CS 314: Ray-Triangle Intersection

Robert Bridson

November 7, 2009

Calculating whether or not (and where) a ray intersects a triangle is a fundamental operation we need for raycasting, and it has to be done both well and fast. There are a couple of approaches possible; I'll briefly outline a classic two step approach often taught in graphics, but this has been superseded by a better approach more in line with our 2D point-in-triangle rasterization test, and it this modern approach on which I will concentrate.

Unlike planes and spheres, triangles do not have a simple implicit representation of the form $\{\vec{x} : F(\vec{x}) = 0\}$ that we can plug the parametric ray equation into to solve for intersections. We will get around this by breaking up the problem into smaller problems—essentially viewing a triangle as constructed from a plane and bounding edges on that plane, and then intersecting the ray with the plane and the spaced defined by those edges.

1 A Two Step Method

The old approach operates in two stages. First we find an implicit description of the plane containing the triangle. If you recall, all we need for that is a point on the plane—any of the triangle's vertices will do—and a vector orthogonal to that plane—a cross-product of two of the sides of the triangle is fine for that. With that description, we can intersect the ray with the plane as before, and if there is an intersection we get the ray parameter value s and therefore the location of the intersection $\vec{x} = \vec{x}_0 + s\vec{d}$ in space. This is the first stage, which either rules out intersection or provides a possible intersection point in 3D which is on the plane of the triangle but not necessarily inside the triangle itself. The second stage checks this 3D point to see if it's inside the triangle: this is almost like our old 2D point-in-triangle rasterization test, but the extra 3rd dimension makes it much harder. To get around that, we just throw away one of the coordinates to get to a 2D point-in-triangle problem where we can use our old test—with the only tricky part being the choice of coordinate to throw away.

The old approach is nice and straightforward, building on what we've already done, but it suffers from two weaknesses:

- it's slower than the modern test,
- and it's more vulnerable to floating-point rounding errors.

Both of these stem mainly from the need to compute an intermediate intersection point with the plane: that requires a relatively expensive division etc., and produces a possible intersection point that is perturbed by rounding error. The second stage 2D point-in-triangle test then has to take into account an unknown rounding error in the point, and to avoid cracks this means artificially declaring intersection for points that are "close enough" to the triangle. Robustly avoiding overdraw is hopeless, and the programmer is on the hook for finding a good definition of "close enough", a margin of error that will produce half-decent results.

2 Orientation in 3D

Instead, let's look at a more direct generalization of our 2D point-in-triangle test. If you recall, determining whether the point is to the left or right of each edge was the crucial operation. That orientation test boiled down to evaluating the sign of a determinant, interpreted in terms of signed area, implicit line descriptions, or cross-products. As an extra bonus, by dividing the "edge functions" by their sum we got the barycentric coordinates of the point with respect to the triangle, which let us do linear interpolation of colour and depth from the vertices.

Our strategy for the ray-triangle test will be to similarly decompose the problem into primitive orientation tests. But first, let's make the input to the problem more uniform—we'll be able to undo this later, but right now it's inconvenient that the ray is represented by a point (the origin) and a direction vector. In fact, raycasting software often includes an end to each ray, a maximum parameter value $s_{\max}$ or equivalently an endpoint $\vec{x}_1 = \vec{x}_0 + s_{\max}\vec{d}$. This is similar in purpose to the far clipping plane in rasterization, providing efficiency in some situations. Our problem then reduces to checking if the segment between $\vec{x}_0$ and $\vec{x}_1$ (the ray's start and end points respectively) intersects a triangle with vertices $\vec{x}_2$, $\vec{x}_3$ and $\vec{x}_4$.

The key tool is defining the *orientation* of four points in 3D, say $\vec{x}_1$, $\vec{x}_2$, $\vec{x}_3$ and $\vec{x}_4$. This can be developed in many ways, and we'll explore some of the different interpretations in a moment; we'll start by working out when the four points are coplanar, i.e. when $\vec{x}_1$ is in the plane containing $\vec{x}_2$, $\vec{x}_3$ and $\vec{x}_4$.

(Compare this to rasterization in 2D, working out when a point is collinear with two other points, i.e. in the line containing the two others.) From a linear algebra perspective, this is exactly when the three vectors $\vec{x}_1 - \vec{x}_4$, $\vec{x}_2 - \vec{x}_4$ and $\vec{x}_3 - \vec{x}_4$ are linearly dependent (one is a linear combination of the others). We can express linear dependence as meaning when the determinant of the matrix with those columns is zero:

$$\vec{x}_1, \vec{x}_2, \vec{x}_3 \text{ and } \vec{x}_4 \text{ are coplanar} \quad \Leftrightarrow \quad \det\begin{pmatrix} x_1 - x_4 & x_2 - x_4 & x_3 - x_4 \\ y_1 - y_4 & y_2 - y_4 & y_3 - y_4 \\ z_1 - z_4 & z_2 - z_4 & z_3 - z_4 \end{pmatrix} = 0$$

I prefer to make this expression more uniform, not singling out $\vec{x}_4$ for special treatment. Using elementary column operations and cofactor expansion (remember your linear algebra rules for determinants!) it's not hard to see this is equal to:

$$\det\begin{pmatrix} x_1 & x_2 & x_3 & x_4 \\ y_1 & y_2 & y_3 & y_4 \\ z_1 & z_2 & z_3 & z_4 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

I'll call this the *orient* function for convenience:

$$\text{orient}(\vec{x}_1, \vec{x}_2, \vec{x}_3, \vec{x}_4) = \det\begin{pmatrix} \vec{x}_1 & \vec{x}_2 & \vec{x}_3 & \vec{x}_4 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

This is exactly analogous to the edge functions we saw for 2D rasterization.

When the points are not coplanar, the *orient* function is nonzero and its sign gives us information on their orientation. With $\vec{x}_2$, $\vec{x}_3$ and $\vec{x}_4$ held fixed, it's clear it is an affine function of $\vec{x}_1$ which is zero only on the plane containing $\vec{x}_2$, $\vec{x}_3$ and $\vec{x}_4$: it therefore must be positive on one side of the plane and negative on the other side.

Expanding out both expressions, it's also not hard to see that

$$\text{orient}(\vec{x}_1, \vec{x}_2, \vec{x}_3, \vec{x}_4) = [(\vec{x}_2 - \vec{x}_4) \wedge (\vec{x}_3 - \vec{x}_4)] \cdot (\vec{x}_1 - \vec{x}_4)$$

This makes for one easy interpretation of the sign of the orientation. The cross-product $(\vec{x}_2 - \vec{x}_4) \wedge (\vec{x}_3 - \vec{x}_4)$ is a vector orthogonal to the plane containing $\vec{x}_2$, $\vec{x}_3$ and $\vec{x}_4$ following the right-hand rule: put your right hand at $\vec{x}_4$ and curl your fingers from $\vec{x}_2$ towards $\vec{x}_3$—your thumb points in the direction of this cross-product. The dot-product of this vector with $\vec{x}_1 - \vec{x}_4$ is positive if $\vec{x}_1$ is on the right-handed side of the plane, and negative otherwise.

This expression is also known as the **triple product** of the three vectors:

$$[\vec{x}_1 - \vec{x}_4,\ \vec{x}_2 - \vec{x}_4,\ \vec{x}_3 - \vec{x}_4]$$

You can search for this name to find more properties and connections.

Orientation also can be interpreted as determining whether the directed line from $\vec{x}_1$ to $\vec{x}_2$ goes past the directed line from $\vec{x}_3$ to $\vec{x}_4$ on the left or right. If it's on the right, the line from $\vec{x}_1$ to $\vec{x}_2$ will hit the plane through $\vec{x}_2$, $\vec{x}_3$ and $\vec{x}_4$ along the right-hand-thumb direction, i.e. the orientation will be positive, and if it's on the left it will be negative.

Finally, the orientation function happens to be six times the signed volume of the tetrahedron with corners $\vec{x}_1$, $\vec{x}_2$, $\vec{x}_3$ and $\vec{x}_4$. This also makes it the natural generalization of the signed area we worked with in 2D.

3 Determining Intersection with Orientations

Our problem once again is finding if the segment $\vec{x}_0 \leftrightarrow \vec{x}_1$ intersects the triangle with corners $\vec{x}_2$, $\vec{x}_3$ and $\vec{x}_4$. This can be broken down into five orientation tests.

First, there can be an intersection only when the end points of the segment are on *opposite* sides of the plane containing the triangle:

`if` $\operatorname{sign}(\operatorname{orient}(\vec{x}_0, \vec{x}_2, \vec{x}_3, \vec{x}_4)) = \operatorname{sign}(\operatorname{orient}(\vec{x}_1, \vec{x}_2, \vec{x}_3, \vec{x}_4))$ `then no intersection.`

If there is a sign difference, we can proceed to further tests. Note that in the case where one or both orientations evaluate to zero, i.e. the start or end of the segment lies exactly on the plane of the triangle, we might as well flag that as *not* being an intersection—for the purposes of raycasting.

If that first test has passed, we still need to know if the segment goes through the triangle or off to one side of it. In fact, the sign of $\operatorname{orient}(\vec{x}_0, \vec{x}_2, \vec{x}_3, \vec{x}_4)$, which we already evaluated, tells us whether the line from $\vec{x}_0$ to $\vec{x}_2$ goes to the right or left of the other edge of the triangle $\vec{x}_3 \rightarrow \vec{x}_4$. If the segment hits the triangle, it must go on the same side of this edge:

`if` $\operatorname{sign}(\operatorname{orient}(\vec{x}_0, \vec{x}_1, \vec{x}_3, \vec{x}_4)) \neq \operatorname{sign}(\operatorname{orient}(\vec{x}_0, \vec{x}_2, \vec{x}_3, \vec{x}_4))$ `then no intersection.`

Similarly we need this sign to be the same for the other two edges for there to be an intersection, exactly as we worked out in the 2D point-in-triangle rasterization test:

`if` $\operatorname{sign}(\operatorname{orient}(\vec{x}_0, \vec{x}_1, \vec{x}_4, \vec{x}_2)) \neq \operatorname{sign}(\operatorname{orient}(\vec{x}_0, \vec{x}_3, \vec{x}_4, \vec{x}_2))$ `then no intersection.`

`if` $\operatorname{sign}(\operatorname{orient}(\vec{x}_0, \vec{x}_1, \vec{x}_2, \vec{x}_3)) \neq \operatorname{sign}(\operatorname{orient}(\vec{x}_0, \vec{x}_4, \vec{x}_2, \vec{x}_3))$ `then no intersection.`

Note that the second orientation expression used in both of these cases is equivalent to what we already evaluated, based on simple determinant rules:

$$\text{orient}(\vec{x}_0, \vec{x}_2, \vec{x}_3, \vec{x}_4) = \text{orient}(\vec{x}_0, \vec{x}_3, \vec{x}_4, \vec{x}_2) = \text{orient}(\vec{x}_0, \vec{x}_4, \vec{x}_2, \vec{x}_3)$$

If all of these sign tests pass, we do have an intersection. For these three tests against the sides of the triangle, we may as well include an exact zero (where the segment exactly hits the edge) as an intersection to avoid cracks.

Summarizing, here's pseudocode for the intersection test:

```
intersect(x0, x1, x2, x3, x4):
  f0234 = orient(x0, x2, x3, x4)
  f1234 = orient(x1, x2, x3, x4)
  if sign(f0234) = sign(f1234): return false
  f0134 = orient(x0, x1, x3, x4)
  if sign(f0134) != sign(f0234): return false
  f0142 = orient(x0, x1, x4, x2)
  if sign(f0142) != sign(f0234): return false
  f0123 = orient(x0, x1, x2, x3)
  if sign(f0123) != sign(f0234): return false
  return true
```

4 Rounding Error Issues

This method has some nice properties when it comes to rounding error: only addition, subtraction and multiplication are used, and ultimately getting the right answer depends only on making sure the signs of the orientation values (not their magnitude!) are correct. Sophisticated methods from computational geometry can be used to ensure this fairly efficiently. In a nutshell, the regular floating point expressions can be used, and only if an estimate of the error threatens the correctness of the sign is a more expensive exact evaluation used—which only happens when the orientation is very close to zero.

However, even with naive code, problems with raycasting—notably cracks—can be avoided fairly easily, by making sure the orientations are evaluated *consistently*. In particular, if two triangles share a common edge, one can arrange that exactly the same value for the orientation of the ray with respect to that edge will be computed—so whichever sign it is, it will register as a "hit" for one of the triangles. One way to do this is always call the orient function with the last two vertices in sorted order (sorted by vertex index in an array where they are stored, or by coordinate value, or...) and if this is the wrong way around from the test needed than simply flip the sign. (It's also important to be careful with inlining—an optimizing compiler might emit slightly different machine code for different cases, which would defeat us.)

A more conservative hack in case this doesn't work out is to include a margin of error ϵ: if orient evaluates to the range $[-\epsilon, \epsilon]$ then it gets counted as zero, which can match either sign to register an intersection. Unfortunately, an appropriate value of ϵ depends on the scene: it should be approximately proportional to, say, the cube of an edge of the triangle.

5 Barycentric Coordinates

In some scenarios, just determining whether there is an intersection or not is enough. However, we often need more information:

- where along the ray the intersection occurs (the ray parameter s), to find out which intersection comes first,
- and where in the triangle it occurs (the barycentric coordinates α, β, and γ), to linearly interpolate colours for example.

Luckily, we can extract this information easily enough from the values of the orientations.

Let's focus on just the barycentric coordinates for now, as the ray parameter value will be most easily found in the next section after we switch back to the origin+direction representation of the ray. Barycentric coordinates should be affine functions that equal zero along one edge of the triangle and sum up to one at all times. The three orientation values of the ray's segment with respect to the triangle edges are also affine functions, but their sum could be different from one—so we simply divide through by their sum:

```
S = f0134 + f0142 + f0123
alpha = f0134 / S
beta = f0142 / S
gamma = f0123 / S
```

That's all there is to it.

6 Returning to Ray Direction Form

Finally, we return to our first operation where we calculated the endpoint $\vec{x}_1 = \vec{x}_0 + s_{\max}\vec{d}$ from the ray origin, the maximum parameter value and the ray direction. We can rewrite our orientation values in terms of these quantities instead, to avoid unnecessary calculation and rounding errors.

First up is `f1234` $= \text{orient}(\vec{x}_1, \vec{x}_2, \vec{x}_3, \vec{x}_4)$. Using rules of determinants we have:

$$\begin{aligned}
\text{orient}(\vec{x}_1, \vec{x}_2, \vec{x}_3, \vec{x}_4) &= \det\begin{pmatrix} \vec{x}_1 & \vec{x}_2 & \vec{x}_3 & \vec{x}_4 \\ 1 & 1 & 1 & 1 \end{pmatrix} \\
&= \det\begin{pmatrix} \vec{x}_0 + s_{\max}\vec{d} & \vec{x}_2 & \vec{x}_3 & \vec{x}_4 \\ 1 & 1 & 1 & 1 \end{pmatrix} \\
&= \det\begin{pmatrix} \vec{x}_0 & \vec{x}_2 & \vec{x}_3 & \vec{x}_4 \\ 1 & 1 & 1 & 1 \end{pmatrix} + \det\begin{pmatrix} s_{\max}\vec{d} & \vec{x}_2 & \vec{x}_3 & \vec{x}_4 \\ 0 & 1 & 1 & 1 \end{pmatrix} \\
&= \text{orient}(\vec{x}_0, \vec{x}_2, \vec{x}_3, \vec{x}_4) + s_{\max}\det\begin{pmatrix} \vec{d} & \vec{x}_2 & \vec{x}_3 & \vec{x}_4 \\ 0 & 1 & 1 & 1 \end{pmatrix}
\end{aligned}$$

Note the zero on the bottom row of this determinant involving the direction vector. It can alternatively be expressed with cross and dot product as:

$$\texttt{d234} = \det\begin{pmatrix} \vec{d} & \vec{x}_2 & \vec{x}_3 & \vec{x}_4 \\ 0 & 1 & 1 & 1 \end{pmatrix} = \vec{d} \cdot (\vec{x}_3 - \vec{x}_2) \wedge (\vec{x}_4 - \vec{x}_2)$$

for example, similar to $\text{orient}(\vec{x}_0, \vec{x}_2, \vec{x}_3, \vec{x}_4) = (\vec{x}_0 - \vec{x}_2) \cdot (\vec{x}_3 - \vec{x}_2) \wedge (\vec{x}_4 - \vec{x}_2)$, which obviously provides some common sub-expressions. It is a simple matter to rewrite the code of the intersection test to compute and use `d234` and $s_{\max}$ instead of computing `f1234`.

If there is an intersection, the parameter value s is where the point $\vec{x}_0 + s\vec{d}$ on the ray is coplanar with the triangle's vertices:

$$\det\begin{pmatrix} \vec{x}_0 + s\vec{d} & \vec{x}_2 & \vec{x}_3 & \vec{x}_4 \\ 1 & 1 & 1 & 1 \end{pmatrix} = 0$$

This simplifies to the equation

$$\texttt{f0234} + s\texttt{d234} = 0$$

with solution

$$s = \frac{-\texttt{f0234}}{\texttt{d234}}$$

This of course should only be calculated at the end of the code, after an intersection has been found for sure.

There remain the three orientations for the triangle edges: `f0123`, `f0134` and `f0142`. These can be evaluated in terms of $\vec{d}$ too:

$$\begin{aligned} \texttt{f0123} = \text{orient}(\vec{x}_0, \vec{x}_1, \vec{x}_2, \vec{x}_3) &= \det \begin{pmatrix} \vec{x}_0 & \vec{x}_1 & \vec{x}_2 & \vec{x}_3 \\ 1 & 1 & 1 & 1 \end{pmatrix} \\ &= \det \begin{pmatrix} \vec{x}_0 & \vec{x}_0 + s_{\max}\vec{d} & \vec{x}_2 & \vec{x}_3 \\ 1 & 1 & 1 & 1 \end{pmatrix} \\ &= s_{\max} \det \begin{pmatrix} \vec{x}_0 & \vec{d} & \vec{x}_2 & \vec{x}_3 \\ 1 & 0 & 1 & 1 \end{pmatrix} \end{aligned}$$

Note that scaling all of them by the positive number $s_{\max}$ has no effect on the sign tests or the calculation of the barycentric coordinates, thus we may as well drop that factor. The determinant can also be expressed with cross and dot product:

$$\det \begin{pmatrix} \vec{x}_0 & \vec{d} & \vec{x}_2 & \vec{x}_3 \\ 1 & 0 & 1 & 1 \end{pmatrix} = -\vec{d} \cdot (\vec{x}_2 - \vec{x}_0) \wedge (\vec{x}_3 - \vec{x}_0) = -\texttt{d023}$$

Similarly $-\texttt{d034} = -\vec{d} \cdot (\vec{x}_3 - \vec{x}_0) \wedge (\vec{x}_4 - \vec{x}_0)$ replaces `f0134` and $-\texttt{d042} = -\vec{d} \cdot (\vec{x}_4 - \vec{x}_0) \wedge (\vec{x}_2 - \vec{x}_0)$ replaces `f0142`. It is left as an exercise how to rewrite the code in terms of `f0234`, `d234`, `d023`, `d034` and `d042`.