Static Aabb Trees

jodavis42@gmail.com

Static Aabb Trees

Sometimes geometry doesn't move

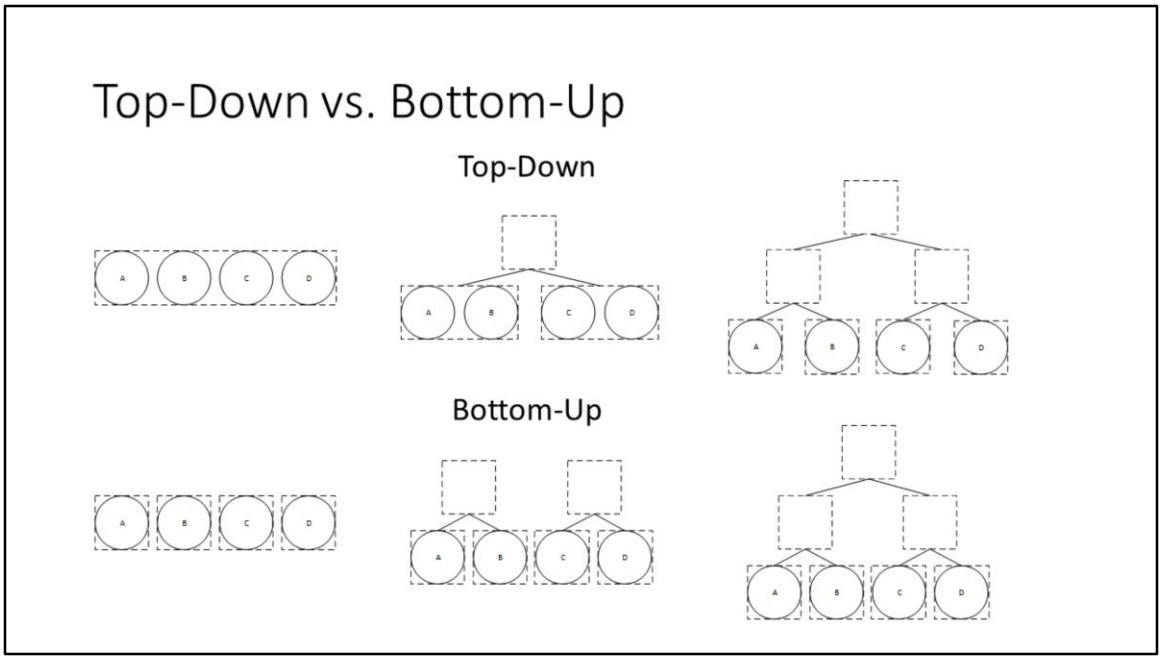
Spend more time to make a better fit tree

2 Approaches:

Top-Down, Bottom-Up

Sometimes we have geometry that doesn't move and we can afford to spend more time to make a better tree (often performed offline). Theoretically if we group our objects differently we could create a more ideal tree based upon some metric (ray-casting, aabb tests, etc…). That being said I'd still always start with a dynamic aabb tree and only optimize if necessary as they tend to be good fit.

So given a group of leaf node aabbs, how can we go about making a better tree? There are 2 main categories of static tree building: Top-Down and Bottom-Up.



Top-Down trees start by grouping all nodes into one aabb. It then splits the data set into the two child aabb nodes. This process recurses either until no more splits are possible or some pre-determined stopping criteria (volume, surface area, tree depth, etc…)

Bottom-up algorithms work in the opposite order. We start with a whole bunch of leaf nodes and then we group them together one pair at a time. This continues until we end up with one node at the root.

Top-Down construction tends to be the more popular method of constructing a static tree as it is easier to implement, however it tends to produce worse trees than bottom-up construction. While static tree building tends to be done in a pre-processing step, it's still important to try to minimize the tree construction time otherwise it'll take too long for a large number of primitives.

Top-Down Construction

```
Node* TopDownConstruction(Array<SpatialPartitionData>& insertionData)
{
  if(ShouldTerminate(insertionData))
    return new Node(insertionData);

  Array<SpatialPartitionData> left, right;
  PartitionObjects(insertionData, left, right);

  return new Node(TopDownConstruction(left), TopDownConstruction(right));
}
```

How do we split?

Top-down construction works by choosing some method to split the data set into two at each level and then recursing down each side until some pre-determined termination condition. Hence a typical algorithm would look something like this. As mentioned before, the stopping condition can range from having only 1 node, reaching some minimum volume, max tree depth, and so on.

The only real thing to discuss here is how to go about splitting the data set.

Top-Down: Splitting

Typically find 2 pieces of data:

- Split axis
- Split point

To find a split plane we need two pieces of data: the split plane's axis (or normal) and the point of the plane. We obviously can't test all of them so we need some form of heuristic to reduce the possibilities down to a reasonable number. That being said we typically try several options at each level and then choose the best.

Top-Down: Split Axis

Possible Axes:

- Cardinal Axes
- Bounding Volume's axes (e.g. K-Dop's axes)
- Axis of most variance
- Etc…

There are quite a few good choices for a split axis. The simplest choices are the 3 cardinal axes. These axes are especially good for aabbs. For other bounding volumes the axes of that volume are another good choice (say k-dops). Some other good choices include the parent bounding volume's axes, the axis through the two most distance points, and the axis of greatest variance.

For our purposes we'll use the 3 cardinal axes as they will cover a good set of axes as well as provide a good fit for aabbs.

Top-Down: Split Point

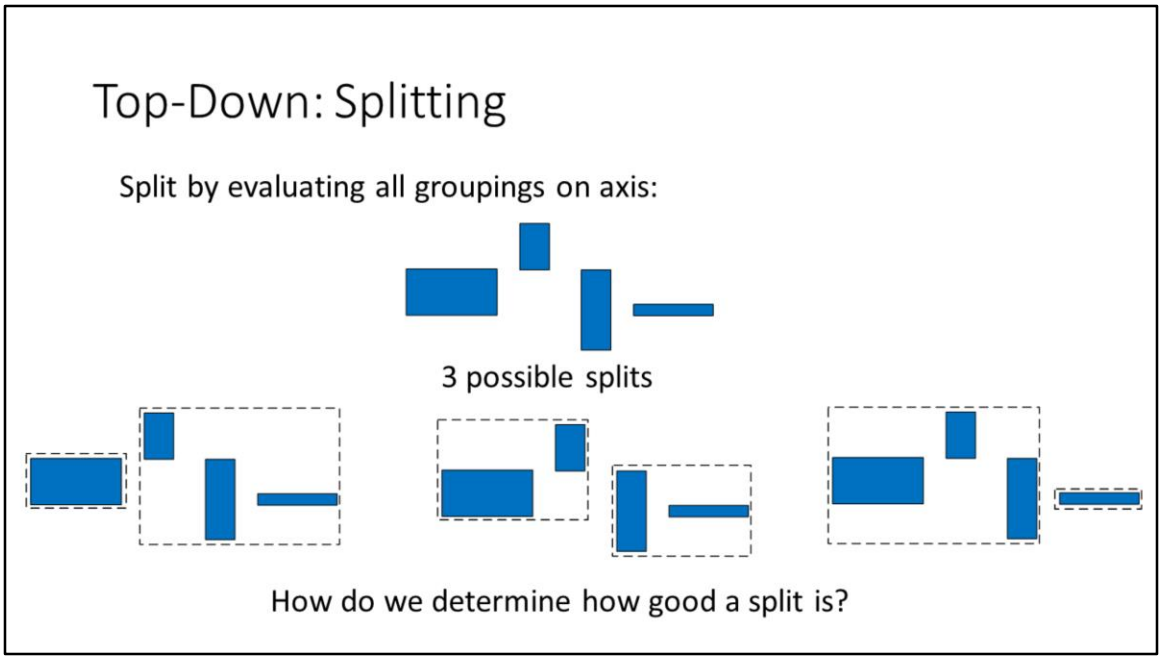
Using object centroids:

- Object Median
- Mean
- Spatial Median
- Even split along axes

Before choosing the split plane's position, we have to decide how to classify our objects against the split plane. The simplest method is to use the aabb's centroid to classify it as either in-front or behind the plane. Once we do this we need to choose where to try to place our plane.

One of the simplest methods is to sort the objects along the given axis and then evenly split them into two sets (object median). This will produce a balanced tree, but is a balanced tree actually the best? Another method is to pick the mean of the object centroids so as to more evenly divide the space. We can also choose to just evenly divide the space by picking the spatial median which is basically just the median of the parent's aabb.

Another method that relies a bit more on brute force is to just evenly subdivide the space the objects span into n subdivisions and test a plane at each point. The idea is that instead of spending a lot of time trying to intelligently guess a good split point we could just test a few and find something good enough.



Another approach of choosing a split point is to not look for an explicit point, but rather to sort along our axis and then test each grouping. We can then iterate through all groupings on the axis and score them with a heuristic and pick the best one.

Kay-Kajiya Median Cut Heuristic

Cost is proportional to surface area

$$Cost_i = \frac{\sum_0^i S_j}{S_t} * i + \frac{\sum_{i+1}^n S_j}{S_t} * (n - i - 1)$$

One such heuristic is the Kay-Kajiya median cut heuristic. The idea is that the probability a ray will hit a node is based upon its surface area. In this formula S_i is the surface area of aabb i and S_t is the total surface area of the node (the surface area of all of the aabbs). This formula basically tries to find the most even surface area split while accounting for how many objects get sent down each side.

Now it might seem expensive to compute the surface area of the aabbs of each grouping. When iterating through an array it is easy to combine one aabb with another though so we can just do one pass to compute the aabbs where we group from left to right and another to compute the aabbs going right to left. From here we can evaluate this formula for each split and choose the best one.

An aabb tree using this would test the X, Y, and Z axes and choose whichever one produced the best cost value.

Bottom-Up Construction

```
Node* BottomUpConstruction(Array<Node*> nodes)
{
  while(nodes.size() != 1)
  {
    size_t i, j;
    FindTwoNodesToMerge(nodes, i, j);

    // Merge the two nodes and fix-up the array
    MergeNodes(nodes, i, j);
  }

  // The last node is the new root
  return nodes[0];
}
```

How do we find two nodes to merge?

Just like with top-down construction, the basic algorithm of a bottom-up tree construction is simple. At each given step we find 2 nodes to merge and then merge them. When we only have 1 node left then we're done.

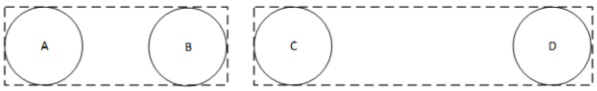
The tricky part is how do we pick which 2 nodes to merge. We could at each level check every pairing. This is $O(n^2)$ which when combined with iterating up through the tree produces an algorithm that is $O(n^3)$ which is too expensive for practical use.

Bottom-Up

Sort on an axis and group neighbor pairs

Produces a balanced tree

not necessarily ideal…

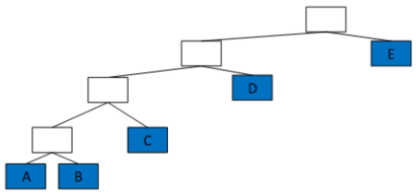


One possible method is to sort on an axis like before and then group neighboring pairs together. We can alternate axes each time we group in an attempt to get a better grouping. This will produce a balanced tree, but once again is balanced good?

The tree that grouped B and C together would produce better results because there would be a lot less wasted space and we could more quickly reject regions (imagine a much large tree).

Bottom-Up

Group nearest neighbors together



Can produce unbalanced trees

Another method of picking pairs is to group the two objects that are closest together to each other. Closest neighbor algorithms can be used to find this efficiently (such as a k-d tree). Unfortunately, picking closest neighbors has no guarantee to produce a good tree. In fact, in some scenarios it can easily produce the worst case scenario.

Bottom-Up

Randomization:

- Pick a random node
- Choose a good partner

To avoid worst-cast:

- Do several pairings at a time

One of the better methods (from computational efficiency) is to use some randomization. Instead of checking all pairs for the best volume or surface area which is $O(n^2)$, we can randomly pick 1 node and then find its best pair which is only $O(n)$. This tends to still produce good results while reducing the complexity.

Unfortunately this still doesn't prevent the worst case scenario from happening (last slide). One method to get around this that can work in almost all of these techniques is to not only group 1 pair at a level. We can pick some small fixed number (like 4) of pairs to form at each level. Whenever we form a pair we don't consider it as a candidate for the remaining pairs of that level. This forces the tree to be a bit more balanced while still allowing a not even split.