Follow up to "Fast Ray-Axis Aligned Bounding Box Overlap Tests with Plücker Coordinates"

Technical report 2004-759-24
(updated January 2005)

Jeffrey Mahovsky
mahovskj@cpsc.ucalgary.ca
University of Calgary

(This technical report has been submitted to *Ray Tracing News* and will appear in an upcoming issue.)

Abstract:

This report summarizes further work on the ray-box overlap test presented in the Journal of Graphics Tools paper titled "Fast Ray-Axis Aligned Bounding Box Overlap Tests with Plücker Coordinates." Performance measurements from an implementation in an actual ray tracer are presented and compared with the results from the JGT paper. A more efficient bounding volume hierarchy traversal scheme is also presented that improves performance by up to 243%. The effectiveness of a common bounding volume hierarchy construction heuristic is evaluated, and comparisons are made with the K-D tree method. Numerical robustness issues are also discussed.

Introduction:

In the Journal of Graphics Tools paper titled "Fast Ray-Axis Aligned Bounding Box Overlap Tests with Pluecker Coordinates," appearing in Volume 9, Number 1, I presented a new method for performing ray-box overlap tests. In this article, I compare the method implemented in a ray tracer as a bounding volume hierarchy (BVH). (The JGT paper did not test the box intersection routines in an actual ray tracer.) I also present a simple method for testing child nodes in the proper order along the ray, providing up to a 243% performance increase for some scenes.

An electronic copy of the JGT paper can be found at:
http://www.8dn.com/~jm/papers/jgt-mahovsky.pdf

Details of the Pluecker coordinate-based algorithm are given in the JGT paper, but a quick summary is that the edges comprising the silhouette of the box are tested against the ray by determining their relative orientations to the ray. To work, the test is split into eight separate cases based on the signs of the ray direction vector components. These cases are labeled MMM, MMP, MPM, MPP, PMM, PMP, PPM, and PPP. (Unfortunately, this results in a lot of code. However, the eight cases are identical except the variables are rearranged.) In the JGT paper, the algorithm was also adapted to

eliminate the need to store the box in Pluecker coordinate form, which consumed additional memory.

The JGT paper also shows that splitting the traditional ray-box tests into eight optimized cases also improves their performance. Hence, comparisons will be made with the 'eight cases' versions of these tests.

One drawback to the Pluecker ray-box test is that it doesn't produce an intersection distance along the ray. This is a problem if the nodes are to be tested in the order of their distance along the ray. An alternative, effective method is presented in part 3 of this article.

1) Numerical Robustness

One issue that I neglected to discuss in the JGT paper is numerical robustness. The traditional ray-box test has a problem where if a ray is parallel or nearly parallel to a box face, values of +/- infinity or extremely large values are generated. These are the result of performing division operations where the divisor is a ray direction component. It is possible to use multiplication by an inverse ray direction component instead, but the inverse may still be +/- infinity or large value.

Brian Smits suggested in a previous JGT paper [1] that this might not be a problem because IEEE floating point numerics can operate on +/- infinity values properly and the algorithm will behave correctly. It is preferable to avoid these situations entirely, however.

The Pluecker method does not perform any division, being composed of only additions, subtractions, and multiplications. This makes it easy to determine bounds on the results of the operations. The computation results are guaranteed to be within a fixed range, given the range of the ray direction vector components (always [-1,0, 1.0]) and the range of the box coordinates (the scene boundaries, which can be easily scaled to fit in a unit cube.)

This makes the Pluecker method suitable for a fixed-point or integer hardware implementation. This should provide significant savings over using hardware floating-point, for several reasons. Firstly, integer computations can be faster than FP due to their simplicity. Secondly, integer arithmetic logic is generally smaller than FP logic, allowing more ray-box intersectors on a chip and increased performance due to greater parallelism.

The Pluecker method is not without potential problems, however. The ray-edge tests can be unstable when their results are very close to zero. The test for a ray and an edge parallel to the x-axis is:

```
float yb = node->ymax - ray->y;
float zb = node->zmax - ray->z;
if(ray->k * yb - ray->j * zb < 0) then MISS
(else try the other five silhouette edges, and if none of them miss,
the box is hit)
```

(See the sample code in section 6 for the complete version that tests all six edges.)

First, consider the case where the ray is parallel to the edge (and the x-axis). In this case, ray->k and ray->j are both zero and the result of the (`ray->k * yb - ray->j * zb`) computation is exactly zero. The box may still be hit with a zero result for this test, so it is important that only results < 0 are considered misses. The robustness problem occurs when the ray is nearly parallel with ray->j and ray->k being tiny but nonzero. Floating point imprecision may cause the computed result to be negative, when it should be positive. Thus the ray will falsely miss the box and an incorrect pixel may result.

Another problem occurs when the ray intersects the edge or is very near the edge, but is not parallel to the edge (imagine two lines intersecting in space.) Thus the result can be very close to zero, and may be negative when it should be positive due to FP imprecision.

The solution to both of these problems is to change the expression to:

```
if(ray->k * yb - ray->j * zb < -EPSILON) then MISS
```

where EPSILON is a suitably small value. This makes the ray-edge test conservative, trading false misses for false hits. False hits may result in slight excess hierarchy traversal but this cannot result in incorrect pixels. In my code, I don't use an epsilon value as I have not observed any incorrect pixels in my images from simply comparing with zero. This doesn't mean it can't happen, though.

There are a few other potentially unstable situations that occur with both the Pluecker and regular BVH traversal methods. The first situation is when a bounding box has zero volume. This can happen, for example, when a bounding box contains a single triangle that's parallel to a coordinate axis. Ray-box tests may not behave correctly when testing boxes that are actually only rectangles. Another potential problem is when the box has non-zero volume, but an object is fully contained within a box face. This raises the issue of what is considered to be 'inside' the box. Is 'inside' in the x-axis direction, for instance, defined as [xmin, xmax] or (xmin, xmax)?

These problems can be handled with careful programming, but it is better to avoid them in the first place. The solution is to slightly increase the size of the box by adding/subtracting an epsilon value from the box coordinates. I use a value of 1e-6 for a scene scaled to fit a [-1, +1] coordinate system, and this is probably a generous value. This expands the box so it is guaranteed to enclose all of its objects (such as the triangle contained within the box face), and ensures no boxes have zero volume. This increases the amount of box traversal by a tiny amount from false hits.

Note that expanding the box also solves the problem of the uncertainty when a ray nearly intersects an edge. It does not solve the uncertainty that results when the ray is nearly parallel to an edge.

2) Performance Tests

One question arising from the JGT paper is the performance of the Pluecker coordinate method when used in an actual ray tracer. My tests confirm that it is still the fastest method, but the improvement is less than shown in the JGT paper.

The fastest traditional ray-box test in the JGT paper was the 'smits-mul-cls' test. This is a slabs-based box intersection test that computes distances along the ray to the planes of the box faces and forms three distance intervals, all of which must overlap for the box to be hit. Multiplication by ray direction component inverses instead of division is used for computation of distances to box planes. The technique is optimized by splitting the algorithm into eight separate cases. Also, the properties of IEEE floating-point are relied upon to properly handle the +/- infinity values that may arise, hence the 'smits' classification.

The Pluecker coordinate method tested here is 'pluecker-cls', where eight ray classifications are used, but no Pluecker coordinates are stored – rather just the six values representing the coordinates of the box corners. Storing actual Pluecker coordinates increases storage requirements, and the JGT paper shows that the benefit is too small to be worthwhile.

Several scenes varying from thousands to millions of triangles were tested. Only triangles were used, but there are no restrictions on the types of geometry contained by the BVHs. BVH construction is loosely based on Chapter 9 of the book "Realistic Ray Tracing" (second edition) [2]. The BVH was constructed in a top-down fashion (similar to a K-D or BSP tree), not bottom-up. Each BVH branch node has two children, constructed from sets of objects divided by a splitting plane perpendicular to an axis.

Images were rendered at 2048x2048 resolution using single-precision FP. Downsized versions are available at http://www.8dn.com/~jm/papers/plu-bvh/

The testing environment was a PC with a Pentium 4 3.0GHz CPU, 2GB RAM, running Windows XP, and using Visual C++ .NET 2003.

SCENES:

Teapot: Polygonal Utah teapot with mirrored background. 2,262 triangles, 2 lights. From: http://www.cs.northwestern.edu/~watsonb/teaching/351/models.html

Spheres: Triangle mesh spheres - solid, reflective, and refractive. 24,578 triangles, 2 lights.

Bunny: Psychedelic reflective Stanford bunny in a colored box. 69,463 triangles, 2 lights. From: http://graphics.stanford.edu/data/3Dscanrep/

Branch: Branch model from a U of Calgary graduate student (Mrs. Julia Taylor-Hell). 312,706 triangles, 2 lights.

Poppy: Poppy model from a U of Calgary graduate student (Mr. Peter MacMurchy). 395,460 triangles, 2 lights.

Powerplant-front: Front view of UNC powerplant model. 12,748,512 triangles, 1 light. From: http://www.cs.unc.edu/~geom/Powerplant/

Powerplant-north: North/bottom view of UNC powerplant model. 12,748,512 triangles, 1 light.

Powerplant-boiler: View inside the boiler of the UNC powerplant. 12,748,512 triangles, 1 light.

Lucy: Angelic statue from the Stanford repository. 28,057,792 triangles, 2 lights. From: http://graphics.stanford.edu/data/3Dscanrep/

RESULTS:

Scene	**smits-mul-cls**	**pluecker-cls**
Teapot	20.4s	18.3s (11% faster)
Spheres	24.9s	22.3s (12% faster)
Bunny	37.5s	33.3s (13% faster)
Branch	21.8s	19.1s (14% faster)
Poppy	14.8s	13.3s (11% faster)
Powerplant-front	91.5s	78.9s (16% faster)
Powerplant-north	264.2s	210.9s (25% faster)
Powerplant-boiler	529.2s	447.3s (18% faster)
Lucy	62.3s	53.1s (17% faster)

An improvement of 11% to 25% in rendering times was achieved in the above results table. This is a good result, however it is less than that predicted by the JGT paper.

The JGT paper's single precision results for the Pentium 4 show that 'pluecker-cls' is between 52% and 65% faster than 'smits-mul-cls.' (The amazing 93% figure quoted in the paper's abstract is for single precision 'pluecker-cls-cff' vs. 'smits-mul-cls', and only for 1 out of the 3 tests with that combination. I didn't use 'pluecker-cls-cff' because it uses too much memory.)

In the JGT paper, the algorithms were benchmarked by pre-generating a set of random boxes and a set of random rays (thousands of each). These were stored in arrays, called rays[] and aabbs[] (axis-aligned bounding boxes). A loop compared rays[i] with aabbs[i], linearly stepping through both arrays. This is extremely efficient because data is contiguous in memory and is accessed sequentially, letting the CPU reduce some of the memory access delays with tricks such as pre-fetching.

In a ray tracer, the hierarchy nodes are typically not accessed or stored in sequential fashion, thus the CPU spends more time waiting for data to be fetched from memory. A portion of the rendering time is also spent intersecting geometry, generating rays, computing textures and shading. Profiling shows that the majority of the rendering time is spent traversing the hierarchy, however. Additionally, in the ray tracer, the traversal is implemented as recursive function calls, while the JGT tests were performed with a simple loop. This adds overhead to both 'smits-mul-cls' and the Pluecker technique, diminishing the overall improvement from faster ray-box testing. Both techniques also need to make decisions based on whether each node is a branch or a leaf, which adds more overhead that was absent from the JGT tests.

3) Correct-order Child Node Testing

I have been able to further improve the technique by using a simple method (I call DirSplitAxis or DSA) to test a node's children in the approximate order they appear along the ray. Some BVH traversal techniques always test the children in the same order, which can actually hurt performance a great deal.

The usual method for determining the order in which to test the children is to compute the ray intersection distances to each, and traverse them in that order. This is simple when using the regular ray-box test because it computes the intersection distance as part of the test. This is a problem for the Pluecker method because no intersection distances are computed.

The DirSplitAxis method does not require intersection distances, but rather uses the direction of the ray (MMM, MMP, etc.) and axis of the splitting plane that separated the children. This requires that the splitting plane axis (but not position) be stored within the node, but this requires negligible storage. (I have a signed numObjects field in my node structure: if numObjects < 0, it's a branch node and I use -1 to denote an x-axis split, -2 for y, and -3 for z.)

The child node traversal order is as follows (assuming child0 is on the negative half of the splitting plane, and vice-versa):

MMM: Test child1 then child0 for x, y, or z splitting planes.
(This does not require any comparison test.)

MMP: Test child1 then child0 for x or y
Test child0 then child1 for z

MPM: Test child1 then child0 for x or z
Test child0 then child1 for y

MPP: Test child0 then child1 for y or z
Test child1 then child0 for x

PMM: Test child1 then child0 for y or z
    Test child0 then child1 for x

PMP: Test child0 then child1 for x or z
    Test child1 then child0 for y

PPM: Test child0 then child1 for x or y
    Test child1 then child0 for z

PPP: Test child0 then child1 for x, y, or z
    (This does not require any comparison test.)

Hence, with one comparison test (for 6 out of 8 cases, the other 2 are free) the optimal child node traversal order is given. This eliminates the need to compute distances to each of the child nodes to determine the traversal order. Note that DSA will not always give the same child node ordering as sorting based on actual intersection distances, because the nodes can overlap in odd ways. Regardless of the actual intersection distances, the majority of child0's objects should be on the negative side of the partition, and the majority of child1's objects should be on the positive side. This will almost always be true even if the boxes overlap in strange ways.

To test the effectiveness of DSA, I compared the 'pluecker-cls' ray-box test (which always tests child0 before child1), to 'pluecker-dsa' which is 'pluecker-cls' using DSA to determine child testing order.

RESULTS:

Scene	**pluecker-cls**	**pluecker-dsa**
Teapot	18.3s	17.9s (2% faster)
Spheres	22.3s	19.3s (16% faster)
Bunny	33.3s	30.1s (11% faster)
Branch	19.1s	19.6s (3% slower)
Poppy	13.3s	12.3s (8% faster)
Powerplant-front	78.9s	37.3s (112% faster)
Powerplant-north	210.9s	176.2s (20% faster)
Powerplant-boiler	447.3s	130.3s (243% faster)
Lucy	53.1s	53.6s (1% slower)

The results show that for some scenes, DSA can improve performance by up to 243%. The Branch and Lucy scenes slow down by 1% to 3%. Generally, DSA seems to be worthwhile.

One obvious question is how does 'smits-mul-cls' compare to the Pluecker method if both traverse their children in the proper order? Also, how effective is DSA compared to traversing the children in the order of their actual intersection distances along the ray?

For 'smits-mul-cls', there are two options. The first is to test the ray against the two children, getting the intersection distance to each and then traversing them in order of their distance along the ray (termed 'smits-dist'.) The second option is to use DSA (termed 'smits-dsa'.) The Pluecker method only uses DSA because, unlike 'smits_mul_cls', the ray-box test does not return an intersection distance.

Here are the results, including the earlier 'incorrect' order 'smits-mul-cls' and 'pluecker-cls' results for comparison:

RESULTS:

Scene	**smits-mul-cls**	**smits-dist**	**smits-dsa**	**pluecker-cls**	**pluecker-dsa**
Teapot	20.4s	20.2s	19.3s	18.3s	17.9s
Spheres	24.9s	22.2s	21.5s	22.3s	19.3s
Bunny	37.5s	38.8s	33.0s	33.3s	30.1s
Branch	21.8s	21.4s	21.5s	19.1s	19.6s
Poppy	14.8s	13.2s	13.5s	13.3s	12.3s
Powerplant-front	91.5s	42.7s	41.2s	78.9s	37.3s
Powerplant-north	264.2s	263.5s	216.7s	210.9s	176.2s
Powerplant-boiler	529.2s	157.5s	156.0s	447.3s	130.3s
Lucy	62.3s	66.0s	60.4s	53.1s	53.6s

The results show that 'smits-dsa' equals or outperforms 'smits-dist' on 7 of 9 tests. DSA is particularly effective on Powerplant-north. The Pluecker ray-box test is not disadvantaged by not returning ray-box intersection distances, since DSA seems to be superior to using the actual distances.

The results also confirm that the Pluecker method is still the fastest, even when testing nodes in the correct order.

4) Surface Area Heuristic / Cost Function

The 'pluecker-dsa' results can be further improved by building a better hierarchy. One effective way to do this is to use the Surface Area Heuristic (SAH) [3]. A cost function is used to determine the optimal splitting axis and optimal location of the splitting plane. The optimum is achieved when the cost function is minimized. Unfortunately, minimization of the cost function is very expensive computationally, because many combinations of splitting axes/planes need to be tried. Child nodes are formed when the cost of splitting a node exceeds that of simply leaving it alone. (Wald presents an excellent discussion of the SAH's application to BSP/K-D trees in his thesis [4]. Its application to BVHs is nearly identical.)

In my implementation, I try candidate splitting planes that are evenly spaced within the node along the x, y, and z axes. This does not always find the lowest cost, but the 'correct'

method requires an exhaustive search which places the splitting plane at the center point of each object (resulting in quadratic run-time complexity.) (Center points are used for the BVH because the center points determine which child node an object belongs to.) Despite only being an approximate minimization, the improvement in rendering speed can be large.

There are two cost function parameters that affect the construction of the hierarchy: costTraverse and costTriIntersect. Only one of these is actually needed because the other can be assumed to be 1.0. I chose the parameter values such that rendering time was roughly minimized without wasting memory.

The BVH for the regular 'pluecker-dsa' tests was constructed in a simple fashion. The splitting axis was always an x, y, z, x, y, z... sequence and was positioned to divide the node into two equal halves. This type of hierarchy construction is extremely fast. Construction times for the 12M triangle Powerplant scenes were only 30 seconds, and 78 seconds for the 28M triangle Lucy scene.

The results for the SAH version of pluecker-dsa are presented below:

Scene	**pluecker-dsa**	**pluecker-dsa-sah**
Teapot	17.9s	14.3s
Spheres	19.3s	13.5s
Bunny	30.1s	19.0s
Branch	19.6s	16.1s
Poppy	12.3s	10.5s
Powerplant-front	37.3s	22.7s
Powerplant-north	176.2s	73.4s
Powerplant-boiler	130.3s	62.9s
Lucy	53.6s	50.5s

The rendering speed more than doubles for two of the Powerplant scenes, demonstrating the importance of building a good hierarchy. Unfortunately, construction times were vastly increased, now taking over 15 minutes for the Powerplant scenes and nearly an hour for Lucy. A smarter cost-minimization technique might be able to reduce the preprocessing times.

5) BVH vs. K-D Tree

For comparison, I have also implemented a K-D tree, and a K-D tree using the SAH. Like the Pluecker case, I chose cost function parameters to roughly minimize rendering time. Construction of a BVH and K-D tree is similar, with both methods proceeding in a recursive top-down fashion. Nodes are split into two children after an appropriate splitting plane is chosen.

(For detailed analysis of K-D trees and hierarchy construction, see Havran's thesis [5]. Havran also compared the K-D tree to the BVH, but the BVH he used was constructed

with a bottom-up technique and performed poorly. The BVH here uses more effective top-down construction.)

These results are not meant to be a rigorous comparison of BVHs vs. K-D trees, but rather to show that BVHs, if efficiently implemented, can compete with K-D trees. There are many things that affect the performance of spatial hierarchies, and a thorough comparison of BVH vs. K-D is too large for this article. Performance is also very sensitive to hierarchy construction parameters. A thorough comparison would vary the parameters for each method to show the tradeoffs between construction time, memory usage, and rendering speed.

Render times for the three cases are presented in the table:

Scene	**pluecker-dsa-sah**	**K-D**	**K-D-sah**
Teapot	14.3s	16.6s	14.4s
Spheres	13.5s	17.2s	13.3s
Bunny	19.0s	24.7s	21.9s
Branch	16.1s	23.9s	17.7s
Poppy	10.5s	13.7s	12.3s
Powerplant-front	22.7s	20.8s	14.7s
Powerplant-north	73.4s	87.8s	54.6s
Powerplant-boiler	62.9s	64.0s	42.6s
Lucy	50.5s	58.3s	-

From the results, it is clear that K-D trees benefit greatly from the SAH, especially for complex scenes. This is similar to the findings of other researchers.

'K-D-sah' has a significant advantage versus the BVH when rendering the complex Powerplant scene, but the other results are much closer. I didn't implement any advanced K-D tree traversal optimizations such as SSE instructions, coherent ray tracing, iterative traversal, or try to optimally pack the K-D tree nodes into memory. These optimization techniques can also be applied to the BVH, however.

The excellent 'K-D-sah' results came at a cost, however. Memory consumption was similar or greater for both 'K-D' and 'K-D-sah' versus the BVH. (I was unable to generate a result for the Lucy scene for 'K-D-sah' because the system began to swap to disk. The other two Lucy tests came close to running out of memory as well, for both BVH and K-D.) This high K-D tree memory usage was a surprise, because a K-D tree node can be stored in only 8 or 12 bytes of memory, while a BVH node typically requires 32 bytes. There are two reasons for the greater memory consumption:

a) K-D trees seem to have more nodes than a BVH (even when using the SAH.)

b) In a K-D tree, an object can belong to multiple leaf nodes, but in a BVH, an object only belongs to one leaf node. Thus, a K-D tree leaf node needs a list of pointers to objects. In the BVH, object pointer lists are unnecessary provided the node's objects are

stored contiguously in memory. Thus, only a single pointer is needed to point to the first object. (This implies swapping objects around in memory during BVH construction, but construction time was actually slightly faster versus maintaining pointer lists.)

For the Powerplant and Lucy scene K-D trees, more memory was consumed by leaf node object pointer lists than by the actual K-D tree nodes. This large 'hidden cost' is easy to overlook.

K-D tree memory consumption can be reduced by adjusting the hierarchy or cost function parameters, but this will increase rendering time - reducing the speed advantage of the K-D tree.

Construction times were also much greater for the K-D tree, but my construction code is not well-optimized for speed. Then again, neither is the BVH construction code. The slower performance could be partly due to the greater number of nodes in the K-D trees, thus more calls to the recursive tree construction code were made. Also, tri/node membership tests are more complex for the K-D tree. In a BVH, nodes are constructed based on the bounding box of the objects. In a K-D tree, it is not sufficient to merely use the object's bounding box to determine whether it belongs to a child node. The object's bounding box may overlap the node, but the object itself may not. Hence, a tri/box test is used to solve this problem [6].

Another problem is that the effective number of objects being processed by the construction code 'grows' as the K-D tree is constructed. If an object is found to overlap both of a node's children, it is added to the object list for both child's subtrees. Thus, it effectively becomes two objects from the construction code's point of view. In a BVH, the object only belongs to one child's subtree, preventing an increase in the effective number of objects to process.

To give an idea of the difference in construction times, the BVH (not SAH) construction times for Powerplant and Lucy were 30 seconds and 78 seconds. The K-D (not SAH) construction times were 615s and 736s. It would be challenging to make up for this factor of 10-20 difference by code optimization alone. A more advanced K-D construction algorithm might be able to achieve this, but the need for an advanced algorithm could be seen as a disadvantage, given that the unoptimized BVH construction code is already simple and efficient.

6) Sample Code

Below is sample code that implements 'pluecker-dsa' traversal for MMP rays. The details of the ray-box test are explained in the JGT paper. A complete version of the code can be found at:
http://www.8dn.com/~jm/papers/plu-bvh/bvhcode.cpp

```
void Traverse_MMP(BBoxNode *node, Ray *ray)
{
   // Perform the Pluecker-coordinate derived ray-box test
   // ray->ex, ray->ey, ray->ez are the coordinates of the ray's endpoint
   // ray->t is the distance to the endpoint along the ray, -1 if the ray is
```

```
   // infinitely long

   if ((ray->x < node->xmin) || (ray->y < node->ymin) || (ray->z > node->zmax) ||
      ((ray->t != -1) && ((ray->ex > node->xmax) || (ray->ey > node->ymax) ||
      (ray->ez < node->zmin))))
      return;  // ray misses the box

   float xa = node->xmin - ray->x;
   float ya = node->ymin - ray->y;
   float za = node->zmin - ray->z;
   float xb = node->xmax - ray->x;
   float yb = node->ymax - ray->y;
   float zb = node->zmax - ray->z;

   if ((ray->i * ya - ray->j * xb < 0) ||
       (ray->j * xa - ray->i * yb < 0) ||
       (ray->k * xb - ray->i * zb < 0) ||
       (ray->i * za - ray->k * xa < 0) ||
       (ray->j * za - ray->k * ya < 0) ||
       (ray->k * yb - ray->j * zb < 0))
      return;  // ray misses the box

   if(node->numTris < 0)      // branch node
   {
      if(node->numTris == -3)   // DSA: z-split
      {
         Traverse_MMP(&node->children[0], ray);
         Traverse_MMP(&node->children[1], ray);
      }
      else                      // DSA: x or y split
      {
        Traverse_MMP(&node->children[1], ray);
        Traverse_MMP(&node->children[0], ray);
      }
   }
   else  // leaf node, call function that tests the ray against the triangles
   {
      TriIntersect(node, ray);
   }
}
```

References:

[1] Brian Smits. "Efficiency issues for ray tracing." Journal of Graphics Tools, 3(2):1-14, 1998.

[2] Peter Shirley and R. Keith Morley. "Realistic Ray Tracing, second edition." AK Peters Limited, 2003.

[3] MacDonald, J. David and Booth, Kellogg S. "Heuristics for Ray Tracing Using Space Subdivision." Proceedings of Graphics Interface '89, pages 152-163, 1989.

[4] Ingo Wald. "Realtime Ray Tracing and Interactive Global Illumination." PhD Thesis, Saarland University, 2004.

[5] Vlastimil Havran. "Heuristic Ray Shooting Algorithms." PhD thesis, Czech Technical University, 2000.

[6] Tomas Akenine-Möller. "Fast 3D triangle-box overlap testing." Journal of Graphics Tools, 6(1):29-33, 2001.