Painter’s Algorithm

- Object-Order Algorithm
- Sort objects by depth
- Display them in back-to-front order
Painter’s Algorithm

• Sort polygons by farthest depth.
• Check if polygon is in front of any other.
• If no, render it.
• If yes, has its order already changed backward?
  – If no, render it.
  – If yes, break it apart.

Which polygon is in front?

Our strategy: apply a series of tests.
  – First tests are cheapest
  – Each test says poly1 is behind poly2, or maybe.
1. If min z of poly1 > max z poly2, 1 in back.
2. The plane of the polygon with smaller z is closer to viewer than other polygon.
\[(a,b,c,)*(x,y,z) >= d.\]

3. The plane of polygon with larger z is completely behind other polygon.

4. Check whether they overlap in image
   a. Use axial rectangle test.
   b. Use complete test.

---

Non-Overlapping x or y

Overlap projection

B is on one side of A
Problem Cases: Cyclic and Intersecting Objects

- Solution: split polygons

- Advantages of Painter’s Algorithm
  - Simple
  - Easy transparency

- Disadvantages
  - Have to sort first
  - Need to split polygons to solve cyclic and intersecting objects

Painter’s Algorithm
Spatial Data-Structures for Visibility

- Octrees (generalization of Binary trees in 1D and Quad trees in 2D)
- Binary-Space Partition Trees (BSP trees) (an alternative generalization of Binary trees in 1D)
- Subdividing architectural buildings into cells (rooms) and portals (doors/windows)

Portals

- Similar to view-frustum culling
- View-independent
- Preprocess and save a list of possible visible surfaces for each portal
Cells and Portals

Images courtesy: Dave Luebke, UVa
Cells & Portals

Images courtesy: Dave Luebke, UVa
BSP Trees

• Idea
  Preprocess the relative depth information of the scene in a tree for later display

• Observation
  The polygons can be painted correctly if for each polygon F:
  – Polygons on the other side of F from the viewer are painted before F
  – Polygons on the same side of F as the viewer are painted after F
Building a BSP Tree

**Typedef struct**

\[
\begin{align*}
\text{struct} & \quad \{ \\
\quad & \quad \text{polygon root;} \\
\quad & \quad \text{BSP\_tree } *\text{backChild, } *\text{frontChild;} \\
\} \quad \text{BSP\_tree;}
\end{align*}
\]

**BSP\_tree  *makeBSP(polygon *list)**

\[
\begin{align*}
\quad & \quad \text{if( list } = \text{ NULL) return NULL;} \\
\quad & \quad \text{Choose polygon F from list;} \\
\quad & \quad \text{Split all polygons in list according to F;} \\
\quad & \quad \text{BSP\_tree* node } = \text{ new BSP\_tree;} \\
\quad & \quad \text{node->root } = \text{ F;} \\
\quad & \quad \text{node->backChild } = \text{ makeBSP( polygons on front side of F );} \\
\quad & \quad \text{node->frontChild } = \text{ makeBSP( polygons on back side of F );} \\
\quad & \quad \text{return node;} \\
\end{align*}
\]
Displaying a BSP Tree

```c
void displayBSP ( BSP_tree *T )
{
    if ( T != NULL ) {
        if ( viewer is in front of T->root ) {    // display backChild first
            displayBSP ( T->backChild );
            displayPolygon ( T->root );
            displayBSP ( T->frontChild );
        }
        else { // display frontChild first
            displayBSP ( T->frontChild );
            displayPolygon ( T->root );
            displayBSP ( T->backChild );
        }
    }
}
```

Display order: 4, 5b, 3, 5a, 2, 1 (only 3 is front facing)
BSP Trees: Analysis

• Advantages
  – Efficient
  – View-independent
  – Easy transparency and antialiasing

• Disadvantages
  – Tree is hard to balance
  – Not efficient for small polygons