COMPUTER ANIMATION AND VIRTUAL WORLDS
Comp. Anim. Virtual Worlds 2014; **25**:553–559
Published online 11 October 2013 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cav.1558

RESEARCH ARTICLE

A faster triangle-to-triangle intersection test algorithm

Ling-yu Wei*

Institute for Interdisciplinary Information Sciences, Tsinghua University, Beijing, China

ABSTRACT

The triangle-to-triangle intersection test is the most basic component of collision detection. And our algorithm, which firstly computes the line segment between triangle A and the plane of triangle B and uses a new method to detect the intersection between this line and triangle B, can reduce about 10% of time on average, compared with the previous fastest algorithm. Our new method divides the plane of triangle B into four quarter planes by two edges of B, and detects intersection depending on the location of the two endpoints of the segment. After using some techniques like avoiding division and projecting the segment and triangle B on XY, YZ, or ZX plane, the total number of arithmetic operations is reduced to at most 87, which is less than any existing algorithms. Copyright © 2013 John Wiley & Sons, Ltd.

KEYWORDS

triangle-to-triangle intersection; case-depending; partial cross product

***Correspondence**

Ling-yu Wei, Institute for Interdisciplinary Information Sciences, Tsinghua University, Beijing, China.
E-mail: cosimo.dw@gmail.com

1. INTRODUCTION

Collision detection is a fundamental problem in many disciplines, including computer animation, virtual reality, robotics, computer simulations, solid modeling, computational geometry and molecular modeling, etc. Given two objects, in particular-two meshes, the goal is to determine whether they intersect or not.

The naïve approach to solve this kind of problem is to test all primitives (i.e., triangles) of one object against all primitives of the other object, in order to determine their intersection. This approach leads to a huge number of triangle-to-triangle intersection tests.

To reduce the number of tests, many hierarchical data structure have been devised [1,2], and lots of algorithms are derived to enhance the performance in collision detection [3,4]. However, at the bottom of these hierarchies and algorithms, triangle-to-triangle intersection tests must still be performed. Therefore, lots of algorithms have been devised to make the collision test perform more quickly. References [5–9] contains a comparison on triangle intersection tests. This paper presents a new algorithm on the basis of the algorithm by Oren Tropp, *et al*. [8] for faster determination of triangle-to-triangle intersection.

The brute force method simply solves six sets of linear equations to test whether there is an intersection of one triangle's edge with the surface of the other triangle. In [6], Möller's algorithm detects intersection by testing if two segments between one triangle and the planar of the other triangle overlap. And Guigue and Devillers [5] presented an improved algorithm of [6], which evaluates the sign of orientation predicates (4×4 determinants) instead of constructing exact intersection. Different from the aforementioned algorithms, which look at the problem geometrically, Oren Tropp [8] uses an algebraic method by constructing the exact segment between one triangle and the plane of the other triangle and computing the intersection between this segment and the three edges on the plane to test intersection.

This paper presents an algorithm, which absorbs the idea from Oren Tropp reducing the intersection test to the segment and triangle in the same planar and uses the idea of classified discussions by comparing directions of coplanar vectors' cross products. Because this algorithm uses only four comparisons to divide the general detection into 16 different situations, we can optimize each case using different comparisons and reduce the total number of operations.

Firstly, this algorithm constructs the exact segment as the algorithm by Tropp, *et al*. does; then, by evaluating the direction of the cross product between two vectors pointing to the endpoints of the segment and two edges of the triangle on the plane, it can quickly reject some disjoint cases and use specific comparisons for each remaining cases to continue the rest detection.

Depending on different cases, the number of its arithmetic operations varies from 81 to 87, whereas the last step of case-depending detection costs 10–16 operations. This is the least number comparing with 95–97 in Tropp [8], 114–144 in Guigue [5], and 126–144 in Möller [6].† Moreover, this algorithm runs 25.1% faster than the algorithm by Tropp, *et al*., and 13.0% faster than the algorithm by Guigue, *et al*. in our experiment, when detecting 100 000 pairs of randomly created triangles in the unit cube.

The rest of the paper is organized as follows: Firstly, we present the overview of algorithm and then the strategies we used to reduce the total operations and to avoid divisions in the algorithm is introduced. After that, we analyze the operations this algorithm used. Finally, we present the experimental results, and conclusion is made in the end.

2. ALGORITHM OVERVIEW

This algorithm consists of three main stages, and we will elaborate how each stage is performed in the succeeding text:

(1) Same as the algorithm by Oren Tropp, *et al*. [8], we construct the (line) segment between triangle A and the plane of triangle B, and reject the case that no segment exists.
(2) Compute the cross product of two vectors that point to two endpoints from a same vertex of B and the two edges linked to that vertex. And use these cross product's direction to determine the location of the segment. Reject more cases that no intersect can occur.
(3) Computing more cross products from previous calculations to detect whether the segment intersects the triangle.

The second and the third stages are what we improve; whereas the first stage is basically the same as the algorithm by Tropp, *et al*.

2.1. Stage1: segment construction

Similar to the algorithm by Tropp, *et al*., we firstly try to find the segment between the triangle A_0, A_1, A_2 and the plane of the other triangle B_0, B_1, B_2, and reduce the detection between two triangles into intersection detection between a line segment and a triangle on the same planar.

Firstly, after we receive the two triangles described as two sets of points, three of each, we add the same vector $-\overrightarrow{OB_2}$ in order to make the point B_2 move to the origin point. And we name the five vectors starting from B_2 and pointing to the other five origin points (which are equivalent to the new points except the new B_2) as:

$$\boldsymbol{r_i} = A_i - B_2 (i = 0, 1, 2)$$

$$\boldsymbol{e_j} = B_j - B_2 (j = 0, 1)$$

We abuse the notation $\boldsymbol{e}_0$ and $\boldsymbol{e}_1$ on purpose, because we are going to detect the intersection on the planar $(\boldsymbol{O}, \boldsymbol{e}_0, \boldsymbol{e}_1)$, and $\boldsymbol{e}_0$ and $\boldsymbol{e}_1$ are actually the base vectors of this planar.

The segment's endpoints must satisfy the following equation:

$$\alpha_0 \boldsymbol{e}_0 + \alpha_1 \boldsymbol{e}_1 = \beta_{ij} \boldsymbol{r_i} + \beta_{ji} \boldsymbol{r_j} \left(i < j, \beta_{ij} + \beta_{ji} = 1 \right)$$

where (i, j) can be (0, 1), (1, 2), or (0, 2), thus it is actually three equations in the preceeding text.

The left side of this general equation means that the endpoints lay on the plane of triangle B, and the right side means that the endpoint is on the line connecting r_i and r_j.

If we have $0 \le \beta_{ij} \le 1, 0 \le \beta_{ji} \le 1$ for some(i, j), then the endpoint is laid between r_i and r_j, thus the vector $\beta_{ij} \boldsymbol{r_i} + \beta_{ji} \boldsymbol{r_j}$ is actually a vector pointing to an endpoint. But if not, then the endpoint is laid outside the edge of the triangle between r_i and r_j. Therefore, we can find out the two endpoints by checking β_{ij} and β_{ji} (if segment existed).

Figure 1 shows one of the two endpoints, which is on the edge between A_0 and A_2, and it is on the planar spanned by $\boldsymbol{e}_0$ and $\boldsymbol{e}_1$.

Geometrically, the endpoints laid on the plane of triangle B is equivalent to the dot-product of $\beta_{ij} \boldsymbol{r_i} + \beta_{ji} \boldsymbol{r_j}$, and the normal vector of the plane is zero. That is,

$$\left(\beta_{ij} \boldsymbol{r_i} + \beta_{ji} \boldsymbol{r_j} \right) \cdot (\boldsymbol{e}_0 \times \boldsymbol{e}_1) = 0$$

Using this and the equation $\beta_{ij} + \beta_{ji} = 1$, we can solve out β_{ij} and β_{ji} for all (i, j). For simplicity, we compute three constants $D_i = r_i \cdot (\boldsymbol{e}_0 \times \boldsymbol{e}_1), i = 0, 1, 2$, and the solutions of the equation are

$$\beta_{ij} = \frac{D_j}{D_j - D_i}, i \neq j,\ D_i \neq D_j$$

It is trivial to see that $0 \le \beta_{ij} \le 1, 0 \le \beta_{ji} \le 1$ if and only if D_i, D_j have different signs (or one of them is 0). And we will discuss how to avoid divisions later in Section 3.3.

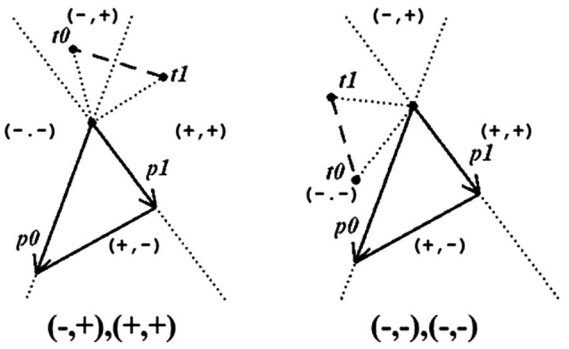

Figure 1. Two examples of separated triangles rejected during Stage 2.

†The numbers are taken from the study of Barequet [1].

DOI: 10.1002/cav

If D_0, D_1, D_2 all have the same sign, then the whole triangle A is on one side of the plane of B, that is, they are disjoint. Also note that if they all have the same sign, D_0, D_1, D_2 may all be equal, meaning that A and B are in parallel planar, thus β_i and β_j may not be solvable if the intersection does not exist.

If D_0, D_1, D_2 are all 0 (or all close to 0), then these two triangles are coplanar. Thus, another coplanar intersection test [6] has to be run to determine whether they intersect. We continue with the not coplanar case in the succeeding text.

If only one or two of D_0, D_1, D_2 are equal to 0 (or close to 0), this is the degenerated case, and as Tropp already pointed out, our algorithm, which is based on construction method, will become less robust. In order to maintain the robustness of our algorithm, we still need to use another intersection test to determine the intersection under such cases.

Despite of the aforementioned special cases, D_0, D_1, D_2 must have different signs, thus we can always construct two endpoints by computing the pair of Ds that have different signs.

In the following discussion, let t_0, t_1 represent the two endpoints, $\boldsymbol{t}_0, \boldsymbol{t}_1$ represent the relevant vectors, and $\boldsymbol{t}$ represent the segment vector (which is equivalent to $\overrightarrow{t_0 t_1}$ or $\boldsymbol{t}_1 - \boldsymbol{t}_0$).

2.2. Stage2: case dividing

Now we compute the four cross product between a vector of each group $\boldsymbol{t}_0, \boldsymbol{t}_1$ and $\boldsymbol{e}_0, \boldsymbol{e}_1$. Because these four vectors are coplanar, their cross products are parallel. By comparing their directions with the vector $\boldsymbol{e}_0 \times \boldsymbol{e}_1$, which we have computed in Stage 1, we can have 16 different outcomes, which are shown in Table I. Hereinafter, we use (+,+) to mean the quarter plane between $\boldsymbol{e}_0$ and $-\boldsymbol{e}_1$, and (−,+) meaning the quarter plane between $-\boldsymbol{e}_0$ and $-\boldsymbol{e}_1$. The sign + and − means whether $\boldsymbol{t}_0 \times \boldsymbol{e}_0$ (and $\boldsymbol{t}_0 \times \boldsymbol{e}_1$) is equidirectional to $\boldsymbol{e}_0 \times \boldsymbol{e}_1$. The four quarter planes are shown in Figure 1, and two examples of 'no intersection' case are shown in Figure 2.

If the location cannot reject the intersection, we have to go to Stage 3 to do more tests.

Table I. Different detection cases depending on location of the segment.

$(\boldsymbol{e}_0 \times \boldsymbol{t}_1, \boldsymbol{e}_1 \times \boldsymbol{t}_1)$ / $(\boldsymbol{e}_0 \times \boldsymbol{t}_0, \boldsymbol{e}_1 \times \boldsymbol{t}_0)$	(+,+)	(+,−)	(−,+)	(−,−)
(+,+)	N	Case 1	N	Case 2
(+,−)	Case 1	Case 3	Case 4	Case 1
(−,+)	N	Case 4	N	N
(−,−)	Case 2	Case 1	N	N

In this table, '+' means the vector is on the same direction and '−' the opposite. 'N' means 'no intersection'.

2.3. Stage3: case-depending detection

Different cases of the location of the segment are shown in Figure 3.

Intersection occurs if and only if the segment intersects the triangle B. More precisely, intersection occurs only when one of the following four situations occurs:

(a) $\boldsymbol{t}$ crosses $\boldsymbol{e}_0$.
(b) $\boldsymbol{t}$ crosses $\boldsymbol{e}_1$.
(c) $\boldsymbol{t}$ crosses $\boldsymbol{e}_1 - \boldsymbol{e}_0$.
(d) $\boldsymbol{t}$ is in triangle B.

However, by dividing the location of the segment into different cases, we only need to test one or two situations for each case.

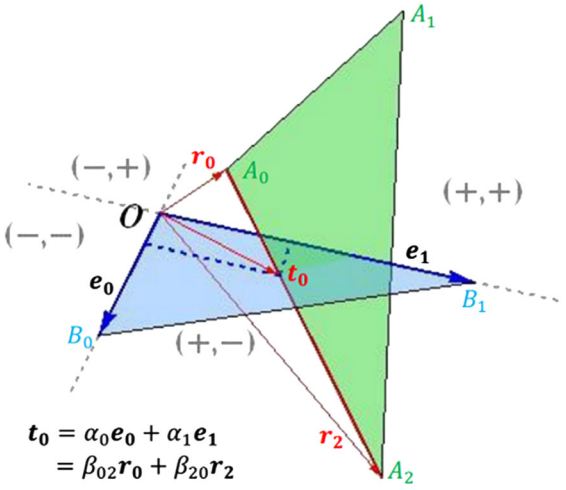

Figure 3. An illustration of the presented variables.

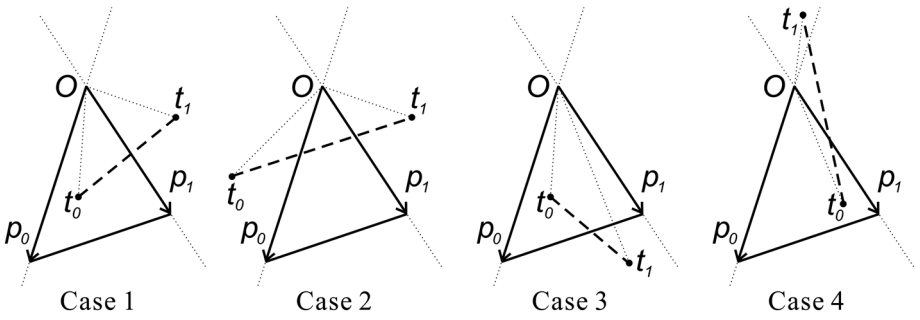

Figure 2. Different cases in Stage 3 detection.

Case 1: Without loss of generality, assume that t_0 is between e_0 and e_1, and t_1 is between $-e_0$ and e_1. The segment t intersects B if and only if ***(c) t crosses*** $e_1 - e_0$ or ***(b) t crosses*** e_1. Note that if t does not cross e_1 but crosses $e_1 - e_0$, t_0 must be in the triangle. Therefore, we can use ***(c′)*** t_0 ***is in the triangle*** to replace **(c)** in order to simplify computation.

From Figure 2, we can transform the occurrence of these two situations to two conditions:

(b) If t crosses e_1, then $e_0 \times e_1$ and $(t_1 - e_1) \times (t_0 - e_1)$ must be equidirectional.
(c′) If t_0 is in the triangle, then $(e_1 - e_0) \times (t_0 - e_0)$ is equidirectional to $e_0 \times e_1$.

If any of these two inequalities is true, then triangles intersect.

Note that because we have already computed cross products such as $e_0 \times t_0, e_0 \times t_1, e_1 \times t_0, e_1 \times t_1, e_0 \times e_1$, we just have to combine them linearly in order to get the cross product we want. This needs very little additional operations.

Case 2: Intersection happens if and only if ***(a) t crosses*** e_0 or ***(b) t crosses*** e_1.

We can change them into three tests:

(1) $t_0 \times t_1$ is equidirectional to $e_0 \times e_1$;
(2) $(t_1 - e_0) \times (t_0 - e_0)$ is equidirectional to $e_0 \times e_1$;
(3) $(t_1 - e_1) \times (t_0 - e_1)$ is equidirectional to $e_0 \times e_1$.

We need only test 2 or 3 to be true, with test 1 being true to ensure intersection. Although 1 and 2 are true, the segment crosses e_0, and although 1 and 3 are true, the segment crosses e_1. Similar to **Case 1**, we have to compute $t_0 \times t_1$ and combine computed cross products linearly.

Case 3: Intersection happens if ***(c) t crosses*** $e_1 - e_0$ or ***(d) t is in triangle B***. However, it can also be optimized because we only need to test if t_0 or t_1 is in the triangle B. When both of them are not in the triangle B, no intersection occurs.

That is, testing whether $e_0 \times e_1$ is equidirectional to $(e_1 - e_0) \times (t_0 - e_0)$ or $(e_1 - e_0) \times (t_1 - e_0)$.

Case 4: The conditions are also ***(a) t crosses*** e_0 or ***(b) t crosses*** e_1.

But in this case, we can compare the direction of $t_0 \times t_1$ first. If $t_0 \times t_1$is equidirectional to $e_0 \times e_1$, then t can only cross e_1. Otherwise, the vector t can only cross e_0. By precomputing t_0 t_1 , we can save one test, compared with **Case 2**.

After this final stage, we can judge whether those triangles intersect.

3. DETAILS IN REDUCING OPERATIONS

This section shows some strategies to reduce the algorithm's operations and its running time. The algorithm can finally be performed in at most 87 operations with no divisions at all.

3.1. Computing only one component of the cross product

This algorithm computes lots of cross products. However, most of the cross products are computed to test whether two vectors are equidirectional or not.

Because all vectors that we need (e_0, e_1, t_0, t_1) are coplanar, all the cross products are parallel to $e_0 \times e_1$, hence we can simply find a nonzero component of $e_0 \times e_1$ and store its sign. Then, we can always compare the relevant component with this sign. Totally, we need two comparisons to find a nonzero component and one comparison to get its sign. Thus, all the remaining cross products needs only three operations, whereas a complete cross product needs nine (each component needs two multiplications and one subtraction).

What is more, we do not have to compute the exact t_0 and t_1, but just two components of them in order to get the relevant component of the cross products, which also decreases our operations a little.

3.2. Comparing values than sign in Stage 2

In **Stage 3**, though we need to compute the linear combinations using the value of cross products in **Stage 2**, we do not need to use all four cross product values in most cases. Therefore, we do not need to compute the exact value in **Stage 2** and compare their value with 0(i.e., use their signs) to go to different branches in **Stage 3**. For example, if $(e_0 \times e_1)$'s z-component is not zero, then we can compare $e_{0x}t_{0y}$ with $e_{0y}t_{0x}$, rather than computing $e_{0x}t_{0y} - e_{0y}t_{0x}$ and compare this with 0, because we may not need the exact value. If we do need this value, we can still compute it in **Stage 3**, only if we have stored these two values before. By avoiding unnecessary subtractions, we can have less arithmetic operations. It is true that because we have to store two values rather than one, we need more spaces and more assignment operations. But nowadays, space of memory is no longer a bottleneck of algorithms, and assignment operations need less time to operate than arithmetic operations.

Note that in case 2 of **Stage 3**, all four exact cross products are needed to make comparisons, which makes it cost 16 operations, and this is the slowest case in **Stage 3**. Therefore, although this strategy does not change the number of the number of operations in the worst case, the second rejection process takes fewer operations so that the average detecting time is still decreased.

3.3. Avoiding divisions

Divisions may take four to eight times of time compared with other operations. Hence, we are going to describe how to avoid divisions in this algorithm.

DOI: 10.1002/cav

In fact, divisions are only used when we compute β_{ij}:

$$\beta_{ij} = \frac{D_j}{D_j - D_i}, i \neq j, D_i \neq D_j$$

Therefore, instead of constructing $\boldsymbol{t}_0$, we construct

$$\boldsymbol{T}_0 = (D_i - D_j)\boldsymbol{t}_0 = D_i\boldsymbol{r_j} - D_j\boldsymbol{r_i}$$

And $\boldsymbol{T}_0$ is equidirectional to $\boldsymbol{t}_0$ if $D_i - D_j > 0$.

Because D_i, D_j have different signs, we can swap i, j to ensure that $D_i - D_j > 0$ is true.

Let $f_0 = D_i - D_j$. Thus, now we get a vector $\boldsymbol{T}_0$ which is equidirectional to $\boldsymbol{t}_0$, with a scalar f_0.

In **Stage 2**, what we need to compute are the directions of the cross products between $\boldsymbol{t}_0, \boldsymbol{t}_1$ and $\boldsymbol{e}_0$, $\boldsymbol{e}_1$. Because $D_i - D_j > 0$, replacing $\boldsymbol{t}_0, \boldsymbol{t}_1$ to $\boldsymbol{T}_0, \boldsymbol{T}_1$ and computing such cross products will not change their directions, that is to say, the first and second rejections will not be affected by avoiding divisions.

In **Stage 3**, we are going to compute the linear combinations of cross products, thus we have to let some clause multiplied by a factor to get right directions. Modifications are easy to find, for example, if we were going to compare the direction of $(\boldsymbol{e}_1 - \boldsymbol{e}_0) \times (\boldsymbol{t}_0 - \boldsymbol{e}_0)$ and $\boldsymbol{e}_0 \times \boldsymbol{e}_1$, or, the linear combination $\boldsymbol{e}_0 \times \boldsymbol{e}_1 + (\boldsymbol{e}_1 - \boldsymbol{e}_0) \times \boldsymbol{t}_0$ and the vector $\boldsymbol{e}_0 \times \boldsymbol{e}_1$.

Because the combination has the same direction with $f_0(\boldsymbol{e}_0 \times \boldsymbol{e}_1) + (\boldsymbol{e}_1 - \boldsymbol{e}_0) \times \boldsymbol{T}_0$, we just need one more multiplication to determine the direction of $\boldsymbol{e}_0 \times \boldsymbol{e}_1 + (\boldsymbol{e}_1 - \boldsymbol{e}_0) \times \boldsymbol{t}_0$.

4. ARITHMETIC OPERATION ANALYSIS

This algorithm needs 81–87 arithmetic operations to ensure intersection, including additions, subtractions, multiplications, and comparisons. No division is used in this algorithm because it may take four to eight times of time comparing with other operations. This is compared with 95–97 in the method by Tropp, *et al.* [8], 114–144 in Guigue, *et al.* [5], and 126–148 in Möller's [6] no-division version of his algorithm (including absolute value operations).

In Table II, we show all the operations that this algorithm takes in detail, and Table III is taken from the paper by Tropp, *et al.* [8], representing the count of operations of other algorithms.

Generally, three comparisons are enough in **Stage 1**, but we add three more comparisons which are used only when degenerate case occurs. Luckily, spending more operations here leads to less after, because when this happens, we do not need to construct $\boldsymbol{t}_0, \boldsymbol{t}_1$ because they must be equal to one or two of $\boldsymbol{r}_0, \boldsymbol{r}_1, \boldsymbol{r}_2$. Totally, 45 or 56 operations are needed to finish **Stage 1**.

In **Stage 2**, the second rejection costs 15 more operations, which means that most rejections would take no more than 71 steps (when degenerate case occurs it becomes 60). Specifically, three comparisons are needed to find out the sign of the nonzero component of $\boldsymbol{e}_0 \times \boldsymbol{e}_1$ and four to compare directions of those cross products.

Also note that this algorithm uses more assignment operations than the algorithm by Tropp *et al.*, which can reduce the number of multiplications in **Stage 3**.

5. EXPERIMENTAL RESULTS

The experiment is run on a 2.2GHz Intel Core 2 Duo processor with Windows 7. MSVC complier and -O2 optimization is chosen. We use five different algorithms, each to detect the same 1 000 000 random triangles, 1 000 000 randomly created intersected triangles, and 1 000 000 randomly created separated triangles. When repeat running the same program for a large amount of time, the time spent onasking for opening memory space cannot be ignored and may affect the running time a lot. Thus, all variables in all algorithms are set as static. Results are shown in Table IV and V.

As seen in the succeeding text, this algorithm takes 0.0785 μs for each set on average, whereas the algorithm by Tropp, *et al.* takes 0.104 μs on average. Thus, this algorithm runs 25.1% faster than its origin algorithm.

From the table, we can also see that the previous fastest algorithm is the algorithm by Guigue, *et al.* [5]. Note that although the algorithm by Tropp, *et al.* runs slower than the algorithm by Guigue, *et al.*, this algorithm still runs about 13% faster than the algorithm by Guigue, *et al.*

Table II. Arithmetic operations in detail.

Operation		+/−	MUL	CMP	ALL
Stage 1	D_0, D_1, D_2 and comparison	24	15	3–6	42–45
	Constructing $\boldsymbol{t}_0, \boldsymbol{t}_1$ (avoid divisions)	6	8	—	14
Stage 2	Computing cross product and comparison	—	8	7	15
Stage 3	Case 1	6	5	2	13
	Case 2	7	6	3	16
	Case 3	6	2	2	10
	Case 4	4	4	2	10
Total (regardless of degeneration)		34–37	33–37	12–13	81–87

Table III. Comparison of arithmetic operations.

Algorithm	+/−	MUL	CMP	DIV	ABS	=
Möller [2]	54	57	12/28	—	3/9	69/75
Guigue, *et al.* [1]	62/76	43/52	9/16	—	—	42/62
Tropp, *et al.* [3]	26/27	56/57	13	—	—	31/35
Mine	34/37	33/37	12/13	—	—	38/39

Table IV. Average testing time.

Algorithm	Random case	Intersected case	Separated case
Tropp, *et al.* [8]	0.1168 μs	0.1314 μs	0.0997 μs
Shen, *et al.* [7]	0.1231 μs	0.1613 μs	0.0934 μs
Möller's [6]	0.1160 μs	0.1557 μs	0.0939 μs
Guigue, *et al.* [5]	0.1005 μs	0.1225 μs	0.0869 μs
My method	0.0875 μs	0.1011 μs	0.0779 μs

Table V. Accelerating percentage (compared with this algorithm).

Algorithm	Random case	Intersected case	Separated case
Tropp, *et al.* [8]	25.13%	23.08%	21.89%
Shen, *et al.* [7]	28.95%	37.34%	16.66%
Möller's [6]	24.61%	35.08%	17.05%
Guigue, *et al.* [5]	13.00%	17.50%	10.43%

Table VI. Average testing time without code optimization.

Algorithm	Random case	Intersected case	Separated case
Tropp, *et al.* [8]	0.1587 μs	0.1754 μs	0.1361 μs
Shen, *et al.* [7]	0.1780 μs	0.2441 μs	0.1333 μs
Möller's [6]	0.1862 μs	0.2651 μs	0.1436 μs
Guigue, *et al.* [5]	0.1808 μs	0.2712 μs	0.1319 μs
My method	0.1273 μs	0.1453 μs	0.1078 μs

However, as mentioned on the paper by Tropp, *et al.* [8], the algorithm by Tropp, *et al.* runs faster than the algorithm by Guigue, *et al.*, and they claimed that their algorithm was the fastest. Such results may be caused by code optimization settings, and Table VI shows the running time of each algorithm using another compiler without any code optimization option. The algorithm by Tropp, *et al.* gets better performance in that table, whereas our method still gets a much better result than all others.

Because our algorithm has the minimum operations, in the intersected case, its advantage is shown obviously. Because of the two fast rejection processes that this algorithm has, its running time in separated case is also 10–20% faster than other algorithms.

6. CONCLUSION

This paper describes a faster algorithm to detect whether two triangles embedded in three dimensions intersect. Its basic idea is to test whether the line segment between one triangle and the plane of the other triangle intersects that triangle by discussing where the segment locates. Then, different test conditions are used so that its detection time can be reduced. Also, this algorithm outperforms all previous ones because of more inter states stored for following steps, and less operations invoked for direction testing, compared to the whole computation for cross products. However, the biggest advantage is its idea to divide the general detection into different cases.

Compared to the algorithm by Tropp, *et al.*, which had the smallest number of operations, this algorithm is 10% less than it and, because it has two rejection tests before the final detection is made, whereas the first rejection test is the same as the algorithm by Tropp, *et al.*, it can run much faster than the algorithm by Tropp, *et al.* in separated case.

This algorithm runs 25.1% faster than the algorithm by Tropp, *et al.* and 13.0% faster than the algorithm by Guigue, *et al.* during random intersection tests (on Core 2 Duo 2.2GHz, Windows 7). It outperforms other algorithms both in intersection case and separation case, but has a larger advantage in intersection case.

This algorithm, however, has its drawback because it cannot compute the exact intersection, which may be important in certain situations. If exact intersection is needed, then other algorithms may be more convenient to be applied, such as the algorithms by Tropp, *et al.* and Möller, whose running time are very close to each other, as shown in Table III.

Another drawback of this algorithm is that it uses 55 variables, which is much bigger compared with 18 in the algorithm by Guigue, *et al.*, whereas its code's length is also longer than others. Though nowadays memory space is no longer a bottleneck that affects programs very much, it is still a considerable problem in some situations like computation on portable devices. Because the algorithm by Guigue, *et al.* uses only 1/3 space of this algorithm, it is still very useful in many situations.

ACKNOWLEDGEMENTS

This work was supported in part by the National Basic Research Program of China Grant 2007CB807900, 2007CB807901 and the National Natural Science Foundation of China Grant 61073174, 61033001, 61061130540.

REFERENCES

1. Barequet G, Chazelle B, Guibas LJ, Mitchell JSB, Tal A. BOXTREE: a hierarchical representation for surfaces in 3D. *Computer Graphics Forum* 1996; **15**: 387–396. DOI: 10.1111/1467-8659.1530387

2. Gottschalk S, Lin M, Manocha D. OBB-tree: a hierarchical structure for rapid interference detection. *ACM SIGGRAPH* 1996; 171–180.
3. Eberly D. Intersection of convex objects: the method of separating axes. 2007. [Online]. Available: http://www.geometrictools.com.
4. Lin MC, Manocha D, Ponamgi MK, Cohen JD. I-COLLIDE: an interactive and exact collision detection. in *Proc. ACM Int. 3D Graphics Conf.*, 1995.
5. Guigue P, Devillers O. Fast and robust triangle-triangle overlap test using orientation predicates. *Journal of Graphics Tools* 2003; **8**(1): 25–42.
6. Möller T. A fast triangle-triangle intersection test. *Journal of Graphic Tools* 1997; **2**(2): 25–30.
7. Shen H, Tang Z, Heng PA. A fast triangle-triangle overlap test using signed distances. *Journal of Graphic Tools* 2003; **8**(1): 3–15.
8. Tropp O, Tal A, Shimshoni I. A fast triangle to triangle intersection test for collision detection. *Computer Animation and Virtual Worlds* 2006; **17**: 527–535.
9. Raabe A, *Describing and Simulating Dynamic Reconfiguration in SystemC Exemplified by a Dedicated 3D Collision Detection Hardware*, PhD dissertation: Bonn, Germany, 2008.

Author biography:

Ling-yu Wei is an undergraduate student in the Institute for Interdisciplinary Information Sciences at Tsinghua University since 2010. His research interests include digital image processing and computer graphics, mainly about optimization, model reconstruction, and real-time rendering.