Watertight Ray/Triangle Intersection

Sven Woop Carsten Benthin Ingo Wald
Intel Labs

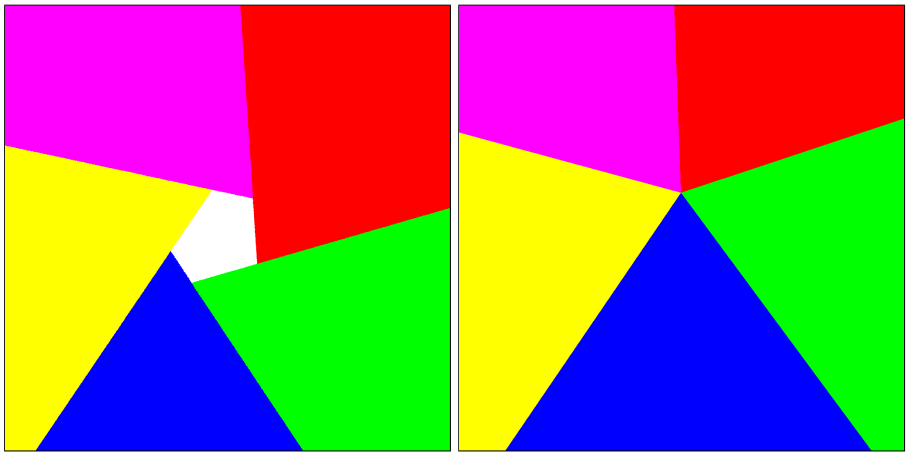


Figure 1. Plücker coordinates guarantee watertightness along the edges, but edges do not meet exactly at the vertices. The algorithm described in this paper fixes this issue, and guarantees watertightness along edges and at the vertices.

Abstract

We propose a novel algorithm for ray/triangle intersection tests that, unlike most other such algorithms, is watertight at both edges *and* vertices for adjoining triangles, while also maintaining the same performance as simpler algorithms that are not watertight. Our algorithm is straightforward to implement, and is, in particular, robust for all triangle configurations including extreme cases, such as small and needle-like triangles. We achieve this robustness by transforming the intersection problem using a translation, shear, and scale transformation into a coordinate space where the ray starts at the origin and is directed, with unit length, along one coordinate axis. This simplifies the remaining intersection problem to a 2D problem where edge tests can be done conservatively using single-precision floating-point arithmetic. Using our algorithm, numerically challenging cases, where single precision is insufficient, can be detected with almost no overhead and can be accurately handled through a (rare) fallback to double precision. Because our algorithm is conservative but not exact, we must dynamically enlarge bounds during BVH traversal to conservatively bound the triangles intersected.

1. Introduction

While *performance* implications of ray-tracing large and complex scenes are well investigated, there is less research that covers the accuracy and consistency of ray-triangle intersection calculations under those conditions. Specifically, in large and complex scenes, many triangles are small and far away from the camera and finely tessellating thin objects such as pipes, wires, and hair often produces needle-shaped triangles. Unless carefully handled, ray tracing such triangles can leave microscopic "cracks" between objects from missed intersections. When a ray incorrectly passes through a crack, it erroneously enters or leaves an object that was intended to be modeled as closed. This may corrupt the final radiance calculation along the ray and appear in the final image as an incorrect pixel value. While these errors may arise from correct tracing of an incorrect model, in this paper, we assume correct geometry and consider the case of errors arising from limited accuracy in the floating-point calculations of the ray-triangle intersection test. These typically arise in the case of triangles whose perimeter is small compared to the magnitude of their coordinates as well as for the needle-shaped triangles, because both of those situations lead to calculations that mix very large and very small values. Those are particularly problematic for single-precision (32-bit) floating-point representations common in rendering applications. Increasing precision to 64- or 80-bit floating-point representations both incurs a disproportional performance reduction on many architectures and does not solve the underlying numerical problem. Furthermore, higher tessellation rates will generally lead to ever smaller and thinner triangles and, thus, increase the severity of such accuracy-based issues.

The computational geometry literature includes algorithms that are more numerically stable than naive ray-triangle implementations common in computer graphics. However, these are also substantially more expensive, and, thus, are often not used in rendering. In this paper, we present a novel algorithm that has the efficiency of traditional single-precision ray-triangle intersection algorithms and is watertight at both edges *and* vertices, even in numerically challenging cases.

2. Previous Work

Approaches from computational geometry, such as traditional floating-point filters using interval arithmetic [Fortune and Van Wyk 1993] with a fallback to arbitrary precision calculations [Shewchuk 1996; Karamcheti et al. 1999] can solve the watertightness problem, but they are slow and complex. In particular, conservatively implementing interval arithmetic typically requires frequent switching of rounding modes, which on most modern architectures significantly reduces instruction throughput. This makes these approaches hard to use for rendering and, in particular, challenging to implement on GPUs.

For rendering, the main focus is typically performance, and very few watertight ray-triangle intersection algorithms have been proposed so far. The only ray-triangle intersection algorithm we are aware of that has been designed explicitly for guaranteeing watertightness is the work by Dammertz and Keller [2006] who describe a recursive refinement approach for watertight intersection of free-form surfaces with the application to triangles as a special case. Like our approach, their approach is stable for all triangle configurations, but their deep subdivision (down to epsilon-sized bounds) significantly diminishes performance.

Badouel [1990] calculates the ray-triangle intersection by first calculating the world space hit location and then projecting this hit location into the xy, xz, or yz plane where a 2D edge test is performed. Shevtsov et al. [2007] and Wald [2004] take a similar approach with slightly modified precalculations. None of these approaches achieves watertightness of adjoining triangles: since the calculation of the world space hit location is dependent on the entire triangle surface (i.e., not just the shared edge between two triangles), two triangles sharing the same edge will end up with slightly different calculations for the shared edge's edge test.

Möller and Trumbore [1997] solve the ray-triangle intersection by directly solving a linear system of equations using Cramer's rule and by evaluating determinants using scalar triple products. Kensler and Shirley [2006] propose a similar ray-triangle intersection algorithm that re-factors the scalar triple products differently, which allows pre-calculation of the geometry normal in addition to the two edges.

To save computations, virtually all of the above-mentioned tests replace the third edge test with a simplified $u+v \leq 1$ test. Though cheaper than a full edge test, no algorithm using this trick can guarantee watertightness along the edges, as the same shared edge could be computed in one way in one triangle, but with a different test in the neighboring triangle. This inconsistency in how the edge value is computed will then, in floating-point arithmetic, lead to slightly different and inconsistent results.

Yet another source of inconsistency in computing an edge test is that neighboring triangles might span the same edge in opposing directions, again leading to slightly inconsistent ways of computing the "same" edge test in neighboring triangles. To guarantee watertightness along an edge, the edge tests have to be anti-commutative under floating-point operations, as done, for example, in [Benthin 2006; Kin and Choi 1995; Davidovič et al. 2012]. While this makes the edge decision consistent along shared triangle *edges*, watertightness at the *vertices* cannot be achieved this way, since the edges can still fail to meet exactly at the vertex; see Figure 1.

A well-known method to perform these edge tests are Plücker coordinates [Erickson 1997; Jones 2000], which can even be precalculated for the ray and each edge to improve performance [Kin and Choi 1995]. We have found Plücker coordinates to be numerically problematic in large scenes where the errors around the vertices can become very noticeable. A numerically very stable approach of calculating the edge test is presented by Davidovic et al. [2012], who use the center of the edge to obtain anti-commutativity and builds on the Chirkov test [Chirkov 2005].

Hanika [2011] describes a fixed point ray-triangle intersection test also based on the Chirkov test [Chirkov 2005]. By representing the cross products of this test with sufficient precision and properly rounding the dot products, the algorithm becomes provably watertight. This fixed-point approach cannot be applied directly to floating-point calculations, since their intermediate results require additional precision. However, one basic idea of their approach is to guarantee that the intersected triangle is larger than the original one. This approach of enlarging the triangle is a very common workaround for watertightness problems, for example, by testing the edge equations not against zero but against a small positive epsilon. However, we don't know of any approach that calculates such an epsilon in order to guarantee watertightness under all circumstances. Further, a large constant epsilon can cause issues with self-shadowing of the surface along the edges.

For rasterization, the watertight and robust handling of triangles is a solved problem. Hardware implementations of rasterization project triangles onto a 2D domain and snap their vertices to fixed-point numbers. The use of sufficient precision for the edge tests makes robust and watertight handling of meshes possible. However, in a ray tracer, rays are not always starting at the same origin and aligned inside a bounded frustum, which makes a direct adoption of this technique not possible in general for ray tracing.

3. Watertight Ray-Triangle Intersection

Our watertight ray-triangle intersection algorithm operates in two stages. In the first stage, an affine transformation is applied to the ray and the vertices of the triangle to simplify the intersection problem. Similar to the setup stages of rasterization, floating-point rounding errors in this first stage do not destroy the watertightness of an input mesh, as long as the mesh contains no T-vertices. In the second stage, the simplified intersection problem is accurately solved using 2D edge tests with a fallback to double precision.

As affine transformation, we choose a transformation that simplifies the ray R to the unit ray R' with origin $P' = (0,0,0)$ and direction $D' = (0,0,1)$. While there are different options for choosing this transformation, we will use a translation followed by a shear and scale. Compared to the alternative approach of using rotations, the shear introduces smaller rounding errors and is more efficient to calculate. Note, in particular, that this transformation is *only* dependent on the ray (i.e., it does *not* depend on whatever triangle is being intersected) and will thus be identical for each ray-triangle intersection this ray performs.

Without loss of generality, we assume for the rest of this section that the z-

component of the ray direction D has the largest absolute value. This can be achieved by renaming the x-,y-, and z-dimensions in a winding preserving way. If D_z is negative, the winding direction of the triangle would get inverted by the pre-transformation described in the following. To compensate for this, we additionally swap the x- and y-coordinates of all calculations in this case (see the pseudocode in Appendix A for details on how to implement this efficiently). After these coordinate changes, the following transformation M is precalculated at ray-generation time and reused for all successive ray-triangle intersections:

$$M = \begin{pmatrix} 1 & 0 & -D_x/D_z \\ 0 & 1 & -D_y/D_z \\ 0 & 0 & 1/D_z \end{pmatrix}.$$

At ray-triangle intersection time, we first calculate the transformed triangle vertices relative to the ray origin P and then transform these translated vertices using the transformation M:

$$A' = M \cdot (A - P),$$
$$B' = M \cdot (B - P),$$
$$C' = M \cdot (C - P).$$

Note, that we do not have to explicitly transform the ray, as we know by construction of the transformation that the ray is transformed to the unit ray. To finish the intersection, we will use the test from Benthin [2006] and calculate scaled barycentric coordinates:

$$U = D' \cdot (C' \times B'),$$
$$V = D' \cdot (A' \times C'),$$
$$W = D' \cdot (B' \times A').$$

As we intersect with the unit ray with $D' = (0,0,1)$, these formulas simplify significantly to 2D edge tests:

$$U = C'_x \cdot B'_y - C'_y \cdot B'_x,$$
$$V = A'_x \cdot C'_y - A'_y \cdot C'_x,$$
$$W = B'_x \cdot A'_y - B'_y \cdot A'_x.$$

If $U < 0$, $V < 0$, or $W < 0$, the ray misses the triangle. We then calculate the determinant of the system of equations as $\det = U + V + W$. If this determinant is zero, the ray is co-planar to the triangle and we assume a miss. This guarantees later divisions by det to be safe in the algorithm.

If neither of these tests fails, we calculate the scaled hit distance T by interpolating the z-values of the transformed vertices:

$$T = U \cdot A'_z + V \cdot B'_z + W \cdot C'_z.$$

As this distance is calculated using non-normalized barycentric coordinates, the actual distance still requires division by det. To defer this costly division until actually required, we first compute a scaled depth test by rejecting the triangle if the hit is either before the ray $T \leq 0$ or behind an already-found hit $T \geq \det \cdot t_{\text{hit}}$. If these tests pass we know that the ray hits the triangle and calculate the normalized barycentric coordinates $u = U/\det$, $v = V/\det$, $w = W/\det$ and unscaled hit distance $t = T/\det$.

While this approach performs backface culling, one can easily intersect two-sided triangles by reporting a miss if at least one of the barycentric coordinates is smaller than 0 and at least one is larger than 0: $(U < 0 \vee V < 0 \vee W < 0) \wedge (U > 0 \vee V > 0 \vee W > 0)$. When doing so, we have to handle the case of a negative determinant correctly in the depth test; this is outlined in detail in Appendix B.

To avoid false negatives for rays hitting exactly *on* the edge, it is important that the chosen triangle test handles an edge that evaluates to 0 as being inside the triangle. Our two-sided triangle test has this property for both sides of the triangle. One often-used optimization is merely testing whether all three edge tests return the same sign; this, however, does not guarantee this property for the back-side of the triangle, and is therefore not being used.

3.1. Watertightness

As the pre-transformation preserves watertightness, we consider the transformed vertices A', B', and C' as accurate inputs to the 2D stage of the algorithm, although they contain rounding errors.

The IEEE 754 floating-point standard [IEEE 1985] requires calculations (such as multiplications) to be internally executed with (principally) infinite precision and finally precisely rounded to a nearby floating-point number. Further, none of the IEEE rounding modes will destroy the ordering of two real numbers; thus, $x \geq y$ always implies $\text{round}(x) \geq \text{round}(y)$. Inserting the precise product $B'_x \cdot A'_y$ for x and $B'_y \cdot A'_x$ for y yields

$$B'_x \cdot A'_y \geq B'_y \cdot A'_x \Rightarrow \text{round}(B'_x \cdot A'_y) \geq \text{round}(B'_y \cdot A'_x).$$

For the W edge test as an example, this equation shows that if the edge test classifies a ray as inside the triangle (left side is true), the floating-point algorithm will also classify the ray as inside the triangle (right side is true).

Conversely, if a ray is classified as outside the triangle using floating-point calculations (right side is false), it would also be classified as outside using infinite-precision arithmetic (left side is false). This is even true if only a single last-digit bit distinguishes the two floating-point products.

However, for the case where the floating-point algorithm calculates two equal products and, consequently, the edge test evaluates to 0, no definite statement about the real ordering of the precisely calculated products can be made. As we conservatively treat this case as a hit, the mesh is guaranteed to be watertight and no misses can occur along the edges and vertices.

One remaining case to discuss is that we classify the triangle as a miss if $\text{det} = U+V+W = 0$, which occurs when the ray is embedded in the plane spanned by the triangle. Classifying this case as a miss is actually wrong, as the ray could very well hit the triangle when lying in the plane; however, we decided to avoid introducing special code for handling this corner-case correctly. Despite this simplification, the algorithm is still watertight as rays propagating parallel through a triangle will hit one of the neighboring triangles with our algorithm.

The $\text{det} = 0$ test further classifies degenerate triangles that form a point or a line segment as a miss. For these triangles, all three edge equations always evaluate to exactly 0 for each ray. Consequently, without this determinant test, each ray would report a hit with such a degenerate triangle.

3.2. Precision

As just described, our test is conservative since rays whose edge equation evaluates to 0 will be counted as hitting the triangle. This conservativeness is important in order to guarantee watertightness, but accuracy should be just as important: a test always returning an intersection would also be watertight, but useless in practice.

The 2D edge tests described above can tolerate large relative errors in the edge value: only a single least-significant bit distinguishing the products of the form $B'_x \cdot A'_y$ and $B'_y \cdot A'_x$ of an edge test is sufficient to accurately resolve the side of the edge the ray passes. This property makes the test very robust, and, without modification, the algorithm already passes most of our stress tests, except for cases of extremely small triangles that the ray misses at a large distance. In these cases, the angle between the transformed vertices A' and B' can become so small that the products $B'_x \cdot A'_y$ and $B'_y \cdot A'_x$ become equal when evaluated with single precision. Conservatively categorizing the case of both products being equal as a hit can cause such triangles to report false positives in areas far away from the triangle. However, these regions are typically culled by an acceleration structure anyway; thus, ray traversal should not visit such problematic triangles in the first place. For this reason, the algorithm described so far works very well in practice.

Nevertheless, an algorithm that works under all circumstances can easily be obtained through a fallback of the 2D edge test to double precision if and only if one of the single-precision tests evaluates to 0. In this case, the single-precision algorithm cannot make a definite statement about the side the ray passes. Double-precision arithmetic is sufficient to accurately store the product of two single-precision floating-point values, since 53 bits of mantissa for double precision is more than two times the 24 bits of mantissa for single precision. Thus, this fallback can accurately resolve all special cases. In our test scenes, this fallback is only required about once per every million ray-triangle intersections. Note, however, that using double precision for an entire ray-triangle intersection algorithm would not be a solution to the watertightness problem; it would only reduce the probability of issues around edges or vertices from occurring.

3.3. World-Space Bounding

While our approach guarantees watertightness, we do *not* make the correct edge decisions between the ray and the actual world-space triangle, since round-off errors obviously occur during the pre-transformation stage. These rounding errors have about the same effect as if the triangle vertices were slightly shifted by a minuscule amount. Since all vertices are shifted equally, no cracks appear between triangles, but some shifted triangles may no longer be completely enclosed by the acceleration structure's world-space bounding boxes, and may thus not be found by the traversal. Though very unlikely, for truly extreme cases this does actually happen. To compensate for this effect, the geometry bounds of the spatial index structure have to get extended by a ray-dependent epsilon region.

This issue is not unique to our algorithm, since most ray-triangle intersection algorithms are indeed intersecting slightly shifted or rotated triangle-like shapes that are not conservatively bounded through their world-space vertices. While these issues are normally ignored, we present a solution in order to obtain full watertightness through the entire ray-tracing algorithm.

The rounding errors in the translation of the vertices and, in particular, in the subsequent shear, require us to enlarge bounding boxes slightly to conservatively bound the intersected triangle shape. In the following we use $\ominus$ and $\odot$ to denote floating-point subtraction and product and $-$ and $\cdot$ to denote subtraction and product of real numbers. We will calculate the maximal error when translating and shearing an arbitrary vertex $A = (x, y, z)$ of the bounding box K. The x-coordinate of the translated and sheared vertex $A' = (x', y', z')$ is calculated using floating-point calculations:

$$x' \quad = \quad (x \ominus P_x) \ominus (\text{round}(S_x) \odot (z \ominus P_z)).$$

Note that we need to round the correct shear constant $S_x = D_x/D_z$ to a floating-point value, while the ray and input vertices are already considered to be present in floating point. Assuming round-to-nearest mode is active, we use the constant $\varepsilon = 2^{-24}$ in the following calculations to bound the value x'. We will use interval-arithmetic notation to extract the relative error of all floating-point calculations:

$$\begin{aligned}
x' &\subset (x-P_x)(1\pm\varepsilon)\ominus(S_x(1\pm\varepsilon)\odot(z-P_z)(1\pm\varepsilon))\\
&\subset (x-P_x)(1\pm\varepsilon)\ominus(S_x(1\pm\varepsilon)\cdot(z-P_z)(1\pm\varepsilon))(1\pm\varepsilon)\\
&\subset (x-P_x)(1\pm\varepsilon)^3\ominus(S_x\cdot(z-P_z))(1\pm\varepsilon)^3\\
&\subset ((x-P_x)(1\pm\varepsilon)^3-(S_x\cdot(z-P_z))(1\pm\varepsilon)^3)(1\pm\varepsilon)\\
&\subset (x-P_x)(1\pm\varepsilon)^4-(S_x\cdot(z-P_z))(1\pm\varepsilon)^4.
\end{aligned}$$

As ε is small, we can bound the interval $(1\pm\varepsilon)^4$ by $1\pm 5\varepsilon$:

$$\begin{aligned}
&\subset (x-P_x)(1\pm 5\varepsilon)-(S_x\cdot(z-P_z))(1\pm 5\varepsilon)\\
&\subset (x-P_x)-S_x(z-P_z)+5\varepsilon(\pm(x-P_x)\pm S_x(z-P_z))\\
&\subset (x-P_x)-S_x(z-P_z)\pm 5\varepsilon(|x-P_x|+|S_x(z-P_z)|).
\end{aligned}$$

Knowing that $|S_x|\in[\frac{1}{2},1]$ we can bound this further to

$$\begin{aligned}
&\subset (x-P_x)-S_x(z-P_z)\pm 5\varepsilon(|x-P_x|+|z-P_z|)\\
&= ((x\pm 5\varepsilon(|x-P_x|+|z-P_z|))-P_x)-S_x(z-P_z).
\end{aligned}$$

The last line shows that accurately translating and shearing a region $x\pm 5\varepsilon(|x-P_x|+|z-P_z|)$ bounds the calculated value x'. Inserting for x the K_{xmin}-coordinate of the box and considering, additionally, all $z\in[K_{\text{zmin}},K_{\text{zmax}}]$ yields a conservative extension for the xmin-coordinate of the box:

$$\varepsilon_{\text{xmin}} = 5\varepsilon(|K_{\text{xmin}}-P_x|+\max(|K_{\text{zmin}}-P_z|,|K_{\text{zmax}}-P_z|).$$

And similarly, for the upper bound and y-dimension:

$$\varepsilon_{\text{xmax}} = 5\varepsilon\cdot(|K_{\text{xmax}}-P_x|+\max(|K_{\text{zmin}}-P_z|,|K_{\text{zmax}}-P_z|),$$

$$\varepsilon_{\text{ymin}} = 5\varepsilon\cdot(|K_{\text{ymin}}-P_y|+\max(|K_{\text{zmin}}-P_z|,|K_{\text{zmax}}-P_z|)),$$

$$\varepsilon_{\text{ymax}} = 5\varepsilon\cdot(|K_{\text{ymax}}-P_y|+\max(|K_{\text{zmin}}-P_z|,|K_{\text{zmax}}-P_z|)).$$

When using a box extended this way, we guarantee enclosing the intersected triangles. The bounds in the z-dimension need no correction as they are not affected by the shear and the later ray-box intersection will make a distance calculation consistent with the transformation applied to the z-coordinate of the vertices.

Some care has to be taken when calculating and using the error bounds with floating-point arithmetic conservatively. Rounding modes would have to be set to round calculations for lower error bounds towards $-\infty$ and calculations for upper error bounds towards $+\infty$. As switching the rounding modes is not possible on some hardware architectures and expensive on others, we use a workaround of rounding a positive number up by one ulp by multiplying it with $1+2^{-23}$ and down by multiplying it with $1-2^{-23}$. Care has also to be taken for some other calculations of the algorithm; some detailed reference source code is given in Appendix B.

The calculated conservative bounds require an exact or conservative ray-box test, such as the first test described in [Williams et al. 2005]. For optimization, we will use precalculated reciprocal ray directions $1/D_x$, $1/D_y$, and $1/D_z$ for the ray-plane distance calculation used in the test. As these reciprocal values have an attached error of half an ulp (or even higher if not calculated through full-accuracy divisions), we have to conservatively use values $(1/D_{x,y,z}) \cdot (1-2^{-23})$ to calculate distances to near planes and values $(1/D_{x,y,z}) \cdot (1+2^{-23})$ to calculate distances to far planes.

We further apply two optimizations to improve performance. First, during traversal we do not evaluate the above error estimate for each bounding box since this would introduce an overhead to the innermost traversal loop. Instead, we pre-calculate the values $\varepsilon_{\text{xmin}}, \varepsilon_{\text{xmax}}, \varepsilon_{\text{ymin}}, \varepsilon_{\text{ymax}}$ conservatively for the entire scene bounding box once per ray traversal.

Second, we do not directly extend the bounding boxes by this epsilon region, but translate the ray origin such that the calculations become conservative. Thus, in the ray-box test, instead of calculating $(K_{\text{xmin}} - \varepsilon_{\text{xmin}}) - P_x$, we calculate $K_{\text{xmin}} - (P_x + \varepsilon_{\text{xmin}})$ (and similarly for other dimensions). This allows us to move the constants of the form $P_x + \varepsilon_{\text{xmin}}$ out of the loop. This optimization results in a traversal loop that has no more inner loop operations than traditional algorithms; see Appendix B.

4. Results

Table 1 compares the performance of tracing primary rays using our method to the standard Möller-Trumbore ray-triangle intersection algorithm [Möller and Trumbore 1997], to the algorithm of Davidovic [2012], and to the work of Dammertz [2006]. When using the same traversal approach, the performance of our watertight ray-triangle intersector is roughly comparable to Möller-Trumbore, slightly faster than Davidovic, and significantly faster than Dammertz et al.

To guarantee watertightness for the entire ray-casting operation, we have to use conservative bounds during traversal, which reduces overall performance by about 12%. This performance reduction comes from various sources, such as the per-ray precalculations required for the conservative traversal, some increased register pressure caused by using two versions of the ray origin and reciprocal direction inside the traversal, and a number of additional traversal steps and ray-triangle intersections (about 1% for our scenes) due to conservative traversal.

Compared to the alternative approach of [Dammertz and Keller 2006], our performance is significantly higher for all of our test scenes. While the approach of Dammertz also guarantees watertightness, it suffers from long subdivision chains of

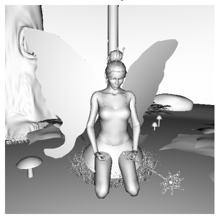
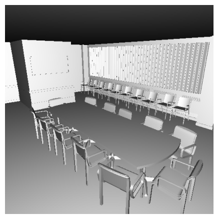
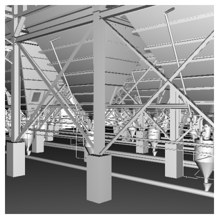
	Fairy Forrest	Conference	Dragon	Dragon Far	Power Plant
					
[Möller and Trumbore 1997]	61.4M (100%)	78.8M (100%)	85.6M (100%)	78.1M (100%)	11.9M (100%)
[Davidovič et al. 2012]	57.0M (93%)	72.7M (92%)	82.5M (96%)	75.6M (97%)	10.2 (85%)
[Dammertz and Keller 2006]	23.6M (38%)	24.2M (30%)	55.5M (64%)	50.0M (64%)	0.9M (8%)
Watertight Intersection	59.1M (96%)	78.3M (99%)	84.6M (99%)	77.9M (99%)	11.9M (100%)
+ Conservative Traversal	53.4M (87%)	69.0M (88%)	77.0M (90%)	69.6M (89%)	11.0M (92%)

Table 1. Performance comparison of ray traversal using a standard Möller-Trumbore ray-triangle intersector, the approach by Davidovic, the watertight ray-triangle intersection algorithm by Dammertz, and our watertight ray-triangle intersection algorithm with standard BVH traversal and conservative BVH traversal. For different scenes, we show primary ray performance in million rays per second measured on a dual socket Xeon E5-2690 CPU (16 hyper-threaded cores total) running at 2.9 GHz. We use single-ray traversal and a 4-wide BVH as spatial index structure and store the triangles using an indexed face set. The watertight triangle test alone is roughly as fast as Möller-Trumbore; the modified traversal is more expensive, but even in that case our method is only at most 10% slower than Möller-Trumbore, and significantly faster than Dammertz et al. The test by Davidovic performs slightly slower than Möller-Trumbore. We rendered the Dragon model from close distance and far distance (with zoom), in order to demonstrate that in practice the conservative traversal does not significantly affect performance for scenes with small triangles viewed from a far distance.

the triangle to achieve floating-point precision. Our implementation of Dammertz's algorithm includes their optimization of falling-back into the slow subdivision case only if a standard ray-triangle test reports a miss. This optimization does not work very well in the power-plant scene, where intersected triangles are often missed.

Our conservative traversal enlarges the bounds slightly, which can result in a performance penalty for small triangles viewed from a far distance. To show that this is not an issue in practice, we have rendered the Dragon model a second time, but from a far distance (10 times larger than normal). For this configuration, the performance of our algorithm degredates only minimally more relative to Möller-Trumbore.

Table 2 shows the number of false negatives obtained with different ray-triangle intersection algorithms. For [Möller and Trumbore 1997], Plücker coordinates [Kin and Choi 1995], and [Davidovič et al. 2012], we use a traversal algorithm that conservatively intersects the bounding boxes. We count the number of false negatives we obtain by rendering the inside of differently triangulated and shifted spheres. The table shows [Möller and Trumbore 1997] to fail about three times in a million intersections. It turns out that the failures are mostly caused by the edge tested through $u+v<1$. Intersection using Plücker coordinates [Kin and Choi 1995] are numerically very unstable for the shifted sphere; thus, we recommend not using this algorithm.

The algorithm of [Davidovič et al. 2012] is watertight along the edges and numerically stable, but nevertheless we measured up to 117 failures per million intersections. Even though we made sure that ray traversal performs a conservative ray-box test, the triangle as intersected by that algorithm is not always contained entirely inside its bounding box; thus, a similar fix as presented in Section 3.3 would be required here, too. Most of the false negatives are caused by this bounding issue. False negatives caused by holes around the vertices are rare and only noticeable when zooming onto them.

As was expected, the algorithm by Dammertz [Dammertz and Keller 2006] shows no false negatives. Also as expected, our algorithm shows no false negatives when

	Sphere40k	Shifted40k	Sphere4M	Shifted4M
[Möller and Trumbore 1997]	41	21	327	339
[Kin and Choi 1995]	0	77 M	127	85 M
[Davidovič et al. 2012]	0	148	0	11778
[Dammertz and Keller 2006]	0	0	0	0
Watertight Intersection	0	0	0	2
+ Conservative Traversal	0	0	0	0

Table 2. This table shows the number of false negatives for shooting 100 M primary visibility rays from the interior of different spheres for different algorithms. We render a unit sphere with about 40 K triangles in the origin (Sphere40k), shifted 1000 units away from the origin (Shifted40k), and the same configurations with 4 M triangles (Sphere4M and Shifted4M).

used with our special conservative traversal, but even when using a standard BVH traversal almost no false negatives occur (about 1 in 50 million). Consequently our approach performs very well even when used with a standard traversal algorithm.

5. Summary and Conclusion

We have presented a fast single-precision floating-point algorithm for ray-triangle intersection, with a fallback to double precision that guarantees watertightness and is numerically stable even for problematic triangles. The algorithm achieves good performance compared to ray tracing with the standard Möller-Trumbore ray-triangle intersection algorithm.

We believe that our algorithm will be useful, in particular, for production rendering, where robust algorithms are desired and scenes typically have a huge number of potentially problematic small triangles. A further application we see is for ray-tracing scenes consisting of subdivision surfaces or NURBs. Approaches that subdivide these higher-order primitives often end up with very small triangles that need robust handling.

Appendix A

Pseudo C++ code for the watertight ray-triangle intersection. For each ray, we precalculate once the maximum dimension *kz* (and orthogonal dimensions *kx* and *ky*) as well as the shear constants.

```
/* calculate dimension where the ray
   direction is maximal */
int kz = max_dim(abs(dir));
int kx = kz+1; if (kx == 3) kx = 0;
int ky = kx+1; if (ky == 3) ky = 0;

/* swap kx and ky dimension to preserve
   winding direction of triangles */
if (dir[kz] < 0.0f) swap(kx,ky);

/* calculate shear constants */
float Sx = dir[kx]/dir[kz];
float Sy = dir[ky]/dir[kz];
float Sz = 1.0f/dir[kz];
```

The second part is the intersection code invoked for each ray-triangle intersection. This version of the code supports backface culling, both enabled and disabled.

```
/* calculate vertices relative to ray origin */
const Vec3f A = tri.A-org;
const Vec3f B = tri.B-org;
const Vec3f C = tri.C-org;

/* perform shear and scale of vertices */
const float Ax = A[kx] - Sx*A[kz];
const float Ay = A[ky] - Sy*A[kz];
const float Bx = B[kx] - Sx*B[kz];
const float By = B[ky] - Sy*B[kz];
const float Cx = C[kx] - Sx*C[kz];
const float Cy = C[ky] - Sy*C[kz];

/* calculate scaled barycentric coordinates */
float U = Cx*By - Cy*Bx;
float V = Ax*Cy - Ay*Cx;
float W = Bx*Ay - By*Ax;

/* fallback to test against edges
    using double precision */
if (U == 0.0f || V == 0.0f || W == 0.0f) {
  double CxBy = (double)Cx*(double)By;
  double CyBx = (double)Cy*(double)Bx;
  U = (float)(CxBy - CyBx);

  double AxCy = (double)Ax*(double)Cy;
  double AyCx = (double)Ay*(double)Cx;
  V = (float)(AxCy - AyCx);

  double BxAy = (double)Bx*(double)Ay;
  double ByAx = (double)By*(double)Ax;
  W = (float)(BxAy - ByAx);
}

/* Perform edge tests. Moving this test before
   and at the end of the previous conditional
   gives higher performance. */
#ifdef BACKFACE_CULLING
  if (U<0.0f || V<0.0f || W<0.0f) return;
#else
  if ((U<0.0f || V<0.0f || W<0.0f) &&
      (U>0.0f || V>0.0f || W>0.0f)) return;
#endif
/* calculate determinant */
float det = U+V+W;
```

```
if (det == 0.0f) return;

/* Calculate scaled z-coordinates of vertices
   and use them to calculate the hit distance. */
const float Az = Sz*A[kz];
const float Bz = Sz*B[kz];
const float Cz = Sz*C[kz];
const float T = U*Az + V*Bz + W*Cz;

#ifdef BACKFACE_CULLING
  if (T < 0.0f || T > hit.t * det)
    return;
#else
  int det_sign = sign_mask(det);
  if (xorf(T,det_sign) < 0.0f) ||
      xorf(T,det_sign) > hit.t * xorf(det, det_sign))
    return;
#endif

/* normalize U, V, W, and T */
const float rcpDet = 1.0f/det;
hit.u = U*rcpDet;
hit.v = V*rcpDet;
hit.w = W*rcpDet;
hit.t = T*rcpDet;
```

Appendix B

Pseudo C++ code for the modified version of the ray-box test used during ray traversal. At the beginning of ray-traversal we do some precalculations.

```
/* Calculate the offset to the near and far planes for
   the kx, ky, and kz dimension for a box stored in the
   order lower_x, lower_y, lower_z, upper_x, upper_y,
   upper_z in memory. */

Vec3i nearID(0,1,2), farID(3,4,5);
int nearX = nearID[kx], farX = farID[kx];
int nearY = nearID[ky], farY = farID[ky];
int nearZ = nearID[kz], farZ = farID[kz];
if (dir[kx] < 0.0f) swap(nearX,farX);
if (dir[ky] < 0.0f) swap(nearY,farY);
if (dir[kz] < 0.0f) swap(nearZ,farZ);

/* conservative up and down rounding */
float p = 1.0f + 2^-23;
float m = 1.0f - 2^-23;
float up(float a) { return a>0.0f ? a*p : a*m; }
float dn(float a) { return a>0.0f ? a*m : a*p; }

/* fast rounding for positive numbers */
float Up(float a) { return a*p; }
float Dn(float a) { return a*m; }

/* Calculate corrected origin for near- and far-plane
   distance calculations. Each floating-point operation
   is forced to be rounded into the correct direction. */

const float eps = 5.0f * 2^-24;
Vec3f lower = Dn(abs(org-box.lower));
Vec3f upper = Up(abs(org-box.upper));
float max_z = max(lower[kz],upper[kz]);

float err_near_x = Up(lower[kx]+max_z);
float err_near_y = Up(lower[ky]+max_z);
float org_near_x = up(org[kx]+Up(eps*err_near_x));
float org_near_y = up(org[ky]+Up(eps*err_near_y));
float org_near_z = org[kz];

float err_far_x = Up(upper[kx]+max_z);
float err_far_y = Up(upper[ky]+max_z);
float org_far_x = dn(org[kx]-Up(eps*err_far_x));
float org_far_y = dn(org[ky]-Up(eps*err_far_y));
float org_far_z = org[kz];

if (dir[kx] < 0.0f) swap(org_near_x,org_far_x);
if (dir[ky] < 0.0f) swap(org_near_y,org_far_y);

/* Calculate corrected reciprocal direction for near-
   and far-plane distance calculations. We correct
   with one additional ulp to also correctly round
   the subtraction inside the traversal loop. This
   works only because the ray is only allowed to
   hit geometry in front of it. */

float rdir_near_x = Dn(Dn(rdir[kx]));
float rdir_near_y = Dn(Dn(rdir[ky]));
float rdir_near_z = Dn(Dn(rdir[kz]))
float rdir_far_x  = Up(Up(rdir[kx]));
float rdir_far_y  = Up(Up(rdir[ky]));
float rdir_far_z  = Up(Up(rdir[kz]));
```

During ray traversal, the ray-box intersection code has the same complexity as traditional algorithms, but we are using the corrected ray origin and reciprocal ray direction calculated above.

```
float tNearX = (box[nearX] - org_near_x) * rdir_near_x;
float tNearY = (box[nearY] - org_near_y) * rdir_near_y;
float tNearZ = (box[nearZ] - org_near_z) * rdir_near_z;
float tFarX  = (box[farX ] - org_far_x ) * rdir_far_x;
float tFarY  = (box[farY ] - org_far_y ) * rdir_far_y;
float tFarZ  = (box[farZ ] - org_far_z ) * rdir_far_z;
float tNear  = max(tNearX,tNearY,tNearZ,rayNear);
float tFar   = min(tFarX ,tFarY ,tFarZ ,rayFar );
bool  hit    = tNear <= tFar;
```

References

BADOUEL, D. 1990. An Efficient Ray-Polygon Intersection. In *Graphics Gems*, A. S. Glassner, Ed. Academic Press Professional, Inc., San Diego, CA, USA, 390–393. 67

BENTHIN, C. 2006. *Realtime Ray Tracing on Current CPU Architectures*. PhD thesis, Saarländische Universitäts- und Landesbibliothek, Postfach 151141, 66041 Saarbrücken. 67, 69

CHIRKOV, N. 2005. Fast 3D Line Segment-Triangle Intersection Test. *J. Graphics Tools 10*, 3, 13–18. 68

DAMMERTZ, H., AND KELLER, A. 2006. Improving Ray Tracing Precision by Object Space Intersection Computation. In *IEEE Symposium on Interactive Ray Tracing 2006*, IEEE, Los Alamitos, CA, 25–31. 67, 74, 75, 76

DAVIDOVIČ, T., ENGELHARDT, T., GEORGIEV, I., SLUSALLEK, P., AND DACHSBACHER, C. 2012. 3D Rasterization: A Bridge Between Rasterization and Ray Casting. In *Proceedings of the 2012 Graphics Interface Conference*, Canadian Information Processing Society, Toronto, Ont., Canada, 201–208. 67, 68, 74, 75, 76

ERICKSON, J. 1997. Plücker Coordinates. *Ray Tracing News 10*, 3. 67

FORTUNE, S., AND VAN WYK, C. J. 1993. Efficient Exact Arithmetic for Computational Geometry. In *Proceedings of the Ninth Annual Symposium on Computational Geometry (SCG '93)*, ACM, New York, NY, 163–172. 66

HANIKA, J. 2011. *Spectral Light Transport Simulation Using a Precision-Based Ray Tracing Architecture*. PhD thesis, Universität Ulm. 68

IEEE. 1985. *IEEE Standard for Binary Floating-Point Arithmetic*. ANSI/IEEE Std 754-1985. IEEE Computer Society Press, Silver Spring, MD. 70

JONES, R. 2000. Intersecting a Ray and a Triangle with Plücker Coordinates. *Ray Tracing News 13*, 1. 67

KARAMCHETI, V., LI, C., PECHTCHANSKI, I., AND YAP, C. 1999. A Core Library for Robust Numeric and Geometric Computation. In *ACM Symposium on Computational Geometry (SCG'99), Applied Track*, ACM Press, New York, NY, 351–359. 66

KENSLER, A., AND SHIRLEY, P. 2006. Optimizing Ray-Triangle Intersection via Automated Search. In *IEEE Symposium on Interactive Ray Tracing 2006*, IEEE, Los Alamitos, CA, 33 –38. 67

KIN, J. A., AND CHOI, K. 1995. Ray Tracing Triangular Meshes. Tech. rep., York University, Dept. of Computer Science. `http://http://www.cse.yorku.ca/~amana/research/mesh.pdf`. 67, 76

MÖLLER, T., AND TRUMBORE, B. 1997. Fast, minimum storage ray/triangle intersection. *journal of graphics tools (jgt)* 2, 1, 21–28. 67, 74, 75, 76

SHEVTSOV, M., SOUPIKOV, A., KAPUSTIN, A., AND NOVOROD, N. 2007. Ray-Triangle Intersection Algorithm for Modern CPU Architectures. In *Procedings of GraphiCon 2007*, Moscow State University, Moscow, Russia. 67

SHEWCHUK, J. R. 1996. Robust Adaptive Floating-Point Geometric Predicates. In *Proceedings of the Twelfth Annual Symposium on Computational Geometry (SCG 96)*, ACM, New York, NY, 141–150. 66

WALD, I. 2004. Realtime Ray Tracing and Interactive Global Illumination. *PhD thesis, Saarland University*. 67

WILLIAMS, A., BARRUS, S., MORLEY, R. K., AND SHIRLEY, P. 2005. An Efficient and Robust Ray-Box Intersection Algorithm. In *ACM SIGGRAPH 2005 Courses*, ACM, New York, NY, SIGGRAPH '05. 74

Author Contact Information

Sven Woop	Carsten Benthin	Ingo Wald
Intel Corporation	Intel Corporation	Intel Corporation
sven.woop@intel.com	carsten.benthin@intel.com	ingo.wald@intel.com

Sven Woop, Carsten Benthin, Ingo Wald, Watertight Ray/Triangle Intersection, *Journal of Computer Graphics Techniques (JCGT)*, vol. 2, no. 1, 65–82, 2013
http://jcgt.org/published/0002/01/05/

Received:	2013-01-24		
Recommended:	2013-03-29	Corresponding Editor:	Carsten Dachsbacher
Published:	2013-06-28	Editor-in-Chief:	Morgan McGuire

This paper was updated on 2013-07-09 to reference Möller and Trumbore's original 1997 paper in place of their 2005 talk; thanks to Stephen Hill for this suggestion.