

# CS 314: Ray Casting

Robert Bridson

October 7, 2008

## 1 Ray Casting

### 1.1 Loop Order

The last hidden surface elimination algorithm we looked at was **ray casting**. One way to interpret this is as a variant of Z-buffer rendering, with the ordering of loops swapped. Z-buffer looks like:

- For every object:
  - For every pixel:
    - Check if the pixel is rasterized in the object, and update pixel if the object is nearest so far here

Ray casting instead looks like:

- For every pixel (i.e. ray):
  - For every object:
    - Check if the ray intersects the object, and update pixel if the object is nearest so far here

(Note that in Z-buffer, right away we improved the “for every pixel” loop to just touch the relevant pixels for each object via bounding boxes—we’ll soon get to a similar optimization for ray casting, looping over just the relevant objects for each ray.)

### 1.2 Simulating Light

This is a useful analogy, but we can also see ray casting as a step in an entirely different direction: simulating the physics of light transport. We alluded to this before, when discussing perspective projection, but the model we presented (painting on a sheet of glass at the near clipping plane) was still a little removed

from real physics. Ultimately it's the model we'll stick with in this course, but it's instructive to take it a step further and see how rendering really can connect to actual physics.

There are several ways of thinking about light—photons (particles), electromagnetic waves, or a more abstract continuous energy field. To keep things as concrete and as easy to think about as possible, we'll stick with the photon idea.<sup>1</sup> In most scenarios photons travel in straight lines between different points in the world—the chief exception being when continuous refraction occurs as in a mirage. Moreover, they travel at nearly the speed of light (slowed down a bit from the full speed of light in a vacuum by the material through which they travel) which is far, far faster than any object in a typical scene; for all intents and purposes, they can be thought of as zapping instantaneously from one spot to another along a straight line. This means light simulation can be set up in terms of straight lines between different points in a scene, rather than explicitly tracking the motion of photons.

If we look at a camera, ultimately the image is formed because photons striking the film (or digital sensor array) leave a record. Where do those photons come from? If we just leave the film out in the open, the answer is from all directions, and the “image” that's formed is just a big messy blur of all incoming light. Cameras of course don't leave film out in the open, but instead block out almost all of these directions to avoid this blur.

The simplest camera design of all is the **pinhole camera**. You may have made one at some point to view a solar eclipse. It consists simply of a box (opaque enough to block out light) with film inside on one wall, and a tiny pin-pricked hole on the opposite side through which light can enter. If you look at any point on the film—lets call it a pixel centre for now—the only photons that hit it have to come through the pinhole, which means only the photons coming along the straight line connecting the pixel centre and the pinhole have an effect. To simulate what colour that pixel should be physically, we can simply trace back along that line, back out through the pinhole, until we find the surface from which the photons are coming—and figure out their intensity and colour from there.

Since we're only tracing along one direction on this line (i.e. ignoring the part of the line that continues on the wrong side of the film) we can call this a **ray**, a half-infinite line. The **origin** of the ray is the point on the film, and its **direction** points from the origin towards the pinhole.

One little inconvenience with the pinhole camera is that the images formed are flipped upside-down; mostly for this reason we generally stick with the earlier sheet-of-glass model in graphics, which is essentially equivalent mathematically. In real life, pinhole cameras aren't particularly popular either, because although they can produce extremely sharp images of still scenes, they block out so much light

---

<sup>1</sup>The rainbow colours of a CD and related “diffraction” phenomena, along with polarization, are examples where the wave model really is critical, but in almost all common computer graphics scenarios the photon model of light is perfectly reasonable.

that it takes a long time to expose the film—moving subjects are impossible to capture. Typical cameras instead use a much larger hole called an **aperture** to let more light in, and add one or more **lenses** to focus at least part of the image. Simulating this lens focusing is actually crucial for photorealistic rendering—for example in film visual effects where computer generated imagery is supposed to match what real film cameras record—but a little more advanced than we need to get into right now.

### 1.3 Computing Rays

One of the core operations in ray casting is determining exactly which rays to cast. A ray is usually characterized by a point of **origin** (a point in 3D) and a **direction** (a 3D vector).

There are several reasonable options for producing rays. For now we will want them to correspond to pixel centres, like our earlier rasterization, but it should be pointed out that to do a better job of simulating the light in a camera we might use more. In a digital camera, say, each pixel really measures the amount of light received over a small area, not just at a single point. Therefore our rendering should try to estimate the total amount of light hitting a pixel—averaging over all rays that originate in the pixel—and can get a better estimate by using several ray origin locations per pixel instead of just the centre. (This is technically known as a **supersampling** approach to **antialiasing** if you’re curious to look up more information on it.)

The different options relate to different coordinate systems. To keep the most similarity with the Z-buffer approach we saw before (and preserve the same matrix tools we’ve used to describe perspective projection etc.) we could generate rays in normalized device coordinates: for pixel centre  $(i, j)$  in an  $m \times n$  image, the ray has origin something like

$$\vec{o}_{\text{ndc}} = \left( \frac{2(i + 1/2)}{m} - 1, \frac{2(j + 1/2)}{n} - 1, -1 \right)$$

and has direction

$$\vec{d}_{\text{ndc}} = (0, 0, 1)$$

which is particularly simple. This corresponds to rays beginning on the near clipping plane ( $z = -1$  in n.d.c.) and going parallel to the n.d.c.  $z$ -axis.

In camera space, with a near clip region at camera  $z = -n$  stretching from camera  $x = l$  to  $x = r$  and  $y = b$  to  $y = t$ , the ray origin can be computed as:

$$\vec{o}_{\text{cam}} = \left( l + \frac{i + 1/2}{m}(r - l), b + \frac{j + 1/2}{n}(t - b), -n \right)$$

and the direction, which should be lined up with the camera space origin (though pointed away from it) and is conventionally normalized to be unit length, is:

$$\vec{d}_{\text{cam}} = \frac{\vec{o}}{\|\vec{o}\|}$$

(Obviously, this assumes perspective projection: for the somewhat less common case of orthographic projection, you should be able to instantly see what the direction in camera space must be.) Sometimes ray casting code will forget about specifying a near clipping plane, and replace the ray origin above with  $(0, 0, 0)$  but keep the direction the same; it's not a big deal either way.

Finally, you could also set up rays in world space coordinates, to match the geometry the rays will intersect against, but this is somewhat more convoluted and ultimately is based on math using camera space coordinates converted to world space. It's simpler to instead generate the rays in either n.d.c. or camera space and then later transform them into world space rays.

## 1.4 Transforming Directions

Transforming the origin of a ray from one coordinate system to another is simple: it's just another point, that can be multiplied by the appropriate  $4 \times 4$  matrix (or its inverse) using homogeneous coordinates. However, the ray direction is a little bit trickier: the direction vector can't be transformed the same way as if it represented a point.

For example, consider a direction  $(1, 0, 0)$  pointing from ray origin  $(5, 1, 1)$  towards the point  $(6, 1, 1)$ . Under a translation by vector  $(2, 3, 4)$ , these two points become  $(7, 4, 5)$  and  $(8, 4, 5)$  and the direction is still  $(1, 0, 0)$ : the direction has *not* been transformed to  $(3, 3, 4)$  adding  $(2, 3, 4)$  like we did with the points. In general, translation (where all points are moved parallel the same amount) should leave directions unchanged. On the other hand, it's not hard to see that rotation *should* rotate directions just like points are rotated.

The key to making sense of it is to think of a direction as the difference between two points: to transform the direction, first transform the points and then take the difference again. For converting from n.d.c. rays to camera space or world space rays, where a perspective transform is involved, this is the best way of doing it in fact.

However if we rule out perspective projection, i.e. only look at  $4 \times 4$  matrices with bottom row equal to  $(0, 0, 0, 1)$ , for example converting from camera space to world space, then we can make this a little more elegant. Here it turns out not to matter which two points we pick to describe the direction: the simplest choice is the origin  $\vec{o}$  and the point with the same coordinates as  $\vec{d}$  which we'll write as  $\vec{o} + \vec{d}$

to try to make it clearer that the direction  $\vec{d}$  is *not* a point. If the transformation matrix is  $M$ , then the transformed direction is:

$$M(\vec{0} + \vec{d}) - M\vec{0}$$

Writing this out in homogeneous coordinates, we get

$$M \begin{pmatrix} 0 + d_1 \\ 0 + d_2 \\ 0 + d_3 \\ 1 \end{pmatrix} - M \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

Note that the origin  $\vec{0}$  has homogeneous coordinate equal to 1, just like other points: it's not the vector of all zeroes when written out in homogeneous coordinates. Using the fact that matrix multiplication is linear, we can simplify this to:

$$M \left[ \begin{pmatrix} 0 + d_1 \\ 0 + d_2 \\ 0 + d_3 \\ 1 \end{pmatrix} - \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \right] = M \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ 0 \end{pmatrix}$$

Aha! Up pops, very naturally, a vector with **zero** for its homogeneous coordinate. Before we ruled this out, saying every point in space is represented with a nonzero homogeneous coordinate: **for directions, we instead take the homogeneous coordinate to be zero**. You can double check that multiplication by any of the transformation matrices we've seen so far (rotations, translations, etc.) apart from perspective projection does the right thing for directions represented this way.

Finally, one last word on the length of a direction vector. It's often convenient to keep the direction of a ray unit length, normalizing it if it's not; transforming a direction by a scaling or shearing matrix can change its length however, so you may need to renormalize to keep this convention after transformation.

## 2 Intersecting Rays and Planes

Now that we can construct rays to cast, the next order of business is writing code which can compute if a ray intersects a given surface. The simplest case of all is when that surface is an infinite plane.

An infinite plane can be described very simply with two vectors: a point  $\vec{p}$  contained in the plane, and a normal vector  $\hat{n}$  orthogonal to the plane (typically with unit length, but this isn't crucial to anything we do in this section). Any other point  $x$  is in the plane if and only if the direction from  $\vec{p}$  to  $\vec{x}$  is orthogonal

to  $\hat{n}$ : this leads to an implicit description of the plane as the points satisfying

$$(\vec{x} - \vec{p}) \cdot \hat{n} = 0$$

Meanwhile, a ray can be described parametrically as

$$\vec{x}(t) = \vec{o} + t\vec{d}$$

where the scalar parameter  $t \geq 0$  produces all possible points on the ray. (If we included negative  $t$ , we would have an infinite line; restricting  $t \geq 0$  gives us just the half-infinite ray.) If the direction vector is unit-length, then  $t$  is actually distance along the ray from the origin.

The question of whether the ray intersects the plane can be rephrased as: is there a point on the ray that is also in the plane? Using the above two descriptions, this boils down to finding if there is a parameter value  $t \geq 0$  where

$$\begin{aligned} (\vec{x}(t) - \vec{p}) \cdot \hat{n} &= 0 \\ \Leftrightarrow (\vec{o} + t\vec{d} - \vec{p}) \cdot \hat{n} &= 0 \end{aligned}$$

We can rearrange this to be just a linear equation in  $t$ :

$$(\vec{d} \cdot \hat{n}) t = -(\vec{o} - \vec{p}) \cdot \hat{n}$$

Does this have a solution with  $t \geq 0$ ?

We'll first eliminate the tricky case where the coefficient of  $t$  is exactly zero—which geometrically corresponds to a ray parallel to the plane—since there might be zero or infinitely many solutions in this case, which is a bit inconvenient later. So if  $\vec{d} \cdot \hat{n} = 0$ , we'll declare there isn't a simple intersection and leave it at that. Otherwise, there is a solution

$$t = \frac{-(\vec{o} - \vec{p}) \cdot \hat{n}}{\vec{d} \cdot \hat{n}}$$

and we get an intersection if and only if this is non-negative.

In fact, solving it this way tells us also (from the value of  $t$ ) how far along the ray the intersection occurred. If ray casting finds multiple surfaces which intersect a ray, it needs to identify the nearest or the "first" along the ray, and it can do this by selecting the one with a minimum  $t$  value. If  $\vec{d}$  is unit length, this is in fact the distance between the viewer and the intersection; otherwise it's the distance divided by length of  $\vec{d}$ .