Introduction to Ray Tracing

Backwards Ray Tracing

Let's begin our exploration in graphics by figuring out how to create an image of an object. For the sake of simplicity, we will let the object be a unit sphere centred at the origin.

The first thing we need to do is define our image. An image is represented in the computer as an array of *pixels*, which we can imagine as cells in a grid. Specifically, each cell is represented by a triplet of values, representing one of 3 channels: red, green, and blue. These three *primary colours* are then *added* together to create the final colour that we can see. Note that this addition is not performed by us. Instead, whenever the image is displayed, different intensities of RGB lights are presented. When the light colours are combined together, they form the correct colour.

With this in mind, the task of representing an image of a sphere is simple: for every pixel in our image, we need to determine whether the sphere "covers" it. If it does, then the we shade the pixel with the colour of the sphere. Otherwise, we leave it blank.

This is fine, but the question now becomes: how exactly do we determine whether a pixel is "covered" by the sphere? To answer this question, we need to look at the way our eyes see things in the real world. Our eyes see things because light is emitted (be it by a bulb, the sun, or a screen, for example) and that light then bounces off an object and into our eyes. Our eyes have special cells that are then stimulated by this incoming light. This stimuli is then transformed into electrical signals which our brain then interprets as an image.

For the sake of simplicity we are going to ignore the fact that light is both a wave and a particle, and instead we will model light as a *ray.* We will define what a ray is later, but intuitively we can think about it as a line that shoots from the light source in a specific direction. So, to render our scene, we would shoot rays from a light source. Only those rays that bounce off our sphere and hit the image (which is the stand-in for our eye) will provide a colour. This technique is known as *forwards ray-tracing.*

The problem here is that light sources emit billions and billions of rays. Even more important is the fact that a fraction of those rays will impact the sphere, and of those, only a fraction will actually impact the image. Computationally speaking, it means that we would be wasting a lot of time on computing rays that we will never see.

Since this seems impractical, we will try a different approach. Instead of shooting rays from the light source, we will shoot rays from the centre of each pixel. This way, we know that if the rays hit the sphere, they are guaranteed to influence the colour of the pixel. If the ray misses the sphere, then the pixel is set to black. This is known as *backwards ray-tracing.* For the time being, we will ignore the existence of the light. It will be introduced later on when we discuss shading.

What is a Ray?

Before we begin implementing anything, we need to define what a ray actually is. Formally, a ray is a function $r : \mathbb{R} \to \mathbb{R}^3$ that is defined by a point and a vector. In symbols, a ray is defined as $r(t) = \mathbf{o} + t\mathbf{d}$, where $\mathbf{o}$ is called the *origin* of the ray, and $\mathbf{d}$ is the *direction* of the ray. Intuitively, the parameter t is simply a way of walking along $\mathbf{d}$ starting from $\mathbf{o}$. Note that the output of the equation of the ray is itself a *point*.

Ray-Sphere Intersection

Now that we have our ray, let's figure out how to render our sphere. First, we will be shooting one ray per pixel, and we will define the origin of each ray to be the centre of said pixel. The direction will be shot perpendicular to the "image plane". Let's assume that our image is centred at $(0, 0, 100)$. We can then simply define the direction of our rays as $(0, 0, -1)$ so that they shoot "forwards" into the scene.

We now have our ray, so the next step is to figure out how to represent our sphere. There are multiple ways of accomplishing this, but for now we will be using a very simple representation called *implicit*. A sphere can be defined as a function $f : \mathbb{R}^3 \to \mathbb{R}$ defined as $f(x, y, z) = (x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 - R^2$ where $R \in \mathbb{R}$ is the radius of the sphere and (x_0, y_0, z_0) is the centre of the sphere. The question now is this: how do we now if our ray intersects the sphere?

To answer that question, note that we can re-write the function the sphere as $(p - c) \cdot (p - c) - R^2$, where $p = (x, y, z)$ and $c = (x_0, y_0, z_0)$. Note that these are both points, but by subtracting them we get a vector. With this formulation, we can substitute the equation of the ray for p and solve for t.

From the resulting equation, we know that t can have 0, 1, or 2 roots. These cases are uniquely determined by the sign of the determinant:

- $d < 0$: there are no roots, and therefore the ray misses the sphere.
- $d = 0$: there is exactly one root, and therefore the ray hits the sphere only once (called a *tangent ray*).
- $d > 0$: there are exactly two roots, and therefore the ray penetrates the sphere.

We can perform a minor optimization here by delaying the evaluation of the square root until we actually know that the ray intersects the sphere. We can now fill in these equations into our program and produce a red sphere.