An Evaluation Framework for Ray-Triangle Intersection Algorithms

Marta Löfstedt
Göteborg University

Tomas Akenine-Möller
Lund University

August 23, 2004

Abstract

We show that comparing geometrical algorithms, in general, is very hard to do in a fair way. Our focus here is on the existing ray-triangle intersection test routines. A set of rules are developed so that a fair comparison can be produced, and this is all implemented in an evaluation framework. If all algorithms are evaluated using different hit rates, on different machines, for different input and output data, and different compilers then it is impossible to single out one "best" algorithm. A program called a "ray-triangle advisor" is developed that people can use in order to determine which algorithm works best under their particular circumstances.

1 Introduction

In computer graphics and geometrical algorithms, there is often an $O(1)$-algorithm involved. For example, in collision detection, both bounding volume vs. bounding volume and triangle/triangle overlap tests are used. These are often called a very large number of times, and can therefore significantly affect the performance. However, there does not appear to be a systematic way of determining which of such algorithms is performing best. In this paper, we concentrate on evaluating different ray-triangle intersection algorithms, but we believe that our approach is applicable to other $O(1)$-algorithms as well. Ray-triangle intersection is often used in ray tracers, for simple collision detection, and for picking. We have found more than 13 different algorithms which have been published over the years and there is still no simple answer to which algorithm that is the fastest.

In the first author's master's thesis [11], it has been concluded that no single fastest algorithm can be singled out, since performance depends on many different factors, such as compiler, computer, application, hit rate, and type of triangle data. The purpose of this paper is to introduce a framework which can be used to evaluate different ray-triangle intersection test algorithms. This test framework should be fair to all algorithms, and be able to provide information on which algorithm a user should choose for best performance on a specific platform. The entire framework and all implementations of the ray-triangle intersection algorithms are put into the public domain.

2 Evaluation Framework

While reviewing publications considering ray-triangle intersection algorithms we found no consensus on how to evaluate their performance. This is unfortunate since it makes it difficult to draw any conclusion on which is the fastest such algorithm based on previous research.

To develop a more general method of testing, we outline some problems we have found with previous test approaches and then suggest solutions. The problems have been:

I. The test have been performed on too few machines, using only a single compiler and including only a few of the available algorithms,

II. all algorithms have been tested on the same input,

III. the algorithms have not calculated the same type of output,

IV. the algorithms have not reported the same amount of hits in a test set,

V. the test set has only had a single hit rate (i.e., the number of intersected triangles divided by the number of tested triangles).

The solution to problem I is simply to test on more machines with several different compilers, and to test all available algorithms. Problem II arises since different algorithms may benefit from different input data. For example, algorithms that use the triangle plane equation may benefit from having this precalculated and stored with the triangle description. On the other hand storing extra information that an algorithm do not need may be disadvantageous in terms of cache performance.

In order to test the included algorithms using their best input data, we provide five[1] different triangle description (assuming each point or vector is stored using three floats):

1. Only triangle vertices [9 floats]

2. Triangle vertices, triangle plane equation and projection indices [13 floats + 3 ints]

3. Plücker coordinates of the triangle edges, and one triangle vertex for t-value calculation [21 floats].

4. Inverse matrix for the Arenberg algorithm [2] and a triangle vertex for t-value calculation [12 floats]

5. Halfplane equations for each edge, triangle plane equation and projection indices [13 floats + 2 ints]

[1] Obviously, it is possible to create more different types of input data, but those reported are the ones that we found most useful.

In this manner, each algorithm can use a triangle description from the list above that the algorithm benefits the most from. We also find it interesting to test every algorithm with the first triangle description type, since it will meet minimal storage requirements and force every algorithm to calculate everything "on-the-fly."

Problem III stems from the fact that the included algorithms do not calculate the same output data that may be useful in different types of applications. We noticed that most of the algorithms that provide, for example, barycentric coordinates or the t-value can be rewritten into faster versions that instead compute "scaled" versions of the same parameters. We therefore suggest that every algorithm should be tested in three different versions. The first version computes both barycentric coordinates and the t-value, the second version computes only the t-value, and for the third no extra information needs to be computed. These three versions were selected because we find that they correspond to the three major usages for ray-triangle intersection algorithms. The first would be useful when tracing regular rays in a ray tracer, the second for intersection testing, and the third may be used when tracing shadow rays in a ray tracer.[2]

An algorithm that intersects a triangle with a line segment can report different results than an algorithm that intersects with a semi-infinite ray. The same problem occurs because some algorithms report intersection behind the ray, e.g., for negative t-values, and some report hits only when the triangle vertices are ordered in counterclockwise order as seen along the ray. Thus, problem IV occurs when algorithms that use different definitions of what a "hit" is are compared. To be fair to all algorithms we must assure that every algorithm report the same amount of hits. This would be very difficult if we test every ray against every triangle as in some previous publications. We therefore suggest that the test set should be composed of pairs of rays and triangles that let us control the positions of the ray versus the triangle. This is easily achieved by composing the ray and triangle from randomized points on the surface of the unit sphere while controlling that the counterclockwise relationship is met.

Some algorithms perform best for test sets where the hit rate is very low (say $< 10\%$), while other perform better when the majority of intersections tests are hits. Furthermore, some applications may have a small percentage of hits, while others have almost only hits. This is what we refer to as problem V. To make our test fair we suggest that several different test sets with different hit rates are created. The previous style of testing every ray against every triangle will make it difficult to control the hit rate in a test set, but our suggested method of generating pairs of rays and triangle provides a simple solution. We simply test if a ray and triangle pair describes a hit or a miss and reject or include this to compose a test set with whatever hit rate we desire to test. Thus, the creation of the test sets are done as a preprocess.

machine	OS	gcc	.NET	Borland	CodeWarrior
G4 450 MHz	Mac OS X	*			*
eMac 800 MHz	Mac Os X	*			
iMac 800 MHz	Mac Os 9				*
G5 dual 2.0GHz	OsX	*			
Ultra Sparc 10	Solaris	*			
Sun Blade 150	Solaris	*			
Sun Blade 1000	Solaris	*			
Sun Blade 2000	Solaris	*			
Celeron 600 MHz	Mandrake Linux/Windows XP	*	*	*	
Celeron 2.0 GHz	Windows XP	*	*	*	
P3 933 MHz	Windows XP	*	*	*	
P3 1000 MHz	Free BSD	*			
P4 1.8 GHz	Free BSD	*			
P4 2.4 GHz	Windows XP	*	*	*	
P4 2.6 GHz	Windows XP	*	*	*	
AMD Athlon 1,33 GHz	Free BSD	*			
AMD Athlon 1,33 GHz(Laptop)	Debian Linux	*			
AMD Athlon 1,73 GHz	Windows XP	*	*	*	

Table 1: Machines and compilers. With the gcc and the Borland compiler we use the O2 optimization. With the CodeWarrior and .NET compilers we use full optimization for speed.

A122 The Shimrat Algorithm 122 [16]. **t-value.**

A2D The Area 2D algorithm based on O'Rourke [14]. **t-value.**

AR The Arenberg algorithm [2]. **t-value, barycentric coordinates.**

AR2 The Arenberg algorithm but with a late division in the same manner as in the **MT2** and **HF2** and coded to only handle counterclockwise cases. **t-value, barycentric coordinates.**

BA The Badouel algorithm [3]. **t-value, barycentric coordinated.**

BO The Bounding planes algorithm [1]. **t-value.**

CH1 This is a version of the **CH3** algorithm without the branching on different projected planes and with an optimization in the number of calculations. **t-value(scaled)**

CH2 This is the Chirkov algorithm with code provided by the author [4]. **t-value(scaled)**

CH3 This is the Chirkov algorithm with the same branching as **CH2**, but with the optimization on calculations as in **CH1. t-value(scaled)**

CR This is a crossings test based algorithm inspired by Haines [7].**t-value**

ER This an adaptation of the ERIT ray-triangle intersection algorithm [9]. **t-value**

HF This is an adaptation of the halfplane algorithm as discussed by Green [6]. **t-value**

HF2 A version of the halfplane algorithm where the tests are performed on scaled values of the intersection point and the division can be performed after we know we have a hit. **t-value(scaled)**

MA This is the Mahovsky optimization of the Plücker algorithm [12].

MT0 The original version of the Möller-Trumbore algorithm [13]. **t-value, barycentric coordinates.**

MT1 The Möller-Trumbore algorithm where the tests are done on unscaled versions of the barycentric coordinates leaving the division until we know we have a hit. **t-value, barycentric coordinates.**

MT2 The Möller-Trumbore algorithm with different branching on different signs of the determinant to divide with. The division is before the tests. **t-value, barycentric coordinates.**

MT3 The Möller-Trumbore as above but with a crossproduct done before branching on determinant sign. **t-value, barycentric coordinates.**

OR This version of the volume of tetrahedron algorithm uses the determinant calculation as suggested by O'Rourke [14], but with a slightly optimized version of the tests.

ORC This is a version of **OR** which only works with triangles which are counter-clockwise versus the ray.

PU This is a Plücker algorithm without the optimizations in **MA**. For references see [1, 5, 10, 17].

SE This is a version of the volume of tetrahedron algorithm with calculation of determinant as in **OR**, but with tests on signs and branching as suggested by Segura and Feito [15].

SNY This is an implementation of the Snyder-Barr algorithm [18]. **t-value, barycentric coordinates.**

SU The Sunday algorithm, with code provided by the author [19]. **t-value, barycentric coordinates.**

Table 2: The different algorithms and what extra information they originally were designed to calculate.

Machine	OS	Com	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	Sum
G4 0.45 GHz	Os 9	cw	AR2i	AR2i	ARi	ARi	ARi	ARi	ARi	ARi	ARi	ARi	ARi	ARi
G4 0.45 GHz	Os X	gcc	OR	MT0	MT0	MT0	MT0	MT0	MT0	MT0	MT0	MT0	MT0	MT0
eMac 0.8 GHz	Os X	gcc	MT1	HF2h	HFh	HF2h	HFh	MT1	HFh	HFh	HFh	HFh	ARi	HFh
iMac 0.8 GHz	Os 9	cw	ARi	ARi	ARi	ARi	ARi	ARi	ARi	ARi	ARi	ARi	ARi	ARi
G5 dual 2.0GHz	OsX	gcc	OR	HF2h	HF2h	HF2h	HF2h	AR2i	AR2i	AR2i	AR2i	AR2i	AR2i	AR2i
Ultra Sparc 10	Sol	gcc	ORC	ORC	ORC	ORC	ORC	MT2	MT2	MT2	ARi	ARi	ARi	ORC
Sun Blade 150	Sol	gcc	ORC	ORC	ORC	ORC	ORC	ORC	ORC	MT2	MT2	MT2	MT2	ORC
Sun Blade 1000	Sol	gcc	ORC	ORC	ORC	ORC	ORC	ORC	ARi	ARi	ARi	ARi	ARi	ORC
Sun Blade 2000	Sol	gcc	ORC	ORC	ORC	ORC	ORC	ARi	ARi	ARi	ARi	ARi	ARi	ORC
Celeron 0.6 GHz	Lin	gcc	CH2p	CH3p	CH3p	CH1p	MT2	MT2	MT2	MT2	MT2	ARi	ARi	MT2
Celeron 0.6 GHz	XP	gcc	CH2p	CH3p	CH1p	CH1p	AR2i	AR2i	ARi	ARi	ARi	ARi	ARi	AR2i
Celeron 0.6 GHz	XP	.NET	MT1	MT1	MT1	MT1	MT1	MT1	MT1	MT1	MT1	MT1	MT1	MT1
Celeron 0.6 GHz	XP	Bo	MT1	MT1	MT1	MT1	MT1	MT1	MT1	MT1	MT1	MT1	MT1	MT1
Celeron 2.0 GHz	XP	gcc	CH1p	CH2p	HF2h	CH2p	HF2h	HF2h	HF2h	HF2h	HF2h	HF2h	HFh	HF2h
Celeron 2.0 GHz	XP	.NET	CH1p	CH1p	CH1p	CH1p	CH1p	AR2i	AR2i	AR2i	AR2i	AR2i	AR2i	AR2i
Celeron 2.0 GHz	XP	Bo	CH1p	CH2p	HF2h	HF2h	HF2h	HF2h	HF2h	HF2h	HF2h	HF2h	HF2h	HF2h
P3 0.933 GHz	XP	gcc	CH3p	CH3p	CH3p	AR2i	AR2i	AR2i	AR2i	MT1	MT1	MT2	MT2	MT1
P3 0.933 GHz	XP	.NET	AR2i	AR2i	AR2i	MT1	MT1	MT1	AR2i	MT1	MT1	MT1	MT1	MT1
P3 0.933 GHz	XP	Bo	MT1	MT1	MT1	MT1	MT1	MT1	MT1	MT1	MT1	MT1	MT1	MT1
P3 1.0 GHz	BSD	gcc	CH1p	OR	OR	ARi	ARi	ARi	ARi	ARi	ARi	ARi	ARi	ARi
P4 1.8 GHz	BSD	gcc	CH1p	HFh	AR2i	HFh	HFh	HFh	HFh	HFh	HFh	HFh	HFh	HFh
P4 2.4 GHz	XP	gcc	CH2p	CH1p	CH1p	HF2h	HF2h	HF2h	CH1p	HF2h	HF2h	HF2h	HF2h	HF2h
P4 2.4 GHz	XP	.NET	CH1p	HF2h	AR2i	CH1p	HF2h	AR2i	AR2i	AR2i	AR2i	AR2i	AR2i	AR2i
P4 2.4 GHz	XP	Bo	CH2p	CH1p	HF2h	HF2h	HF2h	HF2h	HF2h	HF2h	HF2h	HF2h	HF2h	HF2h
P4 2.6 GHz	XP	gcc	CH1p	CH2p	HF2h	CH1p	HF2h	HF2h	HF2h	CH1p	HF2h	HF2h	HF2h	HF2h
P4 2.6 GHz	XP	.NET	CH3p	CH1p	CH1p	HF2h	AR2i	HF2h	AR2i	AR2i	AR2i	AR2i	AR2i	AR2i
P4 2.6 GHz	XP	Bo	CH1p	CH2p	CH1p	HF2h	HF2h	HF2h	HF2h	HF2h	HF2h	HF2h	HF2h	HF2h
AMD 1,33 GHz	BSD	gcc	CH3p	CH3p	OR	MT1	ARi	MT1	ARi	MT1	ARi	MT1	ARi	MT1
AMD 1,33 GHz	Lin	gcc	ORC	MT1	MT2	MT1	OR	MT1	MT1	MT0	MT2	MT3	MT1	MT1
AMD 1,73 GHz	XP	gcc	OR	ORC	ORC	MT1	MT1	MT1	MT0	MT0	MT3	MT1	MT0	MT0
AMD 1,73 GHz	XP	.NET	ORC	OR	CH3p	PU	AR2i	OR	OR	MT3	ORC	MT1	PU	OR
AMD 1,73 GHz	XP	Bo	PU	OR	OR	MT2	AR2i	AR2i	AR2i	AR2i	AR2i	AR2i	AR2i	AR2i

Table 3: Fastest algorithms on the barycentric test at different hit rates on all machines. In the last column the algorithm with the smallest summarized runtime over all hit rates is listed. A first thing to notice in this table is the difference in fastest algorithm between machines, hit rates and compilers. We should also notice that some machines as the G4 450Mhz, Blade 150, Celeron 600MHz, P3 933MHz and AMD 1,33GHz seem to prefer algorithms not using any precalculated data and that algorithms such as the **OR, HF, CH** and **PU** not originally intended to calculate barycentric coordinates may still be fastest at the barycentric test. When referring to the algorithms we use lowercase letters to describe which triangle structures that was used, where p = precalculated plane equation, pl = precalculated Plücker coordinates, i = precalculated inverse for the Arenberg algorithm, hf = precalculated halfplane equations.

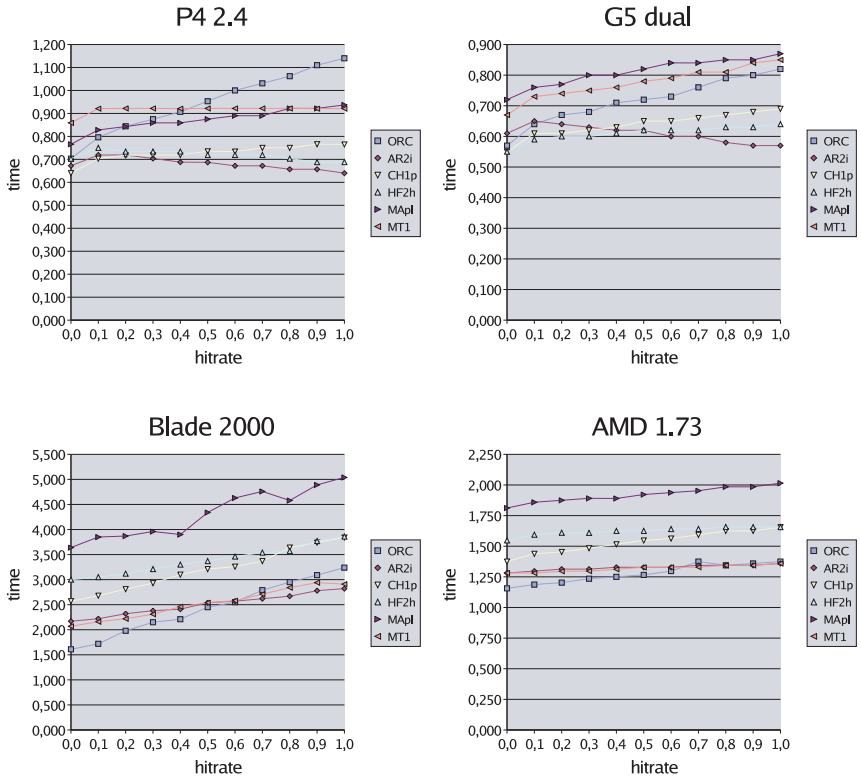

Figure 1: Some of the fastest algorithms with runtime plotted as a function of hit rate. All algorithms calculate barycentric coordinates and t-value. For the P4 and AMD the .NET compiler was used and for the G5 and Blade machines the gcc compiler was used. Note the negative slope for some of the algorithms on the P4 and G5 machines. Since the algorithms return early on a miss, more instructions have to be executed to reach a hit and therefore the runtime should increase with the hit rate and we should not get a negative slope. We believe that this behavior is caused by branch prediction mechanisms in the newer P4 and G5 architectures and the fact that we use a rather large testset of 100,000 ray-triangle pairs.

3 Some Results

The evaluation framework described above has been implemented in C, compiled with different compilers and run on 18 different machines as described in Table 1. The included algorithms are listed in Table 2.

The major result from this test is that which algorithm is still fastest depends on machine, compiler and hit rates. This is illustrated in Table 3 where we list the fastest algorithms on all of our tested machines at 11 hit rates. The difference between machines and hit rate is further described in Figure 1, where we plot the runtime as a function of hit rate on some machines for some of the fastest algorithms[3].

4 Ray-Triangle Advisor

In the text above, we have argued and empirically proved that there does not exist such a thing as *the fastest ray-triangle intersection algorithm*. Unfortunately, this leaves us with a problem when picking an algorithm for a particular project. To give some help with this decision we developed a simplified version of the test discussed above which we call the *ray-triangle advisor*. To save runtime we excluded algorithms that have not been among the fastest on the tests described above. To simplify the analysis of the results we only consider the summed run times over different hit rates and let the user decide which hit rates to summarize over. This gives the user maximal flexibility in setting the parameters so that an algorithm can be chosen that suits her needs. The source code and information on how to use this is available from `http://www.cs.lth.se/~tam/raytri/advisor.tar.gz`.

5 Discussion

We believe that our experience in this work can be used in other studies of geometrical intersection and overlap tests as well. It may not always be feasible to perform such a large study as ours in order to determine which algorithm is best. However, we strongly believe that an attempt should be made to follow the guidelines from Section 2. At the very minimum, we believe that all existing algorithms should be included in the test, and a relative large set of different input data with different hit rates should be used as this gives the reader a hint on when the algorithm will work well and when it will not. In our tests, we used 11 different sets with hit rates of $10k\%$, where $k \in [0, 10]$. Presenting diagrams with execution time on the y-axis, and the hit rate on the x-axis often provide a lot of information about the algorithm's characteristics.

[2]This may not be applicable for algorithms that report intersection on a negative t-value, but since there may be certain data structures that could prevent us from traversing negative t-value space, we still find it interesting to test this class of algorithms in their faster versions.

[3]A more thorough discussion on the results can be found in the first author's master's thesis [11].

References

[1] Amanatides, John, and Kin Choi, "Raytracing Triangular Meshes," *Proceedings of the Eighth Western Computer Graphics Symposium*, April 1997, pp 43–52.

[2] Arenberg, Jeff, "Re: Ray/Triangle Intersection with Barycentric Coordinates," *Ray Tracing News*, ed. Eric Haines, Vol 1, No 11, November 4, 1988, available from `http://turing.acm.org/pubs/resources/RTNews/html/rtnews5b.html`

[3] Badouel, Didier, "An Efficient Ray-Polygon Intersection," *Graphic Gems*, ed. Andrew S. Glassner, Academic Press 1990, pp. 390–393.

[4] Chirkov, Nick ,"Even Faster Ray-triangle Intersection with Minimum-Storage Requirements," submitted.

[5] Ericsson, Jeff, "Plucker Coordinates," in *Ray Tracing News*, ed. Eric Haines, Vol 10, No 3, December 2, 1997, available from `http://turing.acm.org/pubs/resources/RTNews/html/rtnv10n3.html`

[6] Green, Chris, "Simple fast triangle intersection," in *Ray Tracing News*, ed. Eric Haines, Vol 1, No 11, November 4, 1988, available from `http://turing.acm.org/pubs/resources/RTNews/html/rtnv6n1.html`

[7] Haines, Eric, "Essential Raytracing Algorithms," in *An Introduction to Raytracing*, Andrew S. Glassner ed., Academic Press, 1989, pp. 33–77.

[8] Haines, Eric, "Point in Polygon Strategies," in *Graphic Gems IV*, ed. Paul S. Heckbert, Academic Press, 1994, pp. 24–46.

[9] Held, Martin, "ERIT, A Collection of Efficient and Reliable Intersection Tests," *Journal of Graphic Tools*, Vol. 2, no. 4, pp. 25–44.

[10] Jones, Ray, "Intersecting a Ray and a Triangle with Plucker Coordinates," in *Ray Tracing News*, ed. Eric Haines, Vol 13, no. 1, July 16, 2000, available from `http://turing.acm.org/pubs/resources/RTNews/html/rtnv13n1.html`

[11] Löfstedt, Marta, *An Evaluation Framwork for Ray-Triangle Intersection*, Master thesis in computer science. Gothenburg University, 2004. `http://www.cs.lth.se/~tam/raytri/tri_ray_lofstedt.ps`

[12] Mahovsky, Jeffrey, "A Simple Ray-Triangle Plucker Optimization," in *Ray Tracing News*, ed. Eric Haines, Vol 15, October 21, 2002, available from `http://turing.acm.org/pubs/resources/RTNews/html/rtnv151.html`

[13] Möller, Tomas and Ben Trumbore, "Fast, Minimun Storage Ray/Triangle Intersection," *Journal of Graphics Tools*, vol 1, no 2, 1996 pp. 31–50.

[14] O'Rourke, Joseph, - *Computational Geometry in C*, pp. 226–241, Academic Press, 1998, 2nd edition.

[15] Segura, Rafael J., and Fransisco R. Feito, “Algorithms To Test Ray-Triangle Intersection. A Comparative Study,” *WSCG 2001 Conference Proceedings*, ed. Vaclav Skala, February 2001.

[16] Shimrat, M., “Algorithm 112, position of point realive to polygon,” *Communications of the ACM*, 5:434, 1962 and Hacker - Certification of algorithm 112, communications of the ACM, 5:606, 1962.

[17] Shoemake, Ken, “Plucker Coordinate Tutorial,” in *Ray Tracing News*, ed. Eric Haines, Vol 11, No 1, July 11, 1998, available from `http://turing.acm.org/pubs/resources/RTNews/html/rtnv11n1.html`

[18] Snyder, John M., and Alan H. Barr, “Ray Tracing Complex Models Containing Surface Tesselations,” *Computer Graphics (Proceedings of SIGGRAPH)*, vol. 21, no. 4, 1987.

[19] Sunday, Dan, “Intersection of Rays and Segments with Triangles in 3D,” unpublished, available from `http://geometryalgorithms.com/Archive/algorithm_0105/algorithm.html`