a new object obscuring the light we save that object and use it next time when we do shadow tests. The code for doing this would look like this;

Figure 4.1.5

Object *lastShadowing[NUM_LIGHTS] = 0; // all elements is zero

1  func shadeWithShadowCaching( Vector hitPoint, Vector hitPointNormal )
2  {
3      Color color = 0;
4      Vector rayToLight = createReflectionVector( hitPoint, hitPointNormal );
5      for each_light
6       {
7          for each_object
8             {
9              if object.intersecting( ray )
10                 {
11                     color = ......
12                 }
13             }
14       }
15       do light calculations shading
16  }

In the code above I have an array for all the lights in the scene. Each element in the array has space for an object and if there is no object in the element, the element is zero. Before entering the function for the first time the elements in the array are set to zero and if an object is found that is obscuring a light, the element at that light is filled with the object and the next time there is a test this object is tested first.

4.1.3 Skip Pixels

If a surface point is in shadow then it is most likely that the points neighbor are too, so by using a color cache and a toggle changing between true and false every other pixel that's in the shadow (fig 4.1.6), the time spent testing when in the shadows can theoretically be cut to 1/2. When the toggle is set to true, we perform the usual steps to get the color of that pixel, but if the toggle is false we use the previous color and set pixel to that color. The only drawback with this method is the resulting picture is a bit aliased on the right side were the surface point changes from being in shadow to be outside of shadow.

![Fig. 4.1.6] Skip next pixel in shadow, indicated in white color.

4.2 Reflections

To increase realism in ray traced images, reflections can be used to simulate a mirror or another reflective surface. It is easy to extend a ray tracer to support reflection because the ray is cast from the point of intersection instead of from the eye-point but first hit optimization cannot be
used here since the reflective rays are too many, are spread over a large area and aren’t linear as the primary rays are. The pseudo code for a ray tracer with reflection could look like this;

```plaintext
int raytraceDepth = 4;

func primaryRay( Ray _ray )
{
    if object.primaryRayIntersection == true
    {
        ....
    }
    if hit
    {
        Ray newRay;
        newRay.origin = _ray.start + _ray.dir*hitPoint;
        newRay.direction = createReflectiveRay( _ray );
        rayTrace( newRay, raytraceDepth );
        shade();
    }
    /*-----------------------------*/
}

func rayTrace( Ray _ray, int _depth )
{
    if object.rayIntersection
    {
        ....
    }
    if( hit )
    {
        Ray newRay;
        newRay.origin = _ray.start + _ray.dir*hitPoint;
        newRay.direction = createReflectiveRay( _ray );
        if( raytraceDepth>0)
            rayTrace( newRay, raytraceDepth-1 );
        shade( );
    }
    /*-----------------------------*/
}
```

(Each intersection test is done on every object, although it is not shown in the code above.)

As can be seen in the above code, the two functions are very similar, but the primaryRay function has a call to an optimized intersection function, called primaryRayIntersection, and this function takes advantage of the coherence as described earlier. This cannot be used with non-primary rays, since the rays are scattered over a larger area and it's hard to predict where each ray is thrown, and this makes it difficult to speed up the ray tracer. The second call the rayIntersection function, which is a normal intersection function test without any optimization. To know when the reflection ends, a threshold (line 1) is added and as long as it is bigger than 0 (line 29) it calls itself recursively. If there wouldn't exist such a threshold the ray would continue to bounce forever (if the scene was a closed room). One can also keep count of how far the ray has been traveled around the scene and if the traveled distance is greater than a certain value the ray tracing stops, shades the pixel and continues with the next ray. A higher value generates more secondary rays and this is something that generates more intersection tests. When a secondary ray is cast, every object in the scene is tested for an intersection even though some objects are totally behind the intersection point of the object. In the worst case O(n) intersections per ray would be done and each intersection is shaded so it is indeed very expensive to use reflections, although not every object has to have a reflective surface and it's enough with a reflection depth of 1.

The author of Realstorm [12] has implemented a culler for secondary rays. The inverse of the view frustum is mapped to a plane, which is perpendicular to the surface intersection point, and
everything that is outside the frustum is discarded. Apparently this work fine, although it seems that it would be more work to do the actual culling for each secondary ray, than just doing the intersection tests on each object in the scene.

4.3 Results
I excluded the shadow flag test since I didn’t have the proper tools (like a modeling program) to set different flags on each object and if I would have done it by hand it would have take too much time. I also excluded the reflection optimization code since I couldn’t find a good way to do the optimizations. Instead I will see how much the render time will increase when adding more lights and increasing the reflection depth.

4.3.1 Shadows
In this test I started with just one light in all the scenes and enabled the shadows to see how the render time increased. I did this for two and three lights also. There is no doubt that the realism increased with more lights and made the scenes a lot more alive than with just one light or no shadows at all and this can make it up for the increased render times.
The time did increase rather much especially in scene 3 in which we have a lot of spheres (121 pcs) displayed all at once, so it’s not that strange that the results was better.

<table>
<thead>
<tr>
<th>Nr lights</th>
<th>Scene 1</th>
<th>Scene 2</th>
<th>Scene 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>One</td>
<td>0,37</td>
<td>0,26</td>
<td>1,79</td>
</tr>
<tr>
<td>Two</td>
<td>0,46</td>
<td>0,38</td>
<td>2,70</td>
</tr>
<tr>
<td>Three</td>
<td>0,54</td>
<td>0,50</td>
<td>3,87</td>
</tr>
</tbody>
</table>

Tab 4.3.1

4.3.2 Caching
This optimization is non-destructive for the image quality and it is also very effective as can be seen in the table below. It’s most effective in the last scene using three lights where it’s almost one second faster than the same scene without light caching. The question is; would it be faster if there were a Least Recently Used (LRU) cache of objects per lights instead of just one object per light? Assume that we have a cache like this with five objects in each cache per light and move all of the objects down one step in the cache when there is no object in the list that is obscuring the light and add the new one that did. This would of course only work in scenes with many objects since it would probably be a much higher overhead to go through all objects in the cache and then go through all in the scene making it take O(n)+5 per light source in the worst case.

<table>
<thead>
<tr>
<th>Nr lights</th>
<th>Scene 1</th>
<th>Scene 2</th>
<th>Scene 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>One</td>
<td>0,35</td>
<td>0,25</td>
<td>1,62</td>
</tr>
<tr>
<td>Two</td>
<td>0,40</td>
<td>0,34</td>
<td>2,13</td>
</tr>
<tr>
<td>Three</td>
<td>0,46</td>
<td>0,45</td>
<td>3,01</td>
</tr>
</tbody>
</table>

Tab 4.3.2

4.3.3 Skip pixels
Here we have the most effective optimization when comparing it with caching, but it also affects the image quality. When comparing the results in the table below with the un-optimized results, the values in the first scene with three lights are slightly slower than the first un-optimized scene with only one light and that is a good result.
The quality loss is most notable in the last scene with three lights and it is due to the distance between the spheres and their shadows. When a shadow has a sphere directly to the right (the shadow and the sphere’s left edge has contact), the left edge of the sphere will be jagged since the toggle may be indicating that we should use the color we saved to the pixel before which is the pixel color of a shadow. I couldn’t come up with a solution to this problem so there has to be a choice between speed and quality. A nice side effect when using this trick is that when there is a large area of shadows the render time will be much shorter than it would have been without shadows. This depends of course of how many intersections there has to be done in order test for a point in shadow.

When I used the two optimizations together, I got the following results;

<table>
<thead>
<tr>
<th></th>
<th>Scene 1</th>
<th>Scene 2</th>
<th>Scene 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>One</td>
<td>0,27</td>
<td>0,22</td>
<td>1,33</td>
</tr>
<tr>
<td>Two</td>
<td>0,28</td>
<td>0,25</td>
<td>1,60</td>
</tr>
<tr>
<td>Three</td>
<td>0,32</td>
<td>0,34</td>
<td>2,28</td>
</tr>
</tbody>
</table>

This result indicates that the use of shadows and more lights does not make the performance to drop too much that it would have been without any optimizations at all.

### 4.3.4 Reflection

Reflections can really enhance the visual aspect of a scene if used properly. A scene with too many objects with reflective materials can make the scene messy and one should keep in mind that it is wasteful with a reflection level too deep since the user won’t be able to see all the details anyway. If the resolution is low or when looking at a small object that reflects the rest of the scene the second level of reflective rays won’t be seen. This can actually be seen when looking at the result in the table below. There is a big speed decrease when adding one and even two levels of reflection-depths, but by the third level, the speed decrease aren’t that high and I think that this is due to the low resolution and spheres that are not too big. The scenes were rendered without shadows and used three lights.

<table>
<thead>
<tr>
<th>Scene</th>
<th>Scene 1</th>
<th>Scene 2</th>
<th>Scene 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>One</td>
<td>0,47</td>
<td>0,49</td>
<td>2,43</td>
</tr>
<tr>
<td>Two</td>
<td>0,48</td>
<td>0,50</td>
<td>2,52</td>
</tr>
<tr>
<td>Three</td>
<td>0,48</td>
<td>0,52</td>
<td>2,55</td>
</tr>
</tbody>
</table>
5 Hierarchies and Culling

In previous parts of this work I have discussed different aspects of the ray tracing process but I haven’t mentioned what can be done on the scene data. This chapter is devoted to hierarchies and view frustum culling, which both do operations on the scene data.

5.1 Bounding Volume Hierarchy

As mentioned earlier in this work, a lot of rays are cast at each object. For each ray, every object would be tested for intersections and if the objects are built up of many polygons, then there would be many intersections tests per ray even when only a few or none of the rays are intersecting the object. To reduce intersection tests, some bounds can be defined around an object. This is called a bounding volume and can be of many different shapes, but the most common is a sphere or a box. A bounding sphere (BS) is better when doing intersection tests, since the tests are fast and can be done much faster if using first hit optimization for primary rays. When a ray is cast and an intersection is made with the BS, the object would then be tested for intersection with the ray. If there is no intersection at all with the object, a false intersection has happened. A lot of these false intersections happen when the object is tall and thick, like a stick, since the stick doesn’t fit into the bounding sphere very well. To fix this a box can be used instead, because a box has a better fit around an object than a sphere does, but the time it takes to do intersections takes more time. There are two variants of the box. The first is a box that is rotated with the object, called an Object-Oriented Box (OOB). This box keeps its tightness with the object (since it is rotated with the object), but needs eight coordinates to keep track of the corners, and takes longer time to do intersections with. A better box that require less memory and has faster intersection tests is the second variant. In this case the box is aligned to the xyz-axis and is called an axis-aligned bounding box (AABB). When using this type of bounding box, only two coordinates needs to be held in memory, the maximum position and the minimum, and by the using eight variants of minimum and maximum, the maximum and minimum coordinates can make all eight corners.

![Fig. 5.1.1) A teapot with a bounding box and a bounding sphere.](image)

If the object is oriented like a stick lying along the diagonal of the box or has a shape like a tree, there are more false intersections made but the box has a better fit than a sphere in these cases anyway. As can be seen in figure 5.1.1, a BS can be used around the bounding box, to decrease the time needed for intersections with polygons. Since the scenes are is static, I will not include the OOB in the tests and if a deeper discussion of bounding volume is wanted, read Möller et al [9] and Oslejsek [10].

A bounding volume can even be used hierarchically, that is, a bounding volume surrounds one or more bounding volumes to further reduce time spend on intersections, but too many levels within the hierarchy would increase time spend on doing intersection tests. When using hierarchies of bounding volumes there is no problem to mix different bounding volumes in these hierarchies as
long as the bounding volume that surrounds don’t have a lot of unfilled spaces that are not used. These hierarchies can also be used in different stages in the ray-tracing pipeline, for example when doing culling or shadow tests. The figure 5.1.2 shows the three different intersection states.

Fig 5.1.2) A. The ray miss the bounding sphere, B the ray intersects the bounding volume, C the ray intersects the bounding volume and the box.

When the ray miss the bounding sphere it looks like the figure 5.1.2a so no further intersection tests has to be done on the object. In figure 5.1.2b, the ray intersects the bounding sphere so a further test has to be done on the box to see if the ray intersect the box, but in this case it doesn’t which results in a false intersection. Figure 5.1.2c shows a ray that first intersects the bounding sphere and when doing the ray/box intersection it also intersects the box and thus the shading process can be done.

The same would be done on hierarchies, but it would be more tests on sub bounding volumes before the intersection test would be done on the solid object.

5.2 Octree
Although bounding volumes are great performance enhancers and usable in other parts of the pipeline, they are unfortunately not enough, especially if there are very large areas with empty spaces between each solid objects. A similar approach to a hierarchy of AABB bounding volumes is octrees. An octree is a spatial subdivision technique, that is, it is used to subdivide a scene into smaller pieces but still keeping track of the unfilled spaces in a scene. Octrees are a very popular due to its regularity and its use of axis-alignness. To create an octree, we start with an AABB or an axis-aligned cube around an object, in this case the object are built up by polygons, and call this the root-node of the tree. When this is done, we create child-nodes to the root-node by dividing the box into eight equally sized boxes, making the center of the root-node a corner in the eight sub boxes. Now it is time to examine each child-node to see if the space contains some polygons. If there is a polygon that is in more than one box we subdivide the polygon so that each part of the polygon only exist in one of the boxes. Next step is to subdivide the box into eight smaller boxes and pass down the polygons to each of the smaller boxes and do the process over again. When a box doesn’t contain polygons the pointer to it is set to null. If there is a scene or a mesh built up by many polygons we can end up with many tiny triangles in the leaves if we don’t put a threshold to know when to stop. A threshold could be that after a certain level of subdividing, we don’t do any further subdividing and stop and put the polygons we have in the leaves of that box where in to or we could just stop when we have a certain number of polygons in a box.

The steps to create an octree are:

1. Create a bounding box or a cube that are axis-aligned around a mesh.
2. Pass the polygons to the box.
3. Split the polygons if necessary.
4. Test threshold, exit if it is fulfilled.
5. Create eight new sub boxes
6. Pass the polygons from the current box to the children.
7. Jump to step 3 for each of the children

Thanks to the nature of ray tracing we don’t have to split each polygon if they are in more than one box since we don’t render the polygon at once so we can ignore step three and just treat the polygon as it would only be in this box. In each leaf we will have a list with polygons belonging to that box but that is unnecessary as we have not alter the polygon list in any way so we can just have a list with index numbers to the polygon list, saved us some memory.

When casting a ray against an octree we start in the root-node of the octree and do a ray-AABB intersection test. If the ray intersects the root-node we test the eight children to see which of them the ray intersects. When we find a leaf of which the ray intersects, we will add the list for that node to the render list or if we want we can do ray-polygon intersects directly with each and every one of the polygons to see if there is a hit. If the child node we want to test is null then we know that there are no polygons and we can safely ignore that node.

The OctreeNode structure looks like this;

```c
Struct OctreeNode
{
    float min[3], max[3];          // the min and max coordinates
    boolean isLeaf;                // if true then this is a leaf
    OctreeNode node0;              // pointer to upper front left child
    OctreeNode node1;              // pointer to upper front right child
    OctreeNode node2;              // pointer to upper back left child
    OctreeNode node3;              // pointer to upper back right child
    OctreeNode node4;              // pointer to back front left child
    OctreeNode node5;              // pointer to back front right child
    OctreeNode node6;              // pointer to back back left child
    OctreeNode node7;              // pointer to back back right child
    int polygonIndexList[];        // the polygon index list, should be a // pointer to save memory
};
```

The pseudo code for a ray-octree intersection routine would look like this:

```c
float rayOctreeIntersection( Ray ray, OctreeNode octNode )
{
    t = 99999.9f;
    if( rayAABBIntersection( octNode.min, octNode.max, ray )
    {
        if( !octNode.isLeaf )
        {
            if( octNode.node0 != null )
                return rayOctreeIntersection( ray, octNode.node0 );
            if( octNode.node1 != null )
                return rayOctreeIntersection( ray, octNode.node1 );
            if( octNode.node2 != null )
                return rayOctreeIntersection( ray, octNode.node2 );
            if( octNode.node3 != null )
                return rayOctreeIntersection( ray, octNode.node3 );
            if( octNode.node4 != null )
                return rayOctreeIntersection( ray, octNode.node4 );
            if( octNode.node5 != null )
                return rayOctreeIntersection( ray, octNode.node5 );
            if( octNode.node6 != null )
                return rayOctreeIntersection( ray, octNode.node6 );
            if( octNode.node7 != null )
                return rayOctreeIntersection( ray, octNode.node7 );
        }
    }
}```
return rayOctreeIntersection( ray, octNode.node7 );
}
else
{
    for( n = 0 to octree.nPolygons )
    {
        tmp = rayPolygonIntersection( ray, polygonList[octree.polygonIndexList[n]] );
        if( tmp<t )
            t = tmp;
    }
}
return t;

This approach is easy to understand but it is not so efficient since it has to perform eight ray-
AABB intersections per box, which can be very expensive the deeper down the tree we get.

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>64</td>
<td>73</td>
</tr>
<tr>
<td>4</td>
<td>512</td>
<td>585</td>
</tr>
<tr>
<td>5</td>
<td>4096</td>
<td>4679</td>
</tr>
<tr>
<td>6</td>
<td>32768</td>
<td>37447</td>
</tr>
</tbody>
</table>

The first column in the above table is the node-level, were node-level 1 is the root-node. The
second column is number of intersections per level and the third is the total amount of
intersections performed on that level. As can be seen, there will be a lot of intersection tests even
though we probably will never go as far as level 5 or 6 in the process of creating the tree.

To make this process a lot faster one can keep track of where on the box a ray enters and exits
since we then can isolate which of the children in the tree we should visit next. [17] has a more
deeper explanation how this work.

### 5.3 Frustum View Culling

To reduce the number of intersections, or more important, to reduce the number of tests against
objects that is not visible from the eye point and therefore not needed for the tracing, view
frustum culling can be used. A frustum is pyramid with the top cut-off closest to the eye point
and is defined by six planes, upper, lower, near, far, left and right. The smallest plane in the
figure 5.3.1 is the front plane and this is were the view port will be mapped to.

![Fig 5.3.1](image)

*Fig 5.3.1) Left: Perspective view of frustum with objects. Right: Top view of same scene
Red objects are outside of the frustum, blue partly inside and green are within the frustum.*

To see how an object is related to the frustum the object has to be tested against the six-frustum
planes. When the test is done, the object can be in three different stages as seen in the fig 5.3 (left and right). If the object is totally outside the frustum then we can exclude the object from the list of objects that will be used when ray tracing. There are two objects that are totally within the frustum and should therefore be included in the render list. When an object is partly inside (as the teapot) the frustum and the object is for example an hierarchical bounding volume, further tests can be performed with this objects sub objects against the frustum as in fig 5.4 until the sub objects is totally inside, outside or the object is a mesh or a simple sphere etc. When using view frustum culling a lot of time is saved since there is no unnecessary intersection test with objects that is not seen from the eye point.

In figure 5.3.2, there is a bounding sphere (A) with two other bounding spheres inside (B and C). The bounding sphere is partially inside the frustums left plane (P) so more tests are needed to know if some of the sub-objects that are inside the frustum. As can be seen, the bounding sphere C is partially inside so we have to test the objects that are within the sphere to know if that object is inside or outside the frustum. In this case it is both inside and outside so it is added to the list of object that should be tested for intersections. The bounding sphere B is totally outside so no further tests need to be done on that object’s sub objects and the object can be ignored when doing primary ray intersection tests.

The culling can actually be done much faster than this. When using a bounding sphere and a bounding box around an object, you have to perform six sphere/plane checks and another eight point/plane checks if the bounding sphere is partly inside the frustum. Bloom [4] suggest that having a sphere around the frustum to do a rough culling just by doing sphere-to-sphere distance check many objects that lies far away from the frustum can be culled away quickly. By using this method good culling will be done at the near and the far plane but the objects at the other planes will still be inside the frustum sphere although they are outside of the sphere. A solution to this is, instead of testing directly against the frustum planes, to use a cone around the frustum (Bloom [4]) and do the tests against the cone instead. I have not implemented and tested this method because the performance hit when ray tracing (and culling etc) as many objects as 1000-3000 will make the process too slow and it's necessary to keep the object number down just in order to render within a reasonable time-frame. Möller et al [9] has a deeper discussion about view frustum culling and boxes.

**5.4 Results**

In this test suit I will again use scene 1 and 3 since it is easy to add bounding volume hierarchies to them and we have a good test values from earlier and can do good comparison between them. I will exclude the octree test since I didn’t have the time to write the basic octree code. The
scenes are rendered without shadows and reflections.

5.4.1 Bounding Volume Hierarchies
Bounding volumes can boost up the performance if it is used right, therefore I used bounding spheres and AABB when I tested the performance. The figure 5.4.1 and 5.4.2 shows how the bounding volumes would look like if they were visible to the user. An exception being the left figure of XY, were I excluded the top AABB in the picture although it is in the test.

<table>
<thead>
<tr>
<th>BVH</th>
<th>Scene 1</th>
<th>Scene 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spheres</td>
<td>0.14</td>
<td>0.71</td>
</tr>
<tr>
<td>AABB</td>
<td>0.32</td>
<td>0.67</td>
</tr>
<tr>
<td>No hierarchies</td>
<td>0.54</td>
<td>3.87</td>
</tr>
</tbody>
</table>

Tab 5.4.1

When using no hierarchy in the first scene it would be almost four times slower than when using spheres. The AABB version is faster but the preferable choice in this scene is the sphere version since it has a better adaptation and the intersection test is a lot faster. In scene 3 I have a bounding volume for each row in the scene and the AABB version is 82% faster than the original with no hierarchy at all. Obviously as can be seen in right fig 5.4.2 the bounding spheres are overlapping each other (hence the “pill” shape) and there is a lot of empty spaces around each row.

5.4.2 Frustum View Culling
View frustum culling can greatly increase the speed of rendering since we can ignore objects that are not visible and many games use this technique. My idea was that it would really increase the
speed in ray tracing also but it really didn’t work as I expected which will be seen further down. To be able to test culling I moved the camera in the scenes so that some of the objects were out of the frustum. The figure 5.4.3 and 5.4.4 show how the camera was positioned.

![Scene 1](image1.png)

*Fig 5.4.3) Scene 1, 29 visible objects (the camera pitched down a bit).*

![Scene 3](image2.png)

*Fig 5.4.4) Scene 3, 104 visible objects.*

The hierarchies take much more time to render than without. The reason to this, I believe, is that the culling code takes more time than I expected because it adds and removes objects to the render list (a list which have all the objects that should be rendered) and performs 12 box-frustum view tests and 121 sphere-frustum tests. Another reason for this is that my code does culling on partly visible bounding volume (see figure 5.3.2), and if the volume has 12 solid objects and 8 of them are in view frustum then they will be added to the render list and rendered one by one. A possible solution to this would be to have a flag indicating that this is a bounding volume with only solid objects in it and shall be added to the render list as it is.

<table>
<thead>
<tr>
<th></th>
<th>Scene 1</th>
<th>Scene 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>No FVC</td>
<td>0.25</td>
<td>0.86</td>
</tr>
<tr>
<td>FVC</td>
<td>0.26</td>
<td>0.97</td>
</tr>
<tr>
<td>No FVC but hierarchies</td>
<td>0.10</td>
<td>0.71</td>
</tr>
<tr>
<td>FVC and hierarchies</td>
<td>0.26</td>
<td>1.06</td>
</tr>
</tbody>
</table>

Tab 5.4.2
6 Sampling

6.1 Projective Extents

Even though many objects that were not in the frustum were culled there would still be many intersection tests with the objects that are in the frustum, and this is due to the way the ray tracer works. Even if the scene data is structured hierarchical or spatially, each ray will still be tested against every object that is visible. If a cluster of objects would be visible in the lower right corner in the view field, and about 25 percent of the rays would actually be hits on the objects, all the other rays would be wasteful even though the objects are in bounding volumes since there would still be 75 percent intersections that are not needed. A great solution to this problem is to use what is called projection extents of the objects. To get the extents of the object we just use the AABB (for example) of the object and transform the coordinates to screen coordinates to get the result as in figure 6.1.1. Since we use a box there will be eight coordinates but we will only save just two of them, the minimum and the maximum coordinate since they are enough to create a two-dimensional box on the screen.

A ray is mapped to a pixel position in a pixel buffer\(^8\) and to know when to cast a ray we just do some simple comparison between screen coordinates of the rays and the projective extents to that object. If the ray coordinate is within the boundaries of the projection extents then a ray can be cast and tested against the object.

![Fig. 6.1.1] Projective Extents together with the ray traced spheres.

A great way too really increase the speed with this method is to only do the calculations and save the coordinates once every frame instead of recalculate them for each ray.

The following pseudo code shows how the function would look like:

```
Figure 6.1.2
1  Rectangle rectList;
2  
3  void createProjectionExtents( Object objects )
4  {
5      clear( rectList );
6      for each object
7      {
8          float min[2] = {SCREENWIDTH, SCREENHEIGHT };
9          float max[2] = { -1, -1 };
10         for each AABB coordinate
11         {
12             if current_coordinate < min
13                 min = current_coordinate;
14             if current_coordinate > max
15                 max = current_coordinate;
16         }
17         rectList.add( min, max, currentObject );
18      }
19  }
```

\(^8\) See chapter 3 and figure 3.2.7
The function is called once every frame and will fill a list called `rectList` with min and max coordinates and which object that is occupying that space. When casting rays we just have to compare the current coordinate with the coordinates in the rectList and if the coordinate is within the boundaries we know that there is an object there. Of course that depends on what the selected coordinates to calculate the projective extents are chosen to be. One more thing can be done, and this is more a theoretical approach since I haven’t tried it my self. When the projection extents are created, a depth value is also saved together with the min and max values and this value is based on the closest z-coordinate to the camera (as an example) and when all the coordinates are calculated the list is sorted according to these value with the smallest value at the beginning of the list. When traversing the rectList we know that if there is a hit with an object there is no need to do further tests on the list since we know we have found the closest hit.

### 6.2 Adaptive Sub Sampling

When casting rays at a scene, many shaded pixels will have the same color or at least very small differences in color values with some exceptions. Figure 6.2.1 is from a scene that has a plane behind a highlighted sphere, the plane will have almost the same color value over the whole picture and it would be unnecessary to render each pixel of it since the result would be the same as the surrounding pixels.

![Fig. 6.2.1) A plane with a highlighted sphere in front of it.](image)

The only time the color will change (a lot) is on the edge of the sphere and at the highlight as can be seen in figure 6.2.1. By using this knowledge we can divide the whole image in a grid and only render every Nth pixel square and compare each corner with each other in that square. If the hit is at the same object then only comparison between color values are done and if the difference is greater than a certain threshold then the square is divided in four smaller part. When there is a different object hit as in the upper right corner of picture Z then a comparison of color values is done and in this case the difference in color values is less than the threshold then no further division is done and the area that have not been further subdivided, will be filled with an interpolated rectangle. This rectangle can be drawn using a 3D hardware to speed up the rendering instead of doing software based interpolation.

![Fig 6.2.2) Left. The same sphere as in 6.2.1 but with no fill switched on. Right. Each square is filled with the medium color value that the four pixels had.](image)
The drawback with this method is that if an object would be less than Nth pixel in width or height and lying between the sub sampling pixels then the object would not be seen since we would cast rays around it, but that is a small price to pay since we can reduce the needed intersections so much. When using this method the intersections can be kept to very few as long as there are not too much color changes in the scene.

### 6.3 Results

The following optimizations are the ones that makes real time ray tracing worth doing although there are some quality loss, since they can really speed up the rendering. When using adaptive sub sampling a small loss of quality must be accepted. In my ray tracer I didn’t implement the interpolation code to the sub sampling part since it would be to slow if I didn’t do it in MMX or in another code that is closer to the hardware than C.

#### 6.3.1 Projection Extents

This is a good optimization that can really increase render speed since we know almost exactly were each object is on the screen and can therefore only cast rays when needed. Figure 6.3.1 shows the projection extents in all the three scenes.

![Projection extents visible in all the scenes.](image)

When looking at table 6.3.1 we can see that there is a great speed-up in all the scenes except for the middle but this is due to loose object extents (there are based on a bounding box around each object). If the object extents would have better fit then the results would have be better also. The results is very satisfying and the results could probably be better if some sort of depth sorting between the objects was done as I suggested in chapter 6.1.

<table>
<thead>
<tr>
<th>Projection Extents</th>
<th>Scene 1</th>
<th>Scene 2</th>
<th>Scene 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Without</td>
<td>0.26</td>
<td>0.27</td>
<td>0.85</td>
</tr>
<tr>
<td>With</td>
<td>0.17</td>
<td>0.25</td>
<td>0.58</td>
</tr>
<tr>
<td>%</td>
<td>34.6%</td>
<td>7.4%</td>
<td>31.7%</td>
</tr>
</tbody>
</table>

Tab 6.3.1

#### 6.3.2 Adaptive Sub Sampling

This optimization is great although it doesn’t decrease the render time as expected. When ray tracing the first scene without any optimizations turned on, there are 76800 rays thrown from the eye-point and out in the scene. When sub sampling is used there are only 24668 (32%) rays thrown. This is of course a very good thing, but this method has some drawbacks and one of them was mentioned before (image quality) the other one is the fact that the more the intensity of the pixel changes the more branches there will be. This is also the reason for the somewhat lower results than expected and there are probably many cache-misses that rise when entering this routine. Ludwig [5] suggest that when the threshold is exceeded the whole square is ray traced.
instead of dividing it in sub squares, and I think I’m ready too agree on that as long as the square is not too big.

Fig 6.3.2) Using sub sampling but disabled the color interpolation.

The figure 6.3.2 shows how the three scenes look like when interpolation is turned off. The dots in the image are the corners of which we have done the samplings and found that it doesn’t exceed the threshold. Due to the somewhat small picture it looks like there are almost only large areas outside the spheres that are of almost the same color but that is not the case.

<table>
<thead>
<tr>
<th>Subsampling</th>
<th>Scene 1</th>
<th>Scene 2</th>
<th>Scene 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Without</td>
<td>0.26</td>
<td>0.28</td>
<td>0.86</td>
</tr>
<tr>
<td>With</td>
<td>0.12</td>
<td>0.10</td>
<td>0.52</td>
</tr>
<tr>
<td>%</td>
<td>53.8%</td>
<td>64.3%</td>
<td>39.5%</td>
</tr>
</tbody>
</table>

Tab 6.3.2

When looking at the table above it’s easy to see that this optimization is indeed very effective especially in the first and the second scene. This is of course due to the large areas were the color doesn’t change too much and the object are bigger and closer to the camera. The threshold value I used when the colors were compared with each other was at ~10%, that is, if the average colors were above 10% of the upper left corner color, the square was sub divided into four new ones. Of course it would be less sub dividing if the color threshold were higher but this value was what I found being good.
7 Conclusion

7.1 Conclusions
Although there are many algorithms and methods left out, I believe that the results are very satisfying, knowing that there is so much more that can further be optimized to get a higher frame rate. When trying to look at such a huge subject as ray tracing is, it is easy to forget and ignore certain parts that some other would find more interesting than the ones that were used instead. When I did the tests of the ray tracer I noticed that the code for doing ray-intersections with triangles wasn’t as fast as I wished and unfortunately I hadn’t the time to write new code and the test scene had to be removed. When looking at the different parts of a ray tracer one can easily see that there can be much improvement without altering the basic structure of the ray tracer and I’m quite surprised that I could get it quite fast although the code was written do be able to handle the ray tracing with different optimizations turned on or off.

7.2 Results
The problem addressed by this thesis was:
To examine what has to be sacrificed in image quality and complexity of static scenes, in order to achieve real time rate with ray tracing on a single computer.

As I stated before in a previous chapter I had four goals that I draw up in order to make this work done. The first goal was to look at different parts of the ray tracer to see what can be done to decrease render time and I looked at rays, intersections, camera, lights, sampling, scenes, etc to see were optimizations could be done to reach the goal. The second goal was easier to reach since all the algorithms and methods used and implemented in this thesis were pretty simple to do. The hardest goal of them all was the third goal since a whole ray tracer with many different optimizations options had to be coded. This also involved scene management and different sampling algorithms. To handle scenes and different configurations a good file format had to be choosen and the best for this type of application was XML. By using XML files a scene can be described like this:

```
<scene>
  <camera>
    <eye x="0" y="130" z="-300"/>
    <target x="0" y="0" z="0"/>
    <up x="0" y="1" z="1"/>
    <plane near="1.0" far="2000.0"/>
    <fov value="60.0"/>
  </camera>
  <light>
    <position x="0" y="200" z="-200"/>
    <ambient r="0.0" g="0.0" b="0.0"/>
    <diffuse r="0.3" g="0.3" b="0.3"/>
    <specular r="0.8" g="0.8" b="0.8"/>
    <attenuation constant="1" linear="0" quadric="0"/>
    <range value="1000" /> 
  </light>
  <plane>
    <position x="0" y="-10" z="0"/>
    <normal a="1" b="0" c="0"/>
    <material name="gold"/>
  </plane>
</scene>
```

The configuration file looks like this:
Without the fourth goal the process of recompiling every time a change in the scene has occurred the time to do all the tests would have taken to long time so the goal was reach.

Since all the four goals were successfully fulfilled, the work is done.
Appendix A Scene Descriptions

To compare the algorithms used in this work I’m using three different scenes with a static camera positioned at the same place for each of the algorithms in the scenes. The first scene consists of 37 spheres and has three lights, as can be seen in picture A.1, the second scene has nine spheres, a plane and three lights. In the last scene I created 121 spheres structured equally in rows and columns on a plane and added three light sources. It is always hard to select good test scenes and in this case it can be discussed why I selected these three cases, but I tried to make the scenes somewhat interesting to look at and at the same time keep them simple and easy to create and change since I changed the scenes, using only text files.

![Figure A.1). Left, scene 1. Middle, scene 2. Right scene 3.](image)

Through out the tests I will run every scene with the same resolution and at the same color depth and each test is done 15 times in sequence to ensure that I will get a proper average time. To see how long time a scene takes to run I start the clock at the beginning of the frame and stops it when the whole picture has been converted and shown to the user.
Appendix B Rayman

For this project I decided to implement my own ray tracer called Rayman since very few ray tracers with free source code is based on real time ray tracing and when I found one, the source was often to tightly bound to a specific type of ray tracing and would make it very hard to generalize for my needs. Due to the somewhat different parts in my report I decided to keep my code as generalized and clean as possible and therefore used C++ throughout the project since ray tracers have a natural aptitude for object-oriented programming. In my code I used some inheritance and my own vector class so some performance loss should be kept in mind. The pseudo code used in this work is loosely based on the C language but the algorithms are not tied to that particular language. For the developing and testing I used a Pentium III, 863 MHz with 512 Mb RAM.

My ray tracer uses floats in every part of the pipeline until the result is converted to a 32 bits integer and written down to a color buffer. When a frame is done I convert (if needed) the color buffer to a screen buffer, which are then shown to the user. If the screen color depth is of another bit depth then the color buffer there will be some time performance loss when converting the frame.
Appendix C Using All Optimizations

In table C.1 there are the final results with all the different optimizations turned on. In the table there are four test results. When using optimization but with no quality loss I used shadows with only caching turned on and projection extents. With all the optimization that infects the image quality turned off I used shadows with only skip pixel and adaptive sub sampling with a square width of 4 pixels. All test results had a reflection depth of one and no light range checking on. Only in the first test (no optimization) the bounding volume hierarchy was not used and I didn’t use the view frustum culling in any of the test.

<table>
<thead>
<tr>
<th>Optimizations</th>
<th>Scene 1</th>
<th>Scene 2</th>
<th>Scene 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>0,70</td>
<td>0,64</td>
<td>5,42</td>
</tr>
<tr>
<td>No quality loss</td>
<td>0,40</td>
<td>0,41</td>
<td>1,93</td>
</tr>
<tr>
<td>Quality loss</td>
<td>0,34</td>
<td>0,20</td>
<td>1,77</td>
</tr>
<tr>
<td>Both combined</td>
<td>0,30</td>
<td>0,19</td>
<td>1,51</td>
</tr>
<tr>
<td>%</td>
<td>56,4%</td>
<td>71,0%</td>
<td>72,2%</td>
</tr>
</tbody>
</table>

Table C.1

When looking at the results we can see that there are some great improvements done and in the last test scene the values are almost down to an interactive rate, this can probably be achieved with an optimized code using assembler and rewriting it to pure C code instead.
Bibliography


[10] Radek Oslejsek, Bounding Volume Hierarchy Analysis (Case Study), Faculty of Informatics Masaryk University Brno, Czech Republic, December 2000.


[23] Ice Age, Ice Age The Movie, www.iceagemovie.com, 2002