CS 314: Accelerating Ray-Object Intersections

Robert Bridson

October 14, 2008

1 Building a Bounding Volume Hierarchy

Last time we introduced the idea of finding a bounding box around all our geometry, and intersecting rays against that box first: if they don't hit the box, they can't hit the geometry and thus we can skip potentially a lot of tests. Unfortunately, in most situations this doesn't actually help too much, since usually the camera is in the scene, surrounded by geometry, and thus every ray will start within this bounding box and must hit it.

However, we can do better by breaking up the geometry into smaller parts, and put a bounding box around each part. If we break it up well enough, we are likely to find that rays will miss at least some of the boxes permitting us to save time by skipping lots of tests.

There's clearly a trade-off here though: if we split the geometry up too much, in the limit putting a separate bounding box around each individual triangle or sphere or whatever, then we'll end up doing at least as many intersection tests as we did before. We need to find a useful intermediate ground between one overly large bounding box, and too many small bounding boxes.

The particular approach we'll take in this course (amongst many that have been tried out in graphics) is to build what's called a **Bounding Volume Hierarchy** (BVH), which is fairly reliable for automatically finding a good trade-off. We'll begin with a bounding box for all the geometry, but then split it into smaller parts with their own bounding boxes, which are split further, and so on recursively. Intersecting a ray with the geometry can proceed down the tree: check against the root, and if it hits check the children, and so on recursively.

In other words, we will construct a tree of bounding boxes:

- The root of the tree is a bounding box containing all geometry.

- Each non-leaf node is a bounding box containing some geometry; that geometry is split between several children nodes.
- The tree ends at the leaves, which are bounding boxes containing a single triangle each.

In practical terms, it may be more efficient to cut the tree off a little near the root (making it a forest instead of a tree) and near the leaves (so the leaves are bounding boxes containing a small number of triangles each, instead of just one), but for now we'll leave it as is.

The main unanswered question is the second bullet point: how do we split up the geometry in a node between children? There are several possibilities here that are all good (though also many that are bad); we'll go through perhaps the simplest that still can work quite well.

1.1 Using the Tree

Before we get into further details on how to build the tree, we need to emphasize how we are going to use the tree: otherwise we won't be clear on what we should aim for.

Our basic approach for intersecting a ray against the geometry in the scene is to use a tree traversal: check a ray against the root bounding box, and only if it hits do we proceed recursively to the children, eventually getting to just those leaf nodes (i.e. actual triangles) that are likely to intersect the ray.

This could be done with recursive function calls, but in a language like C++ which isn't particularly well-tuned for recursion, it's probably a better idea to use an explicit stack instead to do a depth-first traversal. We'll start with the root node on the stack and then proceed as follows:

- While the stack is not empty:
  - Pop the top off the stack and check this bounding box against the ray.
  - If the ray hits it:
    - If the node is a leaf, check the ray against the triangle contained in it.
    - Otherwise, push the children of this node onto the stack.

We'll improve this algorithm a bit later with some important details, but this gives you an idea of how it's going to work.

In particular, it's clear that we want to build a tree which overall reduces the number of bounding boxes we need to test. The two important aspects to this are:

1. Make sure the tree isn't too high/deep (so we can get to the leaves quickly).

2. Lower the probability we have to check a ray against multiple children (which could exponentially increase the number of tests we have to perform).

We'll tackle the first point first.

1.2 Balancing the Tree

You should remember from building trees in other contexts—data structures for dictionaries for example—that to keep the height of a tree small you need it to be well balanced. For now we'll stick with *binary* trees (where each node has at most two children), though somewhat higher branching factors can be extremely worthwhile. For a binary tree to be balanced we want every non-leaf node to have two children, and for those two children to have roughly the same size (number of leaf descendents). In other words, we want to split the geometry of a node in half when determining its children.

With this notion of balance, a tree built on n triangles will have height $O(\log n)$, and we can hope then that most rays will only need $O(\log n)$ tests to determine where they intersect.

1.3 Splitting up Geometry

The second criterion is that we should lower the probability a ray will hit more than one child of a node: if a ray only hits one child of every node then we will definitely only perform $O(\log n)$ intersection tests, but if it hits every child of every node we're back to $O(n)$ tests which is too slow.

An example of a bad way to split the geometry would be to pick triangles at random to be in one half or the other. Most likely the two halves of the geometry would then be thoroughly intermingled, and their bounding boxes will mostly overlap. If a ray hits one of those bounding boxes, it will likely hit the other, and we're in trouble.

In particular, note that most likely no matter how we split the geometry, the children's bounding boxes will overlap to some degree. However, the size of the overlap region is going to be instrumental in determining if a ray might hit both boxes: if a ray hits the overlap, it must hit both boxes. There's no guarantee it won't hit both boxes even if it misses the overlap, but that's our only chance for efficiency, so it pays to try to make this overlap region as small as possible.

A natural (and the simplest) choice for reducing this overlap is to pick a splitting value along some coordinate axis. For example, if we split along the y axis, we could carefully choose a $\bar{y}$ value and then put every triangle above the plane $y = \bar{y}$ in one child and the rest in the other child. Most likely

there will be some triangles which straddle the plane; to make this concrete we could say we'll base the decision on the **centroid** or **centre of mass** of each triangle (i.e. the average of its vertex coordinates): if the centroid's y coordinate is above $\bar{y}$ it goes in the first child. The triangles that straddle the $\bar{y}$ plane will cause the bounding boxes of the children to overlap; as long as triangles are reasonably well-shaped (i.e. not abnormally tall in the y direction) then we can expect this overlap to be small relative to the two bounding boxes.

This strategy, splitting based on a selected coordinate, exposes two more questions: what splitting value should we choose, and which axis should we split along? From the previous section on balancing trees, it's clear we should pick the **median** as the splitting value, i.e. a value which divides the triangles in two equal or nearly equal subsets: algorithms such as **quickselect** can find the median for us in expected linear time, so this is quite reasonable (simply guessing halfway through the bounding box of the node along that axis is even easier, but may not produce nearly as balanced a tree in some cases). The question of the axis is a little more delicate.

We want to minimize the overlap region. We'll assume, for lack of better knowledge, that its width along the axis we choose is going to be roughly the same for each axis. The size of the overlap is then (probably) determined by the cross-sectional area of the big bounding box orthogonal to the axis. We can minimize this by choosing the axis where the bounding box has the biggest extent, i.e. according to which of `xmax-xmin`, `ymax-ymin` and `zmax-zmin` is the biggest.

1.4 Summary

Putting these choices together into an algorithm, we get a recursive algorithm that begins with all of the geometry in the scene for the root note (which again, can be unwrapped using an explicit stack if desired):

- Find the bounding box of the geometry attached to this node, and store it in this node of the tree.
- If the geometry is just a single triangle, store it here and stop.
- Otherwise, pick the axis (x, y or z) with the longest extent for the bounding box.
- Find the median value of the centroids of the triangles along this axis.
- Partition the triangles to the ones below and above this median, splitting them in half (or nearly half): if some triangles "tie" with their centroid exactly at the median, break the ties arbitrarily so that we do get a split in half.
- Recursively process each half as child nodes of the current node.

Since the work in each node is linear in the number of nodes it contains, and the tree is well balanced, this is an $O(n \log n)$ algorithm for constructing the tree.

1.5 Alternatives

Some other options people have tried for constructing the tree include using different bounding volumes other than axis-aligned boxes (such as spheres or oriented boxes), splitting along arbitrary planes instead of axis-aligned planes, or building the tree structure bottom-up from mesh information (though this isn't always possible to do efficiently) and calculating the bounding volumes in the tree in a bottom-up fashion.

Another approach is to instead split space up into regions (e.g. with a regular grid of boxes) and store pointers to a triangle in every region it overlaps; the Binary Space Partition (BSP) method you can find in the textbook is another example of this.

Unfortunately we don't yet know of a good, practical algorithm for speeding up ray intersection tests that gives provably good performance in a worst-case analysis: virtually any algorithm of practical interest can be made to fail miserably with carefully crafted input geometry. For the algorithm we've suggested, a cylinder approximated with many very long thin triangles, at a $45°$ angle to all the axes, will slow down ray intersection to $O(n)$ time as an example.

2 Intersecting a Ray with a Bounding Box

Before we can make use of the bounding volume hierarchy we've built, we need to fill in a missing ingredient: how to intersect a ray with a bounding box (to know whether we need to check geometry inside the box).

The most elegant approach I know of for this operation is built on the idea that a bounding box is the set-intersection of three **slabs**. A "slab" in this context is the region of space between two parallel planes. The bounding box $[x_0, x_1] \times [y_0, y_1] \times [z_0, z_1]$ is the set-intersection of the slab defined by $x_0 \leq x \leq x_1$, the slab defined by $y_0 \leq y \leq y_1$, and the slab defined by $z_0 \leq z \leq z_1$.

We can take the infinite line containing a ray, and work out the interval along it which intersects a slab: those values of t where $\vec{o} + t\vec{d}$ lies between the planes. Reviewing the ray-plane intersection test we developed last time makes it clear how to do this check, with the proviso that in the tricky case of a ray parallel to the planes the interval will either be the entire real line (if $\vec{o}$ is between the planes) or empty (otherwise) and we need to make sure we make the right special case decisions there.

The interval of t-values where the ray intersects the bounding box, then, is the set-intersection of these slab intervals. If this is empty, or only includes negative t values, the ray doesn't intersect the box.