Ray Tracing Bounding Volume Hierarchies with the Pluecker-AABB Test

Technical report 2004-759-24
(April 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 investigates ray tracing with bounding volume hierarchies using the Pluecker-AABB test that was presented in the Journal of Graphics Tools paper titled "Fast Ray-Axis Aligned Bounding Box Overlap Tests with Pluecker Coordinates." An efficient scheme for testing a bounding volume node's children in the correct order is presented, improving performance by up to 233%. Numerical robustness and BVH construction issues are also discussed.

Introduction:

In the Journal of Graphics Tools paper titled "Fast Ray-Axis Aligned Bounding Box Overlap Tests with Pluecker Coordinates," in Volume 9, Number 1, I presented a new Pluecker coordinate-based method for performing ray-box overlap tests. Here, I compare the Pluecker-AABB test 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 233% performance increase for some scenes. Numerical robustness and BVH construction issues are also discussed.

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

Details of the Pluecker-AABB test 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. 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, but 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 a memory-inefficient Pluecker coordinate form.

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-AABB test is that it doesn't produce an intersection distance along the ray. This can be 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 and special cases.

The traditional "slabs" ray-box test [1] can have problems when a ray is parallel to an axis/box face, because values of +/- infinity are generated. Recall that the "slabs" test 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. The distances are produced by dividing by the ray direction components (for axis-aligned boxes.) It is possible to use multiplication by inverse ray direction components instead, but the inverses may still be +/- infinity. Fortunately, IEEE floating point numerics can operate on +/- infinity values properly and the algorithm should behave correctly [2].

Dealing with infinite values is not the only problem. The "slabs" algorithm computes distances to the planes of the box faces by subtracting the ray origin from the face coordinates, and then dividing by a ray direction component. If both the subtraction result and the direction component are zero, then a 0/0 situation happens which produces a NaN (Not a Number.) The NaN result might cause a failure of the algorithm, since unlike +/- infinity, it is impossible to properly compare values to a NaN. All comparisons with NaN produce a false result, even when comparing NaN to itself. (Interestingly, one way to check for x == NaN is to compare x == x and check for a false result. This is not recommended because a compiler might not understand this trick and optimize the expression out of the code. Fortunately, an isnan() function is available on many platforms.) If multiplying by inverse direction components, a 0 * Inf situation happens, which is also NaN. These problems can be avoided by handling axis-parallel rays as a special case, or by ensuring the ray-box test handles NaNs properly (one could exploit the fact that comparison with NaN always produces false.) Other traversal algorithms that compute distances along the ray to planes (such as k-d trees) must also consider these +/-Inf and NaN issues.

The Pluecker-AABB test 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.) Thus, no infinity or NaN values can be produced.

The Pluecker-AABB test is not without potential problems, however. The ray-edge tests in the Pluecker method are still subject to FP imprecision. For example, one of the ray-edge tests 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 5 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. This is not a box miss because the ray can still pass through the box when the ray is parallel to an axis. Thus, the algorithm works correctly for axis-parallel rays provided the comparison is a "less-than zero" and not a "less-than-or-equal-to zero."

Another potential 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. Fortunately, such edge- and corner-grazing intersections are uncommon and the geometry enclosed by the box is seldom touching the box edges or corners. Thus, even if the box is accidentally missed when it should have been hit, there would probably not be any geometry in that part of the box anyway. (The reader should note that the "slabs" test can also have FP precision problems with edge- and corner-grazing intersections.)

There are a few other potentially unstable situations that occur with both the Pluecker and "slabs" 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. The ray-box tests may not behave correctly when testing boxes that are actually only 2-D 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. One 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 5e-7 for a scene that is scaled to fit a [-1, +1] coordinate system. This expands the box so it is guaranteed to fully and unambiguously enclose all of its objects (such as the triangle contained within the box face), and ensures no boxes have zero volume.

The value of 5e-7 is an educated guess, and may be generous. Recall that the machine epsilon for 32-bit FP is approximately 6e-8, which is the effective coordinate system resolution near the coordinate value of 1.0. (The resolution increases as numbers approach zero, but we're interested in the worst-case scenario.) Hence, epsilon should be at least the machine epsilon value of 6e-8. My testing showed no significant advantage in rendering time or BVH efficiency when using the smaller 6e-8 epsilon value, so I decided to err on the side of caution and use 5e-7. If the epsilon value is too large, it will reduce the effectiveness of the hierarchy because the boxes will have been expanded by too much. This causes "false hits" of boxes and excess hierarchy traversal.

The epsilon value of 5e-7 assumes a scene with limits that fit within the [-1, 1] coordinate system along all 3 axes. If the scene is larger, the epsilon value must be scaled accordingly.

Note that expanding the box also partially addresses the problem of the uncertainty when a ray grazes an edge or corner by providing a margin of safety. I am unsure if all false misses due to edge- and corner-grazing intersections can be eliminated simply by expanding the box, or the amount of box expansion that is necessary to eliminate the problem. This remains to be investigated.

2) Efficient BVH Construction

The BVH construction for the performance tests in Section 3 and 4 is loosely based on Chapter 9 of the book "Realistic Ray Tracing" (second edition) [3]. The BVH was constructed in a recursive, top-down fashion, similar to a k-d or BSP tree. Each BVH branch node has two children. Children are constructed by splitting the set of triangles enclosed by the node into two sets, based on comparing each triangle's center point to a splitting plane. The splitting plane is perpendicular to the x, y, or z axis, and its orientation alternates in round-robin fashion as the BVH is constructed. Each node's bounding box is the minimum bounding box that encloses all of its objects or children.

Hierarchy termination criteria is 6 or fewer objects per node, and a maximum depth of 60 levels. The value of 6 objects per node was experimentally determined to both minimize rendering time and use memory efficiently. Values of less than 4 increased rendering time, as did values greater than 6. Values of 4-5 slightly improved speed but significantly increased memory usage. A value of 6 results in approximately half as many BVH nodes as triangles in the scene, according to the results in Section 3.

None of the scenes reached the maximum hierarchy depth of 60. Limiting the tree depth to a lower number did not seem to improve performance. The BVH algorithm used does not seem prone to "running away" and producing hierarchies of excessive depths.

When constructing a BVH, it is important to avoid empty child/leaf nodes which waste memory and cause excess BVH traversal. There is a simple and effective strategy to avoid this problem. If subdividing a node along an axis produces an empty child, move on to the next axis and try again. If subdividing along all 3 axes produces empty children, then perform an arbitrary split by putting half of the objects in each child, regardless of their positions. With luck, the two children will be able to be subdivided properly. I have observed up to a 30% improvement in rendering speed by using this strategy to avoid empty nodes.

Skinny boxes are also a problem. If a box is too skinny along an axis, move on to the next axis. A box should be at least 2 * epsilon or 1e-6 wide along an axis before attempting a split along that axis. If a box is too skinny along all 3 axes, the node is made a leaf node. Note that the box width used here is the actual width of the box, and not the padded width which has the 5e-7 epsilon value added/subtracted to it as described in Section 1.

It is possible to apply heuristics to BVH construction, such as the Surface Area Heuristic [4]. To date, the use of heuristics has been analyzed in detail for k-d or BSP trees with good results [5]. Detailed investigation has not been performed for the BVH, although my own experiments show that the benefit is similar to that achieved for k-d trees.

I have provided code that implements this BVH construction technique at:
http://www.8dn.com/~jm/papers/plu-bvh/bvhcode.cpp

3) Performance Tests

One question arising from the JGT paper is the performance of the Pluecker-AABB test 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 Pluecker-AABB test used here is called "pluecker-cls" in the JGT paper, and is abbreviated as "plu-cls" in this article. In this method, 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 showed that the benefit is too small to be worthwhile.

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 ray direction cases. The individual cases are simpler than the regular "slabs" algorithm that must handle all eight cases of ray directions. The properties of IEEE floating-point are relied upon to properly handle the +/- infinity values that may arise. NaN values have no effect in the "smits-mul-cls" code. The comparisons are structured such that NaN values are ignored, since comparisons with NaN always evaluate to false.

Several scenes varying from thousands to millions of triangles were tested. BVH construction is described in Section 2.

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 with 512K cache and HyperThreading, 800 MHz bus, 2GB RAM, running Windows XP, and using Visual C++ .NET 2003.
Compiler flags were:

```
/O2 /GL /G7 /GA /D "WIN32" /D "NDEBUG" /D "_WINDOWS" /D "_MBCS" /FD /EHsc /ML /GS /arch:SSE2 /Fo"Release/" /Fd"Release/vc70.pdb" /W3 /nologo /c /Wp64 /Zi /TP
```

The results presented here are only valid for this particular combination of CPU, compiler, and compiler flags. Significantly different results might be obtained on different systems or with different settings. I chose reasonable settings that maximized performance, such as "/O2 /G7 /arch:SSE2."

SCENES:

Teapot: Polygonal Utah teapot with mirrored background. 1,171 BVH nodes, 2,262 triangles, 2 lights. Preprocessing time: 0.01s
From: http://www.cs.northwestern.edu/~watsonb/teaching/351/models.html

Spheres: Triangle mesh spheres - solid, reflective, and refractive. 11,339 BVH nodes, 24,578 triangles, 2 lights. Preprocessing time: 0.04s

Bunny: Psychedelic reflective Stanford bunny in a colored box. 33,377 BVH nodes, 69,463 triangles, 2 lights. Preprocessing time: 0.31s
From: http://graphics.stanford.edu/data/3Dscanrep/

Branch: Branch model from a U of Calgary graduate student (Mrs. Julia Taylor-Hell). 155,257 BVH nodes, 312,706 triangles, 2 lights. Preprocessing time: 1.30s

Poppy: Poppy model from a U of Calgary graduate student (Mr. Peter MacMurchy). 187,551 BVH nodes, 395,460 triangles, 2 lights. Preprocessing time: 1.64s

Powerplant-top: Top view of UNC powerplant model. 6,374,789 BVH nodes, 12,748,512 triangles, 1 light. Preprocessing time: 30.3s
From: http://www.cs.unc.edu/~geom/Powerplant/

Powerplant-front: Front view of UNC powerplant model. 6,374,789 BVH nodes, 12,748,512 triangles, 1 light. Preprocessing time: 30.3s

Powerplant-north: North/bottom view of UNC powerplant model. 6,374,789 BVH nodes, 12,748,512 triangles, 1 light. Preprocessing time: 30.3s

Powerplant-boiler: View inside the boiler of the UNC powerplant. 6,374,789 BVH nodes, 12,748,512 triangles, 1 light. Preprocessing time: 30.3s

Lucy: Angelic statue from the Stanford repository. 13,548,145 BVH nodes, 28,057,792 triangles, 2 lights. Preprocessing time: 78.2s
From: http://graphics.stanford.edu/data/3Dscanrep/

RESULTS:

Scene	**smits-mul-cls**	**plu-cls**
Teapot	19.0s	17.4s (9% faster)
Spheres	26.9s	24.6s (9% faster)
Bunny	30.0s	26.8s (12% faster)
Branch	21.3s	18.5s (15% faster)
Poppy	14.7s	13.2s (11% faster)
Powerplant-top	12.4s	11.4s (9% faster)
Powerplant-front	82.1s	70.8s (16% faster)
Powerplant-north	233s	184s (27% faster)
Powerplant-boiler	441s	366s (20% faster)
Lucy	58.6s	49.6s (18% faster)

An improvement of 9% to 27% in rendering times was achieved in the above results table. This is a good, consistent 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 "plu-cls" is between 52% and 65% faster than "smits-mul-cls."

The JGT paper's tests always used rays of infinite length. In the ray tracer, finite-length rays must also be handled. This added an additional set of comparisons to the "plu-cls" code that were not present in the JGT paper's test code. Specifically, the endpoint of a finite-length ray must also be checked against the box, not just the origin. Thus, some performance is lost due to the additional operations.

For 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.

4) Correct-order Child Node Testing

I have been able to further improve the BVH traversal performance 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 traversal method computes the distance to each child's bounding box, and traverses the closest child first. This is simple when using the "slabs" ray-box test because it computes the intersection distance as part of the test. This is a problem for the Pluecker-AABB test because no intersection distances are computed.

The DSA 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 numTris field in my node structure: if numTris < 0, it's a branch node and I use -1 to denote an x-axis split, -2 for y, and -3 for z. numTris is only positive if the node is a leaf and therefore contains triangles.)

The child node traversal order is as follows (assuming child0 is on the negative half of the splitting plane, and child1 on the positive):

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.)

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 "plu-cls" ray-box test (which always tests child0 before child1), to "plu-dsa" which is "plu-cls" using DSA to determine child testing order.

RESULTS:

Scene	**plu-cls**	**plu-dsa**
Teapot	17.4s	16.6s (5% faster)
Spheres	24.6s	21.4s (15% faster)
Bunny	26.8s	22.4s (20% faster)
Branch	18.5s	18.9s (2% slower)
Poppy	13.2s	12.1s (9% faster)
Powerplant-top	11.4s	9.00s (27% faster)
Powerplant-front	70.8s	33.9s (109% faster)
Powerplant-north	184s	153s (20% faster)
Powerplant-boiler	366s	110s (233% faster)
Lucy	49.6s	49.6s (--)

The results show that for some scenes, DSA can improve performance by up to 233%. The Branch scene slows down by 2%.

One obvious question is how does "smits-mul-cls" compare to the Pluecker method if both traverse their children in DSA 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 children, getting the intersection distance to each and then traversing them in distance order (termed "smits-dist".) The second option is to use DSA ("smits-dsa".) The Pluecker method only uses DSA because, unlike "smits-mul-cls", the ray-box test does not return an intersection distance.

The results are below, including the earlier "incorrect" order "smits-mul-cls" and "plu-cls" results for comparison:

RESULTS:

Scene	**smits-mul-cls**	**smits-dist**	**smits-dsa**	**plu-cls**	**plu-dsa**
Teapot	19.0s	18.6s	18.0s	17.4s	16.6s
Spheres	26.9s	23.7s	23.4s	24.6s	21.4s
Bunny	30.0s	27.2s	24.6s	26.8s	22.4s
Branch	21.3s	20.8s	20.9s	18.5s	18.9s
Poppy	14.7s	13.0s	13.5s	13.2s	12.1s
Powerplant-top	12.4s	8.90s	9.78s	11.4s	9.00s
Powerplant-front	82.1s	38.6s	37.8s	70.8s	33.9s
Powerplant-north	233s	231s	190s	184s	153s
Powerplant-boiler	441s	135s	133s	366s	110s
Lucy	58.6s	62.2s	56.3s	49.6s	49.6s

The results show that "smits-dsa" equals or outperforms "smits-dist" on 7 of 10 tests. DSA is particularly effective on Powerplant-north versus traversing in distance order. The Pluecker-AABB test is not disadvantaged by not returning ray-box intersection distances, since DSA seems to be equal or superior to using the actual distances.

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

5) Sample Code

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

```
void Traverse_MMP(BBoxNode *node, Ray *ray)
{
   // Perform the Plu-AABB 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

   // the comparisons could be rewritten as (ray->i * ya < ray->j * xb) etc.
   // which might be faster on some systems

   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] T. Kay and J. Kajiya. "Ray Tracing Complex Scenes." Proceedings of SIGGRAPH '86, pages 269-278.

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

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

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

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