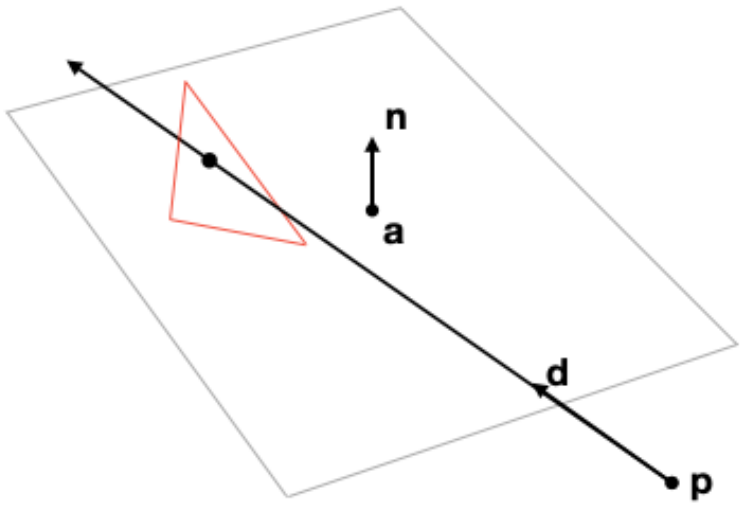
Lecture 10: Barycentric Coordinates and Ray-Triangle Intersection

We'd like to intersect rays with triangles.



One way to do this is to intersect with the plane, then determine whether your position on the plane is within the triangle. We're going to do it this way, but the math will end up solving both in one go.

Ray-Plane Intersection

To intersect a ray with a plane, we need an equation for the plane; ideally it's an implicit equation. Suppose we have a plane that goes through a point **a** and has normal **n**. (see whiteboard notes) for any 3D point **p**, the point lies on the plane if and only if (**p** - **a**) dot **n** equals zero. In fact, (**p** - **a**) dot **n** gives the signed distance between **p** and the plane, with positive distances on the same side as the normal and negative on the opposite side.

If we have a triangle **a, b, c**and we want this representation, we just need to calculate the normal. We've done this before: we used **n** = (**b-a**) x (**c** - **a**). For the point on the plane, we'll just use **a** (we could use any point).

To intersect a ray with a plane, we can plug a point on our parametric ray into the implicit equation for the plane and solve:

$$(\mathbf{p} + t\mathbf{d} - \mathbf{a}) \cdot ((\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a}) = 0 \tag{1}$$

The only unknown here is t; that can be solved for, then plugged back into the ray equation to get the 3D coordinates of the intersection. Now all we need is to determine whether this point is inside the triangle. Our point-in-triangle test from HW0/A0 is a good place to start thinking about this, but the fact that we're in 3D complicates things a little. If we're going to use a technique like that, we need to switch to some kind of 2D representation that's constrained to the plane that the triangle lies in. This brings to mind a general approach involving finding a 2D *basis* that spans the plane.

In fact, that's exactly what we're going to do: the approach here is called **barycentric coordinates**. This is probably going to seem weird, unnatural, and maybe unnecessary. But if we follow this through, we'll end up with a single calculation that gives us **all** of the following three things:

- Where does the ray intersect the triangle's plane?
- Does it intersect inside the triangle?
- What is the weight assigned to each corner of the triangle when interpolating vertex data?

This final one is the kicker: remember that once we hit a point, we need to know surface properties like color, texture coordinates, normals, etc. On triangle meshes these are usually interpolated from vertex data stored at the triangle's corners. Expressing the position of the intersection on the plane in barycentric coordinates makes for a very elegant answer to "how much do I weight the value from each vertex?"

See whiteboard notes for the presentation of barycentric coordinates. Also see section 2.7 of the book.

Super Cool Properties of Barycentric Coordinates

- $\alpha + \beta + \gamma = 1$, no matter where you are in the plane.
- Each coordinate is the *scaled signed distance* to one of the triangle's edges. That is, the coordinate is 0 at the edge and 1 at the vertex opposite the edge. In particular:
  - α is the fraction of the distance from edge **bc** to vertex **a**
  - β is the fraction of the distance from edge **ac** to vertex **b**
  - γ is the fraction of the distance from edge **ab** to vertex **c**
- (α, β, γ) is inside the triangle iff:

$$\begin{aligned} 0 < \alpha < 1 \\ 0 < \beta < 1 \\ 0 < \gamma < 1 \end{aligned} \tag{2}$$

- The coordinates are proportional to the areas of the subtriangles made by **p** inside the triangle, as a fraction of the full triangle's area, are proportional to the coordinates - see the picture on the slides.

These coordinates, if we find them, solve ray-triangle intersection using the "inside the triangle" property above.

Moreover, they also solve the vertex-data interpolation problem! The fact that they sum to 1 and are proportional to subtriangle areas also makes them perfect for smoothly interpolating vertex ata. If we have some values (e.g., texture coordinates, color values, ...), call them v_a, v_b, v_c, stored at each vertex, we can weight them by their barycentric coordinates to interpolate smoothly anywhere in the triangle:

$$v = \alpha v_a + \beta v_b + \gamma v_c \tag{3}$$

Barycentric Ray-Triangle Intersection

A point in the triangle's plane is $\mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a})$, and a point on the ray is $\mathbf{p} + t\mathbf{d}$, so we can set these equal to find the intersection point:

$$\mathbf{p} + t\mathbf{d} = \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a}) \tag{4}$$

We can rearrange this to:

$$(\mathbf{a} - \mathbf{b})\beta + (\mathbf{a} - \mathbf{c})\gamma + (\mathbf{d})t = \mathbf{a} - \mathbf{p} \tag{6}$$

As a reminder, the bold letters are 3-vectors, while β, γ, and t are scalars. With this in mind, we can rewrite this as a 3D square matrix system:

$$\begin{bmatrix} (\mathbf{a} - \mathbf{b}) & (\mathbf{a} - \mathbf{c}) & \mathbf{d} \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} \mathbf{a} - \mathbf{p} \end{bmatrix} \tag{7}$$

Now, the 3x3 matrix is made up of knowns, the 3-vector of β, γ, t contains all the unknowns, and the right-hand side is a 3-vector of known quantities: this has the form $A\mathbf{x} = \mathbf{b}$, which we can solve using standard linear algebra techniques.

Solving Linear Systems on Computers

In a linear algebra class, the solution to the above would generally be given as $\mathbf{x} = A^{-1}\mathbf{b}$. On computers, inverting matrices is a dicey thing to do because of floating-point precision and stability issues. Sometimes you can ignore this and come out just fine, but a better approach to solving linear systems on computers is to use standard numerical techniques that don't explicitly compute the inverse. These techniques are neat, and you should take a numerical linear algebra class and learn all about them, but for our purposes, we can take one of two approaches. The first is to use Julia's built-in matrix solving operator:

```
A = # set up the matrix A with columns a - b, a - c, and d
b = # set up the right-hand side 3-vector a-p
x = A \ b # this solves the system for you!
```

This may or may not do something smart/fast that's specific to 3x3 matrices. Another approach that is about as fast as it gets and does not leave efficiency up to the Julia compiler involves using Cramer's rule to solve the system; this only works on 3x3 matrices (it generalizes, but gets nasty fast) but may end up using fewer flops than the backslash operator.

You can find a very nice exposition on Cramer's rule and how to implement it efficiently in this setting in section 4.4.2 (Ray-Triangle Intersection) of the textbook.