Yet Faster Ray-Triangle Intersection (Using SSE4)

Jiří Havel and Adam Herout

***Abstract*—Ray-triangle intersection is an important algorithm, not only in the field of realistic rendering (based on ray tracing), but also in physics simulation, collision detection, modelling, etc. Obviously, the speed of this well-defined algorithm's implementations is important because calls to such a routine are numerous in rendering and simulation applications.**

Contemporary fast intersection algorithms, which use SIMD instructions, focus on the intersection of ray packets against triangles. For intersection between single rays and triangles, operations such as horizontal addition or dot product are required. The SSE4 instruction set adds the dot product instruction which can be used for this purpose.

This article presents a new modification of the fast ray-triangle intersection algorithms commonly used, which – when implemented on SSE4 – outperforms the current state-of-the-art algorithms. It also allows both a single ray and ray packet intersection calculation with the same precomputed data. The speed gain measurements are described and discussed in the article.

***Index Terms*—Raytracing, Geometric algorithms.**

I. Introduction

RAY-TRIANGLE intersection is a common algorithm in computer graphics. Intersection of rays with various objects is a basic operation in realistic rendering, from ray-tracing to more complex methods. It is used in physical simulation, collision detection, scene and object modelling and various other tasks, including tasks which are performed in real-time, and therefore are time-critical.

SIMD instruction sets like SSE are – in the context of ray-triangle intersection – mostly used for tracing coherent ray packets [1]. The coherence of the rays within the packet is then very important, since the vector instructions are fully used only if all rays go through the same branch of computation.

In situations like physical simulation, collision detection or raytracing in scenes, where rays bounce into multiple directions (spherical or bumpmapped surfaces), coherent ray packets break down very quickly to single rays or do not exist at all. In the above mentioned tasks, packet oriented SIMD computations is much less useful. These situations would benefit from SIMD implementation of single ray intersection, even if the overall performance was worse compared to coherent ray packets.

A. Ray-Triangle Intersection Basics

The intersection of a given ray with the triangle's plane is generally calculated by solving the system of equations

$$O + t \cdot \vec{D} = A + u \cdot (B - A) + v \cdot (C - A) \qquad (1)$$

The autors are with Department of Graphics and Multimedia, Faculty of Information Technology, Brno University of Technology, Božetěchova 2, 612 66, Brno, Czech Republic.

E-mail: {ihavel, herout}@fit.vutbr.cz

Point O and direction vector $\vec{D}$ define the ray and points A, B, C are the triangle's vertices. Note that the triangle sides can also be expresed as vectors $\vec{AB}$ and $\vec{AC}$, instead of subtraction of vertices. The solution of this system of equations are a ray parameter (t) and barycentric coordinates (u and v) of the intersection [2].

The right side of the equation (1) is a slight modification of a common expression of a point P on triangle ABC (using the barycentric coordinates u and v)

$$P = (1 - u - v) \cdot A + u \cdot B + v \cdot C \qquad (2)$$

To determine whether the triangle plane was actually intersected inside the triangle, parameters t, u and v must meet the conditions

$$\begin{gathered} 0 \leq t \leq t_{max} \\ u \geq 0 \\ v \geq 0 \\ (u + v) \leq 1 \end{gathered} \qquad (3)$$

The first condition limits the ray to a segment, which starts at point O and is $t_{max}|\vec{D}|$ long. The rest of the conditions limit the intersection to be within the triangle.

The system of equations (1) can be solved either directly using the Cramer rule or indirectly by substitution. The first approach is used by Möller-Trumbore [3] and Kensler-Shirley [4] methods. The second approach is used by methods such as Badouel [2] and Wald [1]. Very similar is also the method by Shevtsov et al. [5], although it was derived using the Plücker coordinates. All of the last three mentioned methods benefit from projecting the triangle and the ray onto one of the coordinate planes, which reduces the number of operations required for the intersection calculation, but on the other hand, they require the information about the selected coordinate plane to be stored together with the triangle data.

The input data can be either raw triangle vertices (A, B, C in the above equations) or some precomputed values, which further simplify the intersection calculation, but require aditional computations in the preparatory phase. The normal vector or all four coefficients of the triangle plane equation are frequent variants of the precomputed values, since they can be used not only for the intersection calculation, but also for shading, space subdivision, etc.

This computation does not cause significant delay, as boundary volume hierarchies are often built in preparatory phases. BVH's building complexity is typically linearitmic, in contrast to linear complexity of precomputation. More problematic can be the memory footprint of these precomputed data. Most triangle meshes can be stored with much less memory consumption using indexed geometry. Unfortunately, when precomputed data related to triangles is involved, indexed geometry cannot be used because precomputed values depend

on all three vertices in most cases and generally cannot be shared between triangles.

II. Existing Solutions

The ray-triangle intersection method presented in this article is based on principles from Wald's [1] and Shevtsov's [5] methods. Both methods use precomputed values as the input, and they both project the triangles and rays onto one coordinate plane. The selected coordinate axes will be denoted as p and q, the discarded axis as r. In other papers like [1], [5], letters u, v and w are used. These letters were selected to avoid naming collision with barycentric coordinates. The axis with the largest projection of the triangle's normal vector is discarded. This equals to selection of a plane with the biggest triangle projection. Intersection of the ray with the triangle plane is done by

$$\begin{aligned} t &= -\frac{\vec{N}\cdot O + d}{\vec{N}\cdot\vec{D}} \\ \vec{N} &= \vec{AB}\times\vec{AC} \\ d &= -A\cdot\vec{N} \end{aligned} \tag{4}$$

The normal vector $\vec{N}$ and the value d specify the triangle plane, $xN_x+yN_y+zN_z+d=0$ is the plane's equation. Since this equation is independent of the normal vector's length, the normal vector can be scaled so one of its components becomes 1. That component can then be renamed as r and the other two as p and q. With $d = A\cdot\vec{N}$, the ray parameter can be expressed as

$$t = \frac{d-(N_pO_p+N_qO_q+O_r)}{N_pD_p+N_qD_q+D_r}. \tag{5}$$

This modification, however, prevents easy culling of backfacing polygons by testing the sign of the denominator; to allow such backface culling, the N_r sign would need to be stored.

A. Wald's Method

Wald's method [1] calculates the ray parameter t exactly as in the equation (5). The barycentric coordinates p and q are calculated from six precomputed values (note that for example AB_p stands for the p component of the $\vec{AB}$ vector)

$$\begin{aligned} \mathrm{denom} &= AB_p\cdot AC_q - AC_p\cdot AB_q \\ U_p &= AC_q/\mathrm{denom} \\ U_q &= -AC_p/\mathrm{denom} \\ U_d &= \frac{A_q\cdot AC_p - A_p\cdot AC_q}{\mathrm{denom}} \\ V_p &= -AB_q/\mathrm{denom} \\ V_q &= AB_p/\mathrm{denom} \\ V_d &= \frac{A_p\cdot AB_q - A_q\cdot AB_p}{\mathrm{denom}} \end{aligned} \tag{6}$$

and from the ray parameter t as

$$\begin{aligned} P &= O + t\cdot\vec{D} \\ u &= P_pU_p + P_qU_q + U_d \\ v &= P_pV_p + P_qV_q + V_d \end{aligned} \tag{7}$$

The values from equation (6) (U_p, U_q, U_d, V_p, V_q and V_d) and the triangle plane ($\vec{N}$ and d) are precalculated and stored instead of the triangle vertices. Since the values of t, u and v are tested as shown in equation (3), a costly division must be calculated for each test to evaluate t, even if the intersection is rejected.

B. Shevtsov's method

Shevtsov's method [5] uses the triangle plane (N_p, N_q, d), one triangle vertex (A_p, A_q) and scaled edges

$$\begin{aligned} E0_p &= (-1)^r\frac{AB_p}{N_r}, \quad E0_q = (-1)^r\frac{AB_q}{N_r} \\ E1_p &= (-1)^r\frac{AC_p}{N_r}, \quad E1_q = (-1)^r\frac{AC_q}{N_r} \end{aligned} \tag{8}$$

as precomputed values. The intersection is calculated as

$$\begin{aligned} \det &= D_pN_p + D_qN_q + D_r \\ t' &= d - (O_pN_p + O_qN_q + O_r) \\ T_p &= t'D_p - \det\cdot(A_p - O_p) \\ T_q &= t'D_q - \det\cdot(A_q - O_q) \\ u' &= E1_qT_p - E1_pT_q \\ v' &= E0_pT_q - E0_qT_p \\ \begin{bmatrix} t \\ u \\ v \end{bmatrix} &= \frac{1}{\det}\cdot\begin{bmatrix} t' \\ u' \\ v' \end{bmatrix} \end{aligned} \tag{9}$$

The main difference from Wald's method is that the division is performed at the end of the computation. The conditions

$$\begin{aligned} \mathrm{sign}(t') &= \mathrm{sign}(\det\cdot t_{max} - t') \\ \mathrm{sign}(u') &= \mathrm{sign}(\det - u') \\ \mathrm{sign}(v') &= \mathrm{sign}(\det - u' - v') \end{aligned} \tag{10}$$

are equal to the conditions from equation (3), only they can be performed before dividing by $\det$.

III. The New Ray-Triangle Intersection Algorithm

This modification combines Wald's approach (modified by avoiding the projection onto one coordinate plane) with the idea of deferring the division from Shevtsov's method. The triangle is described by the precomputed triangle plane ($\vec{N}$, d) and two other planes, ($\vec{N}_1$, d_1) and ($\vec{N}_2$, d_2)

$$\begin{aligned} \vec{N}_1 &= \frac{\vec{AC}\times\vec{N}}{|\vec{N}|^2}, \quad d_1 = -\vec{N}_1\cdot A \\ \vec{N}_2 &= \frac{\vec{N}\times\vec{AB}}{|\vec{N}|^2}, \quad d_2 = -\vec{N}_2\cdot A \end{aligned} \tag{11}$$

The barycentric coordinates of the intersection point are

$$\begin{aligned} P &= O + t\vec{D} \\ u &= \vec{N}_1\cdot P + d_1 \\ v &= \vec{N}_2\cdot P + d_2 \end{aligned} \tag{12}$$

The barycentric coordinates are expressed as scaled distance from their respective planes. For the u coordinate, the side AC lies on such a plane and the plane's normal vector $\vec{N}_1$ is scaled so that $\vec{N}_1\cdot B + d_1 = 1$. For the v coordinate, the plane is constructed similarly (side AB and vertex C). For every triangle, there exists an infinite number of these planes, but only the planes perpendicular to the triangle plane or planes

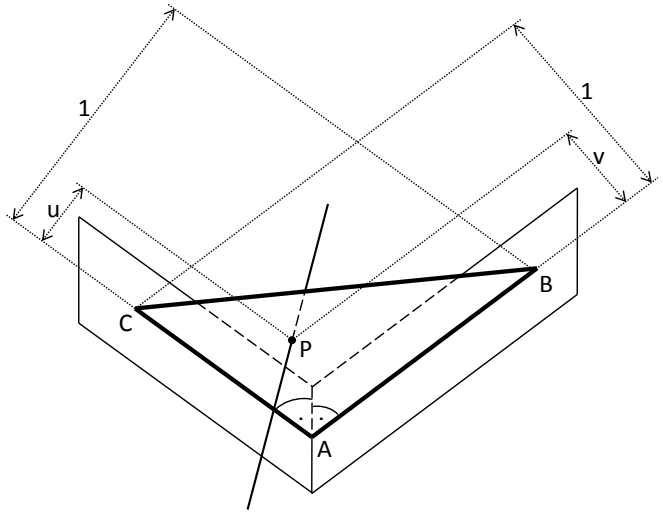

Fig. 1. Perpendicular planes for barycentric coordinate calculation. Planes are perpendicular to the triangle, not to each other.

parallel to one coordinate axis are of practical interest. Figure 1 shows the situation with the perpendicular planes, which are calculated according to equation (11). Note that the triangle normal vector $\vec{N}$ must be $\vec{AB} \times \vec{AC}$ as the denominator $|\vec{N}|^2$ comes from the substitution of the proper point into the barycentric coordinates equations (12).

The intersection is calculated as follows in equation (13). Note that the division can be deferred after testing t', u' and v' by conditions identical to Shevtsov's method in equation (10).

$$\begin{aligned} \det &= \vec{D} \cdot \vec{N} \\ t' &= d - (O \cdot \vec{N}) \\ P' &= \det \cdot O + t' \cdot \vec{D} \\ u' &= P' \cdot \vec{N}_1 + \det \cdot d_1 \\ v' &= P' \cdot \vec{N}_2 + \det \cdot d_2 \\ \begin{bmatrix} t \\ u \\ v \end{bmatrix} &= \frac{1}{\det} \cdot \begin{bmatrix} t' \\ u' \\ v' \end{bmatrix} \end{aligned} \tag{13}$$

This algorithm uses three more floating point values for each triangle than Wald's and Shevtsov's method. However, the amount of memory used by efficient practical implementations is the same. For efficient cache usage, triangle data have to be aligned on 16B to better fit to cache lines [1]. Both Wald's and Shevtsov's method can be implemented without this alignment, but with speed penalty due to worse caching. When memory consumption is a problem, methods like Möller-Trumbore [3] and Kensler-Shirley [4] are more suitable.

IV. SSE Implementation

The most demanding part of the computation in the new method presented are the four dot products. These can be easily calculated by the `DPPS` instruction from the SSE4.1 instruction set. The code calculating the intersection, along with the data structures, is shown below.

```
struct Ray
{
  float ox, oy, oz, ow;
  float dx, dy, dz, dw;
};
struct PrecomputedTriangle
{
  float nx, ny, nz, nd;
  float ux, uy, uz, ud;
  float vx, vy, vz, vd;
};
struct Hit
{
  float px, py, pz, pw;
  float t, u, v;
};
const float int_coef_arr[4] = { -1, -1, -1, 1 };
const __m128 int_coef = _mm_load_ps(helper);
bool Intersect(const Ray &r,
               const PrecomputedTriangle &p, Hit &h)
{
  const __m128 o = _mm_load_ps(&r.ox);
  const __m128 d = _mm_load_ps(&r.dx);
  const __m128 n = _mm_load_ps(&p.nx);

  const __m128 det = _mm_dp_ps(n, d, 0x7f);
  const __m128 dett = _mm_dp_ps(
    _mm_mul_ps(int_coef, n), o, 0xff);
  const __m128 oldt = _mm_load_ss(&h.t);

  if((_mm_movemask_ps(_mm_xor_ps(dett,
    _mm_sub_ss(_mm_mul_ss(oldt, det), dett)))&1) == 0)
  {
    const __m128 detp = _mm_add_ps(_mm_mul_ps(o, det),
      _mm_mul_ps(dett, d));
    const __m128 detu = _mm_dp_ps(detp,
      _mm_load_ps(&p.ux), 0xf1);

    if((_mm_movemask_ps(_mm_xor_ps(detu,
      _mm_sub_ss(det, detu)))&1) == 0)
    {
      const __m128 detv = _mm_dp_ps(pt,
        _mm_load_ps(&p.vx), 0xf1));

      if((_mm_movemask_ps(_mm_xor_ps(detv,
        _mm_sub_ss(det, _mm_add_ss(detu, detv))))&1) == 0)
      {
        const __m128 inv_det = inv_ss(det);
        _mm_store_ss(&h.t, _mm_mul_ss(dett, inv_det));
        _mm_store_ss(&h.u, _mm_mul_ss(detu, inv_det));
        _mm_store_ss(&h.v, _mm_mul_ss(detv, inv_det));
        _mm_store_ps(&h.px, _mm_mul_ps(detp,
          _mm_shuffle_ps(inv_det, inv_det, 0)));
        return true;
      }
    }
  }
  return false;
}
```

The `Ray` structure consists of the ray origin and direction, the `PrecomputedTriangle` structure contains the triangle plane and two planes for the barycentric coordinates. The structure `Hit` contains the intersection point, the ray parameter and the barycentric coordinates. The ray parameter from the `Hit` structure is also used as an input value t_{max}, supposing the value is filled by the previous call to the `Intersect` function. The compiler-dependent structure attributes (`#pragma`, `__declspec` or `__attribute__`) for 16B alignment are omitted from the code listing for simplicity.

The first `if` statement in the code performs the ray parameter test (see equation (10)); the next two `if`'s test the `u` and `v` coordinates similarly. If all the tests pass, the values `t`, `u`, `v` and the intersection point are calculated. The value of `det` is inverted by the `RCPI` instruction, which is followed by one step of the Newton-Raphson iteration method [6].

V. Experimental Evaluation

The experimental evalution was made as similar as possible with the Kensler-Shirley paper [4]. Namely the test data was generated randomly by the same code as in [4], and the evaluation methodology stood on the same principles. The new method, along with the two state-of-the-art methods from section II, were implemented both for single rays and for four ray packets. To evaluate the effect of the deferred division, all methods were implemented with and without deferred division. Since the implementation without the deferred division performed worse in all test cases, it was excluded from the evaluation results. The program was compiled by Microsoft Visual C++ 2008 and Intel C++ compiler 11.0.066 and run on Core 2 Duo E8200 with 2 GB RAM, although only a single core was used. The results are always an average of three runs (separate runs differ only within 1%, so the measured values are not influenced by any asynchronous events), the performance is measured in millions of (ray-triangle) tests per second. The precision was evaluated by evaluating the mean square relative error from the double precision Kensler-Shirley implementation [4].

The worst case scenario tests were done by effectively disabling the early termination using the line parameter t. Such a situation is similar to deeply subdivided scenes, where overlapping of triangles is prevented by space partitioning and an appropriate traversal [1].

The performance of precomputation was measured for many passes over triangle data. Single pass was too fast to provide precise measurement. Again, the difference between runs was under 1%.

A. Evaluation Results

Table I summarizes the experimentally measured results. The performance of the new method for single rays is roughly

Test	Wald	Shevtsov	New	New SSE4
	Microsoft Visual C++			
Single rays	109	107	115	149
Single worst	44	46	44	48
Ray packets	390	391	427	
Packets worst	173	179	177	
	Intel C Compiler			
Single rays	106	106	119	136
Single worst	45	46	44	48
Ray packets	393	376	400	
Packets worst	178	185	178	

TABLE I
Experimental results for Microsoft C Compiler and Intel C Compiler, for the state-of-the-art methods and the presented new method. Values are in millions of tests per second.

the same in the worst case scenario and significantly better in situations with many overlapping triangles. This is due to the lack of coordinate plane selection, which is done at the start of the computation in the alternative approaches. Thanks to that, a large part of the computation can be be skipped by the early termination based on the line parameter t. On the other hand, the same reason causes a greater performance drop in the worst case scenario compared to the other methods: the less computation load performed for each test is exchanged for more computation in the worst case scenario, because the computation takes place in 3D instead of 2D, as in the case of the alternative approaches.

Table II summarizes the results for triangle precomputation. The new method precomputed data roughly 10% to 25% faster.

Compiler	Wald	Shevtsov	New	New SSE4
MSVC	26	32.1	35.5	50.3
ICC	31.9	33.5	42	52.2

TABLE II
Experimental results for triangle precomputation. Values are in millions of triangles processed per second.

This is probably because of branching required in precomputation in Wald's and Shevtsov's methods. The precomputation could be further accelerated by SSE at the cost of 16B vertices alignment.

All the evaluated methods have roughly the same mean square relative error of the intersection's coordinates – around 10^{-8}. Shevtsov's method seems to be the most accurate and Wald's method the least, but the differences are so subtle that all the methods should be perceived as equally accurate. No significant flaws such as missed triangles or false detections were observed with any of the methods.

VI. Conclusion

This article presents a novel modification of the commonly used approaches to the problem of ray-triangle intersection. The existing methods were summarized and a new modification was described; it uses more pre-calculated values (while not extending the memory requirements compared to the "standard" approaches due to memory alignment), and benefits from better performance on current CPU's. Note that the presented method performs as well or better than the known methods, both in the plain C implementation and using the current version of the SSE instruction set; it is not a mere SSE4 implementation of the problem but an improvement of the ray-triangle intersection state-of-the-art.

The method outperforms the known solutions, especially in solving the intersection for a single ray and in scenes without deep space subdivision. For ray packets, this method can be useful in situations where packets often break into single rays. Contrary to the previous methods, the new method can use SIMD instructions even for single rays, though the speed-up is then only between 10 to 25 per cent (compared to roughly 300 per cent in the case of coherent ray packets). More importantly, the method uses identical triangle representation for both single rays and ray packets.

Acknowledgements

This work was supported by the Ministry of Education, Youth and Sports of the Czech Republic under the research program LC-06008 (Center for Computer Graphics) and by the research project "Security-Oriented Research in Information Technology" CEZMSMT, MSM0021630528.

REFERENCES

[1] I. Wald, "Realtime Ray Tracing and Interactive Global Illumination," Ph.D. dissertation, Saarland University, 2004.

[2] D. Badouel, "An Efficient Ray-Polygon Intersection," in *Graphics Gems*, S. A. Glassner, Ed. San Diego, CA, USA: Academic Press Professional, Inc., 1998, pp. 390–393.

[3] T. Möller and B. Trumbore, "Fast, Minimum Storage Ray-Triangle Intersection," *Journal of Graphics Tools*, vol. 2, no. 1, pp. 21–28, 1997.

[4] A. Kensler and P. Shirley, "Optimizing Ray-Triangle Intersection via Automated Search," in *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, Sept 2006, pp. 33–38.

[5] M. Shevtsov, A. Soupikov, and A. Kapustin, "Ray-Triangle Intersection Algorithm for Modern CPU Architectures," in *Proceedings of GraphiCon 2007*, 2007, pp. 33–39.

[6] *Increasing the Accuracy of the Results from the Reciprocal and Reciprocal Square Root Instructions using the Newton-Raphson Method*, Intel Corporation, 1999. [Online]. Available: http://cache-www.intel.com/cd/00/00/04/10/41007_nrmethod.pdf

Jiří Havel received the MS degree at Faculty of Information Technology, Brno University of Technology, Czech Republic. He is currently a PhD student at Department of Computer Graphics and Multimedia at FIT BUT. His research interests include computer graphics and functional programming.

Adam Herout received his PhD from Faculty of Information Technology, Brno University of Technology, Czech Republic, where he works as an assistant professor and leads the Graph@FIT research group. His research interests include fast algorithms and hardware acceleration in computer vision and graphics.