Computer Graphics

Hidden Surface Removal

Hidden-Surface Removal

- All primitives go through the pipeline unless they are specifically eliminated
- In real world, we can not see things that are behind other opaque objects
- Hidden surfaces should be removed or visible surfaces should be detected (final step of the pipeline)
- Visible surface detection *(surfaces in front of other surfaces)*

Categories of Approaches

2 algorithm categories:

- Object-space algorithms

  attempt to order the surfaces, back-to-front

  don't work well with the pipeline *(objects will go through the pipeline in an arbitrary order and we need to have all surfaces available to sort them)*

- Image-space algorithms

  work as part of the projection process

  determine the affecting object(s) for each pixel

Culling

- Faces that we see from behind will not be visible and can be eliminated *(culling)*
- Camera coordinate system

  Direction of camera: -z (V_{view})

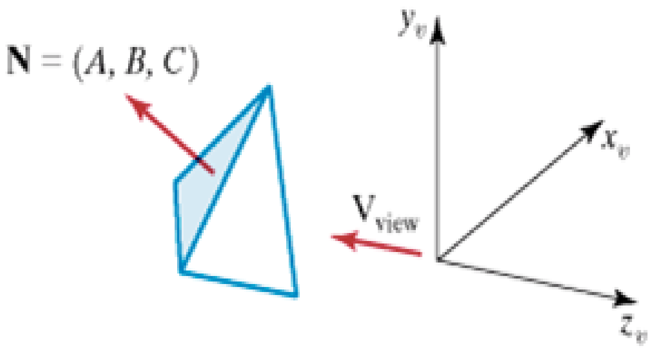
  N: surface normal

  C (z coordinate of N) is negative - back face

  N points away from the camera *(face must be culled)*

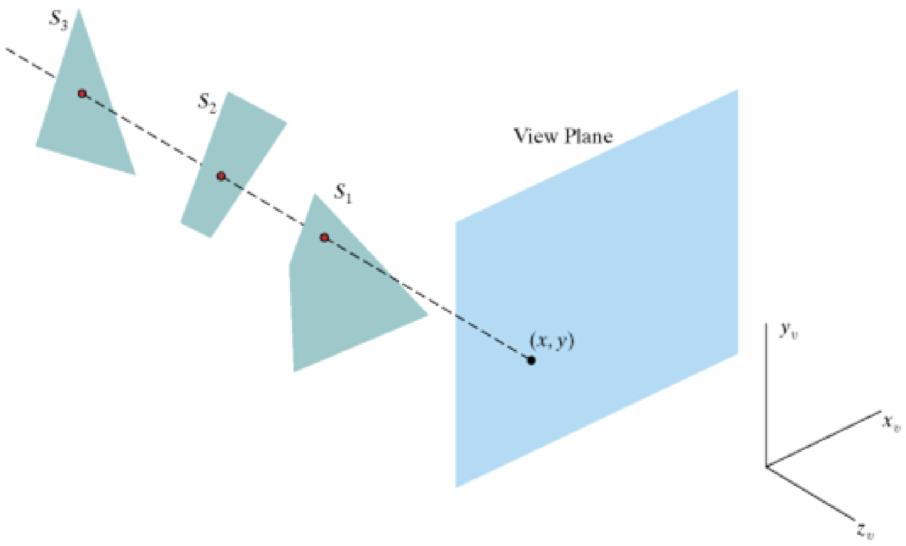
  Correct for scenes with single object

  Can not handle occlusion, need different solutions



Z-Buffer

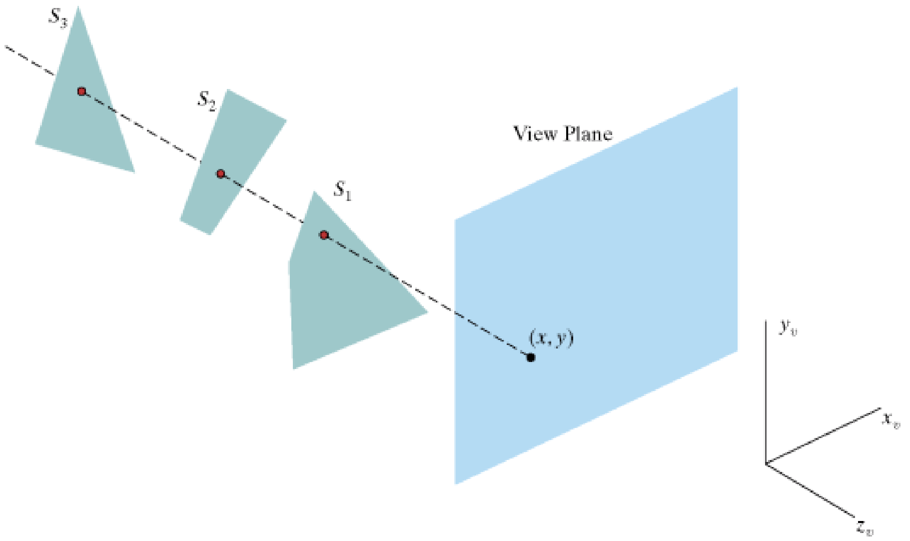
- An image-space algorithm
- For each pixel, draw the closest object
- Objects are in arbitrary order, keep the distance between the pixel and the currently drawn object
- Distance: z coordinate
- For all pixels: depth buffer or z-buffer
- Overlapping pixel position (x,y), S_1 is visible (has the smallest depth)



Z-Buffer

How it works?

- Initialize the z-buffer with a max value *(corresponds to far plane)*
- For each fragment, compare its depth with the current value in z-buffer
  - If greater, already showing a closer surface, ignore
  - Otherwise, it is closer, use it to set the color in color buffer and update the value in z-buffer



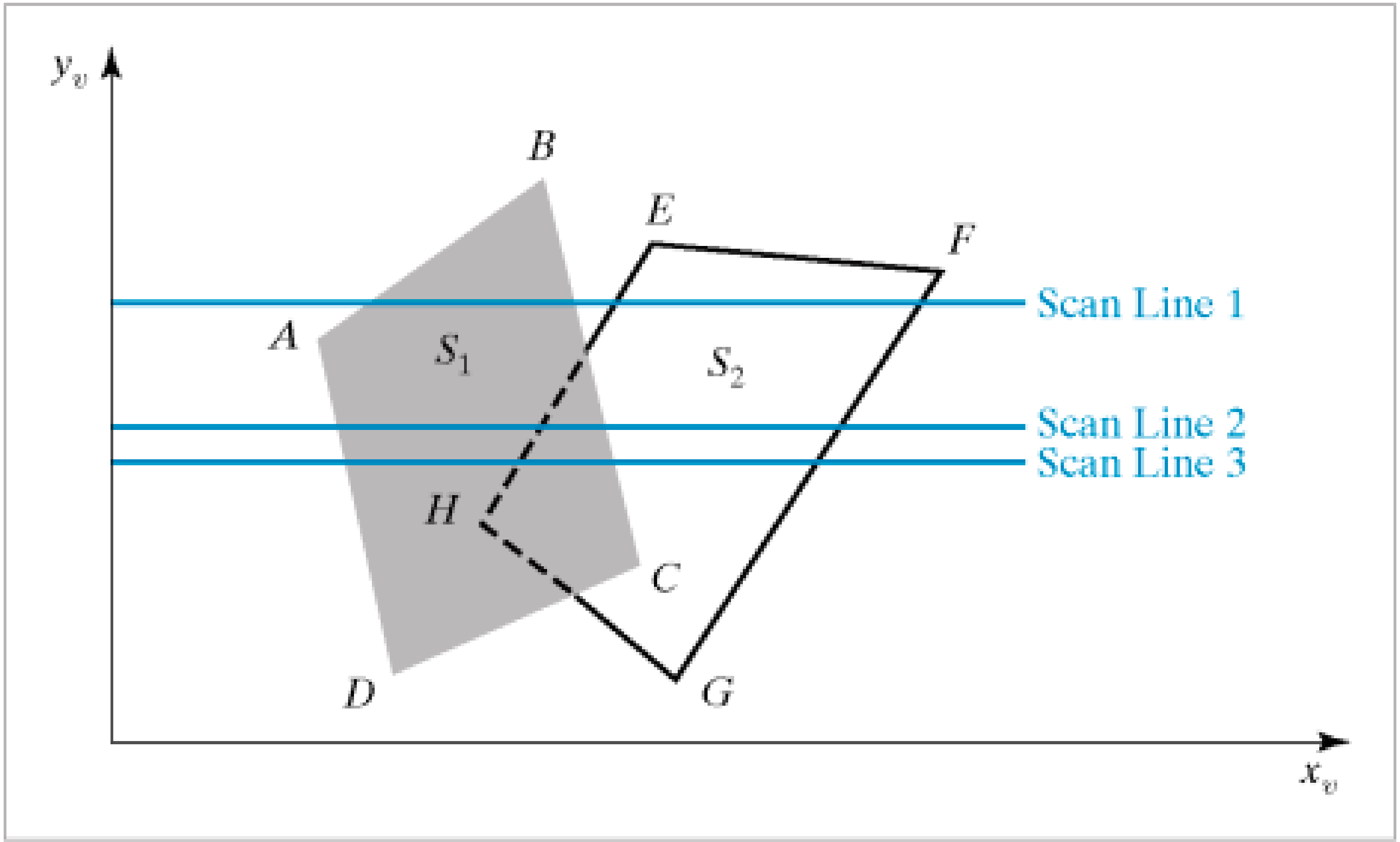
Z-Buffer

- Z-buffer is part of what makes a graphics card “3D“
- Advantage:
  - Computing the required depth values is simple
- Disadvantages:
  - All of the surfaces are evaluated *(fragments of surfaces)*
  - Over-renders - worthless for very large collections of polygons *(if we handle the back surface first, we also put the depth value into z-buffer and than we continue for the other surfaces)*
  - Depth quantization errors can be annoying *(depth values may be close)*
  - Can’t easily do transparency or filtering for anti-aliasing *(Requires keeping information about partially covered polygons)*

A-Buffer Method

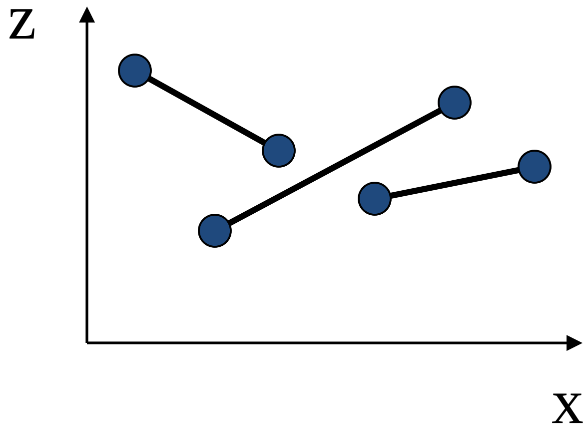
- An extension of the z-buffer idea, for transparency, accumulation buffer
  - developed at LucasFilm Studios
- An antialiasing, area averaging, visibility-detection method
- Accumulation-buffer, stores a variety of surface data in addition to depth values, can do more than z-buffer
  - a pixel color is computed as a combination of different surface colors
- Software implementation *(slow)*

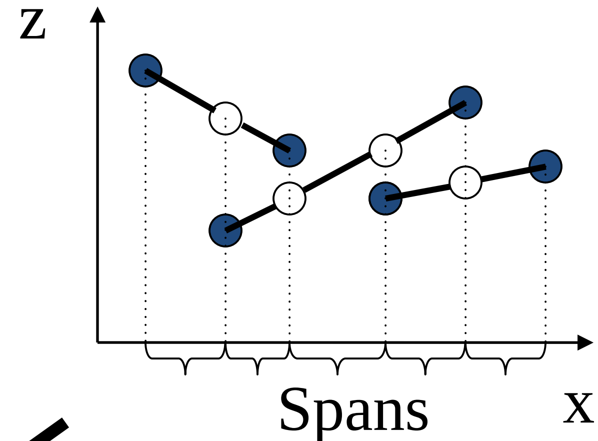
Scan Line Algorithm



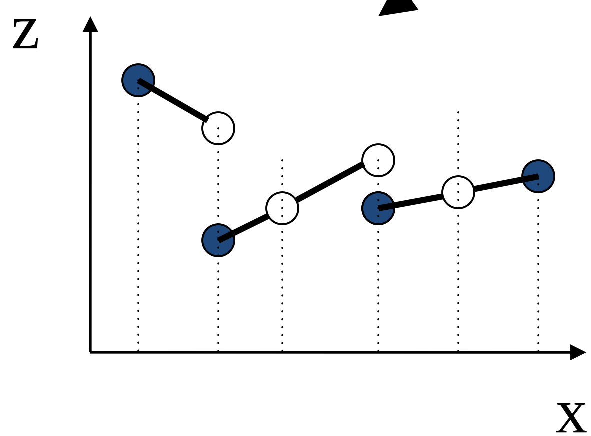
- Assume that polygons don't intersect
- Dashed lines indicate the boundaries of hidden surface sections

Scan Line Algorithm





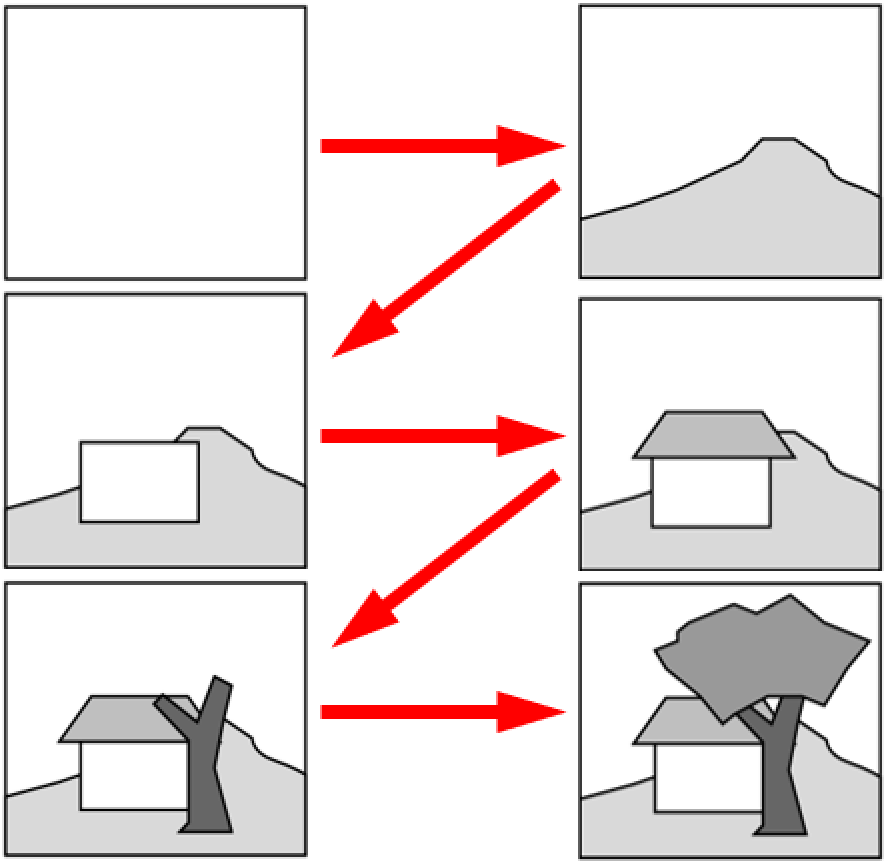
- 3 surfaces
- Looking in x, scan lines move on positive x
- non-intersection criteria is hard to meet



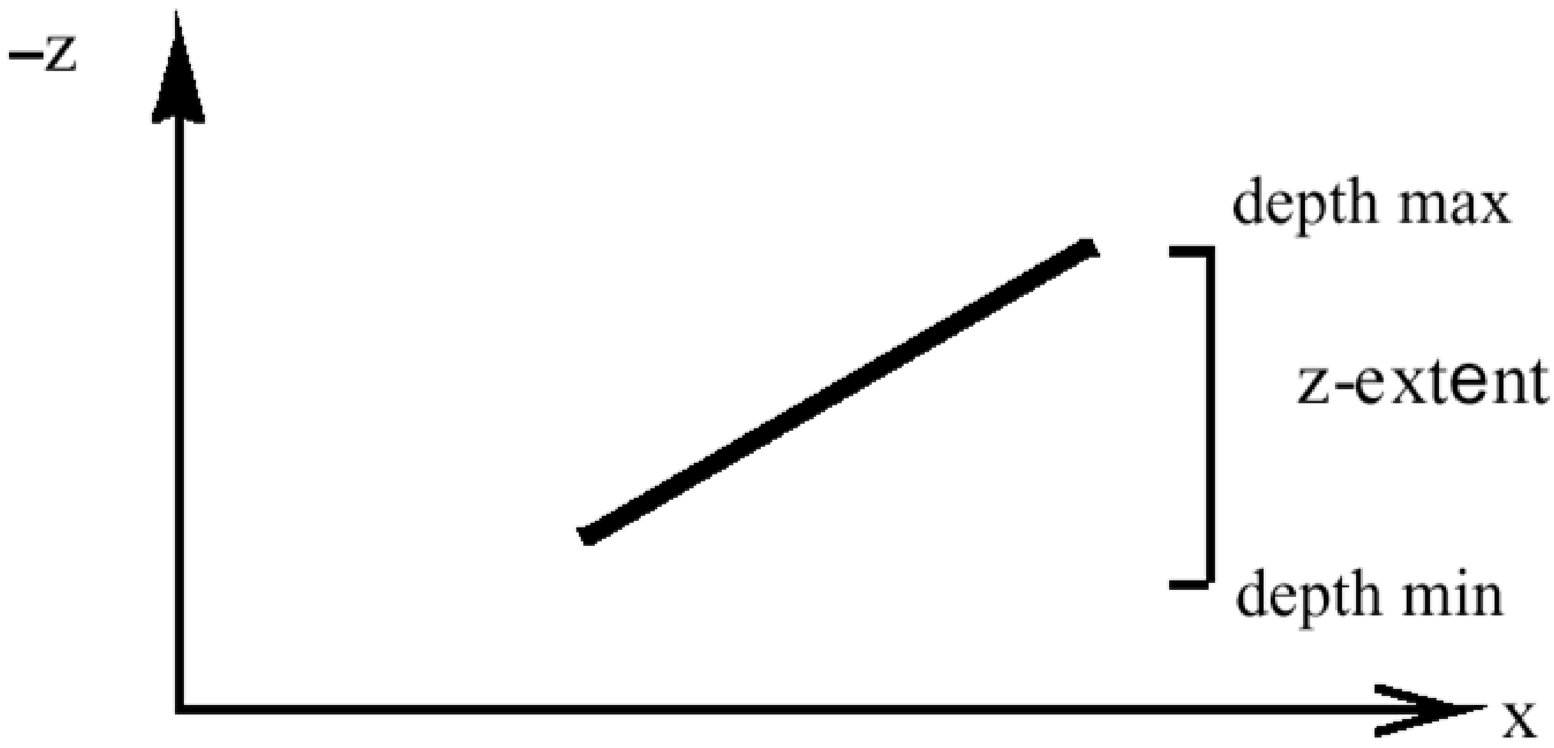
Where polygons overlap, draw front polygon

Depth Sorting

- The ''painter's algorithm'' *(an object space algorithm)*
  - Sort polygons according to depth
  - Resolve ambiguities
  - Render from back to front *(from distant to closer parts like a painter)*
- Very easy if z coordinates of polygons never overlapped
- Depth order ambiguities *happen (cause of overlaps*)

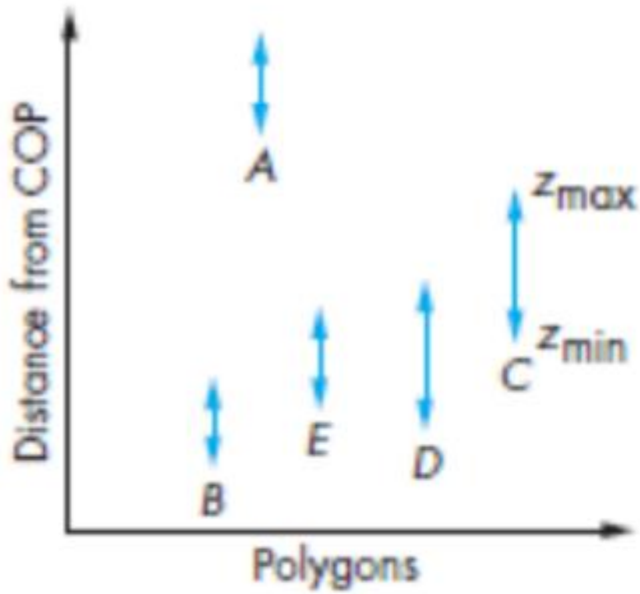


Depth Sorting



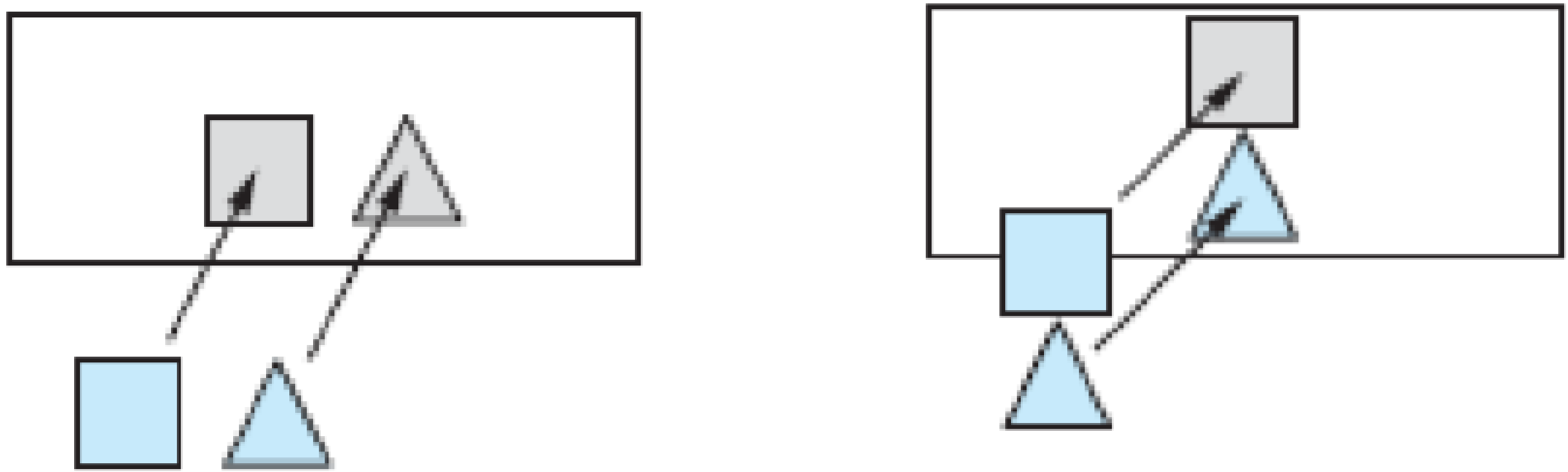
- First determine z-extent or range for each polygon *(range between the maximum and minimum z values)*

Depth Sorting



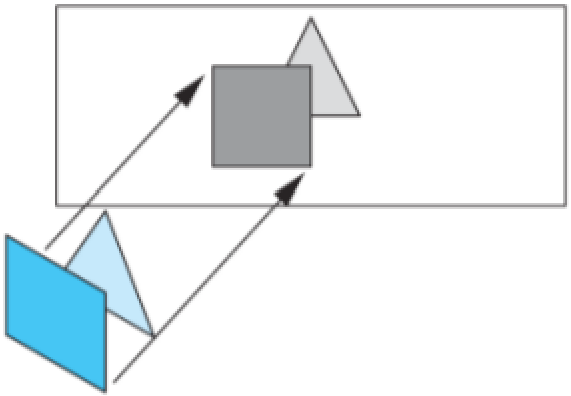
- Suppose that we have the z-extent of 5 polygons as shown
- Polygon A can be painted first *(it has the maximum distance from COP)*
- Can't determine the order for painting the other polygons
- Needs to run a number of increasingly more difficult tests in order to find the ordering

Depth Sorting



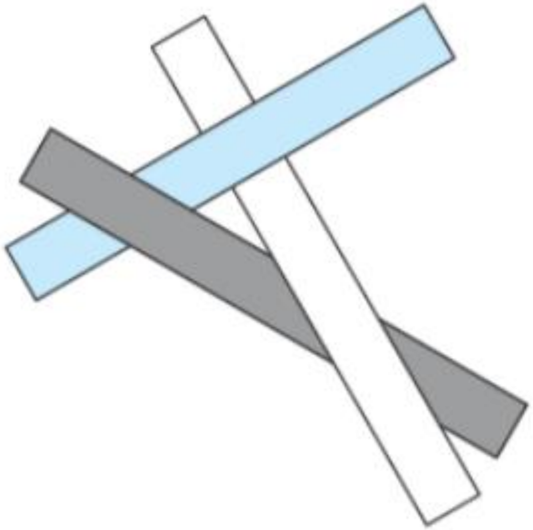
- check the x and y extents of the polygons (the simplest test)
- if either x or y extents do not overlap, neither polygon can obscure the other *(we can paint in any order)*
- On the left: test for overlap in the x extent may be used, on the right: for y extent may be used

Depth Sorting

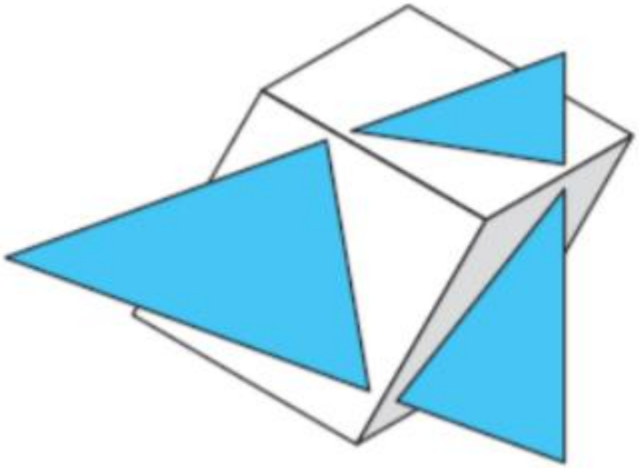


- if these tests fail, we can still determine the order of painting by testing if one polygon lies completely on one side of the other *(look for that if there is an intersection between polygons or not)*

Depth Sorting



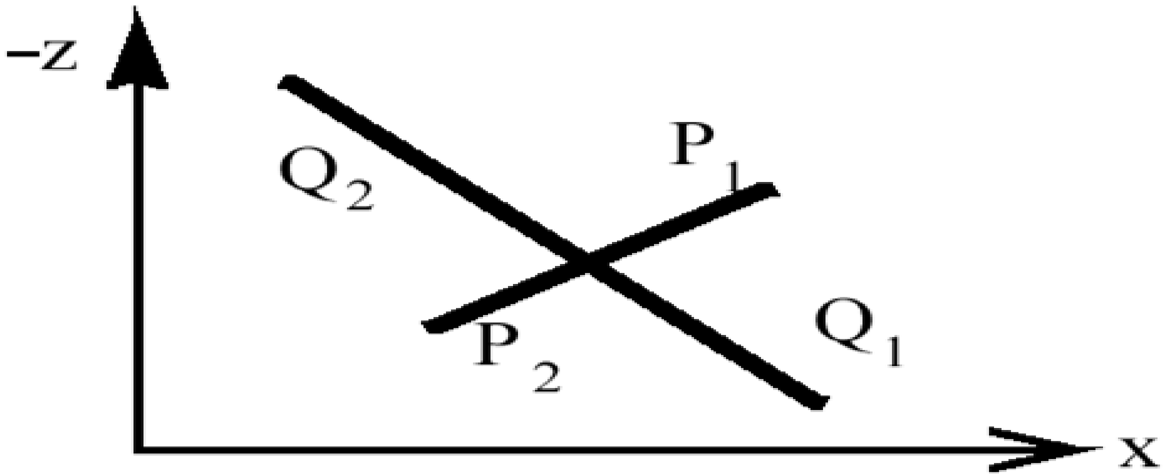
Cyclic overlap



Piercing polygons

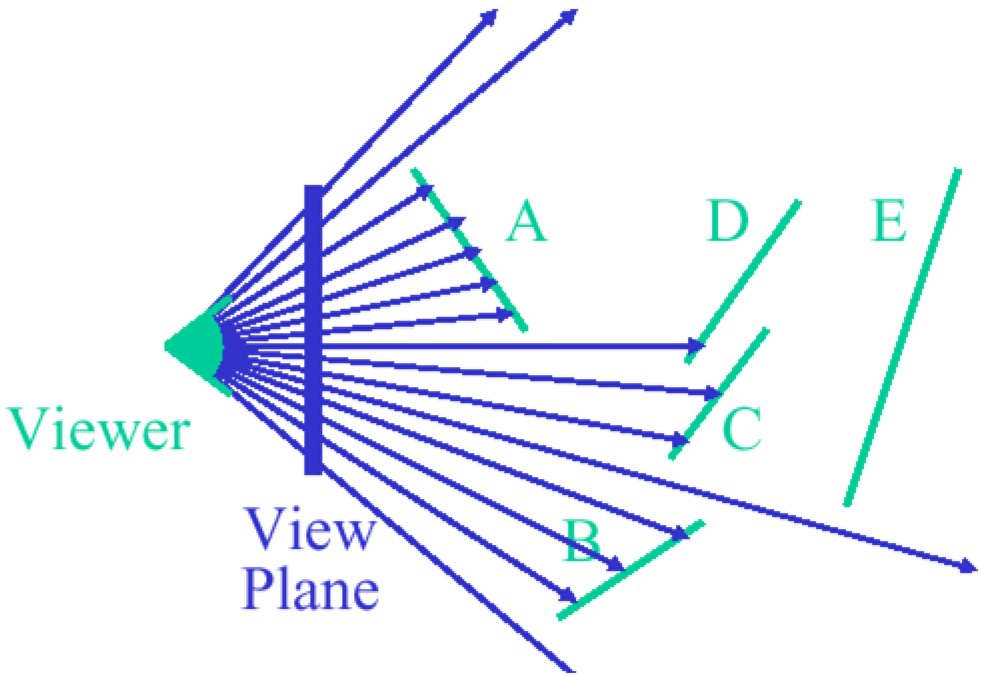
- All of these tests will fail in some cases:
  - Polygons that form a cyclic overlap
  - Polygons that pierce or intersect each other

Depth Sorting



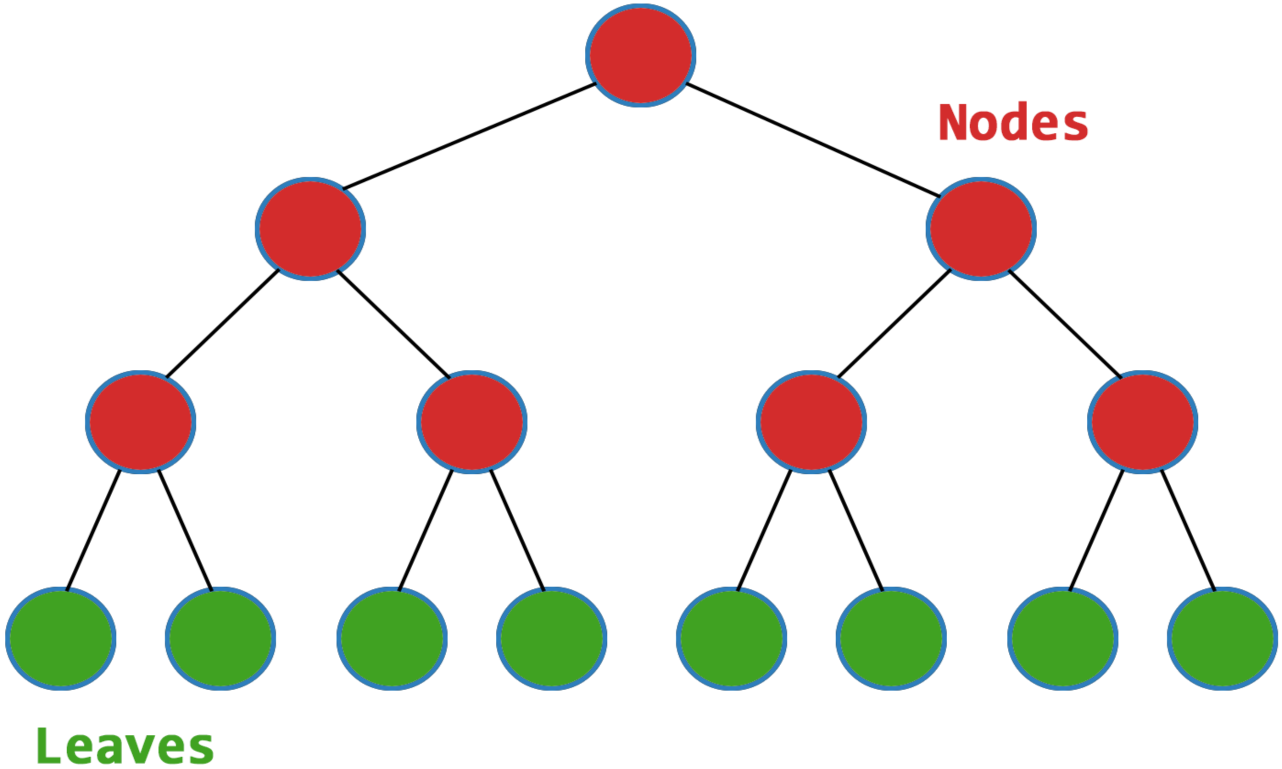
- Solution will be splitting polygons:
  we end up processing with order Q2, P1, P2 and Q1

Ray Casting



- For every pixel construct a ray from the camera *(to find front-most surface)*
- For every object in the scene, find intersection with the ray and keep if closest
- It can be extended to handle global illumination *(this time it is called ray tracing)*
- Casting is about what is visible at the sensor, tracing is about shading, color

BSP (Binary Space Partitioning)



- Idea is preprocessing the relative depth information of the scene in a tree for later display, base for other algorithms
- Efficient when objects don't change very often in the scene *(requires a lot of computation initially)*

BSP (Binary Space Partitioning)

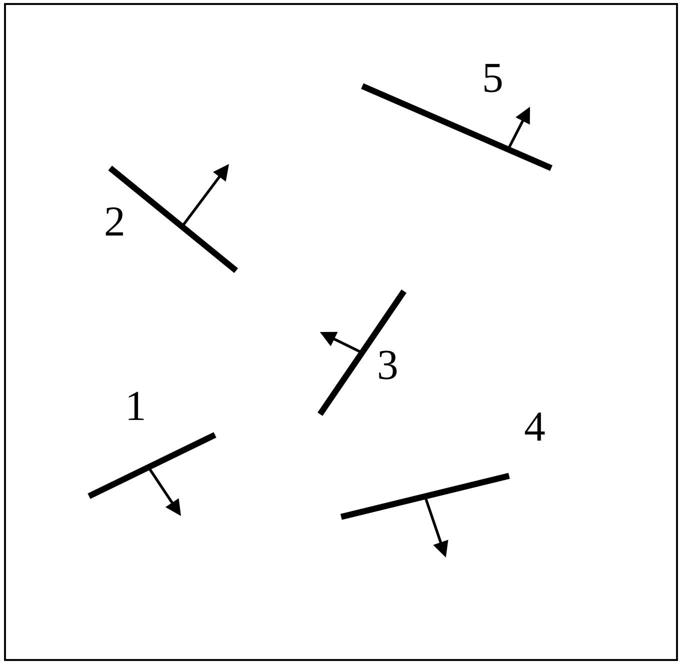
·Choose polygon arbitrarily

·Divide scene into front (relative to normal) and back half-spaces.

·Split any polygon lying on both sides.

·Choose a polygon from each side – split scene again.

·Recursively divide each side until each node contains only one polygon (leaves)



BSP (Binary Space Partitioning)

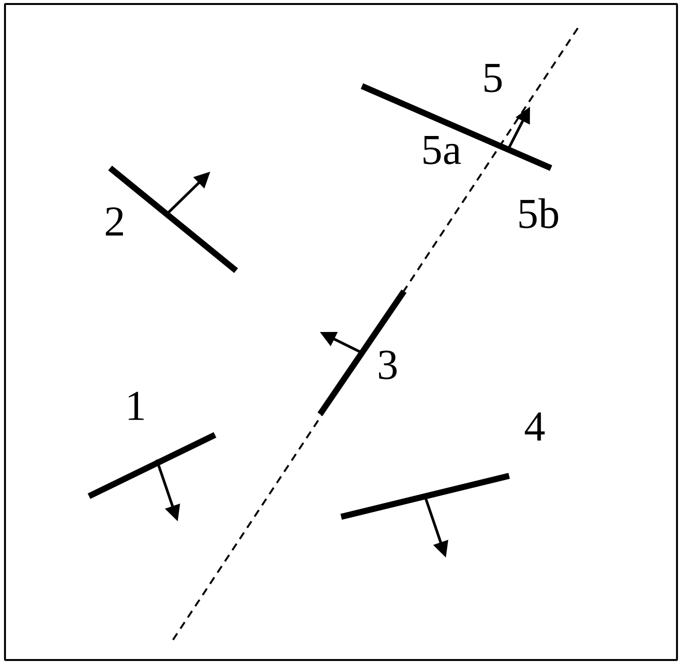
·**Choose polygon arbitrarily (#3)**

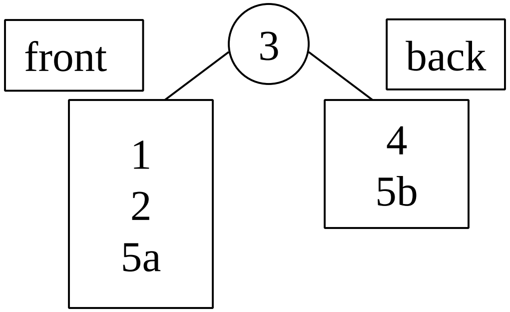
·**Divide scene into front (relative to normal) and back half-spaces (#1 and #2 are on the normal direction of #3 - front; #4 is back)**

·**Split any polygon lying on both sides (#5 must be splitted)**

·Choose a polygon from each side – split scene again

·Recursively divide each side until each node contains only one polygon (leaves)





BSP (Binary Space Partitioning)

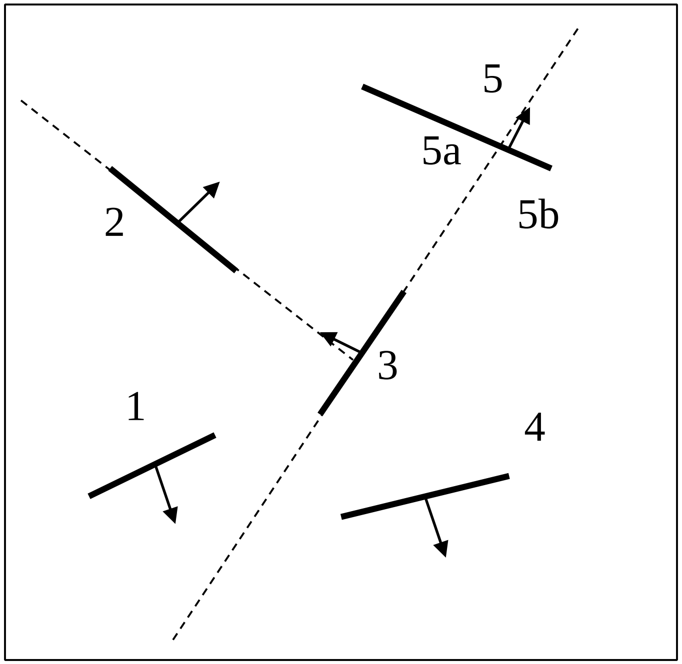
·Choose polygon arbitrarily

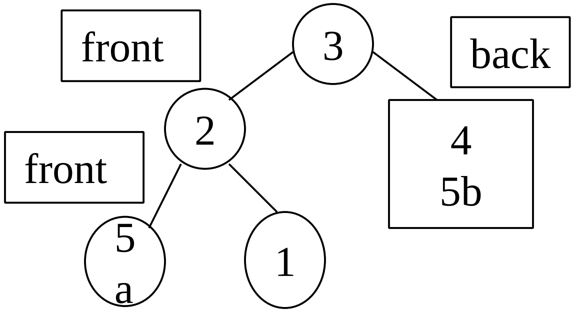
·Divide scene into front (relative to normal) and back half-spaces

·Split any polygon lying on both sides.

·Choose a polygon from each side – split scene again (#2)

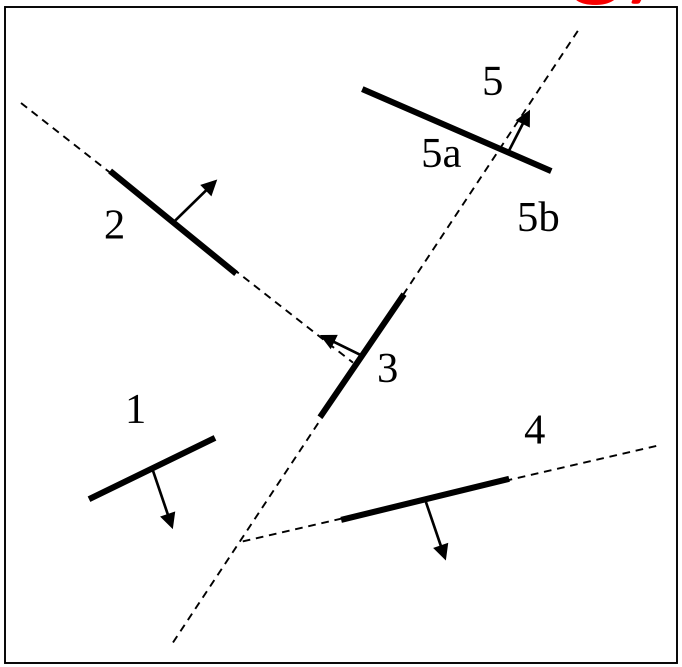
·Recursively divide each side until each node contains only one polygon.

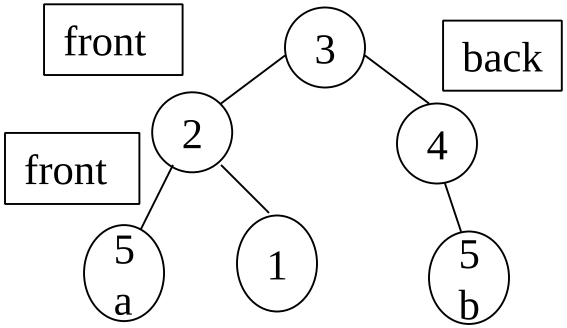




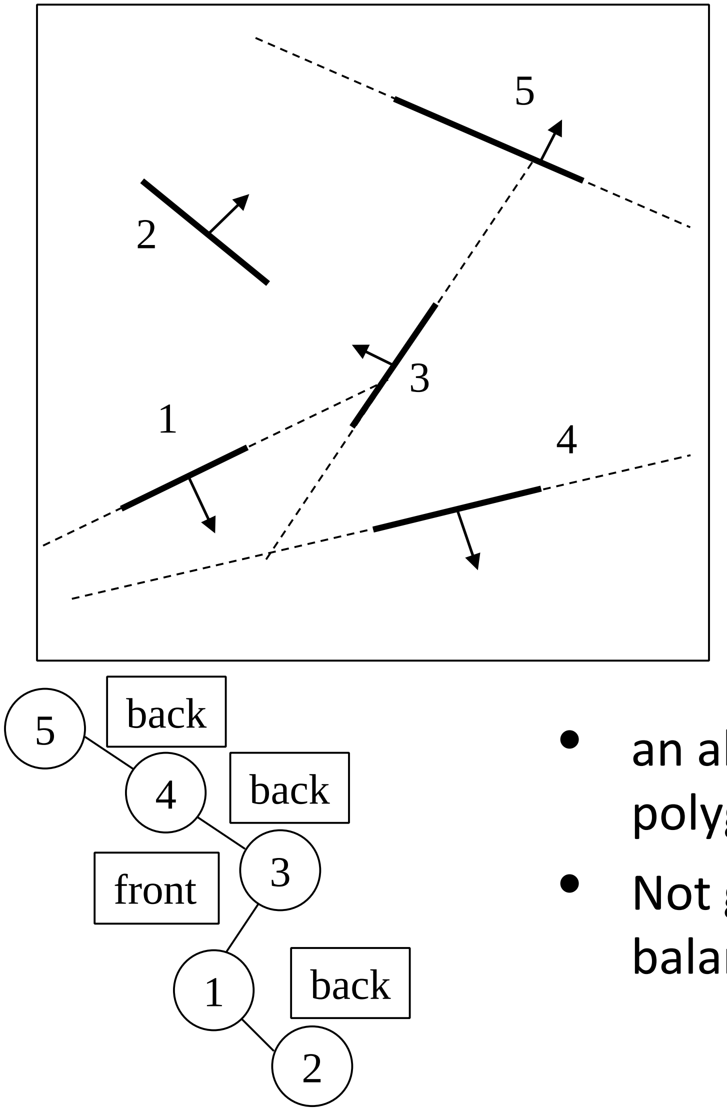
BSP (Binary Space Partitioning)

- Choose polygon arbitrarily
- Divide scene into front (relative to normal) and back half-spaces
- Split any polygon lying on both sides
- Choose a polygon from each side – split scene again
- **Recursively divide each side until each node contains only one polygon** **(#4 is the root)**



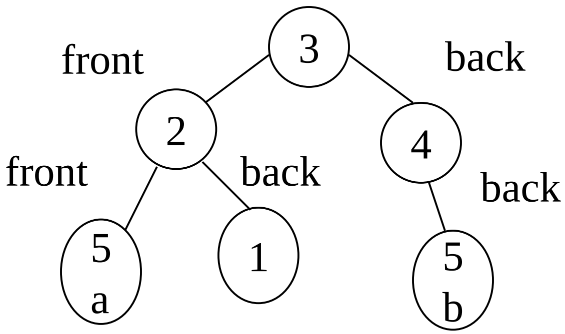


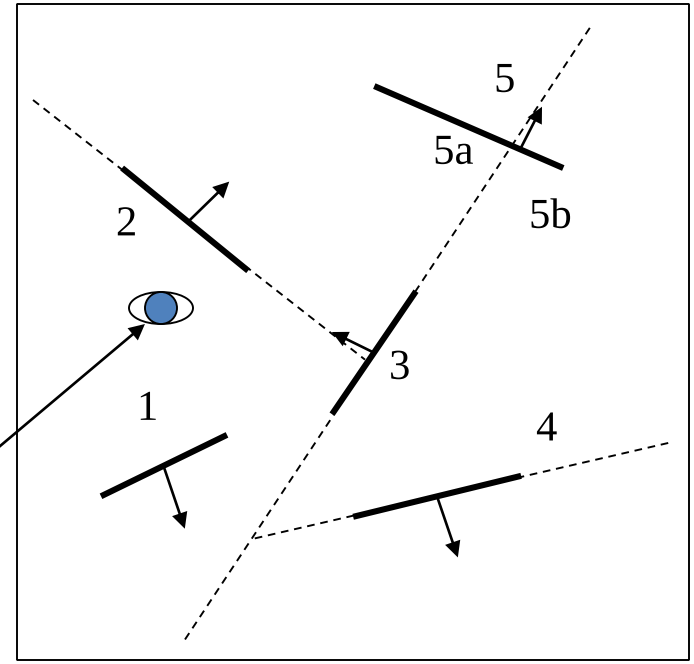
BSP (Binary Space Partitioning)



- an alternate way, started with polygon #5
- Not good, tree is not well-balanced

Drawing with a BSP





Suppose eye is positioned here

Painter's algorithm with BSP

* #3 is root (eye is in front of root)

* draw all, behind 3, then draw 3, and then draw all in front of 3

* when drawing *in front of 3*, we see that eye is behind the subtree root (#2)

* we draw all *<u>in front of 2</u>*, then draw 2 and then draw behind 2

Drawing order is 4, 5b, 3, 5a, 2, 1 *(the later objects can be drawn over the earlier objects)*