Lecture 8

Last lecture we looked at several methods for solving the hidden surface removal problem. Many such methods boil down to the problem of finding, for each pixel, what is the closest surface in the view volume that is 'visible' at that pixel. Mathematically this amounts to defining a ray from the viewer through that pixel out into the scene and finding the first intersection.

The ray casting approach I presented last class was very inefficient since, for each pixel, it blinding tested every polygon in the scene. We would like to solve this problem more efficiently. Today we'll look at three approaches which use clever *data structures.* These methods were developed in the early 1980s.

We'll begin with two methods that use spatial partitions. The first method is a generalization of the painter's algorithm from last class.

Binary Space Partition (BSP) tree

The painters algorithm sorted polygon surfaces by their furthest vertices, and had to deal with several special cases which arose when the z extents of the surfaces overlapped.

One limitation of the painter's algorithm is that if the camera is moving, then we will need to recompute the depth ordering for each camera position and orientation. To see this, consider an extreme case where the camera moves in a circle, and the view direction always points to the center of the circle!

An alternative way to solve the painter's problem is to organize the spatial relationships between the polygons in a way that it the ordering of the polygons from back to front can be determined quickly for any viewing position and direction. The basic idea is this[1]: consider an arbitary plane in the scene, if the camera is on one side of this plane, then any polygons that are on the same side of that plane as the camera cannot be occluded (blocked) by polygons that are on the opposite side of the plane. Thus, if we draw polygons on the far side of the plane first, then these far polygons will never occlude polygons that are on the near side of the plane. That is, the near polygons are drawn after the far ones, just like in the painter algorithm.

The method is reminiscent of binary search trees that you saw in COMP 250 and 251. Recall how you build a binary search tree. You take some element from your collection, and then partition the set of remaining elements into two sets. The selected element is placed at the root of the tree. The elements less than the selected one would go in the left subtree (recursively contructed) and the greater than elements go in the right subtree (recursively constructed).

There are many ways to use a binary search tree. You can search for a particular element. Or you can print out all the elements in order. Or you can print them out in backwards order. (I'm sure you know how to do this so I won't review it.) The analogy to the BSP data structure that I'll describe next is that you want to examine the elements in reverse order. The order in our case is the distance to the surface.

The binary space partition tree which represents a partition of 3D space. There are two aspects here. One is how to construct the tree. The other is how to use it to solve the visiblity problem.

[1] [Fuchs et al, 1980]

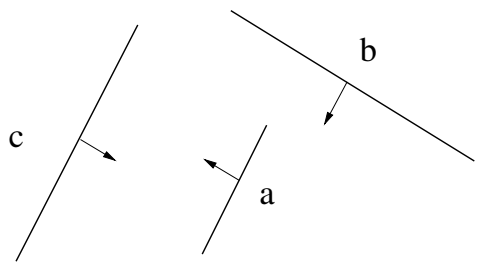
Constructing the BSP tree

Suppose we have a list of polygons. Each polygon is assumed to have a surface normal, so the normal points out of the surface and defines the front side of the polygon. We construct a *binary space partition tree* (BSP tree) as follows. Note that this is a pre-processing step. There is no viewer defined.

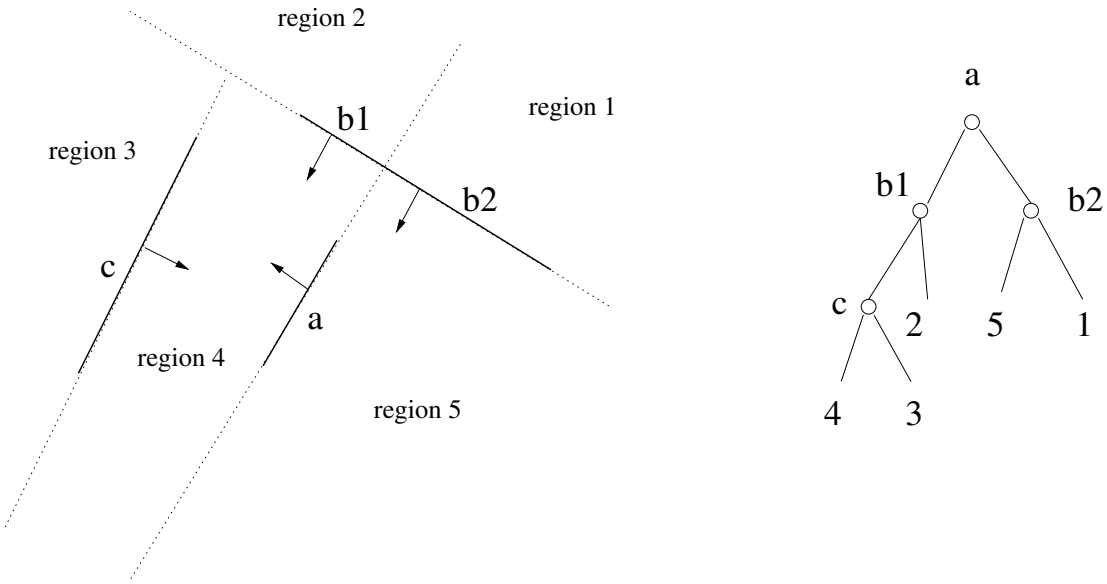
```
makeBSPtree(list of polygons){
   if list is empty
      return(NULL)
   else {
      select and remove a polygon P from list
      backlist := NULL;
      frontlist := NULL;
      for each polygon Q in list
         if all vertices of Q are in front of plane of P
            add Q to frontlist
         else if all vertices of Q are behind plane of P
            add Q to backlist
         else  //  plane P splits Q
            split Q into two polygons and add them to
            frontlist and backlist, respectively
       return   combine( makeBSPtree(frontlist),  P,
                         makeBSPtree(backlist) )
   }
}
```

The relations "front" and "back" refer to the front and back face of polygon P, as defined last class. They do not depend on the viewer's position, but rather only depend on the ordering of the vertices. *We use the convention that the polygons Q in front of plane P belong to the left child and the polygons behind plane P belong to the right child.*

Here is an example. What is the 2D binary space partition tree (BSP tree) defined by the following set of line segments ? Let's suppose that when constructing the BSP tree, we choose edges in alphabetical order (a,b,c).



Here is the solution.



Hidden surface removal using the BSP tree

How do you use a BSP tree to solve the visible surface problem ? As I mentioned earlier, the idea is similar to the painter's method: draw from back to front, relative to the viewpoint. We start at the root of the BSP tree and draw the polygons that lie on the far side of the root polygon's plane, then draw the root, then draw the polygons that lie on the near side of the root polygon's plane. Don't confuse "front" and "back" face with near and far sides of the polygon. Only the latter depend on the viewer position.

```
displayBSPtree( tree, viewer){
   if (tree != NULL)
      if (viewer is on the front side of plane defined by root){
         displayBSPtree(backchild, viewer)
         displayPolygon(root)
         displayBSPtree(frontchild, viewer)
      }
      else{  //  viewer is behind the root note
         displayBSPtree(frontchild,viewer)
         displayPolygon(root)  //  back faced culled,
                                   so not necesssary
         displayBSPtree(backchild,viewer)
      }
}
```

In the above example, suppose the camera was in region 5. We would first examine the root node `a` and see that the viewer is on the back side of polygon `a`, so it needs to draw all polygons on the opposite side (the front side, which is the left subtree. Next, examine the root of that left subtree `b1` and note that the view lies in on the front side of `b1` and hence needs to draw all polygons behind `b1` first, i.e. the right subtree, which is empty in this case. Then draw `b1`, and then draw

the left subtree of `b1`. This brings us to node `c`. The viewer is on the front side of `c`. We thus need to draw its right subtree (region 3 – empty), draw `c`., and then draw `c`'s left subtree (region 4 – empty).

Now we traverse back up the tree, draw `a`, and then examine the right subtree of `a`. We examine `b2`, draw its right subtree (region 1 – empty), then draw `b2`, then draw its left subtree (region 5 – empty). Thus, the order in which we draw the polygons is `b1, c, a, b2`. Similarly, the order in which we traverse the regions is 2,3,4,1,5.

Observations:

- In COMP 250, you learned about in-order, pre-order, and post-order tree traversals. But here the traversal need not be any one of these. The traversal order depends on where (which region) the viewer is.
- We hope depth of the tree grows roughly like $\log n$, where n is the number of polygons. There is no guarentee on this, though.
- You are free to cull back faces if you want to, or not.
- The great advantage of the BSP method over the depth buffer method comes if you will want to make lots of pictures of a scene from many different viewpoints, i.e. move the camera through the scene (translate and rotate). You can use the same BSP tree over and over again, as long as the polygons in the scene aren't moving from frame to frame. If there is motion in the scene, then this is not a good method to use, however. You would be better off using the depth buffer method (like OpenGL).
- We did not discuss the special case that the viewer lies exactly on the plane defined by a node. This special case occurs, then we could move the viewer by a tiny amount off the polygon.

Uniform spatial partitions

The next method also partitions the scene into regions, but it uses a regular grid which is independent of the particular scene. Consider a cube that bounds the scene. Suppose we partition this bounding cube into disjoint volumes, namely each of the x,y,z axis directions is partitioned into N intervals which would define a cubic lattice of N^3 cells. Here we'll consider casting just just one ray through this bounding volume. We allow its direction to be arbitrary.

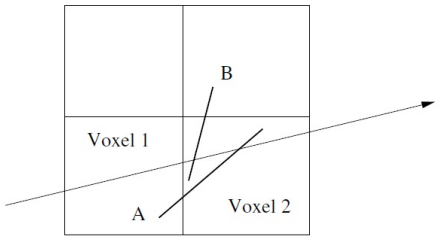
The voxels partition space into disjoint sets of points [2] But the voxels need not partition the set of surfaces into disjoint sets, since a surface might belong to more than one voxel. Here is an algorithm for ray casting using this simple "3D uniform voxel grid" data structure. First we initialize our t and closest polygon `p`.

```
t =  infinity
p = NULL
```

[2]Here we ignore the fact that voxels can share faces, so the word "partition" is being used a bit loosely here.)

```
current_voxel := voxel containing the starting point of ray
while (t == infinity){
    for each surface in current voxel{
        t_intersect = distance to surface along ray    // infinite if no intersection
        if (t_intersect < t) and (intersection point belongs to current voxel)
            t = t_intersect
            p = surface
    }
    current voxel = next voxel hit by the ray
}
```

One subtlety in this algorithm is the test condition that the intersection point belongs to current voxel. Without this condition, the algorithm would be incorrect. Consider the example shown below. *(In fact, it is basically the same example shown twice. I need to fix that figure.)* Suppose we are currently examining voxel 1. Surface A belongs to voxels 1 and 2. The ray intersects surface A in voxel 2. This intersection would be discovered when the current voxel was 1. At that time, t would be set to a finite value. The `while` loop would thus terminate after examining surfaces that belong to voxel 1. However, surface A would in fact *not* be visible along the cast ray, since there is a surface B which is closer. But surface B would only be examined when the current voxel is 2. The algorithm thus needs to ignore the intersection with surface A at the time it is in voxel 1 (since the intersection itself does not occur in voxel 1).



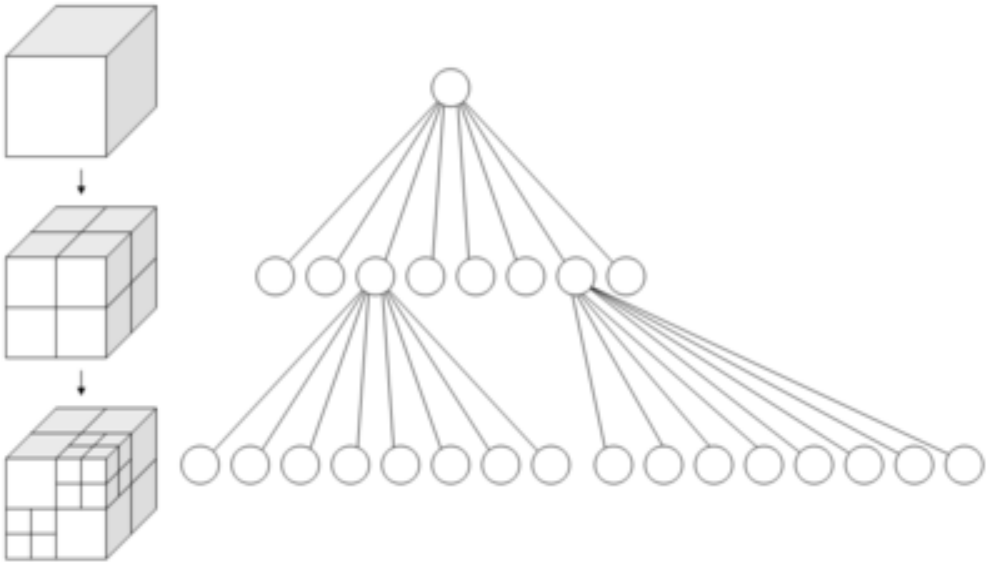
How many voxels should one use? If we use N^3 voxels, how big should N be? If N is very large, then the memory requirements will be large. There will there will be relatively few surfaces intersecting each voxel, and each surface will belong to a large number of voxels. There will be $O(N)$ voxels along each ray which would result in many checks of empty voxels. This would be inefficient. On the other hand, if we use a relatively small N, then there would tend to be some voxels that have many surfaces and so this would lead to testing for intersections of many surfaces and performance would end up being similar to the original ray casting approach. The way to improve the situation is to use non-uniform spatial partitions. Let's turn to a classic one next.

Octrees

Start with a cube. Define an *octree* data structure for partitioning this cube. The cube is partitioned into eight subcubes (voxels) of equal volume, by dividing each of the xyz dimensions in two equal

parts. Each of these subcubes can itself be partitioned into eight equal subcubes, etc. However, we only partition a cube if at least one surface intersects the cube.

Each non-leaf node of an octree has eight children, corresponding to these eight subcubes. A node at depth d is of width 2^{-d} times some constant which is the width of the original cube. See Figure below, which was taken from wikipedia.



At each *leaf* of the octree, store a list of surfaces that intersect the subcube (voxel) defined by that leaf. Surfaces are referenced only at the leaves.

There is some freedom in how one defines the octree, namely how one decides when to stop partitioning a voxel. An octree can be defined to some pre-chosen maximal tree depth, or up to a maximal number of surfaces per leaf, or some other criterion.

How are octrees used in ray casting? The basic idea is to cast the ray and examine the voxels/cells through which the ray passes. As in uniform ray casting, we examine the voxels in front-to-back order. The algorithm for ray casting using an octree is the same as the one for the uniform spatial subdivision above. The only difference is the implementation of the line

```
current voxel := next voxel hit by the ray
```

This requires that we traverse the tree in a particular manner. (See Exercises.)

As in all ray casting methods, the octree method searches from near voxels to far voxel. It is particularly efficient because the search for the closest object tends to terminate quickly, since we have sorted the objects into groups along the ray. There are extra cost one must pay though: the octree needs to be constructed in advance, and the algorithm for traversing the tree is more complex than the algorithm for the uniform space partition.

Octrees are *not* used by OpenGL. They are used (heavily!) by a system called RADIANCE `http://radsite.lbl.gov/radiance/`.

Hierarchical Bounding Volumes

Let's assume that we've clipped the scene already so that all the objects are restricted to lie in the view volume, which obviously does not contain the viewer itself.

Suppose you have a surface with a complex shape that is described by many polygons or other surfaces. For example, a chair might consist of hundreds of polygons that precisely describe its shape. It would be expensive to check if the ray intersects each of these polygons. One quick method to examine whether a ray intersects such a surface is to consider a much simpler volumetric shape (a cube, a sphere, a cylinder) whose volume strictly contains (bounds) the surface. If the ray intersects the surface then it must intersect the bounding volume. We use the contrapositive statement. If the ray *doesn't* intersect the bounding volume, then it *doesn't* intersect the surface. So, to check if a ray intersects the surface, we first *quickly* check if the ray intersects the bounding volume. If it doesn't, then we don't have to check any of the polygons that belongs to the surface. But if the ray does intersect the bounding surface, then it may or may not intersect the surface. We don't know, and we need to check further.

It is common to take this a step further and consider a *hierarchy* of bounding volumes. Consider a *tree* whose internal nodes correspond to bounding volumes, such that the bounding volume of a (non-root) internal node is contained in the bounding volume of its parent node. The bounding volumes could be defined any way you like (e.g. by rectanguloids, spheres, cylinders, or whatever you want). The key is that it is easy to compute the intersection between a ray and a bounding volume.

At each leaf of the tree, rather than having a bounding volume, we have a list of objects/surfaces which are entirely contained in the bounding volume defined by their parent node. It is also common to define the tree so that each leaf consists of just one surface.

Although the bounding volume of a child is contained in the bounding volume of a parent, there is no containment relation for siblings. For example, we might have an internal node whose bounding volume contains a person that is sitting in a chair. The node might have two children, namely a bounding volume for the person and a bounding volume for the chair. These two bounding volumes would overlap, in the case that the person is sitting in the chair.

How do we use this bounding volume hierarchy for efficient ray casting? Let's keep the discussion as general as we can, and consider casting a ray out from some point in a scene. We wish to find the closest object in the scene that is intersected by the ray. To do so, we do a depth first search of the bounding volume tree and try to find the closest surface. We check the bounding volume of the root node first and, if the ray intersects this volume, then we check each of the children, and proceed recursively. The key idea is that if a ray *does not* interest some bounding volume (an internal node), then we *do not* need to check the children of this node.

For some images illustrating bounding volumes (and octrees), see the illustrations in my lectures slides. These illustrations are taken from:
`http://www.cs.princeton.edu/courses/archive/spring14/cos426/lectures/12-ray.pdf`

Below is an algorithm for the above "hierarchical bounding volume" tree traversal.[3] First initialize global variables.

```
p =   NULL     //  Pointer to closest surface
t = infinity   //  Distance to closest surface found so far
```

Then define a recursive procedure that modifies these global variables. We call it with `traverseHBV(root,t)`.

```
traverseHBV( ray, node){
   if (node is a leaf)
      for each surface in leaf
           cast ray i.e. compute t_intersect
            //  infinity if ray doesn't intersect surface
           if (t_intersect < t)
               t := t_intersect
               p := surface  // point to surface
   else // node is bounding volume
       for each child of node
           traverseHBV(ray, child)
}
```

The above algorithm does a depth first search. For each node, it "expands" the node whenever there is a potential that this node contains a closer surface than the current closest surface. One inefficiency of this algorithm is that it could spend its time searching through a complex bounding volume subtree when in fact none of the surfaces in that bounding volume are visible. You would somehow like to design the algorithm to try to avoid this possibility. The basic idea is to be more clever in how you choose which child to recursively call next. See the Exercises.

[3] Rubin and Whitted, SIGGRAPH 1980)