An Introduction to Ray Tracing

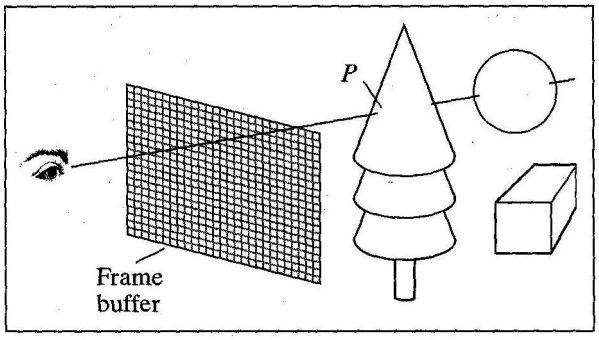
Ryan McFall
Last Revised: December 4, 2008

1. INTRODUCTION

Ray tracing is a computer graphics technique used to generate extremely realistic renderings of images. A ray traced scene consists of several objects located in space, along with a "camera" describing the viewer's location in space and the direction that viewer is looking. The portion of space visible to the viewer is called the ***view plane*.**

Ray tracing works by breaking the view plane up into a rectangular grid of points, called ***pixels***, and then "shooting" rays from the camera through each pixel to determine which object in the scene is first hit by the ray. The color of that pixel is then determined based on the color of the hit object, its location in relation to the light sources in the scene, and properties of the object such as reflectivity and transparency. Once rays through all the pixels in the view plane have been traced, the scene can be drawn.

The figure below, taken from Hill (Hill & Kelley, 2007) shows the basic idea. The figure labels the ***view plane*** as the ***frame buffer*** (which is the term used to refer to the area of memory allocated to hold the information currently shown on the computer's display).



The following tasks make up the process of ray tracing a scene:

1. Determining the properties of a ray through each pixel in the final image. This involves transforming pixel coordinates into the coordinates of the scene, and is a reasonably straight-forward computation.

2. Determining the object in the scene first hit by the ray (if any), and determining properties of the object at the point of intersection. These properties include the normal vector (a vector perpendicular to the surface at the point of intersection) and the color of the object at that point.

3. Determining the amount of light shining upon the point of intersection, and using this information to compute the color of reflected light. This may also involve computations to implement shadowing, reflection, and transparency.

POV-Ray is a powerful application that illustrates the types of images that can be created using a ray tracing approach. It may be helpful for you at this point to play around with POV-Ray to get a feel for the ray tracing process. POV-Ray is freely available at http://www.povray.org/

2. DETERMINING WHEN AN OBJECT IS HIT BY A RAY

To perform step 2 above, we first need to be able to describe a Ray mathematically. Fortunately, this is fairly simple to so. A ray is characterized by both a starting location and a direction. We set the starting location to be the camera location `C`, and construct the direction by computing the vector $P_x - C$, where P_x is the location of a particular pixel in the view plane. The ray is then described parametrically by the equation:

$$R(t) = C + t*(P_x - C)$$

Verify that this ray is indeed at the point `C` when $t=0$ and at the point P_x when $t=1$. For values of $t > 1$, the ray continues on into the scene.

Many simple objects, such as planes, cubes, spheres and cylinders can also be described mathematically. For example, the mathematical definition of a plane is:

$$n \cdot (P - B) = 0$$

where `n` is a vector that is perpendicular to the surface (called a ***normal vector***), `B` is a point on the plane, and `dot` represents the mathematical dot product operator. Any point `P` that satisfies the above equation is on the plane; points that do not satisfy the equation are not on the plane.

As a concrete example, consider the plane of the floor of the room you are in. This is actually a subset of a plane, since planes extend infinitely, but we'll ignore that for simplicity's sake. There are two vectors that are perpendicular to the plane – one points straight up, and one points straight down. The numerical description of these vectors are (0, 1, 0) and (0, -1, 0), with the negative value indicating down while the positive value indicates up.

Just knowing the normal vector doesn't help us identify a particular plane; in the room you are in, both the floor and the ceiling will share the same normal vectors. As soon as you are given a point B on that plane, however, you know whether the plane in question is on the floor or the ceiling (or one of the infinite planes with the normal vectors described above between the floor and the ceiling).

Once we have mathematical representations of the ray and the objects in the scene, a small amount of mathematics can give us the point(s) at which the ray intersects the object. For the plane example given above, we know that the point of intersection must

satisfy the equation `n dot (P-B) = 0`. Every point on the ray is of the form `C+t*(P`$_x$`-C)`. Thus, we can replace `P` in the equation `n dot (P-B) = 0` with `P=C+t*(P`$_x$`-C)`:

`n dot (C+t*(P`$_x$`-C)-B) = 0`

Solving for *t* will find the value of *t* where the intersection occurs; plugging this value of *t* back into the equation of the ray gives us the actual coordinates of intersection.

2.1. Implementing Hit Methods for Generic Objects

The package `mcfall.raytracer.objects` contains classes for several types of generic 3-dimensional shapes. Each of these classes implements the interface `ThreeDimensionalObject` (located in the `mcfall.raytracer` package) which contains the following important methods:

- *`List<HitRecord> hitTime (Ray r)`*
- *`void transform (Matrix m)`*

The *`hitTime`* method gives a list of `HitRecord` objects describing the "times" (values of *t*) where the ray *r* hits that 3-D object. In addition to the time, the normal vector to the surface at the point of intersection is also returned.

The *`transform`* method uses the mathematical matrix multiplication operation to transform the object's shape. Matrices can be used to translate (move), scale (resize) and rotate the object. If a series of the above transformations are applied to an object, it is easy to find the time that a ray hits a transformed version of an object based on the time that a related ray hits a generic version of that object. A sphere of radius 1 centered at the origin is an example of a generic object.

The class `MathematicalObject` implements `ThreeDimensionalObject` and provides the functionality to find the hit time for a transformed object, leaving it up to sub-classes of `MathematicalObject` to compute when a particular ray hits a generic version of that object. `MathematicalObject` leaves this computation up to its sub-classes by declaring an *abstract* method named *`genericHitTime (Ray r)`*.

The responsibility of the generic classes, such as `GenericPlane`, then is to implement the method *`genericHitTime`*. This is generally a straightforward implementation of the mathematical solution derived using the process described in section 2.

3. AN OVERVIEW OF TRACING A SCENE

The `RayTracer` class, located in the package `mcfall.raytracer`, contains the logic that drives the ray tracing process. An instance of a `RayTracer` object is constructed by passing a `Scene` object as a parameter. The `Scene` object contains information about each object in a scene, including what transformations have been applied and properties such as color, how reflective the object is and whether or not it is

transparent. In addition to the objects, the scene also describes the location of the camera viewing the scene, and the size of the view plane that the camera is looking through.

Tracing the scene begins with a call to the *startProduction* method. This method iterates through each pixel in the view plane, determining the ray that starts at the camera and goes through that pixel. The *tracePixel* method is then called for that ray, returning the color of the object, if any, that was hit by the ray.

tracePixel is also simple: it asks the scene to determine which object is first hit by the given Ray, and determines the color of the hit location using the RayTracer method *computeColor* if an object was indeed hit by the ray. *computeColor* uses information about each light source in the scene and the material properties of the object that has been hit. The actual light computations occur in the Light class, and use a simple mathematical model of how light works to compute the color of the pixel. This lighting model is beyond the scope of this project, but references can be found in (Hill & Kelley, 2007).

The Scene class' *firstObjectHitBy* method calls each object's *hitTime* method, as described in section 2.1 on page 3. It stores the hit records returned into an array, so that at the end of the loop there are hit records for each object hit by the ray. It then makes use of the built-in Java sorting algorithm implemented in the static *sort* method of the class Arrays.

4. SCENE FILES

Scenes are described using XML (eXtensible Markup Language) files. For the purposes of this project, all that needs to be known about XML is that it is a text file that uses **markup** to describe the structure of data. The structure of the data is indicated using **elements**, while additional information about a particular element can be included using **attributes** of that element.

Rules about valid elements and their sequencing, as well as the attributes that can be used to annotate the elements, are contained in a file called a **Document Type Declaration (DTD)**. A DTD is provided as part of the project distribution in the file Scene.dtd, located in the root folder of the distribution. However, the easiest way to learn how to create a particular scene file is to use the example files included as part of the distribution.

Bibliography

Hill, F. S.,Jr. and Stephen M. Kelley. *Computer Graphics using OpenGL*. 3rd ed.Prentice Hall, 2007.