Efficiency Issues for Ray Tracing

Brian Smits*
University of Utah

February 19, 1999

Abstract

Ray casting is the bottleneck of many rendering algorithms. Although much work has been done on making ray casting more efficient, most published work is high level. This paper discusses efficiency at a slightly lower level, presenting optimizations for bounding volume hierarchies that many people use but are rarely described in the literature. A set of guidelines for optimization are presented that avoid some of the common pitfalls. Finally, the effects of the optimizations are shown for a set of models.

1 Introduction

Many realistic rendering systems rely on ray casting algorithms for some part of their computation. Often, the ray casting takes most of the time in the system, and significant effort is usually spent on making it more efficient. Much work has been done and published on acceleration strategies and efficient algorithms for ray casting, the main ideas of which are summarized in Glassner [5]. In addition, many people have developed optimizations for making these algorithms even faster. Much of this work remains unpublished and part of oral history. This paper is an attempt to write down some of these techniques and some higher level guidelines to follow when trying to speed up ray casting algorithms. I learned most of the lessons in here the hard way, either by making the mistakes myself, or by tracking them down in other systems. Many of the observations in here were confirmed by others.

This paper will discuss some mid-level optimization issues for bounding volume hierarchies. The ray casting algorithm uses the hierarchy to determine if the ray intersects an object. An intersection involves computing the distance to the intersection and the intersection point as well as which object was hit. Sometimes it includes computing surface normal and texture coordinates. The information computed during an intersection is sometimes called the hit information. In ray tracing based renderers, rays from the eye are called primary rays. Reflected and transmitted rays are known as secondary rays. Together, these rays are called intersection rays. Rays from hits to lights to determine shadowing are called shadow rays.

*bes@cs.utah.edu

2 Principles of Optimization

Optimization can be a seductive activity leading to endless tweaks and changes of code. The most important part of optimization is knowing when not to do it. Two common cases are:

- Code or system is not run frequently.
- Code is a small fraction of overall time.

In other words, code should only be optimized if it will make a significant effect on the final system and the final system will be used frequently enough to justify the programmer's time and the chance of breaking something.

It helps to have a set of principles to follow in order to guide the process of optimization. The set I use is:

- Make it work before you make it fast.
- Profile everything you do.
- Complexity is bad.
- Preprocessing is good.
- Compute only what you need.

2.1 Make it Work Before You Make it Fast

Code should be made correct before it is made fast [2]. As stated repeatedly by Knuth [9] "Premature optimization is the root of all evil". Obviously, slow correct code is more useful than fast broken code. There is an additional reason for the rule, though. If you create a working, unoptimized version first, you can use that as a benchmark to check your optimizations against. This is very important. Putting the optimizations in early means you can never be completely sure if they are actually speeding up the code. You don't want to find out months or years later that your code could be sped up by removing all those clever optimizations.

2.2 Profile Everything You Do

It is important to find out what the bottleneck is before trying to remove it. This is best done by profiling the code before making changes [3]. The best profilers give time per line of code as well as per function. They also tell you how many times different routines are called. Typically what this will tell you is that most of the time is spent intersecting bounding boxes, something that seems to be universally true. It also can tell you how many bounding boxes and primitives are checked.

Like many algorithms, the speed will vary based on the input. Obviously large data sets tend to take more time than small ones, but the structure of the models you use for benchmarking is also important. Ideally you use a set of models that are characteristic of the types of models you expect to use.

Profiling is especially critical for low-level optimizations. Intuition is often very wrong about what changes will make the code faster and which ones the compiler was already doing for you. Compilers are good at rearranging nearby instructions. They are bad at knowing that the value you are continually reading through three levels of indirection is constant. Keeping things clean and local makes a big difference. This paper makes almost no attempt to deal with this level of optimization.

2.3 Complexity is Bad

Complexity in the intersection algorithm causes problems in many ways. The more complex your code becomes, the more likely it is to behave unexpectedly on new data sets. Additionally, complexity usually means branching, which is significantly slower than similar code with few branches. If you are checking the state of something in order to get out of doing work, it is important that the amount of work is significant and that you actually get out of doing the work often enough to justify the checks. This is the argument against the caches used in Section 4.4.

2.4 Preprocessing is Good

In many of the situations where ray casting is used, it is very common to cast hundreds of millions of rays. This usually takes a much longer time than it took to build the ray tracing data structures. A large percentage increase in the time it takes to build the data structures may provide a significant win even if the percentage decrease in the ray casting time of each ray is much smaller. Ideally you increase the complexity and sophistication of the hierarchy building stage in order to reduce the complexity and number of intersections computed during the ray traversal stage. This principle motivates Section 4.3.

2.5 Compute Only What You Need

There are many different algorithms for many of the components of ray casting. Often there are different algorithms because different information is needed out of them. Much of the following discussion will be based on the principle of determining the minimum amount of information needed and then computing or using that and nothing more. Often this results in a faster algorithm. Examples of this will be shown in Sections 4.1 and 4.2.

3 Overview of Bounding Volume Hierarchies

A bounding volume hierarchy is simply a tree of bounding volumes. The bounding volume at a given node encloses the bounding volumes of its children. The bounding volume of a leaf encloses a primitive. If a ray misses the bounding volume of a particular node, then the ray will miss all of its children, and the children can be skipped. The ray casting algorithm traverses this hierarchy, usually in depth first order, and determines if the ray intersects an object.

```
BoundingVolume BuildHierarchy(bvList, start, end, axis)
  if(end - start == 0)       // only a single bv in list so return it.
    return bvList[start]
  BoundingVolume parent
  foreach bv in bvList
    expand parent to enclose bv
  sort bvList along axis
  axis = next axis
  parent.AddChild(BuildHierarchy(bvList, start, (start + end) / 2, axis)
  parent.AddChild(BuildHierarchy(bvList, 1 + (start + end) / 2, end, axis)
  return parent
```

Figure 1: Building a bounding volume hierarchy recursively.

There are several ways of building bounding volume hierarchies [6, 10]. The simplest way to build them is to take a list of bounding volumes containing the primitives and sort along an axis[8]. Split the list in half, put a bounding box around each half, and then recurse, cycling through the axes as you recurse. This is expressed in pseudocode in Figure 1. This method can be modified in many ways to produce better hierarchies. A better way to build the hierarchy is to try to minimize the cost functions described by Goldsmith and Salmon [6].

4 Optimizations for Bounding Volume Hierarchies

4.1 Bounding Box Intersections

Intersecting rays with bounding volumes usually accounts for most of the time spent casting rays. This makes bounding volume intersection tests an ideal candidate for optimization. The first issue is what sort of bounding volumes to use. Most of the environments I work with are architectural and have many axis-aligned planar surfaces. This makes axis-aligned bounding boxes ideal. Spheres tend not to work very well for this type of environment.

There are many ways to represent and intersect an axis-aligned bounding box. I have seen bounding box code that computed the intersection point of the ray with the box. If there was an intersection point, the ray hits the box, and if not, the ray misses. There are optimizations that can be made to this approach, such as making sure you only check faces that are oriented towards the ray, and taking advantage of the fact that the planes are axis aligned [11]. Still, the approach is too slow. The first hint of this is that the algorithm computes an intersection point. We don't care about that, we just want a yes or no answer. Kay [8] represented bounding volumes as the intersection of a set of slabs (parallel planes). A slab is stored as a direction, D_s, and an interval, I_s, representing the minimum and maximum value in that direction, effectively as two plane equations. The set of slab directions is fixed in advance. In my experience, this approach is most effective when there are three, axis aligned, slab directions. This is just another way of storing a bounding box, we store minimum and maximum values

```
bool RaySlabsIntersection(ray, bbox)
   Interval inside = ray.Range()
   for i in (0,1,2)
      inside = Intersection(inside,(slab[i].Range()-ray.Origin[i])/ray.Direction[i])
      if(inside.IsEmpty())
         return false
   return true
```

Figure 2: Pseudocode for intersecting a ray with a box represented as axis aligned slabs.

along each axis.

Given this representation, we can intersect a bounding box fairly efficiently. We show this in pseudocode in Figure 2. This code isn't as simple as it looks due to the comparisons of the IsEmpty and Intersection functions and the need to reverse the min and max values of the interval when dividing by a negative number, but it is still much faster than computing the intersection point with the box.

One important thing to notice about this representation and this intersection code is that it gives the right answer when the ray direction is 0 for a particular component. In this case the ray is parallel to the planes of the slab. The divide by zero gives either $[-\infty, -\infty]$ or $[+\infty, +\infty]$ when the ray is outside the slab and $[-\infty, +\infty]$ when the ray is inside. This saves additional checks on the ray direction.

4.2 Intersection Rays versus Shadow Rays

It is important to know what kind of information you need from the ray casting algorithm in order to keep from doing more work than necessary. There are three commonly used ray casting queries: closest hit, any hit, and all hits. Closest hit is used to determine the first object in a given direction. This query is usually used for primary, reflected, and transmitted rays. Any hit is used for visibility tests between two points. This is done when checking to see if a point is lit directly by a light and for visibility estimation in radiosity algorithms. The object hit is not needed, only the existence of a hit. All hits is used for evaluating CSG models directly. The CSG operations are performed on the list of intervals returned from the all hits intersection routine.

For efficiency reasons it is important to keep these queries separate. This can be seen by looking at what happens when using the most general query, all hits, to implement the others. Any hit will simply check to see if the list of intersections is empty. Clearly we computed more than we needed in this case. Closest hit will sort the list and return the closest intersection. It may seem as if the same or more work is needed for this query, however this is usually not the case. With most ray tracing efficiency schemes, once an intersection is found, parts of the environment beyond the intersection point can be ignored. Finding intersections usually speeds up the rest of the traversal. Also, the list of hit data does not need to be maintained.

Shadow (any hit) rays are usually the most common type of rays cast, often accounting for more than 90 percent of all rays. Because of this, it is worth considering how to make them faster than other types of rays. Shadow rays need not compute any

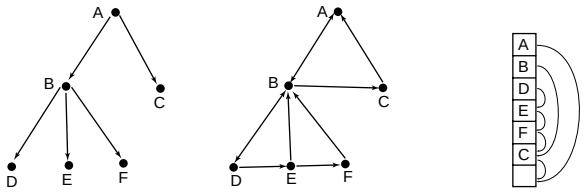

Figure 3: Three different representations for a tree. (a) Children pointers. (b) Left child, right sibling, parent pointers. (c) Array in depth-first order with skip pointers.

of the commonly needed intersection information, such as intersection point, surface normal, uv coordinates, or exact object hit. Additionally, the traversal of the efficiency structure can be terminated immediately once an intersection is guaranteed. A special shadow routine taking these factors into account can make a significant difference in efficiency.

The difference between shadow rays and intersection rays determined which acceleration scheme I use. I have tried both grids [4] and bounding volume hierarchies. In my experience (based on models I typically render) grids are a little faster on intersection rays (closest hit) and slower for shadow rays (any hit). Grids sort the environment spatially, which is good for finding the closest intersection. The bounding volume hierarchies built by trying to minimize Goldsmith and Salmon's cost function [6] tend to keep larger primitives near the root, which is good for shadow rays. It is still unknown as to which acceleration scheme is better, and it is almost certainly based on the model.

4.3 Traversal Code

Casting a ray against a bounding volume hierarchy requires traversing the hierarchy. If a ray hits a bounding volume, then the ray is checked against the children of the bounding volume. If the bounding volume is a leaf, then it has an object inside it, and the object is checked. This is done in depth-first order. Once bounding volume intersection tests are as fast as they can be, the next place for improvement is the traversal of the hierarchy. Traversal code for shadow rays will be used in the following discussion.

In 1991, Haines[7] published some techniques for better traversals. Several of these techniques used extra knowledge to mark bounding boxes as automatically hit and to change the order of traversal. In my experience these methods do not speed up the ray tracer and greatly increase the complexity of the code. This difference in experience may be due to changes in architecture over the last 8 years that make branches and memory accesses instead of floating point the bottleneck. It may also be due to faster bounding box tests. I have found that the best way to make the traversal fast is to make it as minimal as possible.

The simplest traversal code is to use recursion to traverse the tree in depth-first order. Figure 3(a) shows a hierarchy of bounding boxes. Depth first traversal means that bounding box A is tested, then box B, then the boxes with primitives D, E, and F. The idea is to find an intersection as soon as possible by traveling down into the tree. The

```
TreeShadowTraversal(ray, bvNode)
  while(true)      // termination occurs when bvNode→GetParent() is NULL
    if(bvNode→Intersect(ray))
      if(bvNode→HasPrimitive())
        if(bvNode→Primitive().Intersect(ray))
          return true
      else
        bvNode = bvNode→GetLeftChild()
        continue
    while(true)
      if(bvNode→GetRightSibling() != NULL)
        bvNode = bvNode→GetRightSibling()
        break
      bvNode = bvNode→GetParent()
      if(bvNode == NULL)
        return false
```

Figure 4: Traversal of bounding volume tree using left child, right sibling, parent structure.

biggest problem with this is the function call overhead. The compiler maintains much more state information than we need here. We can eliminate much of this overhead by changing our representation of the tree. A representation that works well is to store the left-most child, the right sibling, and the parent for each node, as in Figure 3. Using this representation we can get rid of the recursion by following the appropriate pointers. If the ray intersects the bounding box, we get to its children by following the left-most child link. If the ray misses, we get to the next node by following the right sibling link. If the right sibling is empty, we move up until either there is a right sibling, or we get back up to the root, as shown in pseudocode in Figure 4.

This tree traversal also does too much work. Notice that when the traversal is at a leaf or when the ray misses a bounding volume, we compute the next node. The next node is always the same, there is no reason to be computing it for each traversal. We can pre-compute the node we go to when we skip this subtree and store this skip node in each node. This step eliminates all computation of traversal related data from the traversal. There are still intersection computations, but no extra computation for determining where to go. This is expressed in pseudocode in Figure 5

The final optimization is the recognition that we only need to do depth-first traversals on the tree once it is built. This observation lets us store the tree in an array in depth-first order as in Figure 3. If the bounding volume is intersected, the next node to try is the next node in the array. If the bounding volume is missed, the next node can be found through the skip mechanism. We have effectively thrown out all the information we don't need out of the tree, although it is still possible to reconstruct it. The traversal code can be seen in Figure 6.

The array traversal approach works significantly better than the previous one, and has a couple subtle advantages. The first is better memory usage. In addition to the

```
SkipTreeShadowTraversal(ray, bvNode)
  while(bvNode != NULL)
    if(bvNode→Intersect(ray))
      if(bvNode→HasPrimitive())
        if(bvNode→Primitive().Intersect(ray))
          return true
        bvNode = bvNode→SkipNode()
      else
        bvNode = bvNode→GetLeftChild()
    else
      bvNode = bvNode→SkipNode()
  return false
```

Figure 5: Traversal of bounding volume tree using left child, and skip pointers.

```
ArrayShadowTraversal(ray, bvNode)
  stopNode = bvNode→GetSkipNode()
  while(bvNode < stopNode)
    if(bvNode→Intersect(ray))
      if(bvNode→HasPrimitive())
        if(bvNode→Primitive().Intersect(ray))
          return true
      bvNode++
    else
      bvNode = bvNode→GetSkipNode()
  return false
```

Figure 6: Traversal of bounding volume tree stored as an array in depth-first order.

bounding volume, this method requires only a pointer to a primitive and a pointer to the skip node. This is very minimal. Since the nodes are arranged in the order they will be accessed in, there is more memory coherency for large environments. The second advantage is that this method requires copying data from the original tree into an array. Since the original tree is going to be thrown out, it can be augmented with extra information. Depending upon how the tree is created, this extra information can more than double the cost of each node. Now there is no penalty for this information. Storing the extra information can reduce the time to build the tree and more importantly can result in better trees. The fastest bounding volume test is the one you don't have to do.

4.4 Caching Objects

One common optimization is the use of caches for the object most recently hit. This optimization and variations on it were discussed by Haines[7]. The idea is that the next ray cast will be similar to the current ray, so keep the intersected object around and check it first the next time. To the extent that this is true, caches can provide a benefit,

however rays often differ wildly. Also, cache effectiveness decreases as the size of the primitives get smaller. The realism of many types of models is increased by replacing single surfaces with many surfaces. Now caches will remain valid for a shorter amount of time.

There are two different types of caches, those for intersection (closest hit) rays and those for shadow (any hit) rays. If caches are used for intersection rays, the ray will still need to be checked against the environment to see if another object is closer. Usually the ray will again be checked against whatever object is in the cache. Mailboxes [1] can eliminate this second check (by marking each tested object with a unique ray id and then checking the id before testing the primitive). Mailboxes, however, create problems when making a parallel version of the code. Depending on the environment and the average number of possible hits per ray, the cache may reduce the amount of the environment that must be checked by shortening the ray length. In my experience, the cost of maintaining the cache and the double intersection against an object in it more than outweighs the benefit of having a cache. If your primitives are very expensive and your environments are dense, the benefit of reducing the length of the ray early may outweigh the costs, but it is worth checking carefully.

Evaluating the benefit of caches for shadow rays is more complicated. In cases where there is a single light, there tends to be a speedup as long as the cache remains full much of the time and the objects in it stay there for a long enough time. In cases where there are multiple lights we often lose shadow ray coherence because the lights are in different regions of the environment. Now each shadow ray is significantly different from the previous one. A solution for this is to have a different cache for each light.

For both types of caches, we have ignored what happens for reflected and transmitted rays. These rays are spatially very different from primary rays and from each other. Each additional bounce makes the problem much worse. If rays are allowed to bounce d times, there are $2^{d+1} - 1$ different nodes in the ray tree. In order for caching to be useful, a separate cache needs to be associated with each node. For shadow rays, that means a separate cache for each light at each node This can increase the complexity of the code significantly. Another option is to store a cache for each light on each object (or collection of objects) in the environment as discussed by Haines[7]. Note that caching only helps when there is an object in the cache. If most shadow rays won't hit anything (due to the model or the type of algorithm using the shadow tests) then the cache is less likely to be beneficial. In my experience, shadow caching wasn't a significant enough gain, so I opted for simplicity of code and removed it, although after generating the data for the result section I am considering putting it back in for certain situations. Others have found that caches are still beneficial.

5 Results

Now we look at the cumulative effects for shadow rays of the three main optimizations described in the paper. First we speed up bounding box tests. Next we speed up the traversal using the different methods from Section 4.3. We then treat shadow rays differently from intersection rays and lastly we add a shadow cache. In all of the

	1	2	3	4	5	6	7	8
theater	64	36	30	21	22	11	10	6
lab	79	41	32	22	20	12	12	7
10,000 small	415	223	191	142	110	48	50	27
10,000 mid	392	185	154	103	81	77	79	65
10,000 big	381	179	152	104	82	79	77	69
100,000 small	995	620	550	449	351	62	63	33
100,000 mid	932	473	424	324	230	146	148	89
100,000 big	1024	508	442	332	240	210	212	156
300,000 mid	1093	597	536	421	312	120	121	64

Table 1: Results of the different experiments described in the text on different environments. Times rounded to the nearest second.

experiments 1,000,000 rays are generated by choosing random pairs of points from within a bounding box 20% larger than the bounding box of the environment. In the last experiment, 500,000 rays are generated, each generated ray is cast twice, resulting in 1,000,000 rays being cast overall. The first two test cases are real environments, the rest are composed of randomly oriented and positioned unit right triangles. The number gives the number of triangles. Small, mid, and big refer to the space the triangles fill. Small environments are 20 units cubed, mid are 100 units cubed, and big are 200 units cubed. The theater model has 46502 polygons. The science center model has 4045 polygons. The code was run on an SGI O2 with a 180 MHz R5000 using the SGI compiler with full optimization turned on[1]. No shading or other computation was done and time to build the hierarchies was not included.

The experiments reported in Table 1 are explained in more detail below:

1. Bounding box test computes intersection point, traversal uses recursion, and shadow rays are treated as intersection rays.
2. Bounding box test replaced by slab version from Section 4.1.
3. Recursive traversal replaced by iterative traversal using left child, right sibling, and parent pointers as in Section 4.3.
4. Skip pointer used to speed up traversal as in Section 4.3.
5. Tree traversal replaced by array traversal as in Section 4.3.
6. Intersection rays replaced by shadow rays as in Section 4.2.
7. Shadow caching used as in Section 4.4.
8. Shadow caching used, but each ray checked twice before generating a new ray. The same number of checks were performed.

[1] -Ofast=ip32_5k

The first thing to notice is that real models require much less work than random polygons. This is because the polygons are distributed very unevenly and vary greatly in size. The theater has a lot more open space and even more variation in polygon size than the lab, resulting in many inexpensive rays and a faster average time. In spite of this, the results show very similar trends for all models. In the first 5 experiments we haven't used any model-specific knowledge, we have just reduced the amount of work done. Special shadow rays and caching are more model specific. Shadow rays are more effective when there are many intersections along the ray and are almost the same when there is zero or one intersection. Shadow caching is based on ray coherence and the likelihood of having an intersection. In experiment 7 there is an unrealistically low amount of coherence (none). In experiment 8 we guaranteed that there would be significant coherence by casting each ray twice.

6 Conclusions

The optimization of ray casting code is a double-edged sword. With careful profiling it can result in significant speedups. It can also lead to code that is slower and more complicated. The optimizations presented here are probably fairly independent of the computer architecture. There are plenty of significant lower level optimizations that can be made which may be completely dependent upon the specific platform. If you plan on porting your code to other architectures, or even keeping your code for long enough that the architecture changes under you, these sorts of optimizations should be made with care.

Eventually you get to a point where further optimization makes no significant difference. At this point you have no choice but to go back and try to create better trees requiring fewer primitive and bounding box tests, or to look at entirely different acceleration strategies. Over time, the biggest wins come from better algorithms, not better code tuning.

The results presented here should be viewed as a case study. They describe some of what has worked for me on the types of models I use. They may not be appropriate for the types of models you use.

7 Acknowledgments

Thanks to Peter Shirley, Jim Arvo, and Eric Haines for many long discussions on ray tracing. Thanks to Peter and Eric for encouraging me to write up these experiences, and to both of them and Bill Martin for helpful comments on the paper. This work was partially funded by Honda and NSF grant ACI-97-20192.

References

[1] ARNALDI, B., PRIOL, T., AND BOUATOUCH, K. A new space subdivision method for ray tracing CSG modelled scenes. *The Visual Computer 3*, 2 (Aug. 1987), 98–108.

[2] BENTLEY, J. L. *Writing Efficient Programs*. Prentice-Hall, Englewood Cliffs, NJ, 1982.

[3] BENTLEY, J. L. *Programming Pearls (reprinted with corrections)*. Addison-Wesley, Reading, MA, USA, 1989.

[4] FUJIMOTO, A., TANAKA, T., AND IWATA, K. Arts: Accelerated ray-tracing system. *IEEE Computer Graphics and Applications* (Apr. 1986), 16–26.

[5] GLASSNER, A., Ed. *An Introduction to Ray Tracing*. Academic Press, 1989.

[6] GOLDSMITH, J., AND SALMON, J. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications 7*, 5 (May 1987), 14–20.

[7] HAINES, E. Efficiency improvements for hierarchy traversal. In *Graphics Gems II*, J. Arvo, Ed. Academic Press, San Diego, 1991, pp. 267–273.

[8] KAY, T. L., AND KAJIYA, J. T. Ray tracing complex scenes. In *Computer Graphics (SIGGRAPH '86 Proceedings)* (Aug. 1986), D. C. Evans and R. J. Athay, Eds., vol. 20, pp. 269–278.

[9] KNUTH, D. E. *Literate Programming*. CSLI Lecture Notes Number 27. Stanford University Center for the Study of Language and Information, Stanford, CA, USA, 1992.

[10] RUBIN, S. M., AND WHITTED, T. A 3-dimensional representation for fast rendering of complex scenes. *Computer Graphics 14*, 3 (July 1980), 110–116.

[11] WOO, A. Fast ray-box intersection. In *Graphics Gems*, A. S. Glassner, Ed. Academic Press, San Diego, 1990, pp. 395–396.