Questions

1. In the BSP example in the lecture, we considered the order of the regions (leaves) visited if the camera was in region 5. What is the order of the regions visited (and polygons drawn) if the camera is in regions 1, 2, 3, 4 ?

   Note that because we only specify the region of the camera and not the exact position, it might something not be possible to say from the information in the question what order

2. In the BSP draw algorithm, we didn't specify where the camera is looking. Why not? Shouldn't we *need* to know this information?

3. The octree algorithm needs to decide what is the next voxel to visit along a ray. Sketch out how that might be done.

   Hint: you may assume that the algorithm knows the height of the octree i.e. the smallest voxel in the octree. Now break the problem into two parts. The first part is to find a point that lies in the next voxel. The second part is to find the voxel containing that point.

4. Why is it desirable for bounding volumes to have the following properties? Note they are not mutually exclusive.

   - tightly enclose its surfaces,
   - have a simple shape,
   - sibling bounding volumes to have little overlap,
   - have as few nodes as possible

5. To construct an octree to be used for surface visibility, it is necessary to construct the list of polygons that intersect a voxel. How would you do this (for a given cell)?

6. One problem with the hierarchical bounding volume algorithm discussed in class is that there is no attempt to visit the children of a node (volume) in an optimal order. This is problem because you easily expand a subtree that turns out to be entirely hidden.

   An alternative approach is to traverse the tree by only expanding a node (bounding volumes) when the intersection with this node is closer than the intersection of other "candidate" nodes. This can be done by maintaining a separate data structure of nodes – a priority queue of nodes – whose keys are the distances to the node. How would this work? Write out such an algorithm.

Answers

1. In this solution, I'll ignore backface culling.

   (camera in region 1) 3, c, 4, b1, 2, a, 5, b2, 1

   (camera in region 2) 5, b2, 1, a, ... and then we cannot say what is next since it depends on whether the camera is on the front or back side of c's plane. That is a decision that is made at the time of rendering.

   Tou may ask: why not expand the BSP tree so region 2 gets split by c's plane? The answer is that this would buy you nothing. You need to do is decide whether the camera is on the front or back of c's plane at the time the algorithm is running. (If you think it buys you something to split region 2 and give it children in the tree, then let me know why!)

   (camera in region 3) 1, b2, 5, a, 2, b1, 4, c, 3

   (camera in region 4) 1, b2, 5, a, 2, b1, 3, c, 4

2. No, we don't need to know where the camera is looking. The BSP draw algorithm's job is to choose the order in which to send polygons into the pipeline. It doesn't need also to do clipping. Rather, clipping can be done later. For example, think of the standard pipeline. Each of the polygons that is "drawn" by the BSP algorithm could be sent into the pipeline where its coordinates get transformed and then it gets clipped. The direction of camera lookat then would be handled as part of the pipeline.

3. You know the current voxel so you can assume you know the intersection point of the current voxel and the ray, at the point where the ray is exiting the voxel. Call that point $\mathbf{x}$. From the hint, you know the minimum size of the next voxel. So you can find a point $\mathbf{x}' = \mathbf{x} + \delta\mathbf{x}$ that must lie in the next voxel. The calculation of this point will depend on the direction of the ray and I omit the details, but you can hopefully agree this is something that you could compute and be sure about.

   So now we have a point $\mathbf{x}'$ and we want to find which voxel it lies in. For this, we can do a (octary) search in the octree.

4. - It should tightly enclose its surfaces (so we have few "false positives", that is, intersections with the bounding volume which turn out NOT to intersect the surface)
   - The BV should be simple so ray intersections are easy to compute.
   - Two BVs should have little overlap since we check ray intersection for each child and if there is alot of overlap then the checks may often be redundant.
   - A BV should have as few nodes as possible since it takes time to build up, especially when objects are moving. And we want to traverse it as quickly as possible.

5. Suppose you have a list of polygons that intersect the parent voxel (or just suppose you take all the polygons in the scene). For each polygon in your list, you need to decide if it intersects the given voxel. This problem is formally identical to (polygon) clipping to a view volume. So any solution to clipping works here.

6. Here's one way to do it, described in Kay and Kajiya's 1986 paper. Instead of using recursion, use a `while` loop. Maintain a set of "candidate" nodes to expand next. Each candidate node will have a distance associated with it, namely the distance to the bounding volume. This set of candidates and their distances is organized in a min-heap data structure (more generally, a priority queue). The distance value is the key of the heap.

   The algorithm uses the same bounding volume hierarchy discussed in class. What's different is that you don't think of traversing it in the usual sense of examining only one node at a time as in pre/post/in/breadthfirst, ... traversal. Rather, you maintain a separate data structure that keeps track or points to many nodes (bounding volumes) that are being considered at any one time.

```
p = NULL
t = infinity

if ray hits root's BV{
    insert root into heap
}
while (heap is not empty and distance to heap.peek < t) {
     curNode = remove min from heap
     if curNode is a leaf{
         intersect ray with leaf's surface
         if (t_intersect < t)
             t = t_intersect
             p = surface
     }
     else
         for each child of curNode{
             intersect ray with child's BV
             if t_intersect < t
                 insert child into heap
         }
}
```