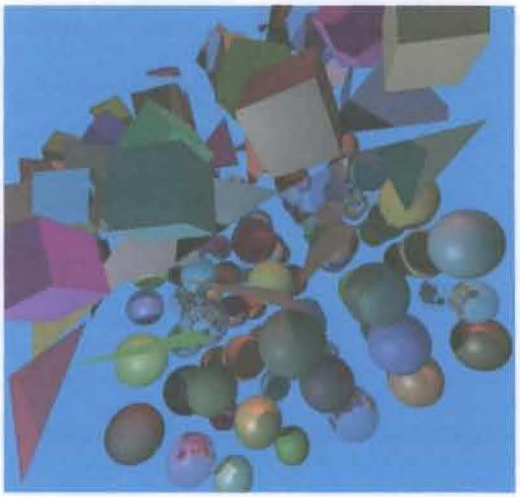
3-29-2005

Using Binary Space Subdivision to Optimize Primary Ray Processing in Ray-Tracing Algorithms

Mark Portolese '05
Illinois Wesleyan University



USING BINARY SPACE SUBDIVISION TO OPTIMIZE PRIMARY RAY PROCESSING IN RAY-TRACING ALGORITHMS

Abstract: Ray-tracing algorithms have the potential to create extremely realistic three-dimensional computer graphics. The basic idea is to trace light rays from the user through the computer screen into the hypothetical three-dimensional world. This is done to determine what objects should be displayed on the screen. Furthermore, these rays are traced back to the light sources themselves to determine shading and other photorealistic effects. However, without optimization these algorithms are slow and impractical. This paper explores the use of the classic binary space subdivision algorithm in order to speed up the process. Binary space subdivision is the use of binary trees to recursively partition the screen into rectangular areas which are then rendered separately. The algorithms were implemented using C++. The use of binary space subdivision dramatically improved the speed of the implementation in most cases, resulting in a doubled or tripled frame rate under favorable circumstances.

Mark Portolese
March 29, 2005

Advisor: Dr. Weiyu Zhu

Committee Members:
Dr. Susan Anderson-Freed
Dr. Zahia Drici
Dr. Hans-Jörg Tiede
Dr. Weiyu Zhu

Table of Contents

1. Introduction

The most popular method used today for generating three-dimensional computer graphics uses some form of polygonal rendering system. In such a system, simple polygons (such as triangles) are used in combination to describe more complex shapes. In order to render an image, the polygons visible in the image are drawn in a specific order. Shading might be determined by calculating the shade of each vertex in the polygon and interpolating the values over the entire polygon. Alternatively, shading might be calculated on a pixel-by-pixel basis. The main advantage of such a system is its speed - with hardware acceleration, many thousands of triangles can be displayed on a screen at an impressive frame rate.

Ray-tracing is an alternative method for generating three-dimensional computer graphics. Its main advantage is its photorealism - its range of visual effects far surpass that of mere shading. However, in its traditional form it is much slower than a polygonal rendering system. Ray tracing performance is rarely measured in seconds per frame, let alone frames per second. In fact, measuring performance in minutes, hours, or even *days* per frame is not unusual in the most complex examples. In comparison to a polygonal rendering system, a ray-tracer appears to have no place in time-intensive applications.

The main purpose of this thesis is to begin to refute this dismal claim. As computers and programming methods continue to improve, so can the performance of a ray-tracing implementation. This thesis introduces one such optimization that can be used to speed up the rendering process. It is the first step towards building a ray-tracer that can compete with the performance of a polygonal rendering system while retaining the photorealistic qualities that set ray-tracing apart from any other rendering method.

1.1 Why Ray-Trace?

First of all, it is important to note that pretty much everything that ray-tracing can do can also be reproduced on a polygonal rendering system. Furthermore, a polygonal rendering system can easily display far more shapes and objects than any given ray-tracer in the same amount of time even without hardware acceleration. However, it does so at a loss of realism. With additional computation additional realism can be obtained, though the computational demands can grow so high that a ray-tracer may as well be used. Therefore, this is a discussion of the advantages a ray-tracer can *easily* implement, rather than what a ray-tracer can do and what a polygonal rendering system cannot.

The first advantage of using ray-tracers is that curved surfaces are perfect (see Figure 1). In a polygonal based rendering system, a sphere may be drawn by describing the sphere as hundreds or even thousands of triangles. However, no matter how many triangles are used, the sphere will always lose its perfectly curved surfaces if it is large enough. In ray-tracing, this problem does not exist simply because a sphere is not described through triangles. A sphere is simply a sphere, and it is drawn as such. This extends to all other curved shapes that can be described mathematically (general quadrics, lathes, etc.).

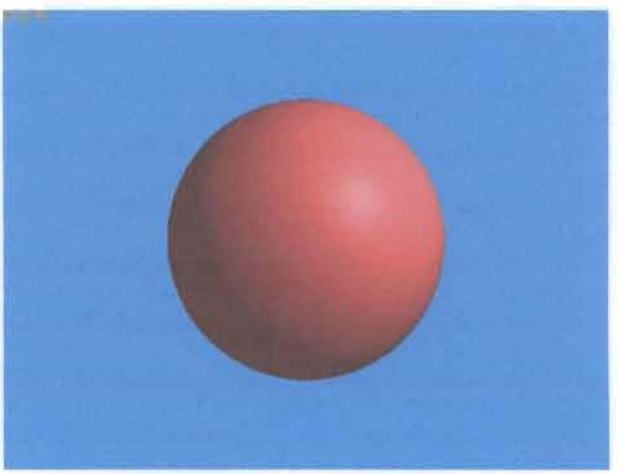


Figure 1: A ray-tracer can easily display a perfectly curved sphere. This image was created with the implementation described in this thesis.

A second advantage would be how easily a ray-tracer can manipulate shadows

(see Figure 2). In this area, a ray-tracer is very flexible. It can handle multiple light sources, which do not need to be fixed. This allows a single object to cast multiple shadows onto other objects. Since the light sources themselves may move, the shadows in turn can easily change location. Light sources themselves can be of many types - area light, volume light, spot-light, etc. They may even have assigned colors. Such flexibility is rarely seen in a polygonal rendering system.

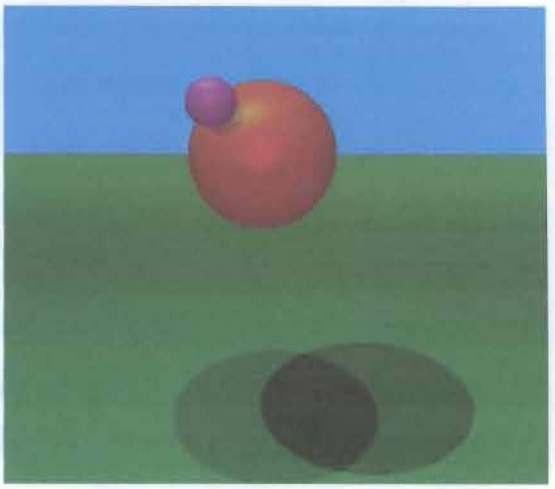


Figure 2: This image uses three light sources, all of which are different colors. It was also created with this thesis's implementation.

The photorealistic effects expand beyond simple shadow manipulation (see Figure 3). True reflectivity and transmittance are natural extensions to the set of ray-tracing algorithms (though sadly it is incredibly time-consuming to process). Such things have also been implemented using polygonal rendering systems, though often the examples are seen only from a single angle or on few objects. Ray-tracing does not have this limitation.

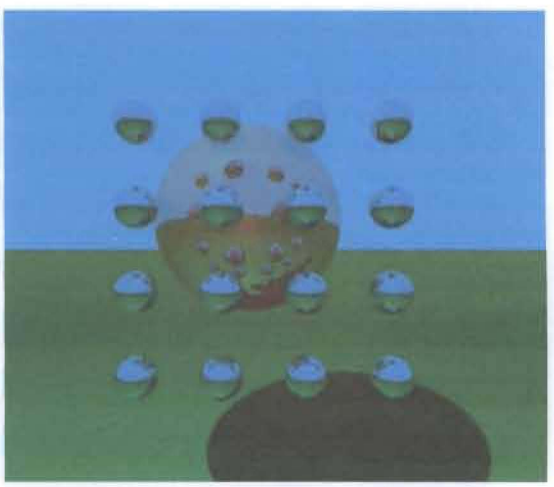


Figure 3: This image demonstrates the reflective surface ability of the implementation described in this thesis.

The final reason to use ray-tracing is to look at the final result (see Figures 4 and 5). Absolutely breathtaking images have been generated by using ray-tracers that are not

only photorealistic, but are also quite beautiful. Many examples of these images can be found at the International Ray-Tracing Competition web-site (http://www.irtc.org) and at the POV-RAY web-site (http://www.povray.org).

Figure 4: "Puddle" by Michael Hunter. This image was rendered using 3D Studio Max and won second place at the International Ray-Tracing Competition in Sept./Oct. 2003. Used with permission. © 2003 Interactive Technologies, Inc.

Figure 5: "Evening at the River" by Christoph Gerber. This image was rendered using POV-RAY and is in the POV-RAY Hall of Fame. Used with permission.

1.2 Ray-tracing in Time-Intensive Applications

Ray-tracing for its own sake has its own obvious set of benefits. However, this thesis views the set of ray-tracing algorithms for use in time-intensive environments. In other words, ray-tracing should be *fast*. The idea seems to be at first laughable and completely infeasible. For example, Michael Hunter's excellent image (see Figure 4) took five hours and 23 minutes to generate. Polygonal rendering systems have been known to generate dozens of images *per second* which contain thousands of triangles. How could a ray-tracer possibly compare to such performance ability?

First of all, it is important to note that polygonal rendering systems can only display one type of polygon (triangles, typically). All other objects must be described through these polygons. A quadric or lathe object can be approximated by using many triangles, though in a ray-tracer, such things are single objects. Therefore, it may not be necessary to process thousands of objects in a ray-tracer, depending on the specific scene.

Second, the idea that a ray-tracer is slow is thirty years old. Computers continue to become faster and faster, and with optimization, it no longer seems completely infeasible to think of a time-intensive ray-tracer.

This thesis focuses on one such optimization which is known as the Binary Space Partition algorithm. It has the advantage of accuracy, meaning that it does not sacrifice the image output. It simply allows the image to be rendered at a faster rate. It is not unique to ray-tracing - it has been used successfully in many different forms of three-dimensional computer graphics. It is very successful with the set of ray-tracing algorithms described in the next chapter.

1.3 Conventions

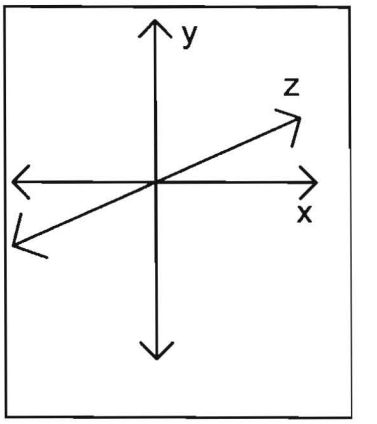


Figure 6: The three-dimensional coordinate system used in this thesis.

Before discussing the algorithms themselves, it is necessary to go over the mathematical conventions that are used in this thesis. The three-dimensional coordinate system shown in Figure 6 is used. Although the traditional mathematical system is slightly different, this is the system most often found in practice. Informally, it can be described as right in the positive x direction, left in the negative x direction, up in the positive y direction, down in the negative y direction, forward in the positive z direction, and backward in the negative z direction.

A three-dimensional point (or vector) has the notation <x,y,z>. A two-dimensional point has the notation (x, y). Vectors as variables are always described in bold-face. For example, vector **v** can be assigned the direction <5,-2,0>, which signifies five units to the right, two units down, and no direction at all in the z direction. Scalars are italicized, as in scalar *s*. A dot product is shown by a dot (·) while a cross product uses an ordinary x. The magnitude or length of the vector **v** will be shown with the notation |**v**|.

Functions have the following syntax:

$$f_a(i_0, i_1, i_2 ... i_{m-1}) = (o_0, o_1, o_2 ... o_{n-1})$$

This is function *a*, which has *m* inputs and *n* outputs. The inputs and outputs may be of any type. If there is only a single input or output, the corresponding parentheses are omitted.

2. Ray-Tracing Algorithm

A ray-tracer is not a single algorithm, but rather a set of algorithms which combine together in order to form a cohesive whole. Ideally, the most efficient algorithms (in terms of both memory allocation and time) should be used. However, the topic of optimization is postponed until the next chapter. For now, it is enough that the algorithms work correctly in a fairly efficient manner.

Overall, the main idea of ray-tracing stems from observations of the ordinary world. In a given room there are many objects that are shaped differently. There are also sources of light, such as a lamp or the Sun. These light sources shine millions of light rays into the room in every direction. These light rays bounce off objects and some of them eventually enter the eye of the observer. The color of the light rays upon entering the eye is the color perceived by the observer.

Ray-tracing works by modeling this world of light rays in reverse. It would be impractical to trace rays from the light source since so very few of these rays enter the eye of the observer. Rather, rays are traced in the opposite direction - from the observer out into the three-dimensional world - in order to find what objects the observer can see. Algorithmically this is described as tracing rays from the observer (who is known as the user) through the pixels of the computer screen into the hypothetical three-dimensional world. These rays are known as *primary rays.* If the rays do not hit a particular object, the pixel is colored black or set to some other predetermined background color. Otherwise, if the rays do hit a particular object, the relationships between the observer, the object in question, and the light sources in the hypothetical world (which is known as the scene) are used in order to determine the color of the pixel [1]. Since rays are traced backward

from the user into the scene, this method of ray-tracing is often called *backward ray-tracing*.

This chapter explores this process in more detail. It begins with a discussion of the projection model used in this implementation, then turns to the topics of the main ray-tracing loop followed by looking at specific algorithms used for specific ray-object calculation. The implemented photorealistic effects are then gone over in detail. The chapter ends with a discussion of topics for further research, a listing of a more detailed version of the main ray-tracing loop, and a look at the efficiency of ray-tracing in general.

2.1 Perspective Projection

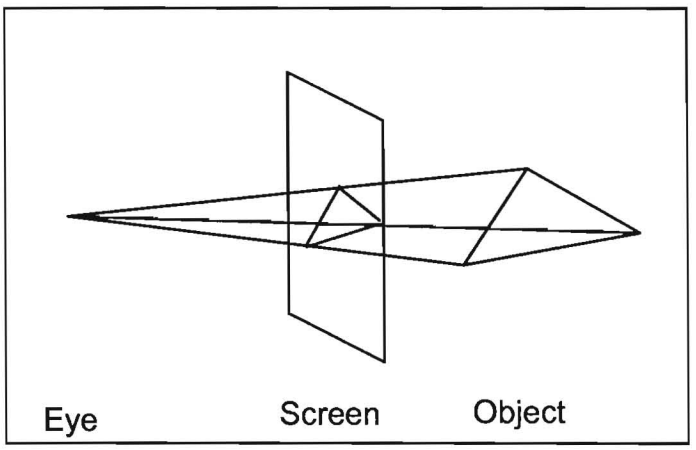


Figure 7: If line segments are drawn from the eye to the object to be displayed, where those line segments intersect the plane of the computer screen is where the object should be displayed on the screen.

A projection is not unique to ray-tracing. Rather, a projection can be defined as a method of transforming points of a given dimension to those of a lesser dimension. In this case, the transformation from a three-dimensional point to a two-dimensional one is the only projection required. The task is to transform three-dimensional points in the hypothetical three-dimensional world to two-dimensional points on the screen. The planar perspective projection model was used in the implementation and therefore will be described here.

The planar perspective projection model is called planar because we are projecting onto a plane (the computer screen itself). The "perspective" element comes from the fact that there exists a center point of perspective which, in the case of three-dimensional graphics, corresponds exactly to the location of the eye of the user [2]. If one were to draw a line segment from this point of perspective through the plane of the computer screen to an object in the hypothetical world, this line segment intersects the plane of the computer screen at a specific point. The object should be drawn on the screen at this specific point. If this is done, the computer screen will give the illusion of three-dimensions (see Figure 7).

While perspective projection can be described through matrices, the complete

transform is unnecessary for the simple ray-tracer described in this thesis. Rather, all that will be discussed is two functions. The first function translates three-dimensional points into their two-dimensional versions, while the second one translates a two-dimensional point into a ray that represents all the possible locations of the two-dimensional point in the three-dimensional world. For simplicity, it is assumed that the user is at location <0,0,0>, while the center of the computer screen is at <0,0,*f*>.

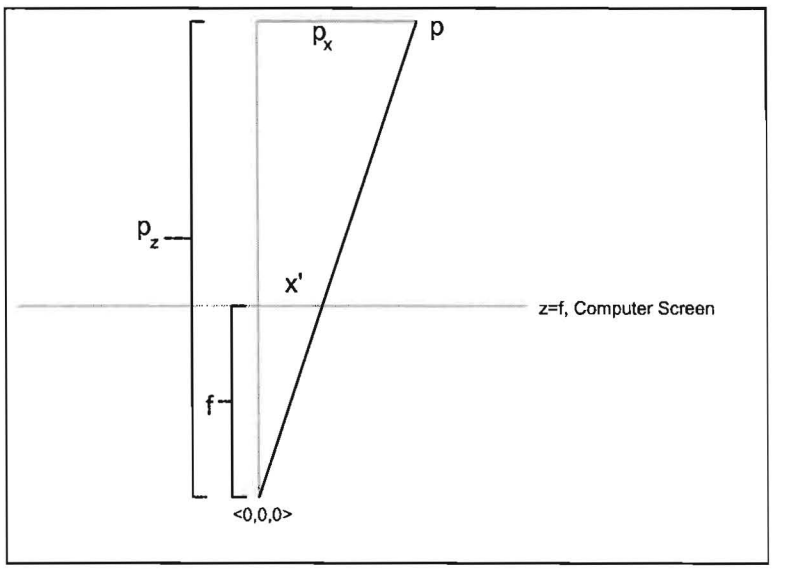


Figure 8: The point **p** to be drawn is at (p_x, p_y). The task is to find x'.

The first function is easy to solve. This function will be denoted as f_1. Its inputs are the point **p** and the variable *f*, while its output is the desired (x',y') ordered pair:

$$f_1(\mathbf{p}, f) = (x',y')$$

Solving this equation is very straightforward. Figure 8 is an overhead view of the problem. The line segment drawn from the origin to point **p** intersects the screen at the point (x', y'). From this figure, it is easy to see that x' can be solved by using the concept of similar triangles. Therefore:

$$\frac{p_x}{p_z} = \frac{x'}{z}$$

$$x' = \frac{p_x f}{p_z}.$$

Using a side-view of the problem, y' can also be found in a similar manner. The final solution is:

$$f_1(\mathbf{p}, f) = \left(\frac{p_x f}{p_z}, \frac{p_y f}{p_z} \right).$$

The second function can be thought of as the inverse of the first function. It is true that it is impossible to determine a three-dimensional point given a single two-dimensional point, though it is possible to determine a ray that passes through all of the possible three-dimensional points. That is the task at hand. This function is defined as follows:

$$f_2(x', y') = (\mathbf{o}, \mathbf{d})$$

Despite the notation, this function returns a ray - which is defined as the origin **o** and the direction **d**. Since one of the basic assumptions of this thesis is that the user is located at the origin, **o** will always be set to <0,0,0>. In order to solve for **d**, the following is observed:

$$\mathbf{o} + k\mathbf{d} = \langle x', y', f \rangle$$

This is the ray equation. Note that the ray passes through the given two-dimensional point at z=*f*. Since this is a ray, any value can be used for *k*. If it is assumed that *k*=1 and **o** is at the origin, then **d**=<*x'*,*y'*,*f*>. Therefore, the final solution is:

$$f_2(x', y') = (\langle 0,0,0 \rangle, \langle x', y', f \rangle).$$

These two equations are the only ones required to use perspective projection in the set of ray-tracing algorithms described throughout the rest of this chapter.

2.2 Main Ray-Tracing Loop

In this section, the overall structure of a basic ray-tracer is discussed. The following pseudocode is repeated for every pixel on the screen in order to generate an image (see Figures 9 and 10). This main loop will be listed in a more detailed form in this chapter after the discussion of the photorealistic effects. However, this is more than enough to start with:

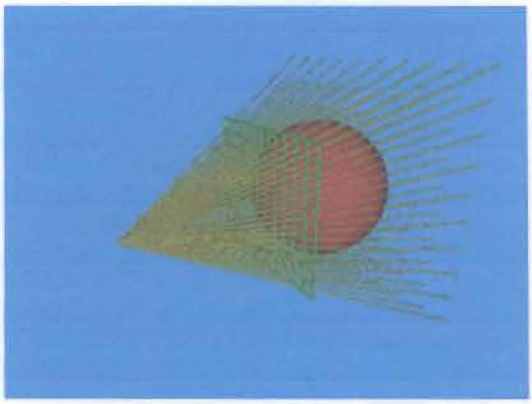


Figure 9: Rays are sent from the eye through every pixel on the screen (the green grid) into the hypothetical world in order to generate an image. This image was generated using POV-RAY.

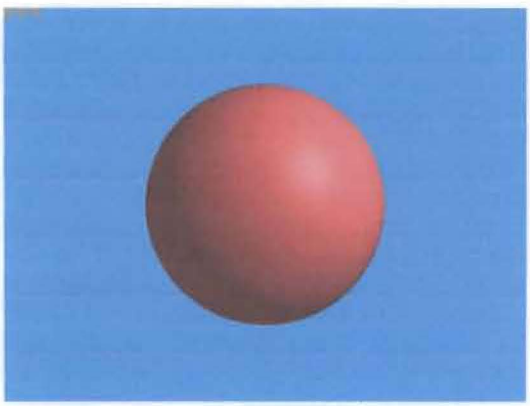


Figure 10: The image rendered using the system in Figure 9. This image was generated using the implementation described in this thesis.

```
Do
  Compute the primary ray for the current pixel
    using function f2
  Find the closest object that intersects the
    primary ray
    If there is no object that intersects the primary
      ray, assign the background color to the
      current pixel
    Else determine shading and other
      photorealistic effects to determine the pixel
      color
Loop for every pixel on the screen
```

This is the essential loop for a ray-tracer in its most basic form. Each step will now be discussed in turn.

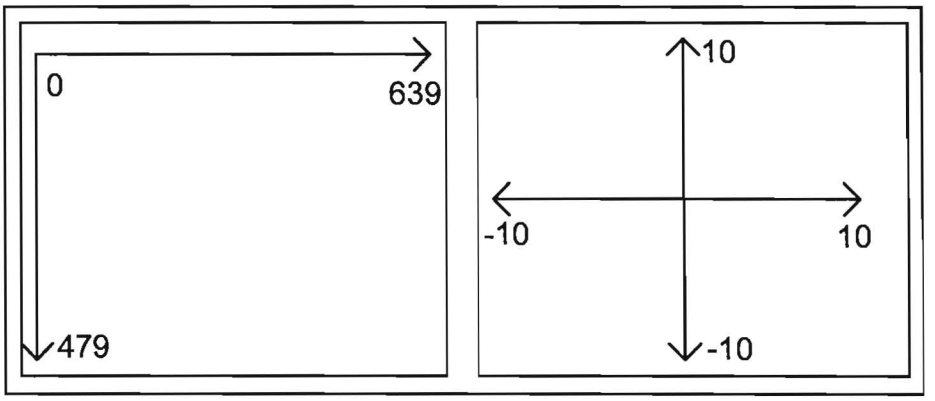


Figure 11: A typical mapping problem. The screen is described in pixels which range from 0 to 639 and 0 to 479, left to right, top to bottom. The world may be described from -10 to 10, left to right, bottom to top. It is necessary to convert from the pixel coordinates to the coordinate system usable by the ray-tracer.

Compute the primary ray for the current pixel using function f_2

This step is almost as simple as plugging in the current pixel's values into the function as described in the previous section. However, the pixel coordinate may need to be mapped and scaled to the correct values first. When using pixels, integers are typically used to refer to each pixel with the origin placed in the upper-left corner. In contrast, when using world coordinates, the origin is typically in the center and is scaled through decimal numbers rather than unique integers (see Figure 11). It is necessary to have a way to convert from one system to another in order to make sure the correct primary ray is traced. The direction for the primary ray will probably need to be normalized as well, depending on the specific ray-object intersection algorithms used.

Find the closest object that intersects the primary ray

The brute-force method of solving this step is to loop through each of the objects in the scene in order to see if they intersect the primary ray at all. If no object intersects the primary ray, then the pixel is assigned the background color. If only a single object

intersects the primary ray, it is used for the shading and photorealistic calculations performed in the next step. If multiple objects intersect the primary ray, only the closest one is used for the shading and photorealistic calculations. The specific ray-object intersection algorithms used in this implementation are given their own subsections later in this chapter.

The optimization described in the third chapter offers an alternative method for this step. With this alternative method it is no longer necessary to loop through *all* of the objects in the scene. This saves processing time and allows the ray-tracer to work much more efficiently no matter which specific ray-object intersection algorithms are used.

Determine shading and other photorealistic effects to determine pixel color

This step turns out to be extraordinarily complex. It is enough to say that for this thesis's implementation the ambient-diffuse model was used for shading along with specular highlighting and reflective surfaces for photorealistic effects. Each of these concepts will be given their own subsection. Furthermore, additional photorealistic effects not implemented here are covered in the section titled "Topics for Further Research."

2.3 Simple Ray-Object Intersection Algorithms

This section features algorithms for determining the intersections between rays and several specific objects: spheres, planes, box objects, and triangles. The main question here is where exactly along the ray the object in question intersects the ray (if at all). For now, all that really matters is the closest point of intersection.

Recall that a ray is described by two vectors: **o** and **d**. **o** represents the origin the ray while **d** represents the direction of the ray. For all primary rays, **o** is the origin. However, in later sections of this chapter, other types of rays will be discussed and therefore it will not be assumed that **o** is always the origin. **d** should be normalized before being used with any the algorithms that are described here. The equation of a ray is as follows:

$$\mathbf{o} + k\mathbf{d} = \mathbf{p}$$

p is a point along the ray while k is the scalar required to reach that point along the ray. Since all primary rays share the origin as their **o**, the object that intersects at the lowest k-value should be rendered into the current pixel. Therefore, if the object does not intersect the ray at all, the algorithms that follow assign ∞ to k.

2.3.1 Sphere-Ray Intersection Algorithm

This geometric solution is more efficient than the straightforward general quadric solution to the problem. It is adapted from [3]. A sphere is designated as having a center **c** and a radius *r*. Therefore, the function that encapsulates this algorithm is this:

$$f_3(\mathbf{o}, \mathbf{d}, \mathbf{c}, r) = k$$

Here is the pseudocode for the algorithm:

1 : $\mathbf{l} = \mathbf{c} - \mathbf{o}$
2 : $d = \mathbf{l} \cdot \mathbf{d}$
3 : $l^2 = \mathbf{l} \cdot \mathbf{l}$
4 : if $(d < 0$ and $l^2 > r^2)$return ∞
5 : $m^2 = l^2 - d^2$
6 : if $(m^2 > r^2)$return ∞
7 : $q = \sqrt{r^2 - m^2}$
8 : if $(l^2 > r^2)$return $d - q$
else return $d + q$

A ray either intersects a sphere at two points, one point, or does not intersect the sphere at all. If the ray intersects at only one point, it is either tangent to the sphere or originates within the sphere itself. If it intersects at two points, it is necessary to determine which point produces a smaller *k*-value.

l as computed in line 1 is the vector from the ray origin to the center of the sphere. *d* is the projection of **l** onto **d**. If line 4 holds, the sphere is behind the ray origin and therefore the ray cannot possibly intersect the sphere. *m* is the distance from the sphere center to the projection, though it is never calculated directly. Rather, its square is calculated and is compared to the square of the radius of the sphere in line 6. If this line is false, the ray definitely intersects the sphere. Determining where requires *q*, which is

calculated in line 7. Line 8 determines which intersection point to return. Figure 12 shows an example with a ray that successfully hits the sphere and another ray that completely misses the sphere.

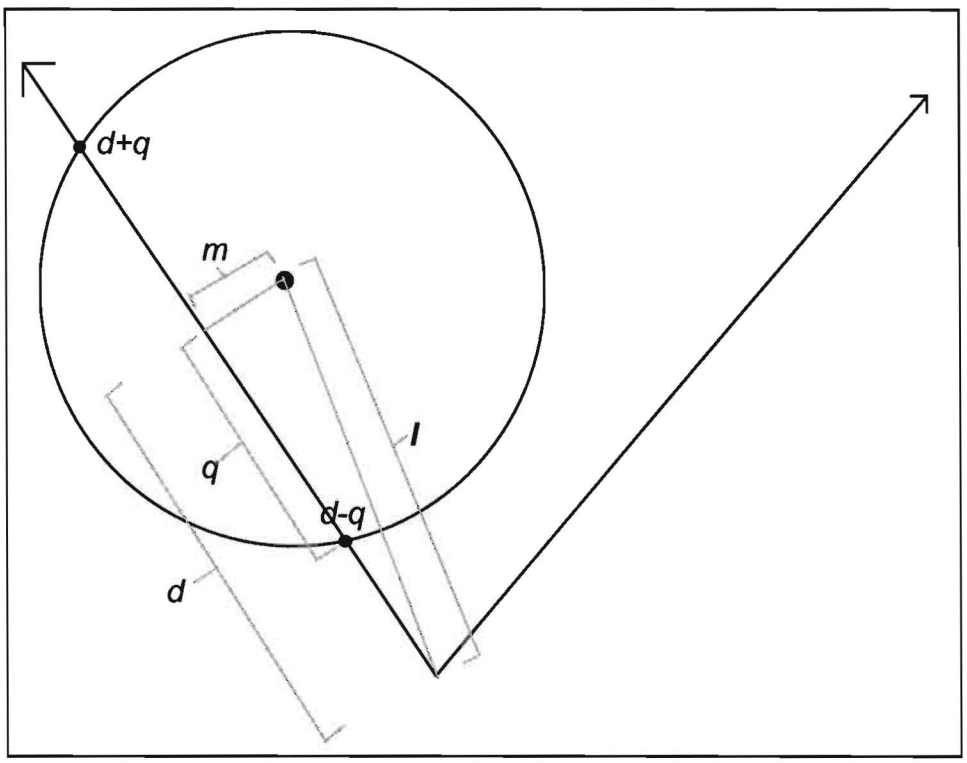


Figure 12: The right ray misses the sphere entirely since its *d* is less than zero. The left ray intersects the sphere at both *d-q* and *d+q*. The algorithm returns *d-q* since that is closest to the ray origin.

2.3.2 Plane-Ray Intersection Algorithm

The most straightforward solution to this is to plug the ray equation directly into the plane equation [1]. A plane is defined by a normal vector **n** and a constant scalar d with the following relationship:

$$\mathbf{n} \cdot \mathbf{p} + d = 0$$

Therefore, by plugging in the ray equation, we solve for k:

$$\mathbf{n} \cdot (\mathbf{o} + k\mathbf{d}) + d = 0$$
$$k = \frac{-d - (\mathbf{n} \cdot \mathbf{o})}{\mathbf{n} \cdot \mathbf{d}}$$

From this, we can conclude two simple rejection tests. First of all, if the denominator is zero, the ray does not intersect the plane. Second, if k is negative the plane is behind the ray and therefore does not intersect it. From this, the following algorithm is derived:

$f_4(\mathbf{o}, \mathbf{d}, \mathbf{n}, d) = k$
$1 : t_1 = \mathbf{n} \cdot \mathbf{d}$
$2 : \text{if}(t_1 = 0) \text{return } \infty$
$3 : t_2 = -d - (\mathbf{n} \cdot \mathbf{o})$
$4 : \text{if}(t_1 \geq 0 \text{ and } t_2 < 0) \text{return } \infty$
$5 : \text{if}(t_1 < 0 \text{ and } t_2 \geq 0) \text{return } \infty$
$6 : \text{return } \frac{t_2}{t_1}$

This algorithm uses a simple sign test in order to delay the division as long as possible. Lines 4 and 5 state that if the numerator and denominator are of opposite signs, k will be negative and therefore the ray does not intersect the plane.

2.3.3 Box-Ray Intersection Algorithm

A box is a complex thing in a three-dimensional world if you want to be able to view it from any orientation. It can be described by a center point **c**, three normalized directions $\mathbf{v_u}$, $\mathbf{v_v}$, and $\mathbf{v_w}$ along with three half-lengths which are l_u, l_v, and l_w. The vectors are all orthogonal to each other (see Figure 13). The advantage of using this description is that a box does not need to be aligned with the standard x, y, and z axes. Any orientation that preserves the orthogonal quality of the three direction vectors is valid. This allows a box to be rotated arbitrarily.

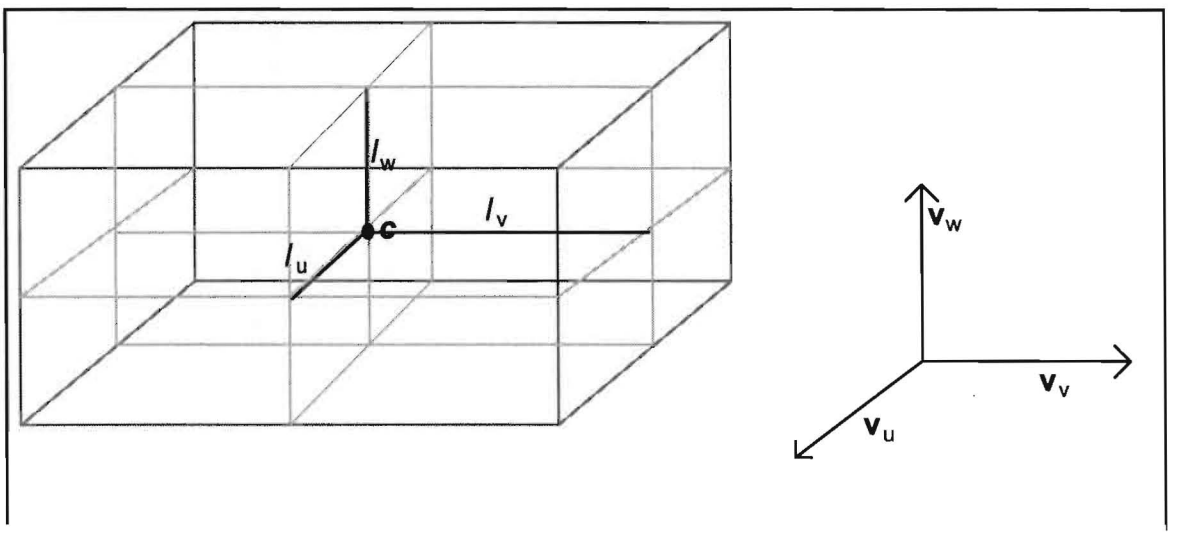


Figure 13: A box. The outermost box is the one displayed on the screen. The three direction vectors $\mathbf{v_u}$ $\mathbf{v_v}$, and $\mathbf{v_w}$ correspond to the three half-lengths l_u, l_v, and l_w.

The following algorithm was also taken from [3]. This algorithm views a box as three pairs of parallel planes. A k-value is defined as the k scalar from the ray equation where the ray intersects a given plane (if it intersects the plane at all). k^{max} and k^{min} are calculated by looking at all three pairs of parallel planes that describe the box. Each pair of parallel planes has two k values, which for simplicity are known as k^{less} and k^{more} .k^{less} is the smaller k value. The smallest of the three k^{more} scalars is called k^{max}. Similarly, the

largest of the three k^{less} scalars is called k^{min}. If k^{min} is smaller than k^{max}, the ray intersects the box. If not, the ray misses. This solution is not intuitive and therefore is difficult to understand. The simpler two-dimensional case of this is depicted in Figure 14. The pseudocode for the algorithm can be found on the next page.

The correct way to read line 4 is to execute lines 5 through 14 three times - once with *u* substituted for *i*, once with *v* substituted for *i*, and once with *w* substituted for *i*. Since *f* is a scalar, the expression |*f*| in line 7 is the absolute value of *f*, rather than the length of a vector.

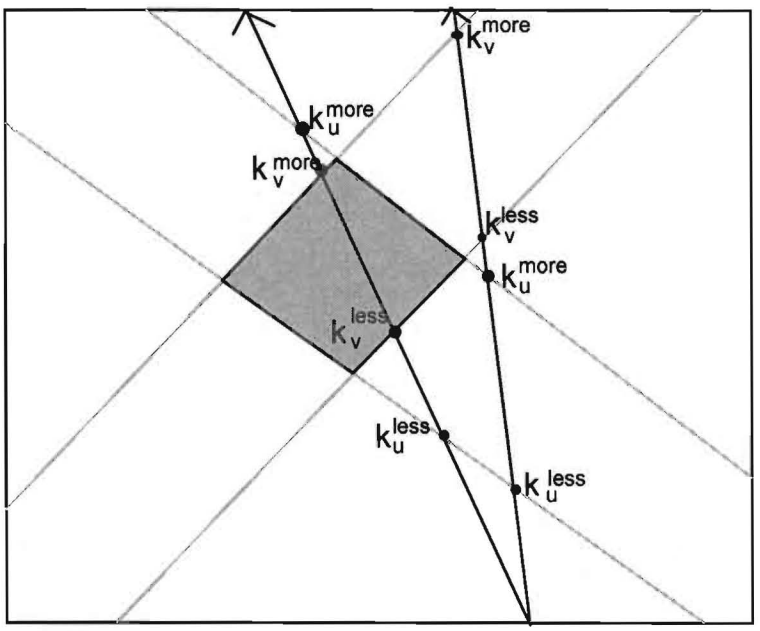


Figure 14: The simpler two-dimensional case. The left ray intersects the shaded box because $k^{min}<k^{max}$. In this case, k_v^{less} is k^{min} and k_v^{more} is k^{max}. The right ray misses the box because $k^{max}>k^{min}$. In this case, k_v^{less} is k^{min} while k_u^{more} is k^{max}. Since k^{min} is further along the ray than k^{max}, $k^{max}>k^{min}$ and the ray does not intersect the box. This extends directly to the three-dimensional version described in the pseudocode.

The purpose of line 7 is to check to see if the ray direction is parallel to the normal direction of the current plane. If this happens, no reasonable intersection can occur (and dividing by *f* would probably result in a divide-by-zero error without the ε check). The *else* portion of this is found in line 15, which tests to see if the ray is outside the space denoted by the two parallel planes. If so, the ray misses the box.

$f_5(\mathbf{o}, \mathbf{d}, \mathbf{c}, \mathbf{v}_u, \mathbf{v}_v, \mathbf{v}_w, l_u, l_v, l_w) = k$

1: $k^{\min} = -\infty$

2: $k^{\max} = \infty$

3: $\mathbf{p} = \mathbf{c} - \mathbf{o}$

4: for each $i \in \{u, v, w\}$

5: $\quad e = \mathbf{v}_i \cdot \mathbf{p}$

6: $\quad f = \mathbf{v}_i \cdot \mathbf{p}$

7: $\quad \text{if}(|f| > \varepsilon)$

8: $\qquad k_{less} = \dfrac{(e + l_i)}{f}$

9: $\qquad k_{more} = \dfrac{(e - l_i)}{f}$

10: $\qquad \text{if}(k_{less} > k_{more})\text{swap}(k_{less}, k_{more})$

11: $\qquad \text{if}(k_{less} > k^{\min})k^{\min} = k_{less}$

12: $\qquad \text{if}(k_{more} < k^{\max})k^{\max} = k_{more}$

13: $\qquad \text{if}(k^{\min} > k^{\max})\text{return } \infty$

14: $\qquad \text{if}(k^{\max} < 0)\text{return } \infty$

15: $\quad \text{else if}(-e - l_i > 0 \text{ or } -e + l_i < 0)\text{return } \infty$

16: end for loop

17: $\text{if}(k^{\min} > 0)\text{return } k^{\min}$

$\quad \text{else return } k^{\max}$

If the conditional in line 7 is successful, lines 8 through 14 are executed. Lines 8-10 determine the *k*-values for the current pair of parallel planes, while lines 11 and 12 update k^{min} and k^{max} as necessary. If line 13 returns, then the ray misses the box as described in Figure 11. If line 14 returns, the box is behind the ray origin and therefore misses the box. Line 17 ensures that only a valid (positive) *k*-value is returned since by this point it has been determined that the ray does indeed hit the box.

2.3.4 Triangle-Ray Intersection Algorithm

Ray-tracers can of course display triangles just as accurately as any polygonal rendering system given a working triangle-ray intersection algorithm. This specific algorithm was also taken from [3]. Its advantages include that it does not require the precomputed normal of the triangle and that it produces the barycentric coordinates (u,v). The relationship between these barycentric coordinates and the (x,y) coordinate system is shown in Figure 15. These coordinates are very useful in texture mapping, which is one of the concepts described in Section 2.5. The triangle itself is described through the points $\mathbf{v}_0$, $\mathbf{v}_1$, and $\mathbf{v}_2$. Here is the pseudocode:

$f_6(\mathbf{o},\mathbf{d},\mathbf{v}_0,\mathbf{v}_1,\mathbf{v}_2) = k$
$1: \mathbf{e}_1 = \mathbf{v}_1 - \mathbf{v}_0$
$2: \mathbf{e}_2 = \mathbf{v}_2 - \mathbf{v}_0$
$3: \mathbf{p} = \mathbf{d} \times \mathbf{e}_2$
$4: a = \mathbf{e}_1 \cdot \mathbf{p}$
$5: \text{if}(a > -\varepsilon \text{ and } a < \varepsilon) \text{return } \infty$
$6: f = \frac{1}{a}$
$7: \mathbf{s} = \mathbf{o} - \mathbf{v}_0$
$8: u = f(\mathbf{s} \cdot \mathbf{p})$
$9: \text{if}(u < 0 \text{ or } u > 1) \text{return } \infty$
$10: \mathbf{q} = \mathbf{s} \times \mathbf{e}_1$
$11: v = f(\mathbf{d} \cdot \mathbf{q})$
$12: \text{if}(v < 0 \text{ or } u + v > 1) \text{return } \infty$
$13: \text{return } f(\mathbf{e}_2 \cdot \mathbf{q})$

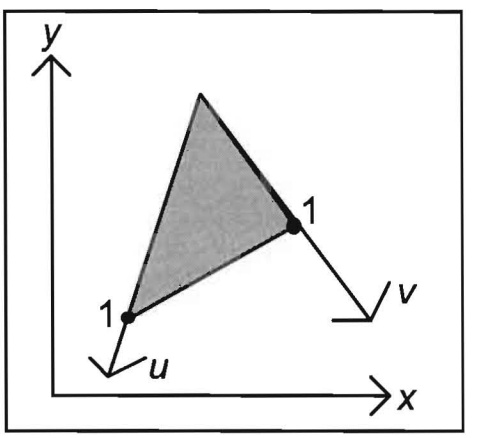


Figure 15: A two-dimensional triangle showing its u and v vectors. A k-value of 1 along either of these vectors is equal to an edge of the triangle. In the three-dimensional extension, the only difference is that u and v now have three dimensions to their directions.

This algorithm assumes that a point $\mathbf{t}(u,v)$ on a triangle has the following relationship to $\mathbf{v}_0$, $\mathbf{v}_1$, and $\mathbf{v}_2$:

$$\mathbf{t}(u,v) = (1-u-v)\mathbf{v}_0 + u\mathbf{v}_1 + v\mathbf{v}_2$$
$$u \geq 0$$
$$v \geq 0$$
$$u+v \leq 1$$

Plugging in the ray equation, the above becomes this:

$$\mathbf{o} + k\mathbf{d} = (1-u-v)\mathbf{v}_0 + u\mathbf{v}_1 + v\mathbf{v}_2$$

With the following variable assignments:

$$\mathbf{e}_1 = \mathbf{v}_1 - \mathbf{v}_0$$
$$\mathbf{e}_2 = \mathbf{v}_2 - \mathbf{v}_0$$
$$\mathbf{s} = \mathbf{o} - \mathbf{v}_0$$

The above can eventually be solved for k, u, and v by using Cramer's Rule. Here is the final result:

$$\begin{pmatrix} k \\ u \\ v \end{pmatrix} = \frac{1}{(\mathbf{d} \times \mathbf{e}_2) \cdot \mathbf{e}_1} \begin{pmatrix} (\mathbf{s} \times \mathbf{e}_1) \cdot \mathbf{e}_2 \\ (\mathbf{d} \times \mathbf{e}_2) \cdot \mathbf{s} \\ (\mathbf{s} \times \mathbf{e}_1) \cdot \mathbf{d} \end{pmatrix}$$

Along with some additional variable assignments and tests, the pseudocode above is almost an exact reproduction of the result.

2.4 Photorealistic Effects

It is finally time to turn to the photorealistic effects that ray-tracers are known for. In this entire section, it is assumed that there is only one type of light source: a point light source. This invisible light source is a point in space that radiates equally in all directions. It does not diminish in strength with increased distance. It may be assigned a color and there may be more than one in the scene. Furthermore, this light source may be placed anywhere in the scene.

This section also introduces the concept of a *secondary ray*. Primary rays are not the only ones cast in a ray-tracer. Secondary rays often have their origin placed where the primary ray intersects an object in the scene. They often point to a light source, another object, or simply serve as the normal for the object they start from. They are very useful with perfecting the photorealistic effects discussed in this section.

This section begins with the topics of shading, shadows, specular highlighting and reflection, all of which were used in the implementation described in Chapter 4. The section following that includes discussion of additional photorealistic effects they may be added to the implementation in the future.

2.4.1 Diffuse-Ambient Model

The diffuse-ambient model produces satisfactory shading for objects based on their relationship to the light sources in the scene. Assume that a primary ray has been found to intersect an object at the point **p**. The normalized normal to the object at this point is the vector **n**, and a single white point light is at the point **l** (see Figure 16). The task is to determine a shading percentage at which to illuminate the sphere at this point.

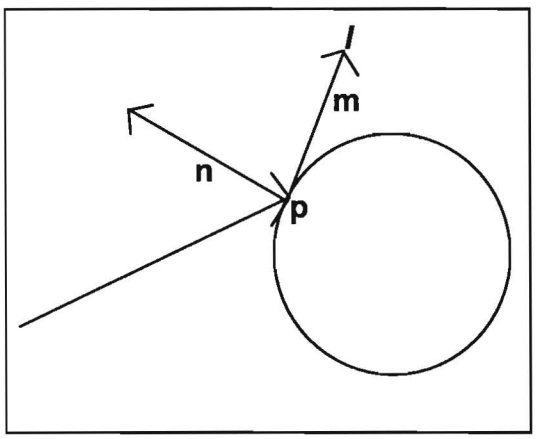


Figure 16: A primary intersects the sphere at point **p**. The normal to the sphere at this intersection point is **n**. The light source is at **l**, and the normalized vector that points from the intersection point to the light source is **m**.

The diffuse portion of the model is simply a restatement of Lambert's Law. If the vector **m** is the normalized direction from **p** to **l**, Lambert's Law states:

$$\mathbf{n} \cdot \mathbf{m} = o$$

where o is the percentage illumination desired [4]. In other words, the percentage illumination is directly related to the angle between **m** and **n**. With additional light sources, the percentage illumination is simply the sum of all the results of using Lambert's law with all of the light sources in the scene.

One problem with this model is that the percentage illumination for a single light source may be negative. In practice, either the absolute value of the result is used or all negative values are replaced with zero. The implementation described in Chapter 4 uses the latter option.

Another problem is the utter lack of attention given to background light. In the

real world, a light source gives off millions of light rays in every direction. These light rays bounce off all the objects in the room until they enter the observer's eye. A light ray that travels from the light source to the object and then directly to the eye is the sort of light ray that the diffuse model traces. However, a light ray may also bounce off many objects before entering the observer's eye. These rays constitute background light and are not traced from the diffuse model alone.

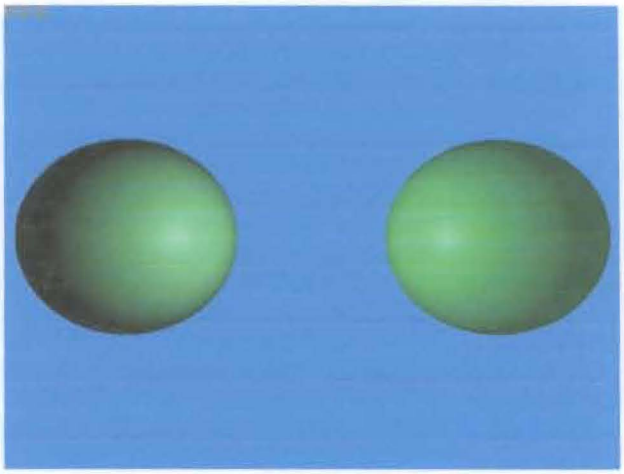


Figure 17: The sphere on the left has no ambience. The sphere of the right has a slight ambience. The light source was placed in between the two spheres. This image was generated using the implementation described in this thesis.

The ambient model allows a cheap (if oversimplified) way of dealing with this problem. Basically, it is assumed that there is a certain amount of background light (which is also known as *ambience*) in the scene. In other words, if no light source directly shines at point **p**, the illumination at that point will not be zero, but rather some baseline ambience value known as a [1]. Figure 17 demonstrates the difference.

Balancing ambience against diffusion can be tricky - many ray-tracers handle the problem differently. This thesis's implementation allows each object to have their own ambient coefficient since it is assumed that different objects reflect background light differently. A diffuse coefficient b is also created in contrast to the ambient coefficient - if the maximum value the ambient coefficient a can have is 1, then b is defined as $(1-a)$. Combining the ambient and diffuse models together yields the following illumination

function:

$f_7(\mathbf{n}, \mathbf{p}, a, \mathbf{l}_0, \mathbf{l}_1, \mathbf{l}_2 ... \mathbf{l}_{i-1}) = o$
$1: b = 1 - a$
$2: o = 0$
$3: \text{for}(j = 0 \text{ to } i - 1)$
$4: \quad \mathbf{m} = \dfrac{\mathbf{l}_j - \mathbf{p}}{|\mathbf{l}_j - \mathbf{p}|}$
$5: \quad k = \mathbf{n} \cdot \mathbf{m}$
$6: \quad \text{if}(k > 0) o = o + k$
$7:$ end for loop
$8: o = b * o + a$
$9: \text{if}(o \leq 1) \text{return } o$
else return 1

There are *i* lights in the scene. Line 1 sets up the diffuse coefficient. Line 3 loops through all the lights in the scene. Line 4 sets up the vector from the intersection point to the current light source and normalizes the result. Line 5 applies Lambert's Law. Line 6 throws out all negative values. Line 8 scales down the result by the diffuse portion and adds the ambient component. Line 9 ensures that the percentage is no larger than 100%.

2.4.2 Shadows

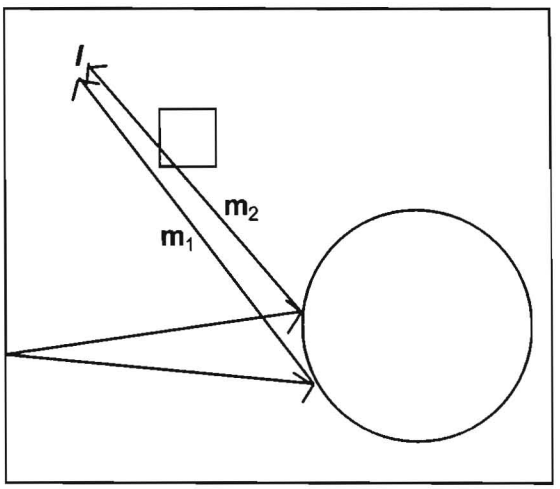


Figure 18: $\mathbf{m}_1$ does not have any obstacles before it reaches the light source and therefore there is no shadow cast. On the other hand, $\mathbf{m}_2$ does have an obstacle and therefore a shadow will be cast at its origin point.

Shadows are not difficult to implement using ray-tracing. They are merely time-consuming. Recall that in the diffuse-ambient model a vector **m** is created that starts at the intersection point **p** and ends at the appropriate light source location. The secondary ray created by using the origin at **p** and using the direction specified by **m** is called a *shadow ray*. If there is an object along this ray before it reaches the light source, there is a shadow. If not, there is no shadow (see Figure 18).

The concept is only slightly complicated through the use of multiple light sources. In this thesis's implementation, if an object casts a shadow over an object for a given light source, the diffuse calculation for that light source will be left out. If there is no shadow, the diffuse calculation will be incorporated into the illumination. The ambient coefficient is included regardless of the shadow's existence.

Determining if there is a shadow can be a costly computation. The brute force solution involves searching through all the objects in the scene in order to see if any of them intersect with the current shadow ray. This thesis's implementation uses a light buffer optimization which involves precalculating which objects can possibly cast shadows onto other objects. Only these objects and no others are considered for the shadow ray intersection tests.

2.4.3 Specular Highlighting

Not all surfaces should be dull. A small amount of shininess can be added to make an object more interesting (see Figure 19). One basic equation to accomplish this is the Phong lighting equation [3]:

$$s(\mathbf{r} \cdot \mathbf{v})^{s_{shi}}$$

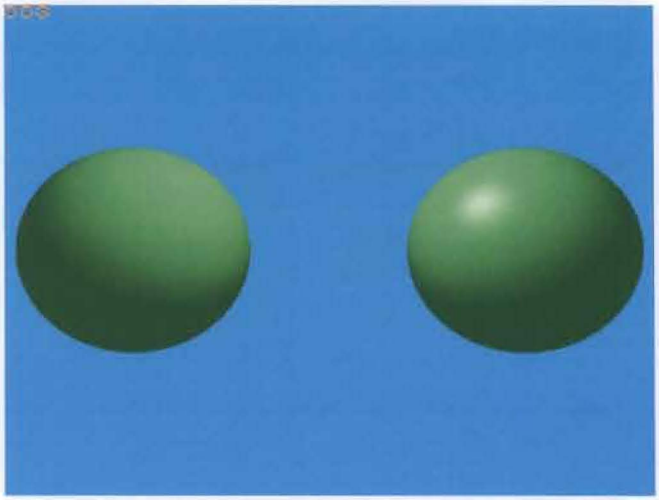


Figure 19: The sphere on the left has no specular component. The sphere on the right has a strong specular component. This image was generated using this thesis's implementation.

s is the specular coefficient. It can be from 0 to 1. 0 means a very dull surface while 1 makes a very shiny surface. **v** is the normalized vector from the intersection point **p** to the user. s_{shi} is the specularity constant. This controls how "tight" the specular component is. Values between 3 and 200 are acceptable. **r** is the reflection of the light vector **m** around the normal **n** and can be calculated in the following manner:

$$\mathbf{r} = 2(\mathbf{n} \cdot \mathbf{m})\mathbf{n} - \mathbf{m}$$

In the implementation described in Chapter 4, the specular component is calculated along with the diffuse component. If there is a shadow over the object, the specular component is omitted.

2.4.4 Reflective Surfaces

With this photorealistic effect, ray-tracing becomes recursive. The idea is simple - if the primary ray encounters a reflective object, trace *another* ray that originates from the intersection point in the direction of **r** as described in the previous section (see Figure 20). The resulting color will be a mix of the object's color and what the reflected secondary ray finds depending on that object's reflective coefficient *r*. Mathematically, the color of the pixel will be (1-*r*) times the color of the object itself added onto *r* times the color of the reflected secondary ray.

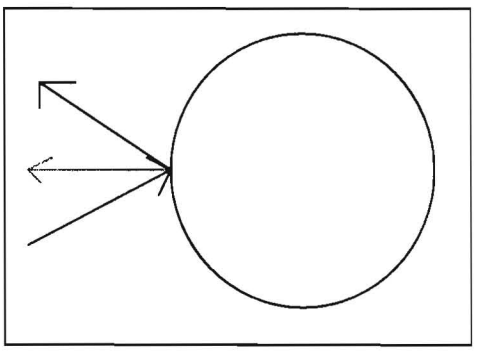


Figure 20: If a primary ray hits a reflective surface, a *new* ray is traced.

In practice, there must be a maximum recursive depth for reflective surfaces. Suppose the reflected ray hits another reflective object. Then it is necessary to cast a second reflected ray. If all the objects in a scene are reflective, this process can continue infinitely. A maximum depth of three or four loops is typically enough to satisfactorily display most scenes.

2.5 Topics for Further Research

Now it is time to look over more advanced concepts. None of the following concepts were successfully implemented though they may be in the future.

The idea of *transmittance* is as equally recursive as implementing reflective surfaces. Transparent objects follow the concepts of Snell's Law in order to describe how the primary ray is refracted through the surface (see Figure 21). Specifically, one refracted

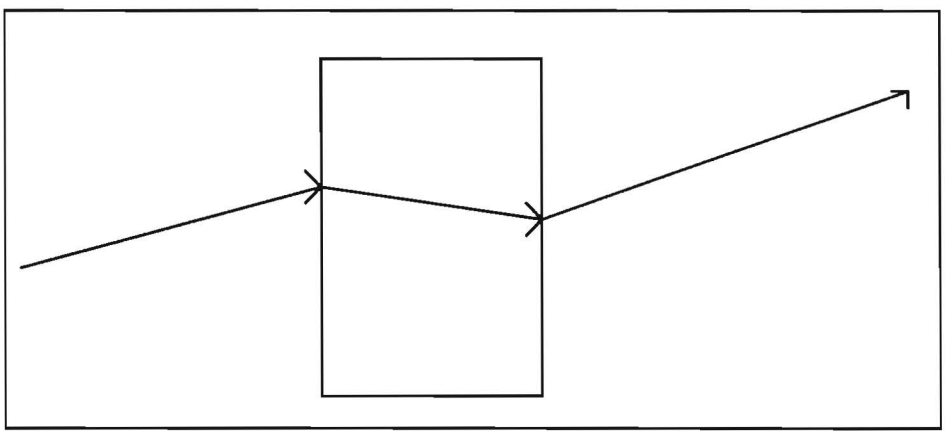


Figure 21: When a light ray hits a transparent object, the ray is refracted according to Snell's Law.

secondary ray would be generated to pass through the transparent object while a second refracted secondary ray would serve as the new recursive ray that is traced. Like the reflective surface described earlier, a transparent object is governed by its *transmittance coefficient*. In other words, an object may only be partially transparent. This effect would also be governed by a maximum recursive depth in order to prevent an overwhelming amount of computation in scenes with a lot of transparent or reflective objects.

Care must be taken in objects that are both transparent and reflective. While such objects exist in the real world (a window, for example), an object must not be allowed to be fully reflective *and* fully transparent. Such an object makes no sense - if 100% of the

Figure 22: A demonstration of planar, cylindrical, and spherical texture mapping. Image generated with POV-RAY.

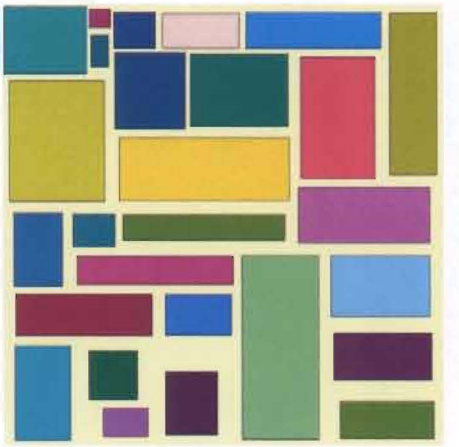


Figure 23: The image map used to generate Figure 22.

surface of that object is determined by what it reflects, how can 100% of the surface also be determined by what can be seen through it? One idea to combat this problem would be to set a maximum for the sum of the reflective and transparent coefficients. For example, if the maximum sum of the coefficients was set to 95%, an object could be 40% reflective and 50% transparent, though not 60% reflective and 50% transparent.

Another feature that could be implemented successfully in a ray-tracer is *texture mapping*. In this thesis's implementation, all objects have solid colors. However, other ray-tracers allow a texture (whether defined by an image map or by a color interpolation equation) to wrap around an object in several ways (see Figures 22 and 23). The process for all texture mapping is the same - generate a transform that converts points on an object to a set of barycentric points which take the form (u, v). These barycentric points correspond directly to the image map that defines the texture.

A similar concept is that of *bump mapping*. In ray-tracing, a bump map simply alters the normals at any given point on an object according to a preconceived pattern. The result is that the surface of the object no longer looks perfectly flat (see Figure 24).

Naturally, this is only a simulation of a bumpy surface. The best way to apply a bumpy surface to an object would be to model the bumps directly into the surface itself. However, a bump map provides an easier, alternative way to give the illusion of a bumpy surface.

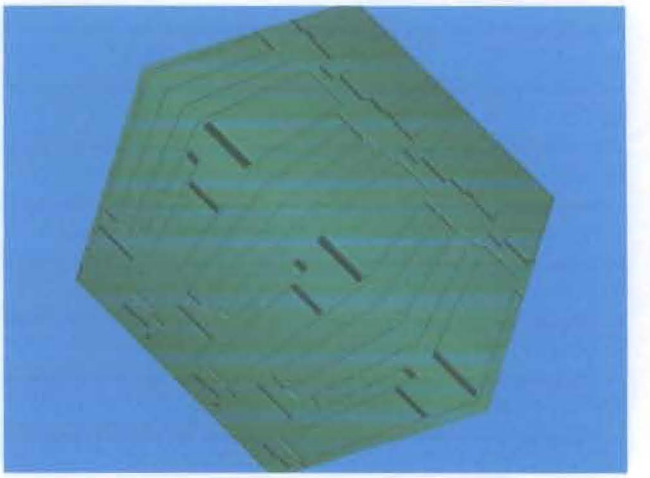


Figure 24: A box with the image in Figure 23 used as a bump map.

Another way in which this thesis's ray-tracer could be improved would be to add more object types - and with them, more ray-object intersection algorithms. An efficient general polygon to ray intersection algorithm known as the Crossings Test makes it feasible to use polygons with any number of vertices within a ray-tracer [3]. Using many of the principles used in the sphere-ray intersection algorithm as described in section 2.3.1, a general quadric to ray intersection algorithm can be derived [1].

Another interesting set of objects that could be added are known as *Constructive Solid Geometry* objects. These objects are formed by taking combinations of objects and applying boolean or arithmetic operations to them. Figure 25 shows two of the most popular applications of such operations.

There are of course many other types of objects that can be discussed here. Any object can be displayed as long as it is possible to define a stable ray-object intersection algorithm. However, it is also possible to incorporate advanced light source types. While the point light system works fairly well under most circumstances, it has its limitations. Ray-tracers have been known to use *area lights* - lights that illuminate equally in all

directions but die off with increased distance - with great success. *Spot lights* are a sort of directional area light in that they point in a certain direction with a given strength. The "spot" generated by this sort of light source is given its own strength, which determines its basic radius and softness around the edges. *Volume lights* remove the point attribute of the light source - they are given physical dimensions. Indeed, volume lights are objects in themselves even though many ray-tracers keep them invisible.

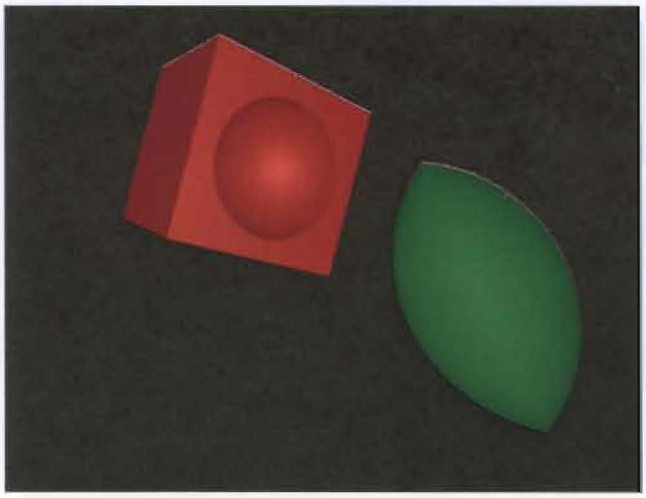


Figure 25: The red object was created by taking the *difference* between a box and the sphere. The green object shows the *intersection* between two spheres. This image was generated using POV-RAY.

A final topic for further research would be to remove the ambient component of the diffuse-ambient model and replace it with the concept of *radiosity*. Although ambience is easy to implement, the idea of assigning a general ambience to each object in a scene is not at all based on physical light theory. Radiosity calculates the amount of background light in a scene by letting light rays acting as photons bounce around the scene until an energy equilibrium is reached [3]. This is of course incredibly time-consuming, and for many scenes the ambient shortcut provides adequate results (see Figures 26 and 27).

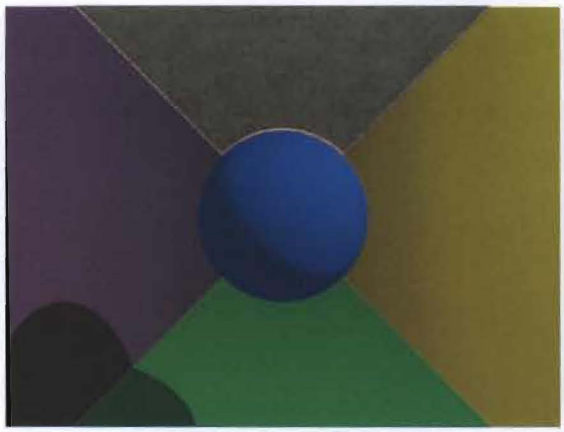


Figure 26: The above image uses the ambient shortcut. It was generated using POV-RAY in 3 seconds.

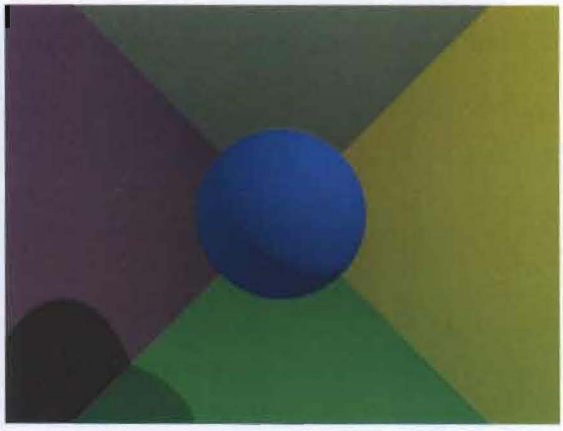


Figure 27: The above image uses radiosity. It was generated using POV-RAY in 13 minutes and 3 seconds.

2.6 Altered Main Ray-Tracing Loop

The following is the main ray-tracing loop in light of the algorithmic discussion that followed the presentation of the first version.

Do for every pixel on the screen
  Calculate the primary ray for the current pixel (f_2) while remapping correctly
  Do for every object in the scene
    Check to see if the ray intersects the current object (f_3, f_4, f_5, f_6)
  If no object intersects the ray, use the background color and proceed to next pixel
  If multiple objects intersect the ray, use the closest object.
  Do for every light in the scene
    Do for every object in the scene
      Determine if a shadow is cast on the object
      If there is no shadow, determine diffuse-ambient (f_7) and specular component
  Scale total diffuse/specular component and add in ambient component
  If object is reflective, trace another primary ray from the normal of the object
  Determine color of pixel based on shading and (if needed) reflective component

The ray-tracing algorithms are elegant and simple. However, as can be seen, there are many time-consuming loops. The efficiency of these algorithms as a set will be discussed in the next section.

2.7 Efficiency

The efficiency of the set of ray-tracing algorithms is a major concern. It depends on the number of pixels (P) on the screen, the number of objects (S) in the scene, and the number of lights (L) in the scene. Without reflection, the efficiency of the algorithm is:

$$O(P,S,L) = PS^2L$$

Obviously it is directly proportional to P since it traces P amount of primary rays. In testing these primary rays against intersection, it loops through all the objects in the scene, hence the multiplication by S. The ray-tracer also traces a maximum of LS shadow rays per pixel, hence the additional multiplication by L and S. With a maximum recursive depth of R added in for reflective objects, the efficiency becomes this:

$$O(P,S,L,R) = P\left(S^2L\right)^R$$

A reflected ray is a recursive operation which involves tracing an additional primary ray. Since tracing this primary ray may in fact lead to another reflective object, and then to another, and another, and so on, the worst-case efficiency of this algorithm becomes exponential. The algorithm in its pure form is in dire need of optimization. Fortunately, the next chapter is dedicated to that cause.

3 Binary Space Partition

In order to generate Figure 28, a total of 66,024,774 primary ray/object intersection tests were performed. Only 653,864 (0.99061%) of these intersection tests returned success - meaning that they actually hit an object. On the other hand, 66,042,774 (99.0099%) returned failure - the ray does not intersect the current object at any point. This chapter is dedicated to using the binary space partition algorithm to improve the efficiency of a ray-tracer by correcting this horrible imbalance.

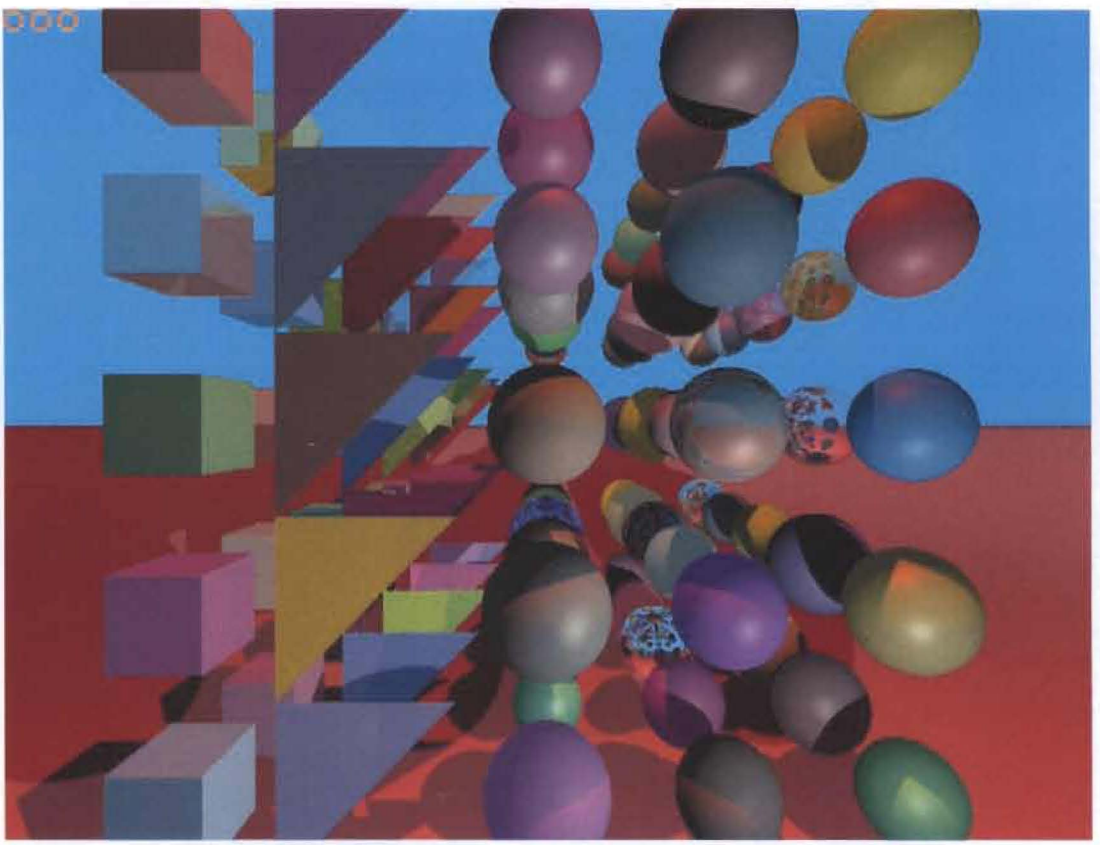


Figure 28: An image generated by this thesis's implementation without any optimizations.

3.1 General Concept

The concept of binary space partitioning is not new. It has been applied successfully to many different rendering techniques. The idea is simple. First, start with the whole screen. Determine which objects are visible on the screen. Then partition the screen into two halves, determining which objects are visible on which half. Then partition each half into smaller halves while determining which objects are visible in these smaller halves. The partitioning should recursively continue until some sort of termination criterion is met. Once this has occurred, stop partitioning. Then, when testing primary rays against objects for intersection, only consider those objects that are within that ray's most specific partition [3].

For example, examine Figure 29. The first step to processing this image is to partition it into two halves, as in Figure 30. Keep in mind that the top image has the sphere while the bottom image has the box. Figure 31 shows the two partitions divided in half again. Note that there is no need to partition the upper-left and lower-right partitions any further since they have no objects. Figures 32 through 34 show the future partitions. Figure 34 actually shows the result of partitioning two more times from the preceding figure. At this point, there is no need to partition further since doing so would not really help to reduce the quantity of

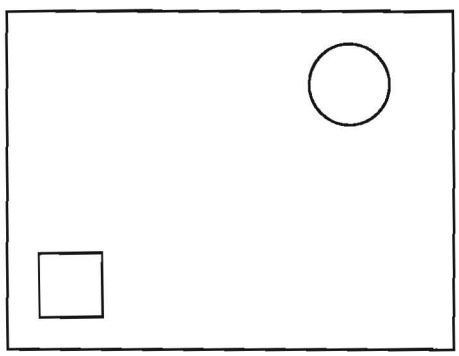


Figure 29: A sample image.

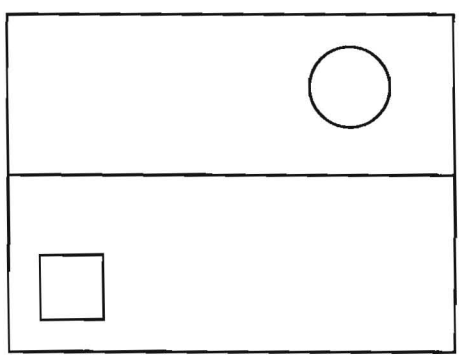


Figure 30: The image divided in two.

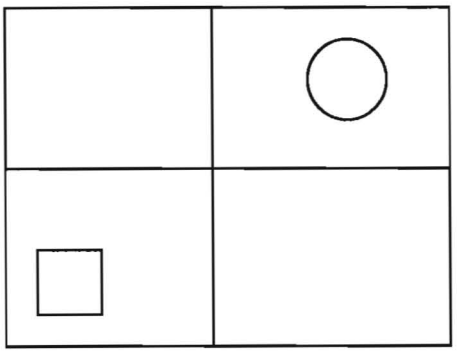


Figure 31: The halves divided again.

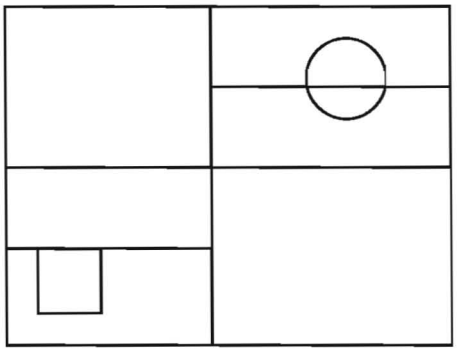


Figure 32: Only the partitions of interest are divided further.

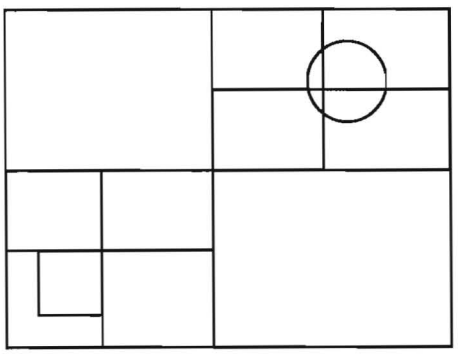


Figure 33: At this point, it is no longer necessary to partition the lower-left quarter of the screen.

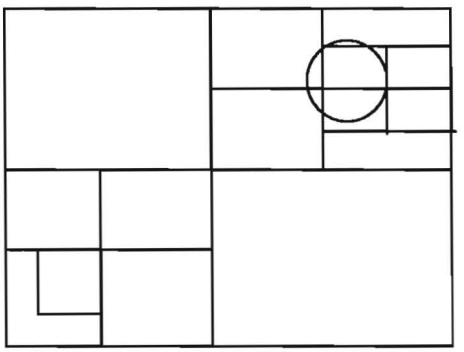


Figure 34: Two more partitions later, there is no need to continue any further.

primary ray calculations by any significant amount.

There are several observations to note about Figure 34. First of all, any pixel in any of the empty partitions of the image will automatically be the background color since there is no object in that portion of the image. Second, a primary ray needs to be tested against the box only if the partition it belongs to contains the box. Likewise, only primary rays that pass through a partition containing the sphere need to be tested against the sphere. This is the power of binary space partitioning. When a primary ray is traced, it is only tested against the objects that it is most likely to hit.

This algorithm has the potential to cut down on the number of failed primary ray/object intersection tests for any scene. However, the termination criterion must be

defined very carefully. There are three very obvious conditions in which a partition no longer must be recursively split any further. The first is when there are no objects in the current partition, as in the upper-left quarter of the image in Figure 31. The second is when there is only a single object in the current partition that completely fills the partition. The third is when a maximum recursive depth is reached.

There may be a fourth acceptable condition depending on the circumstances. If an object *mostly* fills a partition, as in the sphere in Figure 34, there should no longer be any need for any further partitioning. However, such a thing may be difficult and time-consuming to test depending on the object tested. If the overhead of this optimization takes more time than what time the optimization actually saves, the optimization is worthless. Therefore, this thesis's implementation does not include this fourth rule.

There is also some debate over the partitioning itself. Should a partition always be divided in half? Would it make sense for the partitioning to occur at some other point given a specific situation? Such research exists, but is outside the scope of this thesis. In this thesis's implementation, a partition is always split directly in half along the axis with more pixels. For instance, if a current partition is 30 pixels wide and 15 pixels high, it will be split at the middle of its width.

3.2 Binary Trees

This thesis's implementation uses binary trees in order to store the binary space partition information. This is logical since a binary tree is composed of a root and two children that can recursively be defined as additional binary trees. The relationship between a binary space partition and its binary tree is demonstrated through Figures 35 and 36.

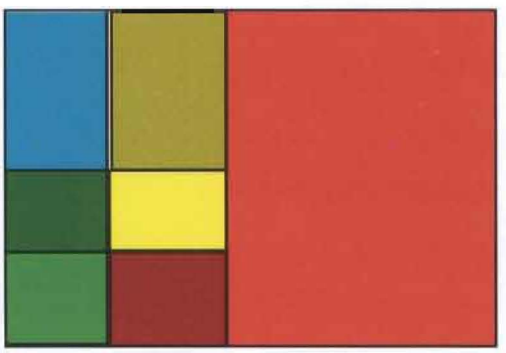


Figure 35: An example binary space partition.

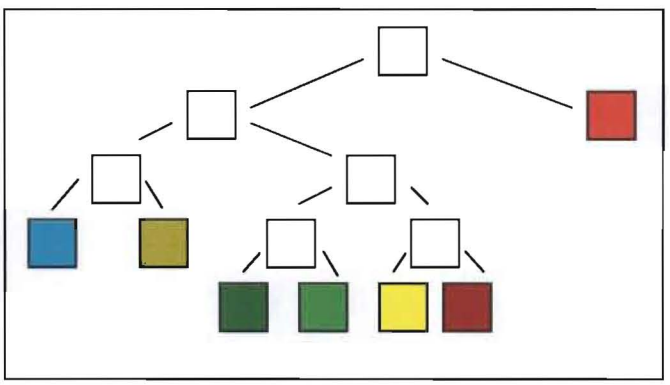


Figure 36: The binary space partition in Figure 35 converted to a binary tree format. The white squares represent larger partitions that are no longer visible.

The process of creating a binary tree from its partition is straightforward. The top node in Figure 36 represents the entire screen. All the objects visible on the screen are stored in this top node. The second level of nodes represent the halves of the screen. The left node represents the left half of the screen while the right node represents the right half. Each node contains the objects that are visible within its corresponding half of the screen. Since the right half is not partitioned any further, this right node of the second level is colored red.

The third level of nodes represents the upper-left quarter and the lower-left quarter of the screen. The upper-left quarter is partitioned only once more to yield the final colored boxes. Since the lower-left quarter is partitioned twice more, there are two more

levels to the tree representing the last partitions.

Each node of the tree has stored in it the objects visible within that partition. When a partition is split into two children, the objects stored in the children are chosen from the parent's list of objects. Note that it is possible for an object to be stored in *both* children of a parent node, as in the sphere in Figure 37.

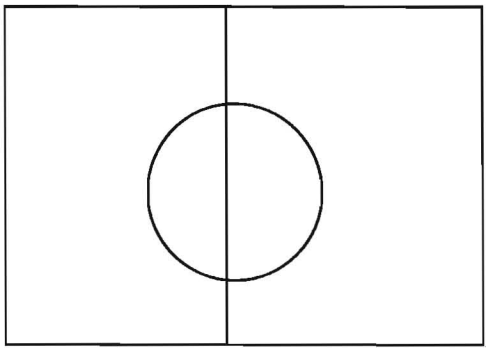


Figure 37: This sphere belongs to both the left partition and the right partition. Therefore, the sphere will be stored in both the left child and the right child in the corresponding binary tree.

When a primary ray is traced, it is only tested for intersection with the objects within its most specific partition. Since this information is stored within the corresponding node of the binary tree, the primary ray is tested for intersection with the objects that are stored within this node within the binary tree.

A binary tree is an ideal way to represent a binary partitioning of the screen since its structure exactly matches that of the partition. A given partition may be represented by a parent node of the tree while the two halves created when the partition is split may also be represented as the two children of the parent node. It is an efficient method to convert the algorithm into an implementation. The specifics of the binary tree implementation is discussed in Section 4.7.

3.3 Efficiency

Shockingly, the binary space partition algorithm does nothing to improve the worst-case efficiency of the ray-tracing algorithms. A worst-case scenario would be where all of the objects are stored in *every* tree location, all the way down to the maximum recursive depth. Visually, this corresponds to every object on the screen intersecting every primary ray generated. Fortunately, this is a rare occurrence. Most scenes involve many objects, most of which take up only a portion of the screen. The binary space partition algorithm works well with such scenes, as will be discussed in Chapter 5.

3.4 Alternative Approaches

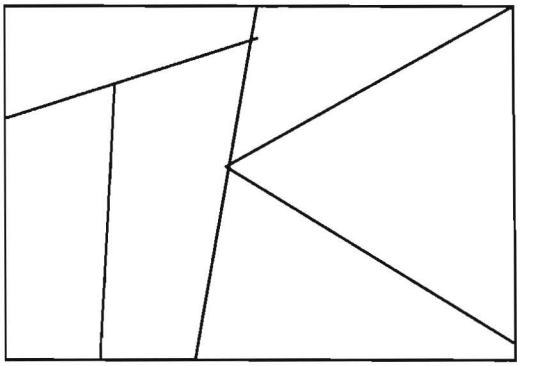


Figure 38: Clipping planes allow the screen to be asymmetrically partitioned along diagonal lines.

The binary space partition algorithm is not the only approach in existence to remove unnecessary primary ray calculations. A specific ray-tracer may instead use the idea of *clipping planes* for optimization [3]. This concept is essentially an extension of the binary space partition algorithm which allows the screen to be partitioned up unevenly along planes that lie along the z axis (see Figure 38). The advantage of such a system is that it may be possible to generate binary trees which are more balanced, though the tree generation is admittedly more difficult.

An additional method may be to quarter the screen into four partitions instead of two. This method is often referred to as the *quadtree method* since quadtrees are required to store the partition information [3]. Note that dividing a screen into four partitions is the same as dividing it in half two times. Therefore, the quadtree method can be thought of as an alternative way as describing the binary space partition algorithm. While it has more overhead processing than the binary space partition algorithm (since there are four partitions to consider at each level), it requires less recursive levels. Essentially, the computational time between the two algorithms is theoretically the same.

The *octree method* [3] contrasts directly with the binary space partition algorithm. The basic idea is to recursively partition the scene into eight octants while keeping track of which objects are in which octant. Primary rays are only tested against objects that are within the most specific octants that the primary ray passes through. The binary space

partition algorithm is different since it partitions the output image rather than the scene itself. As the name suggests, octrees are used to store the partition information.

The binary space partition algorithm was chosen for its simplicity and elegance in both concept and implementation. The following chapter focuses on the implementation that was used to create many of the ray-traced images in this thesis. The alternative methods discussed in this section will be briefly compared to the binary space partition algorithm in Section 5.5.

4 Implementation Overview

The point of this chapter is to give an overview of this thesis's implementation. It is not a line-by-line detailed explanation of the code, since such a discussion would triple or quadruple the length of this thesis. The implementation was programmed in Microsoft Visual C++ with DirectX used to draw the images. Compiled executables can be found at http://www.iwu.edu/~mportole/ray.html and can be run on any modern Windows machine.

There are essentially seven main classes to the implementation, while will be discussed in turn:

1. Image - The Windows and DirectX handler. It will not be discussed further.

2. Vector3D - This class handles all vector operations.

3. Surface - This class stores surface (color, ambient, specular, reflective) information about an object.

4. Color - In addition to defining the class Color, this class defines many predefined colors that can be used in the implementation.

5. Object - This class defines an object. All the specific object classes are derived from this class.

6. Light - This class defines a light. The PointLight class is derived from this class.

7. World - This class encapsulates both the main loop of ray-tracing and the binary space partitioning algorithm. Discussion of this class will be divided into those two sections.

4.1 The Class Vector3D

This is the most interesting part of the Vector3D declaration:

```
class Vector3D {
public:
        float x, y, z;
        void Normalize();
        Vector3D(float x1, float y1, float z1);
};
```

Essentially, a vector is represented through three public floating-point variables *x*, *y*, and *z*. The Normalize() function normalizes these variables. The constructor allows quick initialization of an instance of this class. The declaration continues to make the following all valid:

```
Vector3D a(5.0, 3.0, -2.0), b, c;
float d = 7.5;

b.x = 10.0; b.y = 0.0; b.z = 5.0;
c = Vector3D(1.0, 2.0, 3.0);
c = a + b;
a = b * a;
c += (a + d);
// etc.
```

4.2 The Class Surface

The Class Surface includes information about an object:

```
class Surface {
public:
        Color Color;
        float Diffuse, Specular, Reflect;

        Surface(Color C = White, float D = 0.2, float Sp = 0.2, float R = 0.0) {
                Color = C;
                Diffuse = D;
                Specular = Sp;
                Reflect = R;
        }
};
```

Its main advantage is the fact that it uses default values if the user omits some surface attributes. Every Object has an instance of Surface. The *Diffuse*, *Specular*, and *Reflect* components can have any value between 0.0 and 1.0.

4.3 The Class Color

The Class Color stores information about colors much like the Vector3D class stores information about vectors:

```
class Color {
public:
        float r, g, b;
        Color(float r1, float g1, float b1);
};
```

The floats *r, g,* and *b* are assumed to be within the range of 0.0 to 1.0. The file Color.h has many predefined color macros that are available for use. The following is valid:

```
Color a(Red), b(0.5, 0.5, 0.5), c;
c = b;
// etc.
```

4.4 The Class Object

The Sphere, Box, Plane, and Triangle classes are all derived from this class:

```
class Object {
public:
        Surface Surface;

        virtual float WhereIntersect(const Ray &R);
};
```

The actual class declaration is slightly more complex, but this is the most important part of it. The function WhereIntersect() determines where the intersection between the object and the Ray R occurs. It returns the *k*-value if there is an intersection, the constant INFDIST otherwise. This is of course a completely virtual class. The Sphere, Box, Plane, and Triangle classes all inherit from this class and provide definitions for the WhereIntersect() function according to functions f_3, f_4, f_5, and f_6 as discussed earlier.

The inherited classes also contain additional variables as needed. The Sphere class has a *Center* vector along with a *Radius* float. The Box class has three *Direction* vectors, three *HalfLength* scalars, and a *Center* vector. The Plane class has a *Direction* vector and an *Offset* float. Last of all, the Triangle class has three *Vertex* vectors which represent the three vertices. These additional variables are necessary to completely define the object in question.

4.5 The Class Light

The PointLight class is the only class that currently derives from this class:

```
class Light {
public:
        Color C;
        Vector3D Origin;

        virtual bool Shadow(...);
};
```

Every Light is assigned a color and an origin point. The shadow check is completely encapsulated within the member function Shadow of the Light class using the light buffer technique as described earlier. It returns true if there is a shadow cast over the object at the specified point. It returns false otherwise.

4.6 The Class World: Ray-Tracing

This class is the heart of the ray-tracer. It encapsulates within it the entire main loop of ray-tracing:

```
class World {
public:
        Object **Objects;
        Light **Lights;

        Init(...);

        Sphere *NewSphere(Vector3D Center, float Radius, Surface S);
        Box *NewBox(Vector3D Center, Vector3D *Orientations, float
*HalfLengths,
                Surface S);
        Plane *NewPlane(Vector3D Orientation, float Offset, Surface S);
        Triangle *NewTriangle(Vector3D *Vertices, Surface S);
        PointLight *NewPointLight(Vector3D Origin, Color C);

        void SetBackgroundColor(Color C);

        Color TraceRay(Ray R, int depth);
        void RenderWorld(Image &I);
};
```

This class contains an array of Object pointers and an array of Light pointers. These are all the objects and lights in the scene. The Init() function sets up the ray-tracer and must be called before RenderWorld(). There are a variety of different New...() functions, one for each type of object and light source. These allocate a specific type of Object or Light and sets up its location in the array Objects or the array Lights. A pointer to the object or light is also returned so that the caller has direct control over the attributes of the object or light. The background color can be set by using SetBackgroundColor().

The TraceRay() function traces a ray into the scene, looking for intersections with objects. It does all the necessary ambient, diffuse, and specular calculations. If a reflective object is encountered, TraceRay() calls itself to continue the process. The integer depth is the current recursive depth. This value will be larger than 0 only if the ray processed is a reflected ray.

The function RenderWorld() generates a ray-traced image and stores it in the Image I. It generates a primary ray and calls TraceRay() once for each pixel in the image. The primary ray information is actually stored in a look-up table in order to ease computational demands.

4.7 The Class World: Binary Space Partition

The addition of the binary space partition algorithm changes the class declaration slightly. There are now two functions for tracing rays - one for primary rays and one for all other rays:

```
Color TracePrimaryRay(Ray R, Node *Leaf);
Color TraceRay(Ray R, int depth);
```

The idea is that when a primary ray is being traced, the most specific node corresponding to that primary ray's position is passed along with the ray information. Since the node contains the most likely objects that the ray will hit, TracePrimaryRay() must concern itself only with those objects instead of every object in the scene. TraceRay() is now used only for reflected rays.

The binary tree is stored in an array where the root of the tree is at index 0, its two children are stored at indices 1 and 2, their children are stored at indices 3 through 6, etc. Each index contains a boolean that tells whether the current node is "active." If the node is active, it is *not* partitioned further and therefore should be passed to TracePrimaryRay(). However, if the node is inactive, it is partitioned further. If this inactive node's array index is x, then its (potentially active) children are stored at array indices $(x*2+1)$ and $(x*2+2)$. This can be observed in Figures 39 and 40. The tree is generated by the private member function MakeTree() which is called within RenderWorld(). The pseudocode for MakeTree() is function f_8.

f_8 uses an iterative method to generate the tree. It defines an additional boolean called "process" for each node in the array n. The boolean is true for array index x if the partition at n[x] must be processed by MakeTree(). It has nothing to do with the "active"

0	1	2	3	4	5	6	7	8	9	10

Figure 39: An example array.

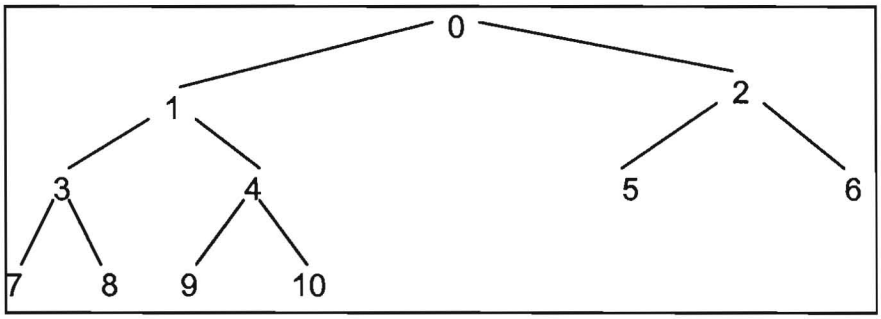


Figure 40: This is the array in Figure 38 mapped to a binary tree. From a given index x, the parent of node x is at array index $\frac{x-1}{2}$ while its children are at array indices ($x*2+1$) and ($x*2+2$).

boolean described earlier. Lines 1-4 clear both of these booleans to false initially. Lines 5-8 adds all objects visible in the image to the node n[0], which represents the entire screen. Lines 9 and 10 set the process booleans for the nodes representing the halves of the screen (nodes 1 and 2) to true so that the screen will be partitioned at least once. Lines 11-22 contains the iterative loop that will generate the rest of the tree. Note that line 12 ensures that the current partition will only be processed if the corresponding process boolean is set to true. Lines 13-16 loop through all the objects stored in the parent node, storing them in the current node if they are visible in the current partition. Lines 17-21 calculate the termination criteria. If the current node does not need to be partitioned further, that node's active boolean is set to true. Otherwise, the process boolean of that node's children is set to true. They will be processed during their corresponding iterations, extending the tree to the next level.

f_s(Array of nodes n[0...$i-1$])
1 : for($j = 0$ to $i-1$)
2 : n[j].active = false
3 : n[j].process = false
4 : end for loop
5 : for($j = 0$ to total number of objects)
6 : if(Object [j]is visible on screen)
7 : Add Object [j]to node n[0]
8 : end for loop
9 : n[1].proce ss = true
10 : n[2].proce ss = true
11 : for($j = 1$ to $i-1$)
12 : if(n[j]process = true)
13 : for($k = 0$ to total number of objects in n$\left[\frac{j\text{-}1}{2}\right]-1$)
14 : if(n$\left[\frac{j\text{-}1}{2}\right]$.Object [$k$]is visible in partition j)
15 : Add n$\left[\frac{j\text{-}1}{2}\right]$.Object [$k$]to node n[$j$]
16 : end k for loop
17 : if(total number of objects in n[j] = 0 or total number of objects in n[j] = 1 and it fills partition j or at maximum recursive depth)
18 : n[j]active = true
19 : else
20 : n[$j*2+1$]process = true
21 : n[$j*2+2$]process = true
22 : end j for loop

RenderWorld() works by cycling through the array that stores the binary tree. If the current node is active, all the pixels in that node are converted to primary rays and traced at once. Otherwise, it continues to the next array element. Since MakeTree() assigns only the most specific partition active status, each primary ray in the image is passed to TracePrimaryRay() only once during the call to RenderWorld().

5 Results

Finally, it is time to look at the results. There are four tests in this chapter. The first is a simple image test to see if the binary space partition algorithm actually does reduce the number of ray-object intersections. All the other tests are concerned with speed efficiency, though the number of ray-object intersections are also recorded. These tests were conducted on a 1.3GHz AMD Athlon processor with 256 MB memory along with a 64 MB GeForce2 MX graphics accelerator. Once again, compiled executables can be found at http://www.iwu.edu/~mportole/ray.html.

From this point on, the binary space partition algorithm is abbreviated as BSP. The algorithm is tested at multiple recursive depths, and the title of the test reflects this maximum depth. For example, at BSP 1, the algorithm is only allowed to divide up the screen into two halves. At BSP 2, the algorithm is allowed to quarter up the screen. This continues all the way up to BSP 10.

5.1 Image Test

Figure 41: The image used for this test.

The point of this test is to see if the BSP algorithm actually does reduce the number of ray-object intersection tests required to generate Figure 28. It is reprinted in Figure 41 for convenience. It is a 800x600 image with 75 spheres, 25 triangles, 25 boxes, 1 plane, and 2 point light sources. Here are the results with this image without the binary space partition algorithm, and with the algorithm with different maximum recursive depths:

Implementation	Total Number Ray-Object Intersection Tests	Succeeded	Percentage Succeeded	Total Failed	Percentage Failed
No BSP	66,042,774	653,864	0.990061%	65,388,910	99.0099%
BSP Depth 1	38,640,800	654,830	1.69466%	37,985,970	98.3053%
BSP Depth 2	23,040,800	645,171	2.80012%	22,395,629	97.1999%
BSP Depth 3	13,800,800	651,330	4.71951%	13,149,470	95.2805%
BSP Depth 4	8,940,800	642,975	7.19147%	8,297,825	92.8085%
BSP Depth 5	6,406,000	646,852	10.0976%	5,759,148	89.9024%
BSP Depth 6	4,673,700	646,913	13.8416%	4,026,787	86.1584%
BSP Depth 7	3,762,050	640,294	17.0198%	3,121,756	82.9802%
BSP Depth 8	3,059,600	642,306	20.9931%	2,417,294	79.0069%
BSP Depth 9	2,839,975	641,913	22.6028%	2,198,062	77.3972%
BSP Depth 10	2,613,175	653,768	25.0182%	1,959,407	74.9818%

As can be observed by the above table, the total number of ray-object intersection tests dropped dramatically from 65,388,910 to 2,613,175 as the BSP maximum depth increased. This result is 25 times smaller than the unoptimized version. Since the number of successful intersection tests did not dramatically change, all of this improvement was due to omitting most of the intersection tests that would have failed.

It is curious that the number of successful intersection tests was not constant. While it did stay within the same general area, all the trials had a slightly different amount of successful intersections. This may be because of the limits of floating-point precision or perhaps there is some undiscovered bug in the ray-intersection algorithms themselves. Regardless, the image output did not look noticeably different.

While increasing the maximum BSP recursive depth did lower the number of intersection tests, it did so with an increased overhead cost of generating and processing the binary tree. The following three tests are designed to measure exactly how much faster the BSP implementation is than the unoptimized version depending on the maximum recursive depth.

5.2 Sphere Test

This test is the first animation test. The scene involves 8 spheres, 1 of which is reflective. There are 2 point light sources. The images are rendered at a resolution of 320x240. See Figure 42 for a sample frame. Here are the test results:

Figure 42: A sample frame from this test.

Version	fps	% faster	Intersection Tests Total	Success Total	% Success	Fail Total	% Fail
No BSP	8	0%	616,960	17,710	2.87053%	599,250	97.1295%
BSP 1	15	87.5%	307,200	17,710	5.76497%	289,490	94.235%
BSP 2	17	112.5%	153,600	17,710	11.5299%	135,890	88.4701%
BSP 3	20	150.0%	76,800	17,710	23.0599%	59,090	76.9401%
BSP 4	22	175.0%	57,600	17,710	30.7465%	39,890	69.2535%
BSP 5	22	175.0%	57,600	17,710	30.7465%	39,890	69.2535%
BSP 6	23	187.5%	48,000	17,710	36.8958%	30,290	63.1042%
BSP 7	24	200.0%	43,200	17,710	40.9954%	25,490	59.0046%
BSP 8	24	200.0%	35,400	17,655	49.8729%	17,745	50.1271%
BSP 9	23	187.5%	30,000	17,648	58.8267%	12,352	41.1733%
BSP 10	23	187.5%	26,960	17,648	65.4599%	9,312	34.5401%

All intersection test data is taken from the first frame. The "% faster" row is in comparison to the "No BSP" frames per second. The BSP algorithm certainly does increase speed efficiency up until a point. With each additional layer of partitioning, there is an additional amount of overhead in processing the binary tree. There were no significant increases in speed after the BSP model with a maximum recursive depth of 4. The data for BSP 4 and BSP 5 is not a misprint - they completed the exact same amount of intersection tests. Furthermore, the same amount of intersection tests succeeded and failed! Apparently the fifth layer of partitioning does not ease the intersection test burden at all in this specific example.

The BSP algorithm dramatically improves performance of the ray-tracer with this specific scene. BSP 7 showed a tripled frame rate when compared to that of the unoptimized version. Of course, there are a small number of small objects which only take up a portion of the screen. The next test tests a large number of small objects which span the entire screen.

5.3 Full Test

This test is simply the test in 5.1 with full range of motion (see Figure 43). Every object in the example except the plane rotate around the same center point. This animation was also rendered at 320x240 resolution. Here is the data:

Figure 43: A sample image from the animation.

Version	hpf	% faster	Total Tests	Total Success	% Success	Total Fail	% Fail
No BSP	171	0.0000%	10,552,626	103,986	0.9854%	10,448,640	99.014%
BSP 1	147	16.3265%	6,144,000	103,987	1.6925%	6,040,013	98.308%
BSP 2	134	27.6119%	3,763,200	102,428	2.7218%	3,660,772	97.278%
BSP 3	119	43.6975%	2,275,440	103,158	4.5335%	2,172,282	95.467%
BSP 4	106	61.3208%	1,507,600	102,188	6.7781%	1,405,412	93.222%
BSP 5	94	81.9149%	1,037,480	102,403	9.8704%	935,077	90.130%
BSP 6	102	67.6471%	768,200	102,591	13.355%	665,609	86.645%
BSP 7	87	96.5517%	593,620	100,854	16.990%	492,766	83.010%
BSP 8	90	90.0000%	506,680	101,520	20.036%	405,160	79.964%
BSP 9	91	87.9121%	439,430	100,709	22.918%	338,721	77.082%
BSP 10	96	78.1250%	373,990	101,051	27.020%	272,939	72.980%

"hpf" stands for hundredths of a second per frame. This test shows similar results to the previous test. The binary space partition algorithm did lower the number of ray-intersection calculations, though after around BSP 5, the overhead became too large in order for the improvement to be observed. The improvement itself was also less pronounced than the sphere test - this time the frame rate was improved to only double that of the unoptimized version.

While these results are still encouraging, perhaps an additional termination criterion would help improve performance even further. As can be seen by looking at Figure 43, the objects are distributed fairly densely across the screen. One of the termination criterion used was if there was only one object in the partition and that object completely filled the partition, do not recurse further. However, perhaps this can be altered to this: if there is one or more objects in the partition and at least one of those objects completely fills the partition, do not recurse further. At first glance, it appears that at a high enough maximum recursive depth and with this specific scene this termination criterion might actually help. However, it is also true that with many scenes this may not help at all (see Figure 44 for an example). It also would be a fairly expensive test computationally. In any case, it is true that the termination criteria may need to be altered given the type of scenes that are most likely to be rendered.

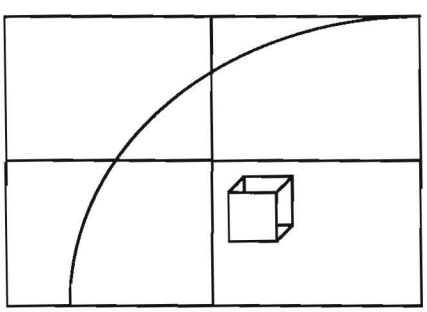


Figure 44: An example where the proposed new termination criterion would probably not be the best idea. The entire lower-right quarter should not need to consider the cube, yet since the sphere completely fills the quarter, it would not be partitioned further.

5.4 Plane Test

Figure 45: The plane test. Every object intersects every primary ray.

The binary space partition algorithm does not improve performance when all the objects in a scene take up the entire image. Therefore, this scene involves 5 vertical planes and 2 light sources. It was rendered at a resolution of 320x240. Figure 45 is a sample frame. Here is the test data:

Version	fps	% faster	Total tests	Total success	% success	Total fail	% fail
No BSP	5	0%	384,000	384,000	100%	0	0%
BSP 1	5	0%	384,000	384,000	100%	0	0%
BSP 2	5	0%	384,000	384,000	100%	0	0%
BSP 3	5	0%	384,000	384,000	100%	0	0%
BSP 4	5	0%	384,000	384,000	100%	0	0%
BSP 5	5	0%	384,000	384,000	100%	0	0%
BSP 6	5	0%	384,000	384,000	100%	0	0%
BSP 7	5	0%	384,000	384,000	100%	0	0%
BSP 8	5	0%	384,000	384,000	100%	0	0%
BSP 9	5	0%	384,000	384,000	100%	0	0%
BSP 10	5	0%	384,000	384,000	100%	0	0%

The binary space partition algorithm does not help this scene since every object intersects every primary ray. However, the overhead from maintaining BSP 10 did not decrease the frames per second in any noticeable fashion. While this is a disadvantage of using the BSP algorithm, this is a mercifully artificial example. Most scenes will not involve situations like this where every object intersects every primary ray.

5.5 Comparison with Alternative Approaches

This section compares the performance binary space partition algorithm with the alternative approaches described in section 3.4. All of the alternative approaches with the exception of the clipping planes idea were tried in the course of researching this thesis. They will be discussed in turn.

The clipping planes idea was never tried with this specific implementation. However, it is assumed that the performance of the algorithm would depend on the specific scene rendered. There is no doubt that there would be some imaginable scene in which clipping planes would be beneficial. At the same time, the algorithm would probably perform just as poorly with the plane test as the binary space partition algorithm. In practice, it would be more difficult to decide how the partitions should be divided since by its very nature it is more complex than the simple square partitions used with the binary space partition algorithm. In addition, it would be more difficult to determine which primary ray is in which partition for the same reason. Since the binary space partition algorithm is fairly good at reducing the amount of primary ray/object calculations with less overhead, it appears unlikely that a ray-tracer that uses clipping planes would perform noticeably better than a ray-tracer that uses the binary space partition algorithm.

The only difference between the binary space partition algorithm and the quadtree method is how each method describes their own partitions. Specifically, the maximum number of partitions created with n number of recursive levels using the binary space partition algorithm produces 2^n partitions while the quadtree method produces 2^{2n} partitions. Therefore, a quadtree implementation requires exactly half as many levels as a

binary space partition algorithm in order to produce the same number of partitions. This may be considered better since the quadtree method requires less levels or worse since there is additional overhead and perhaps more partitions than required. In the implementation created through the research involved in this thesis, there was no noticeable difference between the quadtree method and the binary space partition method when the total number of partitions were equal. Both algorithms performed equally poorly with the plane test.

The octree method is very popular in practice. Nevertheless, the implementation created through this research thesis always performed *slower* than the equivalent test using an unoptimized ray-tracer. Even the plane test performed very slowly. Theoretically, the octree method would very easily be able to cope with the plane test since it takes into account the three-dimensional differences between objects rather than the two-dimensional differences between where objects are rendered in the output image. There must be some way of using the octree method to successfully optimize primary ray processing. However, such a method was not found during the research phase of this thesis.

6 Conclusion

Ray-tracing is a powerful method for rendering three-dimensional graphics. It stands apart from polygonal based rendering systems with the abilities to display perfectly curved objects, movable shadows, shiny and reflective surfaces, as well as many other photorealistic effects. However, the set of algorithms that make up a ray-tracer are very time-consuming. While the calculations themselves are not extremely difficult, they are repeated so many times that even modern processors struggle under the load.

The Binary Space Partition algorithm is a first step to optimizing the performance of the set of ray-tracing algorithms described in this thesis. While the ray-tracing algorithm can be very demanding computationally, it is possible to improve performance by omitting unnecessary calculations.

The Binary Space Partition algorithm as applied here focuses on improving the performance of primary ray-object intersection tests. However, it does not do so by improving the specific object-ray algorithms themselves, but rather by removing as many failure tests as possible. When a specific primary ray is tested against an object in the scene, it will either find a point at which the primary ray intersects an object or it will fail simply because the object does not intersect the ray. However, why should a primary ray be tested against an object if the object does not intersect the ray? The addition of the BSP algorithm allows the primary ray to be tested only with the objects *it is most likely to intersect*. An object that is on the lower-right portion of the screen should not be considered for a primary ray that is in the upper-left portion of the screen. This algorithm allows this to happen.

However, it does so at a price. With each additional recursive level, there is more

and more overhead in processing the binary tree necessary for the algorithm. In other words, it is not worth recursing 20 or 30 levels down the binary tree. Four or five levels were typically enough in the tests discussed in Chapter 5. With the BSP optimization, the full test's frame rate doubled while the simple sphere test's frame rate tripled.

The BSP algorithm works best with small objects that are dispersed throughout the scene. It helps with both large and small quantities of objects. However, it does not do well when the majority of the objects take up a large portion of the screen. The plane test showed absolutely no improvement with the addition of the BSP algorithm. Thankfully, this situation does not occur often, and therefore the BSP algorithm can be used to reliably improve the performance of many applications of ray-tracing.

REFERENCES

[1] N. Wilt, Object Oriented Ray-Tracing in C++, Hoboken, NJ: John Wiley & Sons, 1994.

[2] J. D. Foley, A. Dam, S. K. Feiner, and J. F. Hughes, Computer Graphics: Principles and Practice, New York: Addison-Wesley, 1997.

[3] T. Möller and E. Haines, Real-Time Rendering, First ed. , Natick, MA: A.K. Peters, 1999.

[4] P. Shirley, Realistic Ray Tracing, Natick, MA: A.K. Peters, 2000.