4.11

Loose Octrees

Thatcher Ulrich

The octree is a classic and effective data structure for partitioning 3D datasets into hierarchies of bounding volumes. For datasets with a lot of objects, octrees can greatly accelerate frustum culling, ray casting, proximity queries, and just about any other spatial operation.

However, ordinary octrees do have a few disadvantages. In this article, I will focus on one disadvantage in particular, which is that a small object, depending on its location, may be stored in an octree node with a very large bounding volume. This happens when an object straddles the boundary plane between two large nodes. This creates "sticky" areas in the partitioning hierarchy, keeping small objects high in the tree hierarchy and reducing the effectiveness of the partitioning.

There are various methods of adjusting the basic octree data structure and algorithms to mitigate or avoid this problem, and each method has its unique tradeoffs. In this article, I present one such alternative, the "loose octree." Its primary advantage over an ordinary octree is that it avoids stickiness in the object partitioning, resulting in more precise spatial database queries. For certain applications, such as mutual collision detection between numerous moving objects, the efficiency gain can be significant. There is an additional minor side benefit, in that computing a given object's desired node in the tree is a simple *O(1)* operation. A similar trick can be done using ordinary octrees, but it's not as straightforward.

Its main weakness is that it tends to use more partitioning nodes for a given dataset than an ordinary octree. Limiting the depth of the tree can mitigate this, but it's something to be aware of.

Quadtrees

The octree is a 3D data structure. The analogous 2D data structure is the quadtree, which shares the same basic properties. This remains true of loose quadtrees; they are just a 2D version of loose octrees. Loose quadtrees have the same tradeoffs as loose octrees, with respect to their conventional counterparts, so they can be useful in applications that only require hierarchical partitioning in two dimensions.

Since it's much easier to visualize these data structures in 2D, in this article I'm going to use 2D diagrams based on quadtrees. However, the octree principles are exactly the same, and the extension to 3D is straightforward.

Bounding Volumes

In a conventional octree, the basic node bounding volume is a cube. All objects associated with a node must be contained completely within the node's bounding cube. Each node may also have up to eight child nodes, whose bounding cubes are formed by slicing the parent cube into eight equal sub-cubes. The quadtree version is illustrated in Figure 4.11.1.

The bounding volumes of the child nodes nest perfectly within the bounding volume of the parent node, filling the entire space with no overlap. The child nodes can be further subdivided the same way. If you examine the sizes and spacing of the bounding volumes, you can see that they follow a regular pattern. Consider the edge length of the bounding cubes: at the root of the tree, the cube edge length is equal to the world dimensions. At each level deeper into the tree, the cube edge length is half the size of the previous level's cube edge length. Thus, the formula for bounding cube edge length is:

$$L(depth) = W / (2 \wedge depth)$$

where W is the world size, and depth is the number of levels by which a node is separated from the root. The root node has depth 0.

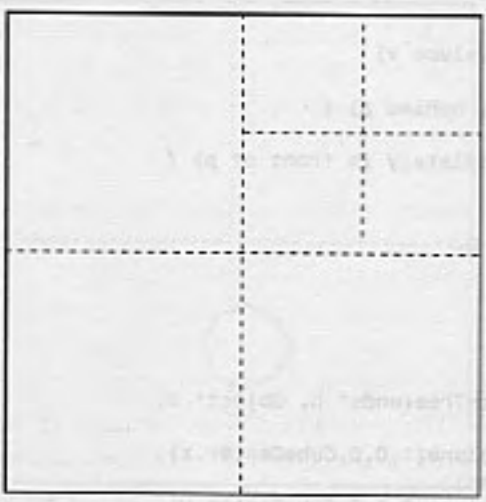

FIGURE 4.11.1 A quadtree node, shown with bounding square in bold, subdivided along dotted lines. Each quadrant becomes a child node. The child nodes can also be subdivided as shown in the upper-right quadrant.

The spacing of the bounding cubes' centers at a given depth follows the same pattern. At the root there's only one node, so node spacing doesn't really have any meaning, but starting at depth 1, the centers of the root's child nodes are spaced *W/2* units apart from their neighboring nodes. Each subsequent level cuts the node spacing in half. The formula for node spacing is:

$$S(depth) = W / (2 \wedge depth)$$

So, for a given depth, the cube edge size and the node spacing are identical. This makes sense because at a given tree level, the bounding cubes are perfectly packed into the world volume with no gaps and no overlap.

Partitioning Objects

Given a set of objects in a virtual world, each object having some finite bounding volume, an octree can be used to partition the objects within the world space, to accelerate various spatial operations such as frustum culling, ray casting, proximity tests, etc. Different criteria can be used for partitioning, but the classic octree partitioning scheme is to associate a given object with the node in the octree whose bounding cube most tightly contains the entire object volume. This node can easily be found by a recursive traversal of the tree. Here is some pseudocode:

```
struct node {
    Vector3 CubeCenter;
    node*   Child[2][2][2];
    ObjectList Objects;
};


int Classify(plane p, volume v)
{
    if (v is completely behind p) {
        return 0;
    } else if (v is completely in front of p) {
        return 1;
    } else {
        // v straddles p.
        return 2;
    }
}


void    InsertObjectIntoTree(node* n, Object* o)
{
    int xc = Classify(plane(1,0,0,CubeCenter.x),
        o.BoundingVolume);
    int yc = Classify(plane(0,1,0,CubeCenter.y),
        o.BoundingVolume);
    int zc = Classify(plane(0,0,1,CubeCenter.z),
        o.BoundingVolume);
```

```
        if (xc == 2 || yc == 2 || zc == 2) {
        {
            // Object straddles one or more of the child
            // partition planes, and so won't fit in any
            // child node, so store it in this node.
            Objects.Insert(o);
        } else {
            // Object fits in one of the child nodes. Recurse to
            // find the correct descendant.
            InsertObjectIntoTree(Child[zc][yc][xc], o);
        }
    }
```

This is a nice, straightforward hierarchical partitioning scheme that handles whatever you throw at it and generally comes up with a decent partitioning. However, it has one disturbing oddity. Notice that if an object straddles any one of a node's partitioning planes, then the object is stored in that node. This happens even if the object is tiny and the node is huge; see Figure 4.11.2 for an example. In practice, if you have lots of small objects, the ones located along the root node partitioning planes can "clog up" the root node by filling it with small, poorly partitioned objects, and reducing the efficiency of spatial operations (e.g., Figure 4.11.3).

I'll call this problem the "sticky planes" problem, the idea being that partitioning planes high in the tree hierarchy attract excess objects to their associated nodes, and are thus "sticky." There are various ways to solve this problem. One method is to split objects on partitioning planes and then classify the pieces individually. Another method is to allow an object to be referenced by more than one node, so an object can

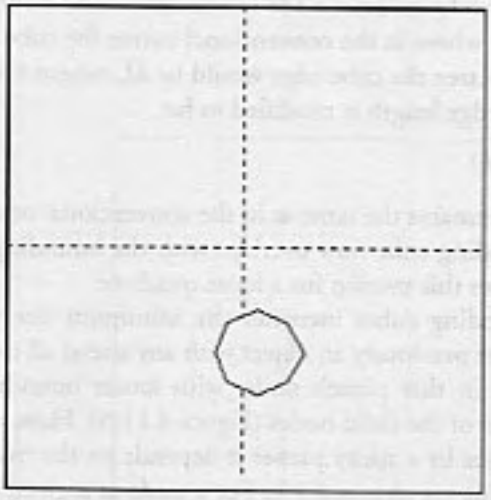

FIGURE 4.11.2 Even though the circle is very small compared to the root node (bold square), it can't be placed in a child node because it straddles one of the (dotted) partitioning lines.

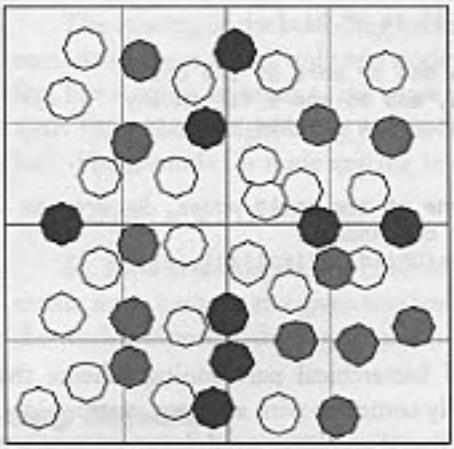

FIGURE 4.11.3 All of the objects are small, but the shaded ones are stuck to higher nodes in the quadtree, due to straddling the partitioning line.

be shared by child nodes on either side of a sticky plane, rather than being stored in the parent node. For static objects, those approaches are effective, but they're not so good for handling dynamic objects.

Making It Loose

The "loose" octree method takes a different tack: it solves the sticky planes problem by adjusting the node bounding volumes. Specifically, by "loosening" the bounding cubes, but leaving the node hierarchy and the node centers as is. The bounding volume of a node is still a cube, but where in the conventional octree the cube edge may have had length L, in the loose octree the cube edge would be kL, where $k > 1$. Thus, the formula for bounding cube edge length is modified to be:

$$L(depth) = k * W / (2 \wedge depth)$$

However, the node spacing remains the same as in the conventional octree. What this means is that a node's bounding cube now overlaps with the bounding cubes of its neighbors. Figure 4.11.4 shows this overlap for a loose quadtree.

This loosening of the bounding cubes increases the minimum size of objects attracted by a sticky plane. Where previously an object with any size at all that crossed a sticky plane would be stored in that plane's node, with looser bounding cubes, smaller objects will fit within one of the child nodes (Figure 4.11.5). How small must an object be to avoid being caught by a sticky plane? It depends on the tree depth of the plane's node, and on the value we choose for k. For a node at a given depth, no object with a bounding radius smaller than $(k - 1) * L / 2$ can be stuck to that node due to straddling a partitioning plane. Instead, since the child nodes' bounding volumes have been enlarged, such objects can fit in one of the child nodes.

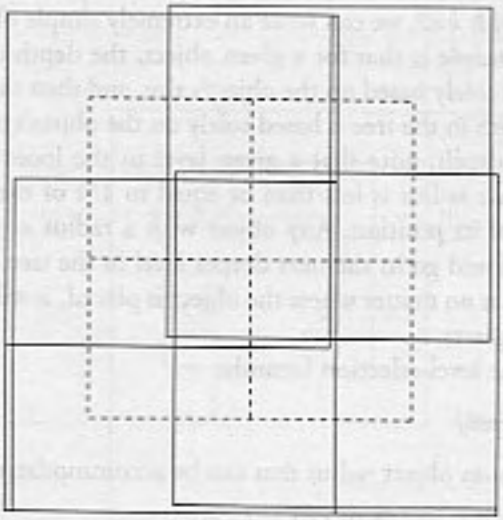

FIGURE 4.11.4 Four nodes. The conventional bounding squares are shown with dashed lines. The same four nodes in a loose quadtree have bounding squares shown in black. The squares have been offset so that they can be distinguished from each other.

So, what's a good value for k? Without fully exploring all the tradeoffs in this article, I propose $k=2$ as a useful all-around value. A tree with k much less than 2 starts to suffer from the sticky planes problem, and a tree with k too much greater than 2 results in excessively loose bounding volumes.

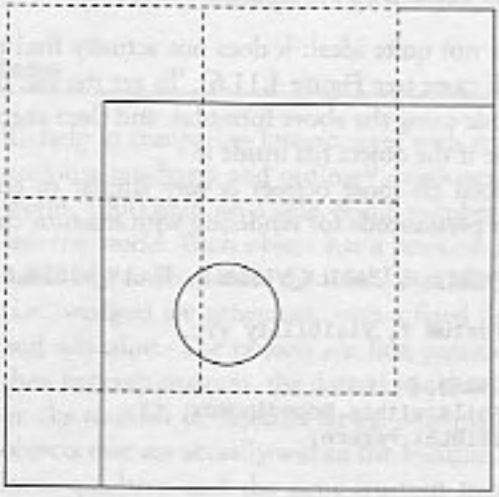

FIGURE 4.11.5 The circle won't fit in any of the conventional child node bounding squares, but it will fit in the loose bounding square of the lower-right child.

Assuming a loose octree with $k=2$, we can write an extremely simple object insertion procedure. The basic principle is that for a given object, the depth of the containing node can be calculated solely based on the object's size, and then the choice of the particular node at that depth in the tree is based solely on the object's center location. To get the formula for depth, note that a given level in the loose octree can accommodate any object whose radius is less than or equal to 1/4 of the bounding cube edge length, regardless of its position. Any object with a radius <= 1/8 of the bounding cube edge length should go in the next deeper level in the tree. For example, in Figure 4.11.5, notice that no matter where the object is placed, it will fit within one of the nodes' bounding squares.

Here's the derivation of the level-selection formula:

$L(depth) = 2 * W / (2 \wedge depth)$

Let $Rmax(depth)$ = maximum object radius that can be accommodated at depth.

$Rmax(depth) = 1/4 * L(depth) = 1/2 * W / (2 \wedge depth)$

Let $depth(R)$ = the first tree depth that can accommodate an object of radius R

$R <= Rmax(depth(R))$
$R <= 1/2 * W / (2 \wedge depth(R))$
$depth(R) >= log2(W / R) - 1$
$depth(R) == floor(log2(W / R))$

Once the depth is known, choosing the particular node at a given depth is simple—just find the closest node to the object's center. Assuming the world is centered at the coordinate system origin, the formula to compute the node indices is:

$index\{x,y,z\} = floor((object.\{x,y,z\} + W/2) / S(depth))$

Note that this procedure is not quite ideal: it does not actually find the tightest possible containing node for all cases (see Figure 4.11.6). To get the last bit of tightness, first find the candidate node using the above formulas, and then check the child node nearest to the object to see if the object fits inside it.

Performing spatial operations on loose octrees is very similar to conventional octrees. For example, this is the pseudocode for rendering with frustum culling:

```
enum Visibility { NOT_VISIBLE, PARTLY_VISIBLE, FULLY_VISIBLE };

void    Node::Render(Frustum f, Visibility v)
{
    if (v != FULLY_VISIBLE) {
        v = ComputeVisibility(this.BoundingBox, f);
        if (v == NOT_VISIBLE) return;
    }

    this.ObjectList.Render(f, v);

    for (children) {
```

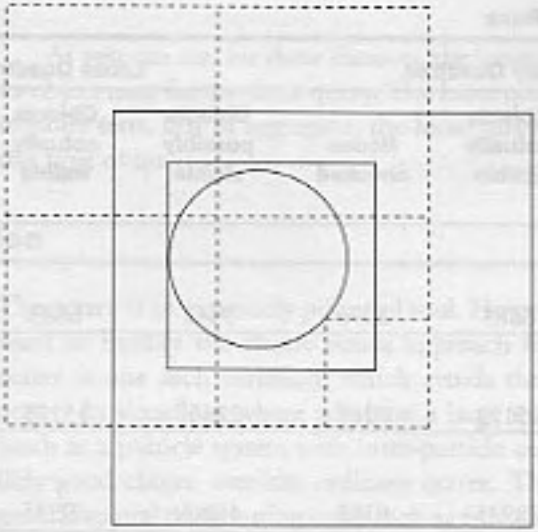

FIGURE 4.11.6 The simple placement formula would put the circle in the node bounded by the large black square, but due to its particular position, the circle has a better fit in the upper-left child node, bounded by the small black square.

```
            child.Render(f, v);
        }
    }
```

The exact same algorithm works with conventional octrees; the only difference is that this `.BoundingBox` would be smaller.

Comparison

To help in comparing loose octrees with regular octrees, I wrote a test program based on loose quadtrees and ordinary quadtrees. The program posits a 2D square virtual world, 1000 units on a side. Some number of circular objects are generated to populate the world. Each object has a position and a bounding radius, which are chosen randomly to fit within the world boundaries. Then a certain number of 2D frusta (i.e., wedges) are generated, with a fixed field-of-view angle, and a random position and direction. The objects are first partitioned using a conventional quadtree, and then for each frustum, the dataset is queried for visible objects. Statistics are gathered on the number of objects that are potentially within the frustum, and the number of objects that are actually within the frustum. Then, the objects are re-classified using a loose quadtree, and the same frustum tests are run and the same statistics are collected.

Results of some sample runs are summarized in Table 4.11.1.

Table 4.11.1 Results of Some Sample Runs

Test Parameters tree max depth = 5 100 frusta FOV = 45° checked	Ordinary Quadtree			Loose Quadtree		
	Objects possibly visible	Objects actually visible	Nodes checked	Objects possibly visible	Objects actually visible	Nodes
500 objects obj min radius=30 obj max radius=30	18859	6883	2976	9442	6883	7024
1000 objects obj min radius=15 obj max radius=15	31133	15173	7265	22457	15173	8815
2000 objects obj min radius=5 obj max radius=100	55451	29935	9102	45209	29935	9075

Note that the frustum queries on the loose quadtree generally return fewer "possibly visible" objects than the same queries on the ordinary quadtree. On the other hand, the loose quadtree queries usually have to check more nodes. So, for frustum culling, the differences between the two are noticeable, but not terribly dramatic.

Things get more interesting when looking at inter-object queries, such as collision detection. In my test program, I added a test in which each object is checked for contact with all the other objects in the dataset, and collected statistics on the checks. The results for the same datasets used previously are listed in Table 4.11.2.

Table 4.11.2 Test Results

Test Parameters tree max depth = 5	Ordinary Quadtree			Loose Quadtree		
	Inter-object contacts	Object-to-object tests	Object-to-Node tests	Inter-object contacts	Object-to-object tests	Object-to-Node tests
500 objects obj min radius=30 obj max radius=30	3034	53469	7351	3034	9125	24839
1000 objects obj min radius=15 obj max radius=15	2730	113989	18040	2730	24609	45658
2000 objects obj min radius=5 obj max radius=100	7094	345377	38107	7094	89276	89312

As you can see, for these datasets, the loose quadtree needs to do far fewer object-to-object tests for the same query. The loose quadtree does require many more object-to-node tests, but in aggregate, the loose quadtree is significantly more efficient for this type of query.

Conclusion

The octree is an extremely powerful tool. However, in certain circumstances, you may want to modify the classic octree approach to better fit your problem. The loose octree is one such variation, which avoids the sticky planes problem of the classic octree. In situations where you have a large number of interacting, dynamic objects (such as a particle system with inter-particle collisions), the loose octree is a particularly good choice over the ordinary octree. The loose octree also performs well for general spatial partitioning tasks such as frustum culling.