Ray-Triangle Intersection Algorithm for Modern CPU Architectures

Maxim Shevtsov, Alexei Soupikov and Alexander Kapustin
Intel Corporation
Nizhniy Novgorod, Russia
{maxim.y.shevtsov, alexey.soupikov, alexander.kapustin}@intel.com

Abstract

We present an algorithm for determining if a ray intersects a triangle interior; and computing intersection point parameters as well as distance of intersection in response to the ray intersecting a triangle interior. Particularly a variation of a hybrid test having all benefits of Plücker and projected barycentric tests is proposed. The test is also vectorized using SIMD instructions for efficient handling ray packets. It is essential for achieving high ray tracing performance on modern CPUs.

Our implementation also detects axis-orthogonal triangles and processing them separately.

For maximum performance we also introduce a method for triangle representation, using only necessary pre-computed values.

We also present inherently thread-safe and memory efficient alternative of mailboxing to avoid unnecessary intersection tests for ray packet in case when many leaves share the same triangle.

Keywords: *ray-triangle intersection, SIMD, Plücker test.*

1. INTRODUCTION

A ray tracing is a well known method used in modeling of a variety of physical phenomena related to light propagation in various media [1, 2]. Although ray tracing is computationally demanding, operations and data access costs can be efficiently amortized over rays bundled in a packet [3, 4]. This allows for reducing the required memory bandwidth, which is known to be one of the major bottlenecks of current CPU architectures.

The ray-tracing algorithm basically consists of the following operations:

- traversing of the scene in a front-to-back manner until a leaf is reached;
- test all entries in the leaf's primitive reference list (typically, indices referring to a list of scene primitives) and retain the nearest intersection;

Researchers proposed algorithms for tracing coherent ray packets instead of single rays [3, 5] using SIMD instructions. Thus a fast ray-triangle intersection test which can be also efficiently implemented using SSE instructions is also the key factor for increasing performance especially since ray-triangle intersection test is the one of the most frequently performed.

In this paper a ray $\boldsymbol{r}(t)$ with origin $\boldsymbol{o}$ and normalized direction $\boldsymbol{d}$ is defined as $\boldsymbol{r}(t) = \boldsymbol{o} + t*\boldsymbol{d}$, while a triangle is defined as $(\boldsymbol{p}, \boldsymbol{e0}, \boldsymbol{e1})$ – by vertex and 2 edges.

The intersection (hit) point can be described by u, v barycentric coordinates to allow representation of the point as a linear combination of triangle vertex and edges:

$\mathbf{p_h} = \mathbf{p} + \mathbf{e0}*u + \mathbf{e1}*v$;

In the ray-triangle intersection problem we want to determine if the ray intersects the triangle and to compute hit point parameters, namely u, v as well as value of t - distance of intersection (see Figure 1). In addition we must avoid numerical errors which can result in visible artifacts. Barycentric coordinates u, v are typically used in ray-tracing for further processing (for example, computation of texture coordinates, normal interpolation and so on).

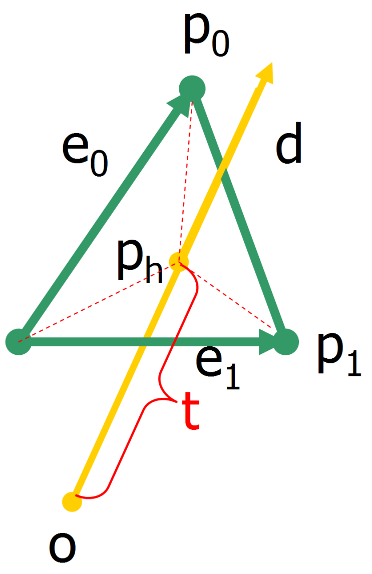


Figure 1: Ray-Triangle Intersection Algorithm finds whether intersection point $\mathbf{p_h}$ exists. If yes, computes the u,v parameters of intersection (such as $\mathbf{p_h} = \mathbf{p} + \mathbf{e0}*\mathbf{u} + \mathbf{e1}*\mathbf{v}$) as well as distance t={o, $\mathbf{p_h}$}.

A triangle may overlap many leaves and it leads to the same primitive being tested multiple times during traversal of a ray packet. These multiple intersections can be avoided by mailboxing technique [1, 6, 7]. We introduce very compact, fast and thread-safe alternative of mailboxing that doesn't require any additional per-primitive data or large look-up tables.

2. PREVIOUS WORK

Initially algorithms solved ray-triangle intersection problem simply by computing the intersection with three boundary planes defining the extent of the triangle and then testing if the intersection point is inside the edges. This approach requires significant memory for the storing planes.

Another approach is to use some parametric representation like [6, 8]. The basic idea behind this test is to project the triangle onto one of the three axis-aligned planes (along so called dominant axis selected according triangle plane orientation) and to perform the computation of the barycentric coordinates in 2D instead of 3D. As the distance computation involves a costly division and the subsequent instructions depend on the result, the Wald's implementation [6] uses Newton-Raphson iteration for computing the inverse resulting in observable but moderate speed up.

Ray-triangle intersection test should be considered in the context of ray tracing where ray usually traverses some acceleration structure and is tested against some number of triangles met during traversal step. The algorithm searches for the closest intersection. So ray-triangle intersection test consists of two logical steps:

a) computing distance from ray origin to the point where the ray intersects the triangle plane and testing if it is the closest intersection and distance is greater or equal to zero (distance test),

b) testing if that ray-plane intersection point lies inside the triangle (aperture test).

Usual strategy in the most of algorithms published is performing distance test first and do so called early exit if distance test is not passed thus skipping an aperture test. The fact that it is not necessarily the best performing strategy stayed unnoticed for a long time. The statistics presented in [5] shows that at least in case of spatial sub-division acceleration structures the distance test passes far more often than aperture test. So performing early exit using results of an aperture test should be more beneficial than performing distance test first. The only problem is developing fast aperture test that doesn't involve distance computations or costly divisions; and Plücker coordinates test exactly solves the problem.

The Plücker test takes advantage of the properties of Plücker coordinates [11, 14], which will be briefly described in the next section. Instead of using barycentric coordinates for the aperture test, the Plücker test relies on testing the relations between a ray and the triangle edges.

We further optimized Plücker test by accomplishing more compact precomputed data representation, reducing overall arithmetic operations count. As additional benefit a branchless implementation of the test is possible by generating a mask for result of computation thus enabling fast ray-triangle intersection on architectures with in-efficient branches (e.g. GPUs).

For further speed up of triangle intersection we use SSE instructions, in the context of ray packets. In order to avoid additional instructions for data rearrangement (which can be costly using SSE), our algorithm relies on a small amount of precomputed data (see section 4.1) for every triangle.

Mailboxing is a technique to avoid multiple intersection tests with triangles that overlap many different leaves [1, 6, 7]. In standard mailboxing, each ray gets a unique ID assigned to it, and each primitive store the ID of the last ray it was tested with. During traversal the duplicate intersection tests can be avoided by simply comparing the current ray ID with the ID of the last tested ray. Such mailboxing requires a significant amount of memory (one integer or pointer per primitive to store ray ID), and can easily lead to costly, incoherent memory accesses and cache thrashing [5]. Furthermore, both memory consumption and cache thrashing get worse when using multiple threads, as the mailbox cannot be shared between threads. Wald in [6] introduces hashed mailboxing which has been shown to be even less efficient than "standard" mailboxing in the general case, though is thread-safe.

Our SIMD-fashion mailboxing alternative is more efficient both in terms of memory and computational resources and is inherently thread-safe.

3. BASICS OF PLÜCKER TEST

Plücker coordinates are an alternate way of describing directed lines in three space using six numbers [1]. By performing a six-dimensional permuted inner product of these numbers we can determine whether two directed lines intersect (the inner product is 0.0) or whether one passes to one side or the other (depending on the sign of the inner product). These three possibilities are illustrated in figure 4 (after Teller in [12]):

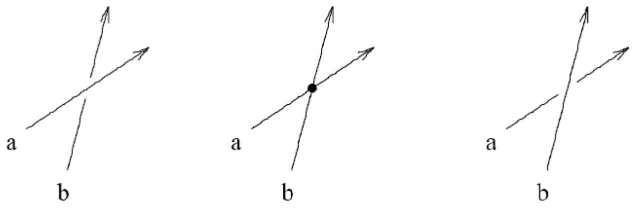


Figure 2: Three possibilities for two directed lines whether one passes to one side or the other (depending on the sign of the inner product).

By determining on which side a line passes with respect to another we can determine if a ray passes through a triangle [14].

So if 6-dim vectors defining edges and a ray are:

$\mathbf{e0}=\{\mathbf{p}-\mathbf{p0},\ \mathbf{p}\times\mathbf{p0}\}$, $\mathbf{e1}=\{\mathbf{p1}-\mathbf{p},\ \mathbf{p1}\times\mathbf{p}\}$, $\mathbf{e2}=\{\mathbf{p0}-\mathbf{p1},\ \mathbf{p0}\times\mathbf{p1}\}$,

$\mathbf{R} = \{\mathbf{d}\times\mathbf{o},\ \mathbf{d}\}$.

Than ray hit test is whether following three dot products of 6-dim vectors have the same sign:

$t0 = (\mathbf{e0}, \mathbf{R})$, $t1 = (\mathbf{e1}, \mathbf{R})$, $t2 = (\mathbf{e2}, \mathbf{R})$.

Note that the inner-product uses only multiplications and additions, allowing for efficient implementation. Single-precision floating-point arithmetic is sufficient for both storing the Plücker coordinates and performing the inner product. The fast test is achieved by pre-computing and storing the Plücker coordinates of triangle edges (**e**0, **e**1, **e**2). The downside of such direct approach is that 18 floats are required to represent a single triangle, although it can be uniquely defined by 9 floating point values. In section 4 we will show that even faster than original test is possible by storing only 9 pre-computed floats and some additional index information.

4. SIMD INTERSECTION ALGORITHM

SIMD-fashion testing multiple rays instead of one ray for intersection with triangle can dramatically reduce the cost of rendering. SIMD architecture performs multiple floating point operations in parallel. This is common technique for speeding up ray-tracing [3, 5, 13]. For traversing the usage of packets wider than current SIMD is used to reduce the average bandwidth required per ray since rays in packet usually access the same nodes and leaves. In [6] traversing of different ray packets size (from 1 to 4096) is compared. For high image-resolution packets of 8 × 8 or even 16×16 rays might be beneficial.

We traverse 4x4 ray SIMD-organized packets simultaneously. Intersection routine is implemented using SIMD instructions. This routine is called 4 times for 4 rays. Our experiments with intersection of larger packets have shown that using more than using 4 rays simultaneously for intersection will not pay off as expected since current CPUs architecture has only 8 SIMD registers. This number of registers is just enough for intersection code which is assumed to work with 4 rays (see Appendix A) to avoid costly register spills/fills.

In the following sections techniques for acceleration of ray-triangle intersection testing are presented. First, section 4.1 describes how the amount of computation required in the rendering process is reduced by preprocessing the scene into a data structure that can be more efficiently used. Second, in section 4.2 the Plücker test using preprocessing data form section 4.1 is described in details. Then, section 4.2.3 shows how one branch or no branches for the whole intersection test can be used.

4.1 Pre-computed triangle information

During the preprocessing stage, full information about a triangle is compressed to only 9 floating point and 3 integer values, based on scaled normal, re-indexing coordinates for triangle vertex **p** and two edges (**e0** and **e1**, see Figure 1). The padding to a 48-byte size allows for a better cache access pattern.

```
struct TriAccel {
        float   nu; //used to store normal data
        float   nv; //used to store normal data
        float   np; //used to store vertex data
        float   pu; //used to store vertex data
        float   pv; //used to store vertex data
        int     ci;  //used to store edges data
        float   e0u; //used to store edges data
        float   e0v; //used to store edges data
        float   e1u; //used to store edges data
        float   e1v; //used to store edges data
        int     pad0; //padding
        int     pad1; //padding
}
```

The preprocessing stage itself is done using SIMD instructions.

4.1.1 Storing Normal

The triangle normal is determined as **n** = (n0, n1, n2). One of the normal components is assumed to be equal to 1.0 and need not to be stored since the intersection algorithm presented does not require any specific normal length. Similarly to projection test, the largest magnitude normal component may be selected and all appropriate values may be scaled by an inverse of this component. In particular, the index w may be defined by the maximum of the absolute value of the triangle normal's components:

$n_w = \max(\mathrm{abs}(ni))$, where i = 0, 1, 2

The two remaining components u, v are then determined:

$nu = n_u/n_w$;

$nv = n_v/n_w$;

where u<v and u+v+w = 3;

The largest magnitude is not a necessary condition for selection of the dropped normal component. Rather, any non-zero component may be used but small magnitude may affect precision.

4.1.2 Storing vertex data

The two components of triangle vertex **p**, see Figure 1, with indices u, v (found at previous section) are simply stored at pu and pv fields of the TriAccel data structure:

$pu = p_u$;

$pv = p_v$;

The dot product of vertex **p** and modified (scaled) triangle normal is stored at np field:

$np = (nu*p_u+nv*p_v+p_w)$;

It is possible to just store the p_w component of **p** instead of dot product, but dot product storage allows additional operations to be saved and allows better register usage on the test/intersection stage (see section 4.2).

4.1.3 Storing edges data

Only two components (having indices u and v, found as described in section 4.1.1) of each of the edges (**e0**, **e1**, see Figure 1) need to be stored, resulting in storing only 4 (e0u, e0v, e1u, e1v), rather than 6, floating point values. In particular, although the edge is a 3D vector, the component having index w (as noted above) need not to be stored. Properly scaled components of **e0** (namely e0u and e0v), as well as of **e1** (namely e1u and e1v) are calculated as follows:

$e0u = (-1)^w e_{0u}/n_w$

$e0v = (-1)^w e_{0v}/n_w$

$e1u = (-1)^w e_{1u}/n_w$

$e1v = (-1)^w e_{1v}/n_w$

The index w itself is stored and used to restore the coordinate components indexing during the intersection test/calculation stage. Only 2 bits are required to store w (since for 3D vectors it takes values 0, 1, 2). Two other indices u and v may be restored by taking one of the rules (u<v) or (v<u) as a convention used throughout preprocessing and intersection test stages (see section 4.2.1). The field in TriAccel structure where the 2-bit w value is stored is referred to as ci.

4.1.4 Axis-Aligned Triangle Flag

Additional information about triangle orientation is also stored. The axis aligned triangles that have two zero components of normal are detected by a preprocessing algorithm. So nu and nv fields equal to zeros. The intersection test (section 4.2) with this particular triangle type is computationally simpler, requires less registers to perform. One bit is allocated in the ci field as an indicator of this particular triangle type.

Major portion of in-door scenes or out-door scenes containing buildings have such triangles which define the surfaces of walls, ceilings, windows, etc. For in-door scenes, actual measurements indicate that more than 50% of rays hit such type of triangles.

4.2 Intersection Algorithm

Generally speaking the ray-triangle intersection problem leads to solving of simple linear system (see Figure 1 for description of vectors):

$\mathbf{p} + u\mathbf{e0} + v\mathbf{e1} = \mathbf{o}+\mathbf{d}t$

if ray hits triangle than

$u+v <= 1$
$u,v,t >=0$

Our intersection algorithm works with a triangle on the basis of two methods:

- Fast hit test based on Plücker coordinates. Using three edges of a triangle, a ray calculation is determined against each edge and whether the ray is clockwise or counterclockwise with respect to the edge is determined. The ray intersects the triangle only when the ray is inside the triangle;
- Intersection point calculation. If ray passed the hit test (thus it is inside the triangle) then intersection point barycentric coordinates namely u,v and scalar distance *t* (see Figure 1) are calculated. This done by solving linear system written above.

4.2.1 Fast Hit Test based on Plücker coordinates

In this section math for fast hit test is presented. Although it is written in pseudo-code it is obviously maps to SIMD implementation for 4-rays packet (see Appendix A).

For description of the **o**,*t* and **d** see the Introduction section. The meaning of TriAccel fields (like e0u, pu, nu) and how u, v and w indices are stored in ci, see section 4.1.

The "det", "dett", "detu" and "detv" are temporary variables introduced for efficient computation re-use. The naming of these temporary variables was selected to illustrate mathematical sense of computed values (for example, "det" is a value proportional to determinant of the linear system of ray-triangle intersection equation described above). "Du" and "Dv" are just temporary values introduced for efficient computation re-use.

$det = d_u$*nu+d_v*nv+d_w;

$dett$=np -(o_u*nu+o_v*nv+o_w) ;

$Du = d_u$ *$dett$ - (pu-o_u)*det

$Dv = d_v$*$dett$ - (pv-o_v)*det

$detu$ = (e1vDu – e1u*Dv)

$detv$ = (e0uDv – e0v*Du)

Having theses values in the hand we can then compute the mask indicating whether values *det - detu – detv*, *detu* and *detv* all have the same sign.

tmpdet0 = *det - detu – detv*

tmpdet0 = *tmpdet0* XOR *detu*

tmpdet1 = *detv* XOR *detu*

tmpdet0 = NOT(*tmpdet0* OR *tmpdet1)*

The elements of *tmpdet0* will have sign bits set to 1 in the positions where the testing values have the same sign and corresponding rays in packet hit the triangle. It could be shown that *detu*, *detv* and *tmpdet0* are equal to above mentioned inner products of Plücker coordinates.

4.2.2 Improved Hit Test for Axis Aligned triangles

Axis aligned triangles have only one non-zero normal's coordinate. In our case this will be nu and nv fields (see section 4.1) equal to zero. Thus, calculations from previous section could be simplified to:

$det = d_w$

$dett$=np- o_w

$Du = d_u$ $dett$ - (pu- o_u)det

$Dv = d_v$ $dett$ - (pv- o_v)det

$detu$ = (e1vDu – e1uDv)

$detv$ = (e0uDv – e0vDu)

4.2.3 Intersection point calculation

After it is determined that the ray intersects the triangle, the exact position of intersection is computed.

Mathematically the following computations have simple meaning of solving linear system of ray-triangle intersection equations using Kramer's rule. These calculations also obviously map to SIMD implementation for 4 rays packet.

rdet = 1/det;

*t = dett * rdet;*

u_{bar} *= detu * rdet;*

v_{bar} *= detv * rdet;*

Found u_{bar}, v_{bar} parameters are barycentric coordinates (see section 1), and *t* - distance of intersection (see Figure 1).

4.2.4 Branchless implementation

The intersection test could be more efficient if it contains no branches because a mis-predicted branch causes a pipeline stall up to the length of the processor's execution pipeline. Streaming architectures like GPU would more benefit form such optimization.

One branch or no branches for the whole intersection test/calculation can be used. In particular, a branchless implementation can be used if the mask is generated on the results of computation. Version with branch could be implemented, for example, as "if(…){ }"construction. Branchless implementation is possible by performing section 4.2.2 computations in both cases (hit and no hit) plus using additional bit-wise logical operations with a bit mask. Particularly, this mask is used to define to either store the intersection parameters with given triangle or keep their values unchanged.

5. INVERSE MAILBOXING

Due to the fact that a reference to the same primitive can exist in multiple acceleration structure leaves, a ray packet could perform the primitive intersection test multiple times. A simple technique to avoid such unnecessary intersection tests is mailboxing [1, 6, 7]. Mailboxing allows for checking if a given primitive has already been intersected by the current ray packet or not.

The recent implementations of mailboxing like [6] assign a unique ID to each ray packet. After an intersection test, the primitive is marked as already tested by assigning the current ray packet ID to the primitive. So unnecessary tests can now be avoided by performing a simple check before every potential primitive intersection: If the current ray packet ID matches the ID assigned to the primitive candidate, an intersection test between the ray packet and the primitive has already been performed and can therefore be omitted.

In contrast, we store a history of triangles tested with currently traversed ray packet. The triangle index which is a 32-bit integer is used as triangle ID in the history array. Since the total number of triangles to be tested with any given ray packet is not known in advance it might seem that such history array should be either of variable size or large enough to accommodate the whole history of triangles. In fact, any, even short, history allows a speed up due to skipping some number of duplicate tests. In addition, when efficient acceleration structure is used the total number of triangles tested for a ray packet is not very high. Statistics collected over a large number of models shows that the history of fixed size of 8 last triangles is the optimal balance between the history size and overhead on history maintenance. The history is organized as a ring buffer ensuring that last 8 triangles are stored. Since the size of the buffer is 8 it requires only 2 SIMD operations to check if specific ID is stored in the buffer.

Due to its small and fixed size the history can be thread-local allowing traversing multiple packets simultaneously in multiple threads. The history can be implemented as an automatic variable located in the traversal+intersection function testing a ray packet intersection with the whole acceleration structure. Thus it is automatically placed on a thread's stack making sure that each thread has its own history.

Since our algorithm stores triangle IDs instead of ray IDs as in traditional mailboxing we call our approach *inverse mailboxing*.

6. RESULTS

We estimate a number of clocks spent in our intersection algorithm working with 16 ray packet. Note that out-of-order execution, caching effects, varying branch table history entries, and etc. make it difficult to determine the exact amount of cycles required for a given algorithm. Therefore, following cycle statistics in this paper should be seen as estimates rather than exact values. We compare our results with approaches recently published in [5], where analogous SIMD implementation of intersection test for 16 rays is used.

	axis-aligned triangles	non axis-aligned triangles		
	our approach	**our approach**	Plücker test from [5]	Projection test from [5]
all 16 rays hit triangle:	**384 clocks**	**460 clocks**	590 clocks	620 clocks
all 16 rays miss triangle:	**184 clocks**	**240 clocks**	310 clocks	420 clocks

Table 1. Cycle statistics of intersection routine measured for main cases in compare to our approach. Note that Carstein in [5] doesn't use optimizations for axis-aligned triangles. All our data was collected using Intel® Pentium IV as in [5].

We also have tested our intersection routine for multi-threaded ray-tracing on a 2-way Intel® Core™2 Duo machine (so 4 threads on 4 cores). We employ 4x4 SIMD ray packet tracing as described in this paper. All rendering performance further is reported for resolution of 1024x1024, using lighting (1 point light source) and shadows.

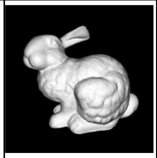
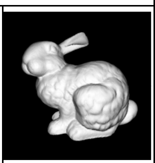
scene and # triangles		Inverse mailboxing usage		
		No	Yes	Improvement Rate
Stanford Bunny, 69k		33.6	36.9	9.7%
Stanford Dragon, 863k		11.3	12.4	9.7%
Happy Buddha 1087K		17.9	22.1	23.4%

Table 2. Rendering performance (in FPS) comparison for different models with inverse mailboxing on/off. Performance numbers are collected for resolution of 1024x1024, using lighting (1 point light source) and shadows.

Thus inverse mailboxing improves caching behavior of ray-tracing (see Table 2) removing any additional memory consumption. It is also well-suited for multi-threading case.

It is also proved by tests that improvement for intersection test (described in section 4.2.2) proposed for axis-aligned triangles pais off well, see Table 3. It is especially beneficial for case when a lot of when axis-aligned triangles are in scene (like "Soda Hall", see Table 3). In opposite case when no axis-aligned triangles are present (like "Bunny" scene, see Table 3) our algorithm doesn't introduce any penalty.

scene and # triangles		considering AA-triangles (inverse mailboxing is used)		
		No	Yes	Improvement Rate
Stanford Bunny, 69k		36.9	36.9	0%
Ward Conference, 282K		27.3	28.9	5,8%

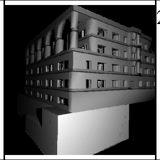
Soda Hall 2195K		23.3	32.4	39%

Table 3. Rendering performance (in FPS) comparison for different models with on/off feature considering axis-aligned triangles. All performance is collected for resolution of 1024x1024, using lighting (1 point light source) and shadows.

7. CONCLUSION

We present a ray-triangle intersection algorithm that quickly determines if a ray intersects a triangle interior and finds parameters of intersection. We further optimized the Plücker test by accomplishing a SIMD implementation and reducing number of operations by clever using certain amount of pre-computing values. The branchless implementation by generating a mask is described.

We show how certain values could be pre-computed to save computations and avoid data rearrangement. We also show how to further save computations for axis-orthogonal triangles by processing them separately.

We also present SIMD-fashion, inherently thread-safe inverse mailboxing to avoid unnecessary intersection tests for ray packet in case when many leaves share the same triangle.

The combination of processor-specific optimizations with algorithms that exploit the coherence of ray-tracing makes it possible to achieve real-time performance on a modern CPU.

8. REFERENCES

[1] Andrew Glassner. "*An Introduction to Ray Tracing*", Morgan Kaufmann, 1989.

[2] M. Pharr and G. Humphreys. "*Physically Based Rendering: From Theory to Implementation*". Morgan Kaufman, 2004.

[3] I. Wald, C. Benthin, M. Wagner, and P. Slusallek, "*Interactive Rendering with Coherent Ray Tracing*". Computer Graphics Forum, v.20 n.3, pp. 53–164, 2001. (Proceedings of Eurographics 2001).

[4] A. Reshetov, A. Soupikov. and J. Hurley, "*Multi-level ray tracing algorithm*". Proceedings of ACM SIGGRAPH (2005), pp.1176-1185.

[5] C. Benthin, "*Realtime Ray Tracing on current CPU Architectures", PhD thesis, Saarland University, 2006.

[6] I. Wald, "*Realtime ray tracing and interactive global illumination*", PhD thesis, Saarland University, 2004.

[7] D. Kirk and J. Arvo, "*Improved Ray Tagging For Voxel-Based Ray Tracing"*. In Graphics Gems II, pp. 264–266. Academic Press, 1991.

[8] J. Arenberg, "R*ay-Triangle Intersection with Barycentric Coordinates*", in Ray Tracing News, edited by Eric Haines, v 1.1, n. 11, November 4, 1988. http://www.acm.org/tog/resources/RTNews/

[9] D. Badouel, "*An Efficient Ray-Polygon Intersection*"- in Graphics Gems- edited by Andrew S. Glassner- Academic Press Inc., 1990, pp. 390-393.

[10] E. Haines, "*Point in Polygon Strategies*", in Graphics Gems IV edited by Paul S. Heckbert, AP Professional, 1994, pp.24-46.

[11] K. Shoemake. "*Plücker Coordinate Tutorial*". In Ray Tracing News, 1998. http://www.acm.org/tog/resources/RTNews/html/rtnv11n1.html.

[12] S.J. Teller. *"Computing the Antipenumbra of an Area Light Source",* Computer Graphics'92, 26(2).

[13] K. Dmitriev, V. Havran, and H.-P. Seidel, "*Faster Ray Tracing with SIMD Shaft Culling"*. Research Report MPI-I-2004-4-006, Max-Planck-Institut für Informatik, Saarbrücken, Germany.

[14] T. Möller, B. Trumbore, "*Fast, minimum storage ray-triangle intersection*", Journal of Graphics Tools, v.2 n.1, pp.21-28, 1997.

APPENDIX A

```
//Main part of ray/triangle hit test (see section 4.2.1) in SIMD.
struct RTSSEVec3f{
__m128 t[3];
};
//directions of rays in packet
RTSSEVec3f d;
//origins of rays in packet
RTSSEVec3f o;
//params loaded from TriAccel and replicated to __m128:
const __m128& nu,np, nv,pu,pv, e0u,e0v,e1u,e1v;
//indices computed from from 'ci' field of TriAccel:
const int& u,v,w;
//temporary variables
__m128 det,dett,detu, detv, nrv, nru, du,dv,ou, ov, tmpdet0,
tmpdet1;

/* ----ray-packet/triangle hit test ---- */
//dett = np -(ou*nu+ov*nv+ow)
dett = np;
dett = _mm_sub_ps(dett,
*((const __m128*)(&reinterpret_cast<const float*>(o.t)[w])));
du = nu;
dv = nv;
ou = pu;
ou = _mm_sub_ps(ou,
*((const __m128*)(&reinterpret_cast<const float*>(o.t)[u])));
ov = pv;
ov = _mm_sub_ps(ov,
*((const __m128*)(&reinterpret_cast<const float*>(o.t)[v])));
du = _mm_mul_ps(du, ou);
dv = _mm_mul_ps(dv, ov);
dett = _mm_add_ps(dett, du);
dett = _mm_add_ps(dett, dv);
//det =du*nu+dv*nv+dw
du =_mm_load_ps(&reinterpret_cast<const float*>(d.t)[u]);
dv =_mm_load_ps(&reinterpret_cast<const float*>(d.t)[v]);
det = nu;
det = _mm_mul_ps(det, du);
nrv = nv;
nrv = _mm_mul_ps(nrv, dv);
```

```
det = _mm_add_ps(det,
_mm_load_ps(&reinterpret_cast<const float*>(d.t)[w]));
det = _mm_add_ps(det, nrv);
//Du = du*dett - (pu-ou)*det
nru = _mm_mul_ps(ou, det);
du  = _mm_sub_ps(du, nru);
//Dv = dv*dett -  (pv-ov)*det
nrv = _mm_mul_ps(ov, det);
dv  = _mm_sub_ps(dv, nrv);
//detu = (e1vDu – e1u*Dv)
nru = e1v;
nrv = e1u;
nru = _mm_mul_ps(nru, du);
nrv = _mm_mul_ps(nrv, dv);
detu = _mm_sub_ps(nru, nrv);
//detv = (e0uDv – e0v*Du)
nrv = e0u;
nrv = _mm_mul_ps(nrv, dv);
dv = e0v;
dv = _mm_mul_ps(dv, du);
detv = _mm_sub_ps(nrv, dv);
/* Having det, detu and detv values in hands we can then
compute the mask indicating whether each of 4 values ‘det - detu
– detv’, ‘detu’ and ‘detv’ all have the same sign indicating that
corresponding rays in packet hit the triangle (see section 4.2.1)*/
```

About the authors

Maxim Shevtsov is a Research Scientist in Nizhniy Novgorod Laboratory of Intel Corporation. He received MS degree in CS from Novosibirsk State University in 2003. His contact email is maxim.y.shevtsov@intel.com

Alexei Soupikov is a Staff Researcher leading advanced graphics research team in Nizhniy Novgorod Laboratory of Intel Corporation. He is MS in CS and EE. His contact email is alexey.soupikov@intel.com

Alexander Kapustin is a Research Scientist in Nizhniy Novgorod Laboratory of Intel Corporation. He received PhD in 1987. His contact email is alexander.kapustin@intel.com