Fast Ray-Triangle Intersections by Coordinate Transformation

Doug Baldwin
State University of New York at Geneseo
Michael Weber
State University of New York at Geneseo

Figure 1. A complex scene ray traced using precomputed global-to-barycentric coordinate transformations. Scene description from http://www.pbrt.org/scenes.php.

Abstract

Ray-triangle intersection is a crucial calculation in ray tracing. We present a new algorithm for finding these intersections, occupying a different place in the spectrum of time-space tradeoffs than existing algorithms do. Our algorithm provides faster ray-triangle intersection calculations at the expense of precomputing and storing a small amount of extra information for each triangle. Running under ideal experimental conditions, our algorithm is always faster than the standard Möller and Trumbore algorithm, and faster than a highly tuned modern version of it except at very high ray-triangle hit rates. Replacing the Möller and Trumbore algorithm with ours in a complete ray tracer speeds up image generation by between 1 and 6%, depending on the image. We have coded our method in C++, and provide two implementations as supplements to this article.

1. Introduction

Triangles are a crucial primitive in geometric models, which in turn means that ray-triangle intersection is a crucial calculation for ray tracing. We present a new algo-

 ISSN 2331-7418

rithm for finding these intersections, occupying a different place in the spectrum of time-space trade-offs than existing algorithms do. In particular, our algorithm provides faster ray-triangle intersection calculations at the expense of storing a small amount of additional information about each triangle.

The standard algorithm for computing ray-triangle intersections in ray tracing is due to Möller and Trumbore [1997]. Given a ray and a triangle, this algorithm transforms the ray from the global coordinate system to a triangle-specific barycentric one and then tests for intersection in that coordinate system. The algorithm is fast because it requires only a modest number of calculations and supports early detection of many cases in which the ray cannot intersect the triangle. No information is precomputed, and thus there is no memory overhead. Our algorithm also transforms rays from global to barycentric coordinate systems, but precomputes the coordinate transformation. Precomputing the transformation allows each intersection calculation to be faster than in Möller and Trumbore's algorithm. Furthermore, we construct the transformation in a way that ensures that many of the coefficients in its matrix form have known values that do not need to be stored, thus minimizing the memory overhead of the precomputation.

Our algorithm was originally developed in a university course, where it was successfully used in student and faculty projects. Subsequently, we coded it as the core of an experimental program from which we could take careful timing measurements, and we built a version of the PBRT ray tracer [Pharr and Humphreys 2010] that used it in place of a version of Möller and Trumbore's algorithm. In the isolated experimental setting our algorithm ran faster than the Möller and Trumbore algorithm, and also ran faster than the tuned Möller and Trumbore algorithm in the Embree framework [Wald et al. 2014]. In the more realistic pbrt setting it slightly out-performed Möller and Trumbore while producing visually equivalent images.

2. Mathematical Foundations

Our precomputed transformation method originates in the idea that any triangle instance can be constructed by transforming a canonical triangle. In particular, a triangle with vertices $\overrightarrow{v_1} = (\overrightarrow{v_1}_x, \overrightarrow{v_1}_y, \overrightarrow{v_1}_z)$, $\overrightarrow{v_2} = (\overrightarrow{v_2}_x, \overrightarrow{v_2}_y, \overrightarrow{v_2}_z)$, and $\overrightarrow{v_3} = (\overrightarrow{v_3}_x, \overrightarrow{v_3}_y, \overrightarrow{v_3}_z)$[1] can be constructed from a canonical right triangle in the xy plane with sides of unit length via a transformation given by the matrix

[1] We use subscripts "x," "y," and "z" to identify the x, y, and z components of a point or vector; e.g., $\overrightarrow{v_1}_x$ denotes the x component of vertex $\overrightarrow{v_1}$. Arrows over variables (e.g., $\overrightarrow{v}$) indicate vectors or points—the distinction between the two types is not significant to our discussion.

$$T = \begin{bmatrix} \overrightarrow{v_2}_x - \overrightarrow{v_1}_x & \overrightarrow{v_3}_x - \overrightarrow{v_1}_x & a & \overrightarrow{v_1}_x \\ \overrightarrow{v_2}_y - \overrightarrow{v_1}_y & \overrightarrow{v_3}_y - \overrightarrow{v_1}_y & b & \overrightarrow{v_1}_y \\ \overrightarrow{v_2}_z - \overrightarrow{v_1}_z & \overrightarrow{v_3}_z - \overrightarrow{v_1}_z & c & \overrightarrow{v_1}_z \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Here, a, b, and c form a "free vector" $\vec{f} = (a, b, c)$, and their values have no influence on the canonical-to-global transformation. They do however, influence the inverse transformation, and the main contribution of this work is to describe how to select them in a manner that makes that inverse efficient to store and apply.

Our main interest in this transformation is that its inverse transforms points and vectors to the canonical triangle's coordinate system, i.e., to a barycentric coordinate system for the instance triangle. In order to simplify discussion of finding the inverse, we define the triangle's edge vectors to be $\overrightarrow{E_1} = \overrightarrow{v_2} - \overrightarrow{v_1}$ and $\overrightarrow{E_2} = \overrightarrow{v_3} - \overrightarrow{v_1}$, writing the transformation as

$$T = \begin{bmatrix} \overrightarrow{E_1}_x & \overrightarrow{E_2}_x & a & \overrightarrow{v_1}_x \\ \overrightarrow{E_1}_y & \overrightarrow{E_2}_y & b & \overrightarrow{v_1}_y \\ \overrightarrow{E_1}_z & \overrightarrow{E_2}_z & c & \overrightarrow{v_1}_z \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

We can now make T invertible by ensuring that it has a non-zero determinant. The determinant of T is $(\overrightarrow{E_1} \times \overrightarrow{E_2}) \bullet \vec{f}$, i.e., we need

$$|T| = (\overrightarrow{E_1} \times \overrightarrow{E_2}) \bullet \vec{f} \neq 0.$$

We further note that $\overrightarrow{E_1} \times \overrightarrow{E_2}$ is a normal to the triangle, and so use $\vec{n} = \overrightarrow{E_1} \times \overrightarrow{E_2}$ in the following. As long as $\vec{n} \neq \vec{0}$, we can find values for a, b, and c that make $\vec{n} \bullet \vec{f} \neq 0$. Furthermore, $\vec{n} = \vec{0}$ if and only if all three vertices of the triangle are collinear, so we can always construct an invertible transformation for any non-degenerate triangle.

Making the free vector a unit-length vector in one of the principle directions ensures that $|T| \neq 0$ and that the inverse transformation can be applied quickly to points and vectors. We use the largest magnitude component of the normal to determine which direction[2]. Specifically, if the x component of the normal has a larger absolute value than any other component, the free vector points in the x direction, i.e., $a = 1$ and $b = c = 0$. Similarly, if the normal's y component has the largest magnitude then we let the free vector be $(0, 1, 0)$, and we let the free vector be $(0, 0, 1)$ if the normal's z component has the largest magnitude. The inverse transformation exists in each of these cases, although each case leads to a different inverse transformation matrix. For the case where $a = 1$ and $b = c = 0$ the inverse matrix is

[2]Thanks to an anonymous referee of an early version of this paper for suggesting this tactic as numerically more stable than our original approach of using any non-zero component.

$$\begin{bmatrix} 0 & \frac{\overrightarrow{E_{2z}}}{\vec{n}_x} & -\frac{\overrightarrow{E_{2y}}}{\vec{n}_x} & \frac{(\overrightarrow{v_3}\times\overrightarrow{v_1})_x}{\vec{n}_x} \\ 0 & -\frac{\overrightarrow{E_{1z}}}{\vec{n}_x} & \frac{\overrightarrow{E_{1y}}}{\vec{n}_x} & -\frac{(\overrightarrow{v_2}\times\overrightarrow{v_1})_x}{\vec{n}_x} \\ 1 & \frac{\vec{n}_y}{\vec{n}_x} & \frac{\vec{n}_z}{\vec{n}_x} & -\frac{\vec{n}\bullet\overrightarrow{v_1}}{\vec{n}_x} \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Similarly, the inverse transformations for the cases where $b = 1$ or $c = 1$ are

$$\begin{bmatrix} -\frac{\overrightarrow{E_{2z}}}{\vec{n}_y} & 0 & \frac{\overrightarrow{E_{2x}}}{\vec{n}_y} & \frac{(\overrightarrow{v_3}\times\overrightarrow{v_1})_y}{\vec{n}_y} \\ \frac{\overrightarrow{E_{1z}}}{\vec{n}_y} & 0 & -\frac{\overrightarrow{E_{1x}}}{\vec{n}_y} & -\frac{(\overrightarrow{v_2}\times\overrightarrow{v_1})_y}{\vec{n}_y} \\ \frac{\vec{n}_x}{\vec{n}_y} & 1 & \frac{\vec{n}_z}{\vec{n}_y} & -\frac{\vec{n}\bullet\overrightarrow{v_1}}{\vec{n}_y} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} \frac{\overrightarrow{E_{2y}}}{\vec{n}_z} & -\frac{\overrightarrow{E_{2x}}}{\vec{n}_z} & 0 & \frac{(\overrightarrow{v_3}\times\overrightarrow{v_1})_z}{\vec{n}_z} \\ -\frac{\overrightarrow{E_{1y}}}{\vec{n}_z} & \frac{\overrightarrow{E_{1x}}}{\vec{n}_z} & 0 & -\frac{(\overrightarrow{v_2}\times\overrightarrow{v_1})_z}{\vec{n}_z} \\ \frac{\vec{n}_x}{\vec{n}_z} & \frac{\vec{n}_y}{\vec{n}_z} & 1 & -\frac{\vec{n}\bullet\overrightarrow{v_1}}{\vec{n}_z} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

respectively.

3. Implementation

Our ray-triangle intersection method has a straightforward implementation: during triangle initialization, compute a global-to-barycentric coordinate transformation matrix as given in section 2, and store that matrix as part of the triangle. More precisely, only the top three rows of the matrices shown in section 2 need to be stored, since the fourth row is only computationally relevant if points and vectors are represented in homogeneous coordinates, which isn't necessary, and any coefficient from the fourth row that ever is wanted can easily be deduced knowing that the row is always $(0, 0, 0, 1)$.

Once the global-to-barycentric transformation matrix is available, determining whether a ray intersects the triangle, and if so where, consists of four steps:

1. Multiply the ray by the transformation matrix to put the ray into the triangle's coordinate frame

2. Calculate a t value for the intersection as

$$t = -\frac{\vec{o}_z}{\vec{d}_z}$$

where $\vec{o}$ is the transformed ray's origin point, and $\vec{d}$ is its direction vector

3. Calculate barycentric coordinates for the intersection as

$$b_1 = \vec{o}_x + t\vec{d}_x, \qquad b_2 = \vec{o}_y + t\vec{d}_y$$

4. If $0 \leq b_1 \leq 1$ and $0 \leq b_2 \leq 1$ and $b_1 + b_2 \leq 1$ the ray intersects the triangle, otherwise it does not.

This series of calculations can terminate early if t is too small or large to represent a valid intersection, or if b_1 is out of the range that permits an intersection.

Our method also has a more space-efficient implementation. The only things not known a priori about the global-to-barycentric coordinate transformation are the values of 9 of its coefficients and which column contains the $(0, 0, 1, 0)$ pattern. Thus, it is only necessary to store 9 coefficients and a 3-valued column selector rather than 12 coefficients. Doing this also allows the global-to-barycentric transformation to be performed with slightly less arithmetic, because it is not necessary to multiply by the coefficient that is known to be 1, or to multiply and add with coefficients known to be 0. This paper's supplemental materials include C++ versions of both this efficient implementation and the straightforward one.

4. Results

We characterized the performance of our precomputed transformation algorithm in two ways: through a series of execution time measurements on stand-alone implementations of our algorithm and two others, and by embedding our algorithm in a complete ray tracer. In both cases, our algorithm proved to be faster than the alternatives we compared it to.

4.1. Stand-Alone Timing Experiments

We compared the raw execution time of our algorithm to that of the original Möller and Trumbore algorithm, from which many deployed ray-triangle intersection functions are descended, and to the adaptation of Möller and Trumbore in the Embree framework [Wald et al. 2014], which is highly tuned for fast execution on modern CPUs. We took execution time measurements from a C++ program that contained the 12-coefficient version of our algorithm, the 9-coefficient version, the original Möller and Trumbore algorithm, and Embree's algorithm. Following recommendations from Löfstedt and Akenine-Möller [2004], the program generated sets of rays and triangles, each ray paired with one triangle, and then ran each intersection algorithm on all the pairs in each set. We varied both data set size and ray-triangle hit rate. Complete code for this program is available in this article's supplemental materials.

We ran this experiment on a 2.6 GHz MacBook Pro with 8 GB of RAM, 3 MB level 3 cache, and 512 GB of SSD "disk," running MacOS X 10.9.5. We compiled the program in XCode 6.1.1, optimizing for fastest speed (-O3).

We used Möller and Trumbore's algorithm as described in their paper [Möller and Trumbore 1997], and took Embree's intersection algorithm from Embree v. 2.9.0 as distributed at `https://embree.github.io/`. Both had to be modified slightly to work in our experimental system, but we made as few modifications as possible and stayed as close as possible to the intent of the original algorithms. Most importantly, Embree exploits SIMD parallelism by (among other things) processing each ray against multiple triangles at once, but this feature is nullified in our experiment which intersects each ray with only one triangle. Furthermore, no effort has yet been made to exploit such parallelism in our algorithm. We therefore did not use SIMD features or streaming instructions (beyond any the compiler might have generated on its own) in our implementation of Embree's algorithm, although we did preserve other features that make it very fast even when testing single rays against single triangles. Comparing our algorithm to a full implementation of Embree's, including studying ways in which our algorithm might also exploit SIMD parallelism, is a natural direction for future research.

Table 1 summarizes the results of this experiment. For complete raw data and analysis, see the supplements to this article. For each combination of data set size (i.e., number of ray-triangle pairs) and hit rate, the table shows the running times of both versions of the precomputed transformation algorithm as percentages of the running times of the Möller and Trumbore algorithm and Embree's algorithm. The columns labeled "Pre(12)" contain data for our algorithm using the straightforward 12-coefficient representation of transformation matrices, while the columns labeled "Pre(9)" contain data from the space-efficient 9-coefficient representation. Under the conditions in this experiment, Pre(12) is consistently faster than Pre(9). Over all data set sizes and hit rates, Pre(12) ran between 2 and 3 times faster than the original Möller and Trumbore algorithm, and outperformed Embree's algorithm for hit rates of 0.1 and 0.5. At the highest hit rate, 0.9, Pre(12) was comparable to or slightly faster than Embree's algorithm. For the 500,000 and 1,000,000 pair data sets the standard errors in the raw data are about the same as the difference between the Pre(12) and Embree times, i.e., the difference may be solely due to variability in the measurements; however this is not the case for the larger data sets. Based on the changes in relative times as hit rates increase, it seems likely that Embree's algorithm would outperform ours for hit rates above 0.9. The slower Pre(9) version of our algorithm outperformed Möller and Trumbore in all cases, taking between $\frac{1}{2}$ and $\frac{3}{4}$ of the time Möller and Trumbore did, and outperformed Embree's algorithm for the lowest hit rate. For the 0.5 hit rate Pre(9) was comparable to Embree's algorithm on small data sets, and slightly faster on large. At the highest hit rate, Pre(9) was uniformly about 40% slower than Embree. It is, however, worth noting that the Embree algorithm requires one vertex, two edges, and the triangle's normal to be precomputed, for a total of 48 bytes of precomputed data (assuming single-precision floating point numbers),

Pairs	Hit Rate	Pre(12) Time as % Of...		Pre(9) Time as % Of...	
		M&T	Embree	M&T	Embree
500,000	0.1	46%	57%	67%	82%
	0.5	46%	74%	59%	96%
	0.9	49%	92%	72%	136%
1,000,000	0.1	46%	56%	68%	83%
	0.5	48%	79%	62%	102%
	0.9	42%	94%	62%	138%
5,000,000	0.1	38%	48%	56%	71%
	0.5	35%	58%	51%	84%
	0.9	42%	95%	62%	140%
10,000,000	0.1	38%	48%	56%	71%
	0.5	35%	57%	51%	84%
	0.9	42%	95%	62%	139%

Table 1. Execution time comparisons between two versions of the precomputed transformation ray-triangle intersection algorithm, Möller and Trumbore's algorithm, and Embree's

while Pre(9) only requires 37 bytes of precomputed data.

4.2. A Complete Ray Tracer

In order to test the precomputed transformation method with complex scenes in a complete ray tracer, we inserted it into the PBRT ray tracer [Pharr and Humphreys 2010] (version 2.0.0), replacing ray-triangle intersection code based on the Möller and Trumbore algorithm. We tested both the 9- and 12-coefficient versions of our algorithm. We modified PBRT's `Triangle` class to store the transformation coefficients (and column selector for the 9-coefficient version), and modified its `Intersect` and `IntersectP` methods to use the stored transformation as outlined in Section 3. Using single-precision floating point values for the coefficients (consistent with how PBRT represents other real numbers) and one byte for the column selector, the overhead of precomputing the transformation was 48 bytes per triangle for the straightforward implementation and 37 for the efficient one. We tested eight scenes provided by the PBRT developers [Pharr and Humphreys 2014] in both the original and modified programs, comparing execution times and image quality. All versions of PBRT were compiled using the Makefile in the PBRT distribution. We used the same computer as in the stand-alone timing experiments; PBRT found and used four cores.

Table 2 describes the scenes we used and gives the timing results. For each scene, the table gives the name of the scene, the number of triangles created to represent it, the time an unmodified PBRT needs to render it, and the improvements in rendering time when using each version of the precomputed transformation method, as

Scene	Triangles	Unmodified Time (Secs)	Pre(9) Time (% of Unmod.)	Pre(12) Time (% of Unmod.)
Teapots	6,804	632.8	94.2%	94.3%
Mesh Buddha	29,892	8078	95.9%	95.7%
San Miguel	2,503,052	47670	96.0%	96.1%
Plants	1,171,562	5025	96.9%	96.6%
Buddha	1,087,720	339.4	97.3%	98.0%
Villa	2,624,966	21950	98.4%	98.4%
Killeroo	66,532	661.1	98.5%	98.9%
Bunny	69,453	962.9	98.5%	98.8%

Table 2. Timing comparisons between an unmodified PBRT and versions of PBRT using the precomputed transformation ray-triangle intersection method

percentages of the unmodified time. Times are total CPU times across all four cores in seconds, averaged over six runs. Complete raw data and analysis is available in the supplements to this article.

The precomputed transformation method consistently ran faster than the original, with the improvement ranging from 1 to 6%. There was no clear distinction between the two forms of the method, although the version that only stores 9 coefficients seems slightly better in most cases. This is an interesting contrast to the stand-alone experiments, in which the 12-coefficient version was clearly superior.

Figures 1 and 2 show some examples of the images we rendered. To the naked eye, the images produced by the precomputed transformation methods were indistinguishable from those produced by the unmodified PBRT. To detect subtler differences, we used an image viewer program to alternate rapidly between images of the same scene from each method; differences between the images then stood out as movements or color jumps as the images alternated. This technique showed that there were still no differences in four of the test cases ("Buddha," "Mesh Buddha," "Bunny," and "Killeroo"), slight changes in apparent texture in two ("Plants" and "San Miguel"), and changes in the position of noise speckles in two more ("Teapots" and "Villa"). We believe that these differences reflect the fact that while mathematically all three methods compute the same intersections, they perform different sequences of computations and so may experience different round-off errors.

The scenes that we tested exercise a number of different ray tracing situations and effects, demonstrating that our precomputed transformations do not interfere with other aspects of ray tracing. Of particular note, the "Mesh Buddha" scene uses object instancing to generate a basic shape from multiple tiny copies of itself. Our method's ability to render it shows that our global-to-barycentric transformation composes smoothly with non-trivial modeling transformations.

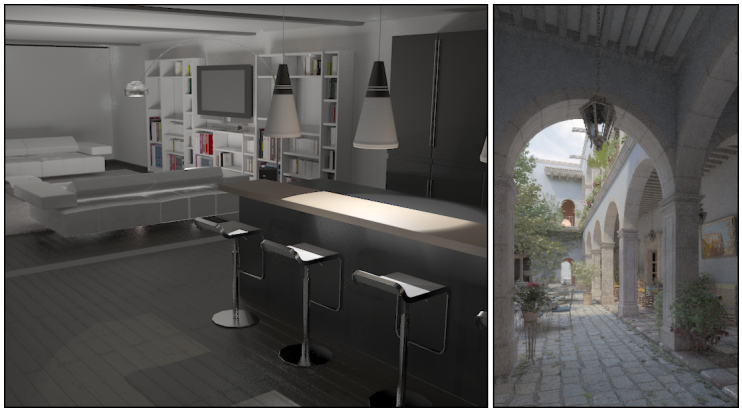

Figure 2. Examples of images rendered by the precomputed transformation algorithm. Scene descriptions from `http://www.pbrt.org/scenes.php`.

5. Conclusion

We have developed, tested, and characterized a method for computing ray-triangle intersections that uses a small amount of precomputed transformation information to support fast intersection tests. The memory occupied by transformation coefficients does not prevent our approach from working well, even on large scenes. Run in isolation, our method substantially outperforms the basic Möller and Trumbore algorithm, and somewhat outperforms a highly tuned version of it. In a complete ray tracer our method delivers modestly but measurably better execution times than that ray tracer's implementation of Möller and Trumbore, with visually equivalent results.

More detailed comparisons between the precomputed transformation algorithm and other ray-triangle intersection methods are still possible. As already mentioned, its ability to exploit SIMD parallelism should be further explored, and it should be compared to a more complete implementation of Embree's algorithm. Further work is also needed to understand exactly when the 9-coefficient version is superior to the 12-coefficient one and vice versa. Yet even as it stands today, our method is a valuable addition to the algorithm kits of ray tracing developers willing to invest memory to improve execution time.

Acknowledgements

Thanks to the students in SUNY Geneseo's spring 2015 Math 384, who provided the first impetus to develop the algorithm reported here and in some cases used it in their own ray tracers. The authors are also deeply grateful to the anonymous referees of an early version of this paper, whose constructive comments were invaluable in developing the present version.

References

LÖFSTEDT, M., AND AKENINE-MÖLLER, T. 2004. An evaluation framework for ray-triangle intersection algorithms. *Journal of Graphics, GPU, and Game Tools 10*, 2, 13–26. URL: http://www.tandfonline.com/doi/pdf/10.1080/2151237X.2005.10129195. 43

MÖLLER, T., AND TRUMBORE, B. 1997. Fast, minimum-storage ray-triangle intersection. *Journal of Graphics Tools 2*, 1, 21–28. URL: http://www.graphics.cornell.edu/pubs/1997/MT97.pdf. 40, 44

PHARR, M., AND HUMPHREYS, G. 2010. *Physically Based Rendering: From Theory to Implementation*, 2nd ed. Elsevier, Inc. URL: http://www.pbrt.org/. 40, 45

PHARR, M., AND HUMPHREYS, G., 2014. Scenes. Accessed July 1, 2015. URL: http://www.pbrt.org/scenes.php. 45

WALD, I., WOOP, S., BENTHIN, C., JOHNSON, G. S., AND ERNST, M. 2014. Embree: A kernel framework for efficient CPU ray tracing. *ACM Transactions on Graphics 33*, 4 (July). Article 143. URL: http://dl.acm.org/ft_gateway.cfm?id=2601199&ftid=1488952&dwn=1&CFID=636782622&CFTOKEN=40405023. 40, 43

Index of Supplemental Materials

The supplements to this article consist of the code, raw data, and data analysis for the comparisons reported above (see http://www.jcgt.org/published/0005/03/04/Supplements.zip), organized as follows:

- Folder "ExperimentCode". The complete C++ code for the stand-alone comparisons described in Section 4.1. Files include the main program ("main.cpp"), header and implementation files for five triangle classes whose `intersect` methods we timed (including a control class with only a dummy `intersect` method), header and implementation files for a superclass that those triangle classes share, and a handful of files of supporting definitions. File names and initial comments identify exactly what each file provides.

- Folder "PBRTCode". The "trianglemesh" files from PBRT modified to use our precomputed transformation method, as described in Section 4.2. Files "trianglemesh12.h" and "trianglemesh12.cpp" implement the straightforward version of our algorithm that stores all 12 coefficients of the transformation matrix. Files "trianglemesh9.h" and "trianglemesh9.cpp" implement the more space-efficient 9-coefficient version.

- File "ExperimentAnalysis.xlsx". A spreadsheet containing the raw data and calculations from the stand-alone comparisons. This is the analysis underlying Table 1.
- File "PBRTAnalysis.xlsx". A spreadsheet containing the raw data and calculations for comparing our algorithms to PBRT's original Möller and Trumbore algorithm. This is the analysis behind Table 2.

Author Contact Information

Doug Baldwin
Dept. of Mathematics
SUNY Geneseo
1 College Circle
Geneseo, NY, 14454
USA
(585) 245-5659
baldwin@geneseo.edu http://www.geneseo.edu/math/baldwin

Michael Weber
Dept. of Mathematics
SUNY Geneseo
1 College Circle
Geneseo, NY, 14454
USA
mjw1815@gmail.com

Doug Baldwin, Michael Weber, Fast Ray-Triangle Intersections by Coordinate Transformation, *Journal of Computer Graphics Techniques (JCGT)*, vol. 5, no. 3, 39–49, 2016
http://jcgt.org/published/0005/03/04/

Received:	2016-1-18		
Recommended:	2016-03-26	Corresponding Editor:	Matt Pharr
Published:	2016-09-27	Editor-in-Chief:	Marc Olano