Lecture 7: Ray-Sphere Intersection, Lights, and Shading

Ray-Sphere Intersection

We've already seen a **parametric** representation of a sphere! Because it's a 2D surface, we needed 2 parameter variables ϕ and θ:

$$x = \cos\theta \sin\phi$$
$$y = \sin\theta$$
$$z = \cos\theta \cos\phi$$

But since our rays are in parametric form, it's going to be easier to intersect a ray with an implicit equation for the sphere. The book has the derivation for general spheres; for intuition building, let's just look at a unit sphere centered at the origin.

Brainstorm: What is true of all points on this sphere?

A: All points are a distance 1 from the origin (generally: a distance R from the center)

The distance to the origin is simply the euclidean norm of the vector:

$$\sqrt{x^2 + y^2 + z^2} = 1$$

Things are cleaner if we square both sides:

$$x^2 + y^2 + z^2 = 1$$

And if we wish to follow tradition and have = 0 on one side, we can write

$$x^2 + y^2 + z^2 - 1 = 0$$

We'd like to substitue our parametric ray equation into this, but our rays are written in vector notation. Let's convert the implicit sphere to vector notation. Let's start by rewriting the sphere as

$$x * x + y * y + z * z - 1 = 0$$

Now if we let $\vec{a} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$, the point on our sphere, then all points $\vec{a}$ on the sphere satisfy:

$$\vec{a} \cdot \vec{a} - 1 = 0$$

Now we can substitute. Points on our ray all have the form

$$r(t) = \vec{p} + t\vec{d}$$

so we'll substitute $r(t)$ for $\vec{a}$ and see for which value(s) of t the ray satisfies the sphere equation.

$$(\vec{p} + t\vec{d}) \cdot (\vec{p} + t\vec{d}) - 1 = 0$$

If we multiply this out (FOIL, except I'm going to write the terms in LOIF order), we get:

$$\vec{d} \cdot \vec{d}t^2 + (2\vec{p} \cdot \vec{d})t + \vec{p} \cdot \vec{p} - 1 = 0$$

Remember that we know the ray, hence $\vec{d}$ and $\vec{p}$ are known, so the only unknown here is t. This is simply an equation of the form

$$At^2 + Bt + C = 0$$

This is a quadratic! We know how to solve those using the quadratic formula:

$$t = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

Plugging in the values of A, B, C from above, we get this monster:

$$t = \frac{-\vec{d} \cdot \vec{p} \pm \sqrt{(\vec{d} \cdot \vec{p})^2 - (\vec{d} \cdot \vec{d})(\vec{p} \cdot \vec{p} - 1)}}{\vec{d} \cdot \vec{d}}$$

Recall that for a perspective camera, the direction vector $\vec{d}$ of our ray goes from the eye to the image plane. We didn't normalize that vector for good reasons, but to simplify our intuition here, let's suppose we had. The length of $\vec{d}$ doesn't change the ray described, it just scales all the values of t. So if $\vec{d}$ is unit length, then the above monster simplifies to

$$t = -\vec{d} \cdot \vec{p} \pm \sqrt{(\vec{d} \cdot \vec{p})^2 - (\vec{p} \cdot \vec{p} - 1)}$$

See the slides for a diagram interpreting this geometrically.

At a geometric level, this makes some intuitive sense because the implicit sphere equation plugged in with coordinates x, y, z represents the point's **signed distance** from the sphere's surface. As the ray approaches the sphere, the distance is positive but shrinking quickly; then it hits the surface, goes negative but slower and slower as it goes towards the center of the chord, then heads back towards positive, crossing the other side of the sphere and heading off into space - exactly the behavior of a parabola. It similarly makes sense that a ray can have either 0, 1, or 2 intersection points with a sphere.

Ray-Scene Intersection

When searching for intersections with objects, we've already talked about wanting to limit the possible values of t: for perspective rays we want to search only values of $t > 1$ to avoid hitting objects that are behind the viewport. It's also handy for our intersection routines to be able to specify a *maximum* value of t, so our function header might look like:

```
ray_intersect(ray, sphere, tmin, tmax)
```

Interesting scenes are likely to have more than one object, and a distant sphere could be occluded by a closer one. So to determine for sure what a ray sees, we'll take a brute force approach: simply intersect the ray with all the objects in the scene, keeping track of the smallest t.

```
find_intersection(ray, scene):
  closest_t = Inf
  closest_surface = nothing
  for object in scene:
    surface, t = ray_intersect(ray, object, 1, closest_t)
    if surface != nothing:
      closest_t = t
      closest_surface = surface
   return closest_t, closest_surface
```

Lights and Shading

Okay so we hit a point on a sphere. We're always interested in the closer point, since that's the one we'd see first along the ray, so we can solve for both but take the smaller value of t. **What now?**

We need to decide what color that point on the sphere appears. The simplest approach is to have a constant color for each object and simply use that color. However, in realistic images, the appearance of a surface depends on both properties of the surface itself as well as properties of the light hitting it.