VI.1 RAY TRACING WITH THE BSP TREE

Kelvin Sung
University of Illinois
Urbana, Illinois

and

Peter Shirley
Indiana University
Bloomington, Indiana

Introduction

In order to speed up the intersection calculation in ray tracing programs, people have implemented divide-and-conquer strategies such as hierarchical bounding volumes and octrees. Uniform subdivision (essentially a three-dimensional binsort) has also been used to speed up this calculation.

Uniform subdivision is undesirable for applications where the objects may be unevenly distributed in space. This is because the amount of memory needed for uniform subdivision is proportional to the highest density of objects, rather than the total number. Hierarchical bounding volumes can be difficult to implement effectively, but can be used to good effect (Kay and Kajiya, 1986). Hierarchical space subdivision techniques do not suffer the memory problems of uniform subdivision and are also relatively easy to implement. In this Gem, we discuss what we think is the best overall hierarchical subdivision technique currently known.

It is often believed that adaptive spatial subdivision approaches to accelerating the tracing of a ray are messy and hard to implement. In our experience with different spatial structures and traversal algorithms, we have found this view to be untrue. It is straightforward to implement the Linear Time Tree Walking algorithm, as proposed by Arvo (1988), on a Binary Space Partitioning (BSP) tree. The resulting system outperforms all of the spatial subdivision approaches we have experienced.

We have implemented and compared the performance of several traversal algorithms on an octree (Glassner, 1984; Arvo, 1988; Sung, 1991) and on a BSP tree (Kaplan, 1985; Arvo, 1988). In order to obtain meaningful

ISBN 0-12-409670-0 (IBM)
ISBN 0-12-409671-9 (Mac)

comparisons, we have kept the rest of our ray tracing system unchanged while replacing the spatial structure building and traversal components for each of these methods. Our experience shows that the Linear Time Tree Walking method (Arvo, 1988) is consistently at least 10% faster than the rest and is usually better than that. We have observed that implementing the tree walking algorithm on a BSP tree and on an octree results in similar performance, but that the implementation is more straightforward on a BSP tree. Finally, it should be pointed out that the *recursive traversal* algorithm introduced independently by Jansen (1986) is a different implementation of the same tree walking idea.

There are two basic modules in a BSP tree intersection code. The first module builds the tree by recursively cutting the bounding box of all objects along a median spatial plane. The second module tracks a ray through the leaf nodes of the BSP tree checking for intersections.

The BSP tree is built by *InitBinTree()* and *Subdivide()*, *RayTreeIntersect()* is the actual tree walking traversal of the BSP tree following a ray. These are the key functions necessary to implement the algorithm. Subdivide( ) builds the BSP tree by subdividing along the midpoint of the current node's bounding volume.

```
Subdivide (Current Node, CurrentTree Depth, CurrentSubdividingAxis)
    if ((CurrentNode contains too many primitives) and (CurrentTreeDepth
    is not too deep)) then
      begin
        Children of CurrentNode ← CurrentNode's Bounding Volume
        /*Note that child[0].max.DividingAxis and
        Child[1].min.DividingAxis are always equal.*/
        /*Depending on CurrentSubdividingAxis, DividingAxis can be
        either x, y, or z.*/
        if (CurrentSubdividingAxis is X-Axis) then begin
          child[0].max.x ← child[1].min.x ← mid-point of CurrentNode's
          X-Bound
          NextSubdivideAxis ← Y-Axis
        end else if (CurrentSubdividingAxis is Y-Axis) then begin
              child[0].max.y ← child[1].min.y ← mid-point of
              CurrentNode's Y-Bound
              NextSubdivideAxis ← Z-Axis
          end else begin
                child[0].max.z ← child[1].min.z ← mid-point of
                CurrentNode's Z-Bound
                NextSubdivideAxis ← X-Axis
```

```
                end
            for (each of the primitives in CurrentNode's object link list) do
               if (the primitive is within children's bounding volume) then
                  add the primitive to the children's object link list
            Subdivide (child[0], CurrentTreeDepth + 1, NextSubdivideAxis)
            Subdivide (child[1], CurrentTreeDepth + 1, NextSubdivideAxis)
          end
```

As suggested by Arvo (1988), RayTreeIntersect( ) avoids recursive procedure calls in the inner loop of tree walking by maintaining an explicit stack. The pseudo-code given here is based on Arvo's article in the Ray Tracing News, where recursion is used for ease of understanding. When calling RayTreeIntersect() the first time, initial values of *min* and *max* should be the distances (measured from the ray origin along the ray direction) to the two intersecting points between the ray and the bounding volume of the root of the BSP tree. Notice that if a ray originates from inside the BSP tree, then the initial value of *min* will be negative.

```
RayTreeIntersect (Ray, Node, min, max)
    if (Node is NIL) then return ["no intersect"]
    if (Node is a leaf) then begin
        intersect Ray with each primitive in the object link list
            discarding those farther away than "max"
        return ["object with closest intersection point"]
    end
    dist ← signed distance along Ray to the cutting plane of the Node
    near ← child of Node for half-space containing the origin of Ray
    far ← the "other" child of Node - i.e. not equal to near
    if ((dist > max) or (dist < 0)) then /*Whole interval is on near side*/
       return [RayTreeIntersect (Ray, near, min, max)]
    else if (dist < min) then            /*Whole interval is on far side*/
        return [RayTreeIntersect (Ray, far, min, max)]
    else begin                           /*the interval intersects the plane*/
        hit_data ← RayTreeIntersect (Ray, near, min, dist)   /*test near side*/
        if hit_data indicates that there was a hit then return [hit_data]
        return [RayTreeIntersect (Ray, near, dist, max)]     /*test far side*/
    end
```

There are a few standard link list functions that are not listed in the C code: *FirstOfLinkList()*, *NextOfLinkList()*, *AddToLinkList()*, and *DuplicateLinkList()*.

One possible improvement over the basic algorithm is to keep track of the nearest intersection and its distance, regardless of whether it is inside of the current node. With this information, as soon as a node is found that starts beyond this nearest intersection distance, we know we are done.

If an object spans multiple tree nodes (spatial cells), then the intersection calculation between this object and a given ray may need to be carried out multiple times, once in each node traversed by the ray. The mailbox idea was proposed (Amanatides and Woo, 1987; Arnaldi *et al*, 1987) to avoid this problem. However, it is observed that this technique is effective only when primitives in the scene are large compared to the spatial cells. When the size of the primitives in a scene is small compared to the size of the spatial cells, the mailbox implementation may slow down the rendering process (Sung, 1991). Also, the mailbox implementation requires the scene database to be updated after each intersection calculation. This implies that to parallelize the algorithm, some kind of database coherence policy must be administrated; this would increase the complexity of an implementation. Based on these observations, we have chosen not to include the mailbox enhancement in our code.

Other work on BSP trees includes Fussell and Subramanian (1988), MacDonald and Booth (1989), and Subramanian and Fussell (1991).

Acknowledgments

We would like to thank Jim Arvo, who encouraged us to write up our experience as a Gem; Frederik Jansen for discussing his experience with BSP Trees; and K. R. Subramanian for sharing the details of his work.

See also G3, E.2.