Crossroads
The ACM Student Magazine

Ray Tracing: Graphics for the Masses

*by **Paul Rademacher***

Although three-dimensional computer graphics have been around for several decades, there has been a surge of general interest towards the field in the last couple of years. Just a quick glance at the latest blockbuster movies is enough to see the public's fascination with the new generation of graphics. As exciting as graphics are, however, there is a definite barrier which prevents most people from learning about them. For one thing, there is a lot of math and theory involved. Beyond that, just getting a window to display even simple 2D graphics can often be a daunting task. In this article, we will talk about a powerful yet simple 3D graphics method known as **ray tracing**, which can be understood and implemented without dealing with much math or the intricacies of windowing systems. The only math we assume is a basic knowledge of vectors, dot products, and cross products. We will skip most of the theory behind ray tracing, focusing instead on a general overview of the technique from an implementation-oriented perspective. Full C++ source code for a simple, hardware-independent ray tracer is available online, to show how the principles described in this paper are applied.

What is Ray Tracing?

Ray tracing is a technique for rendering three-dimensional graphics with very complex light interactions. This means you can create pictures full of mirrors, transparent surfaces, and shadows, with stunning results. We discuss ray tracing in this introductory graphics article because it is a very simple method to both understand and implement. It is based on the idea that you can model reflection and refraction by recursively following the path that light takes as it bounces through an environment

Figure 1. Ray traced image

Some Terminology

Before presenting the full description of ray tracing, we should agree on some basic terminology. When creating any sort of computer graphics, you must have a list of objects that you want your software to render. These objects are part of a **scene** or **world** (**Figure 2**), so when we talk about ``looking at the world,'' we're not posing a philosophical challenge, but just referring to the ray tracer drawing the objects from a given viewpoint. In graphics, this viewpoint is called the **eye** or **camera**. Following this camera analogy, just like a camera needs film onto which the scene is projected and recorded, in graphics we have a **view window** on which we draw the scene. The difference is that while in cameras the film is placed *behind* the aperture or focal point, in graphics the view window is in *front* of the focal point. So the color of each point on real film is caused by a light ray (actually, a group of rays) that passes through the aperture and hits the film, while in computer graphics each pixel of the final image is caused by a simulated light ray that hits the view window on its path towards the eye. The results, however, are the same.

As you may have guessed, our goal is find the color of each point on the view window. We subdivide the view window into small squares, where each square corresponds to one pixel in the final image. If you want to create an image at the resolution of 640x400, you would break up the view window into a grid of 640 squares across and 400 square down. The real problem, then, is assigning a color to each square. This is what ray tracing does.

Figure 2. The eye, view window, and world.

How it works

Ray tracing is so named because it tries to simulate the path that light rays take as they bounce around within the world - they are *traced* through the scene. The objective is to determine the color of each light ray that strikes the view window before reaching the eye. A light ray can best be thought of as a single photon (although this is not strictly accurate because light also has wave properties - but I promised there would be no theory). The name ``ray tracing'' is a bit misleading because the natural assumption would be that rays are traced starting at their point of origin, the light source, and towards their destination, the eye (see **Figure 3**). This would be an accurate way to do it, but unfortunately it tends to be very difficult due to the sheer numbers involved. Consider tracing one ray in this manner through a scene with one light and one object, such as a table. We begin at the light bulb, but first need to decide how many rays to shoot out from the bulb. Then for each ray we have to decide in what direction it is going. There is really an infinity of directions in which it can travel - how do we know which to choose? Let's say we've answered these questions and are now tracing a number of photons. Some will reach the eye directly, others will bounce around some and then reach the eye, and many, many more will probably never hit the eye at all. For all the rays that never reach the eye, the effort tracing them was wasted.

Figure 3. Tracing rays from the light source to the eye. Lots of rays are wasted because they never reach the eye.

In order to save ourselves this wasted effort, we trace only those rays that are *guaranteed* to hit the view window and reach the eye. It seems at first that it is impossible to know beforehand which rays reach the eye. After all, any given ray can bounce around the room many times before reaching the eye. However, if we look at the problem *backwards*, we see that it has a very simple solution. Instead of tracing the rays starting at the light source, we trace them backwards, starting at the eye. Consider any point on the view window whose color we're trying to determine. Its color is given by the color of the light ray that passes through that point on the view window and reaches the eye. We can just as well follow the ray backwards by starting at the eye and passing through the point on its way out into the scene. The two rays will be identical, except for their direction: if the original ray came directly *from* the light source, then the backwards ray will go directly *to* the light source; if the original bounced off a table first, the backwards ray will also bounce off the table. You can see this by looking at **Figure 3** again and just reversing the directions of the arrows. So the backwards method does the same thing as the original method, except it doesn't waste any effort on rays that never reach the eye.

This, then, is how ray tracing works in computer graphics. For each pixel on the view window, we define a ray that extends from the eye to that point. We follow this ray out into the scene and as it bounces off of different objects. The final color of the ray (and therefore of the corresponding pixel) is given by the colors of the objects hit by the ray as it travels through the scene.

Just as in the light-source-to-eye method which takes a very large number of bounces before the ray ever hits the eye, in the backwards method it might take many bounces before the ray every hits the light. Since we need to establish some limit on the number of bounces to follow the ray on, we make the following approximation: every time a ray hits an object, we follow a single new ray from the point of intersection *directly towards* the light source (**Figure 4**).

Figure 4. We trace a new ray from each ray-object intersection directly towards the light source

In the figure we see two rays, **a** and **b**, which intersect the purple sphere. To determine the color of **a**, we follow the new ray **a'** directly towards the light source. The color of **a** will then depend on several factors, discussed in **Color and Shading** below. As you can see, **b** will be shadowed because the ray **b'** towards the light source is blocked by the sphere itself. Ray **a** would have also been shadowed if another object blocked the ray **a'**.

Reflection and Refraction

Just as shadows are easily handled, so are reflection and refraction. In the above example, we only considered a single bounce from the eye to the light source. To handle reflection, we also consider multiple bounces from objects, and to handle refraction, we consider what happens when a ray passes *through* a partially- or fully-transparent object.

If an object is reflective we simply trace a new **reflected ray** from the point of intersection towards the direction of reflection. The reflected ray is the mirror image of the original ray, pointing away from the surface. If the object is to some extent transparent, then we also trace a **refracted ray** into the surface. If the materials on either side of the surface have different **indices of refraction**, such as air on one side and water on the other, then the refracted ray will be bent, like the image of a straw in a glass of water. If the same medium exists on both sides of the surface then the refracted ray travels in the same direction as the original ray and is not bent.

Figure 5. (a) Reflected ray. (b) Refracted ray

The directions of the reflected and refracted rays are given by the following formulas. For a ray with direction **V**, and a surface with normal **N** (the **normal** is just the direction perpendicular to the surface - pointing directly away from it), the reflected ray direction **Rl** is given by

```
c1 = -dot_product( N, V )
Rl = V + (2 * N * c1 )
```

Note that since **V**, **N**, and **Rl** are vectors with x, y, and z components, the arithmetic on them is performed per-component. The refracted ray direction **Rr** is given by

```
n1 = index of refraction of original medium
n2 = index of refraction of new medium
n = n1 / n2
c2 = sqrt( 1 - $n^2$ * (1 - $c1^2$) )

Rr = (n * V) + (n * c1 - c2) * N
```

Recursive Reflection and Refraction

Figure 6. Recursive reflection and refraction

Figure 6 shows an example of objects that are both reflective and refractive (it does not show the rays from each intersection to the light source, for the sake of clarity). Note how much more complicated the rays become once reflection and refraction are added. Now, each reflected and refracted ray can strike an object which spawns another two rays, each of *these* may spawn another two, and so on. This arrangement of rays spawning subrays spawning subrays leads to the idea of a **tree of rays**. The root node of this tree is the original ray (**a** in the figure) from the eye, and each node in the tree is either a reflected or a refracted ray from the ray above it. Each leaf of the tree is either where the ray hit a non-reflective and non-transparent object, or where the tree depth reached a maximum. As complicated as it seems, there is actually a very simple way to implement this tree of rays: with **recursive function calls**. For each point on the view window we call a function trace_ray(), passing it the ray from the eye through the point. If the first object intersected by the ray is non-reflective and non-transparent, the function simply determines the color at the intersection point and returns this color. If this ray strikes a reflective object, however, the function invokes trace_ray() recursively, but with the reflected ray instead. If the intersected object is transparent, the function also calls trace_ray() with the refracted ray as a parameter. The colors returned by these two function calls are combined with the object color, and the combined color is returned. This can be easily expressed in pseudo-code as:

```
Color trace_ray( Ray original_ray )
{

    Color point_color, reflect_color, refract_color
    Object obj

    obj = get_first_intersection( original_ray )
    point_color = get_point_color( obj )

    if ( object is reflective )

        reflect_color = trace_ray( get_reflected_ray( original_ray,
        obj ) )

    if ( object is refractive )

        refract_color = trace_ray( get_refracted_ray( original_ray,
        obj ) )


    return ( combine_colors( point_color, reflect_color, refract_color ))

}
```

where a **Color** is the triple (**R**, **G**, **B**) for the red, green, and blue components of the color, which can each vary between zero and one. This function, along with some intersection routines, is really all that is needed to write a ray tracer.

Intersections

One of the basic computations needed by the ray tracer is an intersection routine for each type of object in the scene: one for spheres, one for cubes, one for cones, and so forth. If a ray intersects an object, the object's intersection routine returns the distance of the intersection point from the origin of the ray, the normal vector at the point of intersection, and, if texture mapping is being used, a coordinate mapping between the intersection point and the texture image (discussed later in **Texture Mapping**). The distance to the intersection point is needed because if a ray intersects more than one object, we choose the one with the closest intersection point. The normal is used to determine the shade at the point, since it describes in which direction the surface is facing, and therefore affects how much light the point receives (see **Color and Shading**).

Intersecting a Sphere

For each type of object that the ray tracer supports, you need a separate intersection routine. We will limit our ray tracer to handling spheres only, since the intersection routine for a sphere is among the simplest. The following derivation of a sphere intersection is based on [4], page 388.

Figure 7. Intersection of a ray with a sphere.

Fig. 7 shows the geometry of a ray **R** (with origin at **E** and direction **V**) intersecting a sphere with center at **O** and radius **r**. According to the diagram,

$v^2 + b^2 = c^2$ and $d^2 + b^2 = r^2$ (by simple geometry)

and so

$d = \mathbf{sqrt}(r^2 - (c^2 - v^2))$

To determine whether an intersection occurs, we compute the value of **d**. If **d** >= 0, then a valid intersection occurs. If the ray does not intersect, then **d** will be less than zero. After finding the value of **d**, the distance from **E** to the intersection point **P** is (**v** - **d**). The pseudocode for all this is:

```
v = dot_product( EO, V )
disc = $r^2$ - ((dot_product( EO, EO ) - $v^2$ )
if ( disc < 0 )

    no intersection
else

    d = sqrt( disc )
    P = E + (v - d) * V
```

Color and Shading

There are many different ways to determine color at a point of intersection. Some methods are very simple - as in **flat shading**, where every point on the object has the same color. Some techniques - such as the **Cook-Torrance method** [1] - are fairly complex, and take into account many physical attributes of the material in question. We will describe here a simple model known as **Lambertian shading** or **cosine shading**. It determines the brightness of a point based on the normal vector at the point and the vector from the point to the light source. If the two coincide (the surface directly faces the light source) then the point is at full intensity. As the angle between the two vectors increases, as when the surface is tilted away from the light, then the brightness diminishes. This model is known as ``cosine shading'' because that mathematical function easily implements the above effect: it returns the value 1 when given an angle of zero, and returns zero when given a ninety degree angle (when the surface and light source are perpendicular). Thus, to find the brightness of a point, we simply take the cosine of the angle between the two vectors. This value can be quickly computed because it is equal to the dot product of the two unit-length vectors. So by taking the dot product of the surface normal and the unit-length vector towards the light, we get a value between -1 and 1. The values from -1 to 0 indicate that the surface is *facing away* from the light source, so it doesn't receive any light. The value of 0 means the surface is directly perpendicular to the light source (it is at **grazing incidence**), and so again does not receive any light. Values above 0 indicate that the surface does receive some light, with maximum reception when the dot product is 1.

In case the result of the dot product is zero or below zero, we still may not want that part of the object to be pitch-black. After all, even when an object is completely blocked from a light source, there is still light bouncing around that illuminates it to some extent. For this reason we add a little bit of ambient light to every object, which means that the *minimum* amount of light that an object can receive is actually above zero. If we set the **ambient coefficient** to 20%, say, then even in total shadow an object will receive 20% illumination, and will thus be somewhat visible. If 20% illumination is always present, then the remaining 80% is determined by cosine shading. The value 80% in this case is known as the **diffuse coefficient**, which is just (1 - ambient coefficient). The final color computation is then:

```
shade = dot_product( light_vector, normal_vector )
if ( shade < 0 )

    shade = 0

point_color = object_color * ( ambient_coefficient +

    diffuse_coefficient * shade )
```

Texture Mapping

So far we've assumed that every point on a given object has the same basic color, varied only through cosine shading to make it lighter or darker. However, there is another way to color objects, which assigns entire images to them instead of single colors. Known as **texture mapping**, in effect we ``paste'' images to the surfaces of objects. An example of this which certainly everyone will recognize is in the ``Doom''-style games, where walls are drawn using stone or metal patterns instead of single colors. These program start off with small images of those patterns (called **textures**), and then cover the walls with these images. This covering is called a **mapping** because it maps pixels from the image onto the points comprising the walls. Texture mapping is much slower than assigning a single color to every object, since it requires determining and then fetching the appropriate texture pixel corresponding to every point we draw. It has become very common in the last few years, however, thanks to advances in processor speeds and faster graphics algorithms.

While texture mapping produces very nice effects, it is not a difficult feature to implement. All that is needed is a routine which returns the coordinates of a texture pixel given the coordinates of a point on an object. The texture coordinates are refered to as (**u**, **v**), where **u** is the horizontal location of the point in the image, and **v** is the vertical location of the point. The ray tracer then fetches the pixel at (**u**,**v**) from the texture image, and adjusts it as necessary using cosine shading to make it lighter or darker.

Texture Mapping Spheres

The easiest way to texture map onto a sphere is by defining **v** to be the latitude of the point and **u** to be the longitude of the point on the sphere. This way, the texture image will be ``wrapped'' around the sphere, much as a rectangular world map is wrapped around a globe.

Figure 8. Mapping a rectangular world map onto a globe

Figure 9. Mapping a texture image onto a sphere

First we find the two unit-length vectors $\mathbf{V_n}$ and $\mathbf{V_e}$, which point from the center of the sphere towards the ``north pole'' and a point on the equator, respectively. We then find the unit-length vector $\mathbf{V_p}$, from the center of the sphere to the point we're coloring. The latitude is simply the angle between $\mathbf{V_p}$ and $\mathbf{V_n}$. Since we noted above that the dot product of two unit-length vectors is equal to the cosine of the angle between them, we can find the angle itself by

```
phi = arccos( -dot_product( $V_n$, $V_p$ ))
```

and since **v** needs to vary between zero and one, we let

```
v = phi / PI
```

We then find the longitude as

```
theta = ( arccos( dot_product( $V_p$, $V_e$ ) / sin( phi )) ) / ( 2 * PI)
if ( dot_product( cross_product( $V_n$, $V_e$ ), $V_p$ ) > 0 )

    u = theta

else

    u = 1 - theta
```

The last comparison simply checks on what side of the equator vector $\mathbf{V_e}$ the point is (clockwise or counterclockwise to it), and sets **u** accordingly. Now the color of the point is the pixel at (**u** * texture_width,**v** * texture_height) within the texture image. For more info on this, read [3], page 48.

Putting it All Together

First, we need to define the scene we want to render. We create a text input file that describes the objects in the scene by specifying the location, attributes (such as radius for spheres), color, and optional texture map of each object. We then specify where the camera and the view window should be located. For simplicity, we can assume that the camera is always on the Z axis pointing towards the origin, and that the view window is located at the origin. We also need to determine the size of the view window, which is specified by a **field of view** (FOV) parameter. The field of view determines the view window's size, and therefore how much of the scene is visible. Based on these parameters, we can calculate the top-left and bottom-right corners of the window (see online code for example).

Now that we have the top-left and bottom-right points, we divide the window into the resolution required for our output picture, e.g., 640 by 400. We step through each of the 640x400 points on the view window, defining a ray from the eye to that point. For each ray, we calculate its intersection with all the objects in the scene. If it does not intersect any objects, we set the ray - and therefore the corresponding pixel - to the background color. Otherwise, we choose the object with the closest intersection. If the object is reflective, we call the ray-tracing function again recursively, passing it the reflected ray direction. If the object is transparent, we do the same thing with the refracted direction. The calling function then computes the color for the object it is dealing with, and combines this color with the reflected and refracted rays' colors.

When the original ray-tracing call from the eye to the view window returns, the color will be the final color for that pixel - taking into account reflection, refraction, and shadows. This pixel color is then written out to the output file. When all points on the view window have been processed this way, you will have a finished ray-traced picture.

The Code

You will find complete C++ code for a simple raytracer at **http://www.cs.unc.edu/~rademach/xroads-RT.** The **README** file describes how to compile and run the program, and shows where to find the important parts of the code so you can see how the ideas described above are implemented. You will find that the code is very basic, and has lots of room for enhancements and tinkering.

Sample images

Here are some images created with the simple code available online. They demonstrate the techniques described in this article. For some truly astounding images created with full-featured, free ray tracers, check out the links in the **Internet Resources** section below.

Internet Resources

There are many ray tracing resources on the Internet. Here's some of the best:

- There's a free, powerful ray tracing program called *POV-Ray* (for Persistence of Vision Ray Tracer), available for almost every platform at **http://www.povray.org**. It comes in both binary and source form, and their web page has links to other ray tracing information as well as a beautiful gallery of images created with POV-Ray.
- Radiance (**http://radsite.lbl.gov/radiance/HOME.html**) is a ray tracer geared towards analysis of lighting, rather than just making pretty pictures.
- Rayshade (**http://www-graphics.stanford.edu/~cek/rayshade/rayshade.html**) is a fast ray tracer with support for parallel processing.
- The Blue Moon Rendering Tools (**http://www.seas.gwu.edu/student/gritz/bmrt.html**) is a ray tracer that implements the Renderman shading language. This language is used by such companies as Industrial Light & Magic and Pixar to create just about every mind-blowing computer special effect in the last twenty years. The professional software is rather expensive, so this free package by Larry Gritz is an ideal way to play with the Renderman language.
- A good general ray tracing page can be found at **http://www.cm.cf.ac.uk/Ray.Tracing/**, with links to ray tracing information and more ray-tracing software.
- The Ray Tracing News (**http://www.acm.org/tog/resources/RTNews/html/index.html**) is a electronic publication with lots of good info on Ray

Tracing.

References

1

Cook, R.L., and Torrance, K.E. A Reflectance Model for Computer Graphics. *ACM Trans. on Graphics* 1, 1 (Jan 1982)

2

Foley, J. D., and Van Dam, A. ***Computer Graphics: Principles and Practice, Second Edition.*** Addison-Wesley New York, N.Y. 1990.

3

Glassner, A. (ed) ***An Introduction to Ray Tracing.*** Academic Press New York, N.Y. 1989.

4

Glassner, A. (ed) ***Graphics Gems.*** Academic Press New York, N.Y. 1990.

5

Watt, A., and Watt, M. ***Advanced Animation and Rendering Techniques.*** Addison-Wesley New York, N.Y. 1990.

Paul Rademacher (**rademach@cs.unc.edu**) is a doctoral student in Computer Science at the University of North Carolina at Chapel Hill, where he works for the Ultrasound Visualization group. Apart from writing introductory ray tracing articles for Crossroads, his interests include just about all fields of computer science (except, of course, for the boring ones).