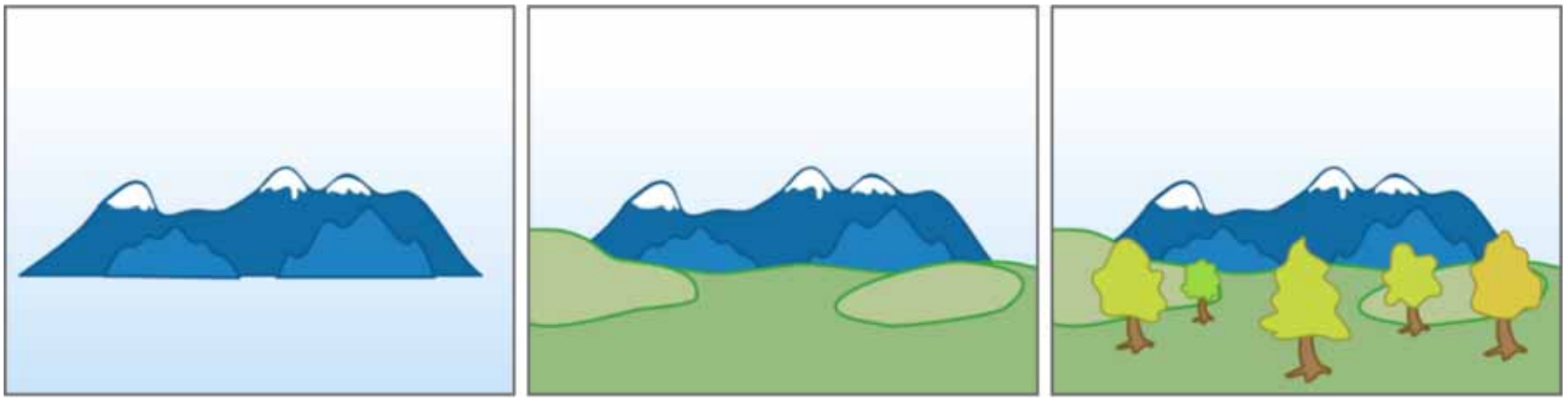
Visibility

Overview

- Given a set of 3D objects and a viewing specification, determine which lines/surfaces are visible and display only them
- Known as:
  - Visible line/surface determination
  - Hidden line/surface elimination/removal

Painters Algorithm

- Start by painting the most distant part of the scene then paint over with closer objects
  - Sort polygons by depth, then paint them in back to front order
  - Paint over parts which are not visible



Painters Algorithm

Given
    List of Polygons { P1, P2, ... , PN }
    An array Intensity [x, y]

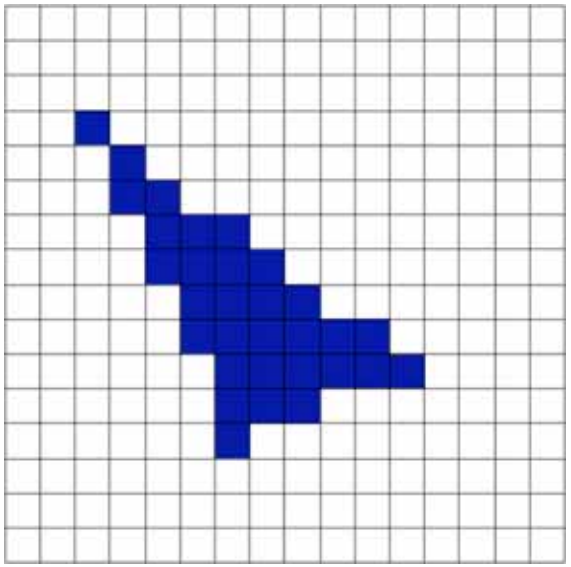
Begin
    Sort polygon list on minimum Z
        (largest Z-value comes first in sorted list)

    for each polygon P in selected list do
        for each pixel (x, y) that intersects P do
            Intensity [x, y] = intensity of P at (x, y)

    Display Intensity array

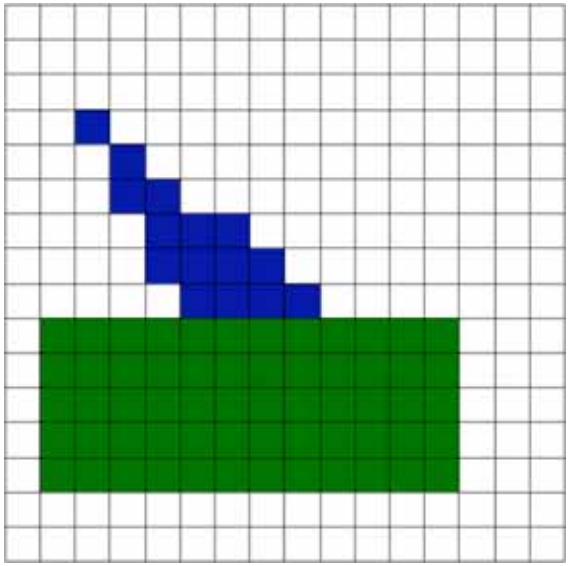
Painters Algorithm

- First polygon:
  - (6, 3, 10), (11, 5, 10), (2, 2, 10)
  - scan it in



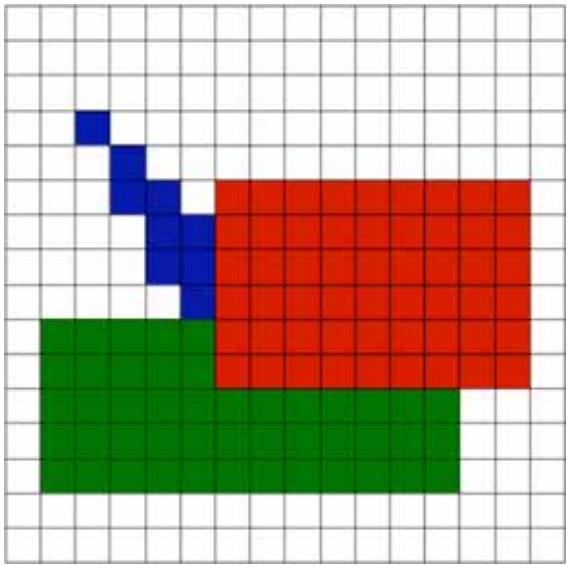
Painters Algorithm

- Second polygon:
  - (1, 2, 8), (12, 2, 8), (12, 6, 8), (1, 6, 8)
  - scan it on top



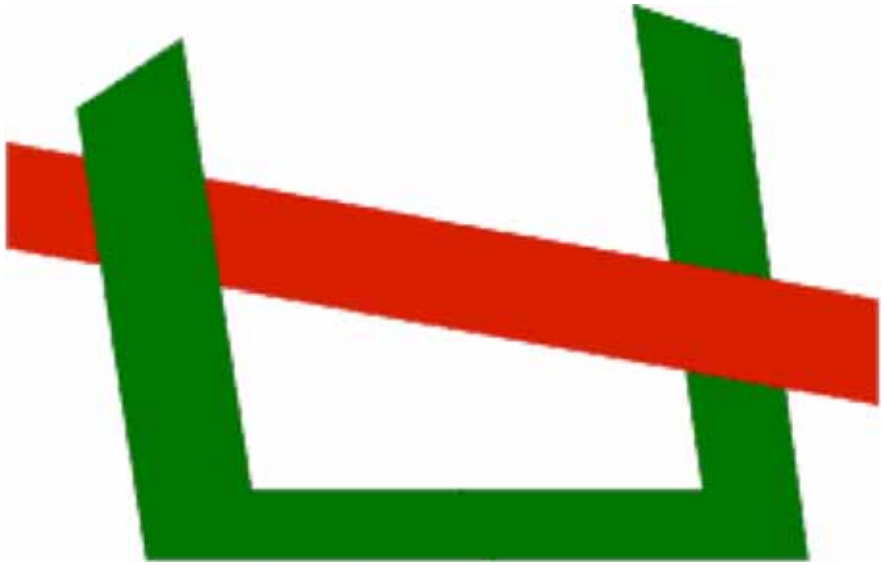
Painters Algorithm

- Third polygon:
  - (6, 5, 5), (14, 5, 5), (14, 10, 5), (6, 10, 5)
  - scan it on top



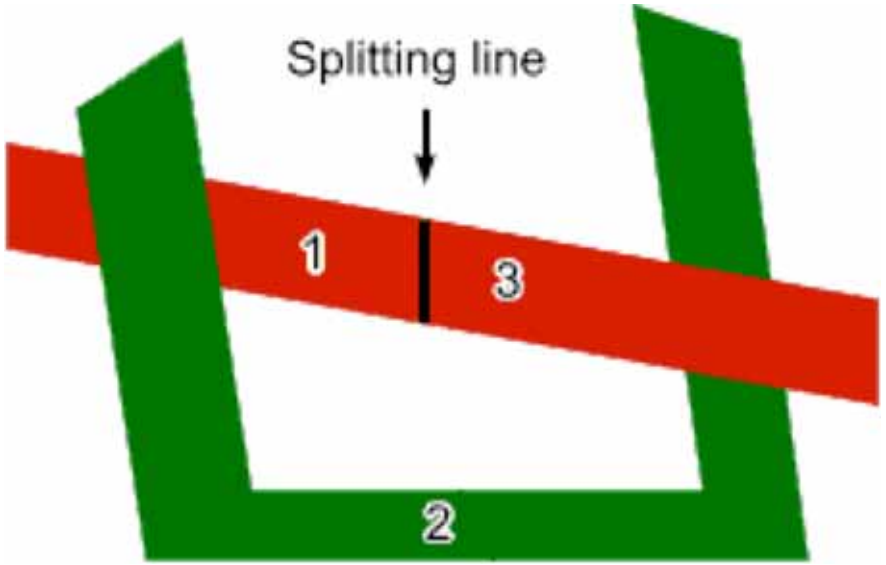
Painters Algorithm – Cyclic Overlapping

- What if z values overlap?



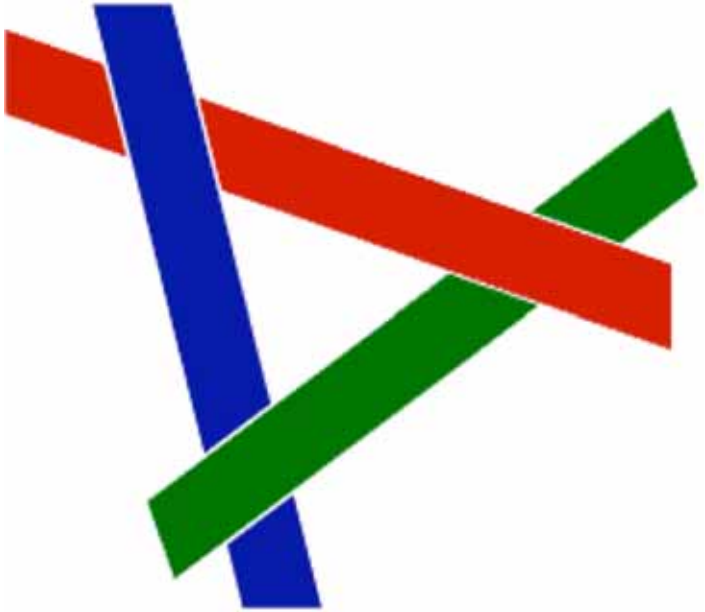
Painters Algorithm – Cyclic Overlapping

- Split polygon(s) along some line(s) and draw in correct depth ordering
  - Split along line
  - Then scan in 1, 2, 3 order



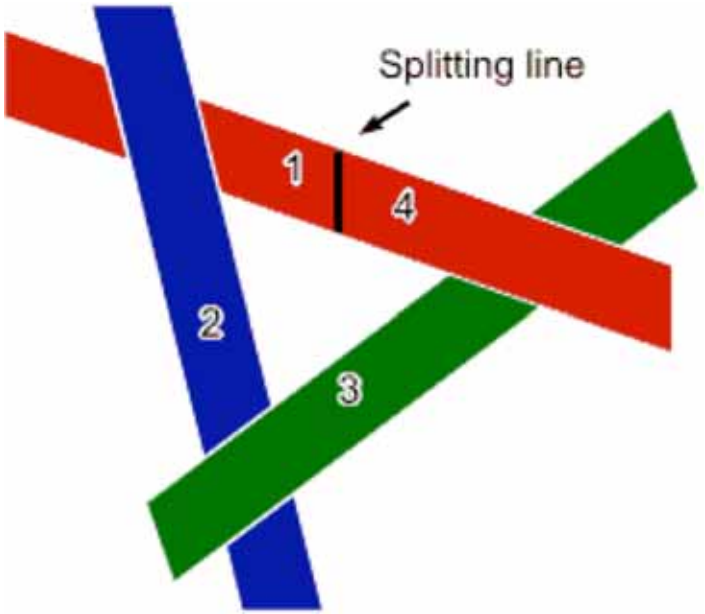
Painters Algorithm – Cyclic Overlapping

- Which order to scan?



Painters Algorithm – Cyclic Overlapping

- Split along line
- Scan in 1, 2, 3, 4 order



Painters Algorithm – Conclusion

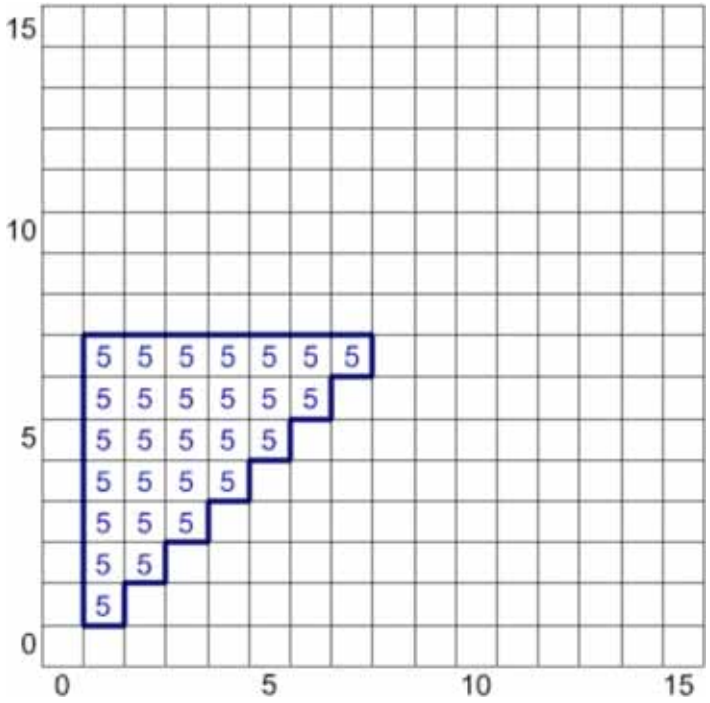
- Good:
  - Painters algorithm is easy and fast to compute
  - Can handle varying degrees of transparency
- Bad:
  - Detecting and splitting cycles
  - Pixel may get re-colored many times

Z-Buffer

- Paint polygons, in addition keep track of the depth computed for each pixel
  - Storage of depth is usually handled in hardware (graphics card)
  - Usually arranged as a 2D array with one element per screen pixel – called the Z-buffer
  - When rendering an object, compare the computed depth for that pixel against what is in the Z-buffer
    - If closer, draw pixel, overwrite z-buffer at location with the new (closer) depth

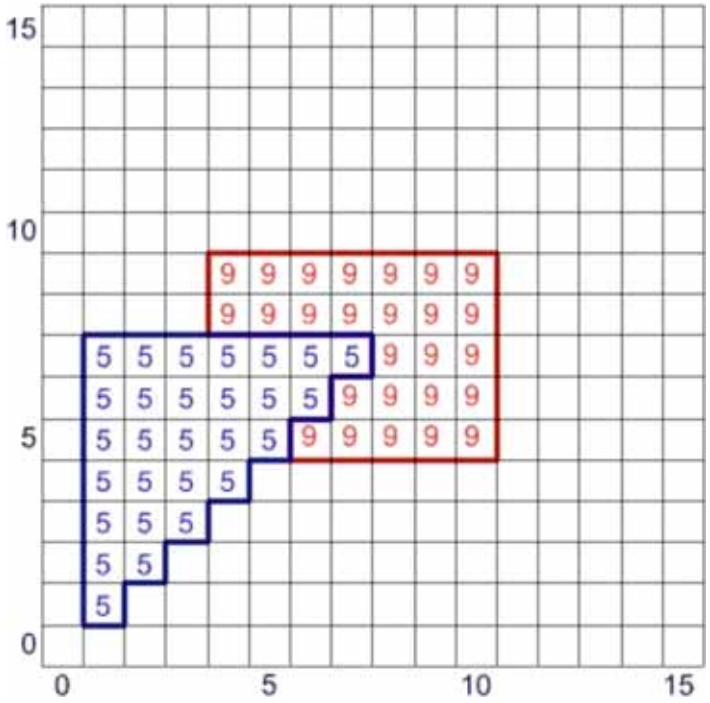
Z-Buffer

- First polygon:
  - (1, 1, 5), (7, 7, 5), (1, 7, 5)
  - scan it in with depth



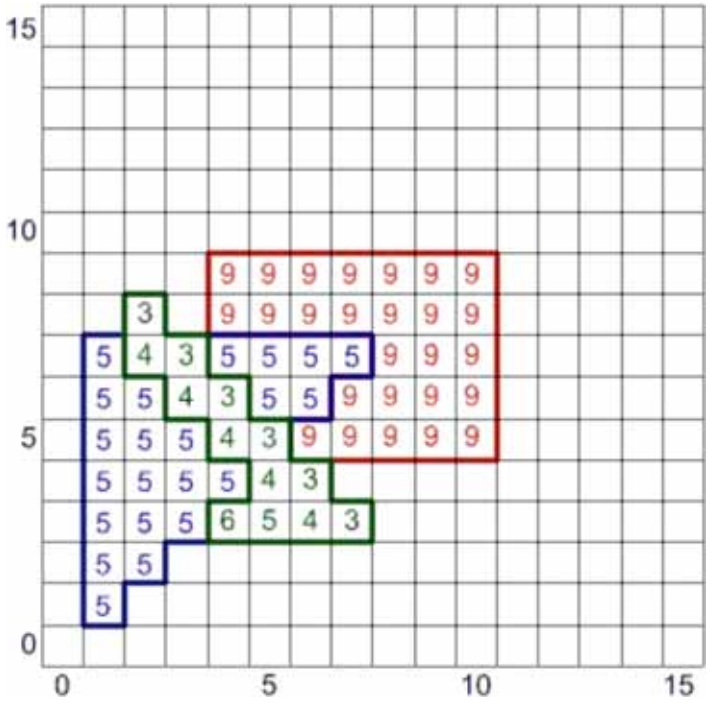
Z-Buffer

- Second polygon:
  - (3, 5, 9), (10, 5, 9), (10, 9, 9), (3, 9, 9)



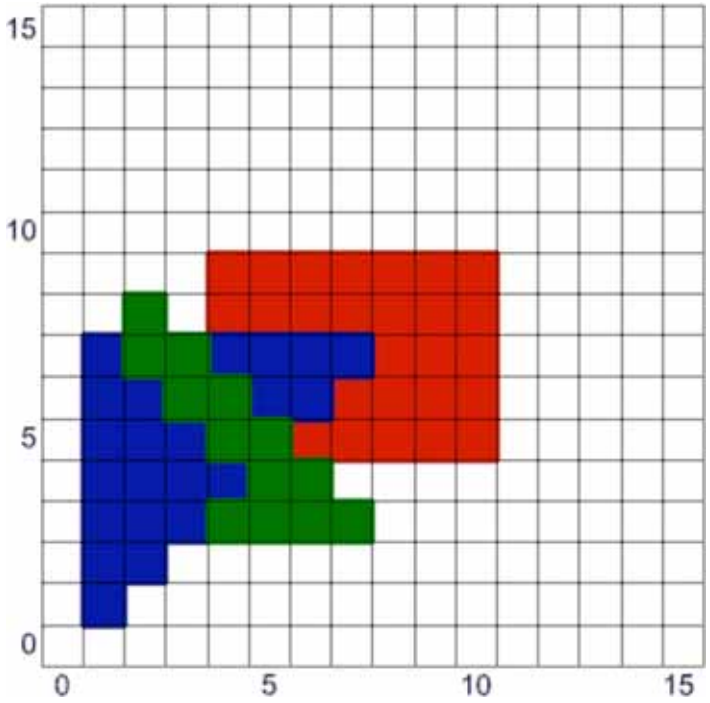
Z-Buffer

- Third polygon:
  - (2, 6, 3), (2, 3, 8), (7, 3, 3)



Z-Buffer

- Result:



Z-Buffer Algorithm

Given

List of Polygons { P1, P2, ... , PN }
An array z-buffer [x, y] initialized to +infinity
An array Intensity [x, y]

Begin

```
for each polygon P in selected list do
    for each pixel (x, y) that intersects P do
        Calculate z-depth of P at (x, y)
        if z-depth < z-buffer [x, y] then
            Intensity [x, y] = intensity of P at (x, y)
            z-buffer [x, y] = z-depth

Display Intensity array
```

Z-Buffer Conclusion

- Good
  - Easy to implement
  - Requires no sorting of surfaces
  - Easy to put in hardware
- Bad
  - Requires lots of memory
    - Usually represented at 24bits or 32bits
    - About 9MB for a 1280 x 1024 display
  - Can alias badly (only one sample per pixel)
  - Can not handle transparent surfaces

A-Buffer

- Basically z-buffer with additional memory to consider contribution of multiple surfaces to a pixel
- Store a list with information about surfaces
  - Color (RGB triple)
  - Opacity
  - Depth
  - Percent area covered
  - Surface ID
  - Miscellaneous rendering parameters
  - Pointer to next

BSP Tree

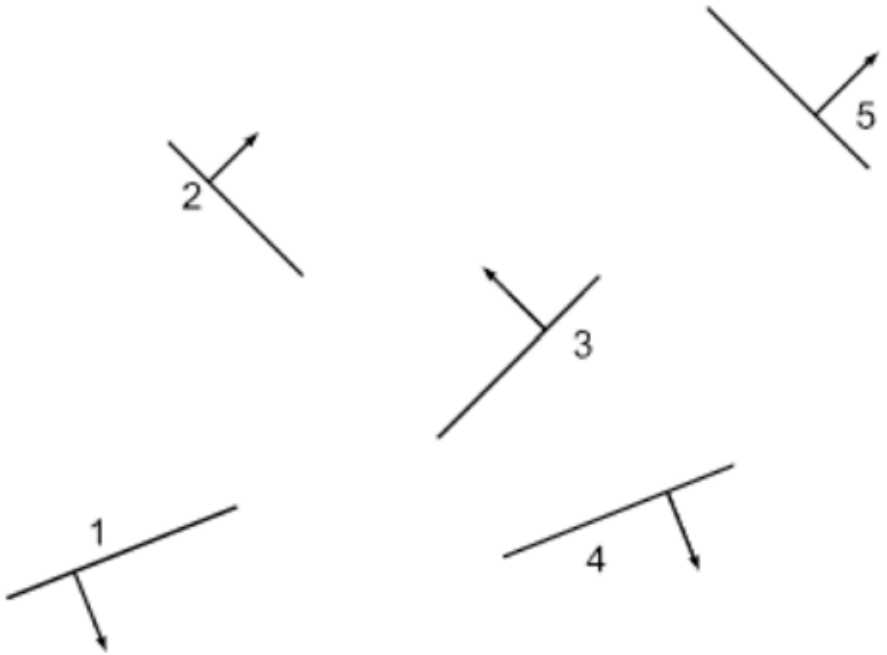
- A Binary Space Partitioning (BSP) tree is a means to divide the scene space
  - Pick a polygon from scene, let that be root of tree
  - Partition remaining scene in terms of in front or behind that polygon based on its normal
    - Any polygon that lies on plane is split, part goes in front, part goes behind
  - Recursively subdivide each sub-tree in same fashion
  - Terminate when each subtree contains only a single node

BSP Tree

- Early game to use a BSP Tree was “Doom”
  - Engine source code since released under the GPL by id Software
  - ftp://ftp.idsoftware.com/idsstuff/

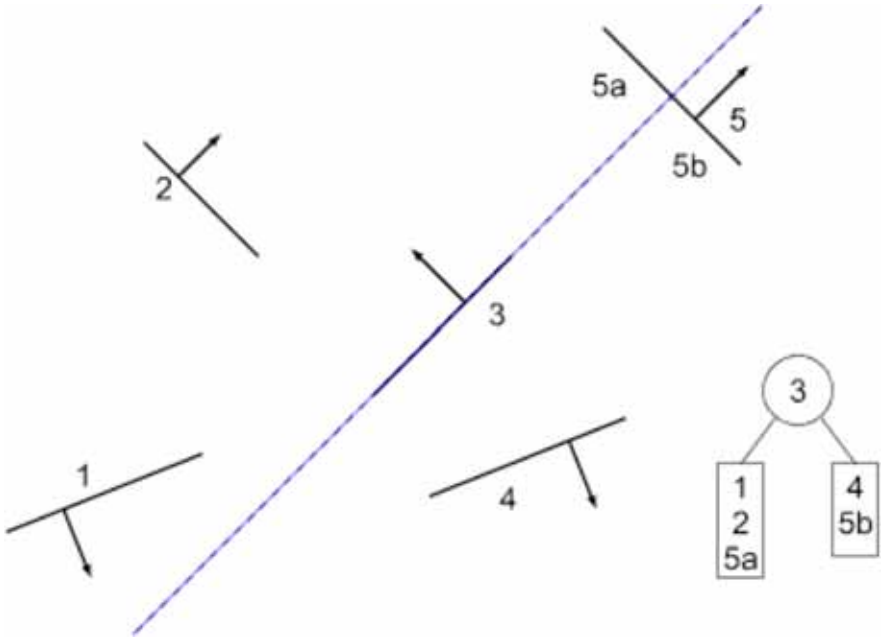
Building a BSP Tree

- Given an initial scene of the following 5 objects



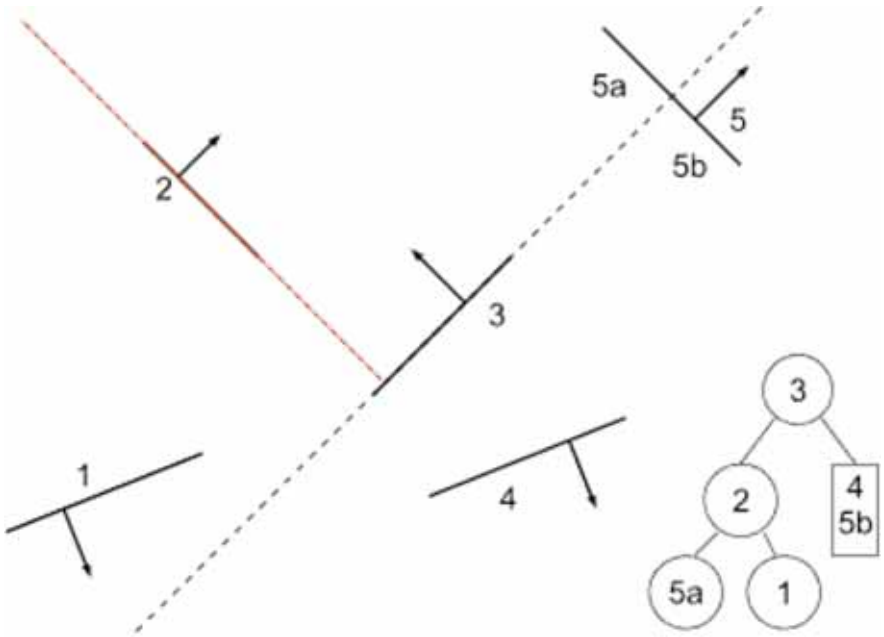
Building a BSP Tree

- Pick a polygon
  - Use 3 as root, split on its plane
  - Polygon 5 split into 5a and 5b



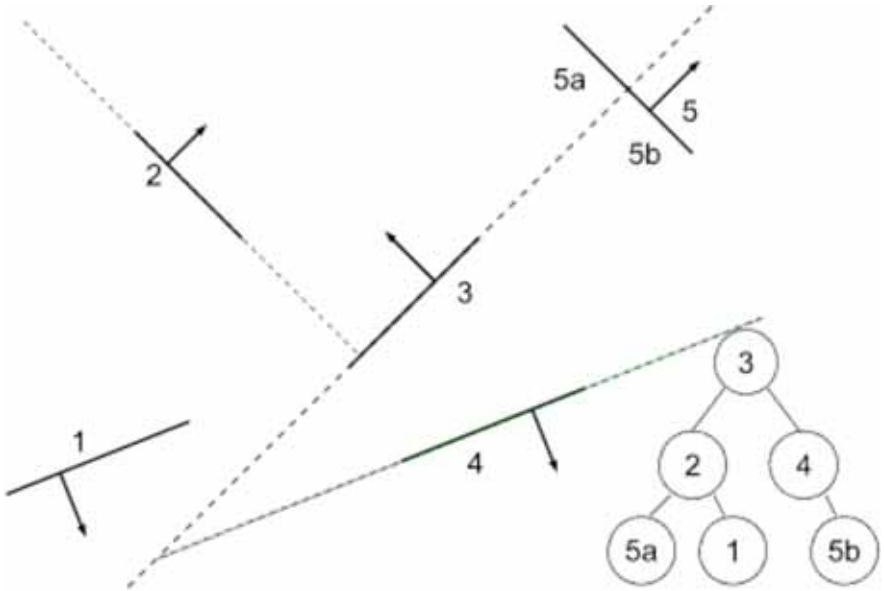
Building a BSP Tree

- Split left subtree at 2
  - 5a is only child in front
  - 1 is only child behind



Building a BSP Tree

- Split the right subtree about 4
  - 5b is remaining child which is behind 4



Displaying a BSP Tree

- Remarkably, displaying a BSP is as easy as an in-order traversal

```
DisplayBSP(tree)
        if (tree not empty ) then
                if (viewer in front of root ) then
                        DisplayBSP ( tree -> back )
                        DisplayPolygon ( tree -> root )
                        DisplayBSP ( tree -> front )
        else
                        DisplayBSP ( tree -> front )
                        DisplayPolygon ( tree -> root )
                        DisplayBSP ( tree -> back )
```

Ray Tracing

- Also known as ray casting, is capable of determining which surfaces are visible at a given pixel
  - We'll revisit ray tracing in more detail next class
- General Premise:

```
for (each scan line) do
        for (each pixel on scan line) do
                for (each object in scene) do
                        if (object is intersected and closest seen thus far)
                                record intersection and object name
                set pixel color to that at closest intersection
```