Lab 7 - Ray Tracing With Spheres

Paul Cernea

November 22, 2014

In this lab we will be implementing ray tracing. Ray tracing is, in some sense, the complement of z-buffering. In a z-buffer we iterate through the (pixels of the) objects and ask, which pixels will be visible? By contrast, in ray tracing, we iterate through different rays of light and ask, which object will they hit, if any? Specifically, in this lab our objects will be spheres.

The equation of a ray (in three-dimensional space) is

$$(p_x, p_y, p_z) + t(v_x, v_y, v_z) \qquad t \geq 0 \tag{1}$$

The vector $\mathbf{p} = (p_x, p_y, p_z)$ is called the *origin* of the ray, and $\mathbf{v} = (v_x, v_y, v_z)$ is the direction. Note that the equation is the same as that of a line, except that we restrict t to be nonnegative. Thus the ray only goes off in one direction.

The equation of a sphere in three-dimensional space, centered at vector $\mathbf{x}_0 = (x_0, y_0, z_0)$ with radius R, is given by

$$(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 = R^2 \tag{2}$$

Thus vector (x, y, z) is in the sphere if, and only if, it satisfies this equation. Note that we are talking about the exterior shell of the sphere, not the bounded region within.

Combining these two equations, we see that a given ray intersects a given sphere at a given t-value if $t \geq 0$ and t satisfies the quadratic equation

$$at^2 + bt + c = 0 \tag{3}$$

where

$$a = \|\mathbf{v}\|^2 \qquad b = 2\mathbf{v} \cdot (\mathbf{p} - \mathbf{x}_0) \qquad c = \|\mathbf{p} - \mathbf{x}_0\|^2 - R^2 \tag{4}$$

Because the equation is quadratic, this means there are three possible situations. Consider the discriminant $\triangle = b^2 - 4ac$. If $\triangle < 0$, there are no intersections. If $\triangle = 0$, there is one intersection (geometrically, the ray is tangent to the sphere). If $\triangle > 0$, there are two intersections. The formula for the t-values is obtained using the quadratic formula

$$t_{\min} = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \qquad t_{\max} = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \tag{5}$$

where $t_{\min} \leq t_{\max}$. At the end of the day, it must be checked whether these t-values are ≥ 0 or not.

Part 1. Implement the *rayIntersectsSphere* function to determine whether a ray intersects a sphere or not. It should set $t_{\min}$ and $t_{\max}$ to their appropriate values if those exist and are nonnegative. Otherwise, set the t-values to -1.

Now we can implement the actual ray tracing. Given a collection of rays, we can determine what is the first object they intersect, if any? There are various contexts in which this can take place. One of the simplest is as follows: Consider the screen as representing $\{z = 0\}$. From each pixel of the screen, we shoot out a ray orthogonal to the screen in the positive z-direction. We draw that pixel if the ray intersects an object, using the color of the object. Thus we draw a two-dimensional representation of what is behind the screen.

Part 2. Implement the *pixelOn* function to determine whether a pixel will be drawn or not. It should return -1 if the current ray does not intersect any spheres. Otherwise, have it return the index of the first sphere it intersects.

Part 3. Now that everything is set up, create some spheres in the *loadSpheres* function and add them to the *spheres* vector. Run your program and see how the spheres look. Try placing spheres in front of one another and having them intersect each other. What happens as the distance changes between the screen and the spheres?

Further Directions. Try experimenting with different configurations of rays: have all rays emanate from a single source (or converge upon a single sink) and see what happens. Change the colors to be brighter/darker based on the distance from the screen (magnitude of the t-value). If this is done continuously it yields a shading effect. If it is done in several discrete jumps it yields a "toon shading" effect. You can use other criteria besides the distance for shading, like the dot product of the ray's direction with the normal vector emanating from the object.