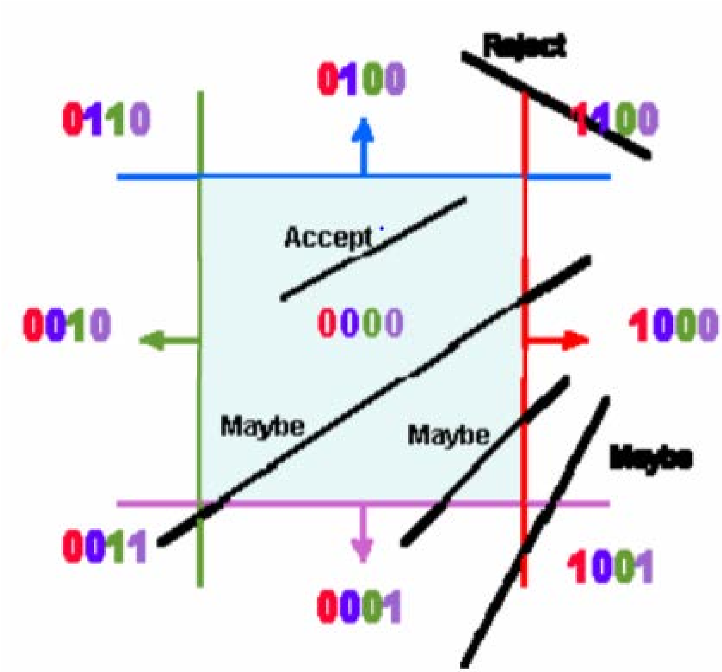
CSC 4356
Interactive Computer Graphics
Lecture 13: Visibility

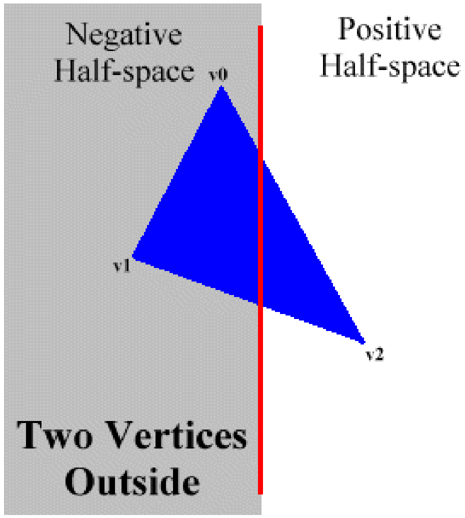
Jinwei Ye

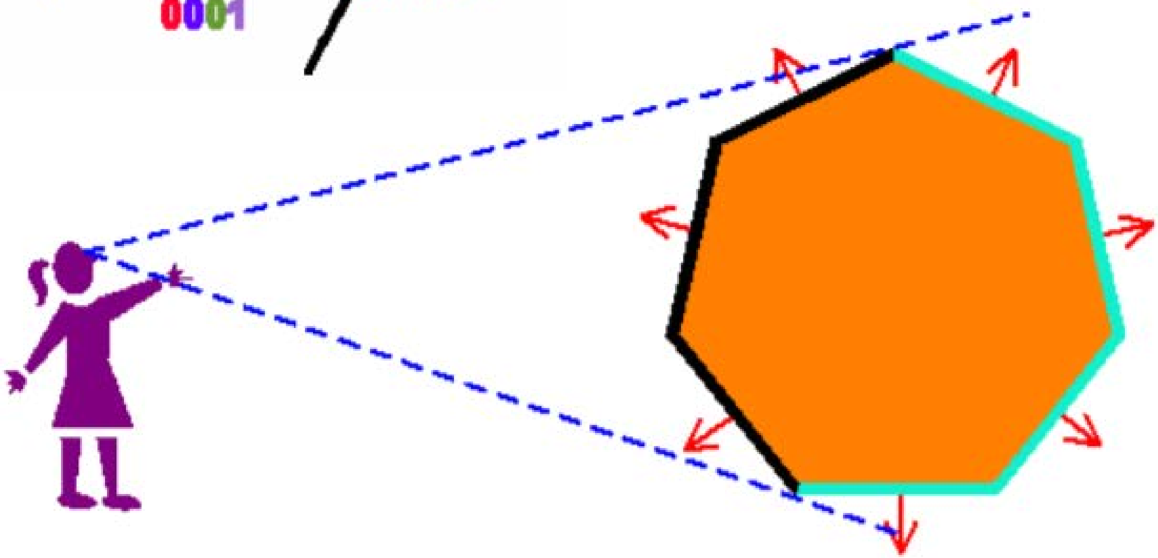
http://www.csc.lsu.edu/~jye/CSC4356/

Tue & Thu: 10:30 - 11:50am
218 Tureaud Hall

Clipping & Culling

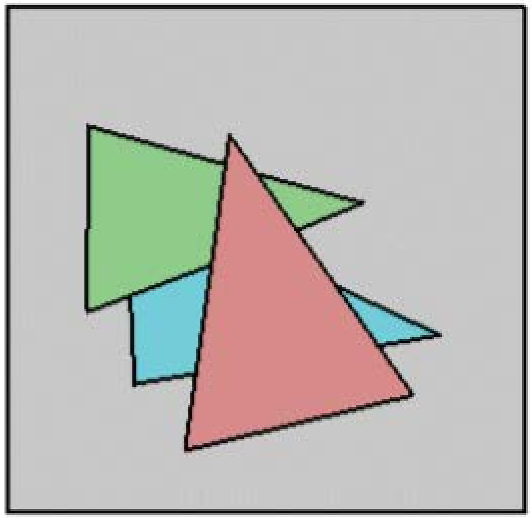


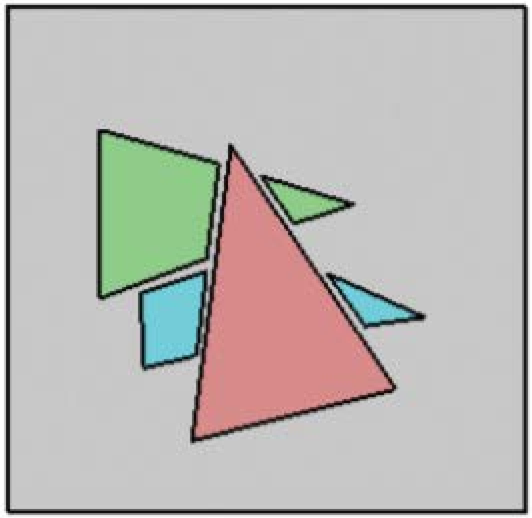




Visibility

- Problem: for most scenes and viewpoints, some polygons will overlap and cause occlusions. So we must determine which portion of each polygon is visible to eye

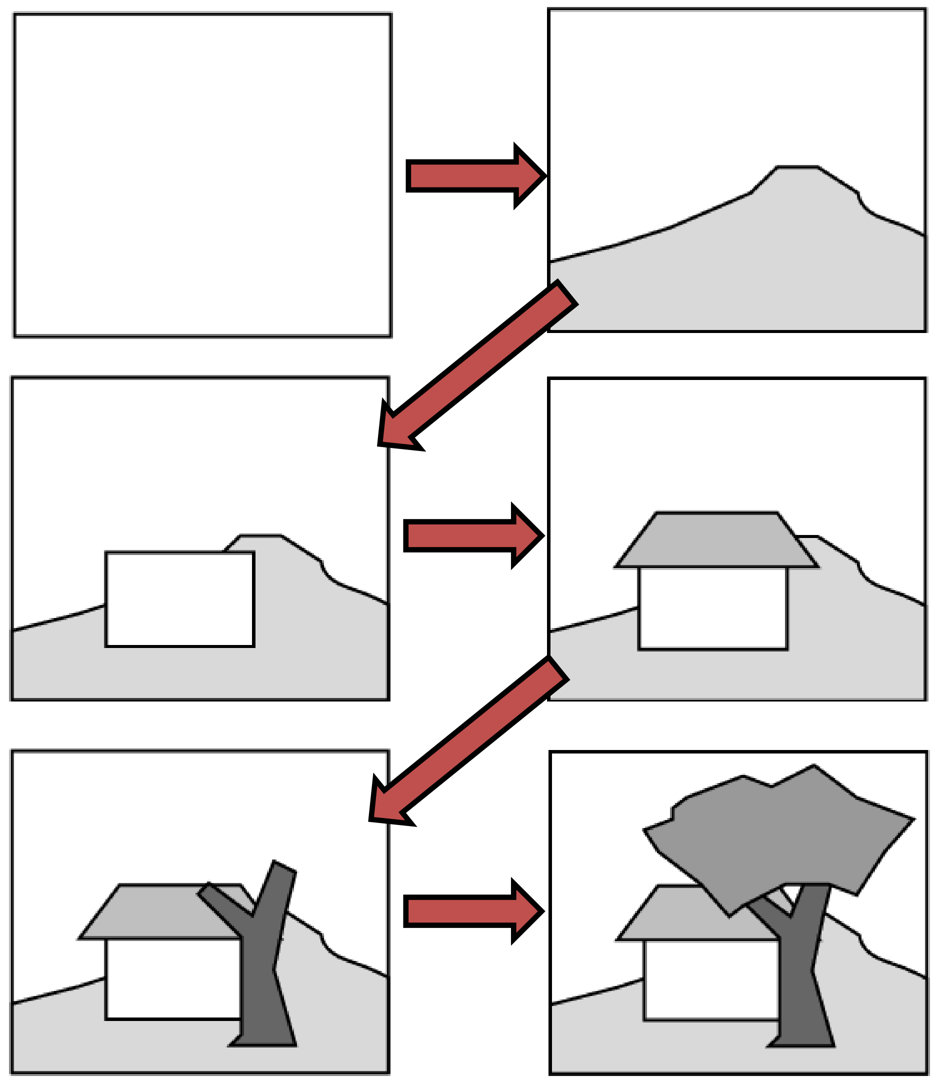




Painter’s Algorithm

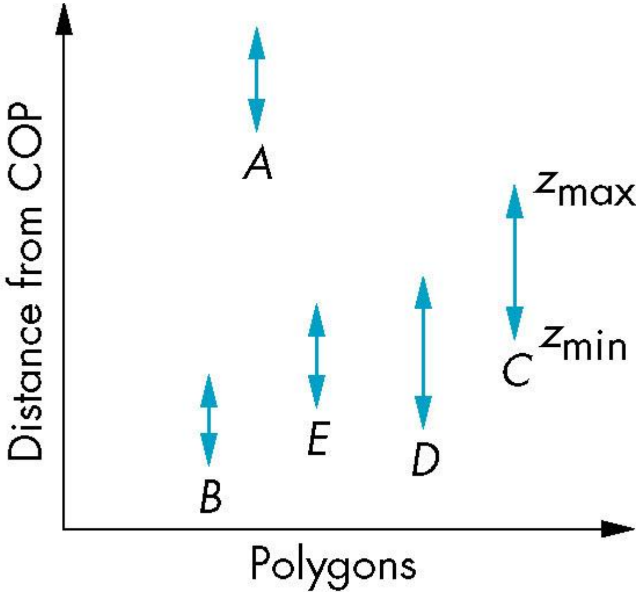
Draw primitives from back to front

- Depth sorting

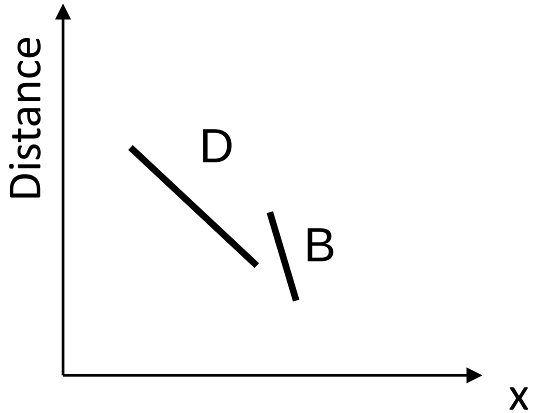


Painter's Algorithm

- Idea: Sort primitives by *minimum* depth, then rasterize from farthest to nearest
- When there are depth overlaps, do more tests of bounding areas to see if one actually occludes the other



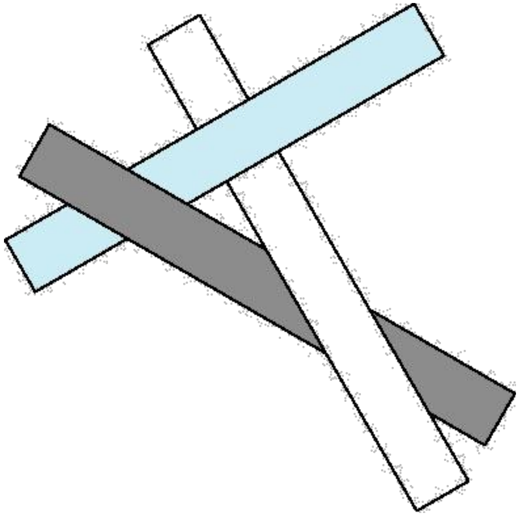
Paint order:
A → C → E → D→ B

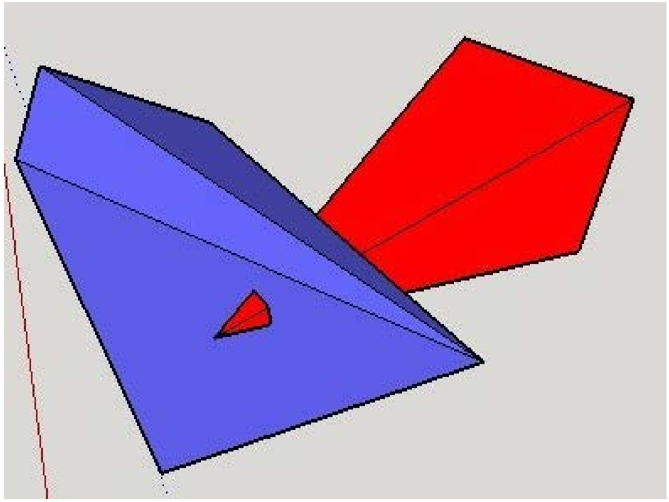


Reorder B & D:
A → C → E → B→ D

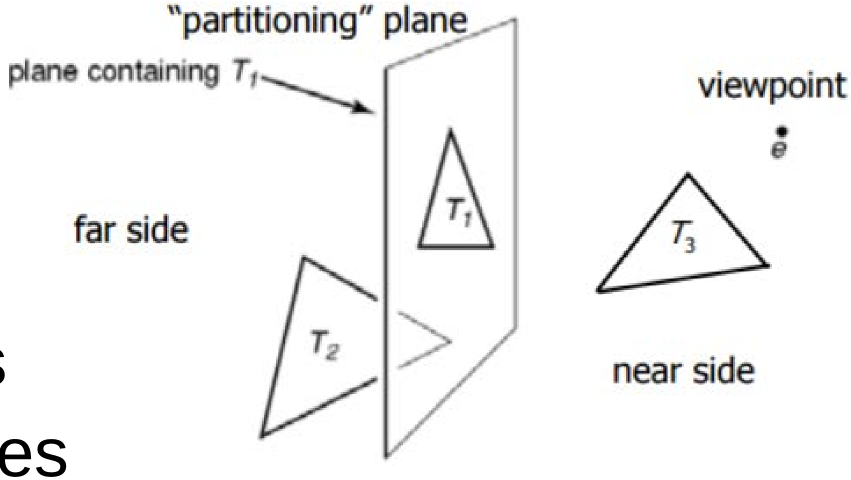
Problems with Painter's

- Invisible parts have already been painted
  - – Waste computation
- Cyclical overlaps and interpenetration are problematic
  - – Impossible to determine depth order





BSP Trees

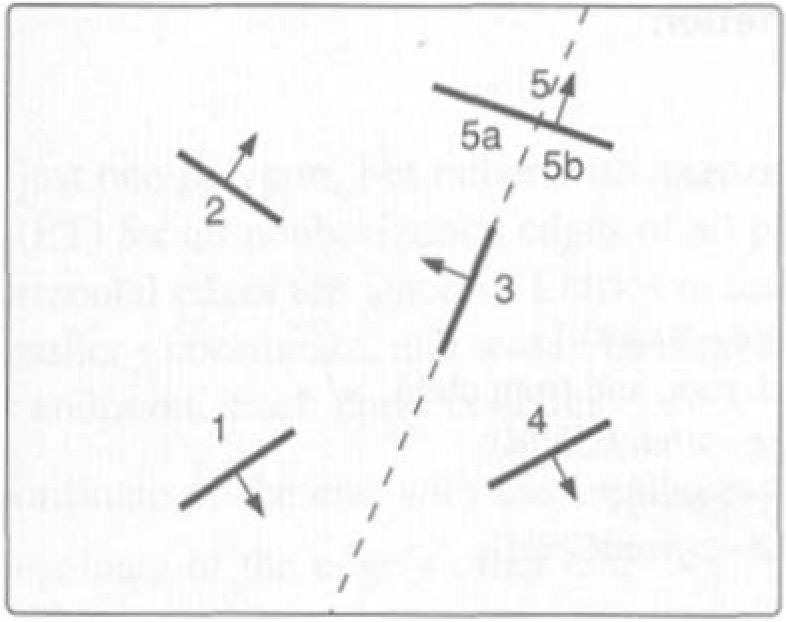


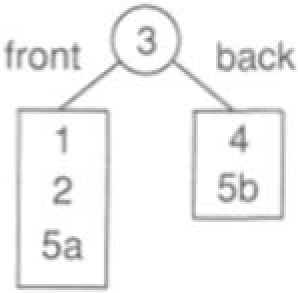
- **B**inary **S**pace **P**artitioning: Divide space into visibility regions
  - In 2-D, boundaries are lines
  - In 3-D, boundaries are planes
- Basic idea: "spatial sorting" keeps track of which side of lines/planes primitives are on
  - Objects on the same side as the viewer can be drawn on top of objects on the opposite side
  - Objects on one side cannot intersect objects on the other side

Building A 2D BSP Tree

- Pick oriented line segment (i.e., has a normal) from list as the root
- Rest of lines partitioned according to which side they are on
  - “Partitioning” line placed at root of subtree
  - Sets of lines on “front” side and “back” side correspond to left & right subtrees, respectively
  - If a line cross the partition line, split it
- Recurse on each child

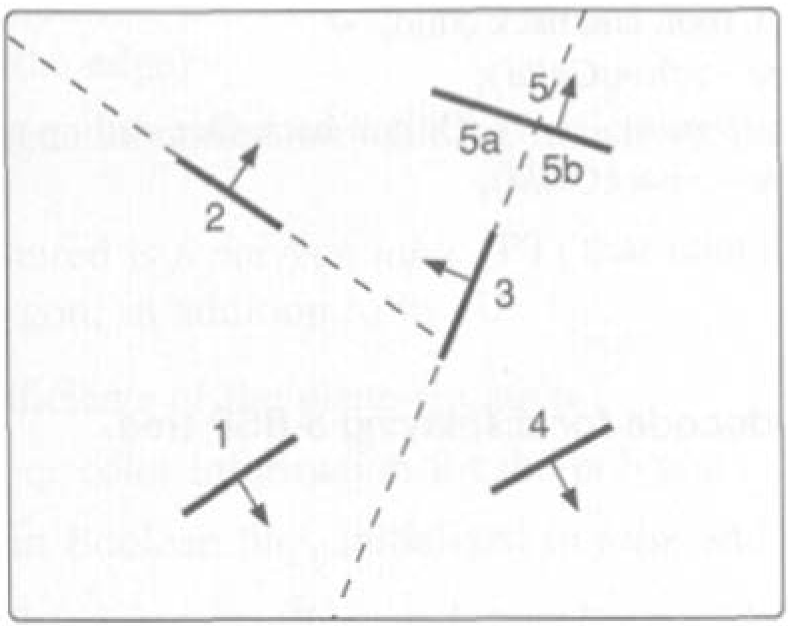
BSP Tree: Building Example

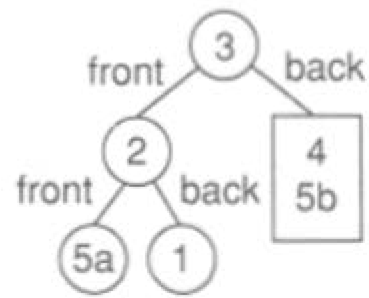




from Foley *et al.*

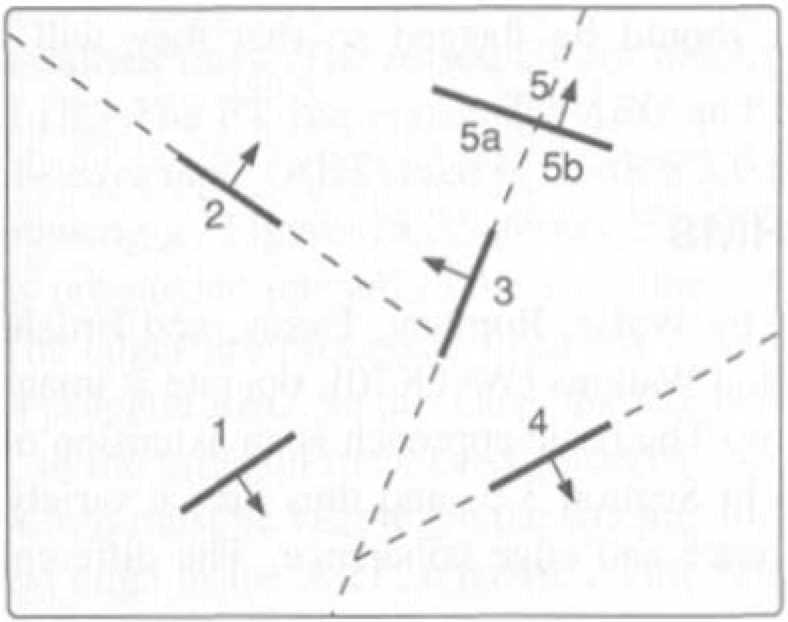
BSP Tree: Building Example

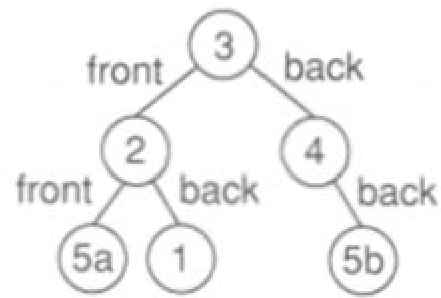




from Foley *et al.*

BSP Tree: Building Example





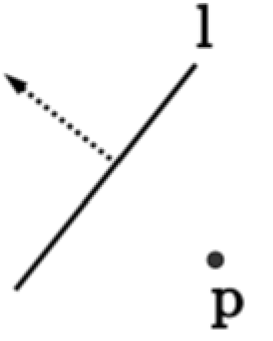
from Foley *et al.*

BSP Tree: Issues

- How to pick partition lines?
  - Every object must be drawn
  - Overall tree size should be as small as possible: minimize splitting
  - Procedure in practice:
    1. Randomly select a small number of candidate partitioning lines (e.g., 5-10 out of 1,000)
    2. Calculate number of lines that cross each candidate
    3. Use candidate with least crossing as the next partition line

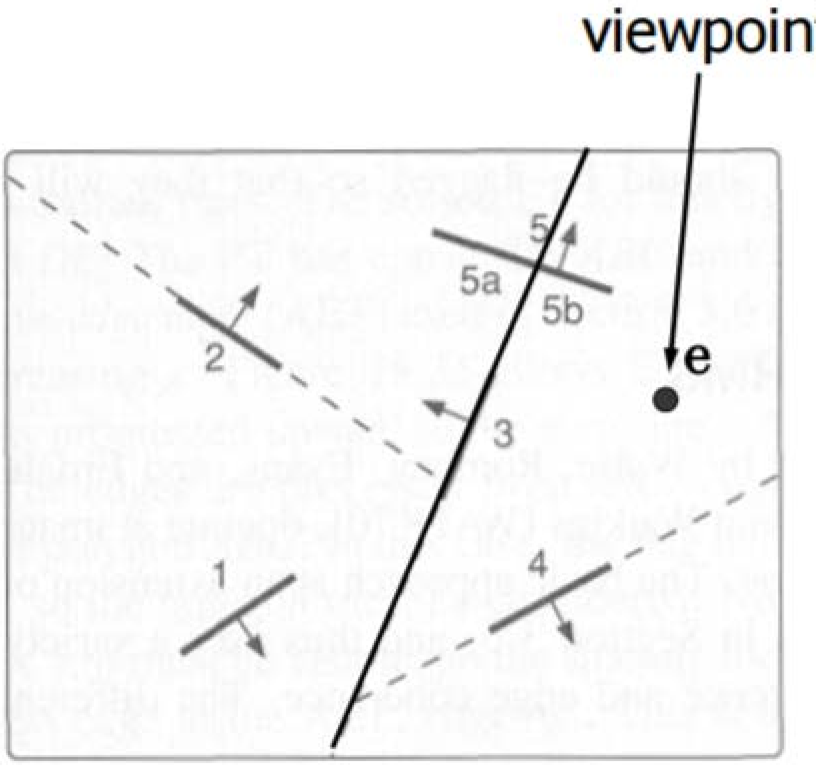
BSP Tree Traversal

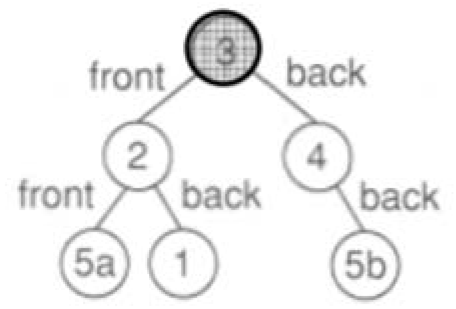
- Follow painter’s algorithm: draw objects from farthest to nearest
  - If view location is on front side of a partitioning line:
    - Lines on back side are farther
    - Lines on front side are nearer
  - If view location is on back side of a partitioning line:
    - Lines on front side are farther
    - Lines on back side are nearer



- How to determine which side of a partitioning line the viewpoint is on?
  - Line/Plane equation test

BSP Tree Traversal: Example

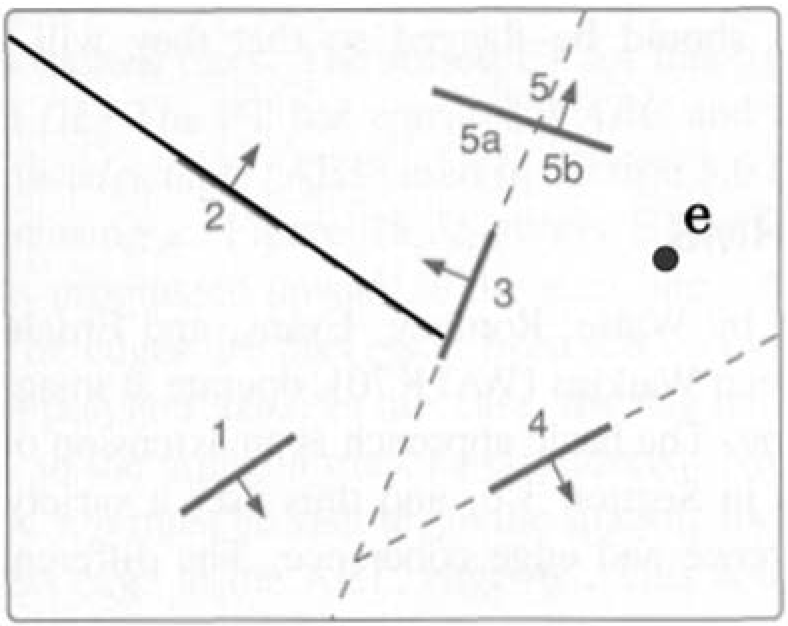


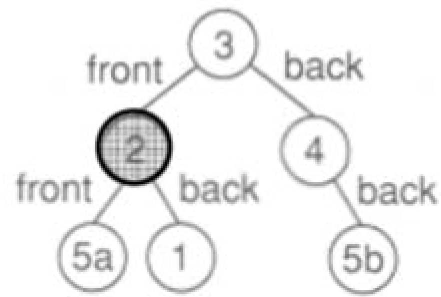


from Foley *et al.*

Behind root (node 3): Display everything in front of (left subtree = nodes 1, 2, 5a), then root (node 3), then everything behind (right subtree = nodes 4 and 5b)

BSP Tree Traversal: Example

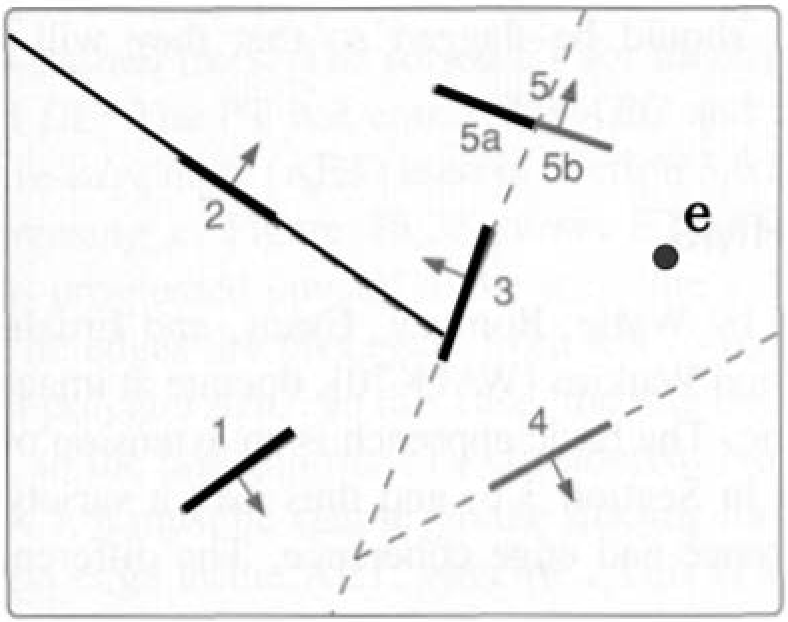


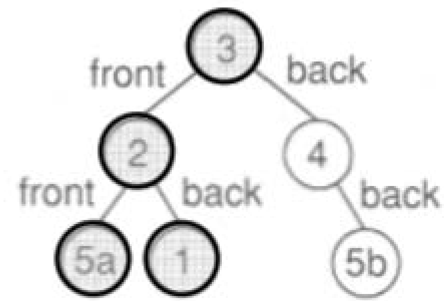


from Foley *et al.*

In front of root (node 2): Display everything behind (right subtree = node 1), then root (node 2), then everything in front of (left subtree = node 5a)

BSP Tree Traversal: Example

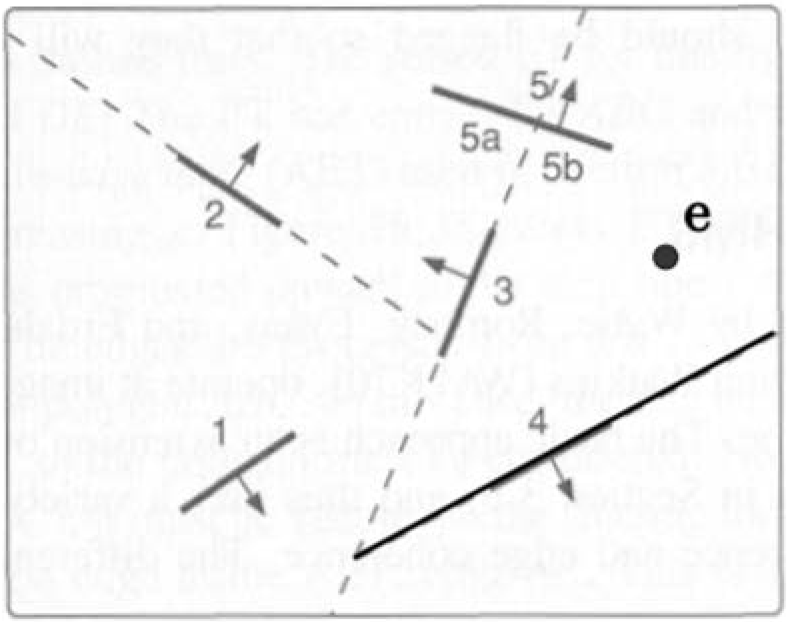


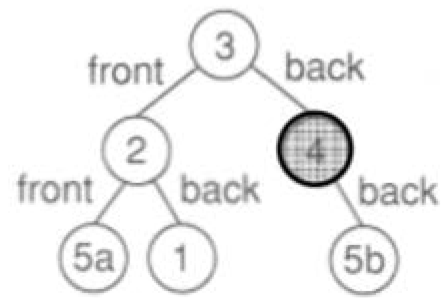


from Foley *et al.*

In front of root (node 2): Display everything behind (right subtree = node 1), then root (node 2), then everything in front of (left subtree = node 5a)

BSP Tree Traversal: Example

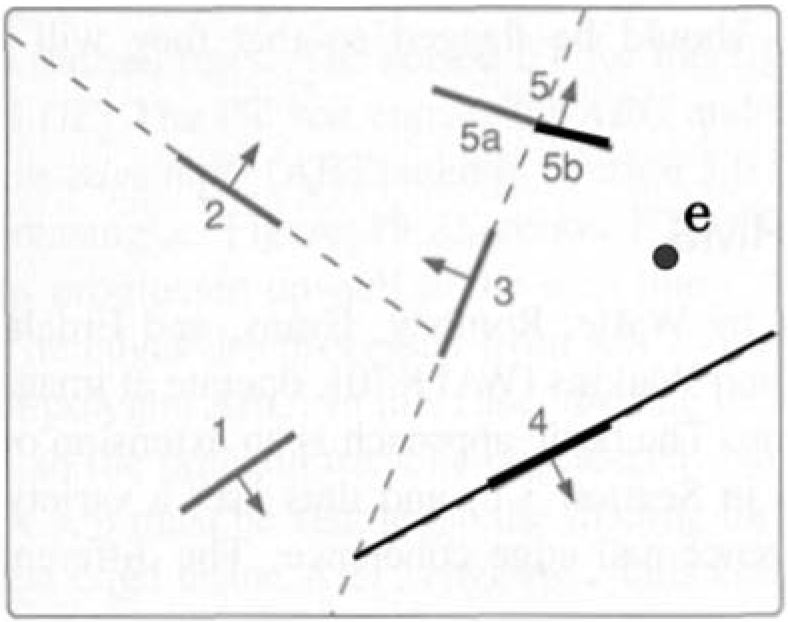


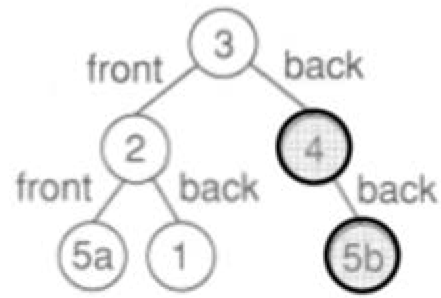


from Foley *et al.*

Behind root (node 4): Display everything in front of (left subtree = NULL), then root (node 4), then everything behind (right subtree = node 5b)

BSP Tree Traversal: Example

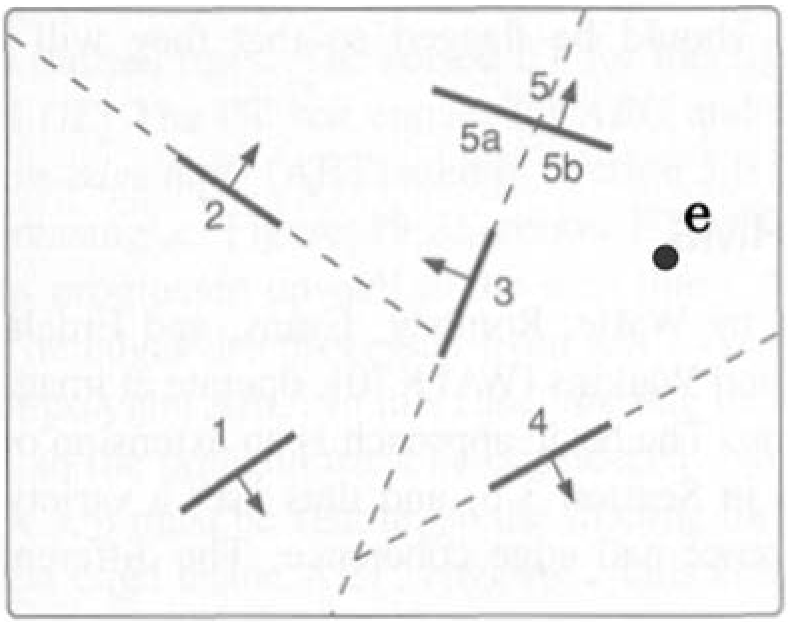


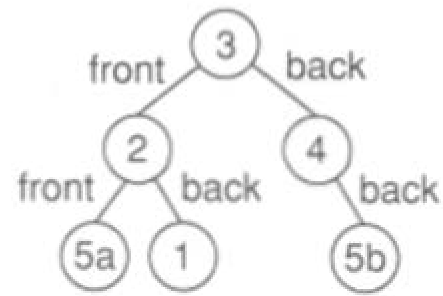


from Foley *et al.*

Behind root (node 4): Display everything in front of (left subtree = NULL), then root (node 4), then everything behind (right subtree = node 5b)

BSP Tree Traversal: Example





from Foley *et al.*

Final order: 1, 2, 5a, 3, 4, 5b

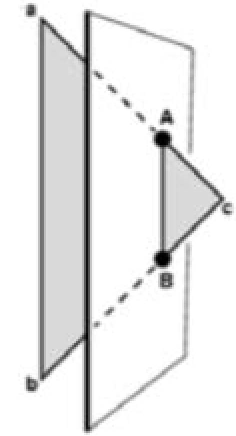
Every node is visited from back-to-front, so this is an O(n) operation (n is the number of primitives *after* splitting)

BSP Tree Traversal: Psuedocode

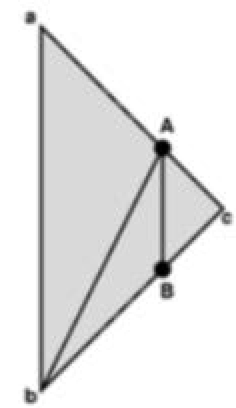
```
void draw_tree(Point eye, bspTree *tree)
{
      if (!tree)
            return;
      if (in_front(eye, tree)) {              // eye is on "front" side of divider
            draw_tree(eye, tree->back);
            draw_object(tree);
            draw_tree(eye, tree->front);
      }
      else if (in_back(eye, tree)) {          // eye is on "back" side of divider
            draw_tree(eye, tree->front);
            draw_object(tree);
            draw_tree(eye, tree->back);
      }
      else {                                  // eye is aligned with divider
            draw_tree(eye, tree->front);
            draw_tree(eye, tree->back);
     }
}
```

3D BSP Tree

- Analog of 2D method, but now we deal with 3D triangles and partitioning planes
- What's different from 2D case?
  - Parameterize partitioning plane from triangle
  - Use plane equation for side test
  - Line (triangle edges)-plane intersection instead of line-line intersection
  - Triangle splitting instead of line splitting



Triangle crossing partitioning plane



BSP Tree: Notes

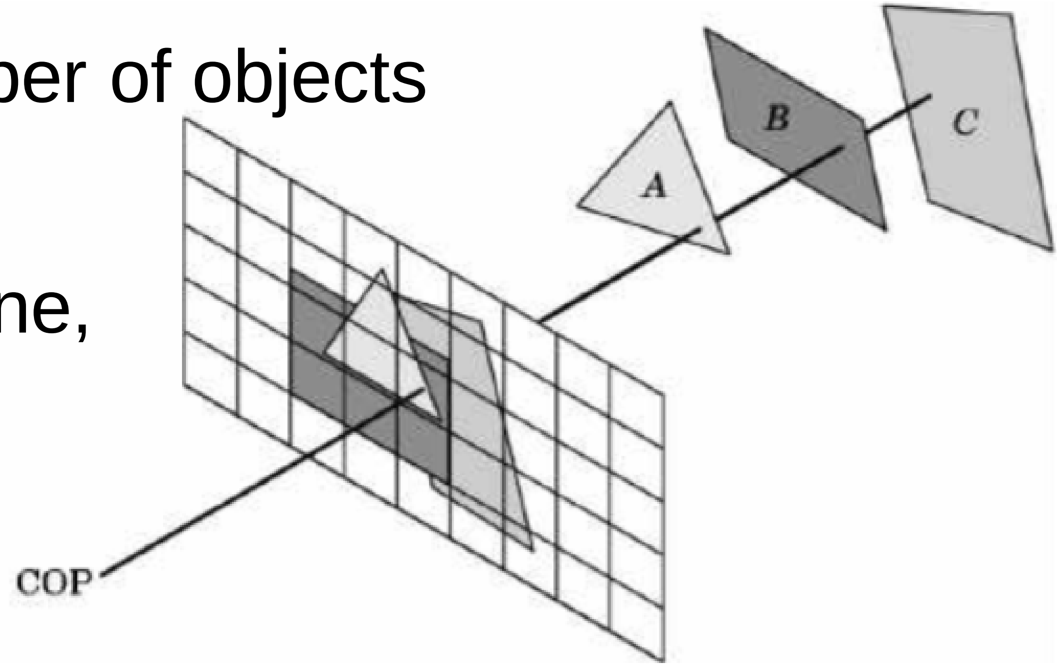
- Works best for moving viewpoint
  - Change viewpoint simply changes traversl order of the tree
- Works best for static scenes
  - Moving primitives can cross partitioning lines
  - Dynamic adjustment of tree possible, but slows things down

Pixel-Level Visibility

- So far, we've considered visibility at the level of primitives (lines/triangles)
- Now we will turn our attention to a class of algorithms that consider visibility at each pixel

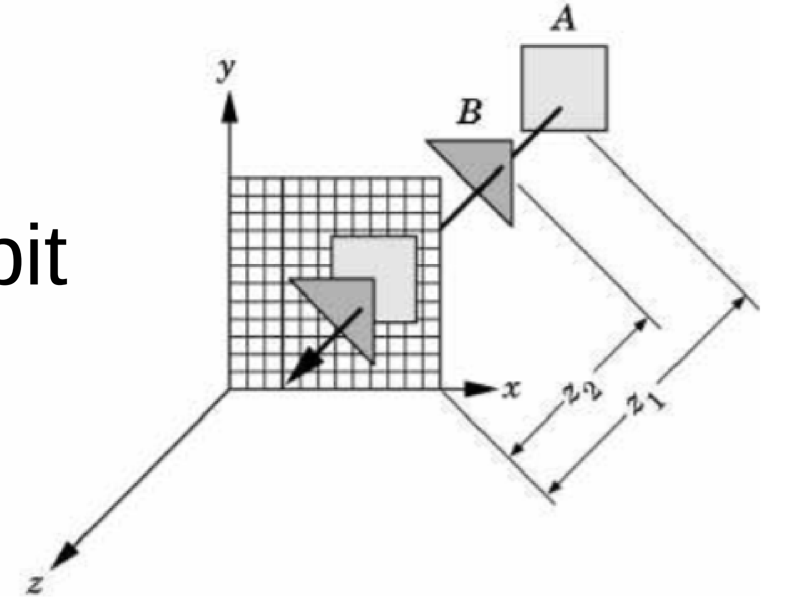
Ray Casting

- **Idea:** Cast a ray from the viewpoint through each pixel and intersect with objects to find the closest one
- Complexity: O(n) in worst case
  where n is the number of objects
- Objects could be
  polygon, sphere, cone,
  cylinder, etc.

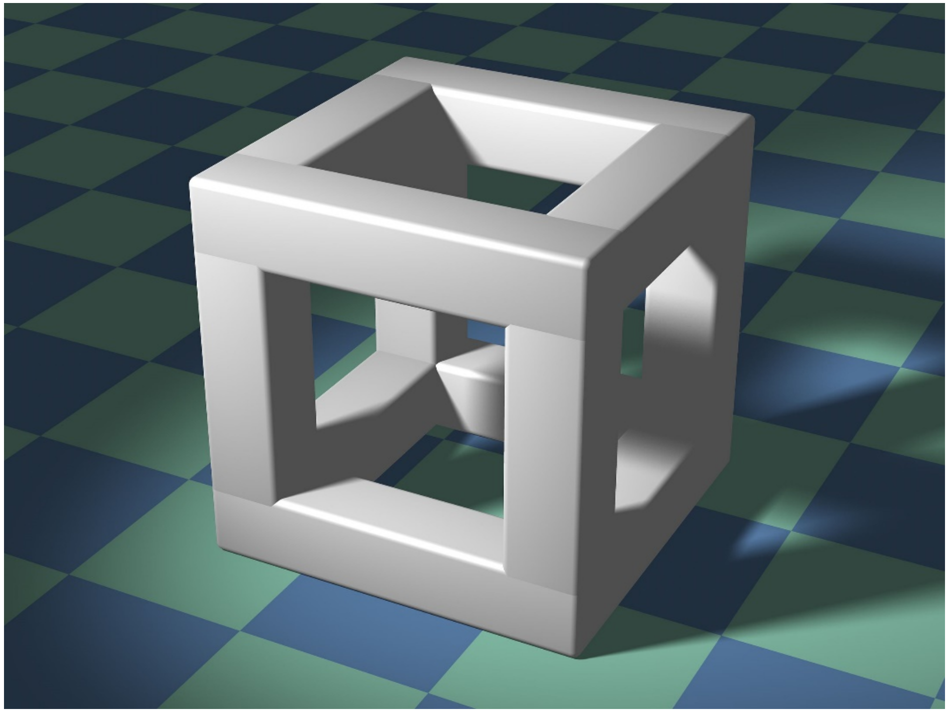


Z-Buffering

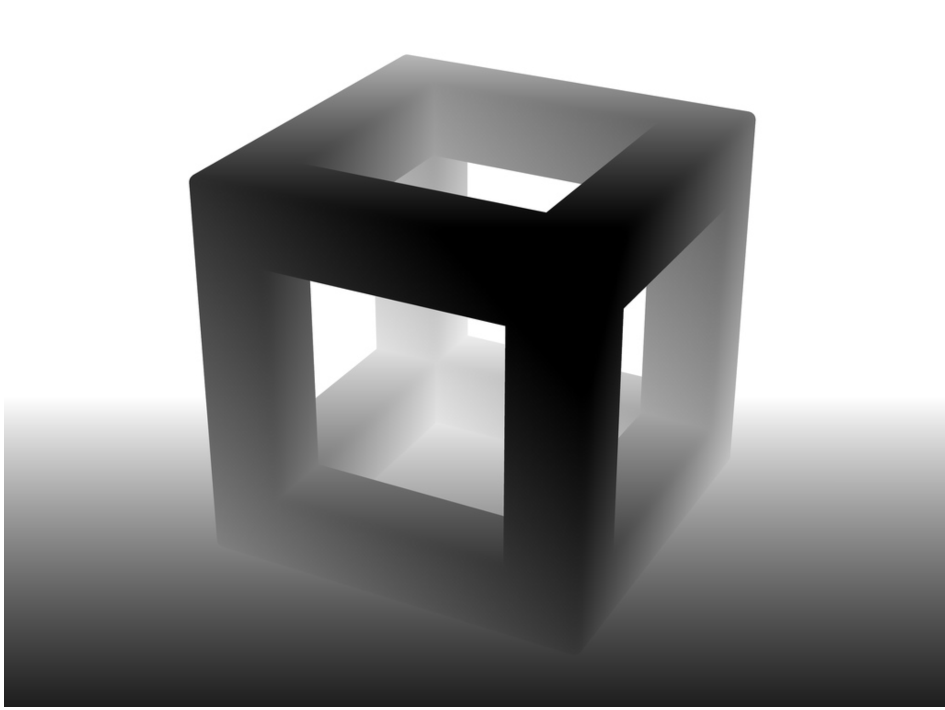
- Idea: Maintain an image-sized z-buffer with z value for each pixel
- What are z values?
  - z value is the distance from a scene point to the viewer (origin)
  - Related to depth values
- Typical z buffer size 24-bit
  - Same as color buffer



Z-Buffer: Example



A Simple Three Dimensional Scene



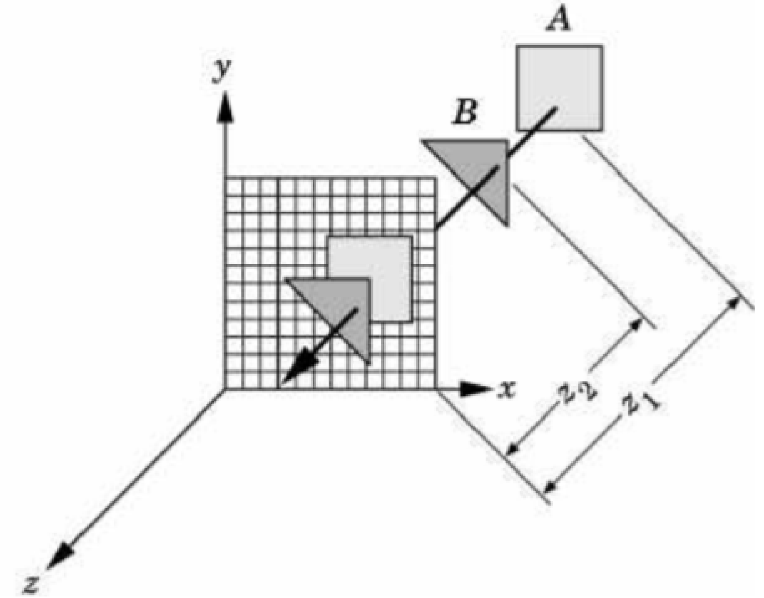
Z Buffer Representation

Z-Buffer: another example

Z-Buffer Algorithm

- Assumptions:
  - Each pixel has storage for a z value (z-buffer), in addition to RGB (frame buffer)
  - All objects are “scan-convertible” (typically are polygons, triangles, lines or points)
- Algorithm:

```
Initialize zbuf to maximal value
for each pixel (i,j) obtained
     by scan conversion
     if znew(i,j) < zbuf(i,j)
        zbuf(i,j) = znew(i,j);
        write pixel(i,j);
```



How to get z-buffer?

- Remember after camera projection, we have

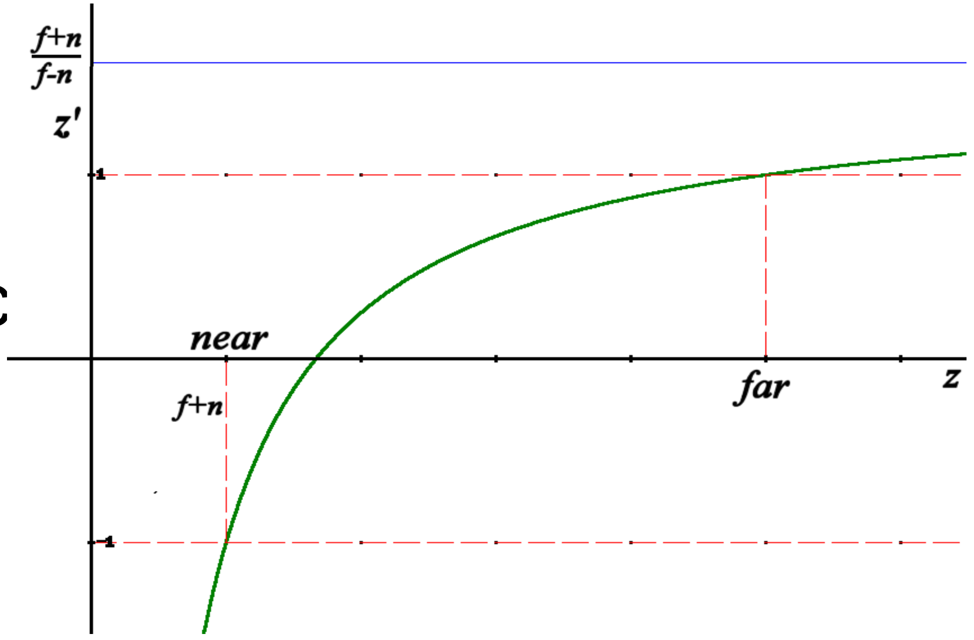
$$\begin{bmatrix} wx' \\ wy' \\ wz' \\ w \end{bmatrix} = \begin{bmatrix} \frac{2 \cdot near}{right - left} & 0 & \frac{-(right + left)}{right - left} & 0 \\ 0 & \frac{2 \cdot near}{top - bottom} & \frac{-(top + bottom)}{top - bottom} & 0 \\ 0 & 0 & \frac{far + near}{far - near} & \frac{-2 \cdot far \cdot near}{far - near} \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Computing Z

- We get the following expression for z from our projection matrix

$$z' = \frac{z \cdot (\,far - near) - 2 \cdot \,far \cdot near}{z \cdot (\,far - near)}$$

- The mapping of z is not linear
  - But still monotonic

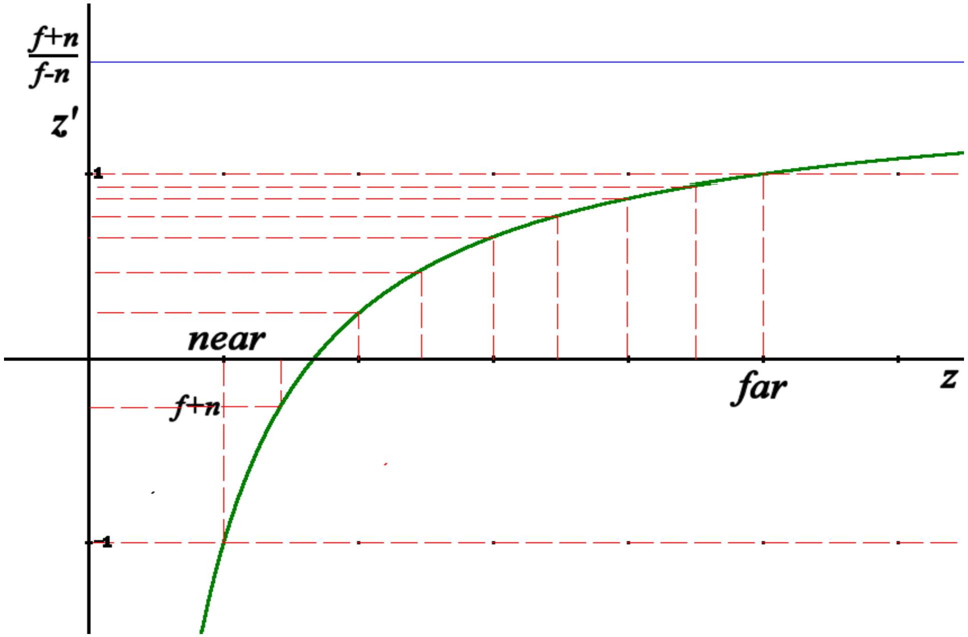


Computing Z

- What is the problem with non-linearity?
  - z values are non-uniformly quantized
  - The number of discrete discernable depths is greater closer to the near plane than near the far plane
- Cons:

  Objects closer to the viewer are displayed with higher precision
- Pros:

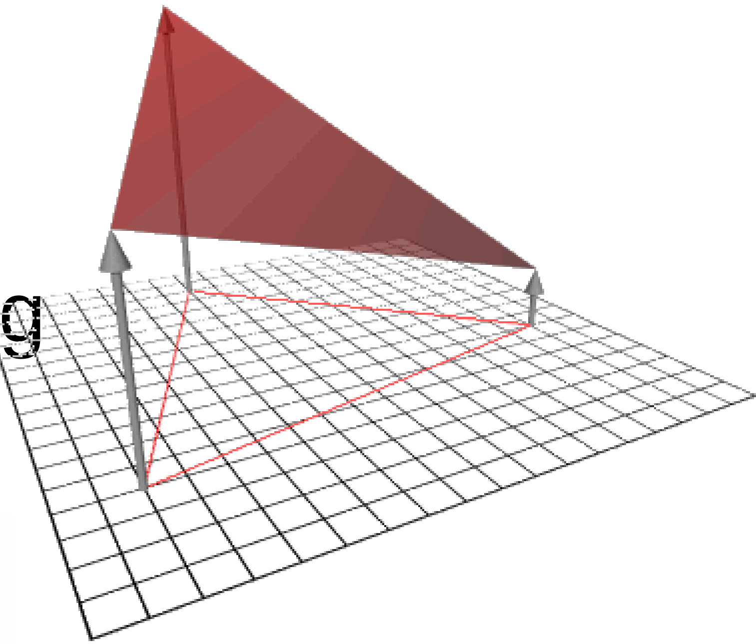
  This may result in far-away objects indiscernible



Interpolating Z

- Linear interpolating the interior z values from triangle vertices
- Plane Equation:

$$z = A_z x + B_z y + C_z$$

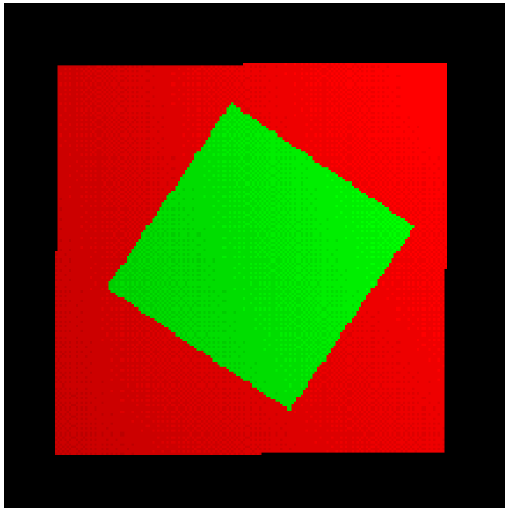


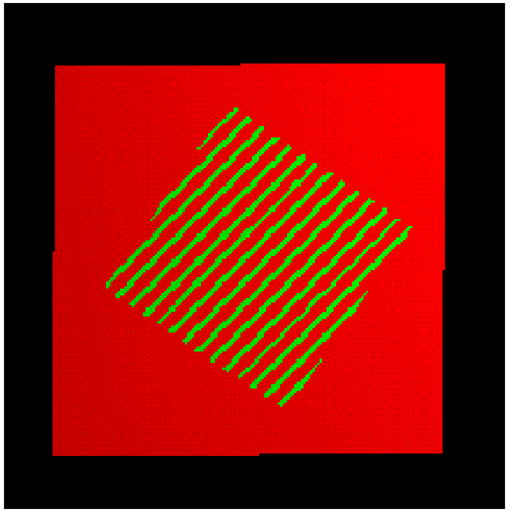
- Compute coefficients using edge parameters

$$\frac{1}{2 \cdot \text{area}} \begin{bmatrix} A_{12} & A_{02} & A_{01} \\ B_{12} & B_{02} & B_{01} \\ C_{12} & C_{02} & C_{01} \end{bmatrix} \begin{bmatrix} z_0 \\ z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} A_z \\ B_z \\ C_z \end{bmatrix}$$

Z Fighting

- Objects closer to each other than minimum z discrimination mean interpenetration/improper display is possible
  - Example: piece of paper on a desk top
  - Minimize with high-precision Z buffer, pushing near clip plane out as far as possible, and/or polygon offset (depth biasing)





Z Fighting Example

Z-buffering: Notes

- **Pros**
  - Interpolation of pixel values from vertex values is easy to do and a key idea in graphics
  - Nearly constant overhead
    - Expensive for simple scenes but good for complex ones
- **Cons**
  - Relatively late in pipeline
  - Extra storage (z-buffer)
  - Precision of depth buffer limits accuracy of object depth ordering for large scale scenes (i.e., nearest to farthest objects)
  - No perfect scheme for handling translucent objects

Z-Buffering in OpenGL

- Initial a window with z-buffer

  glutInitDisplayMode(GLUT_DEPTH)
- Enable per-pixel depth testing with **glEnable(GL_DEPTH_TEST)**
- Clear depth buffer by setting **glClear(GL_DEPTH_BUFFER_BIT)**