Boston College
Computer Science Department

Senior Thesis 2002
John Weicher
Distributed 3D Raytracing
Prof. William Ames

Introduction

Since their advent, computers have been used to aid humans in tasks that would be too complex or too time consuming to do without them. However, as computers became more and more powerful, they also began to show potential usefulness in areas that before were completely beyond our ability at all. People were finding a use for computers in all areas of activity, and visual art was no exception. The idea arose that perhaps computers could be used to generate pictures that looked so real, a person would not be able to tell that they weren't real photographs to begin with. This concept became know as "photo-realism." Photo-realistic images are those that have been generated by a computer by doing mathematical and geometric calculations based on the physics of the real world, but are images which are indistinguishable from two-dimensional photographs taken of a real life three-dimensional scene. As computers became more powerful, several techniques were developed in attempts to do this. Raytracing is one of those techniques, and is probably one of the most popular 3d image-synthesis techniques in use today. Raytracing is actually a remarkably simple process, providing one has a bit of background understanding first. To understand how raytracing works, and where the inspiration for its development came from, one should have an understanding of how vision works within the human eye.

1. Human Vision

Our eyes are actually nothing more than complex "light catchers." The internal surfaces on the backs of our eyeballs are vast arrays of "rods" and "cones", organic devices that are sensitive to different wavelengths, or energies, of light. As is probably common knowledge now, light has been shown to be a particle that travels at immense speeds, always in a strictly straight line, yet oscillating as it goes. This causes a particle to appear to have a wave-like behavior (Fig. 1).

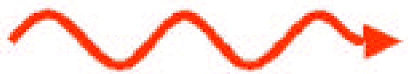


Fig. 1 A Particle of Light

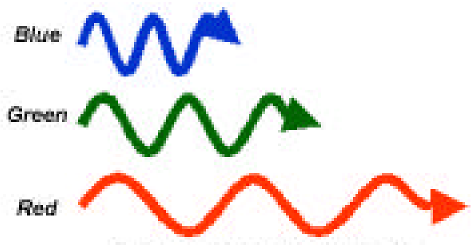


Fig. 2 Different Wavelengths

The varying of the speed of oscillation of a particle enables light to occur in many different wavelengths and energies. It is these different energies of light particles that actually make up the different colors that we can perceive with our eyes (Fig. 2). Our eyes perceive an image by having light particles of different colors enter the eye and strike the rods and cones on the rear surface. These rods and cones then send signals based on the energies of the particles to the brain to be interpreted as colors, and ultimately an image. Without light we would not be able to see anything. And not for the obvious reason of it simply being too dark to see anything, but because it is these particles of light which make the very image our eyes see altogether!

The process of image formation within the eye is very simple. Every scene or environment in which we can see has light sources: things that actively emit light themselves, or things that "emit" light by reflecting it. When an object emits light, such as a light bulb for example, it is actually emitting billions and billions of individual light particles in all directions. These particles of light are generally of all or at least many of the possible wavelengths and colors. They bounce around a scene, such as a room for example, with some being absorbed by various objects, and others reflected. Some of the particles manage, through their chance reflections, to have the necessary path to enter the eye in a straight line and strike the back (Fig. 3). Furthermore, different materials absorb different wavelengths of light. It is the wavelengths which a material or object reflect which give it its color. When we see a red wall for example, it is because the materialsof the wall are absorbing all the colors of light *except* red. Because of this, red light is able to strike the wall, not be absorbed, and perhaps reflect off of it into our eye.

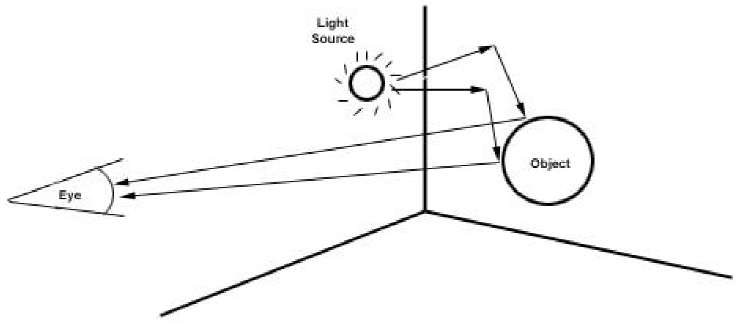


Fig. 3 Human Vision

2. The Raytracing Process

2.1 Forward Raytracing

It is this process through which the human eye perceives images that raytracing tries to mimic. Based on the way light particles produce images within the human eye, the idea behind raytracing is very intuitive. First, a scene and all the objects within it are defined to the raytracing program based on their geometry within a three-dimensional coordinate space. Everything in this world can be described in geometric terms. For example, a ball is really just a three-dimensional sphere, for which we know a geometric equation. Now, this is of course a simplified example for the sake of this explanation. In reality a ball probably isn't a *perfect* sphere. It may have deformities or irregularities in its surface. However, in this way everything (practically) can be defined interms of a collection of geometric primitives, such as spheres, cubes, cylinders, planes, or other shapes defined by higher order geometric equations. Once these objects are defined geometrically, their surface or material properties are defined. These properties would include things like color, how shiny they are or how well they reflect light. Next, light sources are defined according to their location in three-dimensional space, along with their color and intensity. Finally an "eye-point" and direction are defined. This is the location from which the image will be generated. Finally, the raytracing algorithm is applied to the scene. Light rays are simulated leaving the active light sources (those which actively emit light), bouncing off the objects in the scene based on the surface normal of the objects at the points of contact, having their colors altered depending on what objects they come in contact with. Determining the color of light rays that eventually come in contact with the eye-point forms an image (Fig. 4).

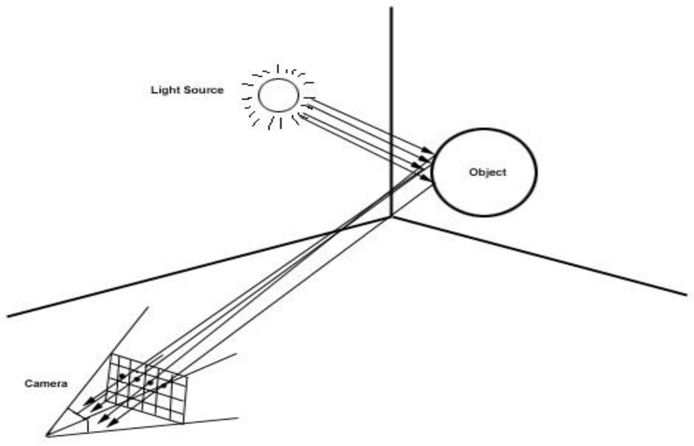


Fig. 4 Forward Raytracing

This process is more accurately called *forward raytracing*, as it models how light rays actually leave their source and travel forward in their journey until they either reach the eye-point or it is determined they never will. In *theory* this is an algorithm which more or less perfectly models real life, and therefore should generate a photo-realistic image. However, in *practice*, this type of algorithm is never implemented. Because of the nature of light, it would take the simulation of potentially billions of rays, each one involving numerous calculations to determine which objects it came in contact with or intersected along the way, before finding all the rays which actually enter the eye-point and form an image. Using this algorithm would take huge amounts of time to actually generate an image, because it involves doing the extra calculations for a vast majority of rays that end up being useless anyway. However, by making a slight alteration to the forward raytracing process, we get a much more practical and usable algorithm.

2.2 Backward Raytracing

By modifying the forward raytracing algorithm and simulating only the rays that we are sure will actually enter the eye-point and be relevant to the formation of an image, we can greatly reduce the number of unnecessary calculations. Therefore we will have a much faster algorithm; one that can actually be effectively implemented. However, to do this we must abandon the notion of tracing light rays from their source to their origin, as we have no idea which of the infinite number of rays are actually the ones that make it to the eye point. Instead we must trace light rays from the eye point into the scene, and determine which objects they intersect to determine what colors they will be. This process is appropriately called *backward raytracing*, as it involves simulating rays backward from their final destination to their source.

3. Implementation

Now that I have explained *what* backwards raytracing is, it is now appropriate to discuss my particular implementation of a backward raytracer. This will also serve to better explain how the raytracing process is actually achieved. I have chosen to implement my raytracer in the C++ language, as C is a very efficient procedural language. Although a language such as Java would have offered a much easier means of opening windows and plotting pixels, as well as of doing network socket operations, the additional overhead that Java brings is unacceptable, considering how time consuming the raytracing process already is. I chose instead to utilize the OpenGL libraries available to Linux for opening windows and plotting pixels, as OpenGL provides a relatively simple way to do this. Furthermore, C++ has the ability to incorporate an object-oriented structure, which was very desirable. Using an object-oriented approach, I was able to easily construct a hierarchical object structure for the objects that my raytracer is able to trace. These individual classes, one for each object, contain all the specific information about the object, such as location, size, as well as physical properties such as color, transparency, etc. These classes also include the object specific methods for determining the intersection between an object of its type and a line.

In addition, I chose to implement my raytracer on a Linux platform, because the network and socket programming that was necessary for the distributed aspect of my program is much more straightforward that on a Windows platform. However, the distributed aspect of this project will be discussed further on in this paper.

3.1 Ray Construction

Because backward raytracing is essentially the process of determining the intersection points of light rays and objects within the scene, an appropriate representation of these components is needed. For the light rays, this can be achieved by representing them as lines. This way we can geometrically solve for the intersections that each of these lines has with the objects in the scene, which are also represented by geometric equations. This will determine which objects the light ray, which this line represents, would have reflected off of.

It is a well-known fact in geometry that any two points make up a line. Therefore, given two correct points for each, we can define the lines that make up the light rays we wish to simulate, or "trace". Once we have these lines we can solve for their actual intersections. But again, so far this is still just a reiteration of the raytracing process in general. We still need a way to determine *which* rays are the ones that we know are going to affect the image. We can do this by observing the way an image is formed on a screen. An image is actually a two-dimensional array of pixels. A pixel is the smallest component of color a screen is able to display. It is a single "dot" of color on a computer screen. We know that the color of each pixel the image we want to create is going to be determined by the ray that must pass through both it and the eye point. This means that the pixel represents a point on the actual light ray line. Using the eye point as one, and the pixel point within the image as a second, we can construct all the lines that we know will create the image by tracing one ray per pixel (Fig. 5).

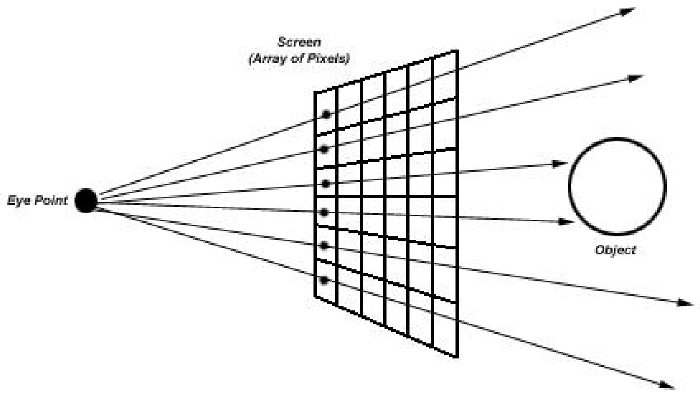


Fig 5. An example of using the pixel locations to isolate the desired rays that are of use to the image

Any point within a three-dimensional coordinate system can be described by three components: the ‘x’, ‘y’, and ‘z’ components of its location. If a user were to sit in front of the screen, the x and y components would represent the horizontal and vertical component of a point, respectively, while the z component measures how far “in” or “out” of the screen the point is. The screen represents a ‘z’ of zero, with negative values increasing “into” the screen. By placing the eye point essentially “outside” of the screen, we are able to use it as one point to define each ray. I created a data structure comprised of three floats, called a “point3d”, to represent a point in the coordinate system. A “ray” is another data structure made up of two point3d variables.

3.2 Object Construction

Information about objects within the scene is stored within C++ classes for these objects. Characteristics such as location, orientation, and surface properties are stored in variables within the object itself. This way theobject “knows” all it needs to know about itself within the 3d world. Also present within the object are all the methods necessary to compute intersections. The main raytracing engine can therefore pass to the object the two points that make up a ray, and the object determines whether or not the ray intersects it. This was a design decision I made because solving for the intersection between a ray and different types of objects is different for each object. This way the raytracing engine needs only to iterate through rays, polling the objects in the scene for intersections, and needs to know nothing of how to solve for intersections with different objects. The objects handle that work themselves.

3.3 Finding the Intersection (and The Depth Problem)

The problem with simply polling the objects for their intersections is that a ray may intersect multiple objects. For example, if there is one object positioned behind another, it is possible that a ray fired at the first object will intersect the second as well. Because we are just doing geometric calculations, there is no mechanism for “stopping” a ray once it has its primary intersection with an object. Therefore, every time more that one intersection is calculated for a ray, it would need to be determined which intersection is "first" or closer to the eye. The algorithm would need to do this to ensure that the proper object's surface information is used to color a pixel, and not the surface information of the object that should actually be covered or obscured by another.

We can solve this problem using parametric equations. Given some ray comprised of two points, p0 and p1, each having an x, y, and z component, we can generate any point further along the ray by using the following equations:

$$X_i = X_{p0} + t(X_{p1} - X_{p0})$$
$$Y_i = Y_{p0} + t(Y_{p1} - Y_{p0})$$
$$Z_i = Z_{p0} + t(Z_{p1} - Z_{p0})$$

In other words, there is some coefficient t, such that when inserted into each of these equations, generates the three components of intersection point on the surface of the object. For example, take the equation of a sphere:

$$(X-a)^2 + (Y-b)^2 + (Z-c)^2 = r^2$$

This equation defines all the points on the surface of a sphere centered at (a, b, c) with a radius of r. If we simply assume that there is an intersection between a ray and this object, we can solve for "when" (or for what t) this intersection occurs along the ray line. By simply plugging the equations for X_i, Y_i, and Z_i in for the X, Y, and Z, of the sphere equation, we can solve for t. If we therefore do this for all the objects in the scene using the same ray, and have the objects return their t values instead of their actual intersection point (which means nothing to the raytracing engine anyway), we can simultaneously calculate what objects intersect the ray, as well as which object does it first. The object that returns the smallest positive t clearly has an intersection with the ray earlier along its path from the eye, and is therefore the intersecting object that is most "in front." This object can then be polled for its color at that point on its surface, and the pixel can be colored.

3.4 The Algorithm

My raytracing engine goes through the following simple algorithm to generate an image. This assumes the scene has already been constructed and passed to the raytracing engine through the appropriate methods. This process of actually constructing the scene will be covered later in this paper. The algorithm follows:

```
First point of all rays is always the eye point
- For each y pixel value in the image
  - For each x pixel value in the image
    - Set the second point of ray to (x, y, 0)
    - Fire ray at scene by passing ray to each object
    - Collect t values from all objects
    - Compare which object returned the smallest positive t
    - Poll this object for its color at the intersection point
```

- Set that pixel to the returned color

All complex calculations are performed within the objects themselves. The main raytracing loop needs only to poll all the objects for their t values, and then request the intersection point color from the appropriate object. Objects with no valid intersection with the ray simply return a negative one for t. In the process of solving for t, each object can also determine the actual point of intersection on its surface. Once it has solved for t, an object can immediately solve for the intersection by putting the ray points and t into the original parametric equations for X_i, Y_i, and Z_i. This intersection can then be stored in the object itself in case the object is later polled for its color at this point.

3.5 Determining Color Values

To determine the color of an object at a certain point, my raytracer implements a simple Lambertian Shading model. Within the Lambertian model, as with most shading models, the point of an object that is going to be most brightly lit is the point on the surface of the object where the perpendicular surface normal of the object at that point happens to be aimed directly at the light source illuminating the object. Colors get proportionally dimmer as the distance from this point. An example of this is the bright highlight on a balloon that is held next to a light. The bright spot always occurs on the balloon in the spot where, if we were able to draw a line perpendicular to the surface of the balloon, the line would point directly to the light bulb. Points on the surface of an object are lit at an intensity which is inversely related to the magnitude of the angle between a vector pointing at the light, and the surface normal vector of the object at that point (Fig. 6). As this angle increases, the object has a lesser degree of illumination by the light source.

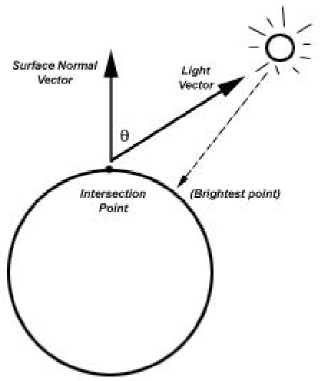


Fig. 6 Lambertian Illumination Model

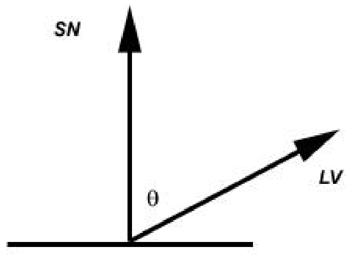


Fig 6b.

The intensity of a pixel's color using this shading model can be defined as:

$$I = C_p * \cos \theta$$

Where C_p is the red, green, or blue component of the original object color (perhaps specified by a texture map), and θ is the angle between the vector pointing towards the light, and the surface normal. Therefore, to calculate the color of an object at an intersection point, all that is needed is the surface normal at that point, a vector pointing to the light source from that point, and the object's original color. Cos θ can be easily computed by taking the dot product of the two vectors after normalization. An example of this calculation is below:

Normalize both vectors:

$$V_L = \text{sqrt}(LV_x^2 + LV_y^2 + LV_z^2)$$
$$LV_{norm} = (LV_x/|V_L|, LV_y/|V_L|, LV_z/|V_L|,)$$

$$V_{SN} = \text{sqrt}(SN_x^2 + SN_y^2 + SN_z^2)$$
$$SN_{norm} = (SN_x/|V_L|, SN_y/|V_L|, SN_z/|V_L|,)$$

Compute the Dot Product:

$$\frac{(LV_x * SN_x)}{(V_L * V_{SN})} + \frac{(LV_y * SN_y)}{(V_L * V_{SN})} + \frac{(LV_z * SN_z)}{(V_L * V_{SN})} = \text{Cos } \theta$$

Apply to original color:

$$C_R = C_R * \text{Cos } \theta$$
$$C_G = C_G * \text{Cos } \theta$$
$$C_B = C_B * \text{Cos } \theta$$

This method accurately shades an object based on its location relative to a light source. And example image of Lambertian shading is included in Appendix B of thispaper.

3.6 Reflectivity

Lambertian shading is only an effective coloring method providing an object has nothing more than a simple color. However, what if an object is to be slightly mirrored? In other words, what if it reflects the light around it? This scenario is also handled accurately through a few more simple calculations. If a ray intersects a mirrored object, the color of that point on the object must be determined by calculating what that ray would intersect with after it "bounces off" that first object into a second.

When an object is polled for its color and it is mirrored, instead of simply returning its own color, it assembles a new ray along the properly reflected vector of the original ray, and fires this *new* ray at the scene to see what *it* hits. This new color is then factored in to the original object's color proportionally to the amount of mirroring. This new modified color is then returned as the pixel color (Fig. 7).

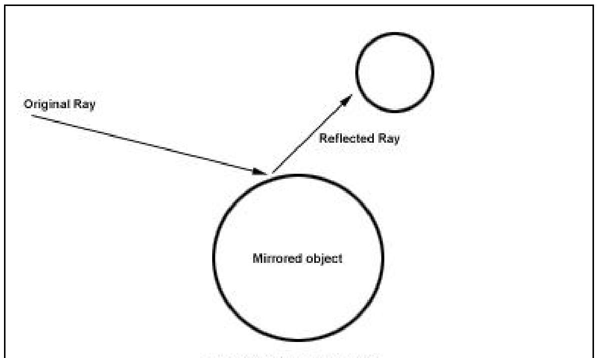


Fig. 7 Typical Ray Reflection

The computation for a reflected vector, much like that of Lambertian shading, is dependant only on the incoming ray (vector), and the surface normal vector of the object at the intersection point. The angle of the incoming vector relative to the surface normal of the object at that point is the same as that angle that the reflected vector will make with the surface normal at the same point (Fig. 8).

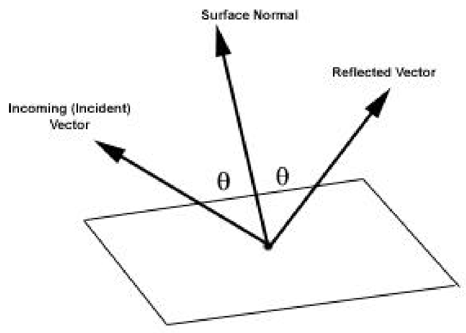


Fig. 8 Reflection Illustration

A reflected vector can be computed using the formula:

$$V_{REF} = 2 * (V_{INC} \bullet V_{SN}) * V_{SN} - V_{INC}$$

The operation within the parenthesis is the dot product operation, just like in the computation for Lambertian Illumination. I will not go through an example of this computation within this paper, as it is a bit time consuming. It is sufficient to know however, that this calculation generates a reflected vector that can be used to generate the reflected ray of a mirrored object.

4. Additional Implementation Information

4.1 Matrix Transformations

One of the problems that I encountered when initially designing my implementation was the deriving of the equations that actually solve for the intersection between the objects and a line. This was often a very tedious and difficult task. For example, deriving the equation to solve for the parametric coefficient “t”, in an intersection between a line and a sphere is relatively simple, albeit a little long. This is because the equation for a sphere already has built into it the ability to easily define an *arbitrary* sphere (one centered at an arbitrary location), and not just one centered at the origin of the coordinate system, or a *unit* sphere. The problem is that for other objects, in particular infinite objects, such as planes, the equations define a set of points in such a manner that deriving the intersection equation for an *arbitrarily oriented* plane was extremely complex. To do this for even more complex objects, such as a torus (a donut shape), while incorporating to ability to have arbitrary definitions was next to impossible, even with the help of software such as Mathematica. Of course these derivations are not literally impossible, but they were beyond my means.

Once I became aware of this problem, I decided to make a design decision that would handle all of these problems. I decided to implement all transformations of objects such as location, rotation, and scale (size), by way of matrices. By doing so, I would only need to derive the intersection equations for *unit* objects, or those centered at the origin, having no arbitrary rotation or scaling. The process is quite simple. By putting any point in a 4x1 matrix, and multiplying it with certain matrices for each of the transformation operations, you get the corresponding point with all the proper transformations applied. For example, if I have the point (0,1,0), and want to know what point results if I rotate it around the x-axis 90 degrees, I can simply multiply this point by the matrix for x-axis rotation, and get the resulting point (0,0,1) (Fig. 9).

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

Fig. 9 Example of a 90 degree rotation around the x axis using matrix multiplication

There are corresponding matrices for each of the possible transformations (which I have listed in an appendix at the end of this paper): Rotation, translation or relocation, and scaling, all three of which can be applied to each of the three axes independently. Furthermore, by multiplying all the desired transformation matrices together in the reverse order from which you would like them applied, the resulting matrix is a Composite Matrix with which all the transformations can be applied to a point at once. Even *further*, an Inverse Composite Matrix can be computed which exactly reverses the transformation s of the Composite Matrix. With these two matrices, all the problems of deriving complex equations to be used for finding intersections are no longer relevant.

When a user defines an arbitrary object, they now only need to describe the object in terms of the transformations they wish to have applied to it. A sphere of radius of 5, located at (10,20,30) is the same thing as a unit sphere of radius 1 centered at the origin with the correct scale and translation then applied to it. With this tool, I no longer needed to do complex derivations of equations.

Now, before raytracing beings, the Composite and Inverse Composite Matrices need to be computed for each object in the scene. The raytracing algorithm then changes in the following way:

- When an object is polled for its intersection, it applies its Inverse Composite Matrix to the *incoming ray*, in order to make it relative to a non-transformed, more simple, unit-version of the object. Because the Inverse Composite Matrix applies the exact opposite of the transformations that the user wanted applied to the object, this is the same as firing the original ray at transformed object.
- Then it solves for the intersection of this transformed ray with a unit version of an object of its type (these equations are much simpler).
- The resulting intersection point is the transformed back to its *actual* location

using the Composite Matrix of the object.

This process produces the same resulting intersections as would be computed using the original ray and the much more complicated equations derived for arbitrary versions of every object. By using this process of matrix transformations, I eliminated the need for doing these derivations. Matrices also allow for an easy means to stretch, rotate, and move objects in creative ways.

4.2 Anti-Aliasing

One of the problems with raytracing is the fact that the pixels of a computer are a finite size, and can only be set to one color. Because pixels are the smallest unit of color on a screen, it is impossible to set one half of a pixel to one color, and the second half to another. This causes problems because situations can arise (and usually do), in which if we could "zoom in" on a scene, we would notice places in the image where the edge of an object really only should cover part of a pixel. This usually occurs because pixels are often represented as a square. Therefore, trying to represent curved edges in particular usually results in an edge that looks "jagged." In Figures 10 and 11, we see how trying to represent a true circle with square pixels is impossible. Figure 10 represents the circle we would *like* to draw on the screen, but Figure 11 shows the "circle" we have to settle with due to the nature of pixels:

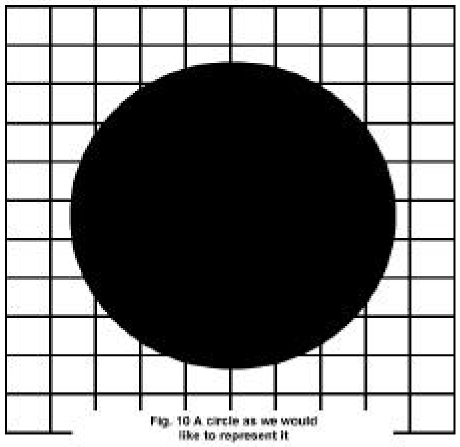


Fig. 10 A circle as we would like to represent it

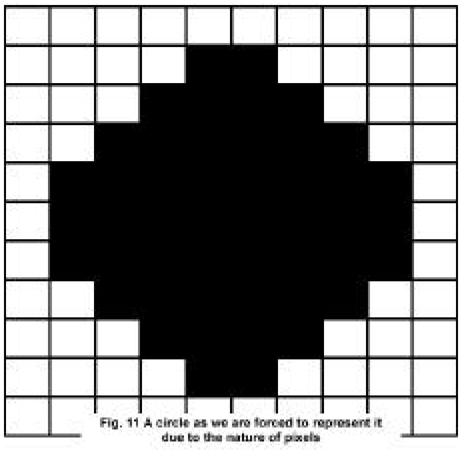


Fig. 11 A circle as we are forced to represent it due to the nature of pixels

In this case it would be ideal to be able to color only part of certain pixels black, and the rest white. This negative side affect of the shape of pixels is called *aliasing*.

There are techniques to correct this problem, however. These techniques are appropriately called *anti-aliasing* techniques. My program implements a simple form of anti-aliasing, which can be turned on or off. When anti-aliasing is enable, the raytracer fires not just one ray per pixel, but several (Fig. 12). Each ray is offset slightly to various locations *all within in the same pixel*. This way, if a pixel should ideally be partially colored by more than one color, some of these rays are going to return these different colors. The pixel is then actually set to a color that is an average of all the colors that are returned by the anti-aliasing rays (Fig. 13). This sometimes produces a blurring affect along the edges of objects, but works very well to eliminate "the jaggies."

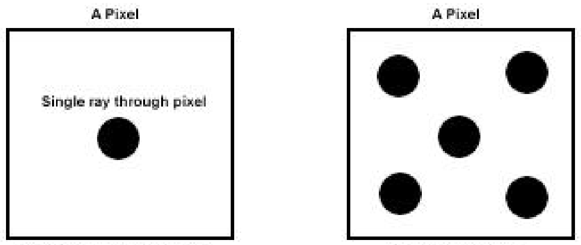


Fig 12. Without anti-aliasing With Anti-aliasing

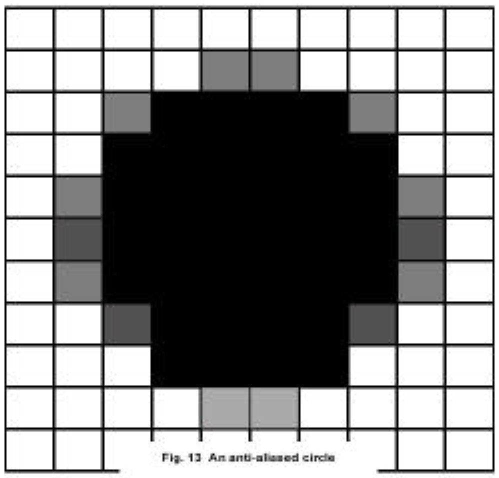


Fig. 13 An anti-aliased circle

When viewing this circle at its normal size, and not enlarged to the pixel level as it is above for the sake of explanation, it would appear as a much more accurate circle. It should also be noted that while anti-aliasing makes an image look more realistic and servers to smooth edges, it obviously takes much longer. In the case of my implementation, there are five times as many rays fired, and so five times the number of calculations to perform per pixel. I have included other examples of anti-aliasing that has been applied to actual images generated by my raytracer in Appendix B of this paper.

4.3 Shadows

Another, very simple to implement component of raytracing is shadowing. To make an image more realistic, objects that are between a light source and other objects should cast shadows on objects behind it. This can be implemented very easily by using "shadow rays." When an object is found to intersect a ray fired from the camera, and to be the object that is most "in front" and so is polled for its color information, it computes its own color at that point, then fires a shadow ray. A shadow ray is simply a ray shot from an intersection point on an object, directly at the light source. If any valid intersections are detect which are not "beyond" the light source, then there is clearly a second object between the light source and the first object. The original object then darkens its color by a preset amount to signify being in shadow, before returning this color to the raytracer. Shadowing is one of the easiest features of raytracing to implement.

4.4 Texture Mapping

The vast majority of objects in this world are not made up of just a single color. Most are comprised of multiple colors or hues. Because of this, it would be nice if there were a practical way applying or "painting" a custom face or "skin" onto an object. Well in fact, this is a very simple thing to do in raytracing. The process through which this is done is called *texture-mapping*. When defining an object within a scene file, the "texture" attribute can be included, along with the name of a supported image file. Upon parsing of the input file and the creation of the scene within the raytracer, this picture file can be opened and read into a buffer within the object class. Additionally, the object classes need to be modified to include methods that determine an appropriate "longitude" and "latitude" of an intersection point on the object, relative to the entire object. This way, when a ray intersects an object and that object is polled for its color, instead of just returning a simple color, the object first determines the longitude and latitude of the intersection point, and maps this to a location in the texture image. The color of the pixel in the texture image at the mapped location is the color that is returned by the object. This process serves to "wrap" the image around the object, much like the peel of an orange.

My initial raytracing program supported texture mapping using image files of the PPM format, a simple bitmap format. The current implementation of my distributed raytracer, however, does not support texture mapping. This is because the client would need to send to a server not just the source file, but also any files that were needed as textures. This way a server would have these needed files when they parsed and built a scene for themselves. Currently, I have not implemented a mechanism within the client to parse a source file prior to sending it, in order to determine additional files that would be needed and to send them as well.

4.5 Scene Construction and File Format: The Parser

One topic that I have not yet discussed is that of how a scene is actually inputted or defined to the raytracer. For this task I had to develop a parsing module for the raytracer, as well as a file format in which a user could describe a scene properly and give it to the raytracer.

The ".ray" format that I decided upon is a very simple text markup language consisting of "tags" which are flags to the parser, such as [OBJECT] and [GLOBAL], and a series of keywords used to set attributes of an object to a particular value. For example, to create a red sphere centered at (10, -10, 10), with a radius of 5.25, the user would put in their source file the following lines of text:

```
[OBJECT]
type=sphere
radius=5.25
color1=<1.0, 0, 1.0>
translate=<10, -10, 10>
[/OBJECT]
```

Colors are specified by a 3-tuple of their red, green, and blue components. Attributes other than the ones seen above are available, such as "mirror", "reflection", and "color2", which causes an interesting checkerboard affect on the object. Objects can be scaled, rotated, and moved through the use of the "scale", "rotate", and "transl ate" keywords. These keywords essentially tell the parser to set the object's transformation matrices to the desired value. A user also has the ability to set certain global scene parameters, such as the ambient lighting value of the scene, the shadowing factor, or to enable or disable anti-aliasing. These attributes must be defined between the [GLOBALS] and [/GLOBALS] tags at the beginning of the file.

A scene file is passed to the raytracer as a command-line argument. The parsing module then parses the file line by line. Object classes are instantiated, and have their attributes set as they are encountered within the file. They are organized within the raytracer as a linked list, which allows an easy, efficient way to query all objects within the scene, as well as allowing for the number of objects within a scene to be dynamic. Resources need not be pre-allocated to accept a certain maximum number of objects, a number that might not be reached with every scene.

An example of an entire scene file is included in Appendix A of this paper.

5. Distributed Raytracing

It is finally appropriate to discuss the second portion of this project. The idea of *distributed* raytracing is not a complicated one, and so very little time is actually needed to explain it. However, it is extremely effective in increasing the performance of the raytracing algorithm, and so is an important adaptation to standard raytracing.

Although the calculations done to determine the color of each pixel are the same (of the same form of course, variables do differ), they are all completely independent of each other. Because of the nature of the raytracing processes, a ray shot through a pixel into a scene never has any influence on other rays shot through other pixels, nor is it dependant on others. Therefore, determining the *color* of one pixel is totally unrelated to determining the color of another. This means that a raytracer could begin doing the calculations for a second pixel, before it is even done with a first. This fact allows the raytracing algorithm to lend itself extremely well to being distributed over multiple computers, all doing the calculations for different portions of the same image.

5.1 Implementation

In order to incorporate distributed processing into my raytracer, the structure of the program had to change. My raytracer in actuality had to become two separate programs. What was at first a stand-alone raytracing application became a collection of a main client program, responsible for breaking up the work and distributing it, and one or more "number-crunching" servers. It is the server components that actually implement the raytracing algorithm. The client component simply distributes the work of one image evenly throughout available servers, and displays the resulting image generated by the servers. The process works as follows:

1) The client program is started with command line arguments of the scene file that is to be traced, as well as an arbitrary number of servers to distribute the work among.
2) The server addresses are checked to make sure they are valid.
3) For each server, a portion of the client program is threaded off for each server to handle communication between it and the server.
4) Each thread then connects to a server, and once a valid connection is made to the waiting server, the scene file and a range of pixels to trace is sent.
5) The server then parses the file and builds its own copy of the scene. It then raytraces the range of pixels that was allocated to it, storing the resulting pixel locations and color values in a buffer.
6) Once a server has finished all its calculations, it sends the contents of its buffer back to the waiting client thread.
7) Upon receiving pixel information from the server, the client thread transfers this information into a global image buffer, and terminates.
8) Once all threads have completed their communication with their respective servers, and all pixel data has been received, the main client thread opens a window and displays the completed image.

The main client program basically becomes one that only has the capabilities of network communication and the ability to open a window and plot pixels via OpenGL. Likewise, the server programs have all the necessary capabilities to raytrace a scene, but no ability to display it, as this would not be necessary.

Distributed raytracing is obviously very effective because it significantly reduces the amount of time it takes to render a single image. This is because the work for an image is not being done by a single computer, but is broken up among many computers all doing work concurrently. It is a very simply process, but one that is very effective.

6. Improvements

It is of course appropriate for me to mention a few aspects of this project that I wish I had been able to do differently, or to do at all. As it stands, my program only supports the raytracing of spheres and planes. If there had been more time available, I would have liked to incorporate more objects, such as the torus, cylinder, cone and cube, to at least support all the geometric primitives. The derivations of intersection equations for these objects even in their unit forms, were things I did not have time to do.

I also would have liked to implement the idea of allowing an arbitrary camera point into the program. As the program stands now, the eye point is fixed and cannot be moved or specified. Because of this a scene must be defined within view or there is really no point to the raytracing. I believe this process would not have involved much more that a series of matrix transformations on the camera, but was one that I could not spend time on developing.

Finally, I would have liked to be able to incorporate the use of multiple light sources. It is very rare that a scene looks good while using a single point light source, and so I would have liked to have been able to support more. This too would not have been too difficult if I have thought of it earlier on in the design process. When it finally occurred to me, too much of my application was already coded hard fast to the idea of a single light source.

These are the three primary improvements that I wish I could have made to my program. Obviously there are countless other *features* that I would have like to incorporate as well, such as the ability to create more complex objects through the use of the boolean operations (union, intersection, subtraction) of simple objects. But this type of feature, as well as most others, would probably require a major change in the design of my program in order to be supported. With this said, I am extremely satisfied with the way my project came out.

Conclusion

Raytracing is not without its faults. The raytracing algorithm is incredibly time consuming as it usually involves the solving of very complex geometric formulas, and doing this thousands, if not millions of times in order to create a single image. Yet because of the nature of the raytracing algorithm, it is also one that is a great candidate for distributed computing. By designing a raytracing application so that it utilizes multiple computers to complete the work that would have otherwise been done by just one, we can drastically reduce the time cost of rendering.

With this said, raytracing is a very intuitive way of creating photo-realistic images on a computer. By simulating the way light behaves in the real world, and how the behavior works to “create images” within our own eyes, we are able to create incredibly realistic pictures of objects or scenes based purely on geometry and math. It is also relatively simple to effectively implement a basic raytracing application, as I think I have shown through my project and paper. Because of this, raytracing has become one of the most popular and widely used techniques for creating stunning visual images.

Bibliography

Angel, Edward, Interactive Computer Graphics: A Top-down Approach with OpenGL. (New York: Addison Wesley Longman, 2000.)

Glassner, Andrew S., Ed. An Introduction to Ray Tracing. (New York: Academic Press, Inc., 1989.)

Ma, Kwan-Liu, Painter, James S., Hansen, Charles D., Krogh, Michael F. "A Data Distributed, Parallel Algorithm for Ray-Traced Volume Rendering." ACM Computer Graphics. (SIGGRAPH Proceedings 1993). (New York: ACM Press, 1993.)

A special thanks to Prof. William Ames for all his time and help.

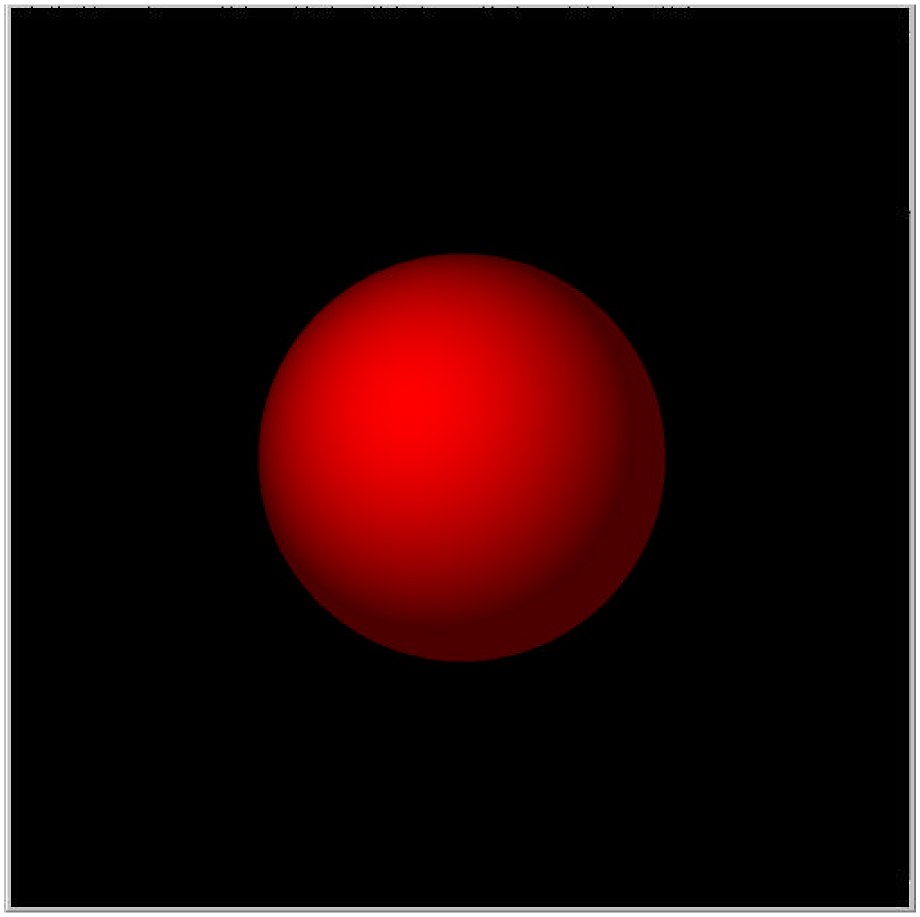
Appendix A – Scene File Format

Below is the format for a scene file:

```
[rayfile]

[GLOBALS]
attribute1 = value1
attribute2 = value2
…
attributen = valuen
[/GLOBALS]

[OBJECT]
type = value
attribute1 = value1
attribute2 = value2
…
attributen = valuen
[/OBJECT]

…

more object definitions

…

[/rayfile]
```

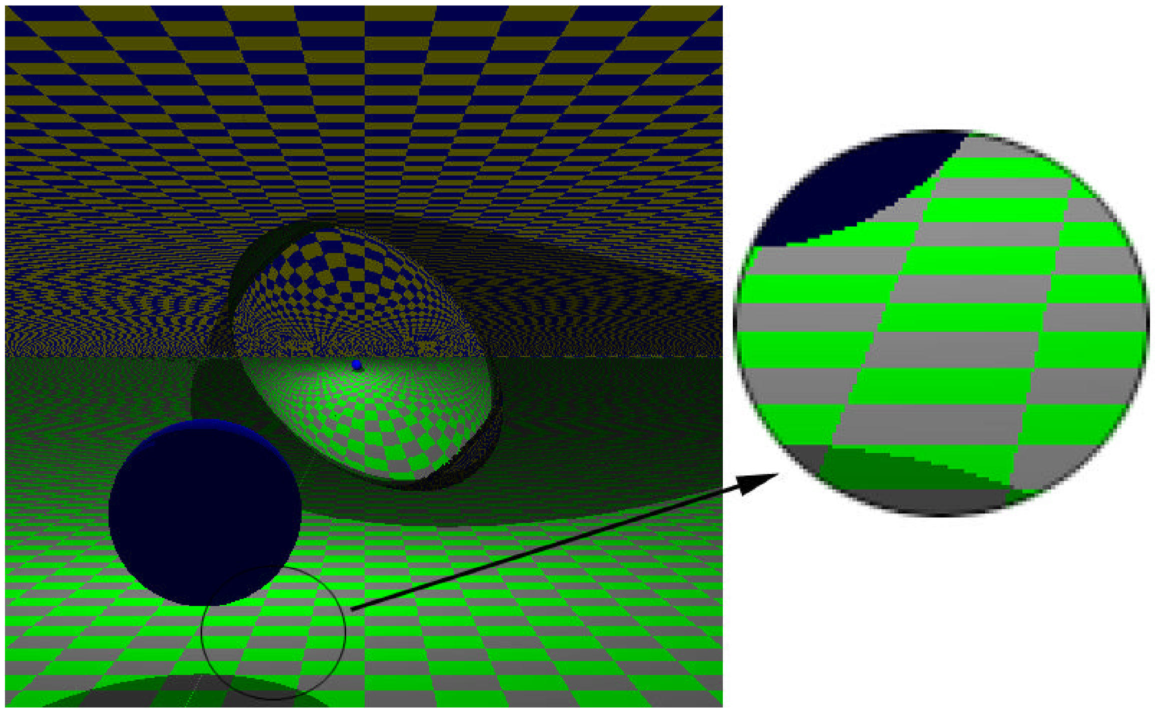
Appendix B – Samples Images

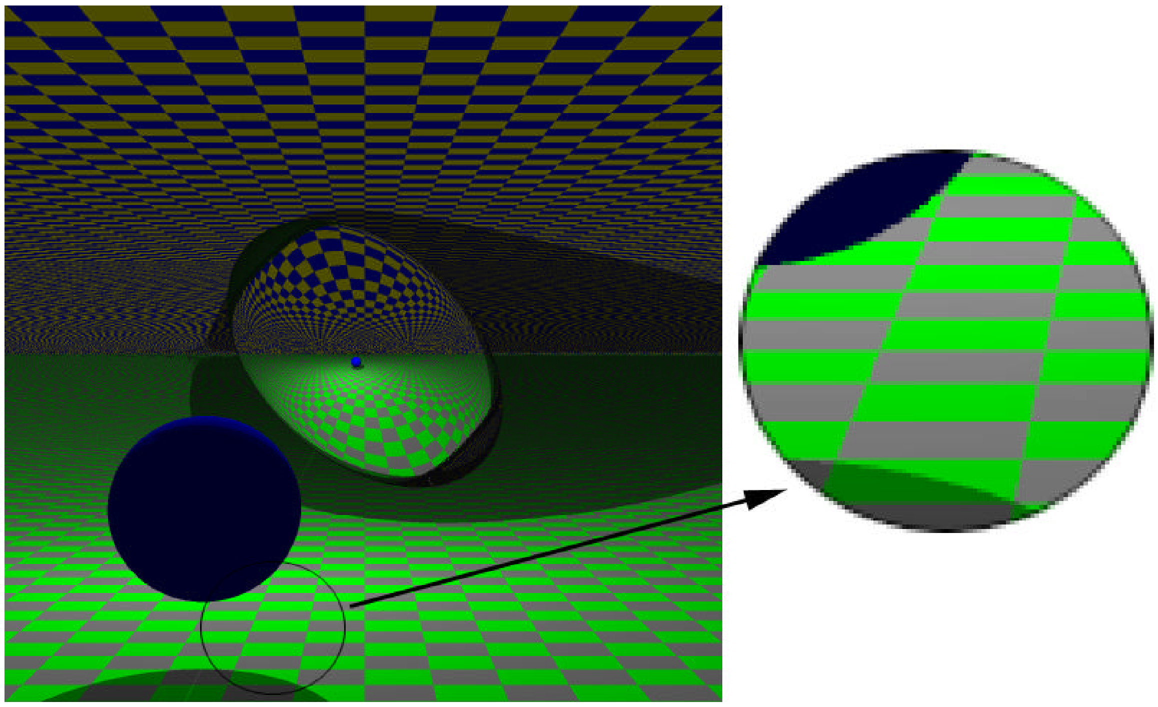
Below is an example image of the Lambertian shading model used on an illuminated sphere:



(App. B Con't)

Below are two sample images illustrating anti-aliasing. The same segment has been enlarged in both images. However, in the first image, the enlarged segment of the image shows jagged edges in the plane, and the edges of the sphere and shadow. The second image has anti-aliasing applied, and it shows how the edges are now softer, and looked more realistic when not "zoomed in" to the pixel level.





Appendix C – Transformation Matrices

Below are the matrices through which the various transformations can be applied to a point: Translation, Scaling, and Rotation in each of the three axes.

Translation

$$\begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scale

$$\begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

X-Axis Rotation

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Y-Axis Rotation

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Y-Axis Rotation

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Appendix D – Source Code

Below is the listing of all source code:

<u>Client:</u>
Main.cpp
Main.h

<u>Server:</u>
Server.cpp
Server.h
Tracer.cpp
Tracer.h
Sphere.cpp
Sphere.h
Plane.cpp
Plane.h
Parser.cpp
Parser.h
Error.cpp
Error.h

A printout of these source code files follow this page.