School of Computing, University of Utah, Technical Report No UUCS-06-10

Geometric and Arithmetic Culling Methods for Entire Ray Packets

Solomon Boulos[1] Ingo Wald[2] Peter Shirley[1]

[1]School of Computing, University of Utah [2]SCI Institute, University of Utah

ABSTRACT

Recent interactive ray tracing performance has been mainly derived from the use of ray packets. Larger ray packets allow for significant amortization of both computations and memory accesses; however, the majority of primitives are still intersected by each ray in a packet. This paper discusses several methods to cull entire ray packets against common primitives (box, triangle, and sphere) that allows an arbitrary number of rays to be tested by a single test. This provides cheap "all miss" or "all hit" tests and may substantially improve the performance of an interactive ray tracer. The paper surveys current methods, provides details on three particular approaches using interval arithmetic, bounding planes, and corner rays, describes how the respective bounding primitives can be easily and efficiently constructed, and points out the relation among the different fundamental concepts.

Keywords: ray tracing, interval arithmetic, culling, ray packets, frustum culling

1 INTRODUCTION

Ray tracing programs are particularly compute-intensive, and methods to make them faster have a long history. Traditionally, research has focused on individual rays, using various spatial or hierarchical index structures [1, 3, 7, 17, 18, 23] to reduce the number of traversal steps and primitive intersections that a ray has to perform. Recently, single ray implementations have been supplanted by those that use sets of rays together to improve the cohence of memory access, to share common operations, and to allow the use of SIMD units in modern CPUs [5, 10, 16, 19, 22, 26–28]. These "packets" of rays introduce several new software architecture issues and opportunities not found in a single-ray (non-packet) implementation. One of these opportunities is to test a whole packet for intersection against a primitive to avoid testing each ray individually.

This paper deals with a particular type of packet-primitive test: determining whether all the rays in a packet miss a primitive. Such a test is most useful for culling which implies a fast conservative test. By "conservative" we mean that the test either determines all rays certainly miss, or that some might or might not hit. Although such tests might or might not miss some opportunities to cull because they are conservative, they can be much faster than the more accurate tests used in applications such as beam tracing [15]. These conservative culling tests are especially useful for terminating acceleration structure traversals (e.g. when the whole packet misses some box) [22, 26, 27] or when testing a ray packet against a geometric primitive such as a sphere or triangle (e.g., [10, 13]). This algorithmic use of ray packets provides a large benefit that any packet based system can take advantage of and can provide much more than the speedup achieved through simply tracing rays in a SIMD fashion. Note that though we focus mostly on "all miss" tests (which are particularly effective as they avoid *any* work if an all miss is detected), these tests generalize naturally to "all hit" cases as well.

We concentrate on three basic approaches to culling: interval arithmetic, plane tests, and corner rays tests. Interval arithmetic is an algebraic approach with implicit underlying geometry, while plane tests and corner ray tests explicitly use geometric reasoning to exclude intersections. We also discuss the ramifications of different types of ray packets, including general ray packets that neither share directions nor share common signs for the Cartesian components of their direction vectors. The rest of the paper derives these concepts in detail. Some of the material has appeared previously at a high level, but we believe many of the details necessary for implementing these techniques, as well as the generalization of some of the techniques, appear here for the first time.

2 TOOLS FOR ARITHMETIC AND GEOMETRIC CULLING

In this section we review both the basics of interval artimetic as well as the use of planes to partition point sets based on side-tests.

2.1 Properties of plane equations

A plane in 3D can be defined by any point $\vec{p}_0$ and a normal $\vec{N}$. For any point $\vec{p}$ on the plane, there is an implicit equation $f(\vec{p}) = 0$. For points not on the plane, the function f returns either a positive number or a negative number depending on which side of the plane the point is on. The magnitude of the number is proportional to distance from the plane (in the case of unit length normal it is exactly the distance from the plane).
Unions of half-spaces defined by lines in 2D or planes in 3D can be used to define convex polygons/polyhedra. This is illustrated for a triangle in Figure 1(a). Any point that is on the same half-space for all three sides is in the triangle.

Half-spaces suggest a straightforward method of conservative convex polygon/polygon and polyhedron/polyhedron overlap preclusion. As shown in Figure 1(b), if all the vertices of one polygon lie to the "outside" of any of the edges of the other polygon, we can conclude that they definitely overlap. Note that if this criterion is not met, the polygons may still not overlap. That is the nature

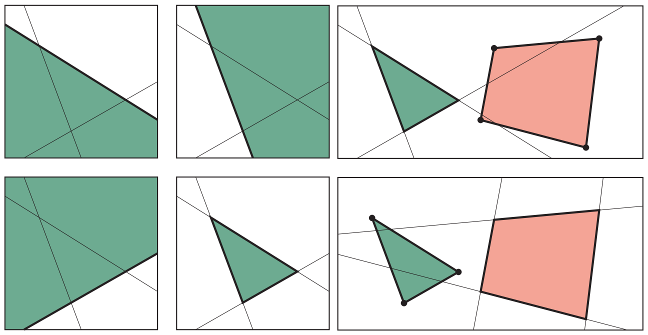

Figure 1: Defining primitives as intersection of half-spaces, and using that for culling. a) Any convex polygon in 2D can be defined by the intersection of the half-planes defined by its edges. The same idea applies for 3D convex polyhedra (such as boxes and frusta). b) If all of the vertices of a polygon lie to the "outside" of any of the half-planes defining another polygon (i.e., the one defined through the intersection of a bounding frustum with the polygon's supporting plane), then the two polygons definitely do not overlap. This is a conservative test; only using the vertices of the quadrilateral (top) does not indicate a rejection, while reversing the tests (bottom) does.

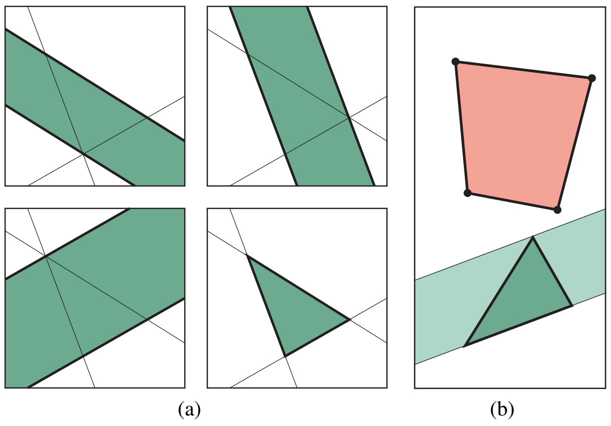

Figure 2: Defining a primitive as intersection of "slabs", and using this for culling. a) A convex polygon can be defined by the intersections of two or more slabs. For a triangle, this can be thought of as restricting all three barycentric coordinates to $[0,1]$ and opposed to restricting them to be non-negative as is done for the intersection of three half-planes. b) We can conclude these polygons do not intersect if all of the vertices of one polygon lie to the same side of any of the slabs defining the other polygon.

of a simple conservative test: it quickly determines whether further computation might be avoided rather than providing a precise answer for all cases.

A variant of the half-space method is to take intervals along various dimensions sometimes called "slabs". For example, a triangle can be defined as the intersection of three slabs as shown in Figure 2(a). This leads to the culling test shown in Figure 1(b) where each vertex in one polygon is tested against each slab in the other polygon. If all vertices are on the same side of any slab, the polygons definitely do not overlap. Note that this test in Figure 1(b) is able to cull the case that the plane test shown in the top of Figure 1(b) does not (but it is still conservative).

2.2 Interval Arithmetic (IA)

Classical arithmetic defines operations on individual numbers; each variable is supposed to represent one exact, individual number, and all operations (addition, multiplication, functions, relations, ...) work on individual numbers, and return invididual numbers. In many applications, however, variables are subject to tolerances, uncertainties, or rounding errors. *Interval Arithmetic* (IA) was developed to allow analysis in such situations. With roots early in the twentieth century, interval aritmetic was developed in its modern form in the 1960s and is now a mature field of study (see Hayes [14] for an introduction and pointers to many of the key references and surveys).

An *interval* is the well known one from elementary mathematics: a set of all the points on the real line between two specified endpoints, e.g., [4,7]. In interval arithmetic we have a variable whose value is known to be in some interval, e.g., $a \in [\underline{a}, \overline{a}]$ is written:

$$a = [\underline{a}, \overline{a}].$$

Here $\underline{a}$ and $\overline{a}$ are the boundaries of the interval and can be read as $a_{\min}$ and $a_{\max}$. Note that the sum of two interval variables in also an interval:

$$a + b = [\underline{a} + \underline{b}, \overline{a} + \overline{b}].$$

Because interval operations are not something most of us use regularly, we think using a small set of kernal operations is a good idea. The ones we use are addition, negation, multiplication, and reciprocal. Division and subtraction can be constructed from these.

Negation is straightforward:

$$-[\underline{a}, \overline{a}] = [-\overline{a}, -\underline{a}].$$

Multiplication is more complicated because of the possibility of negative numbers:

$$ab = [\min(\underline{a}\underline{b}, \underline{a}\overline{b}, \overline{a}\underline{b}, \overline{a}\overline{b}), \max(\underline{a}\underline{b}, \underline{a}\overline{b}, \overline{a}\underline{b}, \overline{a}\overline{b})]$$

The reciprocal operation is complicated only because intervals that include the origin must be accounted for:

$$\frac{1}{a} = \begin{cases} [1/\overline{a}, 1/\underline{a}] & \text{if } a \text{ does not include } 0, \\ [-\infty, \infty] & \text{otherwise} \end{cases}$$

Operations with scalars can be expressed by using the zero-measure intervals $[a,a]$ and thus follow from the above rules. For example,

$$7 + a = [7,7] + [\underline{a}, \overline{a}] = [7 + \underline{a}, 7 + \overline{a}].$$

Set operations In classical arithmetic, a relation (less than, greater than) can return only two values, true or false. In interval arithmetic, if two variables' intervals overlap the outcome of the comparison cannot be decided, yielding three different cases:

$$[a,A] < [b,B] = \begin{cases} true & ; A < b \\ false & ; a \geq B \\ undecided & ; otherwise \end{cases}.$$

This tri-valued outcome of a test also has a direct relation to algorithmic or geometrical tests: most geometrical tests contain some "if this conservative test based on bounding primitive is true, then A() , else B()"; in this formulation the "true" and "false" outcome of an IA comparison correspond to a conservative test, and the "undecided" corresponds to the case where no conservative conclusion could be reached by looking at the geometric bounding primitive.

For both geometric and IA tests a true/false outcome (respectively a passed conservative test) usually means that the test's outcome decides all individual rays' tests with a single operation, while in the "undecided" case all invididual tests have to be performed.

Set operations Because an interval is a set of points, set operations can apply, and the empty interval is just the empty set $\emptyset$. The set operations *union* and *intersection* (often called collapse and join) are fundamental in most interval computations. These are straightforward:

$$a \cup b = [\min(\underline{a}, \underline{b}), \max(\overline{a}, \overline{b})],$$

$$a \cap b = [\max(\underline{a}, \underline{b}), \min(\overline{a}, \overline{b})].$$

Here $[u,v] = \emptyset$ when $u > v$. The reason that interval operations are so fundamental for IA is that they correspond closely to the geometric culling tests: if a primitive is defined through the intersection of several half-spaces or slabs (i.e., sets of 3D points), we can use IA on each of these half-spaces or slabs, perform IA set operations to intersect the resulting intervals, and test the interval for emptyness, in which case there can be no overlap.

Intervals of vectors Often we want to use vectors whose Cartesian components are themselves intervals. For example, $\vec{V} = ([0,1], [-1,1], [3.2, 3.3])$. Operations on such "interval vectors" also produce intervals or interval vectors. For example, the vector dot product is:

$$(X_0, Y_0, Z_0) \cdot (X_1, Y_1, Z_1) = X_0 X_1 + Y_0 Y_1 + Z_0 Z_1.$$

Since we have well defined operations for multiplying and adding intervals, we can use those to implement the equation above. Similarly a cross product of two interval vectors yields an interval vector, and can be implemented using the scalar interval multiplication and addition operations.

Although interval analysis has been used for many tasks in computer graphics [24], we now give the example closest to how we will use it for culling: the intersection test of a single 2D ray against a 2D axis-aligned box. We also now adopt the notation that intervals will be denoted with capital letters (e.g., A) to distinguish them from scalars. The ray is the set of points $\vec{o}+T\vec{v}$. Here $T=[\underline{T},\overline{T}]$, so more precisely the set of points forms a line segment for finite intervals. The box is given by $(x,y)\in B_x\times B_y$, where B_x and B_y are each an axis-aligned slab and their intersection (Cartesian product) is a box. The line containing the ray will intersect each slab with its own interval in the ray parameter space:

$$o_x+T_xv_x=B_x,$$

$$o_y+T_yv_y=B_y.$$

We can find T_x and T_y using our interval operations. For example,

$$T_x=(B_x-o_x)\frac{1}{v_x}.$$

Some care must be taken about handling the sign of v_x here, but otherwise the computation is straightforward. The ray "hits" the box if there is any ray parameter that is in each of T_x, T_y, and T, i.e., $T_x\cap T_y\cap T\neq\emptyset$.

3 Using IA for Packet/Primitive Culling

When using IA, the packet must first have intervals computed and stored for the various quantities of interest. For example, $O_x=[\underline{O_x},\overline{O_x}]$. These may include computed quantities such as the ratio of o_x to v_x which can be tighter than the ratio of the intervals O_x and V_x, i.e., the interval associated with the ratios can be smaller than the ratio of the intervals. For example, the interval that includes the quantities 1/2 and 2/4 is a single point while the interval $[1,2][2,4]^{-1}=[1,2][\frac{1}{4},\frac{1}{2}]=[\frac{1}{4},1]$. We now discuss IA for culling boxes, spheres and triangles against ray packers.

3.1 IA for Axis-aligned Boxes

Applying IA to cull ray-packets for boxes is straightforward[1]. To see the basic idea, consider the specific 2D example in in Figure 3 with $O_x=[-3,-1]$, $O_y=[1,2]$, $V_x=[1,2]$, $V_y=[1,3]$, $V_x=[2,3]$, $V_y=[-1,2]$, where the rays are $\vec{o}+[0,\infty]\vec{v}$ and points in the box are $\vec{p}$. We can compute:

$$T_x=(p_x-o_x)v_x^{-1}=([2,3]-[-3,-1])[1,2]^{-1}=[3,6][\frac{1}{2},1]$$
$$=[\frac{3}{2},6]$$

Similarly,

$$T_y=(p_y-o_y)v_y^{-1}=([-1,2]-[1,2])[1,3]^{-1}=[-3,1][\frac{1}{3},1]$$
$$=[-3,1]$$

Because $T_x\cap T_y=\emptyset$, we can conclude that none of the rays in the packet can hit the the box without needing to test the ray interval $[0,\infty]$ (if T_x and T_y overlapped, we would need to compare against the ray parameter interval as well).

In 3D, we start with the interval on ray paramaters T, as well as the interval vectors $\vec{O}$ and $\vec{V}$ for the packet. We also have the

[1]Applying IA to 1D planes as used by k-d trees is simply a subcase of the analysis used for AABB and is straightforward.

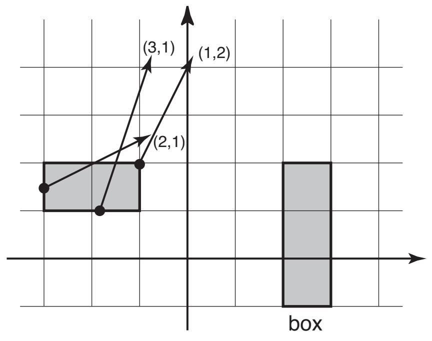

Figure 3: Example of walkthrough of a 2D interval rejection.

interval vector B describing the box. As in the 2D example above we compute the intervals T_x, T_y and T_z. For example, as in 2D:

$$T_x=(B_x-O_x)\frac{1}{V_x} \qquad (1)$$

We can cull the packet if $T\cap T_x\cap T_y\cap T_z=\emptyset$. Note that this can be coded very cleanly in $C++$ with an interval class that supports multiplication, subtraction and reciprocal.

For the special case of shared origin ray packets, Equation 1 can use an interval minus scalar operation for B_x-o_x as the ray origin is a collasped interval (single number). For the special case where the components of each dimension of $\vec{V}$ have the same sign, the reciprocal operation can omit the check for spanning the origin, i.e.,

$$T_x=(B_x-O_x)[1/\overline{V_x},1/\underline{V_x}].$$

If we store the ratio $r=o_x/v_x$ We can tighten the interval by allowing:

$$T_x=B_x(\frac{1}{V_x})-R$$

3.2 IA for Spheres

We can first compute the intersection of a ray with a sphere. As before, a capital letter indicates an interval or a vector with intervals for components. These can later be replaced with scalers for special cases. The intersection points of a sphere and ray are usually computed in ray parameter space:

$$\begin{aligned}\|\vec{O}+T_s\vec{V}-\vec{C}\|^2 &= r^2\\ (\vec{OC}+T_s\vec{V})\cdot(\vec{OC}+T_s\vec{V}) &= r^2\\ \vec{OC}\cdot\vec{OC}+2T_s\vec{V}\cdot\vec{OC}+T_s^2\vec{V}\cdot\vec{V} &= r^2\\ T_s^2\vec{V}\cdot\vec{V}+2T_s\vec{V}\cdot\vec{OC}+\left[\vec{OC}\cdot\vec{OC}-r^2\right] &= 0\end{aligned}$$

We can cull the sphere when the discriminant D is negative. The discriminant is:

$$D=(\vec{V}\cdot\vec{OC})^2-(\vec{V}\cdot\vec{V})\left[\vec{OC}\cdot\vec{OC}-r^2\right]$$

Values in that interval are less than or equal to zero whenever the intervals maximum is zero or smaller. In the case of unit length directions, the $\vec{V}\cdot\vec{V}$ term is unity and drops out.

Note that this culling is for the 3D line containing the ray and does not take into account the allowed interval T in ray parameter space. If we want to also check for a proper T_s interval, we need to solve for the range of possible T_s and see if it overlaps T. If D has a positive component we can make $D = D \cup [0, \infty]$ and then check for the values possible for T_s:

$$T_s = \frac{-\vec{V} \cdot \vec{OC} \pm \sqrt{D}}{\vec{V} \cdot \vec{V}}$$

In the case of normalized $\vec{V}$ this becomes:

$$T_s = [-(\overline{\vec{V} \cdot \vec{OC}}) - \sqrt{\overline{D}}, -(\underline{\vec{V} \cdot \vec{OC}}) + \sqrt{\overline{D}}]$$

3.3 IA for Triangles

For the case of triangles, there are numerous ray-triangle intersection routines [2, 4, 20, 21, 25] but the general interval arithmetic approach can be used for any of them. It is important to note, however, that for the majority of tests it is important that the ray directions have the same length (we recommend normalizing the rays to be unit length as it allows for more optimizations) so that computed quantities like distances to supporting planes are not scaled differently for different rays. This allows for very simple conversion of single ray-triangle intersection routines into conservative interval-triangle routines.

As an example, many ray-triangle intersection tests compute a determinant (or equivalent triple-product or Plüker inner product). One way the barycentric coordinate α can be computed is:

$$\alpha = \frac{[(\vec{p}_2 - \vec{o}) \times (\vec{p}_0 - \vec{o})] \cdot \vec{v}}{[(\vec{p}_1 - \vec{p}_0) \times (\vec{p}_2 - \vec{p}_0)] \cdot \vec{v}}.$$

For the shared origin case in particular this can produce intervals efficiently. The resulting interval for α can then be checked against $[0, 1]$.

3.4 Implementation notes

IA is extremely straightforward to implement. We have used a C++ class that implements the operations described in Section 2. Once such a class structure is in place, the operations take care of all the special cases and problems. However, some care must be taken as a naive implementation of IA can generate overly large intervals (i.e., taking the intervals after the division of ray origin and direction components). Most of what we discussed here assumes a general ray packet. However, when there are special cases such as shared origin, the code could be modified to replace interval-interval operations with interval-scaler operations for efficiency. However, this should be looked at on a case-by-case basis as sometimes there is no real savings.

4 Bounding Planes

As mentioned before, while interval arithmetic provides excellent analysis through equations it is also possible to use geometric interpretations of all the possible rays in a packet. This naturally leads to bounding planes that surround the rays in a packet. Given a ray packet then, it is important to consider how to build a geometric representation of the full packet that is useful for culling tests.

4.1 Building the Planes

4.1.1 Major Axis

Following the previous use of bounding planes for restricted ray packets [27], we define the notion of a "major axis" for a packet of rays. A major axis is one for which all ray directions agree in sign (either positive or negative) and allows us to reason about a forward expanding packet (the packet expands in this direction). The other component axes shall be denoted u and v as in [27].

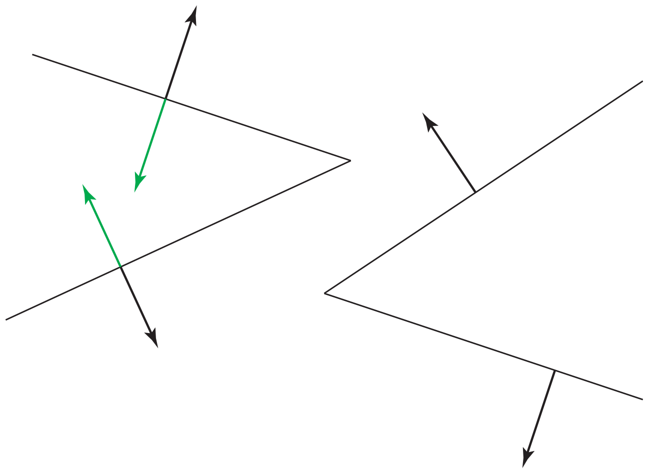

Figure 4: Left: Rays with $dk < 0$. Right: Rays with $dk > 0$. Correct normals are shown in black; incorrectly generated normals in green. Because $dk < 0$ on the left, the top ray has a negative slope while the bottom ray has a positive slope. The normal must be flipped to account for this.

In [27] the major axis is chosen according to the maximum component of the first ray under the assumption this will work for all rays in the packet; this is fine for coherent primary rays, but for general packets a more robust mechanism is necessary. We will simply follow the definition and choose the first axis in which all ray directions have the same sign (either all positive or all negative).

4.1.2 Slopes

Once we have chosen a major axis, we can examine the relation of the other two axes to the major axis. This gives us a simple 2D projection of the packet, where ray directions may be reasoned about in terms of slopes. As before this is similar to the method used in [27] but we wish to expound the details.

For each ray in the packet, its ray direction in this 2D space can be written as $[dk, du]$ where k is the major axis, u is the other axis under consideration and dk (respectively du) are the components of the ray direction for that axis. From this we may rewrite the ray direction as $[1, du/dk]$ which simply gives us a ray that expands in the major axis with slope du/dk. For now, we will ignore the case where $dk < 0$ and will handle it later; however, it should be noted that we avoid the case of $dk = 0$ for any ray in the packet if we use the product test to choose an axis.

Bounding lines for a packet in this 2D space can be chosen by finding the maximum and minimum slope over all rays in a packet. Again if $dk < 0$ the maximum and minimum slope will have different roles, but we will need both terms anyway.

4.1.3 Plane Normals

Given a slope of a ray, we can determine its normal by choosing a suitable perpendicular vector. There are two possible choices: $[m, -1]$ and $[-m, 1]$ where m is the ray slope. For the "top" plane, we want $[-m, 1]$ where m is the maximum slope while for the "bottom" plane we want $[m, -1]$. This is fairly clear when we consider that the "top" plane should point upwards (similarly the bottom plane normal should point downwards) so that our normals point outside (see Figure 4, right).

At this point we must make the only change necessary to handle $dk < 0$. If $dk < 0$ then we must flip the normals so that they appropriately point outside the region where rays may exist.

4.1.4 Plane Origins

Once we have normals for our planes, we must determine where to place them to ensure that they bound all the rays in our packet. To do so, we simply insert the origins of each ray into the plane equation and keep track of the distance term. The ray origin that generates the maximum distance for a plane can be used directly as the plane origin. Alternatively, once we have computed the maximum distance term, we may keep that alone (the origin itself is not explicitly required).

4.1.5 Full Algorithm

Now that we have a method for generating plane normals and origins from ray directions, we describe a simple method to generate a set of four planes that bound a ray packet. We begin by calculating the maximum and minimum slope of the rays in the 2D spaces that use the major axis as the first component: the *ku* and *kv* planes (as defined in [27]). From these slopes we compute normals and origins for the planes as outlined above. This gives us two planes in *ku* and two more from *kv*. These four planes then bound our packet.

4.2 Applications

Now that we have planes that bound our packet, we may use them as a conservative representation of our packet. In particular, the "interior" of the packet is the intersection of the four half-spaced defined by these planes, and if any primitive lies to the outside of *any* of these half-spaces, it cannot intersect any ray in the packet.

4.2.1 Axis Aligned Bounding Box

An axis aligned bounding box can be defined as the region inside the 8 corner vertices of the box. If we take each vertex and insert it into a plane equation we will extract a simple distance term which allows us to decide which side of the plane it is on. If each of the 8 vertices is outside any of the bounding planes of our ray packet, we can conclusively decide that all the rays in the packet miss the box.

Note, however that we do not have to test all of the 8 corners, and that testing the closest one is sufficient. Based on the signs of the plane's normal components finding the vertex that is hindmost along the direction along the normal is trivial, and testing this vertex is sufficient (note that this, in fact, is another application of IA!).

4.2.2 Sphere

A sphere is slightly more interesting in that it is not a polhedron, but is defined to be all points a distance r from a point $\vec{C}$. In this case the simple framework of testing all vertices of a primitive against the planes does not make sense; however, we may test the center of the sphere against each plane to determine the minimum distance between the plane and the point. If this minimum distance is greater than the sphere radius, r, then the ray packet can not intersect the sphere (this test was recently used by Gribble et al. [12]).

4.2.3 Triangles

A triangle is similar to the previous case of an axis aligned box. If all the vertices of the triangle lie to the outside of one of the ray packet's bounding planes, we can immediately determine that none of the rays in the packet may intersect the triangle. Note that this directly corresponds to frustum culling used in GPU rendering.

4.2.4 Moving Primitives

The case of moving primitives deserves some attention. For distribution ray tracing [6, 9], it is important to allow primitives to move within the frame of animation to provide motion blur. Bounding planes may be used to test for conservative culling with moving primitives (defined by linear movement) by simply ensuring the all vertices are on the outside of a single plane. For example, if all 6 vertices of a moving triangle (3 from the start of the movement, 3 from the end) are on the same side of a plane then any point in between is similarly on that side. If all 6 are on the outside of the plane, then we may cull the moving primitive.

4.3 Implementation Notes

Though we want mostly want to abstract from implementation issues, two details are worth noting: First, bounding planes are usually orthogonal to at a coordinate plane (xy, yz, or xz), and this is particularly true for the construction method outlined above. In that case, one of the normal's component is always zero, so computing a point's distance to the plane gets quite simple, in particular if the plane is stored in the "$\vec{x}.\vec{n} - d$" form instead of "$(\vec{x} - \vec{a}).n$". One can also re-sale the normal to have a '1' in one of the components, saving anther multiplication per test. If spheres are to be tested, however, it is necessary to keep the normal unit length or at least retain the scaling factor for bookkeeping. Second, since we always have 4 planes to be tested against, it is a natural choice to use SIMD extensions to perform the four tests in parallel.

5 Corner Rays

While bounding planes may be used to cull various sets of primitives, they may not cull a large portion of cases due to their axis aligned nature. For example, when using an acceleration structure one might expect the case shown in Figure 1(b) to be handled in the acceleration structure (the ray packet will then not test this triangle). However, if the ray packet projection instead touches inside the bounding box of the triangle it is likely that we will still need to test the triangle and that an acceleration structure might not avoid this case.

For this situation, authors have used corner rays[5, 10, 22, 26, 27] to bound the packet of rays. In most of these papers, the concept of corner ray culling has only been used for primary rays with shared origin, for which the "corner rays" (hence the name) are defined by the rays passing through the corners of a tile of pixels. For primary rays with common origin, these corner rays bounds all interior rays, for secondary rays—or if depth-of-field is added to the primary rays [6]—this property is not necessarily true any more.

Though corner ray culling has been mainly proposed for primary rays with shared origin, it can be generalized to arbitrary ray: the convex hull any four rays spans a volume (bounded, in general, by bilinear patches on the sides, and if this volume does not intersect a given primitive then no ray inside this volume can, either[2] As such, for every set of N rays we can find a set of four rays whose convex hull bounds these rays, and use these as "corner rays". Note, however, that we do not have to pick 4 out of the original N rays, but that we can pick entirely "virtual" rays that are not part of the original set of N rays; though in this case the original meaning of the term "corner rays" does no longer apply, we will continue to use it, even though "bounding rays" would be more exact.

Though in general any set of four rays whose convex full encloses the rays, finding a good set of rays is not trivial. We will therefore restrict ourselves to rays whose convex hull has planar sides, which are easier to generate. We will now outline how

[2]Note that we have argued about the *volume* spanned by the corner rays, not about these rays themselves. For any type of primitive there are cases where all corner rays miss, but the spanned volume still intersects the primitive [22]. One particularly trivial of these cases is where the primitive lies entirely inside the spanned volume, but more cases exist.

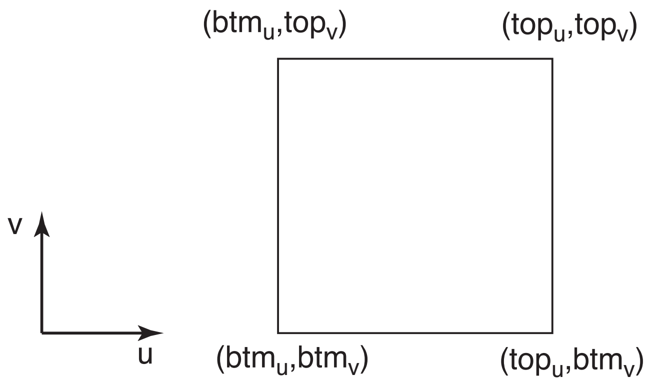

Figure 5: When projecting the packet into any uv plane the top and bottom bounding planes form a square cross-section, and the corners of this square (along the $\vec{K}$ axis) form our corner rays. For the quad formed by the packet's back plane along $\vec{K}$, the corners are the origins of our corner rays.

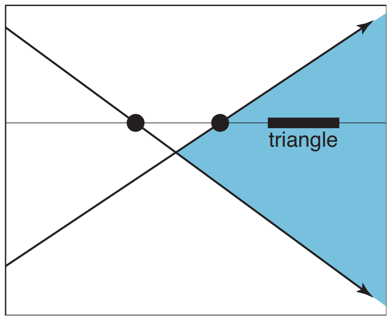

Figure 6: A 2D projection of a ray packet and a triangle within the region bounded by the corner rays. The dots show the intersection of each ray with the supporting plane of the triangle. The lower ray intersects the triangle's supporting plane behind the ray origin ($t < 0$); rays between the top and bottom corner ray will not have hit points between those of the corner rays but rather outside.

to generate these corner rays from our previous bounding planes. Note, however, that all the concepts naturally generalize to any four bounding rays, as even in the general case the intersection between the (bilinear patch-bounded) convex hull with the triangle's embedding plane forms a convex quadrilateral.

5.1 Building Corner Rays from Bounding Planes

As mentioned above, for primary rays the generation of corner rays is trivial, but for the general case can be more complicated. In the previous section, however, we have already described how a set of geomeric bounding planes can be found for general ray packets, and once these are known we can use them to compute bounding rays as well. If we take any pair of two neighboring planes, their intersection form an infinite line which we can use as a corner ray. The direction of the line is given by the cross product of the normals (it is important to ensure that the cross product is given inputs in the correct order to ensure that the resulting direction agrees with the major axis). Determining the point of origin is slightly more complicated.

We know that the line of intersection between the two neighboring planes contains all points that satisfy both plane equations. If we intersect these four bounding planes with the "back plane" (i.e., the plane passing through the hindmost ray origin along the major march direction), then we get a regular, axis-aligned quad (see Figure 5). The 3D position of this quad form the origins of our virtual corner rays. Assuming our major axis is $\vec{K}$, then the $\vec{K}$ component of all these four points is k_{back}, the position of the back plane. In addition, the point (k_{back}, v_{top} in the (kv) plane has to fulfill the top plane's plane equation $n_k^{(top)} k_{back} + n_v^{(top)} v_{top} - d^{(top)} = 0$, yielding $v_{top} = \frac{d^{(top)} - n_k^{(top)} k_{back}}{n_v^{(top)}}$. Computing the remaining three values is straightforward.

5.2 Using Corner Rays

Once we have our four corner rays for a packet, we may insert them into any ray-triangle test we wish (similar to the case of interval arithmetic) that returns barycentric coordinates. As shown in Figure 2(b), this results in the projection of the cross-section of the ray packet into the plane of the triangle (forming a quad). As long as we determine that the vertices of this projection are all on one side of a slab/edge we may conclude that the rays miss the triangle. For example, if $\alpha < 0$ for all the corner rays then all the rays pass on the left side of an edge (alternatively if $\alpha > 1$ then we can conclude they all pass on the right side of an edge).

One important caveat of this method is the assumption that the ray packet builds a quad in the plane of the triangle and that all rays within the packet will fall within this quad. This is not the case if the corner rays do not generate hit points with positive ray parameters (i.e. $t < 0$). Figure 6 demonstrates that even in 2D negative ray parameters imply that rays within the packet do not go between the hit points of the corner rays, but rather lie outside them. Once this case is handled, however, corner rays produce culling tests that bounding planes and interval arithmetic may not detect.

5.3 Implementation Notes

What kind of triangle test is being used for the actual test is completely irrelevant. Though we had originally assumed that the different tests would lend differently well to culling tests, no matter what test is being used (Badouel [4], Arenberg [2], Wald [25], Möller-Trumbore [21], in one or another (and more or less obvious) way they all have to test the ray to the sides of the triangle; then, if all rays fail at the same side, the entire packet misses, whatever test is used for the actual computations, and whatever special kind of packet was used for generating the corner rays. Of course, this generalizes to all convex polygons, not only to triangles.

However, one important note from an implementation standpoint is that any test that can quickly generate barycentric coordinates may be faster than other tests. For example, if a test can determine that the four corner rays all have a negative value for the first barycentric coordinate then the other coordinates do not need to be checked (we already know that all the corner rays agree). Furthermore, tests that use a distance test first are not particularly advantageous: corner rays do not allow us to test the packet for distance to the plane but only allow for half-space culling.

6 Future Work

Of course the most pressing future work at a high level is to better understand the relative merits of existing culling techniques as well as to develop new ones. But there are some more specific ideas for future work we can list here.

Two of these "all miss" tests can be easily inverted to perform a simple "all hit" test. For example, interval arithmetic may reveal that all the rays in a packet must lie within the triangle because the barycentric coordinate intervals are always valid. Similarly, the corner rays can say the same thing: if all the corner rays lie within the triangle then we know the barycentric test must always pass.

This simple all hit test may be useful for architectures with high branching penalties, such as the Cell, where knowing that no tests are needed could be beneficial. We have not yet implemented these techniques in a Cell based ray tracer, but would expect them to yield performance benefits similar to those on the CPU.

Interval arithmetic can lead to quickly bloating intervals. Affine arithmetic [8] is designed to address this, and it could be useful. The amount of expansion in the dot product and cross product operations is fairly large and doesn't necessarily relate to any geometric interpretation of the operations. For example, the dot product between the direction interval vector and itself can be much greater than 1 even if all the ray directions are of unit length. Similarly, the cross product operation on intervals will expand significantly as each component of the resulting vector is the difference of two products of intervals.

7 Conclusions

Because ray packets have taken such a central role in modern ray tracing systems, culling entire packets against a primitive have become an important topic. In this paper we have both surveyed the common techniques discussed in the literature, as well as provided many details we have not seen and have had to (re-)invent for our own systems. We are especially careful to point out how to handle ray packets that do not share an origin because we expect this to become the common case for the next generation of ray tracing programs. We have not taken any position on what culling techniques are likely to be fastest, as this depends significantly on the application, scene, and implementation. As with frustum culling for GPU renderers, we expect the community to take several years to understand the merits and problems of the various methods in different applications.

Though we do not want to take a stand on what techniques are better than others (in fact, we have already shown several equivalences between the various tests), a few of our experiences with the various tests and culling methods are worth of mentioning. First, having considered all the various tests and, in particular, their commonalities has led us to a much better general understanding of the various tests. While we originally believed that the different published ray-triangle tests would be differently suited to the various culling methods, and would work differently well for the various kinds of packet types (primary, shadow, seconary, …) we found that, for example, for the corner ray case they are all equivalent, and that the different kinds of packets only influence how the corner rays are built.

Second, we found it increasingly obvious that the relation of primitive size and packet size influence the effectiveness of the techniques. For example, a packet-box culling technique applied to traversing a hierarchical index structure will work efficiently only until the traversed subtree is enclosed in the packet's frustum. At that stage, a packet would have to be split to smaller packets (similar to [22]) to remain effective. How well the different culling methods map to this packet splitting; so far, however, we believe IA to be easier to apply to this case, as no explicit planes or rays have to be (re-)constructed, and a few only elementary intervals have to be recomputed, which is both fast and conceptually simple.

Third, the difference in interval arithmetic and geometric methods (both corner rays and bounding planes) seems to lie less in their efficieny, and more in their ease of use. Generally, we found geometric tests, once derived, to be somewhat more appealing, as one can build a mental image of the geometric setups involved. This makes it easier to judge the effectiveness of a given test, or to find cases where it may be inefficient. On the other hand, such geometric images can be deceiving, as they only correspond to a specific setup, and the same test might fail for a different setup than the one imagined (this is particularly true if signs and orientations are involved). For IA, on the other hand, the geometric meaning of a test is often not obvious, but modifications of a test are easier to guarantee to remain correct. In particular, this implies that IA can more easily be used, as one can use it without requiring any geometric understanding of the situation: for example, a Plücker [11] test works in 6D space; imaging geometric bounding primitives in 6D is not straightforward, but IA is trivial to apply.

Among the geometric tests, we have found that in a ray tracer the corner ray tests often work somewhat better than culling the primitive on the frustum's planes. Though we have previously argues that both are mostly dual to each other (and thus in general have similar culling efficiency), in a real ray tracer packet-primitive intersections are not performed with random primitives, but only with those that are visited during traversal. As most acceleration structures use axis-aligned planes or bounding volumes, primitives lying outside a packet's axis-aligend bounding planes are likely to have been culled by the traversal, anyway, but testing a set of corner rays with the side of a primitive may still work if this side is not axis aligned. For example, a triangle inside a box will cover at most 50% of that box's projected area, and a packet hitting that bounding box can have a high chance of missing the triangle; a corner ray test would most likely detect this case, but the frustum plane test would not (the frustum test could only detect a miss if the bounding box of the triangle were outside the frustum, in which case that box would likely have been culled during traversal).

In summary, all of the invidiual methods have their merits, and we do not want to point to any as being better than the other (in fact the three can be combined as each handles cases that the others may not find). Instead we hope this paper will be useful to system designers exploring the option space for the culling components of their programs. In particular, we hope that this will incite new specific culling tests for specific applications or special cases that have not yet been conceived.

References

[1] John Amanatides and Andrew Woo. A Fast Voxel Traversal Algorithm for Ray Tracing. In *Eurographics '87*, pages 3–10. Eurographics Association, 1987.

[2] Jeff Arenberg. Ray/Triangle Intersection with Barycentric Coordinates. *Ray Tracing News*, 1(5), 1988. http://www.acm.org/tog/resources/RTNews/html/rtnews5b.html.

[3] James Arvo and David Kirk. A survey of ray tracing acceleration techniques. In Andrew S. Glassner, editor, *An Introduction to Ray Tracing*. Academic Press, San Diego, CA, 1989.

[4] Didier Badouel. An Efficient Ray Polygon Intersection. In David Kirk, editor, *Graphics Gems III*, pages 390–393. Academic Press, 1992. ISBN: 0124096735.

[5] Carsten Benthin. *Realtime Ray Tracing on current CPU Architectures*. PhD thesis, Saarland University, 2006.

[6] Solomon Boulos, Dave Edwards, J. Dylan Lacewell, Joe Kniss, Jan Kautz, Peter Shirley, and Ingo Wald. Interactive distribution ray tracing. Technical Report UUSCI-2006-022, SCI Institute, University of Utah, 2006.

[7] James H. Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554, 1976.

[8] J. L. D. Comba and J. Stolfi. Affine arithmetic and its applications to computer graphics. In *Proceedings of SIBGRAPI*, pages 9–18, 1993.

[9] Robert Cook, Thomas Porter, and Loren Carpenter. Distributed Ray Tracing. *Computer Graphics (Proceeding of SIGGRAPH 84)*, 18(3):137–144, 1984.

[10] Kirill Dmitriev, Vlastimil Havran, and Hans-Peter Seidel. Faster Ray Tracing with SIMD Shaft Culling. Research Report MPI-I-2004-4-006, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 2004.

[11] Jeff Erickson. Pluecker Coordinates. *Ray Tracing News*, 1997. http://www.acm.org/tog/resources/RTNews/html/-rtnv10n3.html#art11.

[12] Christiaan P. Gribble, Thiago Ize, Andrew Kensler, Ingo Wald, and Steven G. Parker. A Coherent Grid Traversal Approach to Visualizing Particle-based Visualization Data. Technical Report UUSCI-2006-024, SCI Institute, University of Utah, 2006. (submitted for publication).

[13] Vlastimil Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, Faculty of Electrical Engineering, Czech Technical University in Prague, 2001.

[14] Brian Hayes. A lucid interval. *American Scientist*, 91(6):484–488, 2003.

[15] Paul S. Heckbert and Pat Hanrahan. Beam tracing polygonal objects. In *Proceedings of SIGGRAPH*, pages 119–127, 1984.

[16] James T. Hurley, Alexander Kapustin, Alexander Reshetov, and Alexei Soupikov. Fast ray tracing for modern general purpose CPU. In *Proceedings of GraphiCon*, 2002.

[17] Frederic Jansen. Data structures for ray tracing,. In *Proceedings of the Workshop in Data structures for Raster Graphics*, pages 57–73, 1986.

[18] Michael Kaplan. The uses of spatial coherence in ray tracing. In *ACM SIGGRAPH '85 Course Notes 11*, July 1985.

[19] Christian Lauterbach, Sung-Eui Yoon, David Tuft, and Dinesh Manocha. RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs. Technical Report 06-010, Department of Computer Science, University of North Carolina at Chapel Hill, 2006.

[20] Marta L ofsted and Tomas Akenine-M oller. An Evaluation Framework for Ray-Triangle Intersection Algorithms. *Journal of Graphics Tools*, 10(2):13–26, 2005.

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

[22] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. Multi-level ray tracing algorithm. In *Proceedings of ACM SIGGRAPH*, pages 1176–1185, 2005.

[23] Steve Rubin and Turner Whitted. A 3D representation for fast rendering of complex scenes. In *Proceedings of SIGGRAPH*, pages 110–116, 1980.

[24] John M. Snyder. Interval analysis for computer graphics. In *Proceedings of SIGGRAPH*, pages 121–130, 1992.

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

[26] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics*, (conditionally accepted, to appear), 2006.

[27] Ingo Wald, Thiago Ize, Andrew Kensler, Aaron Knoll, and Steven G. Parker. Ray tracing animated scenes using coherent grid traversal. *ACM Transactions on Graphics*, 25(3):485–493, 2006. (Proceedings of ACM SIGGRAPH).

[28] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum*, 20(3):153–164, 2001. (Proceedings of Eurographics).