Fast Ray–Tetrahedron Intersection using Plücker Coordinates

Nikos Platis and Theoharis Theoharis
Department of Informatics & Telecommunications
University of Athens
Panepistemiopolis, GR–157 84 Ilissia, Greece
{nplatis|theotheo}@di.uoa.gr

Abstract

We present an algorithm for ray–tetrahedron intersection. The algorithm uses Plücker coordinates to represent the ray and the edges of the tetrahedron and employs a robust and efficient test to determine the intersection. The algorithm is highly optimized and provides a significant performance increase over related algorithms.

1 Introduction

Tests for intersection of directed lines with graphics primitives are at the heart of many rendering and processing algorithms. The current litterature does not treat explicitly the ray–tetrahedron intersection problem, even though tetrahedral meshes are used increasingly for the representation of volumetric models [GCMS01]. In this paper we investigate efficient solutions to this problem.

Notation and Conventions In the ray–tetrahedron algorithms that follow, the ray will be assumed an infinite directed line, determined by a point P and a direction L. The vertices of the tetrahedron will be marked V_0, V_1, V_2, V_3, and its faces by the index of the opposite vertex:

$$F_3\,(V_0V_1V_2), \quad F_2\,(V_1V_0V_3), \quad F_1\,(V_2V_3V_0), \quad F_0\,(V_3V_2V_1).$$

We will use the notation V_0^i, V_1^i, V_2^i to refer to the vertices of face F_i; for example, the vertices of face F_2 above are $V_0^2 = V_1$, $V_1^2 = V_0$ and $V_2^2 = V_3$. Similarly, the edges of face F_i will be subscripted by the index of the opposite vertex: $e_0^i(V_1^iV_2^i)$, $e_1^i(V_2^iV_0^i)$, $e_2^i(V_0^iV_1^i)$. We will assume that the tetrahedron is oriented so that the outward-pointing normal of face F_i is directed away from vertex V_i.

If the ray intersects the tetrahedron, in general there will be two intersection points, P_{enter} and P_{leave}. Special cases with one or infinite intersection points arise if the ray intersects one or more edges; these cases can be handled uniformly by the general case as shown in Figure 1, but in some applications it may be beneficial to mark them explicitly.

For the two intersection points, we require that the algorithms compute:

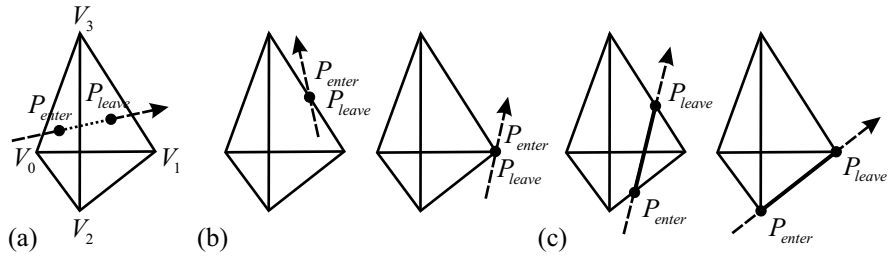

Figure 1: Ray–tetrahedron intersection: (a) two, (b) one, (c) infinite intersection points.

- Their Cartesian coordinates P_{enter} and P_{leave}.
- Their barycentric coordinates u_1^{enter}, u_2^{enter} and u_1^{leave}, u_2^{leave} with respect to the faces that they intersect, F_{enter} and F_{leave}, such that

$$P_k = (1 - u_1^k - u_2^k)V_0^k + u_1^k V_1^k + u_2^k V_2^k \quad \text{for } k = enter, leave.$$

- Their parametric distance t_{enter} and t_{leave} from the ray origin, such that

$$P_k = P + t_k L \quad \text{for } k = enter, leave.$$

2 Related Work

2.1 Approaches for ray–tetrahedron intersection

A ray–tetrahedron intersection test can be solved by a general ray–convex polyhedron intersection algorithm. Haines [Hai91] handles this general case with an algorithm that works similarly to the familiar Liang-Barsky [FvDFH96] line clipping algorithm. This algorithm is fairly efficient also for tetrahedra and is used as a base for our comparisons. It works with the parametric equation of the ray and computes the intersection points using their parametric distance from the ray origin; thus it has the advantage that restrictions concerning this distance (for example, a bound on the maximum valid distance along the ray, useful in ray tracing) can be applied early. On the other hand, it does not use barycentric coordinates, which concequently must be computed as a post-processing for the intersection points; moreover, it can recognize the special cases of intersection mentioned above only through these barycentric coordinates.

Alternatively, the ray–tetrahedron intersection problem can be solved by testing each face of the tetrahedron for intersection with the ray and combining the results; several efficient ray–triangle intersection algorithms exist, and it is important to use one that can better adapt to this specific problem. For instance, Möller and Trumbore [MT97] provide a very efficient such algorithm, which also uses the parametric equation of the ray and at the same time optimizes the computation of barycentric coordinates; however, the ray–tetrahedron intersection algorithm that we constructed using this algorithm did not perform well. In the next section we present the ray–triangle intersection algorithm that forms the basis of our ray–tetrahedron intersection algorithm.

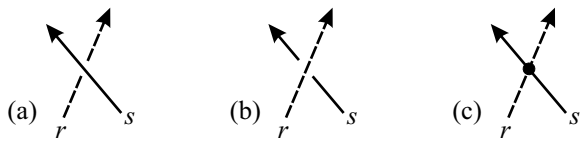

Figure 2: Relative orientation of two lines: Looking towards the direction of r, (a) s goes counterclockwise around r, (b) s goes clockwise around r; (c) s intersects r.

2.2 Ray–Triangle Intersection using Plücker Coordinates

Plücker coordinates [Eri97, Sho98, Sto91, TH99] are a way of specifying directed lines in three-dimensional space using six-dimensional vectors. Given a ray r determined by point P and direction L, its Plücker coordinates are given by the six-vector

$$\pi_r = \{L : L \times P\} = \{U_r : V_r\} \tag{1}$$

In our context, their important property is that, given two rays r and s, the *permuted inner product*

$$\pi_r \odot \pi_s = U_r \cdot V_s + U_s \cdot V_r \tag{2}$$

indicates their relative orientation (Figure 2):

$$\begin{array}{lcl} \pi_r \odot \pi_s > 0 & \Leftrightarrow & s \text{ goes counterclockwise around } r \\ \pi_r \odot \pi_s < 0 & \Leftrightarrow & s \text{ goes clockwise around } r \\ \pi_r \odot \pi_s = 0 & \Leftrightarrow & s \text{ intersects or is parallel to } r \end{array}$$

This property is the basis of a ray–triangle intersection test. Suppose we are given a ray r and a triangle $\Delta(V_0, V_1, V_2)$ with edges $e_0(V_1V_2)$, $e_1(V_2V_0)$, $e_2(V_0V_1)$. Then r intersects Δ iff it has the same orientation (cw or ccw) relative to all its edges or it intersects at most two of its edges (Figure 3). Thus the following hold:

$$\begin{array}{rcl} r \text{ intersects (enters) } \Delta & \Leftrightarrow & \pi_r \odot \pi_{e_i} \geq 0 \ \ \forall\, i \ \ \text{AND} \ \ \exists\, j : \pi_r \odot \pi_{e_j} \neq 0 \\ r \text{ intersects (leaves) } \Delta & \Leftrightarrow & \pi_r \odot \pi_{e_i} \leq 0 \ \ \forall\, i \ \ \text{AND} \ \ \exists\, j : \pi_r \odot \pi_{e_j} \neq 0 \\ r \text{ is coplanar with } \Delta & \Leftrightarrow & \pi_r \odot \pi_{e_i} = 0 \ \ \forall\, i \end{array} \tag{3}$$

This test is used successfully in [AC97] (and in [SF01] as the equivalent 4×4 determinant method) to speed up ray tracing of triangular meshes. It is robust and efficient since it requires few floating point operations, no division, and relies only on sign comparisons; moreover, calculations for an edge can be shared among neighboring triangles. Additionally, it is important that if the ray and the triangle are not coplanar, the permuted inner products $\pi_r \odot \pi_{e_i}$ provide directly the (unscaled) barycentric coordinates of the intersection point P_k with respect to the vertices V_i [Jon00]. Thus setting

$$w_i^k = \pi_r \odot \pi_{e_i} \quad \text{and} \quad u_i^k = w_i^k / \sum_{i=0}^{3} w_i^k, \tag{4}$$

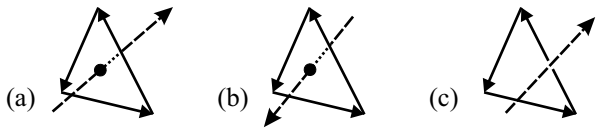

Figure 3: (a) The ray enters the triangle; (b) The ray leaves the triangle; (c) The ray and the triangle do not intersect.

the Cartesian coordinates of the intersection points can be computed as

$$P_k = u_0^k V_0 + u_1^k V_1 + u_2^k V_2.$$

Subsequently, their parametric distance from the ray origin can be trivially computed by solving for t_k a coordinate of the equation $P_k = P + t_k L$ for which L is non-zero.

3 Ray–Tetrahedron Intersection using Plücker Coordinates

3.1 Basic Algorithm

The above ray–triangle intersection test leads to a simple ray–tetrahedron intersection algorithm. In the following, referring to face F_i of the tetrahedron, we will denote by π_j^i the Plücker coordinates of edge e_j^i and by σ_j^i the sign of the permuted inner product $\pi_r \odot \pi_j^i$ for the given ray r:

$$\sigma_j^i = \text{sign}(\pi_r \odot \pi_j^i) = \begin{cases} 1, & \text{if } \pi_r \odot \pi_j^i > 0 \\ 0, & \text{if } \pi_r \odot \pi_j^i = 0 \\ -1, & \text{if } \pi_r \odot \pi_j^i < 0 \end{cases}$$

The algorithm tests each face of the tetrahedron in turn and determines if it is F_{enter} or F_{leave} according to equation (3); if either of them has been found, the relevant sign tests need not be performed for the remaining faces.

$F_{enter} = nil$
$F_{leave} = nil$
for $i = 3, 2, 1, 0$ **do**
  Compute σ_0^i, σ_1^i and σ_2^i
  if $((\sigma_0^i \neq 0)$ OR $(\sigma_1^i \neq 0)$ OR $(\sigma_2^i \neq 0))$
    if $((F_{enter} == nil)$ AND $(\sigma_0^i \geq 0)$ AND $(\sigma_1^i \geq 0)$ AND $(\sigma_2^i \geq 0))$
      $F_{enter} = F_i$
    else if $((F_{leave} == nil)$ AND $(\sigma_0^i \leq 0)$ AND $(\sigma_1^i \leq 0)$ AND $(\sigma_2^i \leq 0))$
      $F_{leave} = F_i$
    end if
  end if
end for

This algorithm has the advantage that it can compute the barycentric coordinates of the intersection points with little additional work, using equation (4).

It can also readily discern, if required, the special cases of boundary intersections (Figure 1(b,c)), which correspond to some of the σ_j^i being zero. Compared to Haines' algorithm, it has the drawback that it can accomodate restrictions concerning the parametric distance along the ray only as a post-processing step.

As we will show, this basic algorithm can be greatly optimized and adapted to the problem being solved. Some simple enhancements are possible by unrolling the **for** loop and changing the successive tests as follows:

- As soon as F_{enter} and F_{leave} are determined, the algorithm may terminate.
- The inner test needs to be performed for at most three of the four faces of the tetrahedron. If none of the three faces is intersected, the fourth will not be intersected either; otherwise it will be F_{enter} or F_{leave} depending on which face is not already found.
- Computations of the permuted inner products can be reused for the faces that share each edge. Care should be taken to adjust the sign of the product according to the edge orientation within each face; for example, edge V_0V_1 is used for face $F_3\,(V_0V_1V_2)$ whereas V_1V_0 is used for $F_2\,(V_1V_0V_3)$, and $\pi_r \odot \pi_{V_0V_1} = -(\pi_r \odot \pi_{V_1V_0})$, see equations (1),(2). This is important when the permuted inner products are not calculated in advance, since only one must be computed for each pair of opposite directed edges; it is even more important when ray-tracing tetrahedral meshes, where each edge is shared by multiple tetrahedra.

3.2 Optimizations

While the enhancements mentioned in the previous paragraph require few modifications of the basic algorithm, further optimizations are possible at the expense of increased code complexity. Our aim is to use as little information as possible in order to decide whether a face is intersected by the ray.

- We notice that the inner test examines all σ_j^i at once; however, if two of them do not agree, the third one need not be examined at all and the computation of the relevant Plücker coordinates and permuted inner product can be avoided. Care must be taken since any two (but not all three) σ_j^i may be zero. Thus the inner test for face F_i becomes:

```
Compute σ_0^i and σ_1^i
{Check if σ_0^i and σ_1^i agree, or they are zero}
if ((σ_0^i == σ_1^i) OR (σ_0^i == 0) OR (σ_1^i == 0))
    Compute σ_2^i
    {Find the sign (orientation) σ^i of the face}
    σ^i = σ_0^i
    if (σ^i == 0)
        σ^i = σ_1^i
        if (σ^i == 0)
            σ^i = σ_2^i
        end if
    end if
    {At this point, σ^i will be equal to σ_0^i or σ_1^i, whichever is non-zero;}
    {in that case it must agree with σ_2^i, or σ_2^i can be zero.}
```

```
        {If both σ_0^i and σ_1^i are zero, σ^i will be equal to σ_2^i}
        {and it must be non-zero.}
        if ((σ^i ≠ 0) AND ((σ_2^i == σ^i) OR (σ_2^i == 0)))
            if (σ^i > 0)
                F_enter = F_i
            else
                F_leave = F_i
            end if
        end if
    end if
```

- When one face intersected by the ray has been found, the above test can be simplified for the remaining faces: edges shared with the intersected face will have conforming sign and need not be tested again; furthermore, the σ_j^i for the remaining edges should satisfy $\sigma_j^i \leq 0$ if F_{leave} is not found or $\sigma_j^i \geq 0$ if F_{enter} is not found.
- When one face intersected by the ray has been found and only two faces remain untested, it is possible to determine the other one by performing only a subset of the sign tests normally required. The choice depends on the sign of their common edge: suppose that F_{leave} must be determined between $F_1\,(V_2V_3V_0)$ and $F_0\,(V_3V_2V_1)$; if $\pi_r \odot \pi_{V_2V_3} < 0$, then $F_{leave} = F_1$, otherwise $F_{leave} = F_0$ (unless all three σ_j^0 are zero, in which case also $F_{leave} = F_1$).

The last two optimizations will be especially useful when ray-tracing tetrahedral meshes. In this setting, having found a tetrahedron intersected by the ray, connectivity information of the mesh provides the tetrahedron intersected next and F_{enter} is already known for it (this holds unless the ray leaves the last tetrahedron through an edge, in which case the next tetrahedron cannot be determined directly).

3.3 Revisiting Geometry

The ray–tetrahedron intersection problem presents a geometric property which appears as a potentially useful mechanism to guide the intersection algorithm. Unfortunately, in practice it does not accelerate the fully optimized algorithm of the previous section. We describe this property here for completeness and for its geometric interest.

Suppose that the first face examined is $F_3\,(V_0V_1V_2)$. If it is intersected by the ray, the algorithm continues as described in the previous section. If not, the ray hits its supporting plane inside one of the six regions formed by the lines along the edges of F_3 (Figure 4); these regions correspond to different ranges of barycentric coordinates on this plane with respect to V_0, V_1 and V_2. Depending on the region and the ray direction, the ray may enter or leave the tetrahedron only through a specific face; if it does not, there is no intersection at all. For example, if it "enters" the plane in region (E), it may enter the tetrahedron only through $F_1\,(V_2V_3V_0)$; if it "enters" the plane in region (A), it may leave the tetrahedron only through $F_0\,(V_3V_2V_1)$. We note that the ray "enters" the plane if $\sum_{j=0}^{2}(\pi_r \odot \pi_j^3) > 0$ (it is easy to show that this is equivalent to the test $L \cdot N < 0$, where N is the face normal).

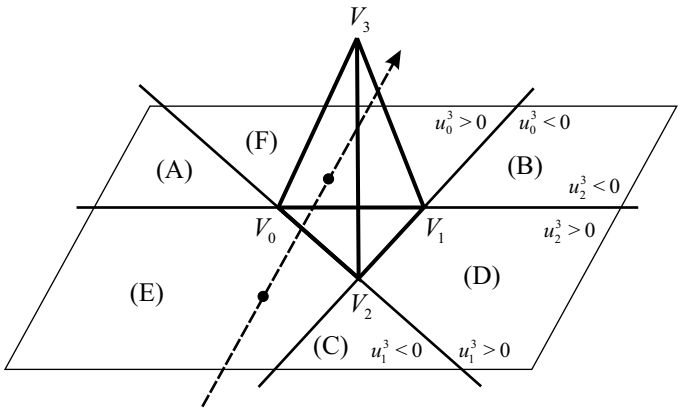

Figure 4: If a ray does not intersect $F_3\,(V_0V_1V_2)$, it intersects its supporting plane in one of the six regions shown (A–F).

Using this test is slower than testing the remaining faces as per section 3.2, for two reasons: first, all three products $\pi_r \odot \pi_j^3$, $j = 0, 1, 2$, must be computed in order to apply this property, whereas one of them could potentially be avoided as mentioned earlier; second, a rather involved examination of the barycentric coordinates is required in order to determine the region of intersection on the plane.

4 Results

We tested our algorithms on random pairs of rays and tetrahedra. Multiple sets of 10,000 random ray–tetrahedron pairs were generated, with increasing percentage of intersecting pairs from 0% to 100%; in this way the efficiency of each algorithm when different proportions of the tests succeed was assessed. Five algorithms were compared: the three variations of our algorithm corresponding to Sections 3.1, 3.2 and 3.3, Haines' algorithm adapted for tetrahedra, and an algorithm similar to the one of Section 3.2 but using the Möller/Trumbore ray–triangle test. All algorithms produce, when the test is successful, the two intersection points P_{enter} and P_{leave}, the intersected faces F_{enter} and F_{leave}, the corresponding barycentric coordinates u_1^{enter}, u_2^{enter} and u_1^{leave}, u_2^{leave}, and the parametric distances t_{enter} and t_{leave}. Barycentric coordinates for Haines' algorithm were computed as described in [BA88]. Tests were performed on an Intel Celeron PC at 900MHz running Linux.

Figure 5 attests that our algorithms using Plücker coordinates outperform the other algorithms in all cases. The optimizations of Section 3.2 provide an important performance gain, whereas the modifications of Section 3.3 have negative impact, as already noted. It is also interesting that Haines' algorithm performs comparatively better when few pairs intersect; this is due to the fact that, in some cases, it is able to decide that there is no intersection without examining all four faces of the tetrahedron, but it must process them all to find the intersection points.

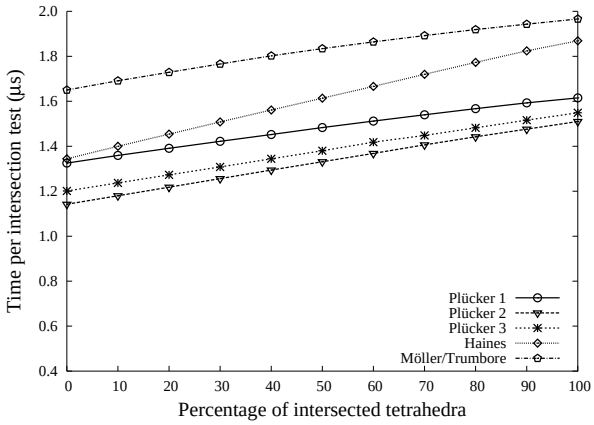

Figure 5: Results for ray–tetrahedron intersection tests.

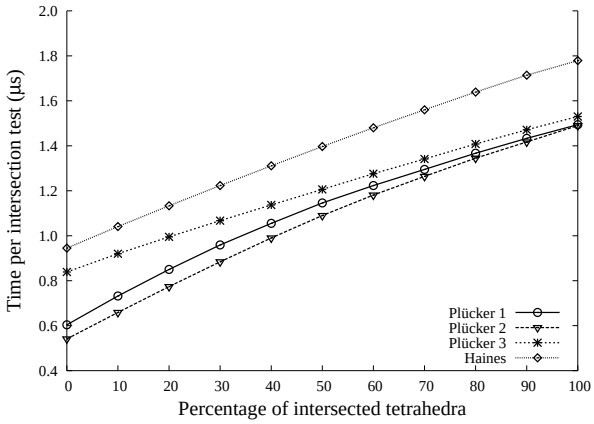

Figure 6: Results for intersection tests with precomputed quantities.

In a practical situation, a spatial data structure can be used to identify the tetrahedra along the path of the ray; for these tetrahedra several computations can be performed once for each ray. We modified the algorithms so as not to include such computations in the timings ($\pi_r \odot \pi_j^i$ and σ_j^i for all i, j in the case of our algorithms, similar relevant quantities for Haines' algorithm; the algorithm using the Möller/Trumbore test was not included in this test since it would require storing a large number of precomputed quantities). The results, shown in Figure 6, are remarkably consistent with the previous ones and affirm the potential of our algorithms. In any case, our fully optimized algorithm is the fastest.

References

[AC97] John Amanatides and Kin Choi. Ray Tracing Triangular Meshes. In *Proceedings of the Eighth Western Computer Graphics Symposium*, pages 43–52, April 1997.

[BA88] Rod Bogart and Jeff Arenberg. Ray/Triangle Intersection with Barycentric Coordinates. *Ray Tracing News*, 1(11), November 1988. `http://www.acm.org/tog/resources/RTNews/html/rtnews5b.html#art3`.

[Eri97] Jeff Erickson. Plücker Coordinates. *Ray Tracing News*, 10(3), December 1997. `http://www.acm.org/tog/resources/RTNews/html/rtnv10n3.html#art11`.

[FvDFH96] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics, Principles and Practice.* Addison-Wesley, second edition, 1996.

[GCMS01] Fabio Ganovelli, Paolo Cignoni, Claudio Montani, and Roberto Scopigno. Enabling cuts on multiresolution representation. *The Visual Computer*, 17:274–286, 2001.

[Hai91] Eric Haines. Fast Ray–Convex Polyhedron Intersection. In James Arvo, editor, *Graphics Gems II*, pages 247–250. Academic Press, 1991.

[Jon00] Ray Jones. Plücker Coordinate Tutorial. *Ray Tracing News*, 13(1), July 2000. `http://www.acm.org/tog/resources/RTNews/html/rtnv13n1.html#art8`.

[MT97] Tomas Möller and Ben Trumbore. Fast, Minimum Storage Ray/Triangle Intersection. *journal of graphics tools*, 2(1):21–28, 1997.

[SF01] Rafael J. Segura and Francisco R. Feito. Algorithms to Test Ray-Triangle Intersection. Comparative Study. In Vaclav Skala, editor, *WSCG 2001 Conference Proceedings*, February 2001.

[Sho98] Ken Shoemake. Plücker Coordinate Tutorial. *Ray Tracing News*, 11(1), July 1998. `http://www.acm.org/tog/resources/RTNews/html/rtnv11n1.html#art3`.

[Sto91] Jorge Stolfi. *Oriented Projective Geometry.* Academic Press, 1991.

[TH99] Seth Teller and Michael Hohmeyer. Determining the Lines Through Four Lines. *journal of graphics tools*, 4(3):11–22, 1999.

Web Information

Sample C++ code implementing the ray–tetrahedron intersection algorithm of section 3.2 is availabe online at

`http://www.acm.org/jgt/papers/PlatisTheoharis03`