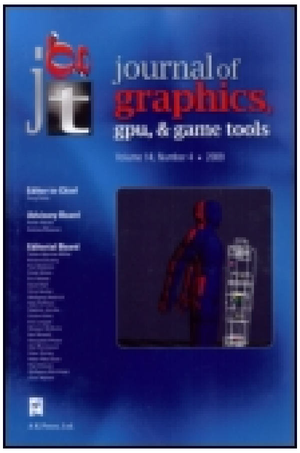

Journal of Graphics Tools
Publication details, including instructions for authors and subscription information:
http://www.tandfonline.com/loi/ujgt19

Fast, Minimum Storage Ray-Triangle Intersection

Tomas Möller [a] & Ben Trumbore [b]

[a] Prosolvia Clarus AB, Chalmers University of Technology, Gardaviigen, S-41250, Gothenberg, Sweden E-mail:

[b] Program of Computer Graphics, Cornell University, Ithaca, NY, 14853 E-mail:

Version of record first published: 06 Apr 2012.

To cite this article: Tomas Möller & Ben Trumbore (1997): Fast, Minimum Storage Ray-Triangle Intersection, Journal of Graphics Tools, 2:1, 21-28

To link to this article: http://dx.doi.org/10.1080/10867651.1997.10487468

Vol. 2, No. 1: 21–28

Fast, Minimum Storage Ray-Triangle Intersection

Tomas Möller
Prosolvia Clarus AB

Ben Trumbore
Cornell University

Abstract. We present a clean algorithm for determining whether a ray intersects a triangle. The algorithm translates the origin of the ray and then changes the base to yield a vector $(t\ u\ v)^T$, where t is the distance to the plane in which the triangle lies and (u, v) represents the coordinates inside the triangle.

One advantage of this method is that the plane equation need not be computed on the fly nor be stored, which can amount to significant memory savings for triangle meshes. As we found our method to be comparable in speed to previous methods, we believe it is the fastest ray-triangle intersection routine for triangles that do not have precomputed plane equations.

1. Introduction

A ray $R(t)$ with origin O and normalized direction D is defined as

$$R(t) = O + tD, \tag{1}$$

and a triangle is defined by three vertices V_0, V_1, and V_2. In the ray-triangle intersection problem we want to determine if the ray intersects the triangle.

Previous algorithms have solved this problem by first computing the intersection between the ray and the plane in which the triangle lies and then testing if the intersection point is inside the edges [Haines 94].

Our algorithm uses minimal storage (i.e., only the vertices of the triangle need to be stored) and does not need any preprocessing. For triangle meshes, the memory savings are significant, ranging from about 25 percent to 50 percent, depending on the amount of vertex sharing.

In our algorithm, a transformation is constructed and applied to the origin of the ray. The transformation yields a vector containing the distance t to the intersection and the coordinates (u, v) of the intersection. In this way the ray-plane intersections of previous algorithms are avoided. Note that this method has been discussed previously, by for example Patel and Shirley [Patel 96] [Shirley 96].

2. Intersection Algorithm

A point $T(u, v)$ on a triangle is given by

$$T(u,v) = (1 - u - v)V_0 + uV_1 + vV_2, \tag{2}$$

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, and more. Computing the intersection between the ray $R(t)$ and the triangle $T(u, v)$ is equivalent to $R(t) = T(u, v)$ which yields:

$$O + tD = (1 - u - v)V_0 + uV_1 + vV_2. \tag{3}$$

Rearranging the terms gives:

$$\begin{bmatrix} -D, & V_1 - V_0, & V_2 - V_0 \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = O - V_0. \tag{4}$$

This result 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, as illustrated in Figure 1 (where $M = [-D,\ V_1 - V_0,\ V_2 - V_0]$ is the matrix in Equation (4)).

Arenberg [Arenberg 88] describes a similar algorithm to the one above. He also constructs a 3×3 matrix but uses the normal of the triangle instead of the ray direction D. This method requires storing the normal for each triangle or computing it on the fly.

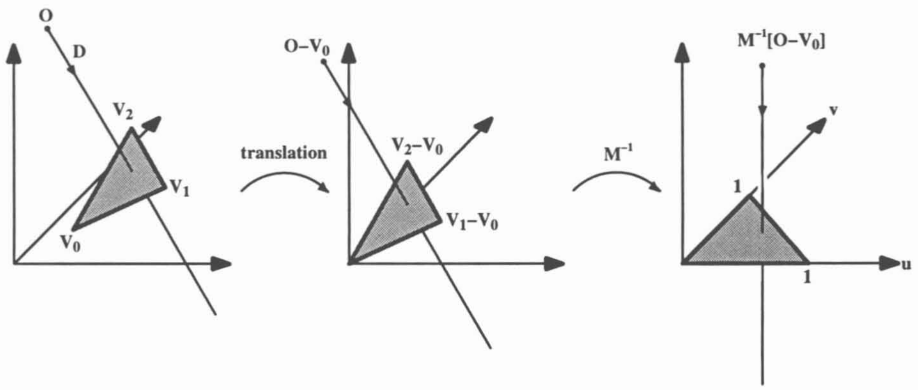

Figure 1. Translation and change of base of the ray origin.

Denoting $E_1 = V_1 - V_0$, $E_2 = V_2 - V_0$, and $T = O - V_0$, the solution to Equation (4) is obtained by using Cramer's rule:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{|-D,\ E_1,\ E_2|} \begin{bmatrix} |\,T,\ E_1,\ E_2\,| \\ |-D,\ T,\ E_2| \\ |-D,\ E_1,\ T| \end{bmatrix}. \tag{5}$$

From linear algebra, we know that $|A,\ B,\ C| = (A \times B) \cdot C = -(A \times C) \cdot B = -(C \times B) \cdot A$. Equation (5) could therefore be rewritten as

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(D \times E_2) \cdot E_1} \begin{bmatrix} (T \times E_1) \cdot E_2 \\ (D \times E_2) \cdot T \\ (T \times E_1) \cdot D \end{bmatrix} = \frac{1}{P \cdot E_1} \begin{bmatrix} Q \cdot E_2 \\ P \cdot T \\ Q \cdot D \end{bmatrix}, \tag{6}$$

where $P = (D \times E_2)$ and $Q = (T \times E_1)$. In our implementation we reuse these factors to speed up the computations.

3. Implementation

The following C implementation (available online) has been tailored for optimum performance. Two branches exist in the code; one which efficiently culls all back-facing triangles (**`#ifdef TEST_CULL`**) and the other which performs the intersection test on two-sided triangles (**`#else`**). All computations are delayed until they are required. For example, the value for v is not computed until the value of u is found to be within the allowable range.

The one-sided intersection routine eliminates all triangles where the value of the determinant (**`det`**) is negative. This procedure allows the routine's only division operation to be delayed until an intersection has been confirmed. For shadow test rays this division is not needed at all, since our only requirement is whether the triangle is intersected.

The two-sided intersection routine is forced to perform this division operation in order to evaluate the values of u and v. Alternatively, this function could be rewritten to conditionally compare u and v to zero based on the sign of `det`.

Some aspects of this code deserve special attention. The calculation of edge vectors can be done as a preprocess, with **edge1** and **edge2** being stored in place of **vert1** and **vert2**. This speedup is only possible when the actual spatial locations of **vert1** and **vert2** are not needed for other calculations and when the vertex location data is not shared between triangles.

To ensure numerical stability, the test which eliminates parallel rays must compare the determinant to a small interval around zero. With a properly adjusted **EPSILON** value, this algorithm is extremely stable. If only front-facing triangles are to be tested, the determinant can be compared to **EPSILON**, rather than zero (a negative determinant indicates a back-facing triangle).

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 point of the triangle ($u = 1$). Although not actually testing an edge of the triangle, this second test efficiently rules out many intersection points without further calculation.

```
#define EPSILON 0.000001
#define CROSS(dest,v1,v2) \
          dest[0]=v1[1]*v2[2]-v1[2]*v2[1]; \
          dest[1]=v1[2]*v2[0]-v1[0]*v2[2]; \
          dest[2]=v1[0]*v2[1]-v1[1]*v2[0];
#define DOT(v1,v2) v1[0]*v2[0]+v1[1]*v2[1]+v1[2]*v2[2]
#define SUB(dest,v1,v2)
          dest[0]=v1[0]-v2[0]; \
          dest[1]=v1[1]-v2[1]; \
          dest[2]=v1[2]-v2[2];

int
intersect_triangle(double orig[3], double dir[3],
                   double vert0[3], double vert1[3], double vert2[3],
                   double *t, double *u, double *v)
{
   double edge1[3], edge2[3], tvec[3], pvec[3], qvec[3];
   double det,inv_det;

   /* find vectors for two edges sharing vert0 */
   SUB(edge1, vert1, vert0);
   SUB(edge2, vert2, vert0);

   /* begin calculating determinant - also used to calculate U parameter */
   CROSS(pvec, dir, edge2);

   /* if determinant is near zero, ray lies in plane of triangle */
   det = DOT(edge1, pvec);

#ifdef TEST_CULL          /* define TEST_CULL if culling is desired */
   if (det < EPSILON)
      return 0;

   /* calculate distance from vert0 to ray origin */
   SUB(tvec, orig, vert0);

   /* calculate U parameter and test bounds */
   *u = DOT(tvec, pvec);
   if (*u < 0.0 || *u > det)
      return 0;

   /* prepare to test V parameter */
   CROSS(qvec, tvec, edge1);

    /* calculate V parameter and test bounds */
   *v = DOT(dir, qvec);
   if (*v < 0.0 || *u + *v > det)
      return 0;

   /* calculate t, scale parameters, ray intersects triangle */
   *t = DOT(edge2, qvec);
   inv_det = 1.0 / det;
   *t *= inv_det;
   *u *= inv_det;
   *v *= inv_det;
#else                    /* the non-culling branch */
   if (det > -EPSILON && det < EPSILON)
     return 0;
   inv_det = 1.0 / det;

   /* calculate distance from vert0 to ray origin */
   SUB(tvec, orig, vert0);

   /* calculate U parameter and test bounds */
   *u = DOT(tvec, pvec) * inv_det;
   if (*u < 0.0 || *u > 1.0)
     return 0;

   /* prepare to test V parameter */
   CROSS(qvec, tvec, edge1);
   /* calculate V parameter and test bounds */
   *v = DOT(dir, qvec) * inv_det;
   if (*v < 0.0 || *u + *v > 1.0)
     return 0;

   /* calculate t, ray intersects triangle */
   *t = DOT(edge2, qvec) * inv_det;
#endif
   return 1;
}
```

4. Results

Badouel [Badouel 90] presented a ray-triangle intersection routine that also computes the barycentric coordinates. We compared this method to ours. The two nonculling methods were implemented in an efficient ray tracer. Figure 2 presents ray-tracing runtimes from a Hewlett-Packard 9000/735 workstation for the three models shown in Figures 3–5. In this particular implementation, the performance of the two methods is roughly comparable (detailed statistics are available online).

Model	Objects	Polygons	Lights	Our method sec.	Badouel sec.
Car	497	83408	1	365	413
Mandala	1281	91743	2	242	244
Fallingwater	4072	182166	15	3143	3184

Figure 2. Contents and runtimes for data sets in Figures 3–5.

5. Conclusions

We present an algorithm for ray-triangle intersection which we show to be comparable in speed to previous methods while significantly reducing memory storage costs, by avoiding storing triangle plane equations.

Acknowledgments. Thanks to Peter Shirley, Eric Lafortune, and the anonymous reviewer whose suggestions greatly improved this paper.

This work was supported by the NSF/ARPA Science and Technology Center for Computer Graphics and Scientific Visualization (ASC-8920219), and by the Hewlett-Packard Corporation and by Prosolvia Clarus AB.

Figure 3. Falling Water.

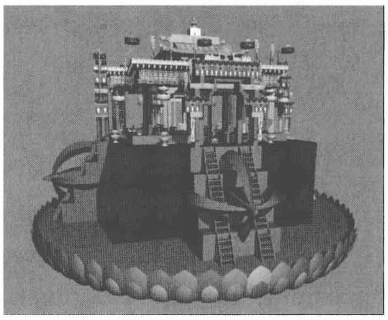

Figure 4. Mandala.

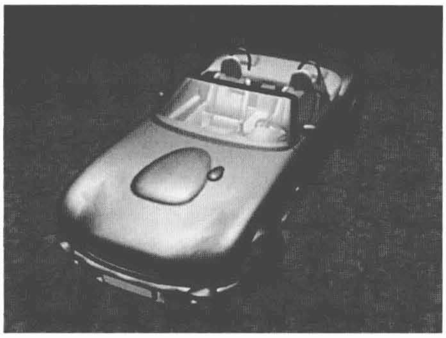

Figure 5. Car (model is courtesy of Nya Perspektiv Design AB).

References

[Arenberg 88] Jeff Arenberg. "Re: Ray/Triangle Intersection with Barycentric Coordinates." *Ray Tracing News* 1(11):(November 4, 1988). `http://www.acm.org/tog/resources/RTNews/`.

[Badouel 90] Didier Badouel. " An Efficient Ray-Polygon Intersection." In *Graphics Gems*, edited by Andrew S. Glassner, pp. 390–393. Boston: Academic Press Inc., 1990.

[Haines 94] Eric Haines. "Point in Polygon Strategies." In *Graphics Gems IV*, edited by Paul S. Heckbert, pp. 24–46. Boston: AP Professional, 1994.

[Patel 96] Edward Patel. Personal communication, 1996.

[Shirley 96] Peter Shirley. Personal communication, 1996.

Web Information:

Source code, statistical analysis, and images are available online at
http://www.acm.org/jgt/papers/MollerTrumbore97/

Tomas Möller, Prosolvia Clarus AB, Chalmers University of Technology,
Gårdavägen, S-41250, Gothenberg, Sweden (tompa@clarus.se)

Ben Trumbore, Program of Computer Graphics, Cornell University,
Ithaca, NY 14853 (wbt@graphics.cornell.edu)

Received June 17, 1996; accepted October 17, 1996