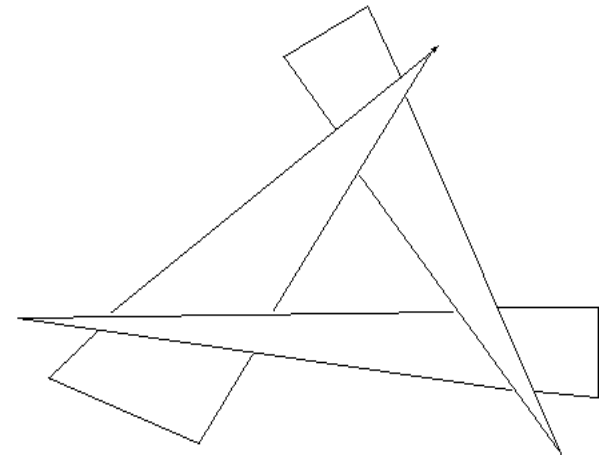


# Visibility - painters algorithm

- Algorithm
  - Choose an order for the polygons based on some choice (e.g. depth to a point on the polygon)
  - Render the polygons in that order, deepest one first
- This renders nearer polygons over further.
- Works for some important geometries (2.5D - e.g. VLSI, mazes--but more efficient algorithms exist)
- Doesn't work in this form for most geometries (see figure)



# The Z - buffer

- For each pixel on screen, have a second memory location - called the z-buffer
- Set this buffer to a value corresponding to the furthest point
- As a polygon is filled in, compute the depth value of each pixel
  - if depth < z buffer depth, fill in pixel and new depth
  - else disregard
- Typical implementation: Compute Z while scan-converting. A  $\partial Z$  for every  $\partial X$  is easy to work out.

# The Z - buffer

- Advantages:
  - simple; hardware implementation common
  - efficient z computations are easy.
  - ok with lots of surfaces (if there are lots, they tend to be small, and not much difference to this algorithm)
- Disadvantages:
  - over renders - can be slow for very large collections of polygons - may end up scan converting many hidden objects
  - quantization errors can be annoying (not enough bits in the buffer)
  - doesn't do transparency, or filtering for anti-aliasing.

# The A - buffer

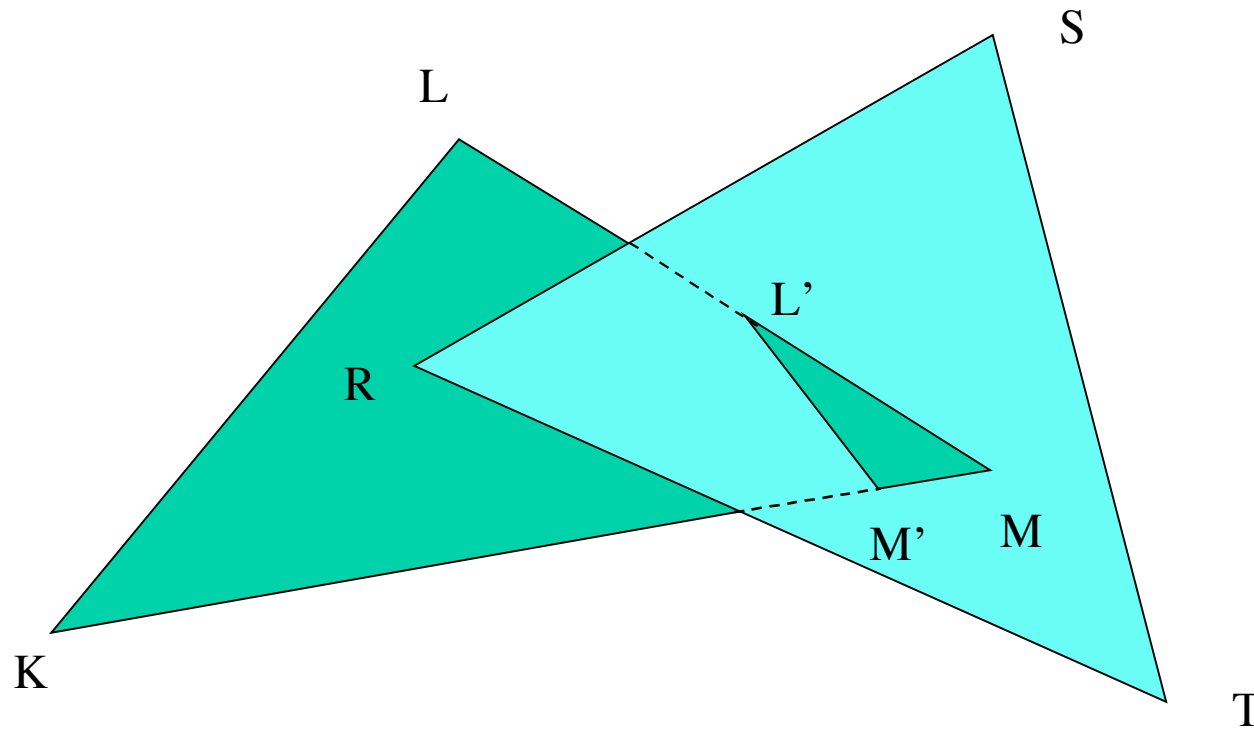
- For transparent surfaces and filter anti-aliasing:
- Algorithm: filling buffer
  - at each pixel, maintain a pointer to a list of polygons sorted by depth.
  - when filling a pixel:
    - if polygon is opaque and covers pixel, insert into list, removing all polygons farther away
    - if polygon is opaque and only partially covers pixel, insert into list, but don't remove farther polygons
- Algorithm: rendering pixels
  - at each pixel, traverse buffer using brightness values in polygons to fill.
  - values are used either in transparency or for filtering for aliasing

# Scan line algorithm

- Assume polygons do not intersect one another.
- Observation: on any given scan line, the visible polygon can change only at an edge.
- Algorithm:
  - fill all polygons simultaneously at each scan line, have all edges that cross scan line in AEL
  - keep record of current depth at current pixel - use to decide which is in front in filling span

# Scan line algorithm

- To deal with penetrating polygons, split them up



# Scan line algorithm

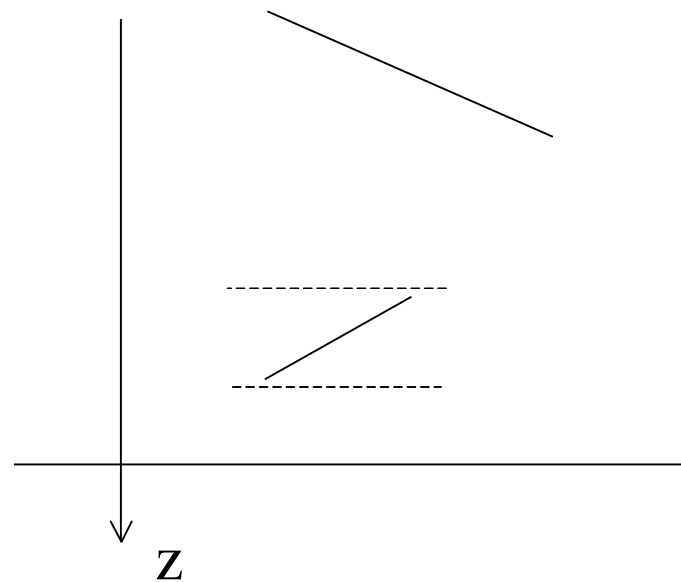
- Advantages:
  - potentially fewer quantization errors (typically more bits available for depth, but this depends)
  - filter anti-aliasing can be made to work.
- Disadvantages:
  - invisible polygons clog AEL, ET (can get expensive for complex scenes).

# Depth sorting

- Sort in order of decreasing depth
  - use closest point
- Render in sorted order
- For surface  $S$  with greatest depth
  - if no depth overlaps (z extents intersect) with other surfaces, then render (like painter's algorithm), and remove surface from list
  - if a depth overlap is found, test for problem overlap in image plane
  - if  $S$ ,  $S'$  overlap in depth and in image plane, swap and try again
  - if  $S$ ,  $S'$  have been swapped already, split one across plane of other (like clipping) and reinsert
- Testing image plane problem overlaps (test get increasingly expensive):
  - xy bounding boxes do not intersect
  - *or*  $S$  is behind the plane of  $S'$
  - *or*  $S'$  is in front of the plane of  $S$
  - *or*  $S$  and  $S'$  do not intersect
- Advantages:
  - filter anti-aliasing works fine
  - no depth quantization error
  - works well if not too much depth overlap (rarely get to expensive cases)
- Disadvantages:
  - gets expensive with lots of depth overlap (over-renders)

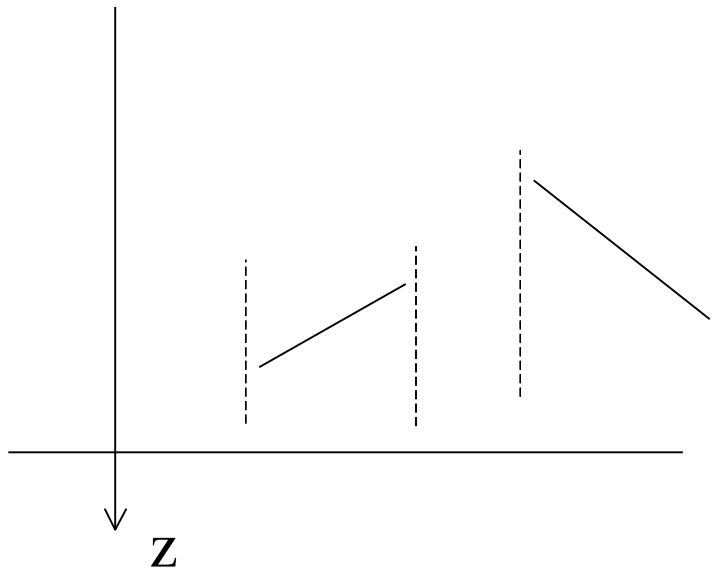


# Depth sorting (2D illustrations)



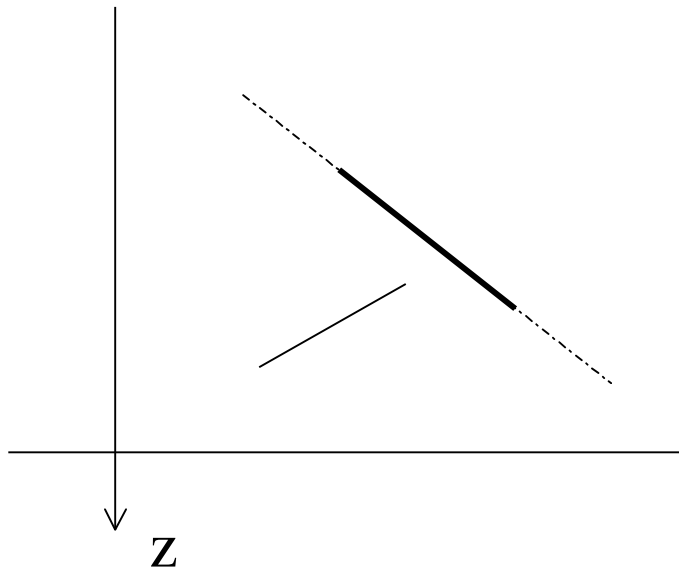
No depth overlap  
between furthest  
object and rest of  
list--paint it.

# Depth sorting (2D illustrations)



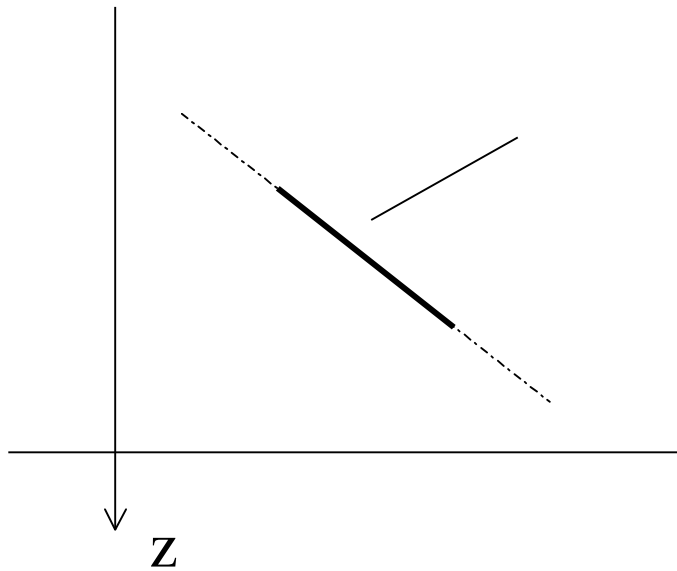
Overlap in depth,  
but lack of overlap  
in image plane can  
be resolved by  
bounding boxes. \

# Depth sorting (2D illustrations)



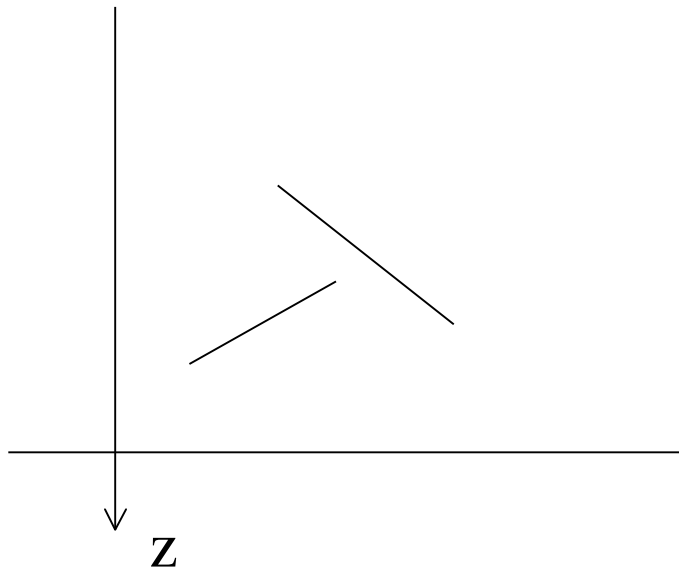
It is safe to paint the furthest, but figuring this out requires observing that the nearest is all on the same side of the plane (line in the 2D figure) of the furthest.

# Depth sorting (2D illustrations)



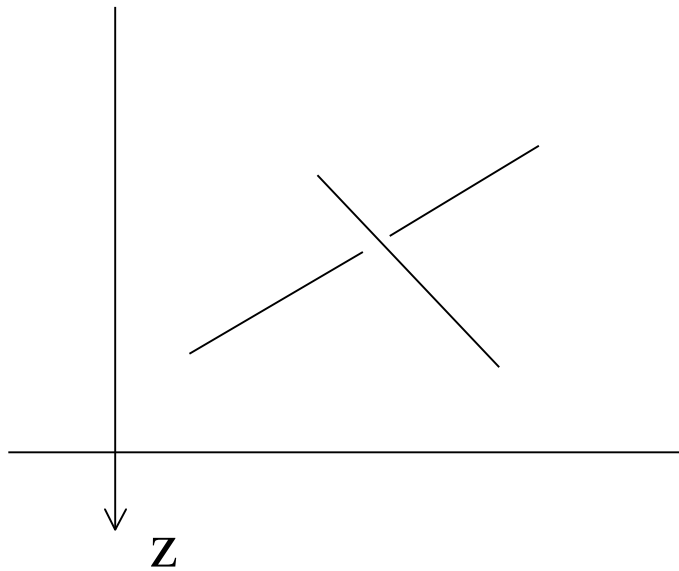
It is safe to paint the furthest, but figuring this out requires observing that it is all on the other side of the plane (line in the 2D figure) of the nearest plane.

# Depth sorting (2D illustrations)



It is still safe to paint the furthest, but figuring this out requires reversing the nearest and furthest in the previous test.

# Depth sorting (2D illustrations)



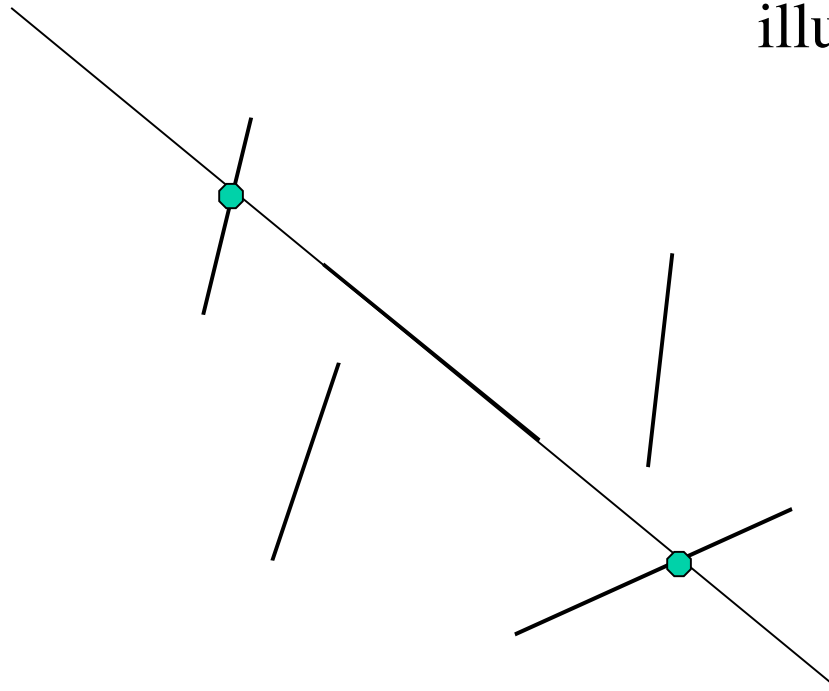
If the preceding tests fail, we have to split the far polygon (line in the drawing) with the plane of the near polygon (basically a clip operation), and put the pieces into the list, and carry on.

# BSP - trees

- Construct a tree that gives a rendering order
- Tree splits 3D world into cells, each of which contain at most one piece of polygon.
- Constructing tree:
  - Choose polygon (arbitrary)
  - split its cell using plane on which polygon lies
  - continue until each cell contains only one polygon

# BSP - trees

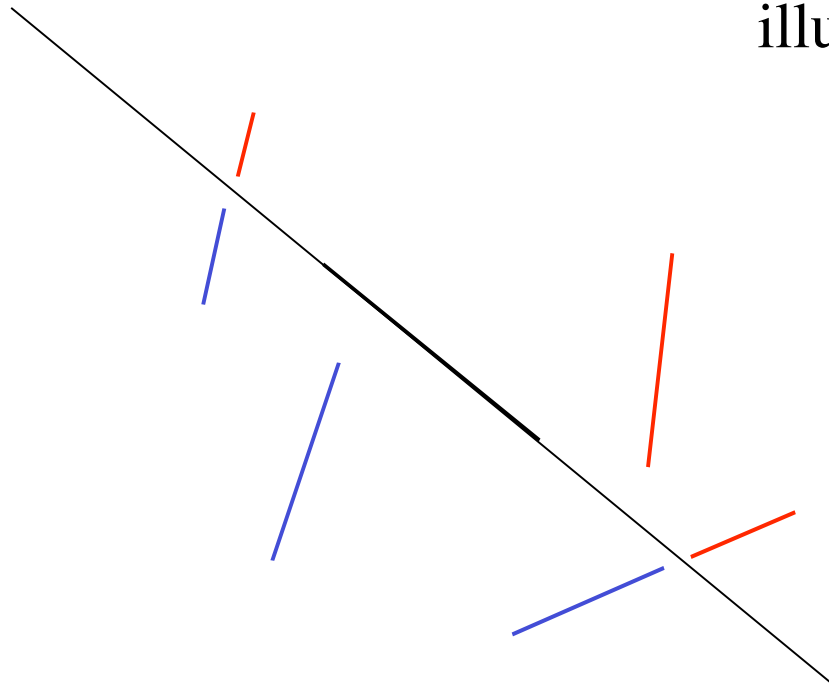
2D version for  
illustration





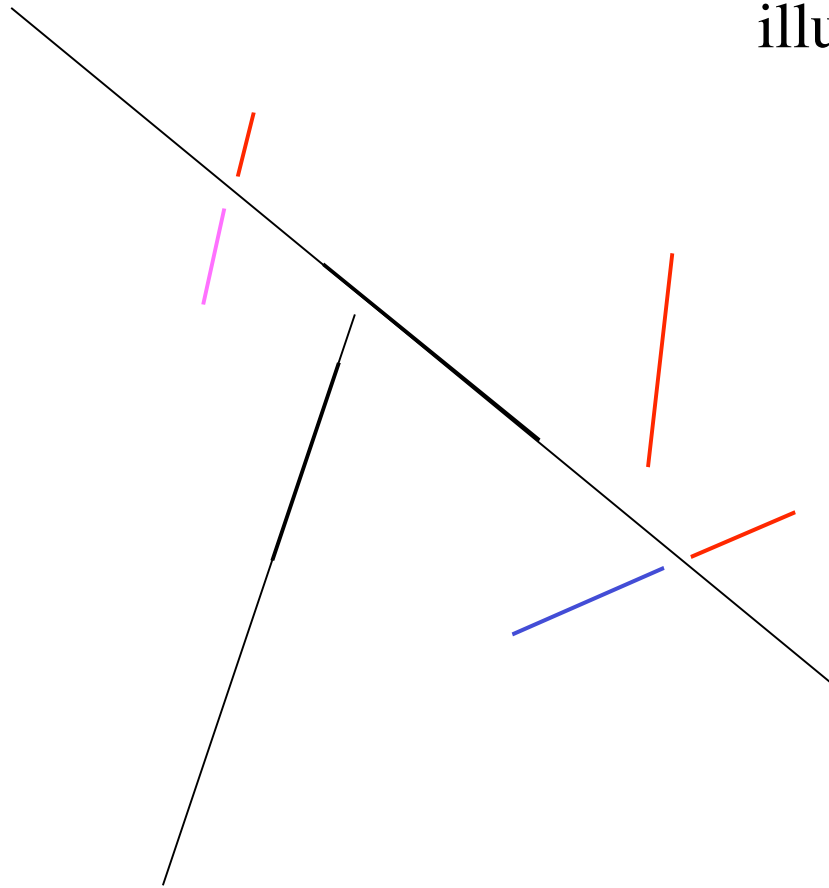
# BSP - trees

2D version for  
illustration

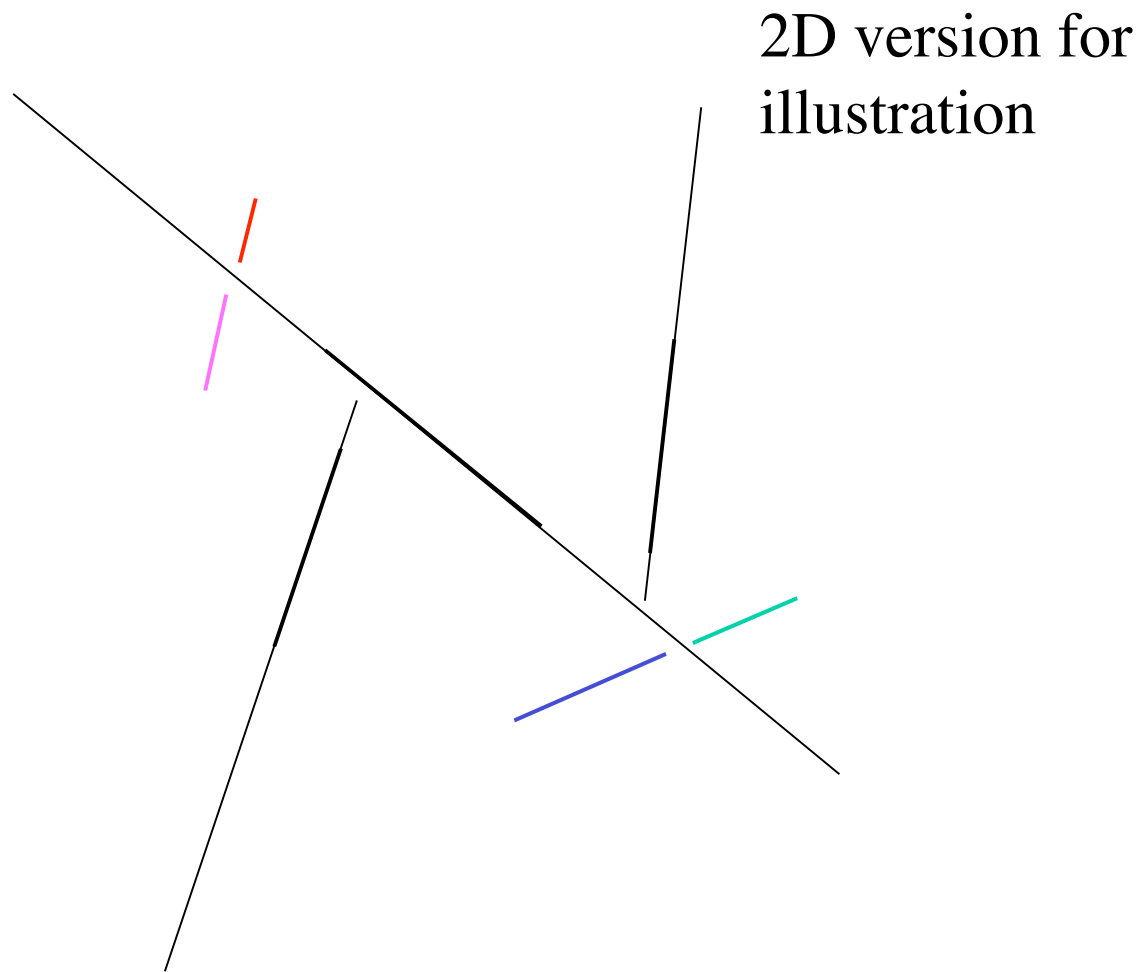


# BSP - trees

2D version for  
illustration



# BSP - trees

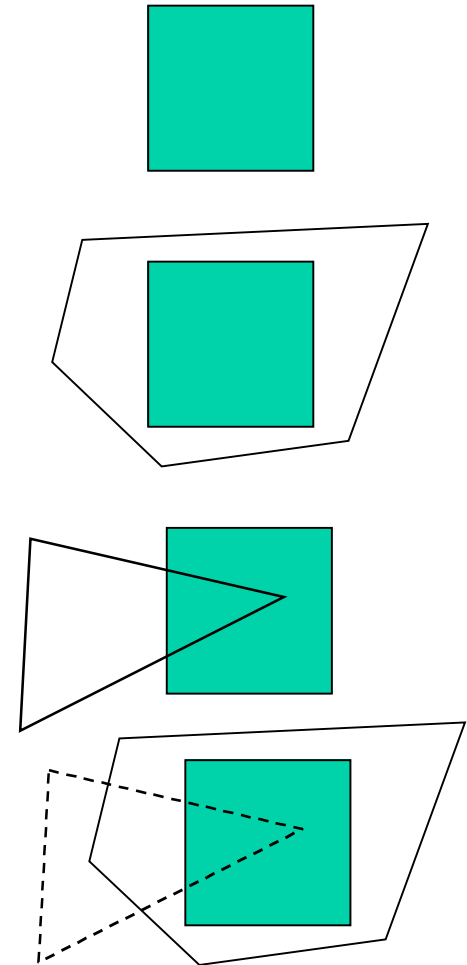


# BSP - trees

- Rendering tree:
  - recursive descent
  - render back, node polygon, front
- Disadvantages:
  - many small pieces of polygon (more splits than depth sort!)
  - over rendering (does not work well for complex scenes with lots of depth overlap)
  - hard to get balanced tree
- Advantages:
  - one tree works for any focal point (good for cases when scene is static)
  - filter anti-aliasing works fine, as does transparency
  - data structure is worth knowing about

# Area subdivision

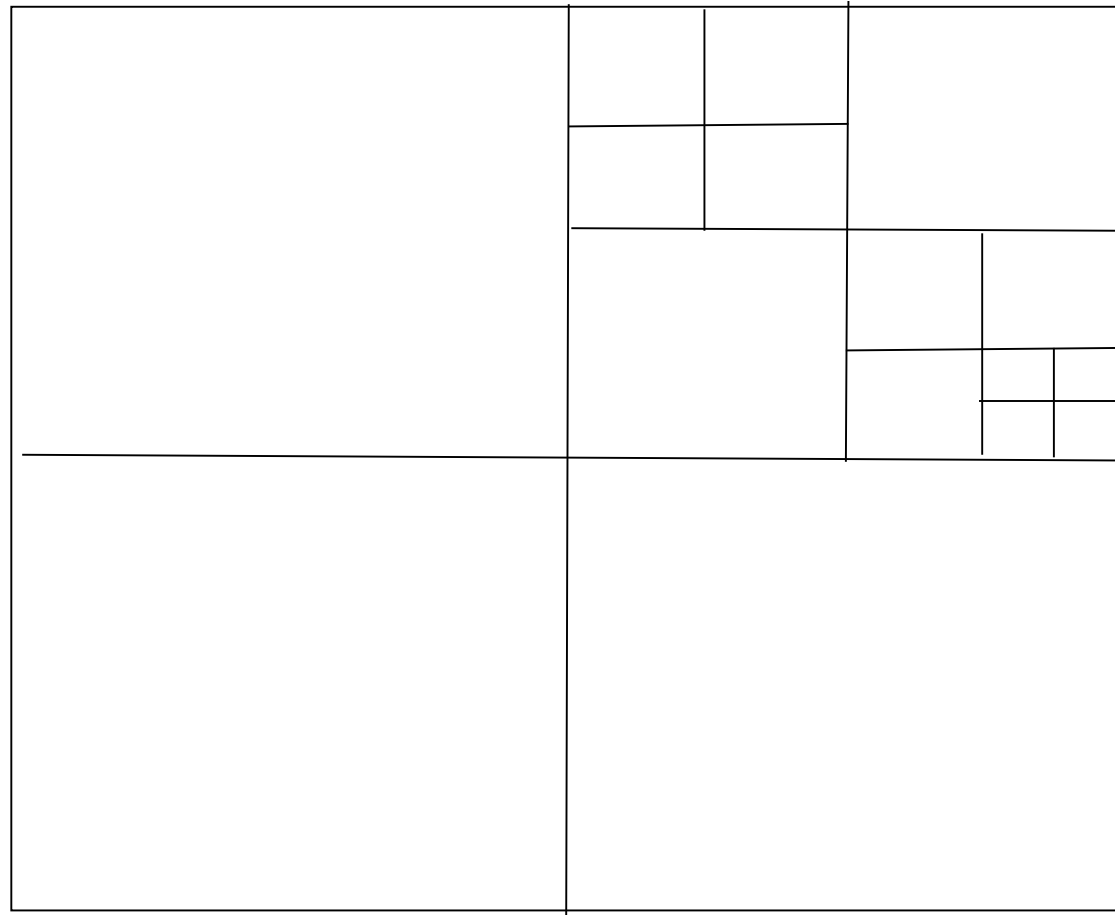
- Four tractable cases for a given region in image plane:
  - no surfaces project to the region (do nothing or paint background)
  - only one surface completely surrounds the region (paint surface)
  - only one surface is completely inside the region or overlaps the region (do nothing or paint background and then scan convert region)
  - a polygon is completely in front of everything else in that region determined by considering depths of the polygons at the corners of the region (paint surface)



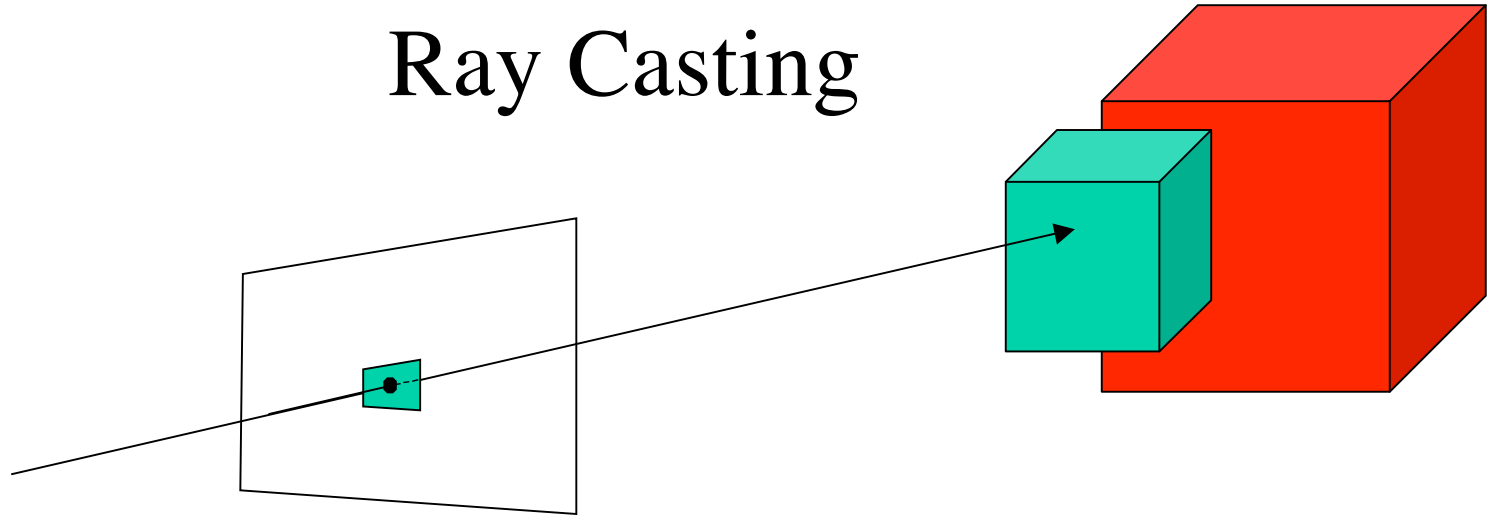
# Area subdivision

- Algorithm:
  - subdivide each region until one of these cases is true or until region is very small
  - if case is true, deal with it
  - if region is small, choose surface with smallest depth.
  - determining cases quickly makes use of the same ideas in depth sort (depth sorting, bounding boxes, tests for which side of the plane), and the difficult cases are deferred by further subdivision.
- Advantages:
  - can be very efficient
  - no over rendering
  - anti-aliases well (subdivide a bit further)

# One Subdivision Strategy



# Ray Casting



- Image precision algorithm
- For each pixel cast a ray into the world
  - For each surface
    - determine intersection point with ray
  - Render pixel based on closest surface



# Ray Casting

- First step in ray tracing algorithm
- Expensive
- Good performance usually requires clever data structures such as bounding volumes for object groups or storing world occupancy information in octrees.
- Other main problem is computing intersection.
- For polygons, we can use the standardized orthographic space where we can work in 2D.
- See book for intersections with spheres in perspective space.
- Useful for “picking”--not expensive here (why?)

# Ray Tracing--teaser

- Idea is very simple--follow light around
- Following all the light around is intractable, so we follow the light that makes the most difference
- Work backwards from what is seen
- Simple ray tracer
  - Cast a ray through each pixel (as in ray casting for visibility)
  - From intersection point cast additional rays to determine the color of the pixel.
  - For diffuse component, must cast rays to the lights
  - We may also add in some “ambient” light
  - For mirrors, must cast ray in mirror direction (recursion--what is the stopping condition)

# Current state of intro students graphic's ability

Know how to draw polygons

Know about cameras

Know how to map 3D polygons onto the screen

Know how to draw the bits closest to the cameras

## Issues

Should we live in a polygonal world?

How do you get polygons for complex objects?

What color should each pixel be?