Optimizing Ray-Triangle Intersection via Automated Search

Andrew Kensler Peter Shirley

School of Computing
University of Utah

ABSTRACT

In this paper, we examine existing direct 3D ray-triangle intersection tests (i.e., those that do not first do a ray-plane test followed by a 2D test) for ray tracing triangles and show how the majority of them are mathematically equivalent. We then use these equivalencies to attempt faster intersection tests for single rays, ray packets with common origins, and general ray packets. We use two approaches, the first of which counts operations, and the second of which uses benchmarking on various processors as the fitness function of an optimization procedure. Finally, the operation-counting method is used to further optimize the code produced via the fitness function.

Keywords: determinants, ray tracing, triangles

1 INTRODUCTION

Ray-object intersection is one of the kernel operations in any ray tracer [10], and a different function is implemented for each type of geometric primitive. Triangles are one of the most ubiquitous rendering primitives in use. They typically find use as a "lowest common denominator" between modelers and renderers, due to their simplicity, uniformity and the ease of tessellating more complex primitives into triangles. Many renderers even use triangles as their sole primitives for these reasons. Thus, high performance when rendering triangles is a key feature in nearly every renderer.

There are two basic classes of ray-triangle tests commonly in use (see [6] for a thorough list and empirical comparison for single ray tests). The first intersects the ray with the plane containing the triangle, and then does a 2D point-in-triangle test in the plane of the triangle (e.g. [9]). The second does a direct 3D test based on some algebraic or geometric observation such as provided in Cramer's rule, triple products, ratios of determinants, or Plücker coordinates (e.g., [1]). This paper examines only the direct 3D test, and observes that "under the hood" these methods are all taking ratios of volumes, and differ mainly in what volumes are computed.

For these 3D methods we optimize ray-triangle intersection in two different ways. First we do explicit operation counting for the cases of single rays, packets of rays with common origins, and general packets of rays. Next we do code evaluation by letting a genetic algorithm modify the code using profiling on various computers as a fitness function. The implementation is based on SIMD and ray packets to improve the chances of relevance for modern implementations.

2 BACKGROUND: SIGNED VOLUMES

The signed area of the parallelogram shown in the left of Figure 1 is given by

$$\text{area} = \begin{vmatrix} x_a & x_b \\ y_a & y_b \end{vmatrix}.$$

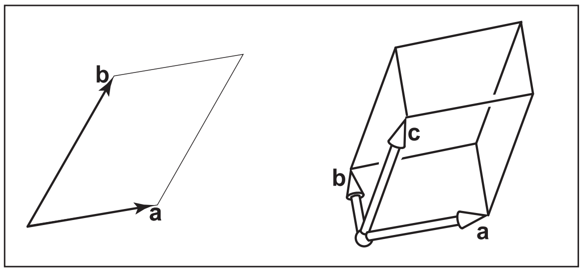

Figure 1: The signed area/volume of these objects are given by determinants with the Cartesian coordinates of the vectors as matrix rows or columns. The sign of each of these examples is positive via right hand rules.

If we were to switch **a** and **b**, the sign would change. The sign is positive when the second vector is in the counterclockwise direction from the first. There is a similar signed volume rule for parallelepipeds such as the one shown in the right of Figure 1:

$$\text{volume} = \begin{vmatrix} x_a & x_b & x_c \\ y_a & y_b & y_c \\ z_a & z_b & z_c \end{vmatrix}.$$

This volume is positive if the vectors form a right-handed basis, and negative otherwise. The volume of the tetrahedron defined by the three vectors is one-sixth that of the parallelepiped's.

The volume formula can be used to compute 2D triangle area by embedding the triangle in 3D with the three vertices on the $z = 1$ plane as shown in Figure 2:

$$\text{triangle area} = \frac{1}{2}\begin{vmatrix} x_0 & x_1 & x_2 \\ y_0 & y_1 & y_2 \\ 1 & 1 & 1 \end{vmatrix}. \tag{1}$$

The reason for the first one-half is that the area of the triangle is three times the volume of the tetrahedron defined by the three column vectors in the matrix, and the determinant is six times the volume of that tetrahedron. We can also use the determinant rule to observe:

$$\text{triangle area} = \frac{1}{2}\begin{vmatrix} x_1 - x_0 & x_2 - x_0 \\ y_1 - y_0 & y_2 - y_0 \end{vmatrix}.$$

This second (2D) determinant is the area of the parallelogram defined by the two 2D edge vectors of the triangle, and has the same value as the determinant in Equation 1, although this is not algebraically obvious. This is an example of why interpreting determinants as area/volume computations can be better, especially for geometric thinkers.

The volume of a tetrahedron defined by four vertices $\mathbf{p}_i$ can be found by taking the determinant of three of the vectors along its edges, or by a 4D determinant on a $w = 1$ plane:

$$\text{volume} = -\frac{1}{6}\begin{vmatrix} x_0 & x_1 & x_2 & x_3 \\ y_0 & y_1 & y_2 & y_3 \\ z_0 & z_1 & z_2 & z_3 \\ 1 & 1 & 1 & 1 \end{vmatrix} = \frac{1}{6}\begin{vmatrix} x_1 - x_0 & x_2 - x_0 & x_3 - x_0 \\ y_1 - y_0 & y_2 - y_0 & y_3 - y_0 \\ z_1 - z_0 & z_2 - z_0 & z_3 - z_0 \end{vmatrix}.$$

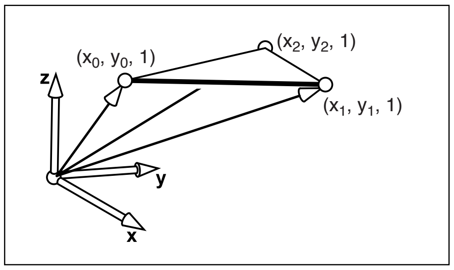

Figure 2: The area of the triangle can be computed from the volume of the parallelepiped determinants with the Cartesian coordinates of the vectors as matrix rows or columns. The sign of each of these examples is positive via right-hand rules.

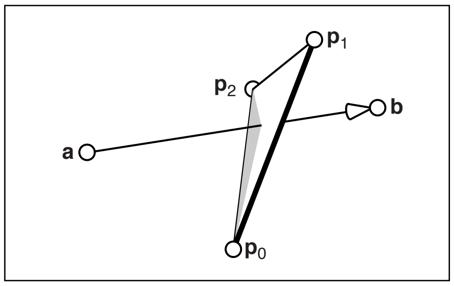

Figure 3: Geometry for a ray edge test

The minus sign before the first determinant is not a mistake. Some care must be taken on the ordering rules for different matrix forms in the various dimensions; the odd dimensions have a sign change between the edge-vector based method and the $w = 1$ hypervolume method.

There are two other ways by which signed volumes are often computed in 3D. The first is the triple product (equivalent to a determinant in 3D):

$$\text{volume} = \frac{1}{6}\left[(\mathbf{p}_1 - \mathbf{p}_0) \times (\mathbf{p}_2 - \mathbf{p}_0)\right] \cdot (\mathbf{p}_3 - \mathbf{p}_0).$$

Another method for computing a signed volume uses the Plücker inner product for the directed line segments $\mathbf{p}_0\mathbf{p}_1$ and $\mathbf{p}_2\mathbf{p}_3$. This is algebraically the same as the determinant and triple product methods [4].

3 Using signed volumes for intersection

The basic signed volume idea has been used by several researchers for ray-triangle intersection, and the equivalence between Plücker inner products, triple products, and determinants for intersection has been pointed out by O'Rourke [8]. For example, consider the configuration in Figure 3. The signed volume of the tetrahedron $\mathbf{abp}_2\mathbf{p}_0$ is given by:

$$V_1 = \frac{1}{6}\left[(\mathbf{p}_2 - \mathbf{a}) \times (\mathbf{p}_0 - \mathbf{a})\right] \cdot (\mathbf{b} - \mathbf{a}).$$

If this sign is negative, then the ray is to the "inside" side of the edge. The magnitude of V_1 is proportional to the area of the shaded triangle. Similarly, areas V_0 and V_2 can be computed with respect to the edges opposite $\mathbf{p}_0$ and $\mathbf{p}_2$ (see Figure 4). If all three V_i are the same sign, then the infinite line through $\mathbf{a}$ and $\mathbf{b}$ hits the triangle. The barycentric coordinates can also be recovered. For example:

$$\alpha = \frac{V_0}{V_0 + V_1 + V_2}, \;\; \beta = \frac{V_1}{V_0 + V_1 + V_2}.$$

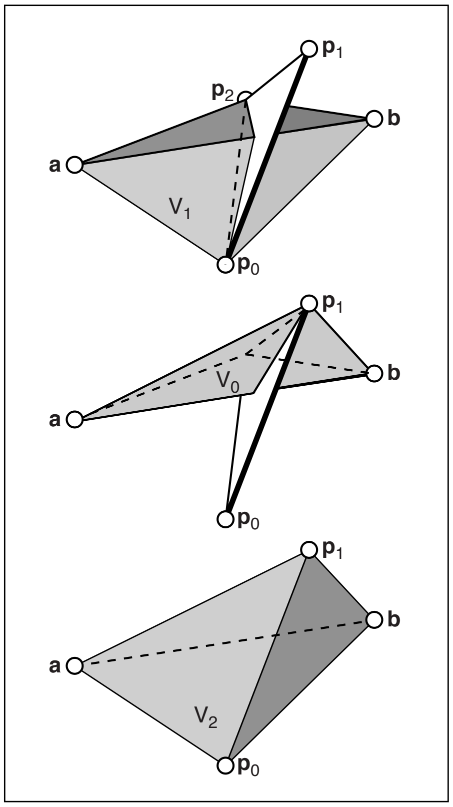

Figure 4: Geometry for a ray edge test

The segment hits the triangle if the signed volume of the tetrahedron $\mathbf{p}_0\mathbf{p}_1\mathbf{p}_2\mathbf{a}$ and $\mathbf{p}_0\mathbf{p}_1\mathbf{p}_2\mathbf{b}$ have the opposite signs. If these volumes are V_a and V_b, and V_a is positive, then the ray parameter is given by:

$$t = \frac{V_a}{V_a - V_b}.$$

Note that you could also compute the volume of $V_0 + V_1 + V_2$ directly:

$$V = V_0 + V_1 + V_2 = \frac{1}{6}\left[(\mathbf{p}_1 - \mathbf{p}_0) \times (\mathbf{p}_2 - \mathbf{p}_0)\right] \cdot (\mathbf{b} - \mathbf{a}).$$

Note that that is the denominator in Cramer's Rule test, which under-the-hood is computing volumes.

If volumes are to be used, there are several degrees of freedom which can be exploited to yield different tests. For example, one can compute the inside/outside test for the whole ray in several ways:

1. compute V_0, V_1, V_2, test for same sign;
2. compute $\alpha = V_0/V$, $\beta = V_1/V$, $\gamma = 1 - \alpha - \beta$, test for all in $[0, 1]$.
3. compute $\alpha = V_0/V$, $\beta = V_1/V$, $\gamma = V_2/V$, test for all in $[0, 1]$.

In addition, each of the volumes can be computed via several different edge tests. For example, the volume V_1 has six edges, any of which can be followed in either direction. Any three edges that are not all mutually coplanar will yield the same volume, though possibly with the opposite sign. For a volume defined by 4 points, there are 384 unique ways to compute the same signed volume. Given 3 points and a direction vector, there are 36 ways to compute the same signed volume. The one above allows a ray packet to precompute the cross product if the ray origin is shared. This may or may not be useful for sharing computations (i.e. subexpressions).

For the ray parameter test, $V = V_a - V_b$ is in fact the same V as above. Overall, a test must directly compute at least two of V_0, V_1, V_2, and at least one of V_a and V_b. Finally one of the remaining three volumes must be computed directly.

4 Minimizing total operation counts

In this section we try to use the equations that minimize the total number of operations. Because of the large number of possible equations, we use a brute force searching method to examine all cases. In the next section we use a more sophisticated and empirical method to optimize performance on real processors.

Volume-based triangle tests require the computation of at least four volumes for a successful intersection. These are generally either one of V_0 or V, plus V_1, V_2 and V_a. The exhaustive search considered every possible set of four scalar triple products to compute these volumes and for each of these sets, the total number of floating point operations, taking into account common subexpressions.

For possibilities, our program makes a list of the cost in number of arithmetic operations associated with each subexpression. For the example expression $c = ((\mathbf{p}_1 - \mathbf{p}_0) \times (\mathbf{p}_0 - \mathbf{p}_2)) \cdot (\mathbf{p}_o - \mathbf{a})$, $(\mathbf{p}_1 - \mathbf{p}_0)$, $(\mathbf{p}_0 - \mathbf{p}_2)$, $(\mathbf{p}_0 - \mathbf{a})$ count as three subtractions each. $((\mathbf{p}_1 - \mathbf{p}_0) \times (\mathbf{p}_0 - \mathbf{p}_2))$ counts as 6 multiplies and 3 subtractions (since $\mathbf{p}_1 - \mathbf{p}_0$ and $\mathbf{p}_0 - \mathbf{p}_2$ will already have been counted). And the whole expression for c costs 3 multiplies and 2 additions since, again, the subexpressions for the dot product were counted elsewhere.

With these lists in hand, the program looks at every combination of scalar triple products for the volumes. For example, if the ray stores $\mathbf{d} = \mathbf{b} - \mathbf{a}$, it would examine:

$$\begin{aligned}
V\; &= (\mathbf{d} \times (\mathbf{p}_2 - \mathbf{p}_0)) \cdot (\mathbf{p}_1 - \mathbf{p}_0) \\
V_a &= ((\mathbf{a} - \mathbf{p}_0) \times (\mathbf{p}_1 - \mathbf{p}_0)) \cdot (\mathbf{p}_2 - \mathbf{p}_0) \\
V_1 &= (\mathbf{d} \times (\mathbf{p}_2 - \mathbf{p}_0)) \cdot (\mathbf{a} - \mathbf{p}_0) \\
V_2 &= ((\mathbf{a} - \mathbf{p}_0) \times (\mathbf{p}_1 - \mathbf{p}_0)) \cdot \mathbf{d}
\end{aligned} \tag{2}$$

while checking every combination of scalar triple products for calculating each of the four volumes (i.e., 36*384*36*36 possibilities).

So for each of these combinations of expressions, it counts the number of operations, but duplicates are only counted once each. With the example above, it would count $(\mathbf{p}_2 - \mathbf{p}_0)$ once for 3 subtractions, but only one time (not 3), and $\mathbf{d} \times (\mathbf{p}_2 - \mathbf{p}_0))$ only once (not twice) for 6 multiplies and 3 subtractions, etc. Given this, it counts up a total number of arithmetic operations under the assumption that all subexpressions are computed once and then the results are saved. The result of this is a list of sets of each set of expressions for the volumes with the lowest cost found.

The program also had a few switches to consider different cases. These mainly affected how the cost was computed. For example, for packets, any subexpression that does not involve **d** or **a** is independent of the ray, and counts at 1/64th the normal cost (i.e., as though it were amortized over an 8x8 packet.) The exact divisor does not matter hugely since the total flops in the best expressions sets already total well below 64. The sum of these amortized computations in these best cases never totals above 1.0, which means that it will not cause it to beat out cases where it does not choose to amortize. But the fraction does serve as a tie-breaker to get it to minimize the amount of per-triangle precomputation that it does.

For the general case of single rays and choosing to find V instead of V_0, there were six optimal formulations requiring a total of 47 operations. Three of these were symmetric cases of the other three with triangle edges reversed and appropriate adjustments made to preserve signs. One of these formulations corresponds to the Möller-Trumbore algorithm [7], and was already given in Equations 2. When V_0 is used instead of V, the case is similar and there are still just six analogous best formulations, each requiring 47 operations.

With ray packets, however, all computations involving only the triangle vertices can be amortized over all of the rays in the packet. Assuming that the number of rays in the packet is large enough that all computations that can be amortized over the packet are essentially “free” (though not with zero cost), and that we again choose to use V instead of V_0, there were exactly two optimal formulations, each symmetric with the other:

$$\begin{aligned}
V &= ((\mathbf{p}_1 - \mathbf{p}_0) \times (\mathbf{p}_0 - \mathbf{p}_2)) \cdot \mathbf{d} \\
V_a &= ((\mathbf{p}_1 - \mathbf{p}_0) \times (\mathbf{p}_0 - \mathbf{p}_2)) \cdot (\mathbf{p}_0 - \mathbf{a}) \\
V_1 &= ((\mathbf{p}_0 - \mathbf{a}) \times \mathbf{d}) \cdot (\mathbf{p}_0 - \mathbf{p}_2) \\
V_2 &= ((\mathbf{p}_0 - \mathbf{a}) \times \mathbf{d}) \cdot (\mathbf{p}_1 - \mathbf{p}_0)
\end{aligned}$$

This formulation requires just over 32 operations per ray plus the amortized per-triangle operations. Note that the per-triangle computation simply involves finding two edges plus the unscaled normal of the triangle.

If all of the rays in the packet share a common origin, as is typical for primary rays and shadow rays for point light sources, it is possible to do far better yet. For these cases, all computations involving only the triangle vertices and **a** are amortized over the packet. There are twelve optimal formulations (six being symmetrical with the other six), and requiring just over fifteen operations per ray to find the volumes in the inner loop. At this point, V_a may be amortized entirely as well as all of the cross products, leaving only the three dot products:

$$\begin{aligned}
V &= ((\mathbf{p}_1 - \mathbf{p}_0) \times (\mathbf{p}_0 - \mathbf{p}_2)) \cdot \mathbf{d} \\
V_a &= ((\mathbf{p}_1 - \mathbf{p}_0) \times (\mathbf{p}_0 - \mathbf{p}_2)) \cdot (\mathbf{p}_0 - \mathbf{a}) \\
V_1 &= ((\mathbf{p}_0 - \mathbf{p}_2) \times (\mathbf{p}_0 - \mathbf{a})) \cdot \mathbf{d} \\
V_2 &= ((\mathbf{p}_1 - \mathbf{p}_0) \times (\mathbf{p}_0 - \mathbf{a})) \cdot \mathbf{d}
\end{aligned}$$

When V_0 is used instead of V, the case is similar and there are still just twelve analogous best formulations, each requiring fifteen operations. This property has been used to advantage by Benthin in his dissertation, although he derived it through Plücker coordinates. [2]

5 A Genetic Algorithm for Improved Performance

While operation counts are correlated to performance, the increasingly complex hardware and compiler technology makes optimization largely an empirical process. Since the number of possibilities is so large for how code can be written, exhaustive search by hand is not a good option. In this section we use genetic algorithms to improve our choices among coding options in a spirit similar to that shown effective for sorting [5].

Before applying any genetic search, we first formalize the space of choices we have. For example, we can compute V_0, V_1, and V_2 and derive V, or we could compute V_0, V_1 and V derive V_2. Another option is whether to test for early exit if a given variable is outside its allowed range. This test must of course come after that variable is computed. On the other hand, some quantities can be computed in any order. This suggests a dependency graph.

For the ray/triangle intersection tests, we used a dependency graph with 1294 nodes. The allowed parameter space included all legitimate choices for the signed-volume computations for the t-value, V, V_0, V_1, and V_2, the choice between computing V directly with a single signed volume computation or by summation of the three, how long to postpone the division, whether to use a barycentric in/out test or to test in/outness by comparing the signs of the volumes, whether and where to use early exits if all four rays in an SSE bundle are known to miss, etc.

The genetic algorithm approach used to sort among these options consists of three parts:

- the main genetic algorithm driver,
- the benchmark,
- the code generator.

The main genetic algorithm implementation is an evolution algorithm very similar to that used by Li et al. [5]. In this, the best individuals in each generation survive to the next generation entirely unchanged. Genetic recombination applies only to creating the new offspring to replace the least fit individuals. These are also subject to occasional random mutations. As with their system, we used a population of 50 individuals through 100 generations. At each generation, the 20 most fit were kept unchanged while the 30 least fit were replaced with pairs of offspring created through recombination from two parents with a two-point crossovers from the parent generation (a random middle segment from one parent is replaced with those values in the order that they appear in the other parent, and vice versa, to produce a pair of offspring that are still permutations. The reason that genomes must be permutations is due to the way they control code emission and is explained later.) The parents were chosen with probability proportional to their fitness. After this, two mutations were applied to the offspring at random, by swapping a random pair of values in their genetic sequence.

After this, the new generation is evaluated for fitness, which in this case consists of using each genome to output code for a ray/triangle intersection test combined with a benchmark for speed. The created program consists of a fixed, handwritten template for the benchmark with the genetically derived intersection test inserted into the template. This is compiled and executed, and the measured speed in millions of intersections per second becomes the fitness value for that individual. The benchmark code measures the performance of 20000 random triangles each intersected by 400 packets of 64 random rays each. The positions of the rays and triangles are chosen such that the intersection probability is approximately 25%. This probability was chosen to mimic the case for a good acceleration structure where rays that reach the point of intersecting a triangle have a high probability of success. 50% is a best case for this due to the typical tessellation of quads into pairs of triangles, where each triangle in the pair will typically have significantly overlapping bounding boxes but only a 50% or so chance of hitting, once a ray reaches the bounding box.

The code generation from the genomes is the most complex part of our process and is inspired by the approach in Fang et al. [3]. Each individual's genome encodes the algorithm as a permutation of the first 1294 natural numbers. A DAG of dependencies, starting with a goal node gives the list of possible code chunks to output (generally at the level of a single scalar or vector operation) along with any dependencies that must be satisfied before the chunk can be output. These dependencies take two forms: "required" and "optional" dependencies. For each chunk, all required dependencies must be satisfied before the a statement can be output, while only one or more of the optional dependencies needs to be. This distinction means that any generated program that satisfies this dependency graph will have the freedom to choose from alternative code paths where necessary, but will also be constrained to always generate legal programs which will compile and execute correctly. For example, computing a barycentric coordinate may be done through any of the numerous choices for computing the signed volume, but an early exit test based on the coordinate always requires the coordinate computation as a prerequisite.

Output from this dependency graph is guided by each individuals genome. The genome, as a permutation of the whole numbers, gives the priority for each node in the dependency graph. Code is emitted by applying a modified topological sort to the dependency graph where ties for which statement to emit next are broken according to the priority given in the genome. If an optional dependency has not already been satisfied due to another node, the optional dependency with the highest priority is chosen. An initial depth-first walk of the dependency graph from the goal node marks live nodes, so that only these are considered for output during the topological sort. Thus, so long as each genome remains a proper permutation of the first n natural numbers, where n in this case is 1294 – the number of nodes in the dependency graph – the code generator will always emit a valid and nearly minimal code sequence for it. The genetic algorithm still has tremendous freedom in choosing the relative order of the statements, and through careful encoding in the dependency graph nearly any choices for valid code may be given to the genetic algorithm.

We ran the GA code both for general and common origin packets. We implemented the code in C++ with SSE extensions. The best program for both packet conditions was then hand optimized making minor performance improvements.

The hand tuning was quite minimal. We examined the code from the GA to determine what choices it made for how to compute the signed volumes. Then, we examined the list of optimal operation-count expressions from the exhaustive search in the previous section and found the most similar set of expressions to that from the GA. We then changed the code from the GA to use the expressions from the optimal search, trying to change the code and especially the basic structure as little as possible. Typically this involved reversing the direction of an edge here and changing the operands for a dot or cross product there. Next we cleaned up the dead code left over from the previous step, since taking better advantage of common subexpressions meant that some of the former computations were now extraneous. Lastly, we cleaned up the artifacts from the GA – for example, as the final SIMD mask is the result of ANDing the masks from several tests, and these may be done in any order, the mask is initially set to all true before being ANDed with the first test. The obvious optimization, however, is to initialize the mask to the result of the first test. There were one or two similar cases where artifacts from the GA could be cleaned up by the compiler's optimizer. We simply applied the same transformations to streamline source. Overall, the changes we made were quite mechanical and not large.

The code from the GA and the hand-improved code were tested against a direct ray packet and SIMD adaptation of Möller-Trumbore test as indicated in Table 1. As can be seen, significant speedups are possible. The code for the GA+, along with the testing code, for general packets, is shown in the Appendix.

6 Conclusion

We have presented two methods for optimizing ray triangle intersection. Both of these differ from most previous approaches in that they are targeted toward implementations with ray packets. The first is based on simple operation counts. The second uses a more empirical approach and is probably more practical given the complexities of modern processors and compilers. In addition, the second uses knowledge from the first to improve performance further. An interesting question is whether the genetic algorithm approach can be extended to other components of ray tracing programs. Another question is whether the direct 3D approach examined here is not as efficient as the hit plane and 2D approach.

Prog	GCC402/Opt/Opt	GCC402/P4/Opt	ICC90/P4/Opt	GCC335/X/X	ICC90/P4/X	GCC402/P4/P4	ICC90/P4/P4	GCC401/C/C	Average
GA	158.665	115.509	135.386	158.838	163.386	97.072	167.825	81.561	134.780
GA+	164.707	141.652	158.816	172.978	180.265	102.610	190.968	106.279	152.284
GA (co)	201.202	158.817	182.153	189.348	207.289	173.531	205.890	112.827	178.882
GA+ (co)	190.867	182.115	203.336	216.120	228.410	158.043	229.335	125.212	191.680
MT	82.036	71.545	104.517	89.035	124.728	66.948	115.592	53.593	88.499

Table 1: Performance numbers are millions of ray/triangle intersections per second. Top two are for general packets, with GA+ being the hand-improved version. Fourth and fifth are for shared origin. MT is Möller-Trumbore. GCC402/Opt/Opt, etc. = Compiler / Compiler code gen. and opt. target / Test platform. Opt = 2.4GHz Dual Core Opteron (One core used). X = 3.2GHz Dual Core Xeon (One core used). P4 = 3.0GHz Pentium 4, Canterwood. C = 1.83GHz Core Duo (One core used).

A GA+ GENERAL PACKET CODE

This annotated code shows our best performing triangle code for general packets, and shows in detail how we tested its performance.

```
#include <mmintrin.h>
#include <xmmintrin.h>
#include <emmintrin.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include <fstream>
#include <iostream>
using namespace std;
static const int packet_size = 64;
static const int number_of_packets = 400;
static const int number_of_triangles = 20000;
static const float eye_range = 3.0f;
static const float target_range = 0.6f;
static const float ray_jitter = 0.04f;
// Triangle vertex positions
float p0xf[number_of_triangles];
float p0yf[number_of_triangles];
float p0zf[number_of_triangles];
float p1xf[number_of_triangles];
float p1yf[number_of_triangles];
float p1zf[number_of_triangles];
float p2xf[number_of_triangles];
float p2yf[number_of_triangles];
float p2zf[number_of_triangles];
// Ray origins, directions and best t-value
float oxf[number_of_packets][packet_size];
float oyf[number_of_packets][packet_size];
float ozf[number_of_packets][packet_size];
float dxf[number_of_packets][packet_size];
float dyf[number_of_packets][packet_size];
float dzf[number_of_packets][packet_size];
float rtf[number_of_packets][packet_size];
int main(int argc, char **argv) {
  int seed_time = time(0);
  unsigned short seeds[] = {
    static_cast<unsigned short>(seed_time & 0xffff),
    static_cast<unsigned short>((seed_time >> 8) & 0xffff),
    static_cast<unsigned short>((seed_time >> 16) & 0xffff) };
  seed48(seeds);
  // Setup tests with random triangles and packets
  for (int ti = 0; ti < number_of_triangles; ++ti) {
    p0xf[ti] = drand48() - drand48();
    p0yf[ti] = drand48() - drand48();
    p0zf[ti] = drand48() - drand48();
    p1xf[ti] = drand48() - drand48();
    p1yf[ti] = drand48() - drand48();
    p1zf[ti] = drand48() - drand48();
    p2xf[ti] = drand48() - drand48();
    p2yf[ti] = drand48() - drand48();
    p2zf[ti] = drand48() - drand48();
    float mx = (p0xf[ti] + p1xf[ti] + p2xf[ti]) / 3.0f;
    float my = (p0yf[ti] + p1yf[ti] + p2yf[ti]) / 3.0f;
    float mz = (p0zf[ti] + p1zf[ti] + p2zf[ti]) / 3.0f;
    p0xf[ti] -= mx;
    p0yf[ti] -= my;
    p0zf[ti] -= mz;
    p1xf[ti] -= mx;
    p1yf[ti] -= my;
    p1zf[ti] -= mz;
    p2xf[ti] -= mx;
    p2yf[ti] -= my;
    p2zf[ti] -= mz;
  }
  for (int pi = 0; pi < number_of_packets; ++pi) {
    float ex = (drand48() - drand48()) * eye_range;
    float ey = (drand48() - drand48()) * eye_range;
    float ez = (drand48() - drand48()) * eye_range;
    float tx = (drand48() - drand48()) * target_range;
    float ty = (drand48() - drand48()) * target_range;
    float tz = (drand48() - drand48()) * target_range;
    for (int ri = 0; ri < packet_size; ++ri) {
      oxf[pi][ri] = ex + (drand48() - drand48()) * ray_jitter;
      oyf[pi][ri] = ey + (drand48() - drand48()) * ray_jitter;
      ozf[pi][ri] = ez + (drand48() - drand48()) * ray_jitter;
      dxf[pi][ri] = tx - ex +
        (drand48() - drand48()) * ray_jitter;
      dyf[pi][ri] = ty - ey +
        (drand48() - drand48()) * ray_jitter;
      dzf[pi][ri] = tz - ez +
        (drand48() - drand48()) * ray_jitter;
      rtf[pi][ri] = 1000000.0f;
    }
  }
  timeval start;
  gettimeofday(&start, 0);
  // Intersection test begins here
  for (int pi = 0; pi < number_of_packets; ++pi) {
    for (int ti = 0; ti < number_of_triangles; ++ti) {
      // Get triangle corners, compute two edges and normal.
      // (Alternatively, can precompute and store them)
      const __m128 p1x = _mm_set_ps1(p1xf[ti]);
      const __m128 p1y = _mm_set_ps1(p1yf[ti]);
      const __m128 p1z = _mm_set_ps1(p1zf[ti]);
      const __m128 p0x = _mm_set_ps1(p0xf[ti]);
      const __m128 p0y = _mm_set_ps1(p0yf[ti]);
      const __m128 p0z = _mm_set_ps1(p0zf[ti]);
      const __m128 edge0x = _mm_sub_ps(p1x, p0x);
      const __m128 edge0y = _mm_sub_ps(p1y, p0y);
      const __m128 edge0z = _mm_sub_ps(p1z, p0z);
      const __m128 p2x = _mm_set_ps1(p2xf[ti]);
      const __m128 p2y = _mm_set_ps1(p2yf[ti]);
      const __m128 p2z = _mm_set_ps1(p2zf[ti]);
      const __m128 edge1x = _mm_sub_ps(p0x, p2x);
      const __m128 edge1y = _mm_sub_ps(p0y, p2y);
      const __m128 edge1z = _mm_sub_ps(p0z, p2z);
      const __m128 normalx = _mm_sub_ps(
                             _mm_mul_ps(edge0y, edge1z),
                             _mm_mul_ps(edge0z, edge1y));
      const __m128 normaly = _mm_sub_ps(
                             _mm_mul_ps(edge0z, edge1x),
                             _mm_mul_ps(edge0x, edge1z));
      const __m128 normalz = _mm_sub_ps(
                             _mm_mul_ps(edge0x, edge1y),
                             _mm_mul_ps(edge0y, edge1x));
const __m128 zeroes = _mm_setzero_ps();
// Loop over "packlets", computing four rays at a time
for (int ri = 0; ri < packet_size; ri += 4) {
  // Load origin, current t-value and direction
  const __m128 ox = _mm_load_ps(&oxf[pi][ri]);
  const __m128 oy = _mm_load_ps(&oyf[pi][ri]);
  const __m128 oz = _mm_load_ps(&ozf[pi][ri]);
  const __m128 oldt = _mm_load_ps(&rtf[pi][ri]);
  const __m128 dx = _mm_load_ps(&dxf[pi][ri]);
  const __m128 dy = _mm_load_ps(&dyf[pi][ri]);
  const __m128 dz = _mm_load_ps(&dzf[pi][ri]);
  // Compute volume V, the denominator
  const __m128 v = _mm_add_ps(_mm_add_ps(
                            _mm_mul_ps(normalx, dx),
                            _mm_mul_ps(normaly, dy)),
                      _mm_mul_ps(normalz, dz));
  // Reciprocal estimate of V with one round of Newton
  const __m128 rcpi = _mm_rcp_ps(v);
  const __m128 rcp = _mm_sub_ps(
                        _mm_add_ps(rcpi, rcpi),
                        _mm_mul_ps(_mm_mul_ps(rcpi, rcpi),
                                   v));
  // Edge from ray origin to first triangle vertex
  const __m128 edge2x = _mm_sub_ps(p0x, ox);
  const __m128 edge2y = _mm_sub_ps(p0y, oy);
  const __m128 edge2z = _mm_sub_ps(p0z, oz);
  // Compute volume Va
  const __m128 va = _mm_add_ps(_mm_add_ps(
                             _mm_mul_ps(normalx, edge2x),
                             _mm_mul_ps(normaly, edge2y)),
                       _mm_mul_ps(normalz, edge2z));
  // Find Va/V to get t-value
  const __m128 t = _mm_mul_ps(rcp, va);
  const __m128 tmaskb = _mm_cmplt_ps(t, oldt);
  const __m128 tmaska = _mm_cmpgt_ps(t, zeroes);
  __m128 mask = _mm_and_ps(tmaska, tmaskb);
  if (_mm_movemask_ps(mask) == 0x0) continue;
  // Compute the single intermediate cross product
  const __m128 intermx = _mm_sub_ps(
                            _mm_mul_ps(edge2y, dz),
                            _mm_mul_ps(edge2z, dy));
  const __m128 intermy = _mm_sub_ps(
                            _mm_mul_ps(edge2z, dx),
                            _mm_mul_ps(edge2x, dz));
  const __m128 intermz = _mm_sub_ps(
                            _mm_mul_ps(edge2x, dy),
                            _mm_mul_ps(edge2y, dx));
  // Compute volume V1
  const __m128 v1 = _mm_add_ps(_mm_add_ps(
                             _mm_mul_ps(intermx, edge1x),
                             _mm_mul_ps(intermy, edge1y)),
                       _mm_mul_ps(intermz, edge1z));
  // Find V1/V to get barycentric beta
  const __m128 beta = _mm_mul_ps(rcp, v1);
  const __m128 bmask = _mm_cmpge_ps(beta, zeroes);
  mask = _mm_and_ps(mask, bmask);
  if (_mm_movemask_ps(mask) == 0x0) continue;
  // Compute volume V2
  const __m128 v2 = _mm_add_ps(_mm_add_ps(
                            _mm_mul_ps(intermx, edge0x),
                            _mm_mul_ps(intermy, edge0y)),
                       _mm_mul_ps(intermz, edge0z));
  // Test if alpha > 0
  const __m128 v1plusv2 = _mm_add_ps(v1, v2);
  const __m128 v12mask = _mm_cmple_ps(
                            _mm_mul_ps(v1plusv2, v),
                            _mm_mul_ps(v, v));
  // Find V2/V to get barycentric gamma
  const __m128 gamma = _mm_mul_ps(rcp, v2);
  const __m128 gmask = _mm_cmpge_ps(gamma, zeroes);
  mask = _mm_and_ps(mask, v12mask);
  mask = _mm_and_ps(mask, gmask);
  if (_mm_movemask_ps(mask) == 0x0) continue;
  // Update stored t-value for closest hits
        _mm_store_ps(&rtf[pi][ri],
                     _mm_or_ps(_mm_and_ps(mask, t),
                               _mm_andnot_ps(mask, oldt)));
        // Optionally store barycentric beta and gamma too
      }
    }
  }
  // Show speed in millions of intersections per second
  timeval now;
  gettimeofday(&now, 0);
  float elapsed =
      (static_cast<float>(now.tv_sec - start.tv_sec) +
       static_cast<float>(now.tv_usec - start.tv_usec) /
       1000000.0f);
  if (argc > 1) {
    ofstream out(argv[1], ios::out);
    out << (number_of_packets * packet_size
           * number_of_triangles
           / elapsed / 1000000);
  }
  else
    cout << (number_of_packets * packet_size
            * number_of_triangles
            / elapsed / 1000000) << endl;
  return 0;
}
```

References

[1] J. Amanatides and K. Choi. Ray tracing triangular meshes. In *Western Computer Graphics Symposium*, pages 43–52, 1997.

[2] Carsten Benthin. *Realtime Ray Tracing on Current CPU Architectures*. PhD thesis, University of Saarland, 2006.

[3] Hsiao-Lan Fang, Peter Ross, and Dave Corne. A promising genetic algorithm approach to job-shop scheduling, re-scheduling, and open-shop scheduling problems. In *Proceedings of the International Conference on Genetic Algorithms*, pages 375–382, 1993.

[4] Ray Jones. Intersecting a ray and a triangle with Plücker coordinates. *Ray Tracing News*, 13(1), 2000.

[5] Xiaoming Li, Maria Jesus Garzaran, and David Padua. Optimizing sorting with genetic algorithms. In *Proceedings of the international symposium on Code generation and optimization*, pages 99–110, 2005.

[6] Marta Löfsted and Tomas Akenine-Möller. An Evaluation Framework for Ray-Triangle Intersection Algorithms. *Journal of Graphics Tools*, 10(2):13–26, 2005.

[7] Tomas Möller and Ben Trumbore. Fast, minimum storage ray triangle intersection. *JGT*, 2(1):21–28, 1997.

[8] Joseph O'Rourke. *Computational geometry in C*. Cambridge University Press, New York, NY, USA, second edition, 1998.

[9] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Saarland University, 2004.

[10] Turner Whitted. An improved illumination model for shaded display. *CACM*, 23(6):343–349, 1980.