In terms of code, this routine is fast, but it has the disadvantage of not being able to return the intersection distance.

16.7.3 Ray Slope Method

In 2007 Eisemann et al. [301] presented a method of intersecting boxes that appears to be faster than previous methods. Instead of a three-dimensional test, the ray is tested against three projections of the box in two dimensions. The key idea is that for each two-dimensional test, there are two box corners that define the extreme extents of what the ray "sees," akin to the silhouette edges of a model. To intersect this projection of the box, the slope of the ray must be between the two slopes defined by the ray's origin and these two points. If this test passes for all three projections, the ray must hit the box. The method is extremely fast because some of the comparison terms rely entirely on the ray's values. By computing these terms once, the ray can then efficiently be compared against a large number of boxes. This method can return just whether the box was hit, or can also return the intersection distance, at a little additional cost.

16.8 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 many 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 (two-dimensional) point is inside the (two-dimensional) triangle. Several such methods exist, and they have been reviewed and compared by Haines [482], with code available on the web. See Section 16.9 for one popular algorithm using this technique. A wealth of algorithms have been evaluated for different CPU architectures, compilers, and hit ratios [803], and it could not be concluded that there is a single best test in all cases.

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, and for dynamic geometry, we do not need to recompute any extra data, such as the plane equation of the triangle every frame. This algorithm, along with optimizations, was discussed by Möller and Trum-

bore [890], and their presentation is used here. It is also worth noting that Kensler and Shirley [644] noted that most ray-triangle tests operating directly in three dimensions[6] are computationally equivalent. They develop new tests using SSE to test four rays against a triangle, and use a genetic algorithm to find the best order of the operations in this equivalent test. Code for the best-performing test is in the paper.

The ray from Equation 16.1 is used to test for intersection with a triangle defined by three vertices, $\mathbf{p}_1$, $\mathbf{p}_2$, and $\mathbf{p}_3$—i.e., $\triangle\mathbf{p}_1\mathbf{p}_2\mathbf{p}_3$.

16.8.1 Intersection Algorithm

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

$$\mathbf{f}(u, v) = (1 - u - v)\mathbf{p}_0 + u\mathbf{p}_1 + v\mathbf{p}_2, \tag{16.19}$$

where (u, v) are two of 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 16.9.

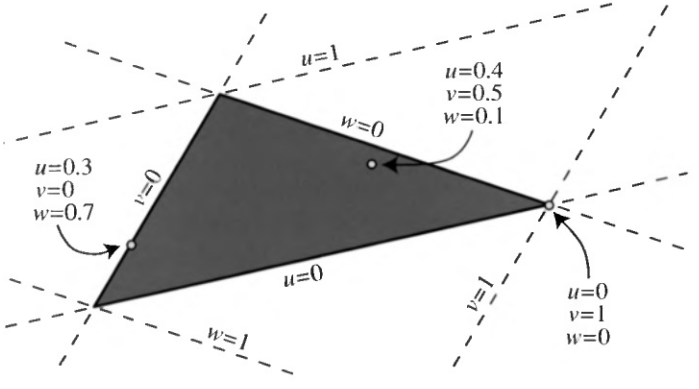

Figure 16.9. Barycentric coordinates for a triangle, along with example point values. The values u, v, and w all vary from 0 to 1 inside the triangle, and the sum of these three is always 1 over the entire plane. These values can be used as weights for how data at each of the three vertices influence 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.

[6] As opposed to first computing the intersection between the ray and the plane, and then performing a two-dimensional inside test.

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

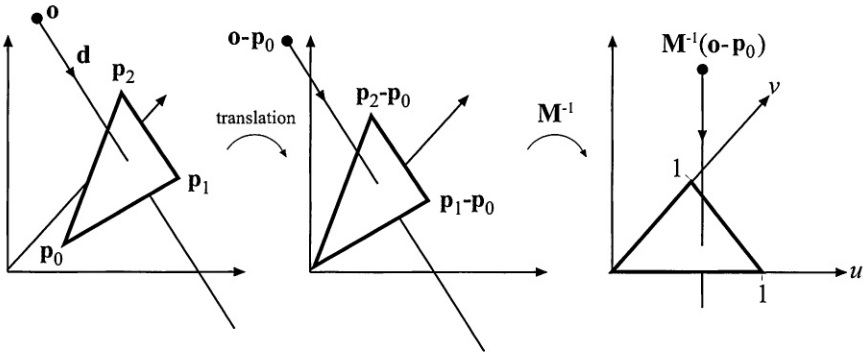

Figure 16.10. Translation and change of base of the ray origin.

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

$$\mathbf{o} + t\mathbf{d} = (1 - u - v)\mathbf{p}_0 + u\mathbf{p}_1 + v\mathbf{p}_2. \tag{16.20}$$

Rearranging the terms gives

$$\begin{pmatrix} -\mathbf{d} & \mathbf{p}_1 - \mathbf{p}_0 & \mathbf{p}_2 - \mathbf{p}_0 \end{pmatrix} \begin{pmatrix} t \\ u \\ v \end{pmatrix} = \mathbf{o} - \mathbf{p}_0. \tag{16.21}$$

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

This manipulation 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 16.10. If $\mathbf{M} = (-\mathbf{d} \quad \mathbf{p}_1 - \mathbf{p}_0 \quad \mathbf{p}_2 - \mathbf{p}_0)$ is the matrix in Equation 16.21, then the solution is found by multiplying Equation 16.21 with $\mathbf{M}^{-1}$.

Denoting $\mathbf{e}_1 = \mathbf{p}_1 - \mathbf{p}_0$, $\mathbf{e}_2 = \mathbf{p}_2 - \mathbf{p}_0$, and $\mathbf{s} = \mathbf{o} - \mathbf{p}_0$, the solution to Equation 16.21 is obtained by using Cramer's rule:

$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \frac{1}{\det(-\mathbf{d}, \mathbf{e}_1, \mathbf{e}_2)} \begin{pmatrix} \det(\mathbf{s}, \mathbf{e}_1, \mathbf{e}_2) \\ \det(-\mathbf{d}, \mathbf{s}, \mathbf{e}_2) \\ \det(-\mathbf{d}, \mathbf{e}_1, \mathbf{s}) \end{pmatrix}. \tag{16.22}$$

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 16.22 can therefore be rewritten as

$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \frac{1}{(\mathbf{d}\times\mathbf{e}_2)\cdot\mathbf{e}_1}\begin{pmatrix} (\mathbf{s}\times\mathbf{e}_1)\cdot\mathbf{e}_2 \\ (\mathbf{d}\times\mathbf{e}_2)\cdot\mathbf{s} \\ (\mathbf{s}\times\mathbf{e}_1)\cdot\mathbf{d} \end{pmatrix} = \frac{1}{\mathbf{q}\cdot\mathbf{e}_1}\begin{pmatrix} \mathbf{r}\cdot\mathbf{e}_2 \\ \mathbf{q}\cdot\mathbf{s} \\ \mathbf{r}\cdot\mathbf{d} \end{pmatrix}, \quad (16.23)$$

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

If you can afford some extra storage, this test can be reformulated in order to reduce the number of computations. Equation 16.23 can be rewritten as

$$\begin{aligned} \begin{pmatrix} t \\ u \\ v \end{pmatrix} &= \frac{1}{(\mathbf{d}\times\mathbf{e}_2)\cdot\mathbf{e}_1}\begin{pmatrix} (\mathbf{s}\times\mathbf{e}_1)\cdot\mathbf{e}_2 \\ (\mathbf{d}\times\mathbf{e}_2)\cdot\mathbf{s} \\ (\mathbf{s}\times\mathbf{e}_1)\cdot\mathbf{d} \end{pmatrix} \\ &= \frac{1}{-(\mathbf{e}_1\times\mathbf{e}_2)\cdot\mathbf{d}}\begin{pmatrix} (\mathbf{e}_1\times\mathbf{e}_2)\cdot\mathbf{s} \\ (\mathbf{s}\times\mathbf{d})\cdot\mathbf{e}_2 \\ -(\mathbf{s}\times\mathbf{d})\cdot\mathbf{e}_1 \end{pmatrix} \\ &= \frac{1}{-\mathbf{n}\cdot\mathbf{d}}\begin{pmatrix} \mathbf{n}\cdot\mathbf{s} \\ \mathbf{m}\cdot\mathbf{e}_2 \\ -\mathbf{m}\cdot\mathbf{e}_1 \end{pmatrix}, \end{aligned} \quad (16.24)$$

where $\mathbf{n} = \mathbf{e}_1\times\mathbf{e}_1$ is the unnormalized normal of the triangle, and hence constant (for static geometry), and $\mathbf{m} = \mathbf{s}\times\mathbf{d}$. If we store $\mathbf{p}_0$, $\mathbf{e}_1$, $\mathbf{e}_2$, and $\mathbf{n}$ for each triangle, we can avoid many ray triangle intersection computations. Most of the gain comes from avoiding a cross product. It should be noted that this defies the original idea of the algorithm, namely to store minimal information with the triangle. However, if speed is of utmost concern, this may a reasonable alternative. The tradeoff is whether the savings in computation is outweighed by the additional memory accesses. Only careful testing can ultimately show what is fastest.

16.8.2 Implementation

The algorithm is summarized in the pseudocode below. Besides returning whether or not the ray intersects the triangle, the algorithm also returns the previously-described triple (u, v, t). The code does not cull backfacing triangles, and it returns intersections for negative t-values, but these can be culled too, if desired.

```
     RayTriIntersect(o, d, p0, p1, p2)
     returns ({REJECT, INTERSECT}, u, v, t);
1 :  e1 = p1 − p0
2 :  e2 = p2 − p0
3 :  q = d × e2
4 :  a = e1 · q
5 :  if(a > −ε and a < ε) return (REJECT, 0, 0, 0);
6 :  f = 1/a
7 :  s = o − v0
8 :  u = f(s · q)
9 :  if(u < 0.0) return (REJECT, 0, 0, 0);
10 : r = s × e1
11 : v = f(d · r)
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 $\mathbf{M}$. This is followed by a test that avoids determinants close to zero. With a properly adjusted value of ϵ, this algorithm is extremely robust.[8] In line 9, the value of u is compared to an edge of the triangle ($u = 0$).

C-code for this algorithm, including both culling and nonculling versions, is available on the web [890]. 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.

16.9 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 $0 \leq i <$

[8] For floating point precision and "normal" conditions, $\epsilon = 10^{-5}$ works fine.