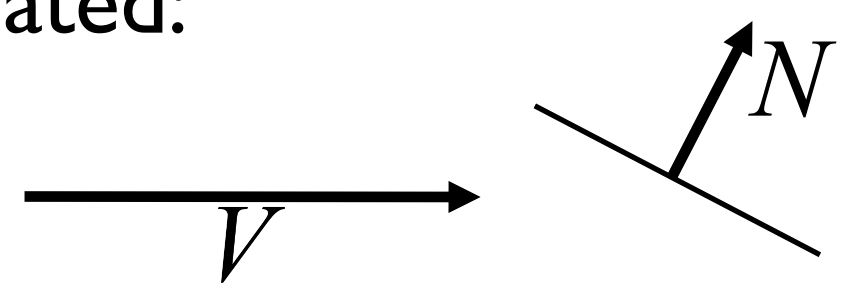
Hidden Surface Removal

Visible-Surface Algorithms

- Given a set of 3D objects and a viewing specification, determine which lines or surfaces of the objects should be visible
- *Image-precision* algorithms: determine what is visible at each pixel
- *Object-precision* algorithms: determine which parts of each object are visible

Back-Face Culling

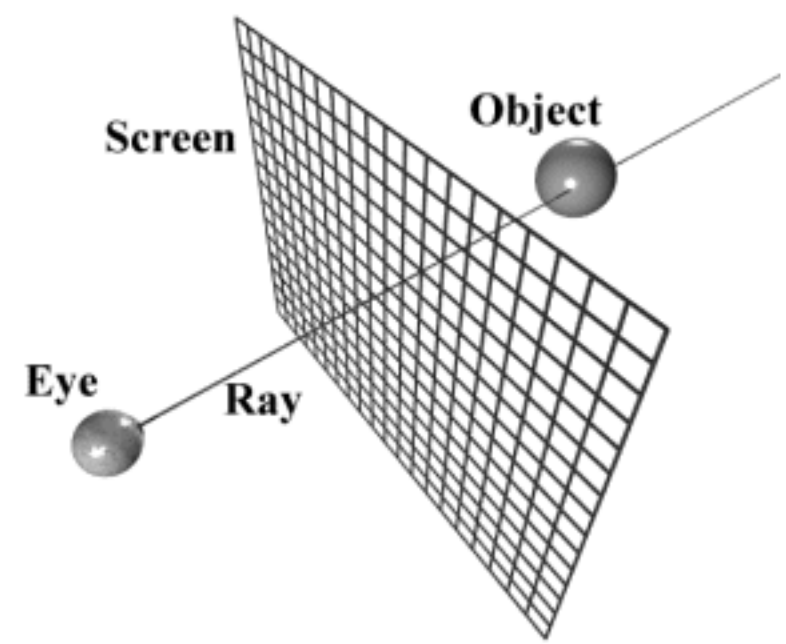
- If objects are represented by closed surfaces, polygons facing away from the viewer are always hidden and can be eliminated:



- Back-face test: $V \cdot N > 0$
- Back-face culling solves the hidden surface removal problem for a certain class of objects. What is this class?

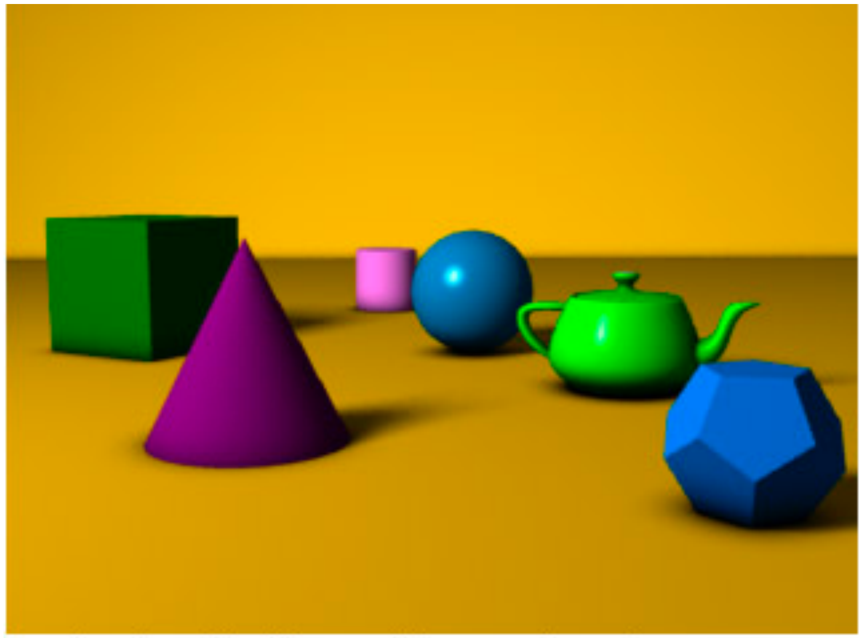
Ray Casting

- For each pixel, generate the line of sight (ray) from the center of projection passing through the pixel.
- To find the surface visible through the pixel:
  - Intersect ray with all surfaces in the scene
  - Return intersection closest to the center of projection
- Complexity: O(pN), where p is the number of pixels and N is the number of geometric primitives.



The Z-Buffer Algorithm (Catmull 1974)

- In addition to the frame buffer, keep a Z-buffer of the same dimensions containing the depth value of each pixel.
- Invariants:
    - Each Z-buffer pixel holds the depth (z-coordinate) of the nearest object seen through that pixel (so far).
    - Each frame buffer pixel holds the corresponding color


A simple three dimensional scene


Z-buffer representation

The Z-Buffer Algorithm (Catmull 1974)

- Initialize frame buffer to background color, and the Z-buffer to the depth of the far clipping plane.
- Scan-convert all polygons in an arbitrary order:
  - For each pixel (x,y) covered by the polygon, incrementally compute a color C, as well as a depth Z
  - If Z < Z-buffer(x,y) then FrameBuffer(x,y) := C;
  - Z-buffer(x,y) := Z

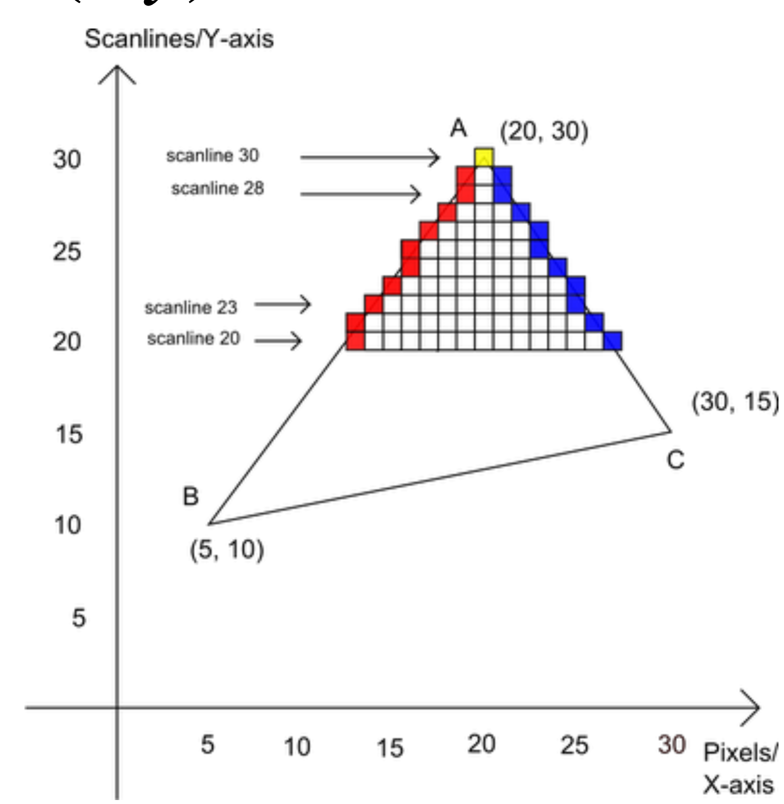
The Z-Buffer Algorithm (Catmull 1974)

- Let $Ax + By + Cz + D = 0$ be the plane equation of the scan-converted polygon.
- The depth at pixel (x,y) is: $z(x,y) = -(Ax + By + D)/C$
- Incremental depth computation: proceeding along a scan line the goal is to compute $z(x+1,y)$ from $z(x,y)$ as efficiently as possible:

$$z(x+1,y) = -\big(A(x+1) + By + D\big)/C$$

$$z(x+1,y) = -\big(Ax + By + D\big)/C - \frac{A}{C}$$

$$z(x+1,y) = z(x,y) - \frac{A}{C}$$



The Z-Buffer Algorithm

- Advantages:
  - Simple and easy to implement both in software and in hardware
  - Separately rendered images can be composited using their Z-buffers
- Disadvantages:
  - Requires extra memory (not so much of a problem anymore)
  - Finite depth precision can cause problems
  - Might spend a lot of time rendering polygons that are not visible in the image

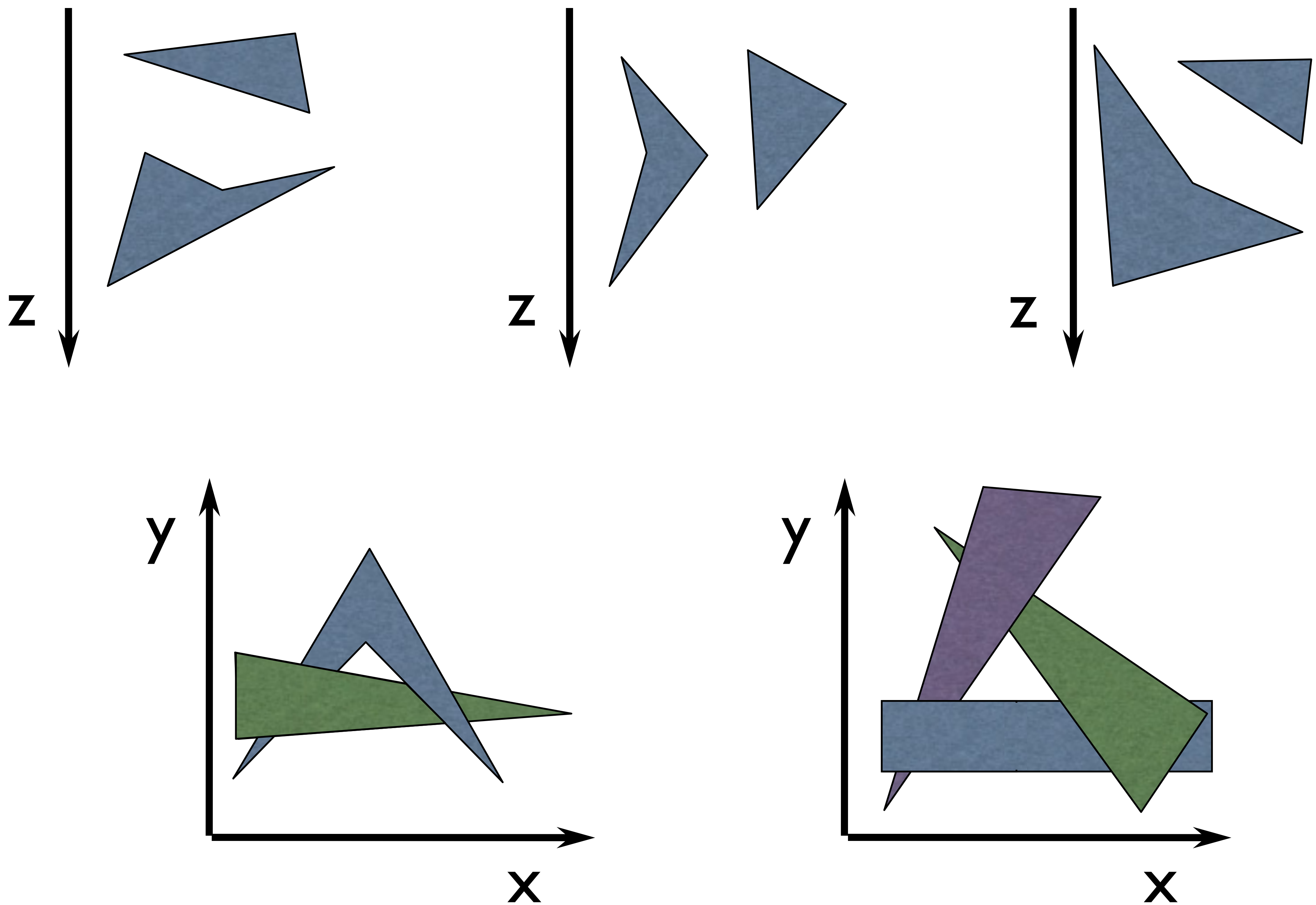
List-Priority Algorithms

- Determine an ordering for objects ensuring that a correct picture results if objects are drawn in that order.

- Example: *painter's algorithm*. If all of the polygons in the scene are sorted by their depth, drawing them _____ to _____ will give the correct result.

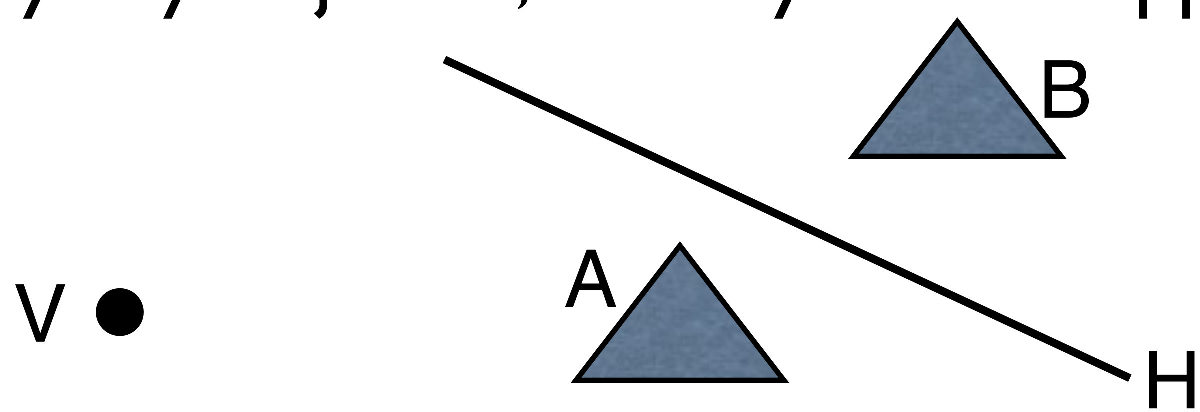
- Question: does a depth ordering always exist?

Depth Ordering



Planar Separation Principle

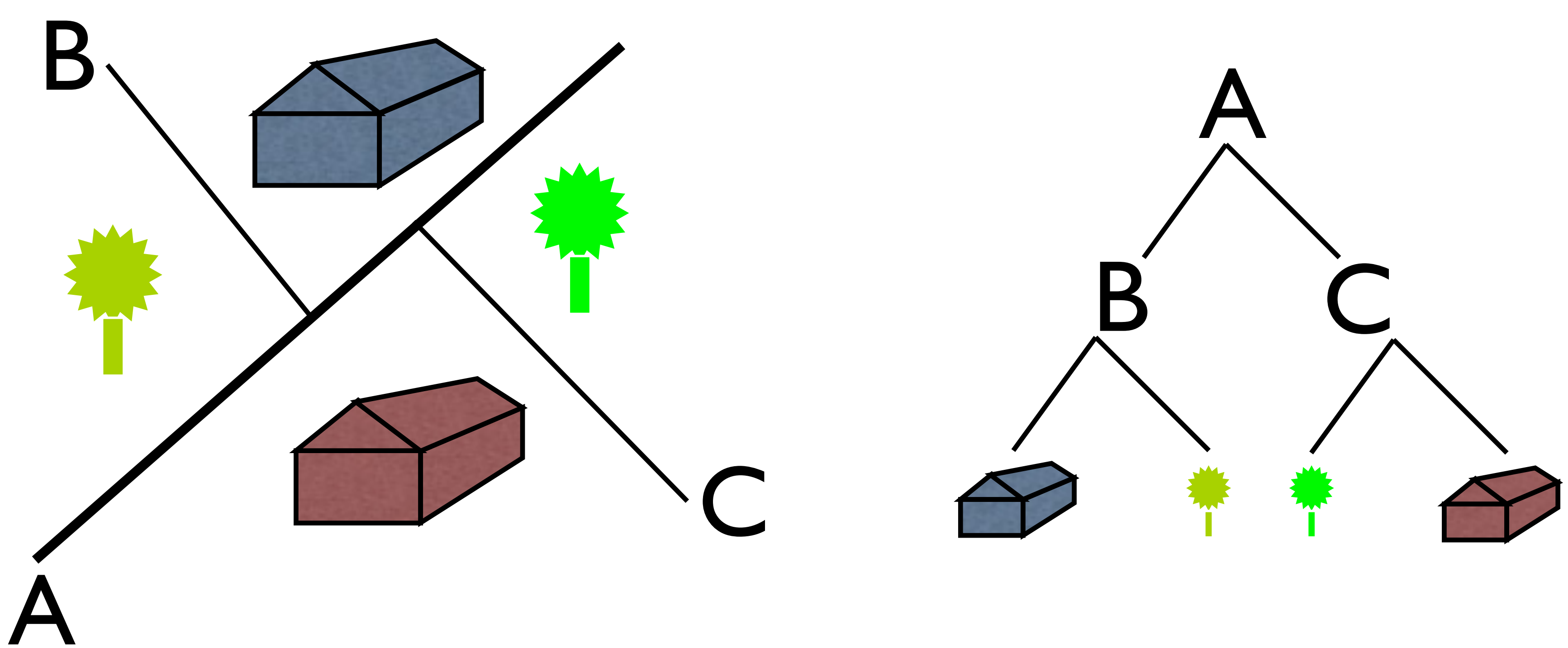
- Let V be a viewpoint, and H be a hyperplane. Any object A, entirely on the same side of H as V, cannot be occluded by any object B, entirely on the opposite side of H.



- If $H(V) > 0$ then no object A, $H(A) > 0$, can be occluded by B, where $H(B) < 0$.
- If $H(V) < 0$ then no object A, $H(A) < 0$, can be occluded by B, where $H(B) > 0$.

The BSP Tree

- BSP = Binary Space Partitioning
- Each tree node corresponds to a convex region of the entire space
- Each interior node is associated with a partitioning plane (splitting its region into two half-spaces)



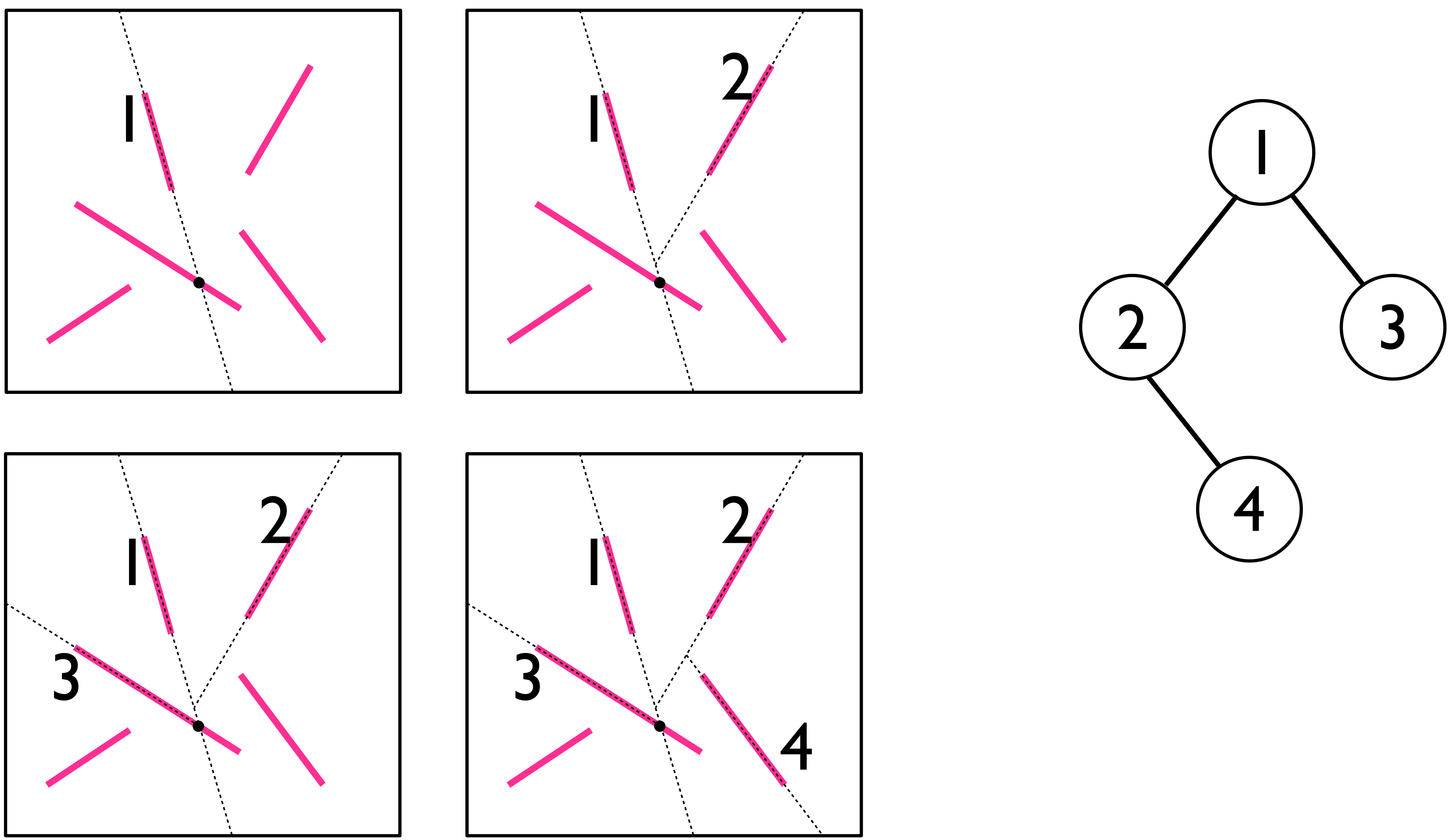
BSP-tree Traversal

- Back2Front(bspNode, viewpoint)
  - if (InFrontOf(bspNode.plane, viewpoint))
    - Back2Front(bspNode.backChild, viewpoint)
    - Back2Front(bspNode.frontChild, viewpoint)
  - else
    - Back2Front(bspNode.frontChild, viewpoint)
    - Back2Front(bspNode.backChild, viewpoint)

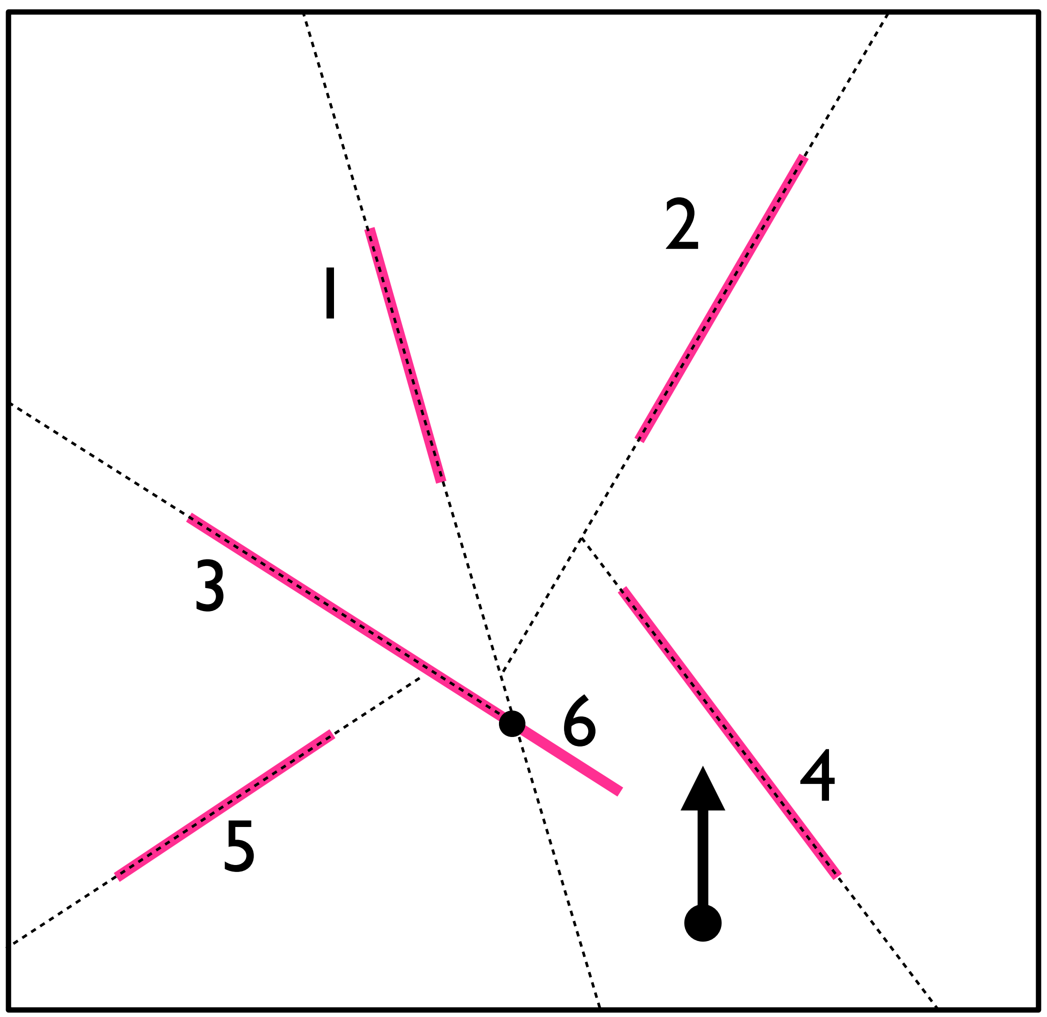
BSP-tree Painter's Algorithm

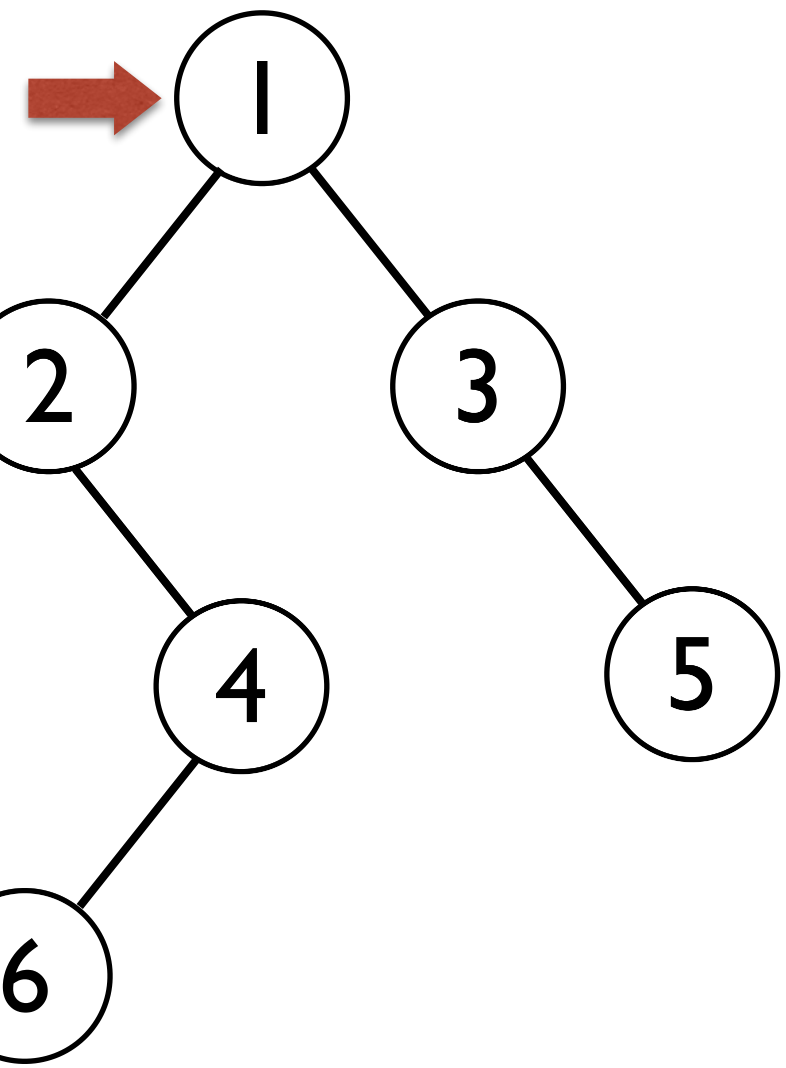
- Construct a BSP tree:
  - Pick a polygon, let its supporting plane be the root of the tree.
  - Create two lists of polygons: those in front, and those behind (splitting polygons as necessary)
  - recurse on the two lists to create the two sub-trees.
- Display:
  - Traverse the BSP tree back to front, drawing polygons in the order they are encountered in the traversal.

BSP-tree Construction



BSP-tree Traversal





BSP-tree Traversal