Bounding Hierarchies

Overview

Since standard ray tracing is extremely computationally expensive, techniques must be found to increase the computational efficiency of the algorithm. Whitted discovered that 75 percent of the time required for simple scenes was in ray-object intersections. Therefore many algorithms have been researched to speed up basic ray-object intersection calculations.

The most common technique to reduce the ray-object intersection cost is to replace the objects with simple bounding objects that require less time to determine intersection. If the bounding object is not intersected, then the true object cannot have an intersection.

The bounding object technique reduces the cost of the intersections, but it still remains linear in the number of objects. However, if we construct a hierarchy of such objects, we can discard entire subtrees of objects if the subtrees bouunding box is not intersected. With this technique, we can reduce the intersection calculations to logarithmic in the number of objects.

Building the Hierachy

When attempting to construct a bounding hierarchy, one immediately notices that many such trees can be constructed. Additionally, the time to render an image can vary by many orders of magnitude simply depending on the choice of bounding tree. Therefore, it is important for us to determine a bounding hierarchy that will help to reduce the time spent rendering an image.

In order to create a bounding hierarchy that reduces rendering time, we must estimate the cost of adding a new object to the hierarchy. Everytime a bounding object is intersected, we must perform the intersection with all of its children. Therefore, the measure of cost that we will use is the probability that the object will be intersected times the number of children that the object has plus the cost of all its children. This metric will give us an approximate cost of adding this node to the hierarchy.

In order to compute the above formula, we need to estimate the probability of a ray intersecting a given bounding object. This is proportional to the surface area of the bounding object divided by the surface area of the object that bounds

the entire scene. However, since all of the costs will contain this denominator (surface area of root node), it can be removed from the cost calculation. The cost functional then becomes:

$$Cost(nodeN) = area(N) * N.numChildren + \Sigma\,(N.child(i).cost)$$

We also define the cost of a leaf node in the hierarchy to be zero since it has no children.

Now that we have an absolute cost measure, we can define an incremental cost due to inserting a node m at the node n:

```
If(leaf(n))
        Incr = 2 * area( bound( n , m ) )
else
        Incr = -area(n) * n.numChildren + area(bound(n,m)) * (n.numChildren + 1)
```

After defining our incremental cost measure, we are ready to begin constructing our bounding hierarchy. Given an existing hierarchy, we wish to find the spot in the hierarchy that produces the smallest increase in cost for the tree and add the node there. With this algorithm, we can incrementally build our bounding hierarchy. The pseudocode for adding a new node m to an existing subtree n is given below:

```
insert(n,best,m)
    oldArea = area(n)
    if(!leaf(n))
        for all children calculate incr(n.child(i),m)
        nc = child node with min incr
        best = insert(nc, nc.incr < best.incr ? nc : best, m)
    if(n == best) addChild(n,m)
    else if(oldArea < area(n))
        n.cost += (area(n) - oldArea) * n.numChildren
    return best
```

```
addChild(n,m)
    if(leaf(n))
        n.child[0] = new node(n)
        n.child[1] = m
        m.cost = 0
    else
        n.child[n.numChildren] = m
        n.numChildren++
    n.cost += n.incr
```

Traversing the Hierarchy

In order to use the bounding hierarchhy, one simply intersects rays with a volume (starting with the entire scene volume) and if an intersection exists, performs the intersection test on each of the children. There now comes the question of what order to process the child objects. One can easily process the objects in the order that they were added to the tree, but this would not take into account any information about the scene. Instead, we can store the bounding objects that we need in a heap ordered by the distance along the ray. Now, we can stop processing objects once the closest object in the heap is further than the current intersection point. The psuedo code is given below:

```
traverse
    initialize closest intersection point to infinity
    initialize closest intersection object to none
    initialize active node heap
    while(heap not empty)
        extract nearest node from heap
        if(closest intersection point is closer than node)
            break
        if(node is a primitive node)
            compute intersection with primitive
            if(intersects && is closer)
                update closest intersection point
                update closest intersection object
        else
            for(each child of node)
                intersect ray with bounding volume
                if(ray intersects bounding volume && is closer)
                    insert node in ative node heap
```