Ray-Triangle Intersection Algorithm for Modern CPU Architectures

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

Abstract

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

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

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

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

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

1. INTRODUCTION

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

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

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

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

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

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

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

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

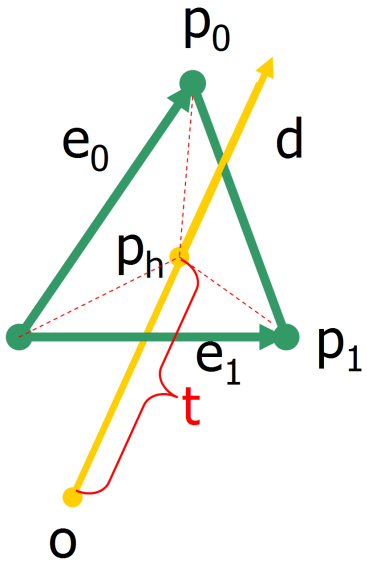

Figure 1: Ray-Triangle Intersection Algorithm finds whether intersection point p_h exists. If yes, computes the u,v parameters of intersection (such as $p_h = p + e0*u + e1*v$) as well as distance $t=\{o, p_h\}$.

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

2. PREVIOUS WORK

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

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

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

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

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

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

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

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

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

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

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

3. BASICS OF PLÜCKER TEST

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

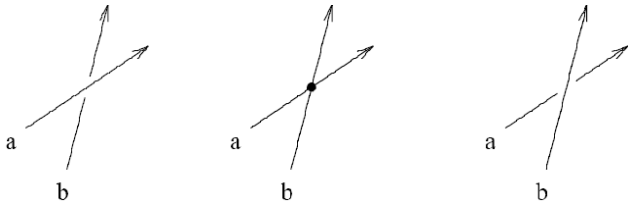

Figure 2: Three possibilities for two directed lines whether one passes to one side or the other (depending on the sign of the inner product).

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

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

$\mathbf{e}0=\{\mathbf{p}-\mathbf{p}0,\ \mathbf{p}\times\mathbf{p}0\}$, $\mathbf{e}1=\{\mathbf{p}1-\mathbf{p},\ \mathbf{p}1\times\mathbf{p}\}$, $\mathbf{e}2=\{\mathbf{p}0-\mathbf{p}1,\ \mathbf{p}0\times\mathbf{p}1\}$,

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

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

$t0 = (\mathbf{e}0, \mathbf{R})$, $t1 = (\mathbf{e}1, \mathbf{R})$, $t2 = (\mathbf{e}2, \mathbf{R})$.

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

4. SIMD INTERSECTION ALGORITHM

SIMD-fashion testing multiple rays instead of one ray for intersection with triangle can dramatically reduce the cost of rendering. SIMD architecture performs multiple floating point operations in parallel. This is common technique for speeding up ray-tracing [3, 5, 13]. For traversing the usage of packets wider than current SIMD is used to reduce the average bandwidth required per ray since rays in packet usually access the same