kD Trees

by Ivan Pocina

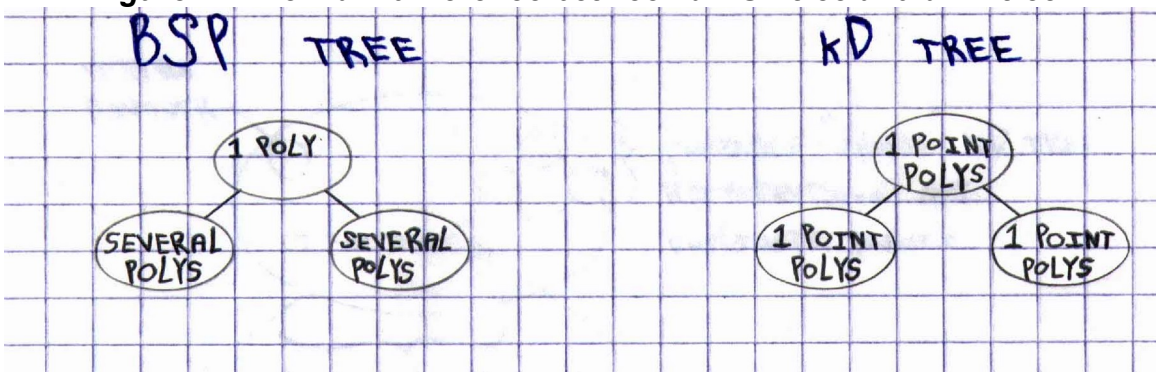
Introduction

kD trees is another class of divide-and-conquer algorithms that tries to solve the VSD (Visible Surface Determination) problem. Invented in the 1970's, this algorithm is used to represent 2D or 3D geometry. That's what the "k" in *kD tree* represents--the dimension #. For example, k = 2 means that we are creating a 2D tree, k = 3 means that we are creating a 3D tree, and so on. I don't know of anyone who is creating a kD tree where k = 4. Something to think about when going into 4D hyperspace!

kD Tree Storage Setup

A kD tree is like a BSP tree in that it also uses a binary tree data structure to represent the actual algorithm. For those of you who need a refresher, BSP (Binary Space Partitioning) trees generally store *only* polygonal data in each of its nodes. 3D points are *not* stored. The internal nodes of a BSP tree store *exactly* one polygon and its leaf nodes may store several polygons at a time. Unlike BSP trees, kD trees have a different storage setup. Each node of a kD tree--this includes both internal and leaf nodes--contains *exactly one* 3D point *and* all of the convex polygons *common* to this 3D point. Anyway, I'll expand more on this later. But, for now, the only thing to remember is that kD trees store *both* 3D points *and* polygons. BSP trees *only* store the latter. See figure 1 for a storage comparison between both trees.

Figure 1. The main difference between a BSP tree and a kD tree.



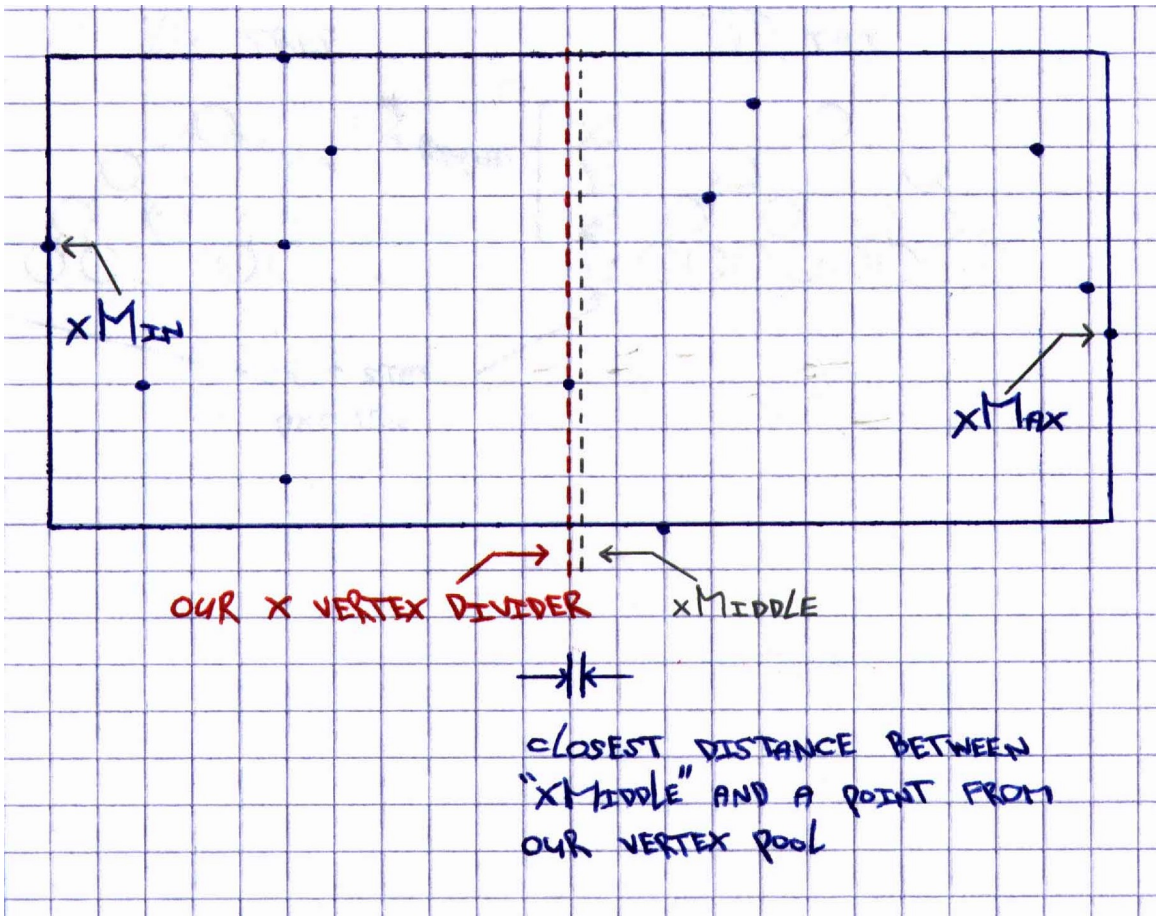
Building a kD Tree

Building a kD tree is similar to building a BSP tree. Since these algorithms are both represented by a binary tree data structure, the functions to build these

trees tend to work best if they are written recursively. When I first wrote my first function to build a kD tree with an *iterative* implemention, I ended up with over 2000 lines of code! Eventually, I learned from my mistake and switched to a *recursive* implementation and got the function down to a mere 473 lines of code! Thank goodness for recursion. In addition, like BSP trees, kD trees should only be built *once* because kD trees experience the same problem that BSP trees experience: a dynamically changing 3D world. This means that a 3D world built around a kD tree algorithm cannot have any of its polygons moving. This includes both rotations and translations of any kind. Polygons cannot even translate within their own infinite plane--well, almost. If we are building a kD tree, where k = 2, then doors can be implemented *exactly* like in DOOM. That is, doors *can* translate within their own plane. But, in true 3D--with a kD tree, where k = 3--this is a problem. The solution? Make the doors into 3D objects that are *separate* from the rest of the 3D world mesh. Now, you can make any 3D door objects into any shape and size, and they can do any movements--rotations and/or translations, etc.--*as long as the 3D door objects--or any 3D objects in general--are NOT part of the continuous 3D world mesh!!!* What this last statement means is that when building a 3D engine that uses the kD tree approach, always make sure that the 3D world and its 3D objects are treated as separate and distinct entities. Like in the Quake games, for example, a 3D world is treated much differently (rendering-wise) from the actual 3D objects that inhabit it--monsters and the like: The 3D world is converted into spans and all 3D objects are simply z-buffered.

Here, I am going to illustrate how you can build a kD tree, where k = 2. Extending this to 3D merely means adding similar code for the third dimension. In this discussion, I am going to use only x and z coordinates. Y coordinates are not relevant here, just as they are not relevant in a 2D BSP tree. Anyway, here it goes! Given a collection of 3D points, from a 3D game level, for example, find a 3D point with the smallest x coordinate--call it ***xMin***. Then find a 3D point with the largest x coordinate--call it ***xMax***. From this info, we can easily find the *true* or *absolute* middle x coordinate: ***xMiddle= (xMax + xMin) / 2.0***. From this piece of info, we can easily find--from the orginal vertex pool--which 3D point's x coordinate is the closest x coordinate to ***xMiddle***. See figure 2 for details. The 3D point found is flagged to indicate that this 3D point has been

Figure 2. How to find the very first vertex divider.

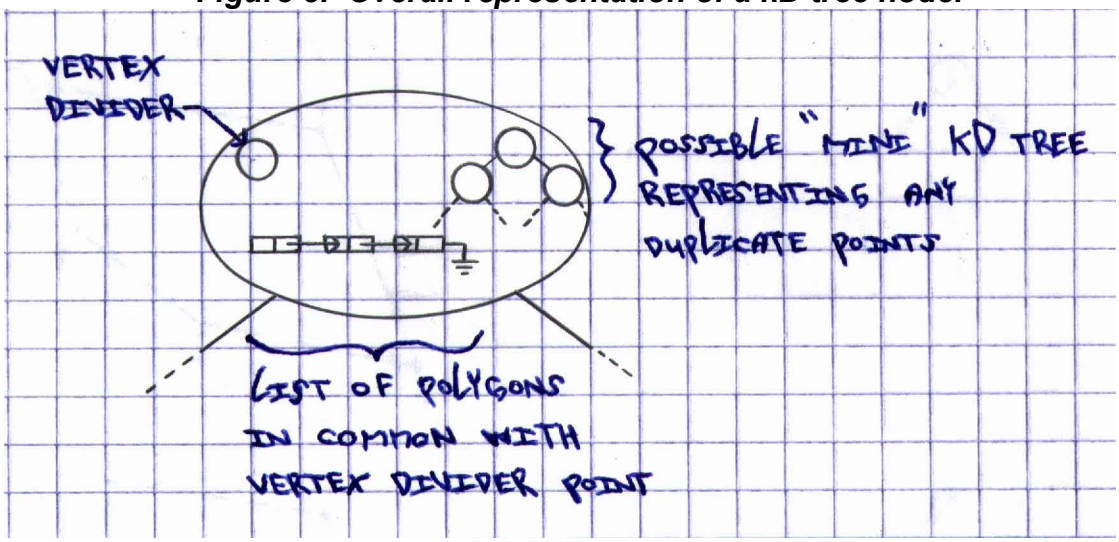


processed for selection. This 3D point selected from the main vertex pool is called a *vertex divider*. We call it a vertex divider because it is the *vertices* that do the actual subdividing of a 3D world. Since we have just processed an x coordinate value, the 3D point just processed is called an *x vertex divider*. We can also obtain a y or a z vertex divider in the same way.

Once we obtain this x vertex divider, we insert it into the kD tree. We still have to check if there are other 3D points within the main vertex pool that have the same x coordinate value as this x vertex divider that was just found. We, then, eventually hunt for these 3D duplicates, if any, and, if found, insert them into the same kD tree node as the current x vertex divider. In addition, we flag these 3D duplicates, indicating that they, too, have been processed. Still working with the same kD tree node in which the x vertex divider has been inserted, we search through an entire global polygon list--representing the current 3D world in question--and find any polygons whose vertices match--or are in common--with this current x vertex divider. If we find any of these polygons, we create a linked list hanging from this kD tree node and insert--in any order--these polygons into this created list. We do exactly the same for any duplicate 3D points in the same node as the x vertex divider is in. Ohh yeah, almost forgot! Remember those duplicate 3D points? How are they stored relative to the vertex divider? Well, within the same kD tree node that we're working with, the duplicate points are

actually organized into another *mini kD tree* that is stored within the same kD tree node as the current vertex divider. If confused, look at figure 3 to get the whole picture. Anyway, kD tree nodes basically come in two varieties: those nodes that contain *exactly one* vertex divider and those nodes that contain a vertex divider *and* its

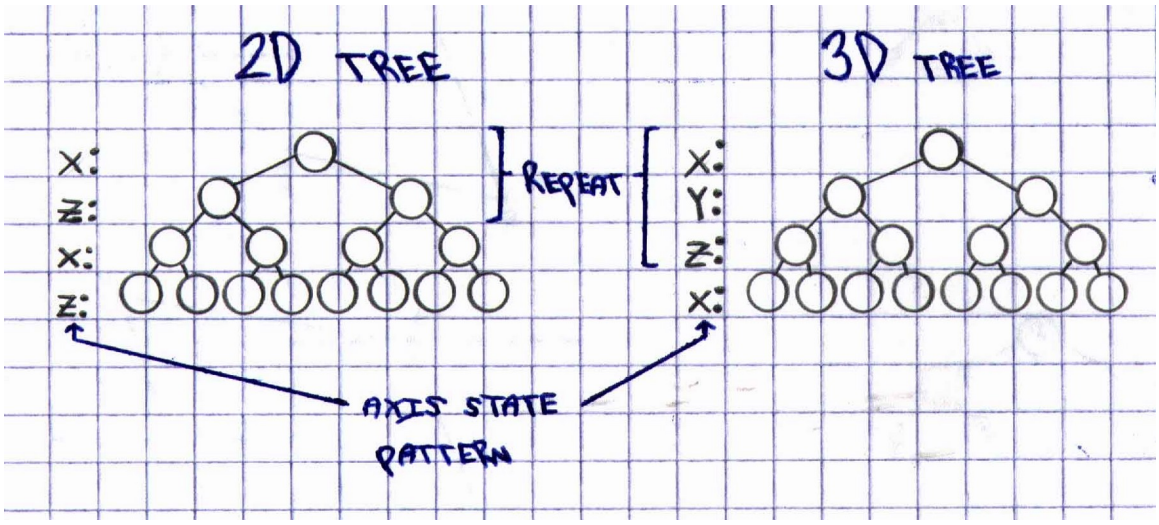
Figure 3. Overall representation of a kD tree node.



duplicate 3D points. Both types of nodes, of course, contain those linked list of polygons!

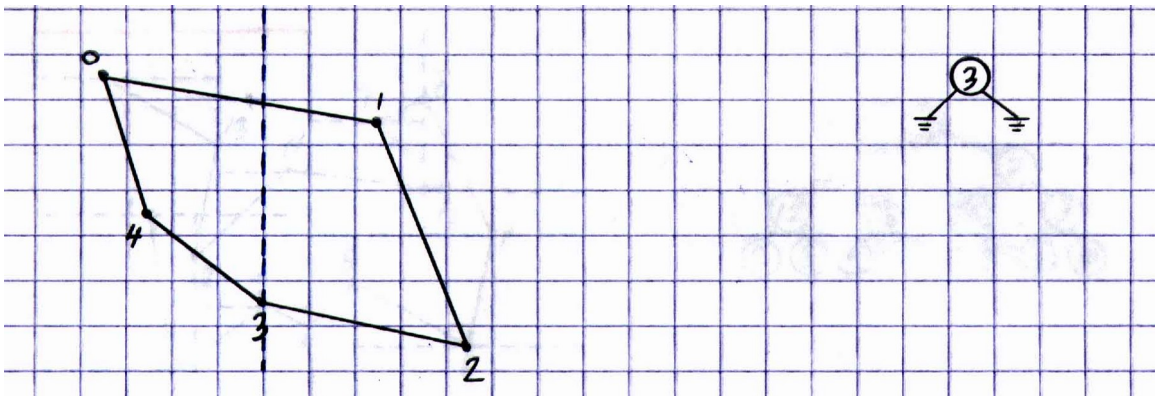
Once we've selected an x vertex divider, its duplicate 3D points--and an accompanying mini kD tree--if any, and a collection of linked lists of polygons, the current kD tree node has, then, officially been fully processed. This must be a sigh of relief for you! Quite the contrary, however! What I show you next is the heart of the kD tree building process. Everything we do from now on is just a recursive reflection of the last two previous paragraphs. The only thing different now is that instead of finding x vertex dividers, we switch our attention to finding *z vertex dividers*. So basically, we have this consistent and *alternating* mechanism of finding vertex dividers. First we find x vertex dividers and then we find z vertex dividers, and back to x, and so on, constantly alternating our build algorithm's *axis state* until we have no more 3D points to process in the original vertex pool. Thus, a pattern emerges like that in figure 4--for 2D trees, at least--where we have axis states alternating all the way down the entire kD tree.

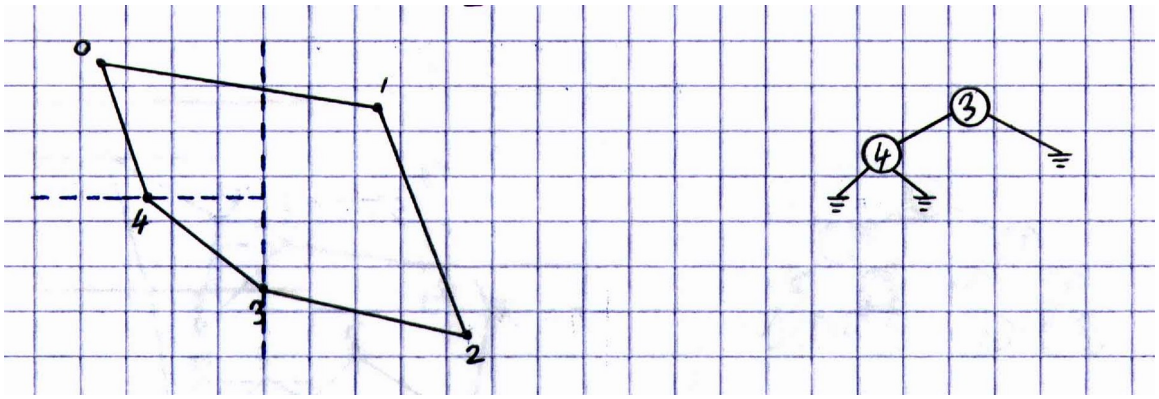
Figure 4. Alternating axis state pattern in a kD tree.

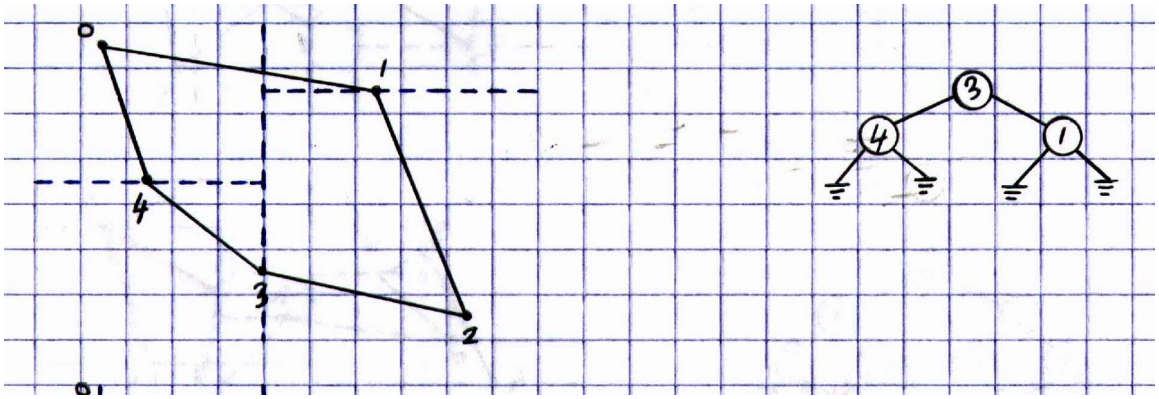


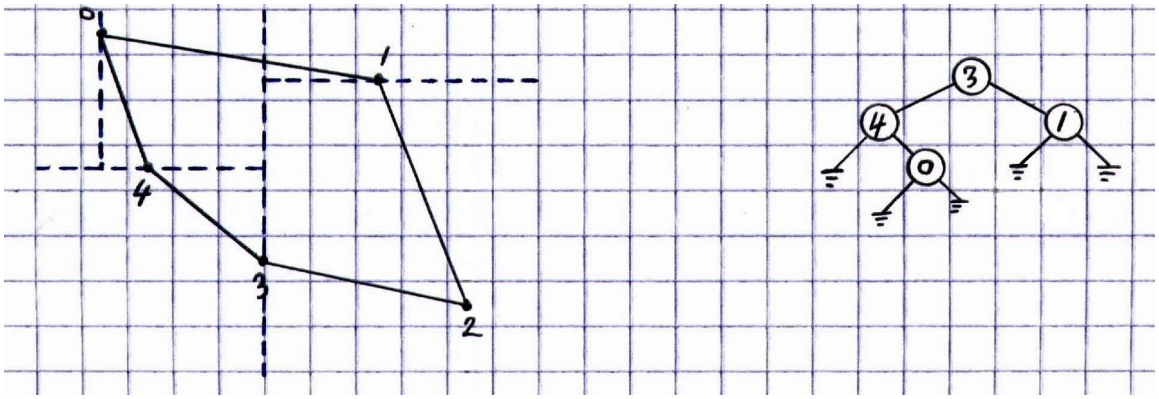
I wasn't thinking of providing a software demo on how the kD tree creation process actually goes. Do to timing constraints, I have provided an illustration that shows (step-by-step) on how a 3D world is converted--and stored--into a kD tree. See figure 5 in all its glory. This illustration should definitely help! If you understand this illustration thoroughly, then you fully understand the

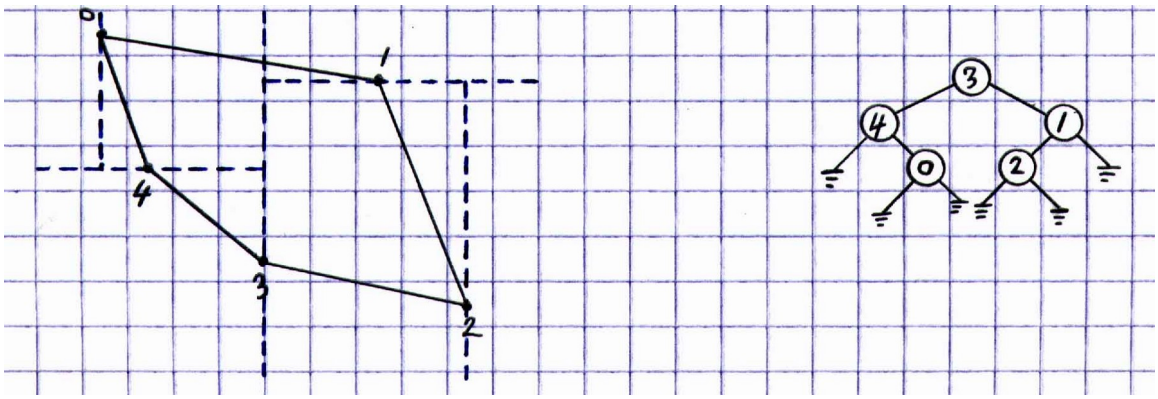
Figure 5. Step-by-step example of the kD tree building process.




0
1
4
3
2
3
4


0
1
4
3
2
3
4
1


0
1
4
3
2
3
4
1
0



kD tree creation process. Before I almost forget, here is some general code that builds a kD tree, where k = 2:

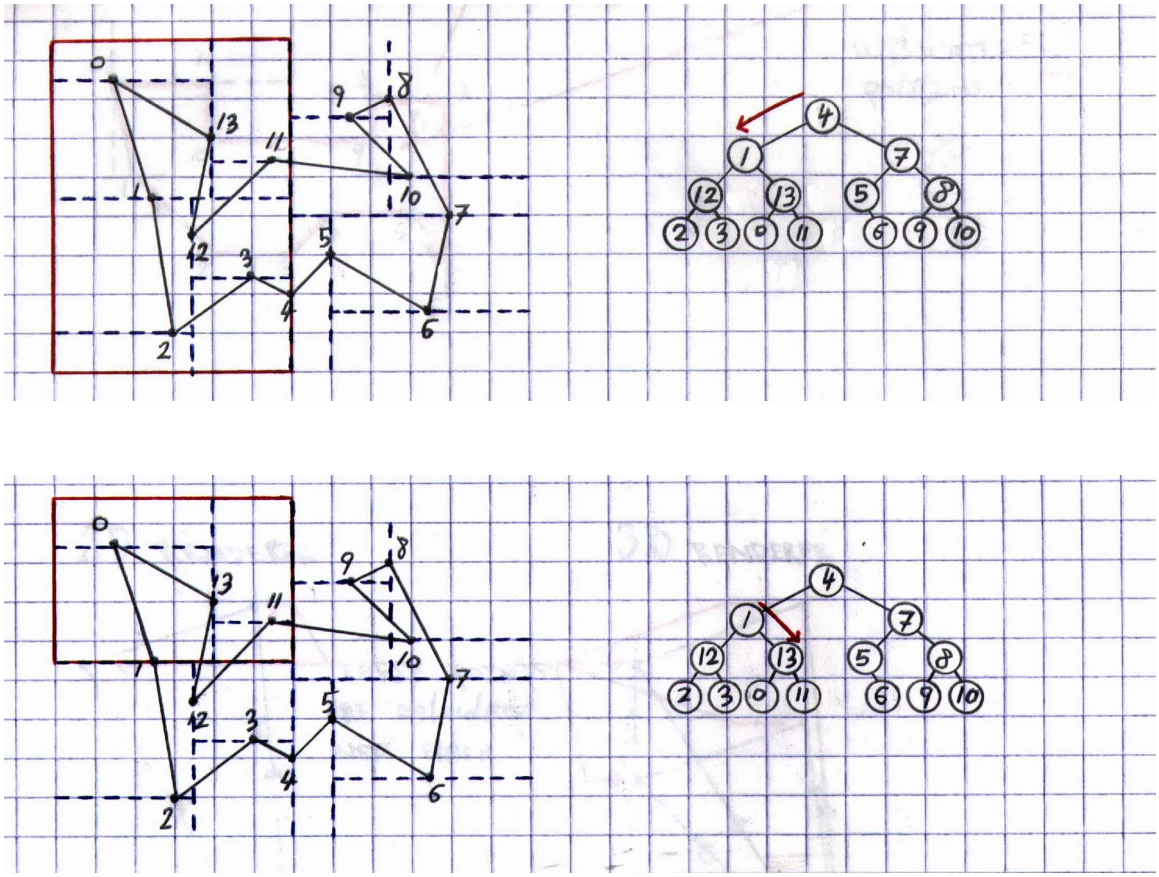
```
void buildTree(node *root)
{
     if (root == NULL)
          return;

     if (axisState == X_AXIS)
     {
          //find x vertex divider
          //rest of block to split vertex pool
          //into 2 lists
     }

     else //must be Z_AXIS
     {
          //find z vertex divider
          //rest of block to split vertex pool
          //into 2 lists
     }

     axisState^= 1; //flip "axisState"
     buildTree(root->left);
     buildTree(root->right);
}
```

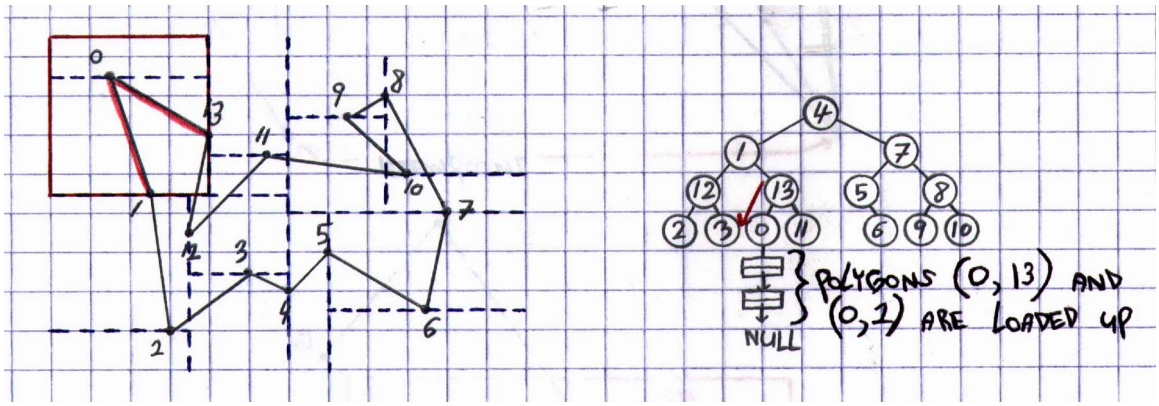
Getting to a Leaf: The kD Tree Traversal

Traversal from a kD tree root node to any of the leaves is easy. As a matter of fact, it is even easier than the BSP traversal! In the BSP traversal, we must calculate the dot product at every node that we visit. This dot product is a test to see on which side of the current polygon the viewpoint is on. As we all

know in the computer graphics world, dot products are rather expensive if many of them are done. In the kD tree traversal, however, we eliminate all these dot product tests and just perform *exactly one* simple comparison (with each kD node we visit) with one of the viewpoint coordinate values--x, y, or z--and the current vertex divider value. For instance, if we got to a node that contains a y vertex divider, we compare this y value with the y value of our viewpoint and proceed to the appropriate child node. See figure 6 for a simple demonstration of our kD tree traversal.

Figure 6. A simple example of a kD tree traversal.





Anyway, here is some general code to traverse a kD tree:

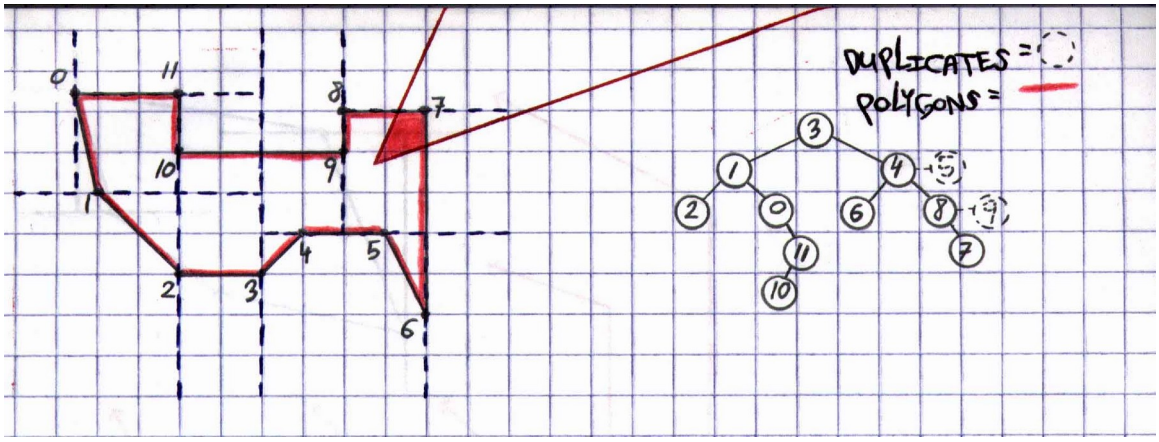
```
void traverseTree(node *root, point viewPoint)
{
     if (root == NULL)
          return;

     else
     {
          if (axisState == X_AXIS)
          {
               if (viewPoint.x < root.vertexDivider)
                    traverseTree(root->left)
               else
                    traverseTree(root->right);
          }

          else //must be Z_AXIS
          {
               if (viewPoint.z < root.vertexDivider)
                    traverseTree(root->left);
               else
                    traverseTree(root->right);
          }
     }

     axisState^= 1;
}
```

Rendering from a Leaf

Once we traverse all the way down a kD tree, we eventually reach a leaf in which our viewpoint is in. We now have access to all of the polygons within this leaf. Now, we perform the usual HSR test on only the polygons that are in this leaf and render--in any order--the ones that we actually see. For *each* polygon that gets rendered, we send the scan-converted version of the polygon to our image-precision zero-overdraw algorithm. Here, it is guaranteed that *exactly one* pixel will ever get drawn by a 3D polygon that is part of a 3D polygonal world. So, in the best case scenario, one polygon can cover the entire view plane and no more rendering is needed for this frame due to the zero-overdraw algorithm. But, if we still have *at least one* pixel to draw, we continue to test and render, if any, polygons that are seen from the current position and orientation of our viewpoint.

Eventually, we test *all* of the polygons that are *only* within the same leaf as our viewpoint and must make a big decision in our object-precision algorithm: If we have rendered the *entire* screen of pixels from *only* the set of polygons that belong in the same leaf as the viewpoint--as can be seen from figure 7--rendering for the current frame has just come to a close and the next frame of animation awaits our attention. Otherwise, our main object-precision algorithm has more

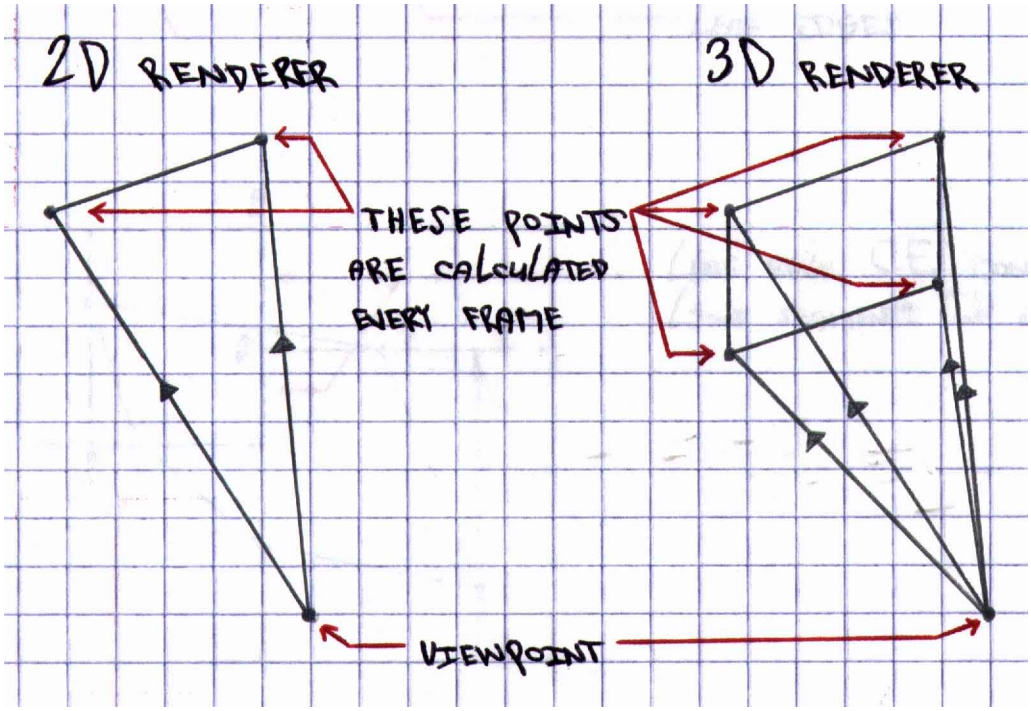
Figure 7. A viewpoint's FOV seeing only the polygons within the same leaf as the viewpoint.



work to do. Dare to read on!

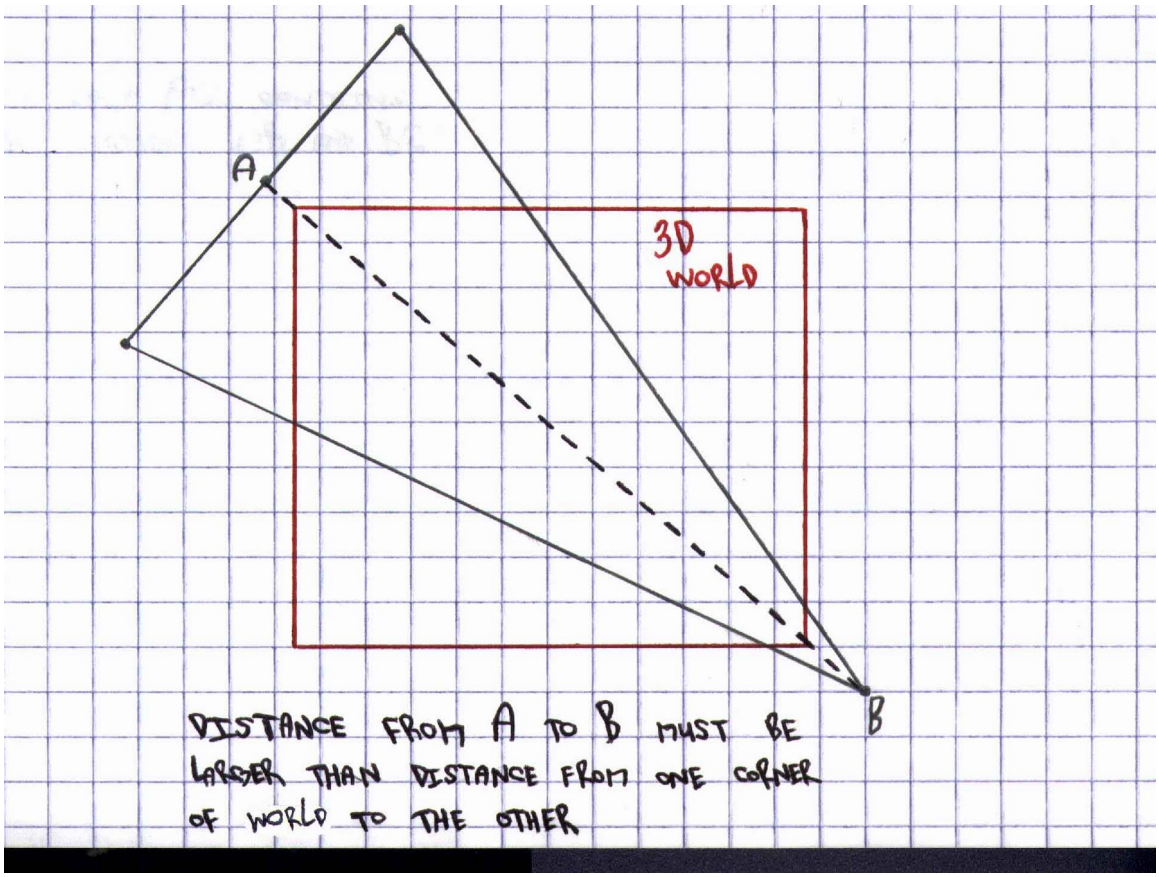
Basically what we do now is perform some kind of ray tracing, but not using too many rays. No, no! That will be far too expensive! What we do instead is cast only *two* rays from the viewpoint. Well, we do this at least in 2D. In 3D, we must cast at least *four* rays because our view volume is not a simple 2D triangle but a 3D viewing pyramid. Anyway, figure 8 shows all the details in a 2D and in a 3D implementation. As you can see from the figure, these rays are

Figure 8. The rays that are going to be used in our renderer.



actually line segments that are identical in length. Calculating these *FOV points* requires some simple trig. on your part. Also, given any 3D world, this FOV must be calculated such that it is big enough for this 3D world. See figure 9 for details. As you can see, no matter what

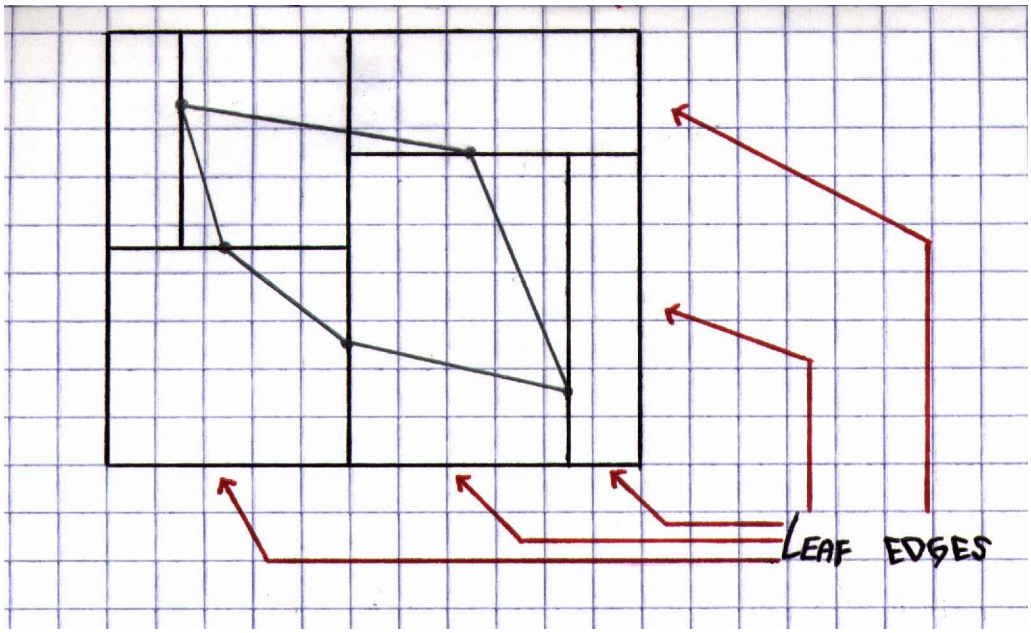
Figure 9. The FOV must be large enough so that no polygons are accidentally missed for HSR testing.



the viewpoint's position or orientation, the FOV will always go *slightly* past the edge (or edges) of the world itself, thus ensuring that no polygons ever get missed for HSR testing.

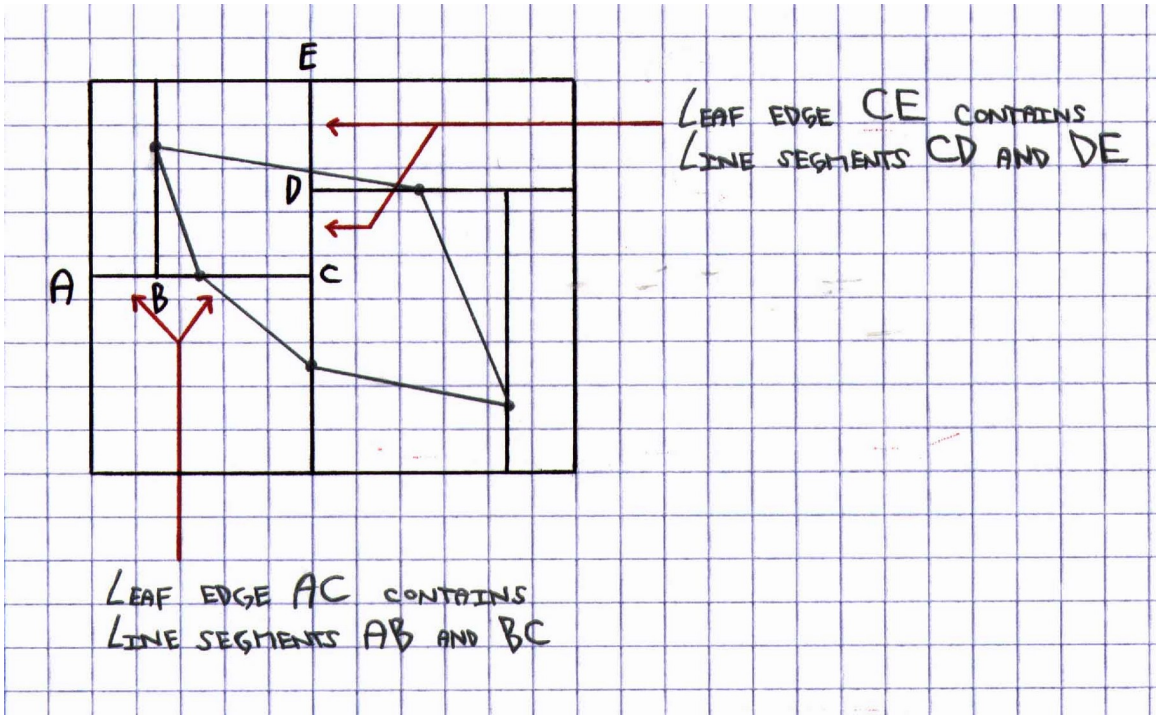
So, how do we figure out which polygons, if any, get rendered with these rays *and* how do we know which ones get rendered first? You're probably thinking that we need some kind of polygon sorting mechanism. WRONG!!! Actually, what we need is another geometric database that holds information about the edges bounding each kD node leaf. Remember, the edges that I'm talking about are NOT real polygon edges but imaginary *leaf* edges. See figure 10. As you can see, all the kD leaves are bounded by its leaf edges and are denoted by dark black lines.

Figure 10. All kD leaves are bounded by its leaf edges.



Still, we need one more piece of data: the smaller *line segments* that lie on each edge. These line segments are carved by the intersection--or intersections--of other perpendicular vertex dividers. See figure 11. Anyway, it is *these* line segments that are going to help us pick out the right kD leaves for testing at the right time, so that we can test the proper polygons for HSR

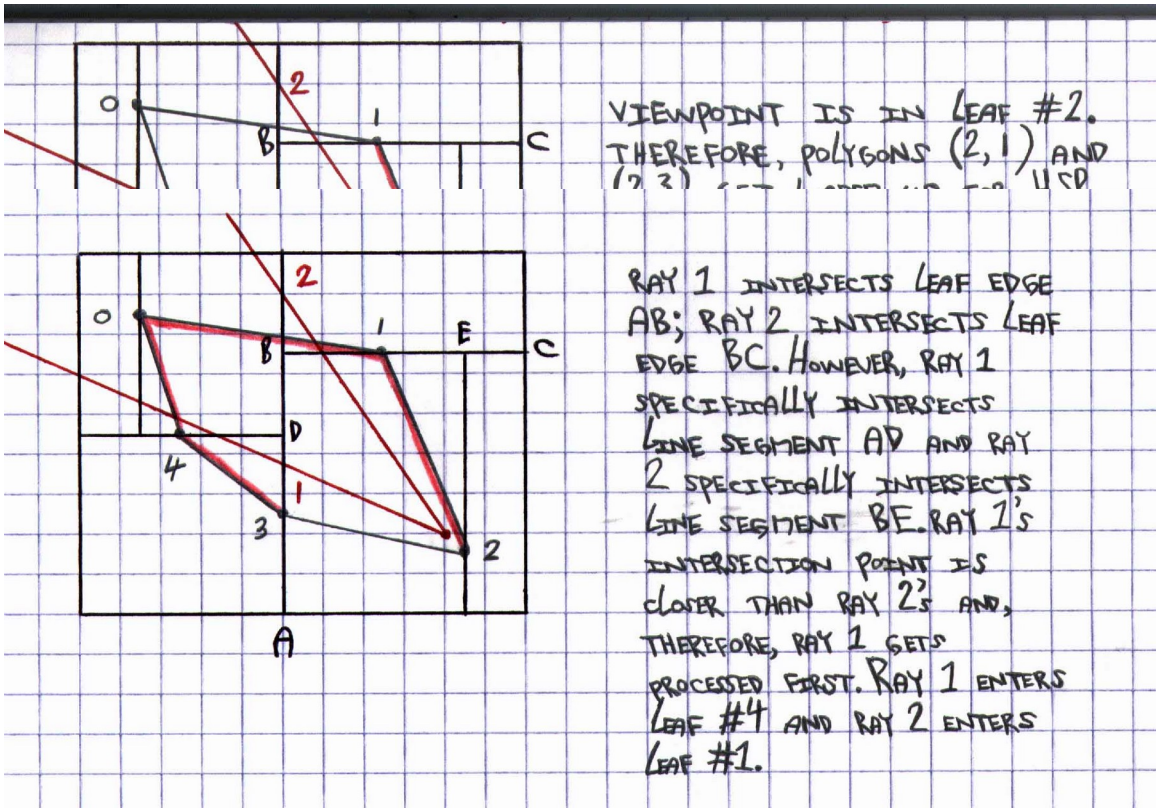
Figure 11. Leaf edges are made up of smaller line segments.

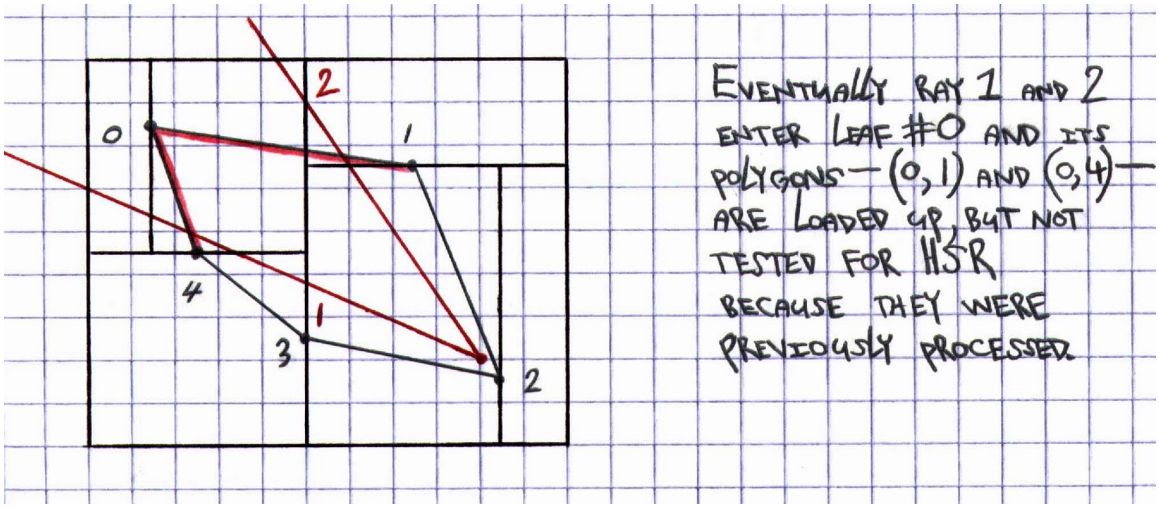


testing at the right time, and, therefore, ensure that proper polygon rendering order is established.

So, how does all this fit together? Well, it goes like this. The *very* first kD leaf that gets found--via our kD tree traversal--will always have our viewpoint in it. As before, we have access to all of the polygons in this current kD leaf and we can render, if any, polygons in any order. Now, if more rendering might need to be done, we must perform extra rendering tests *outside* of this leaf. So, this is the test that we perform: for *each* ray that we shoot out, we must perform a parametric line-clipping test with each of the leaf edges associated with the current kD leaf that we are still currently in--in this case, our viewpoint leaf. When we *do* find a leaf edge that intersects with our current ray, we then stop testing for the other remaining leaf edges in the current kD leaf and start testing *along* this intersected leaf edge that we just found and try to determine in which of its line segments the same ray intersects. We then use this intersected line segment to gain access to info regarding the adjacency of other immediate kD leaves that are touching our viewpoint leaf--or, for that matter, any leaf in general. In other words, for every line segment that we find intersected by our current ray, we, in turn, find the actual leaves that touch each other along this same line segment. Since only *two* leaves can ever touch each other--well, at least in 2D--we know that we are currently in one of these leaves. From the results of the parametric line-clipping tests--and the line segment found--we know in which *new* leaf our current ray has entered, and the entire rendering process repeats for this new found kD leaf. Once again, being the nice guy I am, I'm going to show you a step-by-step demo of how the kD tree renderer renders a sample sceen.

Figure 12. Step-by-step example of the kD tree rendering process.





In addition, here is some pseudo-code for the overall kD tree renderer:

```
void kDtreeRenderer(node *viewPointLeaf)
{
     node *currLeaf= *viewPointLeaf;

     while(at least one pixel needs to be drawn)
     {
          for (each ray) //2 rays in 2D; 4 rays in 3D
          {
               while(1)
               {
                    if (ray intersects
                    currLeaf.currLeafEdge)
                         break;
               }
          }

          //Sort the rays based on their distances from
          //the viewpoint to their respective
          //intersection points, sorting from smallest
          //distance to largest distance.

          for (each ray)
          {
               //Based on this ray's intersection
               //point, figure out between which
               line //segment on same intersected leaf
               edge //the current nearest ray falls on.
```

```
                    //From this line segment, we know
                    //exactly in which new leaf the current
                    //ray has entered.
                    node *newLeafEntered=
                         currLeaf.currLeafEdge.currLineSeg;

                    currLeaf= newLeafEntered;

                    //Perform HSR test for polygons in
                    //"currLeaf."

                    //Image-precision zero-overdraw code
                    //goes here.
               }
          }
     }
```

Conclusion

You have seen some strengths and weaknesses of the kD tree algorithm. You have experienced first-hand that building a kD tree is a little more complex than building a BSP tree. However, once this kD tree is built, traversing one is as easy as pie. In addition, you have also seen how the powerful kD tree renderer *immediately* begins to render what it sees within the 2D triangle or 3D view volume. The beauty of this algorithm is that there is *never* any need to test what lies outside of the view volume!