GPU-Based Ray-Triangle Intersection Testing

Philippe C.D. Robert
robert@iam.unibe.ch

Daniel Schweri
daniel.schweri@bluewin.ch

Research Group on Computational Geometry and Graphics
Institute of Computer Science and Applied Mathematics
University of Bern, Neubrückstrasse 10, 3012 Bern, Switzerland

Abstract

We take advantage of recent improvements in graphics hardware, originally designed to increase the visual quality of rendered scenes, to study intersection testing algorithms on programmable graphics hardware. We implement two of the most commonly used algorithms in Nvidia Cg and compare the performance of our implementations with traditional, ANSI-C based variants to demonstrate the advantages and disadvantages of using programmable graphics hardware as general purpose coprocessor.

CR Categories and Subject Descriptors: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling; C.1.2 [Processor Architectures]: Multiple Data Stream Architectures

Keywords: Programmable Graphics Hardware, GPGPU, Intersection Testing

1 Introduction

A quite common and fundamental problem in computer graphics is how to perform stable intersection tests of rays with geometric primitives in minimal time. A multitude of graphics applications consequently depend on efficient implementations of such algorithms, in many cases limited to ray-triangle intersection, e.g. ray tracing, collision detection, picking operations and so on. Due to the vast evolution of graphics hardware in recent years it has become viable to address such problems using programmable graphics processing units (GPU). This has enabled many authors to demonstrate the raw power of GPUs in different fields of general purpose computations. Moreover, the development of specialised high-level shading languages, such as Nvidia Cg or the OpenGL Shading Language, facilitate the use of programmable vertex and fragment shader units for more complex applications.

1.1 Motivation

In this report we outline various procedures for mapping geometric algorithms to the GPU and demonstrate the advantages and disadvantes of such implementations. For this purpose we analyse multiple GPU based implementations of two ray-triangle intersection algorithms. We compare their runtime behaviour with their CPU based counterparts and discuss ways of further improving the execution performance of these methods.

1.2 Related Work

Because of its important role in computer graphics plenty of research has been done in the field of intersection testing algorithms. With regard to ray-triangle intersection testing, the algorithms proposed by Snyder and Barr [10], Badouel [1] and Moller [8] are perhaps the most influential ones, most notably because of their efficiency and elegance[1].

Recent efforts in using GPUs for general purpose computations have explicitely demonstrated that the computation power of graphics hardware may not only be used for common rendering related tasks but for a wider range of applications. Purcell *et al.* [9] point out that current programmable GPUs can be used perfectly as parallel stream coprocessors. Krüger and Westermann [6] describe a framework for the implementation of linear algebra operators on the GPU, providing a tool for designing more complex algorithms in the field of numerical simulation. In [3] and [2] Bolz *et al.* show that numerical computations can be performed efficiently on the GPU by implementing a sparse matrix conjugate gradient solver and a regular-grid multigrid solver on a GeForceFX, while Harris *et al.* [5] use graphics hardware to perform physically-based visual simulations. Based on Moller's algorithm, Carr *et al.* [4] implemented a fixed-point ray-triangle intersection testing engine on an ATI R200, whereas Purcell *et al* [9] developed a complete ray tracer on their own GPU simulator.

Others have contributed to this emerging topic, these works are available on the website related to general purpose computation on the GPU[2].

1.3 Organisation

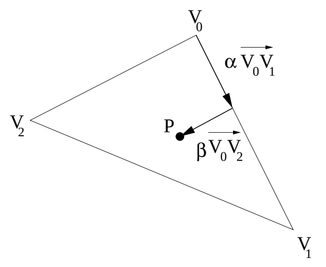
This report is structured as follows. Focused on general purpose computation we give a short overview of the capabilities of current generation graphics processing units in Section 2, followed by an introduction of the two intersection testing algorithms we used in Section 3. In Section 4 we describe our implementations, the performance results are then presented in Section 5. The conclusion are given in Section 6. Future work is indicated in Section 7.

2 Current Graphics Hardware

Modern graphics processing units are parallel, stream orientated processors, consisting of multiple programmable vertex and fragment processing units. They are featuring almost complete single-precision IEEE-754 floating-point arithmetic as well as advanced programmability including (limited) flow control, loops and a fully orthogonal instruction set optimised for vector processing.

It is notably interesting to use the fragment processing units as arithmetic coprocessors because they offer a much higher throughput than vertex units, and even more important, because they allow for direct texture memory access.

[1]Other algorithms do exist, but they are usually based on similar ideas and concepts.
[2]http://www.gpgpu.org



V_0
$\alpha \overrightarrow{V_0V_1}$
V_2
P
$\beta \overrightarrow{V_0V_2}$
V_1

Figure 1: Parametric representation of the ray-triangle intersection point P.

However current GPUs still impose a couple of restrictions which have to be taken into account when using them as coprocessors for general purpose computations. Among these are limited computation precision, the lack of integer and bit operations and poor memory management capabilities. Moreover it is not possible at all to use recursive functions on GPUs, and worse, fragment processors do not yet support true code branching. Another severe restriction imposed by the current generation of programmable graphics hardware is the limited program length. Unfortunately this limitation affects fragment shaders even more than vertex shaders. To give an example, the NV3x GPU supports fragment shader programs with a maximum length of 1024 instructions, while vertex shader programs can have a length of up to 65535 instructions.

3 Ray-Triangle Intersection Testing

For our comparative study we use two ray-triangle intersection testing algorithms, the first one is Badouel's approach [1], the second one is Moller and Trumbore's algorithm [8]. Both algorithms are known to be stable and highly efficient. Because we are mostly interested in performance related aspects of intersection testing, we will skip correctness validations and refer to the original publications instead. Consequently we omit Snyder and Barr's approach [10] altogether, for it is less efficient (in time), as it is shown in [1].

3.1 Badouel's Algorithm

The algorithm proposed by D. Badouel is similar to Snyder and Barr's earlier approach. It is split into two phases:

1. The ray is tested for intersection with the triangle's embedding plane, defined by the three vertices $V_i, i \in \{0, 1, 2\}$ of the triangle. Combining the parametric representation of the ray r and the implicit plane equation leads to

$$t = -\frac{d + N \cdot O}{N \cdot D} \tag{1}$$

   where O = ray origin, D = ray direction, N = normal of the embedding plane, $d = -V_0 \cdot N$ and $r(t) = O + Dt$.

   Based on the evaluation of the parameter t, the intersection is rejected if either the ray and the triangle are parallel ($N \cdot D = 0$), the intersection

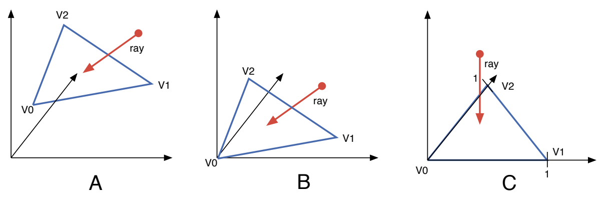


Figure 2: The transformations applied to the triangle and ray by the Moller-Trumbore algorithm.

point lies behind the origin of the ray ($t \leq 0$) or a closer intersection has already been found ($t > t_{nearest}$).

2. If the ray intersects with the embedding plane, the coordinates of the intersection point P are determined. As shown in Figure 1 point P can be expressed as

$$\vec{V_0P} = \alpha \vec{V_0V_1} + \beta \vec{V_0V_2} \tag{2}$$

Finally, the intersection point P is inside the triangle if

$$\alpha \geq 0, \beta \geq 0 \, and \, \alpha + \beta \leq 1 \tag{3}$$

For a more detailed derivation of the algorithm we refer to Badouel's original article [1].

3.2 Moller-Trumbore's Algorithm

The algorithm proposed by Moller and Trumbore does not test for intersection with the triangle's embedding plane and therefore does not require the plane equation parameters. This is a big advantage mainly in terms of memory consumption and – especially on the GPU – execution performance. The algorithm goes as follows:

1. In a series of transformations the triangle is first translated into the origin and then transformed to a right-angled unit triangle in the $y - z$ plane, with the ray direction aligned with x. Figure 2 illustrates the procedure. This can be expressed by a linear equation

$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \frac{1}{P \cdot E_1} \begin{pmatrix} Q \cdot E_2 \\ P \cdot T \\ Q \cdot D \end{pmatrix} \tag{4}$$

where $E_1 = V_1 - V_0$, $E_2 = V_2 - V_0$, $T = O - V_0$, $P = D \times E_2$ and $Q = T \times E_1$.

2. This linear equation can now be solved to find the barycentric coordinates of the intersection point (u, v) and its distance t from the ray origin.

Again we refer to the original article [8] for a more detailed explanation. Optimised variations of the original implementation can be found in Moller's follow-up article [7].

4 Implementation

Our code is solely based upon the original implementations of the two algorithms as described in [1] and [7], consequently we do not make use of SIMD extensions or any other substantial changes which might lead to better results on modern CPUs.

Our fragment shader implementations of the ray-triangle intersection algorithms were written using Nvidia's Cg shading language, therefore it was relatively easy to turn the original C code into fragment shader programs, see Appendix A for the code details. During this process several Cg language features have proven to be very useful, namely the native support for vector data types, basic vector arithmetic functionality through operator overloading and not to forget efficient implementations of more complex, mathematical functions[3] provided by the Cg Standard Library.

Unfortunately we also had to deal with a few shortcomings of current GPUs, most notably the absence of true branching and premature return statements. On the one hand this led to less efficient code compared to the C version, luckily we can safely assume that next generation GPUs will overcome these limitations in the near future. On the other hand, the code length restriction of fragment shader programs was not a problem in our case, the fragment shader code roughly consists of 50 instructions. For a detailed overview over Cg language features and graphics hardware limitations we refer to the *Cg Toolkit User's Manual* [4].

An important aspect of this project was to study potential benefits and drawbacks of different shader program integration models when using the GPU as streaming coprocessor. This includes the passing of parameters to as well as reading back results from the GPU. In the following subsections we describe the various parameter passing approaches we have implemented and benchmarked later on with both intersection testing algorithms.

4.1 Direct Parameter Passing

As with normal functions it is possible to set - and obtain - shader program parameters in a straight foward fashion, either using low-level OpenGL functionality or the Cg runtime API. In our case this leads to a massive communication overhead since for every intersection test the parameters have to be updated accordingly. This is not very convenient and obviously results in a serious performance degradation. We therefore implemented this mode solely for the sake of performing our benchmarks, it is hardly an option for real-world applications.

4.2 Indirect Parameter Passing

To make the best usage of the GPU as a high performance, general purpose coprocessor, it is important to cope with the stream orientated pipeline architecture of graphics hardware. Consequently it is crucial not to stall the graphics pipeline while rendering[5]. This is of course heavily affected by the way input parameters are passed to the shader program.

[3] E.g. `cross()` and `dot()`.

[4] Available as download from http://developer.nvidia.com/object/cg_toolkit.html

[5] Rendering means executing the intersection testing shader program in this context.

Indirect parameter passing is basically accomplished by encoding the various parameter values into one or more textures which are then used by our shader program to access the required input parameters. This promises to be much faster than direct parameter passing, though unfortunately there are texture size limitations which limit the number of parameters which can be sent to the GPU in one pass.

The results of the executed intersection tests are then written to a RGBA 32-bit floating-point Pbuffer, which can be read back to host memory as a contiguous memory block after performing the tests. We have implemented two versions of indirect parameter passing, which are explained in more details below.

4.2.1 Textures Only

Our first implementation uses 5 equally sized 2D FLOAT textures for parameter passing purposes. We store the 3 vertices of a triangle in 3 separate textures, whereas the additional 2 textures are used to store ray origin and direction parameters. See Figure 3 for more details.

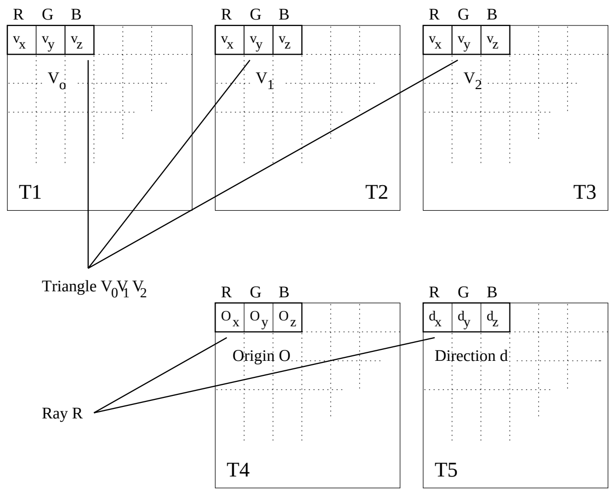


Figure 3: Parameter encoding using 5 2D FLOAT RGB textures. Note that all parameters are stored at the same position within the textures.

Consequently if a texture of size $n \times m$ is passed to the fragment shader, exactly $n \times m$ ray-triangle intersection tests can be performed simply by drawing an equally sized quad to the screen[6]. Doing so every rendered pixel corresponds to one interesection test. Moreover it is noteworthy that with this approach every passed ray is tested for intersection with only one of the passed triangles, namely the one which is stored at the corresponding texture coordinate. So on one side this leads to a great amount of flexibility, but on the other side this will add redundancy and memory transfer overhead when a ray has to be tested for intersection with many triangles, because in this case the ray has to be stored

[6]Note that our implementation uses square textures only.

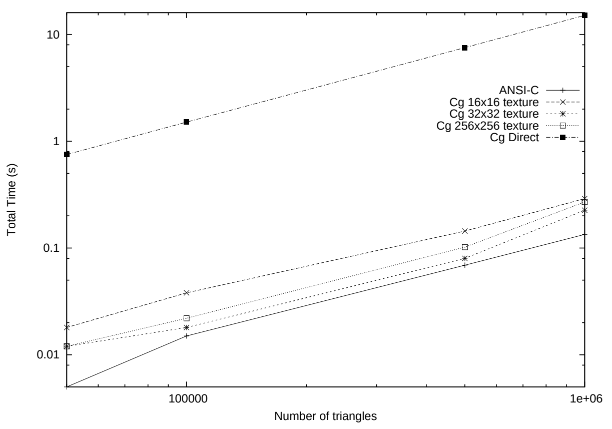


Figure 4: The nearly linear performance characteristics of the ANSI-C and Cg variants of the Moller-Trumbore algorithm. Note the logarithmic scale of the plot.

multiple times in the texture. This drawback can be avoided by using different texture lookup indices for rays and triangles.

4.2.2 Textures and Vertex Data Attributes

Our second implementation is a mixed mode implementation, inspired by Purcell's work [9]. We call it *mixed mode* because it uses 2 equally sized 2D FLOAT textures to hold the ray parameters, but in contrast to the previously described method triangles are sent through the geometry pipeline encoded as texture coordinates using 4 texture units. So the dimensions of the textures are encoded into the texture coordinates of texture unit 0, while the 3 other units are used to store the coordinates of the triangle vertices.

This approach leads to a true SIMD model where every triangle is tested for intersection with every ray stored in the texture and might thus be more suitable for real world applications, such as ray tracing.

5 Performance Results

We benchmarked our implementations on a 2.8 GHz Pentium IV running Red Hat Linux 9 using a NVIDIA NV35 GPU[7]. The C code was compiled with gcc 3.2.2, without support for SIMD extensions, such as SSE or 3DNow!. The Cg code was compiled with the Cg compiler release 1.1.

5.1 CPU Based Execution

As expected, the CPU versions for both algorithms show almost linear performances when testing different numbers of ray-triangle intersection, see Figure 4. We were able to perform roughly $8 * 10^6$ intersection tests per second using the algorithm proposed by Moller-Trumbore.

[7]Asus V9950 GeForce FX 9500, driver version 53.36

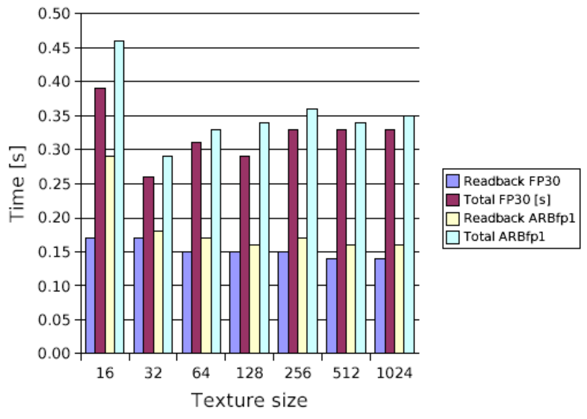


Figure 5: Testing 2^{20} triangles and rays for intersection using textures of varying dimensions. As expected, the fp30 render profile is in all cases superior to the OpenGL ARB fp1 profile, moreover memory readback proves to be a real performance killer.

5.2 GPU Based Execution

While implementing both algorithms in Cg we found that the GPU version of Badouel's algorithm is generally less efficient than Moller-Trumbore's approach, even though the algorithm itself is not particularly more complex. This has two main reasons:

1. Badouel's algorithm requires additional input parameters, namely the triangle's embedding plane parameters. Consequently they either have to be precomputed on the CPU and passed to the GPU or computed on the fly on the GPU. Obviously both approaches lead to additional overhead, compared to Moller-Trumbore's approach.

2. Badouel's implementation makes some clever usage of array programming to calculate the coordinates of $\vec{V_0P}$, $\vec{V_0V_1}$ and $\vec{V_0V_2}$, see Figure 1. While computing the array indices itself is not a big deal, it is not possible to map the array based code onto the GPU, simply because Cg does not yet support C-style arrays. As a consequence conditional code has to be used which is inefficient on current graphics hardware.

For these reasons we used an implementation of Moller-Trumbore's algorithm throughout all our performance benchmarks.

5.2.1 Direct Parameter Passing

Passing the parameters directly to the GPU leads to a linear performance, similar to the CPU based implementation. However, the execution speed on the GPU is drastically slower. This demonstrates the negative effect of frequent, unoptimised parameter passing on the total execution performance, resulting in approximately $7 * 10^4$ intersection tests per second using Moller-Trumbore's algorithm.

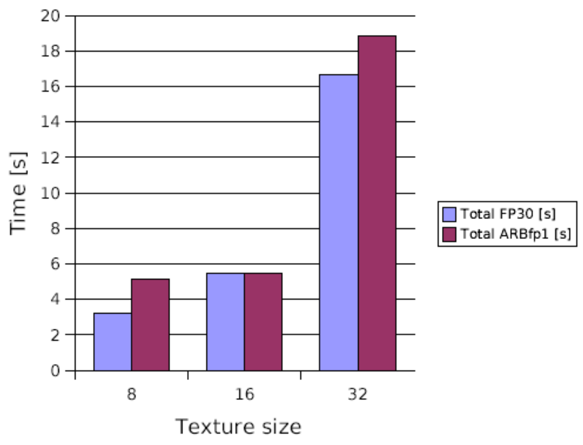


Figure 6: Mixed mode intersection testing of 100000 triangles with 64, 256 and 1024 rays, using both the native fp30 as well as the ARB fp1 render profile.

5.2.2 Parameter Passing using Textures

When passing parameters by texture the behaviour of the GPU based execution is still more or less linear, although different texture sizes will result in different execution results due to varying texture handling overhead.

It is obvious that using larger textures results in less frequent parameter transport passes, doubling the texture size leads to four times less texture uploads as well as framebuffer readbacks. Unfortunately the use of larger textures will eventually lead to a lot more texture cache misses on the GPU, which again has negative impacts on the execution performance.

It turned out that in our case and with our setup the optimal texture size was always 32×32 pixels, leading to a still disappointing peak performance of $4 * 10^6$ intersection tests per second. Figure 5 shows the performance results from computing $2^{20} = 1048576$ ray-triangle intersection tests using textures of varying dimensions. It should be mentioned that our implementation currently accepts only square sized textures, it would be interesting to see how different dimensions in x and y impact the texture cache efficiency on the GPU.

5.2.3 Mixed Mode Parameter Passing

The mixed mode setup lead to the best GPU based performance values, see Figure 6 for the details. But as with all other setups the readback bandwidth limitation makes it impossible to accelerate ray-triangle intersection rigorously by using the GPU as coprocessor in a straightforward fashion. Neglecting this limitation we were able to perform $8 * 10^6$ intersection tests per second on our GPU, which is more or less on the same performance level as the CPU based version, but still noticeably slower than eg. Purcell's results, obtained on a fixed-point GPU.

5.3 Technical Aspects

For our comparisons we used two fragment program profiles, namely Nvidia's native fp30 profile and the OpenGL ARB fp1 equivalent. As expected on our graphics hardware, Nvidia's fp30 profile, which is based on the proprietary `NV_fragment_program` OpenGL extension, was faster than the ARB fp1 profile.

In order to optimise the memory readback performance we furthermore adopted the OpenGL `pixel_data_range` extension, which permits the driver to allocate physically contiguous PCI memory or cachable AGP memory, depending on the performance characteristics of the device and thus should lead to noticeably better performance results. Surpisingly though this did not happen, we got only minor performance improvements.

6 Conclusion

We presented several implementations of efficient ray-triangle intersection tests on the GPU using the Nvidia Cg shading language. We showed that the raw power of GPUs itself does not guarantee superior performance compared to CPU based execution. Instead several key aspects have to be taken into account when using the GPU as a coprocessor in order to achieve acceptable performances. It is especially important to adapt the code to the streaming architecture of programmable GPUs, porting an algorithm to the GPU essentially means using a stream based programming model.

By porting the algorithms to the GPU using Nvidia Cg we were able to get initial results with little effort, but it is noteworthy that without careful tuning of the Cg code performances are not optimal. We were able to show that clever instruction reorderings can result in measurable speed gains.

We furthermore demonstrated that parameter passing and particularly memory readback has a great influence on the overall performance and thus is a major obstacle when using the GPU as coprocessor. The current AGP 8x standard allows for a 2.1 GB/s memory transfer to the card and roughly 250 MB/s from the card to the host. This bandwidth limitation is a serious bottleneck for general purpose computations on the GPU. Our observations confirmed that especially memory readback transfer is a performance killer and should be avoided whenever possible. Consequently the mixed mode implementation clearly proved to be the fastest approach, although it did not match CPU based performance due to the slow readback transfer.

Another limitation we had to deal with is that GPUs only support 32-bit floating-point precision or less[8], moreover it seems that using 32-bit precision even may lead to performance penalties on Nvidia NV3x hardware. Consequently if 64-bit floating-point precision or higher is a requirement then GPU based execution is not a viable option.

Finally, we found that Moller-Trumbore's algorithm was superior to Badouel's approach because of technical peculiarities with respect to GPU programming. We demonstrated that the GPU performance is well capable to handle general purpose computation tasks, but the AGP bus is not. We expect though that

[8]ATI GPUs deliver 24-bit floating-point precision only.

the forthcoming PCI Express bus[9] will soon leverage general purpose computation on the GPU as it promises to eliminate the memory transfer bottleneck altogether.

7 Future Work

There are numerous possibilities for follow-up research efforts, among these we see two areas which promise to be of some interest for future applications.

First of all we would like to implement more complex algorithms on the GPU to make better usage of the powerful arithmetic instruction set of current and next generation graphics hardware. This also includes multipass implementations as well as other strategies which help minimising the memory readback bottleneck, which will hopefully lead to better performance results.

Second, it is important to further exploit concurrent programming techniques to scale real world applications using GPUs as streaming coprocessors. Among these are better parallelism and sophisticated time multiplexing strategies, for example.

Last but not least it would be interesting to compare various aspects of NVIDIA's Cg shading language with the OpenGL Shading Language.

[9]PCI Express is the designated successor of the AGP and PCI bus technologies.

References

[1] Didier Badouel. An efficient ray-polygon intersection. pages 390–393, 1990.

[2] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schrder. The gpu as numerical simulation engine.

[3] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. *ACM Trans. Graph.*, 22(3):917–924, 2003.

[4] Nathan A. Carr, Jesse D. Hall, and John C. Hart. The ray engine. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 37–46. Eurographics Association, 2002.

[5] Mark J. Harris, Greg Coombe, Thorsten Scheuermann, and Anselmo Lastra. Physically-based visual simulation on graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 109–118. Eurographics Association, 2002.

[6] Jens Krüger and Rüdiger Westermann. Linear algebra operators for gpu implementation of numerical algorithms. *ACM Trans. Graph.*, 22(3):908–916, 2003.

[7] Tomas Möller. Practical analysis of optimized ray-triangle intersection.

[8] Tomas Möller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *J. Graph. Tools*, 2(1):21–28, 1997.

[9] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21(3):703–712, July 2002.

[10] John M. Snyder and Alan H. Barr. Ray tracing complex models containing surface tessellations. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 119–128. ACM Press, 1987.

A Cg Code

The straight Cg port of the Moller-Trumbore ray-triangle intersection testing implementation, which uses textures only for parameter passing, highly resembles its C pendant.

```
#define EPSILON 0.000001

struct vertex_out_fragment_in {
        float4 position         : POSITION; // not used for fragment shader
        float2 texCoord0        : TEXCOORD0;
        float4 color            : COLOR0;
};

float4 intersect(float3 orig, float3 dir, float3 v0, float3 v1, float3 v2)
{
    float  t, u, v;
    float3 edge1, edge2, tvec, pvec, qvec;
    float  det;
    float  inv_det;
    bool   intersection = true;

    /* find vectors for two edges sharing v0 */
    edge1 = v1 - v0;
    edge2 = v2 - v0;

    /* begin calculating determinant - also used to calculate U parameter */
    pvec = cross(dir, edge2);

    /* if determinant is near zero, ray lies in plane of triangle */
    det = dot (edge1, pvec);

    if (det > -EPSILON && det < EPSILON)
       intersection = false;

    /* calculate distance from v0 to ray origin */
    inv_det = 1.0 / det;
    tvec = orig - v0;

    /* calculate U parameter and test bounds */
    u = dot (tvec, pvec) * inv_det;

    if (u < 0.0 || u > 1.0)
       intersection = false;

    /* prepare to test V parameter */
    qvec = cross(tvec, edge1);

    /* calculate V parameter and test bounds */
    v = dot (dir, qvec) * inv_det;

    if (v < 0.0 || u + v > 1.0)
       intersection = false;
    /* calculate t, ray intersects triangle */
    t = dot (edge2,qvec) * inv_det;

    float4 ret_val = float4(t,u,v,0.0);
    if (intersection)
        ret_val.w = 1.0;

    return ret_val;
}

float4 main( vertex_out_fragment_in  IN,
             uniform samplerRECT vertices0   : TEXUNIT0,
             uniform samplerRECT vertices1   : TEXUNIT1,
             uniform samplerRECT vertices2   : TEXUNIT2,
             uniform samplerRECT ray_origins : TEXUNIT3,
             uniform samplerRECT ray_dir     : TEXUNIT4) : COLOR
{
    float3 v0     = texRECT(vertices0, IN.texCoord0).rgb;
    float3 v1     = texRECT(vertices1, IN.texCoord0).rgb;
    float3 v2     = texRECT(vertices2, IN.texCoord0).rgb;
    float3 origin = texRECT(ray_origins, IN.texCoord0).rgb;
    float3 dir    = texRECT(ray_dir, IN.texCoord0).rgb;

    return intersect(origin, dir, v0, v1, v2);
}
```