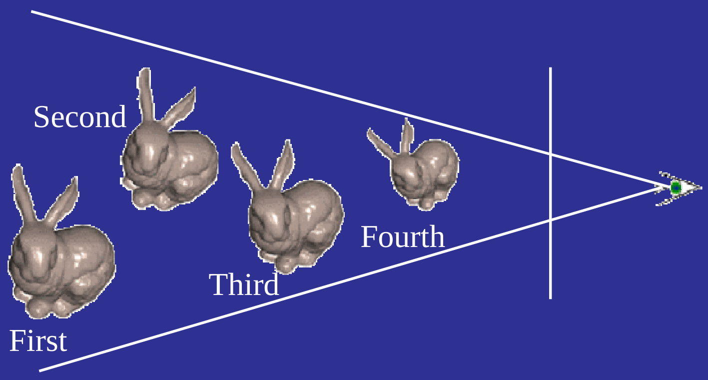
Algorithms for Visibility Determination

- Object-Order
  - Sort the objects and then display them
- Image-Order
  - Scan-convert objects in arbitrary order and then depth sort the pixels
- Hybrid of the above

Painter's Algorithm

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

Painter’s Algorithm



Painter’s Algorithm

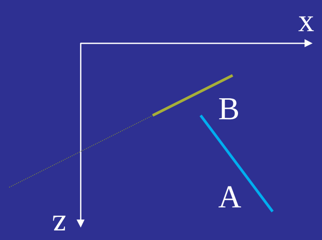
- Sort polygons by farthest depth.
- Check if polygon is in front of any other.
- If no, render it.
- If yes, has its order already changed backward?
  - If no, render it.
  - If yes, break it apart.

Which polygon is in front?

Our strategy: apply a series of tests.

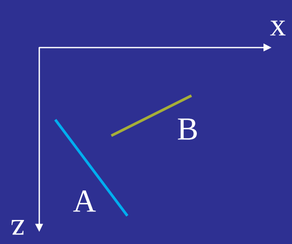
- First tests are cheapest
- Each test says poly1 is behind poly2, or *maybe.*

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.

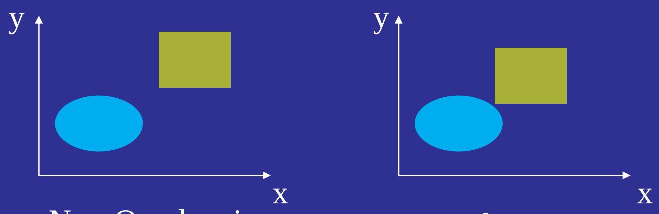


3. The plane of polygon with larger z is completely behind other polygon.

4. Check whether they overlap in image

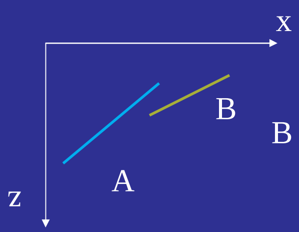
a. Use axial rectangle test.

b. Use complete test.



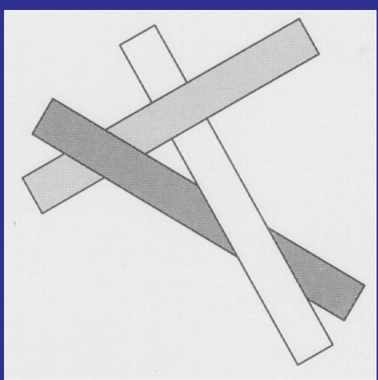
Non-Overlapping x or y

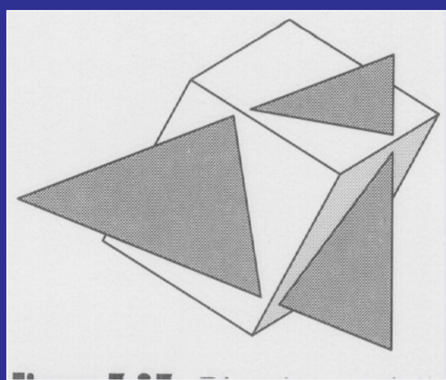
Overlapping projection



B is on one side of A

Problem Cases: Cyclic and Intersecting Objects





Painter's Algorithm

- 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

Z-Buffer Algorithm

- Image precision, object order
- Scan-convert each object
- Maintain the depth (in Z-buffer) and color (in color buffer) of the closest object at each pixel
- Display the final color buffer
- Simple; easy to implement in hardware

Z-Buffer Algorithm

```
for( each pixel(i, j) )        // clear Z-buffer and frame buffer
{
    z_buffer[i][j] = far_plane_z;
    color_buffer[i][j] = background_color;
}

for( each face A)
    for( each pixel(i, j) in the projection of A)
    {
        Compute depth z and color c of A at (i,j);
        if( z > z_buffer[i][j] )
        {
            z_buffer[i][j] = z;
            color_buffer[i][j] = c;
        }
    }
```

Efficient Z-Buffer

- Incremental computation
- Polygon satisfies plane equation

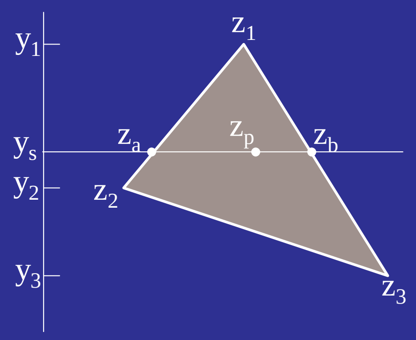
$$Ax + By + Cz + D = 0$$

- Z can be solved as

$$z = \frac{-D - Ax - By}{C}$$

- Take advantage of coherence
  - within scan line: $\Delta z = -\frac{A}{C}\Delta x$
  - next scan line: $\Delta z = -\frac{B}{C}\Delta y$

Z Value Interpolation



$$z_a = z_1 - (z_1 - z_2)\frac{y_1 - y_s}{y_1 - y_2}$$

$$z_b = z_1 - (z_1 - z_3)\frac{y_1 - y_s}{y_1 - y_3}$$

$$z_p = z_b - (z_b - z_a)\frac{x_b - x_p}{x_b - x_a}$$

Z-Buffer: Analysis

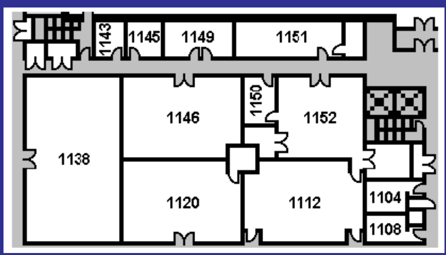
- Advantages
  - Simple
  - Easy hardware implementation
  - Objects can be non-polygons
- Disadvantages
  - Separate buffer for depth
  - No transparency
  - No antialiasing: one item visible per pixel

Spatial Data-Structures for Visibility

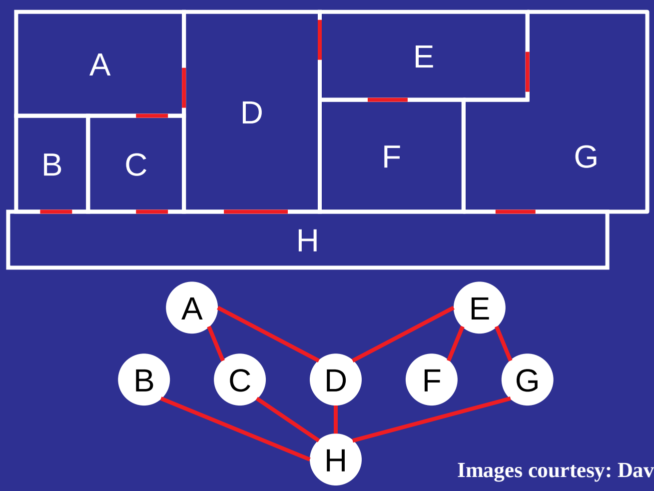
- Octrees (generalization of Binary trees in 1D and Quad trees in 2D)
- Binary-Space Partition Trees (BSP trees) (an alternative generalization of Binary trees in 1D)
- Subdividing architectural buildings into cells (rooms) and portals (doors/windows)

Portals

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

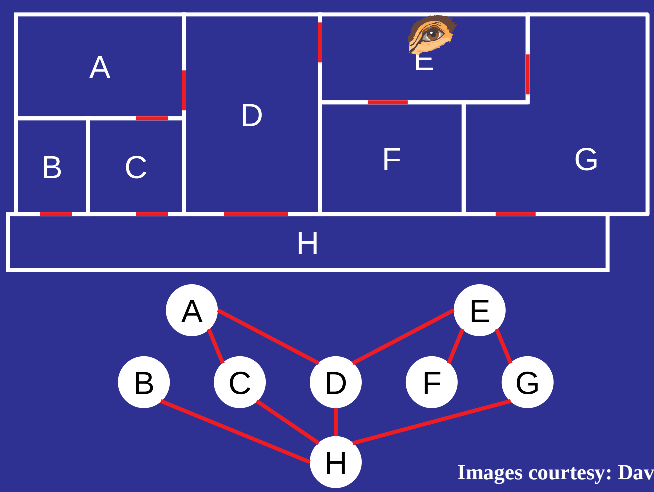


Cells and Portals



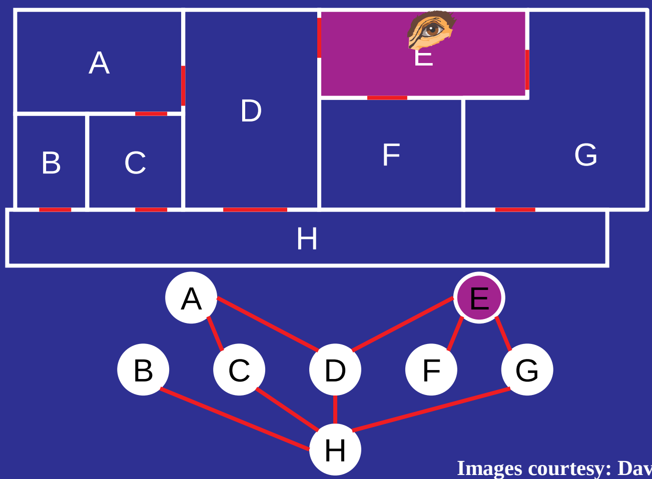
Images courtesy: Dave Luebke, UVa

Cells and Portals



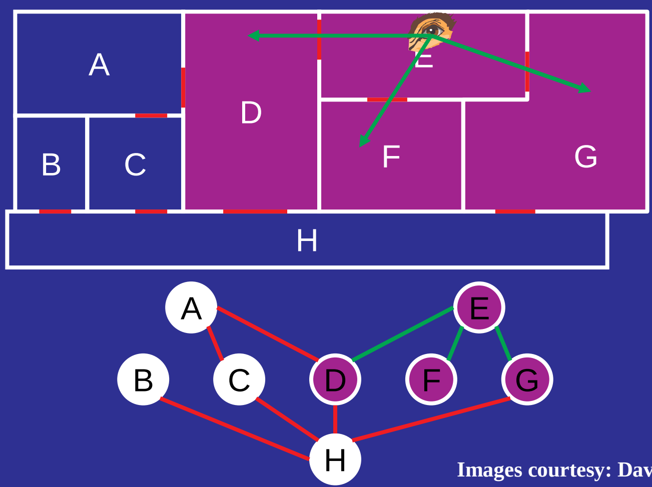
Images courtesy: Dave Luebke, UVa

Cells & Portals



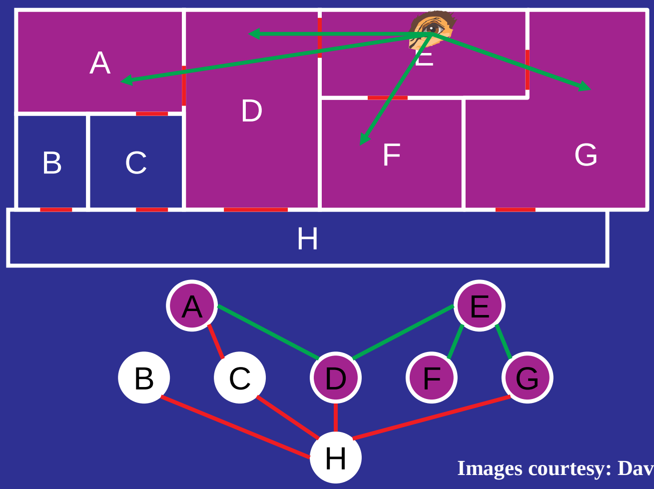
Images courtesy: Dave Luebke, UVa

Cells & 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

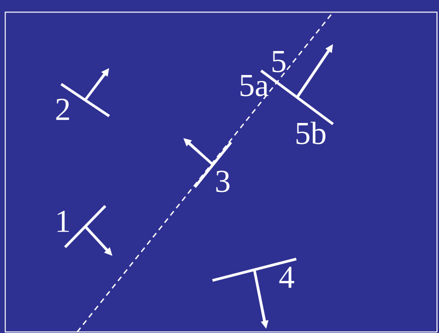
- Observation

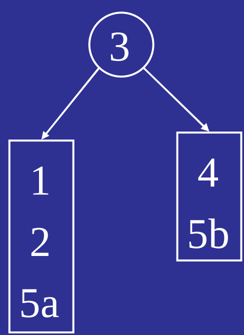
  The polygons can be painted correctly if for each polygon F:
  - Polygons on the other side of F from the viewer are painted before F
  - Polygons on the same side of F as the viewer are painted after F

Building a BSP Tree

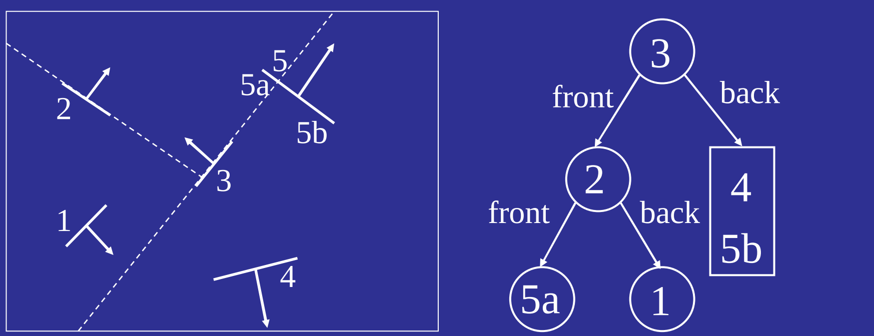
```
Typedef struct {
    polygon root;
    BSP_tree *backChild, *frontChild;
} BSP_tree;

BSP_tree  *makeBSP(polygon *list)
{
    if( list = NULL) return NULL;

    Choose polygon F from list;
    Split all polygons in list according to F;

    BSP_tree* node = new BSP_tree;
    node->root = F;
    node->backChild = makeBSP( polygons on front side of F );
    node->frontChild = makeBSP( polygons on back side of F );
    return node;
}
```

Building a BSP Tree (2D)

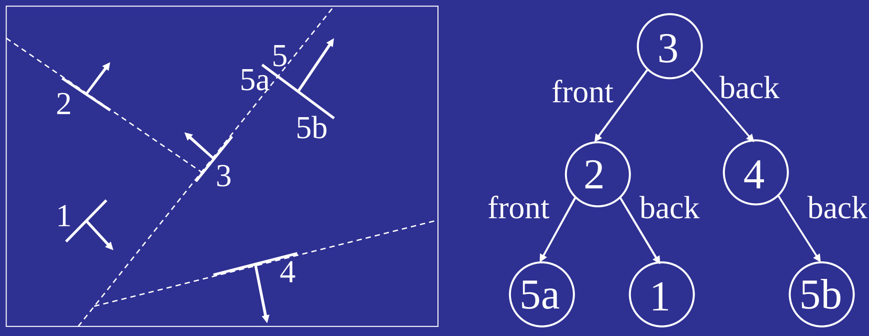




Building a BSP Tree (2D)



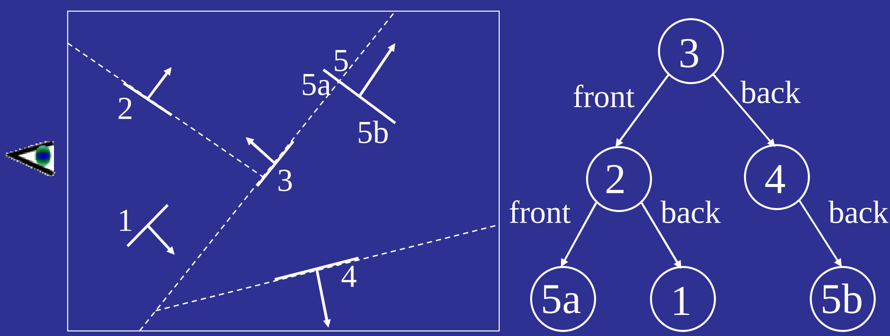
Building a BSP Tree (2D)



Displaying a BSP Tree

```
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 );
            }
}
```

Displaying a BSP Tree



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