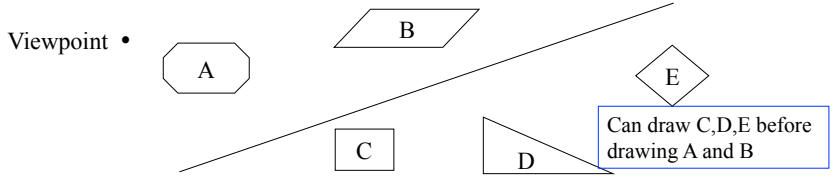
CS770/870 Spring 2017
Binary Space Partitioning

Preview

- Binary Space Partitioning (BSP)
  - preprocess scene by a tree of separating planes
  - can speed up some kinds of processing later
- For visible surface calculation
  - Especially effective if world model changes less frequently than viewing position

Separating Plane

- Given a plane and a view point
  - No polygon on the viewpoint side of the plane can be obscured by any polygon on the other side of the plane
  - Hence, can get correct visibility by drawing the other side first using a *painter's algorithm*
  - Can then define new planes to separating the subsets on each side of the first plane – and do this recursively

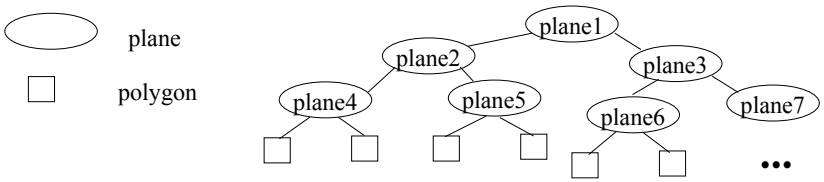


BSP Components

- *makeTree*: preprocessing module that converts a set of input polygons into a BSP tree.
- *traverse*: traverses the BSP tree for a given viewpoint to produce a priority order for drawing the polygons in an essentially *back-to-front* order

Partitioning

1. Find a separating plane to partition polygons into 2 sets: those on the + side of the plane and those on the - side of the plane.
2. For each subset, repeat.
   find a separating plane to partition those polygons …



Building the Tree

- How can we efficiently find separating planes?
- Can we keep the tree simple?
- Overview
  - select a polygon from the polygon list; use its plane as the root of the tree
  - partition remaining polygons to the left and right subtrees
  - if a polygon is intersected by a plane, use the plane to divide it into 2 parts

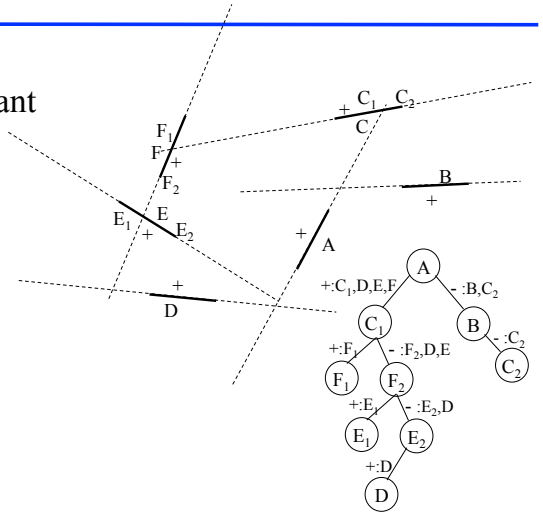
makeTree

```
Tree makeTree( polyList )
   if polyList empty
      return null
   else
      rootPoly = remove poly from polyList
      for each p in polyList
         if p on + side of root
            leftList.add( root.left, p )
         else if p on - side of root
            rightList.add( p )
         else
            splitPoly( p, rootPoly, leftList, rightList )
      node.left = makeTree( leftList )
      node.poly = rootPoly
      node.right = makeTree( rightList )
    return node
```


Build a Tree

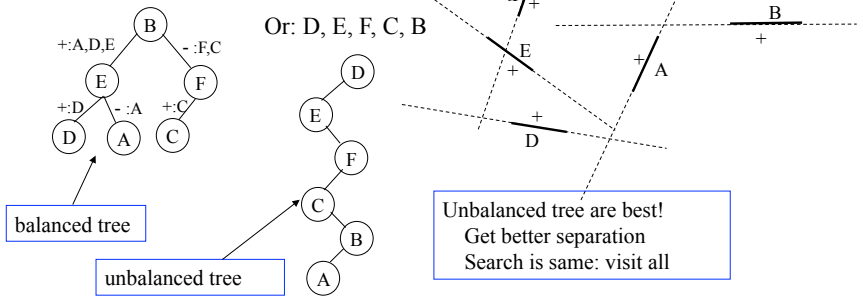
- Root choice at each level can be significant

1. Pick A as root; splits C
2. Pick C_1; splits F
3. Pick B, then C_2
4. Pick F_1 at node C_1
5. Pick F_2; splits E
6. Pick E_1
7. Pick E_2
8. Pick D



Alternate choices

- Suppose we start with B, then E and F



traverseTree

- Given a viewpoint, generate display

```
traverseTree( node, vp )
  if node not null
     if vp on - side of node.plane
       traverseTree( node.plus, vp )
       display( node.polygon )
       traverseTree( node.minus, vp )
     else // vp on + side of node.plane
       traverseTree( node.minus, vp )
       display( node.polygon )
       traverseTree( node.plus, vp )
```

- Notes
  1. display method transforms to image space, clips, etc.
  2. display on - side can be eliminated if do backface culling

Build Tree Variations

1. At each step, select polygon whose plane intersects fewest remaining polygons; lots of intersection tests.
2. Select *n* polygons randomly; select the one of these that cuts fewest remaining polygons
   - Fuchs did experiments; recommended about 5 candidate polygons gives reasonable balance of performance to create and reducing intersections

Dynamic BSP Tree Creation

- BSP tree creation time doesn't matter if scene never changes, but that is a limited environment.
- If most objects don't change at run-time, there are algorithms to "update" a tree.
- Easy algorithm:
  - remove old polygons; relatively easy
  - add polygons at new locations
  - but, not very efficient
- More complex algorithms try to modify in place