# Notes on efficient ray tracing

Solomon Boulos
University of Utah

There are many ways to make your ray tracer faster. If you're writing an interactive ray tracer, you've got to turn to your bottlenecks in your code and make them scream. You're probably spending the majority of your time computing ray-scene intersections (in some applications, ray-scene intersection may not be the bottleneck, for example Perlin noise is commonly a performance bottleneck for applications that use it heavily). To speed up ray-scene intersections, you use acceleration structures, but how do you get that extra factor of two in performance? This document is some informal notes on experience we've had at Utah on this topic. I do not include citations here. For the sources of these techniques see the bibliography for the chapters from the second edition of *Fundamentals of Computer Graphics* included in these notes. Several papers discussing these techniques are also included in these notes.

I cover two different classes of acceleration structures and what you can do to make them even faster: bounding volume hierarchies (BVHs) and uniform grids (UGs). We do not have as much experience with BSP trees and interested readers should see the work from the University of Saarland group for BSP tree implementation techniques. I'll show code, and discuss the trade-offs involved between each choice. The code examples from the BVH section are slightly modified versions of code from *Realistic Ray Tracing, 2nd Edition*. That original code is available at http://www.cs.utah.edu/~shirley/galileo/.

## Bounding volume hierarchies

A BVH is conceptually simple. It's a tree of bounding volumes, where a bounding volume is usually an axis aligned bounding box that encloses all the surfaces you've got underneath it in the tree. An example of a simple BVH class in C++ looks like this:

```
class BVH : public Surface
{
public:
    // Constructors and such here
    BBox bbox;
    Surface* left;
    Surface* right;
};
```

As you can see, we have a bounding box for our node and pointers to our two children. To build a BVH, you choose some way to split up a list of primitives into two separate lists and put them into the left and right children as you see fit while making sure that your bounding box surrounds all the primitives. In C++ you get something like this:

```
BVH::BVH(Surface** surfaces, int num_surfaces, int axis)
{
   if (num_surfaces == 1) { *this = BVH(surfaces[0], surfaces[0], axis); return; }
   if (num_surfaces == 2) { *this = BVH(surfaces[0], surfaces[1], axis); return; }

   // surround all the objects in the list
   bbox = surround(surfaces, num_surfaces);
   Vector3 pivot = (bbox.max() + bbox.min()) / 2.0;

   // split up the primitives and tell me where the end of the left node is
   int mid_point = qsplit(surfaces, num_surfaces, pivot[axis], axis);

   // create a new bounding volume
   int next_axis = (axis + 1) % 3;
   left  = buildBranch(surfaces, mid_point, next_axis);
   right = buildBranch(&surfaces[mid_point], num_surfaces - mid_point, next_axis);
}
```

This constructor takes a list of Surface pointers and an axis, and produces a BVH. You include the axis parameter so you can switch which axis you split the primitives along. The *qsplit* function called here is similar to the way a standard qsort works, except that we only have to move objects to one side of a splitting plane (the pivot point). Again, the choice of construction algorithm is entirely up to you and the performance of your BVH depends strongly upon it, but it is an open question as to how you might build an optimal BVH (or at least something that performs really well for a variety of situations). *buildBranch* is essentially a copy-and-paste from this default constructor except you can return something other than a BVH pointer for those first cases (e.g. return surfaces[0] if there is only one object).

One of the nicest things about the BVH is how simple it is to intersect with a ray:

```
bool BVH::hit(Ray &r, HitRecord& rec, Context& context)
{
    if ( !(bbox.rayIntersect(r, r.tmin, r.tmax))) return false;

    bool isahit1 = left->hit(r, rec, context);
    bool isahit2 = right->hit(r, rec, context);
    return (isahit1 || isahit2);
}
```

From this code we can see that we first test a ray against our bounding volume. If we don't hit the bounding volume, we immediately return false. If we do hit the bounding volume we recurse. You may have just realized we're about to do a lot of bounding box intersection tests. Currently, the best method I know of asks for a little bit of extra storage in your Ray class but gives a substantial improvement in performance (Williams et al. 2005). There is also a recent JGT submission discussing a Ray-box test using Plücker coordinates, but we have not implemented this algorithm ourselves.

So those are the basics of BVH. How do we make it better? Assuming you think your construction is rock solid, but you just wish the traversal were faster, the first question is probably "why left before right, why not right before left?" This is a very good question. In fact, if you switch between left and right you'll even notice a difference for some scenes. What if we could choose the side based on something we know about the ray? Since we were already using the Williams bounding box test, which required us to store

bitwise values that determined whether or not we are going in the positive or negative x,y and z directions, we use this to our advantage. The BVH node changes slightly:

```
class BVH : public Surface
{
public:
    // Constructors and such here
    BBox bbox;
    Surface* child[2];
    int split_axis;
};
```

and the hit function changes similarly:

```
bool BVH::hit(Ray &r, HitRecord& rec, Context& context)
{
    if ( !(bbox.rayIntersect(r, r.tmin, r.tmax))) return false;

    bool isahit1 = child[r.posneg[(split_axis*2)]]->hit(r, rec, context);
    bool isahit2 = child[r.posneg[(split_axis*2)+1]]->hit(r, rec, context);
    return (isahit1 || isahit2);
}
```

Here *r.posneg* stores a 0 if the ray is moving to the right in that axis and a 1 otherwise. In our experience this modification gives a non-trivial performance benefit over either static choice (left then right or right then left). Alternatively, if we switch the order of traversal, we perform worse than either of the static choices. It should be noted that this modification is essentially an algorithmic change in traversal. You're trying to find the earliest intersection in a scene, so this algorithm chooses the node that would be "in your way". If you're going to the right, it first checks the left node, and if you're going left it first checks the right node.

Other researchers have tried other things like reordering the nodes in depth first search order to obtain higher memory coherence (Smits 1998). There have been other discussions of how to choose a splitting axis, but the most commonly used scheme is that shown here: to start with some axis and cycle through the axes in order. More discussion about these issues can be found in the *Ray Tracing News* (http://www.acm.org/tog/resources/RTNews/html/rtn_index.html#spatial).

## Uniform grids

The UG is also conceptually simple. Take your list of objects, build a big box around them then cut it up into smaller boxes. When a ray hits your grid, you iteratively traverse your grid using a 3D-DDA algorithm (Woo 87). Grid traversal has been covered in great detail, and the basic thing to remember to do is to avoid recomputing anything you don't need to during the inner most loop. Grid construction has also been discussed by many researchers and the best resource for any of this is the ray tracing news. Once you've got a basic grid implementation, the question is how to make it faster. First, if you're adding adding geometry to grid cells because their bounding boxes overlap, you're paying a high price without a good reason. Most likely, your large scene has at least some number of triangles in it. An excellent code for box-triangle overlap is on Tomas Akenine-Möller's web site. It is faster and more stable than previous methods and very simple

to add to your code library. I strongly recommend that any object you are inserting into your grid is tested to make sure it actually overlaps your grid cell. This simple change gave a 15-17% boost in performance for a simple scene with the Stanford bunny. Other Box-object tests also exist, and a list of them can be found on the *Real-Time Rendering* website (http://www.realtimerendering.com/int).

Most grid implementations have some sort of way they store their grid data, for example a 3D array of lists of pointers. In a similar manner to the common Matrix-Matrix multiply optimization, you get very different results based on how you traverse this data due to the memory layout. If for example, you had a 3D array such as this:

```
Surface* data[nx][ny][nz];
```

you would pay very little cost in memory penalties for traversing in the z direction, but a very large cost for traversing in the x direction (and this only gets worse as your memory requirements increase). In ray tracing, and more so in path tracing, rays are bouncing in all sorts of directions. You could definitely layout your memory for a particular view if you wanted to, but doing this for each view is incredibly costly (and could almost certainly never be done interactively). Instead, it is better to arrange your data in a bricked fashion so that you never pay a huge cost in stride for any direction you travel. You won't necessarily do as well for the rays that would have been at ideal cost, but you won't do nearly as poorly for the rays that would have had the worst cost possible.

Again, a lot of improvement can also be gained at the algorithmic level. If we instead use a hierarchy of uniform grids (unfortunately there is no standard term for this in the literature) we can reduce the size of the object lists in each cell. Automatically generating a grid to perform well is essentially black magic, but without any explanation of how to do it, you can achieve up to 40% improvements in run time from simply building a new uniform grid whenever a cell is too densely populated. For example, if you build a grid by having a 3D array of lists of object pointers, you could do a pass over the grid after you've built it and check for lists that are say longer than 16 elements. In any such cell, you could take that list of objects and turn it into a new grid. This would be a particularly simple implementation, and seems to work pretty well in practice.

Another improvement involves maximizing cache coherence. For example, if you allocate a pointer for each object as you insert it into the grid, you will cause a large amount of fragmentation within your grid. If instead in a first pass you created a "grid" holding the number of objects that overlap a cell (instead of pointers to the objects that will eventually go there) and then allocated a big chunk of memory you can remove the penalty of fragmentation (you then loop over your grid again plugging in values for the pointers). This technique may also reduce your construction time despite the two passes due to the reduction in memory allocation calls (system calls always cost a fortune).

## Grids vs BVHs

So the question now is which one to use? Or should you use a BSP? The short answer is that it depends. The long answer involves explaining what it depends on. The correct answer is that nobody really knows. But I'll give the long answer.

There are a few different issues that warrant some (mostly high-level) discussion, including very large scenes, object distribution and material properties. We'll talk about each of these issues in turn, and remember that for the most part this discussion assumes that each of your acceleration structures is implemented equally well (which may or may not be true in practice as people have very different mileage for each data structure).

## Large scenes

Large scenes are those which are simply not possible in a 32-bit address space. A scene including the David model from Stanford (the model file alone is 1.1GB) would be a good example. In the 64-bit address space pointers are now 8 bytes long to allow you to address all that memory. The impact for you is that your data structures may now suddenly require twice as much storage.

Instead of using pointers we can store a list of objects that we wish to access and index into them using an appropriately sized integer value. For example, as long as you have less than $2^{32}$ objects in your scene, you can get away with a simple 4 byte unsigned integer. In the general case, you only need $n$-bit indices, where $n$ is such that $2^n$ is greater than the number of objects you need to index. Unless memory was really tight, I'd recommend sticking with the simple integer.

This brings up a common technique whenever you have lots and lots of instances of a data structure: make it smaller. For example, a common representation for a KD-Tree node would contain two pointers to child nodes, an integer for the split axis and a floating point position of the splitting plane. This leads to at least a 24 byte structure on a 64-bit machine. Ingo Wald has demonstrated a more efficient representation requiring only 8 bytes of storage. This improves cache line reuse and greatly reduces memory requirements, and his representation does so without a loss in accuracy.

## Object distribution

So let's say you have a big list of primitives (spheres, polygons, etc), what kind of an acceleration structure should you put them into? I find this question is best answered by looking at each data structure separately and then comparing them afterwards.

A BVH is ideal for sparse scenes. When you build a BVH, you have the ability to group the objects into two separate clusters that may be separated by large portions of space. Also, if the bounding boxes of your node's children (the left and right children's bounding boxes) don't overlap you can get an instant stopping criteria. For example, assume that you have a ray entering from the right and some objects clustered as shown in Figure 1. If you were to test the right box first, you'd find the first intersection and produce a shorter ray, which would no longer hit the left box. This exit early condition is not possible if the boxes overlap a lot because even the clipped ray will still hit the other box. So one of the biggest weaknesses of the BVH is when you have geometry with strong overlap.

What does this mean to you? For a dense mesh, such as the Stanford Buddha, your BVH may not perform as well as it would for a scene composed with the same number of non-overlapping primitives. This doesn't mean it won't perform pretty well, but there's definitely room for improvement. The take home message: BVHs are very natural for sparse scenes, since you can take advantage of early exits but maybe you should use something else for dense regions of your scene.

In contrast to the BVH, the Uniform Grid is meant for dense data. When you build a uniform grid, you usually dice up the overall bounding box into equally sized cells, which means that for a sparse scene, you have a lot of empty cells, which you'll still end up checking when you go along intersecting (although you still move pretty quickly through them). Wasting time moving through empty cells, and worse yet spending any amount of storage on empty cells is a problem for uniform grids. To avoid this problem, you can make cells bigger so that you jump over more empty space more quickly, but then you have some cells with lots of primitives inside of them. This problem is commonly referred to as the "teapot in the stadium problem", where you have a high resolution version of the Utah teapot in the center of a large low resolution stadium. This scene would have a very large bounding box, but to obtain a suitable grid resolution for the teapot you might have to use very small cells.

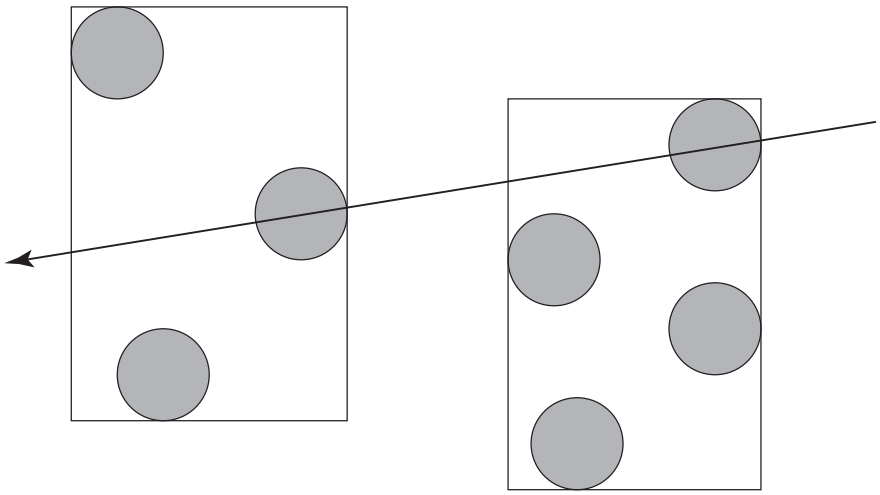

Figure 1: A ray coming from the right should test the right subtree first.

The common solution for this is not to put things into any acceleration structure blindly. Most likely, you have a high level understanding that your teapot is an object on its own and could make a uniform grid out of this object and then place the teapot-grid into a BVH in combination with the stadium (thus taking advantage of the sparse structure of the stadium). How do you do this automatically? There have been lots of papers, but this is largely an unanswered question. One of the largest barriers to answering the question is that there isn't a suitable set of test scenes to test the performance of acceleration structures. The SPD scenes have been useful as a a ray tracing benchmark, but are no longer representative of the types of scenes you would want to render in a modern rendering system.

The take home message: grids work very well for dense data such as meshes and volumes, but you pay a price for traversing and storing the empty cells. There has not been much work in adaptive resolution grid structures in ray tracing (although there were a handful between 1987 and 1997), but the basic idea involves automatically isolating dense regions of space and putting them into a structure and then placing the result into a coarser representation (or a different structure entirely, such as a BVH).

## Ray casting versus ray tracing

This is not a commonly discussed problem with these different acceleration structures, but in practice is incredibly important. For example, some acceleration structure papers have only considered ray casting (sending primary rays from the observer towards the scene) which involves no secondary bounces. This usually means that all rays are starting well outside the acceleration structure and are very coherent (traveling in the same direction and likely to touch adjacent memory). A more interesting situation occurs when we consider rays that start on or inside the acceleration structure.

The two data structures we've discussed above perform quite differently for what I'll call a "starting cost." For example, for a uniform grid, you can determine in constant time the grid cell you are in when you start a ray inside the grid. For a BVH, you usually provide a ray to the top level node and traverse down the hierarchy, despite the fact that you might know you're inside the acceleration structure already. As you consider larger and larger scenes, the height of the hierarchy continues to grow and suddenly the $\log n$ traversal starting cost becomes larger and larger. This applies to all hierarchical data structures.

This basic problem leads to an optimization present in some interactive ray tracers: if you don't allow

objects to be placed inside your dielectrics, you can avoid a scene intersection test for transmitted rays and only perform a test against the dielectric object. This is an interesting optimization because it offers a huge performance benefit for large scenes containing dielectrics (imagine a glass coffee table in a complex scene). An open question would be how to take advantage of this property automatically, without requiring the user to tell you that nothing is inside the space you're interested in testing. This problem comes up any time you have rays entering the model, but again as I mentioned the same issue is true when you're sending secondary rays from off of the model as well. It would be interesting to see more research on data structures that might be able to take advantage of these situations. For the most part the uniform grid already achieves this due to its negligible (constant time) startup cost, so the simplest solution might be to investigate how to create adaptive resolution hierarchical grids, so that you can avoid the empty cells.

## Summary

We've discussed two of the most common data structures for accelerating ray-scene queries. Hopefully some of the basic optimizations such as memory layout ( data bricking, compact data structures ) and algorithmic optimizations ( early exits, precomputed results ) came across clearly as they can greatly improve the performance of your renderer.

There are a lot of other solutions out there, but at Utah we've found the advice given here to be fairly useful in practice. All of the techniques apply to parallel code as well, and we haven't spent any time considering optimizations that only work on a single processor (e.g. mailboxing). For the most part, the basic rules of optimization always hold: optimize the portions of the code that show up in profiling, always consider improving your algorithm and getting it right is more important than making it fast.