From
Akenine-Möller & Haines
Real Time Rendering, 2nd Edition
AK Peters, 2002

13.7 Ray/Triangle Intersection

In real-time graphics libraries and APIs, triangle geometry is usually stored as a set of vertices with associated normals, and each triangle is defined by three such vertices. The normal of the plane in which the triangle lies is often not stored, in which case, it must be computed if needed. There exist many different ray/triangle intersection tests, and the majority of them first compute the intersection point between the ray and the triangle's plane. Thereafter, the intersection point and the triangle vertices are projected on the axis-aligned plane (xy, yz, or xz) where the area of the triangle is maximized. By doing this, we reduce the problem to two dimensions, and we need only decide whether the (2D) point is inside the (2D) triangle. Several such methods exist, and they have been reviewed and compared by Haines [307], with code available on the web. See Section 13.8 for one popular algorithm using this technique.

Here, the focus will be on an algorithm that does not presume that normals are precomputed. For triangle meshes, this can amount to significant memory savings. This algorithm, along with optimizations, was discussed by Möller and Trumbore [560], and their presentation is used here. DirectX 8.1 uses this technique in their sample pick code.

The ray from Equation 13.1 is used to test for intersection with a triangle defined by three vertices, $\mathbf{v}_1$, $\mathbf{v}_2$, and $\mathbf{v}_3$—i.e., $\Delta\mathbf{v}_1\mathbf{v}_2\mathbf{v}_3$.

13.7.1 Intersection Algorithm

A point, $\mathbf{t}(u,v)$, on a triangle is given by the explicit formula

$$\mathbf{t}(u,v) = (1-u-v)\mathbf{v}_0 + u\mathbf{v}_1 + v\mathbf{v}_2, \tag{13.16}$$

where (u,v) are the *barycentric coordinates*, which must fulfill $u \geq 0$, $v \geq 0$, and $u + v \leq 1$. Note that (u,v) can be used for texture mapping, normal interpolation, color interpolation, etc. That is, u and v are the amounts by which to weight each vertex's contribution to a particular location, with $w = (1-u-v)$ being the third weight.[7] See Figure 13.10.

Computing the intersection between the ray, $\mathbf{r}(t)$, and the triangle, $\mathbf{t}(u,v)$, is equivalent to $\mathbf{r}(t) = \mathbf{t}(u,v)$, which yields:

$$\mathbf{o} + t\mathbf{d} = (1-u-v)\mathbf{v}_0 + u\mathbf{v}_1 + v\mathbf{v}_2. \tag{13.17}$$

[7] These coordinates are often denoted α, β, and γ. We use u, v, and w here for readability and consistency of notation.

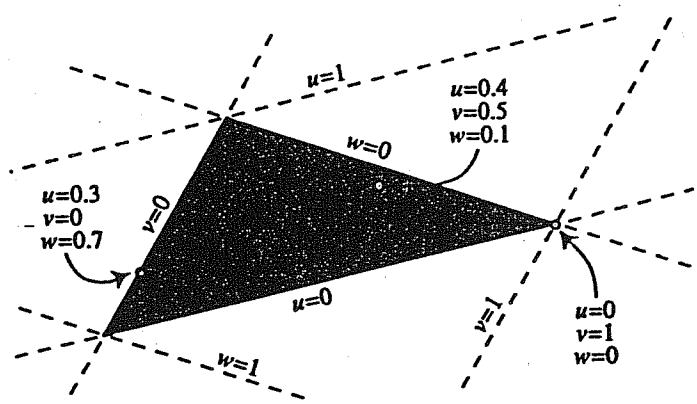

Figure 13.10. Barycentric coordinates for a triangle, along with example point values. The values u, v, and w all vary from 0 to 1 inside the triangle, and the sum of these three is always 1 over the entire plane. These values can be used as weights for how data at each of the three vertices influence any point on the triangle. Note how at each vertex, one value is 1 and the others 0, and along edges one value is always 0.

Rearranging the terms gives:

$$\begin{pmatrix} -\mathbf{d} & \mathbf{v}_1 - \mathbf{v}_0 & \mathbf{v}_2 - \mathbf{v}_0 \end{pmatrix} \begin{pmatrix} t \\ u \\ v \end{pmatrix} = \mathbf{o} - \mathbf{v}_0. \tag{13.18}$$

This means the barycentric coordinates (u, v) and the distance t from the ray origin to the intersection point can be found by solving the linear system of equations above.

The above can be thought of geometrically as translating the triangle to the origin and transforming it to a unit triangle in y and z with the ray direction aligned with x. This is illustrated in Figure 13.11. If $\mathbf{M} = (-\mathbf{d} \;\; \mathbf{v}_1 - \mathbf{v}_0 \;\; \mathbf{v}_2 - \mathbf{v}_0)$ is the matrix in Equation 13.18, then the solution is found by multiplying Equation 13.18 with $\mathbf{M}^{-1}$.

Denoting $\mathbf{e}_1 = \mathbf{v}_1 - \mathbf{v}_0$, $\mathbf{e}_2 = \mathbf{v}_2 - \mathbf{v}_0$, and $\mathbf{s} = \mathbf{o} - \mathbf{v}_0$, the solution to Equation 13.18 is obtained by using Cramer's rule:

$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \frac{1}{\det(-\mathbf{d}, \mathbf{e}_1, \mathbf{e}_2)} \begin{pmatrix} \det(\mathbf{s}, \mathbf{e}_1, \mathbf{e}_2) \\ \det(-\mathbf{d}, \mathbf{s}, \mathbf{e}_2) \\ \det(-\mathbf{d}, \mathbf{e}_1, \mathbf{s}) \end{pmatrix}. \tag{13.19}$$

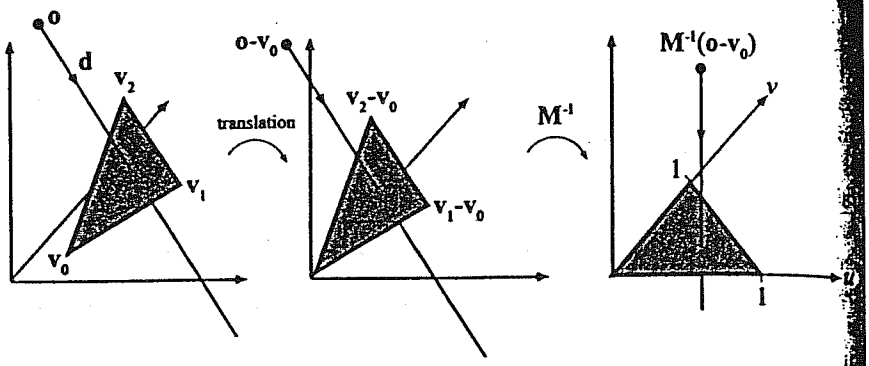

Figure 13.11. Translation and change of base of the ray origin.

From linear algebra, we know that $\det(\mathbf{a},\mathbf{b},\mathbf{c}) = |\mathbf{a}\ \mathbf{b}\ \mathbf{c}| = -(\mathbf{a}\times\mathbf{c})\cdot\mathbf{b} = -(\mathbf{c}\times\mathbf{b})\cdot\mathbf{a}$. Equation 13.19 can therefore be rewritten as:

$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \frac{1}{(\mathbf{d}\times\mathbf{e}_2)\cdot\mathbf{e}_1}\begin{pmatrix} (\mathbf{s}\times\mathbf{e}_1)\cdot\mathbf{e}_2 \\ (\mathbf{d}\times\mathbf{e}_2)\cdot\mathbf{s} \\ (\mathbf{s}\times\mathbf{e}_1)\cdot\mathbf{d} \end{pmatrix} = \frac{1}{\mathbf{p}\cdot\mathbf{e}_1}\begin{pmatrix} \mathbf{q}\cdot\mathbf{e}_2 \\ \mathbf{p}\cdot\mathbf{s} \\ \mathbf{q}\cdot\mathbf{d} \end{pmatrix}, \quad (13.20)$$

where $\mathbf{p} = \mathbf{d}\times\mathbf{e}_2$ and $\mathbf{q} = \mathbf{s}\times\mathbf{e}_1$. These factors can be used to speed up the computations.

Arenberg [24] describes an algorithm that is similar to the one above. He also constructs a 3×3 matrix, but uses the normal of the triangle instead of the ray direction $\mathbf{d}$. His method requires storing the normal for each triangle or computing each normal on the fly, while the one presented here does not.

13.7.2 Implementation

The algorithm is summarized in the pseudocode below. Besides returning whether or not the ray intersects the triangle, the algorithm also returns the above described triple (u, v, t). The code does not cull backfacing triangles.

```
     RayTriIntersect(o, d, v0, v1, v2)
     returns ({REJECT, INTERSECT}, u, v, t);
1 :  e1 = v1 − v0
2 :  e2 = v2 − v0
3 :  p = d × e2
4 :  a = e1 · p
5 :  if(a > −ε and a < ε) return (REJECT, 0, 0, 0);
6 :  f = 1/a
7 :  s = o − v0
8 :  u = f(s · p)
9 :  if(u < 0.0 or u > 1.0) return (REJECT, 0, 0, 0);
10 : q = s × e1
11 : v = f(d · q)
12 : if(v < 0.0 or u + v > 1.0) return (REJECT, 0, 0, 0);
13 : t = f(e2 · q)
14 : return (INTERSECT, u, v, t);
```

A few lines may require some explanation. Line 4 computes a, which is the determinant of the matrix M. This is followed by a test that avoids determinants close to zero. With a properly adjusted value of ϵ, this algorithm is extremely stable).[8] In Line 9, the value of u is compared to an edge of the triangle ($u = 0$), and also to a line parallel to that edge, but passing through the opposite vertex of the triangle ($u = 1$). See Figure 13.10. Although not actually testing an edge of the triangle, this second test efficiently rules out many intersection points without further calculation. However, on modern CPUs where conditional statements tend to be fairly expensive, this test may actually decrease performance.

C-code for this algorithm, including both culling and nonculling versions, is available on the web [560]. The C-code has two branches: One that efficiently culls all backfacing triangles, and one that performs intersection tests on two-sided triangles. All computations are delayed until they are required. For example, the value of v is not computed until the value of u is found to be within the allowable range (this can be seen in the pseudocode as well).

The one-sided intersection routine eliminates all triangles where the value of the determinant is negative. This procedure allows the routine's only division operation to be delayed until an intersection has been confirmed.

[8] For floating point precision, $\epsilon = 1.0^{-5}$ works fine.

10^{-5} ?

The investigation by Möller and Trumbore [560] also shows that this method is the fastest ray/triangle intersection routine that does not need to store the normal of the triangle plane, and that it is comparable in speed to Badouel's method [35], which also computes barycentric coordinates (and so makes the comparison fair).

13.8 Ray/Polygon Intersection

Even though triangles are the most common rendering primitive, a routine that computes the intersection between a ray and a polygon is useful to have. A polygon of n vertices is defined by an ordered vertex list $\{\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_{n-1}\}$, where vertex $\mathbf{v}_i$ forms an edge with $\mathbf{v}_{i+1}$ for $0 \leq i < n-1$ and the polygon is closed by the edge from $\mathbf{v}_{n-1}$ to $\mathbf{v}_0$. The plane of the polygon[9] is denoted $\pi_p : \mathbf{n}_p \cdot \mathbf{x} + d_p = 0$.

We first compute the intersection between the ray (Equation 13.1) and π_p, which is easily done by replacing $\mathbf{x}$ by the ray. The solution is presented below:

$$\begin{gathered} \mathbf{n}_p \cdot (\mathbf{o} + t\mathbf{d}) + d_p = 0 \\ \Longleftrightarrow \\ t = \frac{-d_p - \mathbf{n}_p \cdot \mathbf{o}}{\mathbf{n}_p \cdot \mathbf{d}}. \end{gathered} \tag{13.21}$$

If the denominator $|\mathbf{n}_p \cdot \mathbf{d}| < \epsilon$, where ϵ is a very small number[10], then the ray is considered parallel to the polygon plane and no intersection occurs.[11] Otherwise, the intersection point, $\mathbf{p}$, of the ray and the polygon plane is computed: $\mathbf{p} = \mathbf{o} + t\mathbf{d}$, where the t-value is that from Equation 13.21. Thereafter, the problem of deciding whether $\mathbf{p}$ is inside the polygon is reduced from three to two dimensions. This is done by projecting all vertices and $\mathbf{p}$ to one of the xy-, xz-, or yz-planes where the area of the projected polygon is maximized. In other words, the coordinate component that corresponds to $\max(|n_{p,x}|, |n_{p,y}|, |n_{p,z}|)$ can be skipped and the others kept as two-dimensional coordinates. For example, given a normal $(0.6, -0.692, 0.4)$, the y component has the largest magnitude, so all y coordinates are ignored. Note that this component information could be precomputed once and stored within the polygon for efficiency. The topology of the polygon and the intersection point is conserved during this projection (assuming the polygon is indeed flat; see Section 11.2 for more

[9] This plane can be computed from the vertices on the fly or stored with the polygon, whichever is most convenient. It is sometimes called the supporting plane of the polygon.

[10] An epsilon of 1.0^{-20} or smaller is fine, as the goal is to avoid overflowing when dividing.

[11] We ignore the case where the ray is in the polygon's plane.

ie. no intersection in this case

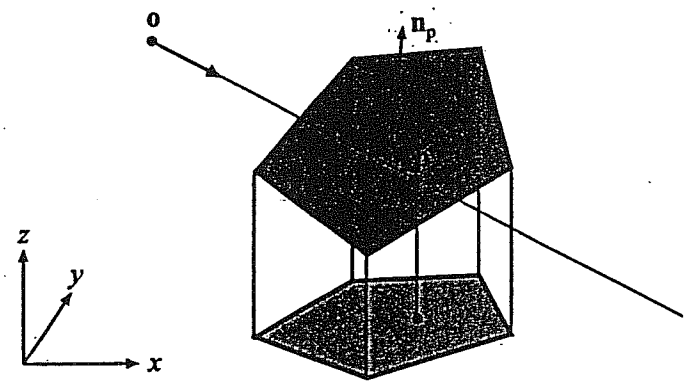

Figure 13.12. Orthographic projection of polygon vertices and intersection point p onto the xy-plane, where the area of the projected polygon is maximized. This is an example of using dimension reduction to obtain simpler calculations.

on this topic). The projection procedure is shown in Figure 13.12. A two-dimensional bounding box (in the above mentioned plane) for the polygon is also sometimes profitable. That is, first compute the intersection with the polygon plane and then project to two dimensions and test against the two-dimensional bounding box. If the point is outside the box, then reject and return; otherwise, continue with the full polygon test. This was found to be a better approach than using a three-dimensional bounding box for a polygon [824].

The question left is whether the two-dimensional ray/plane intersection point p is contained in the two-dimensional polygon. Here, we will review just one of the more useful algorithms—the "crossings" test. Haines [307] provides an extensive survey of two-dimensional, point-in-polygon strategies. More recently, Walker [786] has presented a method for rapid testing of polygons with more than 10 vertices. Also, Nishita et al. [587] discuss point inclusion testing for shapes with curved edges. A more formal treatment can be found in the computational geometry literature [56, 605, 634].

13.8.1 The Crossings Test

The crossings test is based on the *Jordan Curve Theorem*, which says that a point is inside a polygon if a ray from this point in an arbitrary direction crosses an odd number of polygon edges. This test is also known as the parity or the even-odd test. This condition does not mean that all areas enclosed by the polygon are considered inside. This is shown in Figure 13.13.

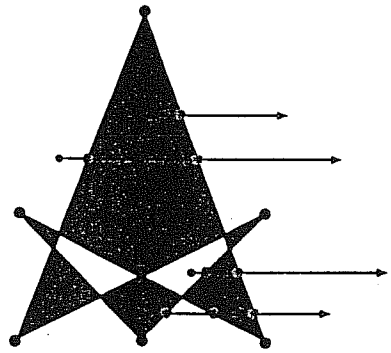

Figure 13.13. A general polygon that is self-intersecting and concave, yet all of its enclosed areas are not considered inside (only gray areas are inside). Vertices are marked with large, black dots. Three points being tested are shown, along with their test rays. According to the Jordan Curve Theorem, a point is inside if the number of crossings with the edges of the polygon is odd. Therefore, the uppermost and the bottommost points are inside (one and three crossings, respectively), while the two middle points each cross two edges and are thus considered outside the polygon.

The crossings algorithm is the fastest test that does not use preprocessing. It works by shooting a ray from the projection of the point p along the positive x-axis. Then the number of crossings between the polygon edges and this ray is computed. As the *Jordan Curve Theorem* proves, an odd number of crossings indicates that the point is inside the polygon.

The test point p can also be thought of as being at the origin, and the (translated) edges may be tested against the positive x-axis instead. This option is depicted in Figure 13.14. If the y-coordinates of a polygon edge have the same sign, then that edge cannot cross the x-axis. Otherwise, it can, and then the x-coordinates are checked. If both are positive, then the number of crossings is incremented. If they differ in sign, the x-coordinate of the intersection between the edge and the x-axis must be computed, and if it is positive, the number of crossings is again incremented.

These enclosed areas could be included as well, however; see Haines [307] for treatment.

Problems might occur when the test ray intersects a vertex, since two crossings might be detected. These problems are solved by setting the vertex infinitesimally above the ray, which, in practice, is done by interpreting the vertices with $y \geq 0$ as lying above the x-axis (the ray). The code becomes simpler and speedier, and no vertices will be intersected [305].

The pseudocode for an efficient form of the crossings test is given. It was inspired by the work of Joseph Samosky [671] and Mark Haigh-Hutchinson,

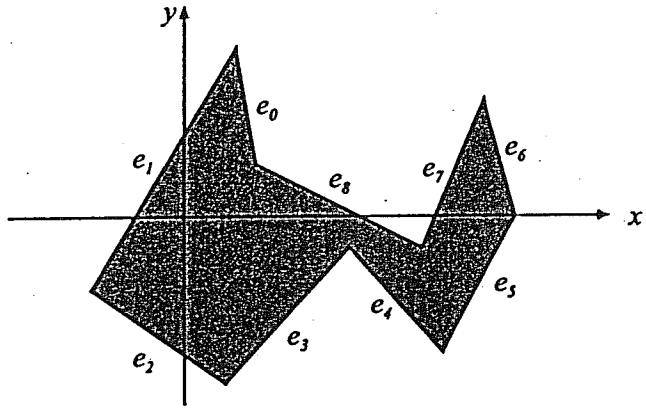

Figure 13.14. The polygon has been translated by $-\mathbf{p}$ ($\mathbf{p}$ is the point to be tested for containment in the polygon), and so the number of crossings with the positive x-axis determines whether $\mathbf{p}$ is inside the polygon. Edges e_0, e_2, e_3, and e_4 cannot cross the x-axis, since they each have both vertices on one side of the x-axis. The intersection between edge e_1 and the x-axis must be computed, but will not yield a crossing, since the intersection has a negative x-component. Edges e_7 and e_8 will each increase the number of crossings, since the vertices of these edges have positive x-components and one negative and one positive y-component. Finally, the edges e_5 and e_6 share a vertex where $y = 0$ and $x > 0$, and they will together increase the number of crossings by one.

and the code is available on the web [307]. Two-dimensional test point $\mathbf{t}$ and polygon P with vertices $\mathbf{v}_0$ through $\mathbf{v}_{n-1}$ are compared.

```
      bool PointInPolygon(t, P)
      returns ({TRUE, FALSE});
1 :   bool inside = FALSE
2 :   e0 = v_{n-1}
3 :   e1 = v0
4 :   bool y0 = (e0y ≥ ty)
5 :   for i = 1 to n
6 :      bool y1 = (e1y ≥ ty)
7 :      if(y0 ≠ y1)
8 :         if(((e1y − ty)(e0x − e1x) ≥ (e1x − tx)(e0y − e1y)) = y1)
9 :            inside = ¬inside
10 :     y0 = y1
11 :     e0 = e1
12 :     e1 = vi
13 :  return inside;
```

Line 4 checks whether the y-value of the last vertex in the polygon is greater than or equal to the y-value of the test point $\mathbf{t}$, and stores the result in the boolean y_0. In other words, it tests whether the first endpoint of the first edge we will test is above or below the x-axis. Line 7 tests whether the endpoints e_0 and e_1 are on different sides of the x-axis formed by the test point. If so, then Line 8 tests whether the x-intercept is positive. Actually, it is a bit trickier than that: to avoid the divide normally needed for computing the intercept, we perform a sign-cancelling operation here. By inverting *inside*, Line 9 records that a crossing took place. Lines 10 to 12 move on to the next vertex.

In the pseudocode we do not perform a test to see whether both endpoints have positive x-coordinates. Although this is how we presented the algorithm, code based on the pseudocode above often runs faster without this test. This is because we avoid the division needed to compute the x-intercept value, since all we want to know is whether the intercept is to the left or right of the test point. When you are optimizing this routine, we recommend trying both variants and seeing which is faster in practice.

The advantages of the crossings test is that it is relatively fast and robust, and requires no additional information or preprocessing for the polygon. A disadvantage of this method is that it does not yield anything beyond the indication of whether a point is inside or outside the polygon. Other methods, such as the ray/triangle test in Section 13.7.1, can also compute barycentric coordinates that can be used to interpolate additional information about the test point [307].