7. From Vertices to Fragments

Lecture Overview

- Clipping
  - Line-Segment Clipping
  - Polygon Clipping
- Rasterization
  - Line Grawing Algorithms
    - DDA, Bresenham's Algorithm
  - Polygon Rasterization
- Hidden-Surface Removal
- Antialiasing

- Reading: ANG Ch. 7, except 7.13

Implementation I

Objectives

- Introduce basic implementation strategies
- Clipping
- Scan conversion

Overview

- At end of the geometric pipeline, vertices have been assembled into primitives
- Must clip out primitives that are outside the view frustum
  - Algorithms based on representing primitives by lists of vertices
- Must find which pixels can be affected by each primitive
  - Fragment generation
  - Rasterization or scan conversion

Required Tasks

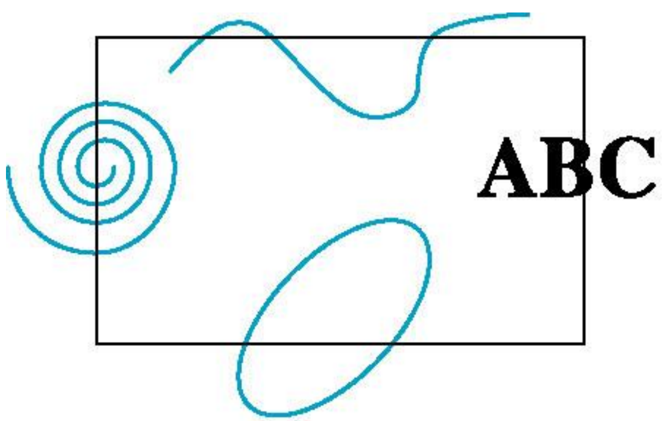
- Clipping
- Rasterization or scan conversion
- Transformations
- Some tasks deferred until fragement processing
  - –Hidden surface removal
  - –Antialiasing

Rasterization Meta Algorithms

- Consider two approaches to rendering a scene with opaque objects
- For every pixel, determine which object that projects on the pixel is closest to the viewer and compute the shade of this pixel
  - Ray tracing paradigm
- For every object, determine which pixels it covers and shade these pixels
  - Pipeline approach
  - Must keep track of depths

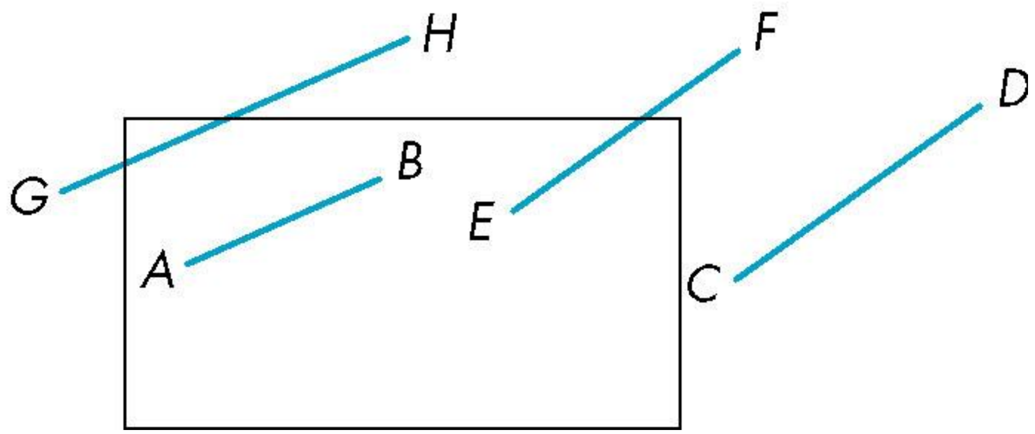
Clipping

- 2D against clipping window
- 3D against clipping volume
- Easy for line segments polygons
- Hard for curves and text
  - –Convert to lines and polygons first



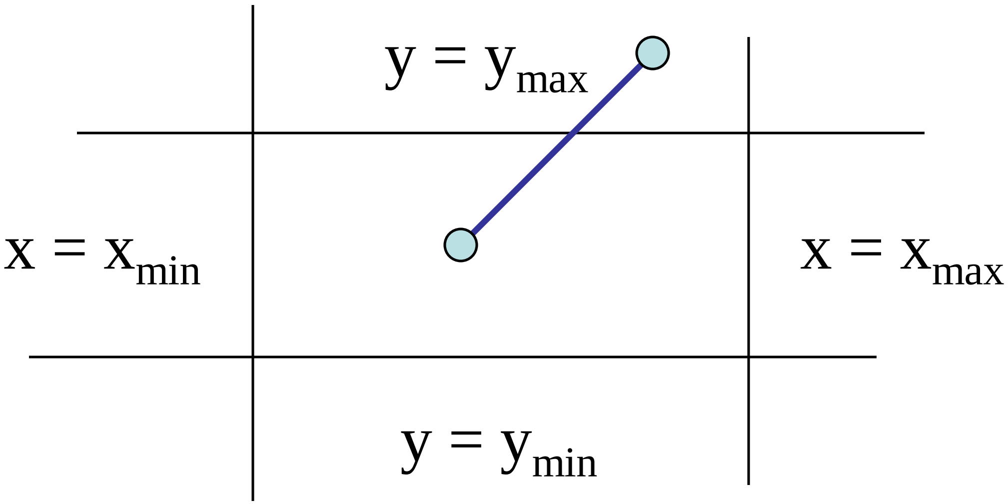
Clipping 2D Line Segments

- Brute force approach: compute intersections with all sides of clipping window
  - –Inefficient: one division per intersection



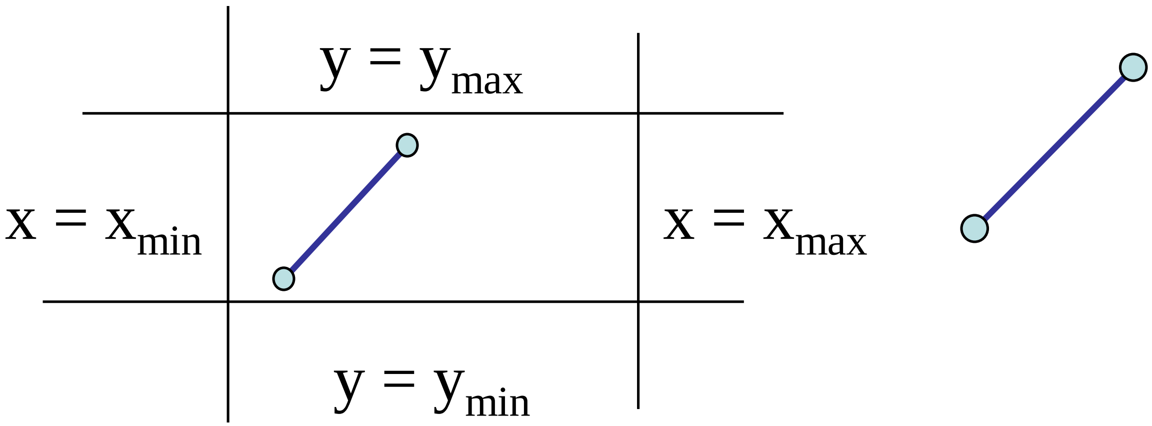
Cohen-Sutherland Algorithm

- Idea: eliminate as many cases as possible without computing intersections
- Start with four lines that determine the sides of the clipping window



The Cases

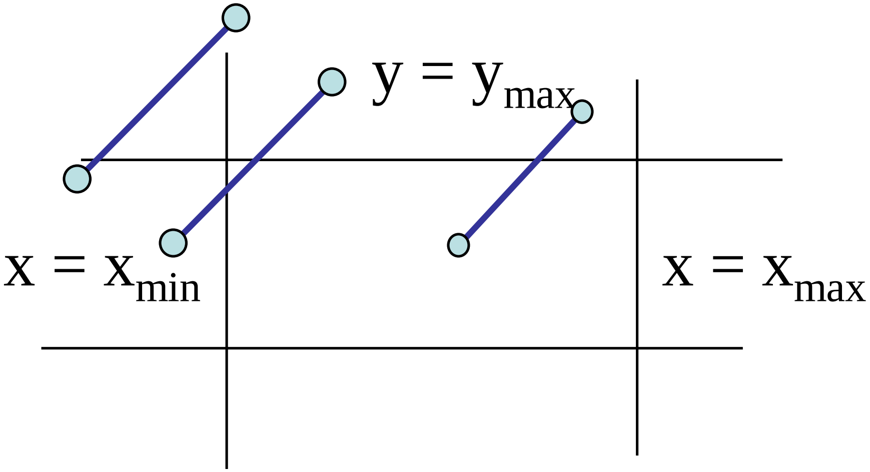
- Case 1: both endpoints of line segment inside all four lines
  - Draw (accept) line segment as is



- Case 2: both endpoints outside all lines and on same side of a line
  - Discard (reject) the line segment

The Cases

- Case 3: One endpoint inside, one outside
  - –Must do at least one intersection
- Case 4: Both outside
  - –May have part inside
  - –Must do at least one intersection

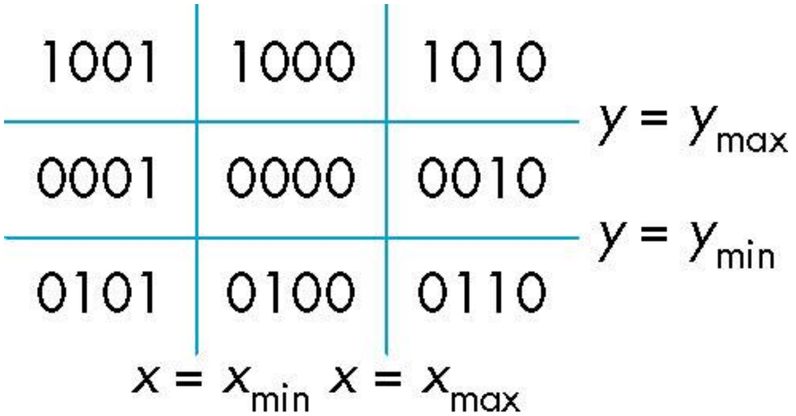


Defining Outcodes

- For each endpoint, define an outcode

$b_0 b_1 b_2 b_3$

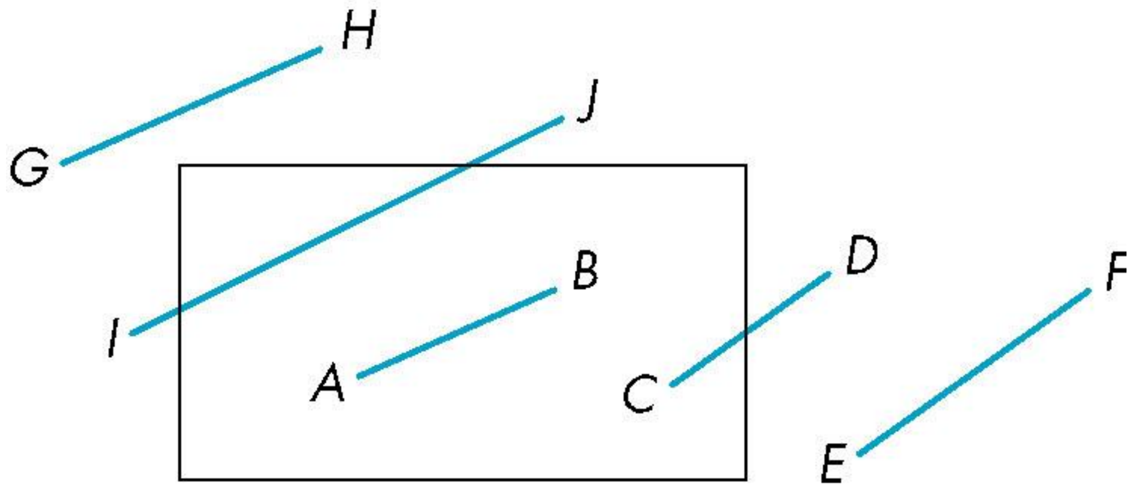
$b_0 = 1$ if $y > y_{max}$, 0 otherwise
$b_1 = 1$ if $y < y_{min}$, 0 otherwise
$b_2 = 1$ if $x > x_{max}$, 0 otherwise
$b_3 = 1$ if $x < x_{min}$, 0 otherwise



- Outcodes divide space into 9 regions
- Computation of outcode requires at most 4 subtractions

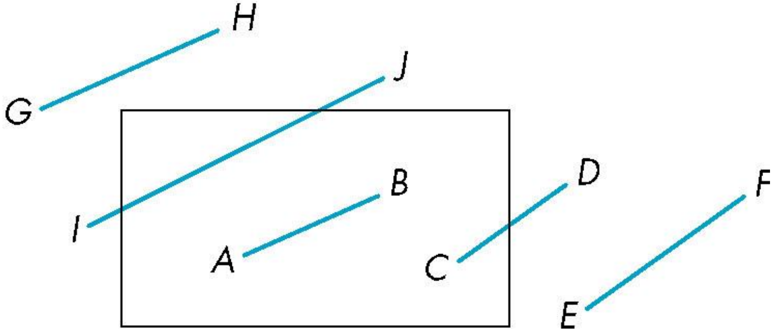
Using Outcodes

- Consider the 5 cases below
- AB: outcode(A) = outcode(B) = 0
  - Accept line segment



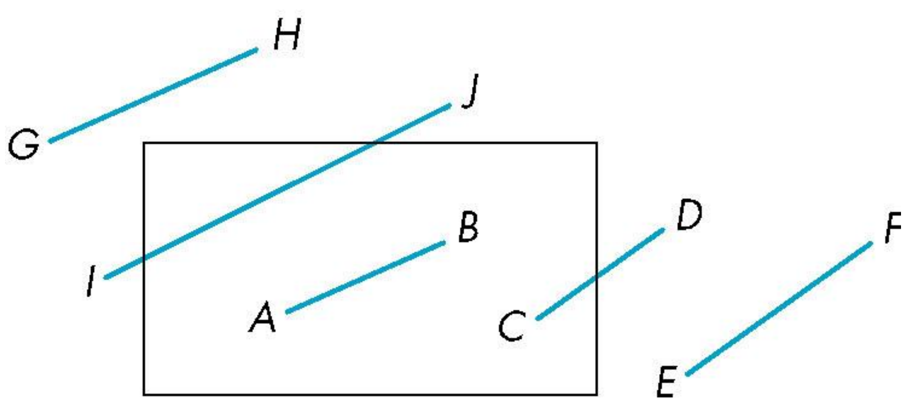
Using Outcodes

- CD: outcode (C) = 0, outcode(D) ≠ 0
  - Compute intersection
  - Location of 1 in outcode(D) determines which edge to intersect with
  - Note if there were a segment from A to a point in a region with 2 ones in outcode, we might have to do two interesections



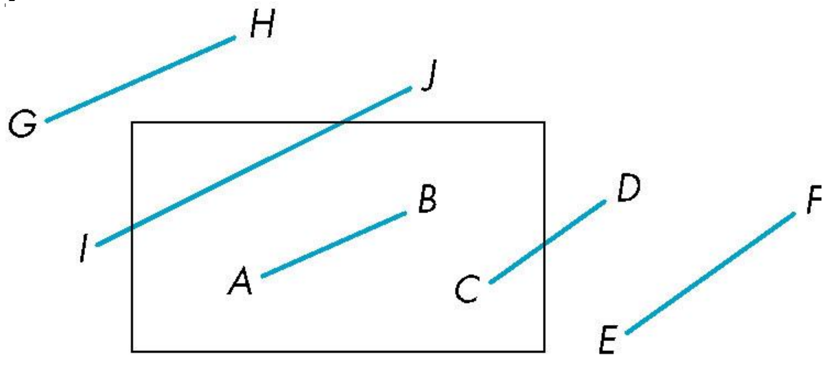
Using Outcodes

- EF: outcode(E) logically ANDed with outcode(F) (bitwise) $\neq 0$
  - Both outcodes have a 1 bit in the same place
  - Line segment is outside of corresponding side of clipping window
  - reject



Using Outcodes

- GH and IJ: same outcodes, neither zero but logical AND yields zero
- Shorten line segment by intersecting with one of sides of window
- Compute outcode of intersection (new endpoint of shortened line segment)
- Reexecute algorithm

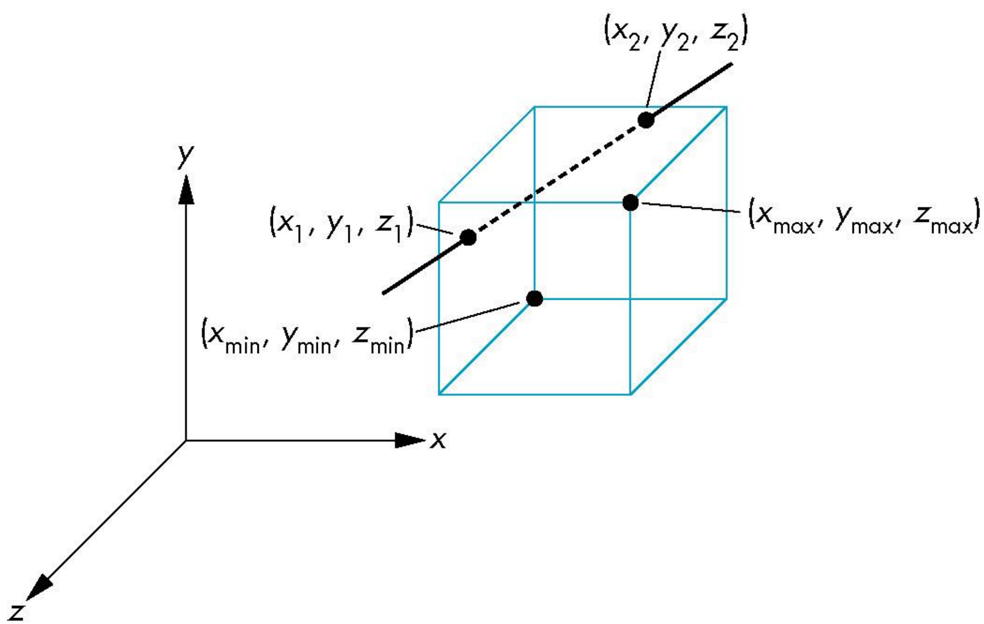


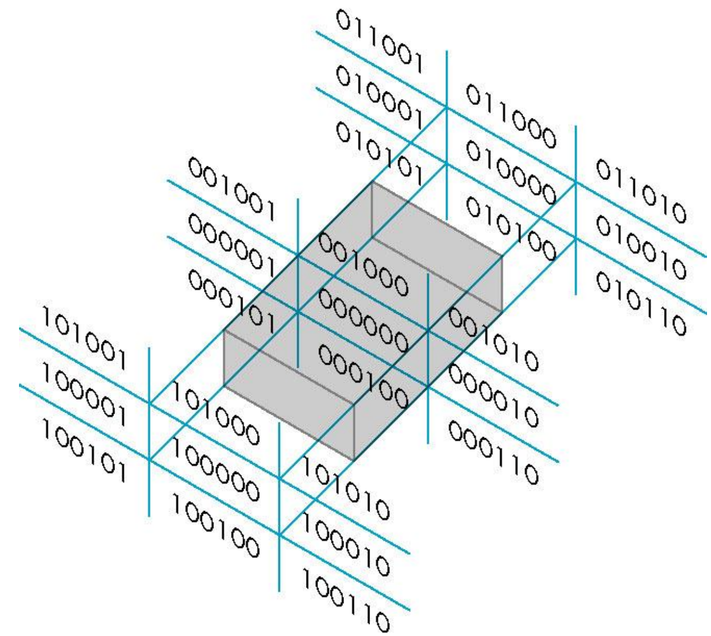
Efficiency

- In many applications, the clipping window is small relative to the size of the entire data base
  - Most line segments are outside one or more side of the window and can be eliminated based on their outcodes
- Inefficiency when code has to be reexecuted for line segments that must be shortened in more than one step

Cohen Sutherland in 3D

- Use 6-bit outcodes
- When needed, clip line segment against planes

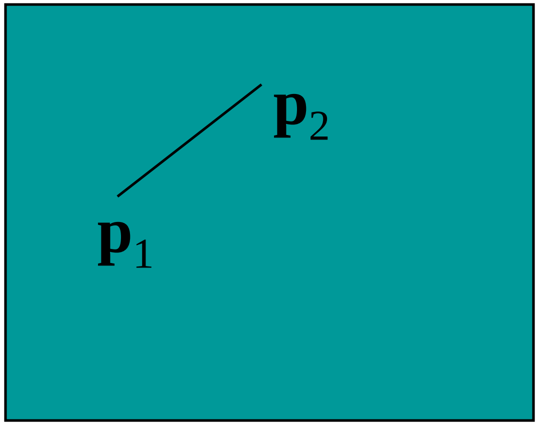




Liang-Barsky Clipping

- Consider the parametric form of a line segment

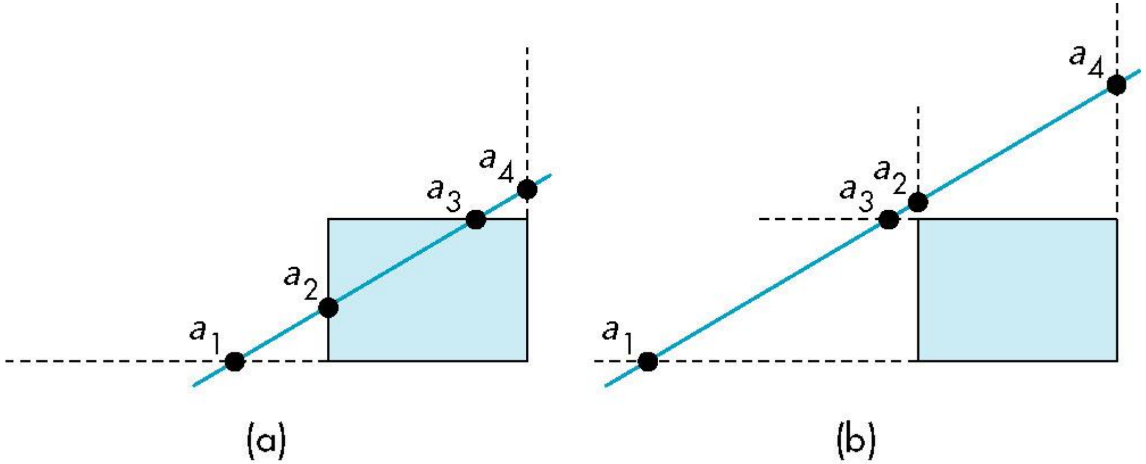
$$\mathbf{p}(\alpha) = (1-\alpha)\mathbf{p}_1 + \alpha\mathbf{p}_2 \quad 1 \geq \alpha \geq 0$$



- We can distinguish between the cases by looking at the ordering of the values of α where the line determined by the line segment crosses the lines that determine the window

Liang-Barsky Clipping

- In (a): $\alpha_4 > \alpha_3 > \alpha_2 > \alpha_1$
  - Intersect right, top, left, bottom: shorten
- In (b): $\alpha_4 > \alpha_2 > \alpha_3 > \alpha_1$
  - Intersect right, left, top, bottom: reject

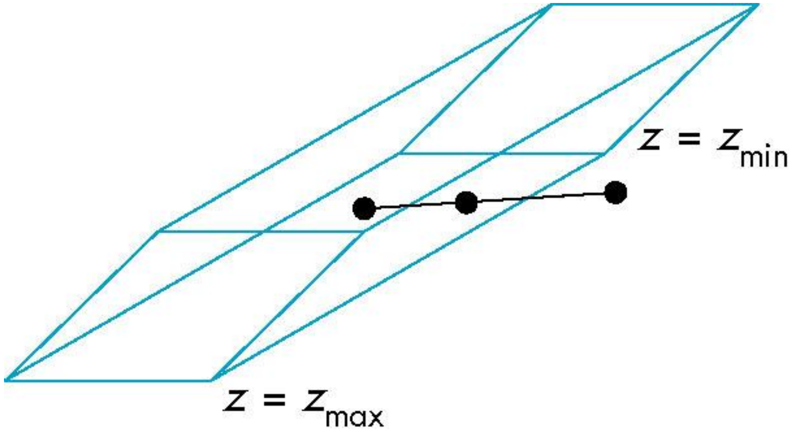


Advantages

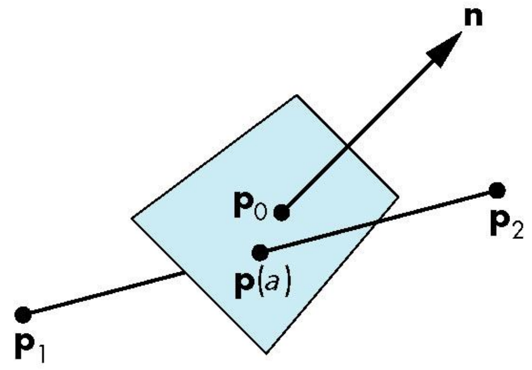
- Can accept/reject as easily as with Cohen-Sutherland
- Using values of α, we do not have to use algorithm recursively as with C-S
- Extends to 3D

Clipping and Normalization

- General clipping in 3D requires intersection of line segments against arbitrary plane
- Example: oblique view



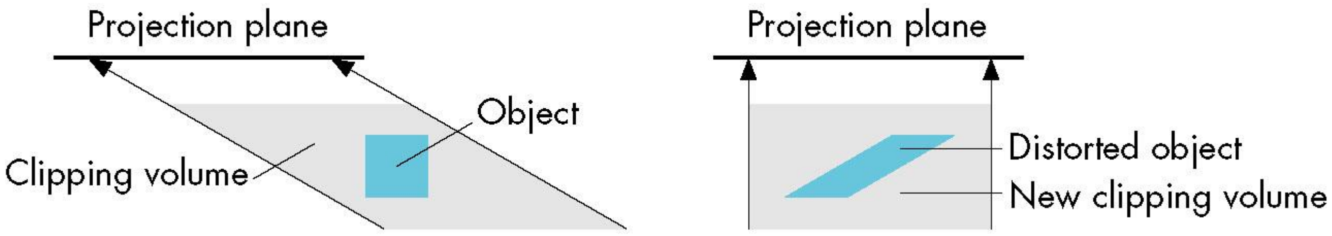
Plane-Line Intersections



$$a = \frac{n \bullet (p_o - p_1)}{n \bullet (p_2 - p_1)}$$

Normalized Form

top view



before normalization after normalization

Normalization is part of viewing (pre clipping) but after normalization, we clip against sides of right parallelepiped

Typical intersection calculation now requires only a floating point subtraction, e.g. is $x > x_{max}$?

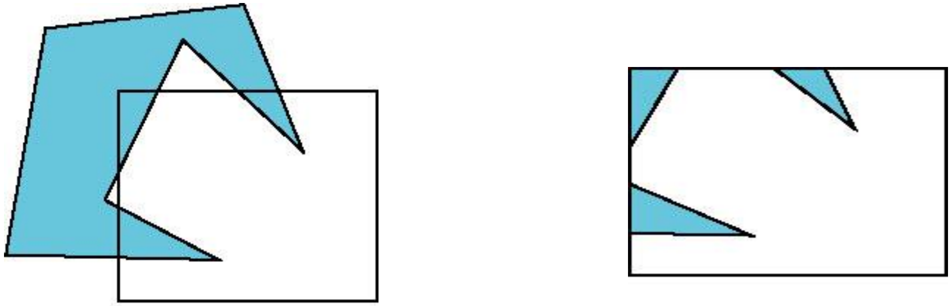
Implementation II

Objectives

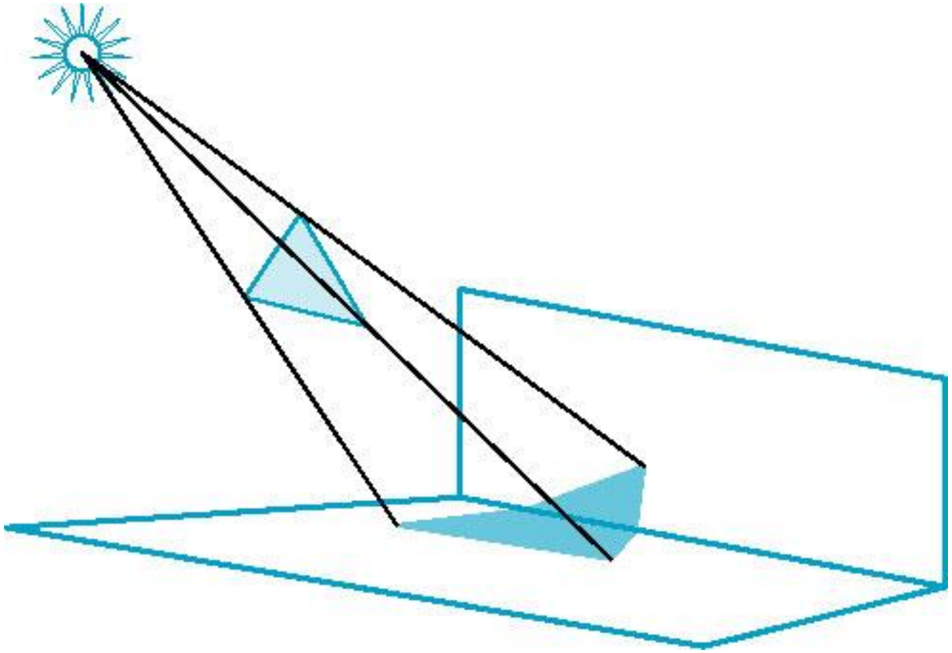
- Introduce clipping algorithms for polygons
- Survey hidden-surface algorithms

Polygon Clipping

- Not as simple as line segment clipping
  - Clipping a line segment yields at most one line segment
  - Clipping a polygon can yield multiple polygons



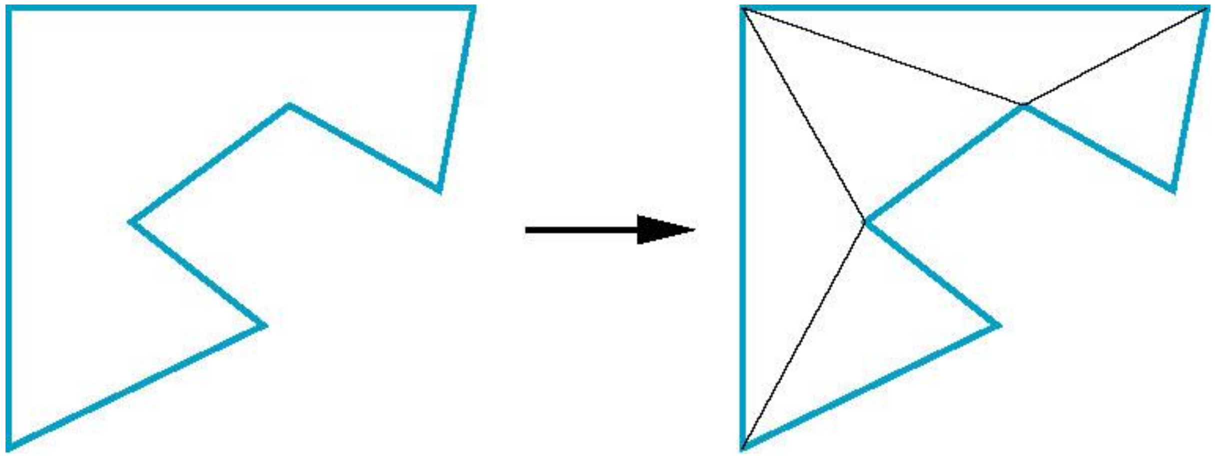
- However, clipping a convex polygon can yield at most one other polygon



Polygon clipping in a shadow generation

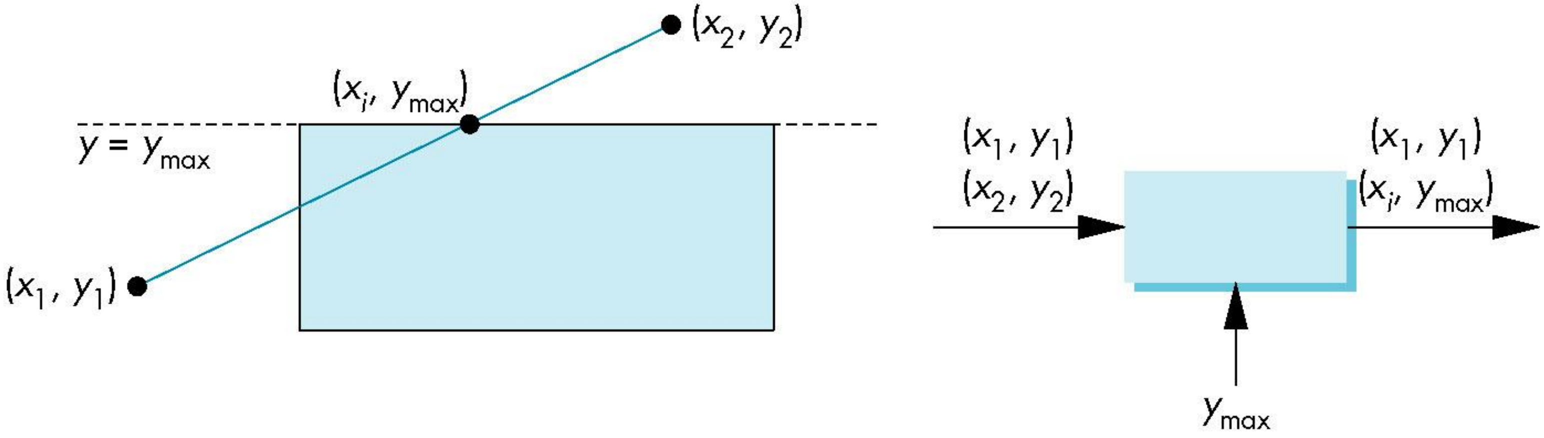
Tessellation and Convexity

- One strategy is to replace nonconvex (*concave*) polygons with a set of triangular polygons (a *tessellation*)
- Also makes fill easier
- Tessellation code in GLU library



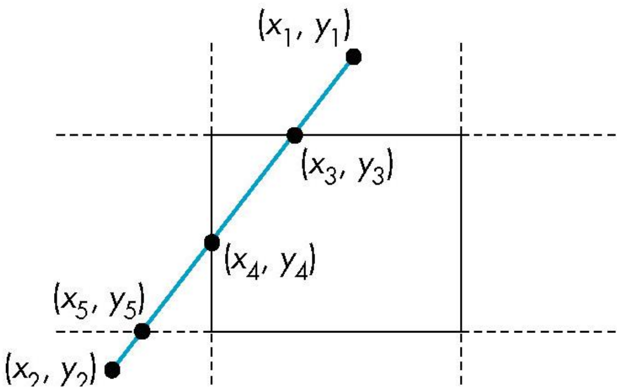
Clipping as a Black Box

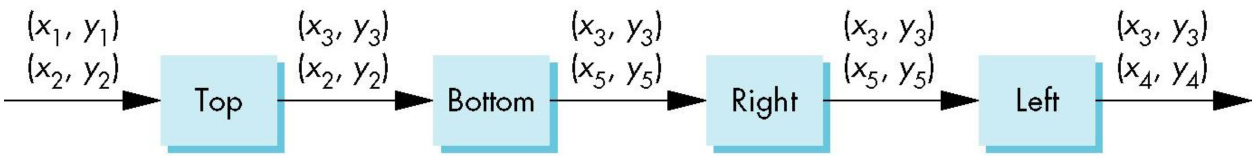
- Can consider line segment clipping as a process that takes in two vertices and produces either no vertices or the vertices of a clipped line segment



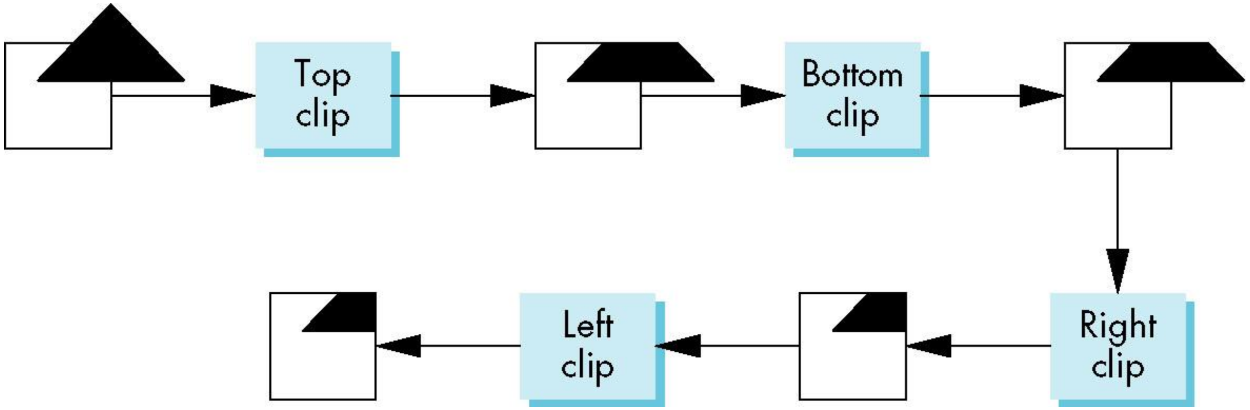
Pipeline Clipping of Line Segments

- Clipping against each side of window is independent of other sides
  - Can use four independent clippers in a pipeline





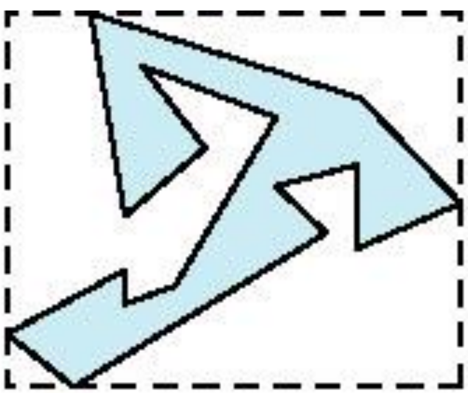
Pipeline Clipping of Polygons



- Three dimensions: add front and back clippers
- Strategy used in SGI Geometry Engine
- Small increase in latency

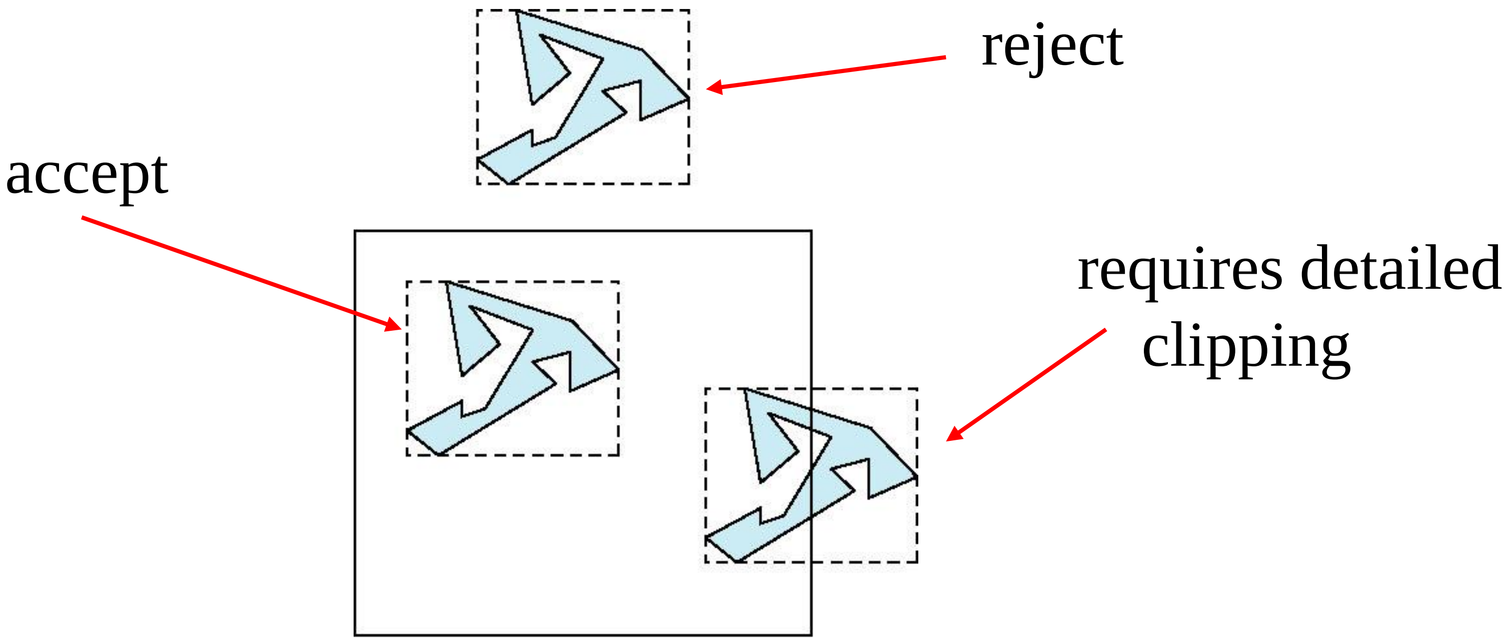
Bounding Boxes

- Rather than doing clipping on a complex polygon, we can use an *axis-aligned bounding box* or *extent*
  - Smallest rectangle aligned with axes that encloses the polygon
  - Simple to compute: max and min of x and y



Bounding boxes

Can usually determine accept/reject based only on bounding box

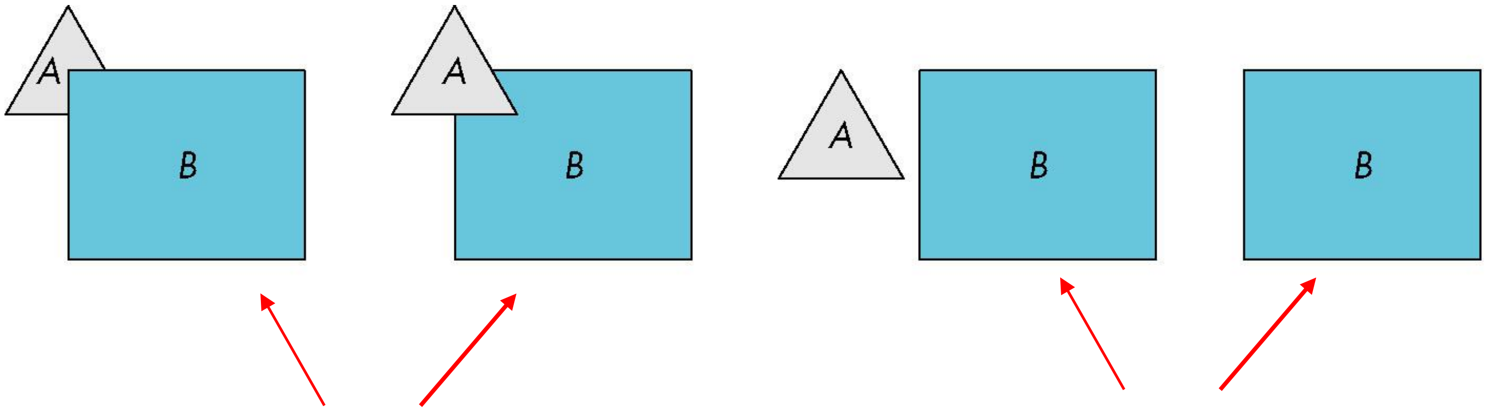


Clipping and Visibility

- Clipping has much in common with hidden-surface removal
- In both cases, we are trying to remove objects that are not visible to the camera
- Often we can use visibility or occlusion testing early in the process to eliminate as many polygons as possible before going through the entire pipeline

Hidden Surface Removal

- Object-space approach: use pairwise testing between polygons (objects)



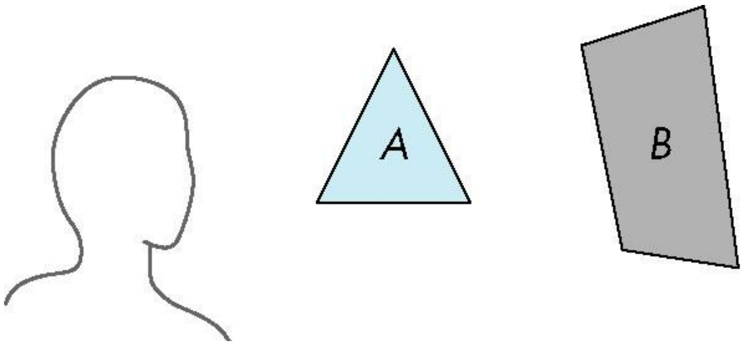
partially obscuring

can draw independently

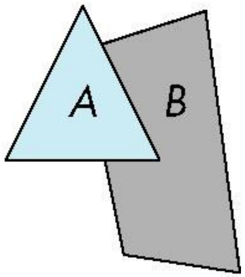
- Worst case complexity $O(n^2)$ for n polygons

Painter's Algorithm

- Render polygons a back to front order so that polygons behind others are simply painted over



B behind A as seen by viewer



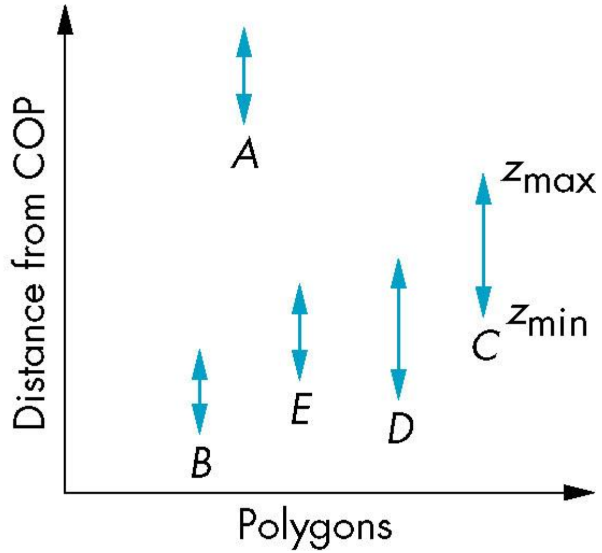
Fill B then A

Depth Sort

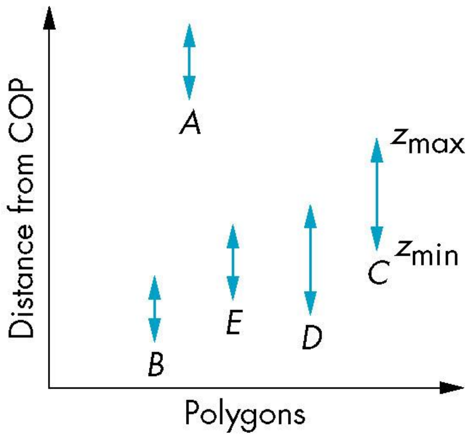
- Requires ordering of polygons first
  - O(n log n) calculation for ordering
  - Not every polygon is either in front or behind all other polygons

- Order polygons and deal with easy cases first, harder later

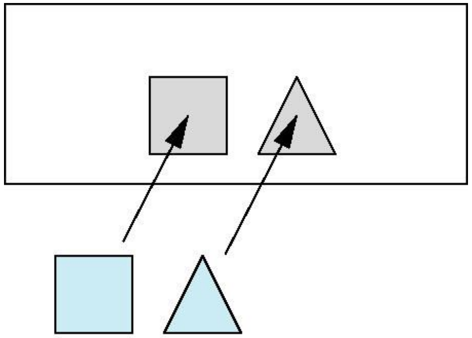
Polygons sorted by distance from COP

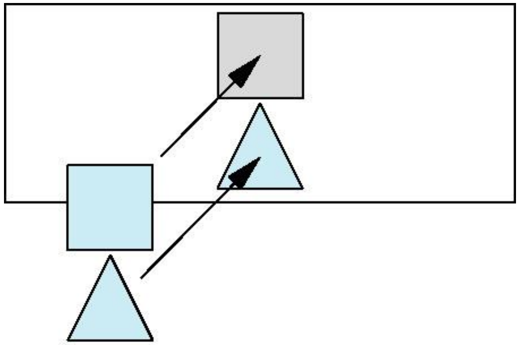


Easy Cases

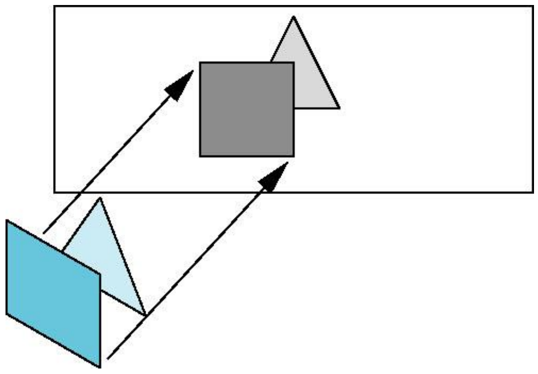


- A lies behind all other polygons
  - –Can render
- Polygons overlap in z but not in either x or y
  - –Can render independently

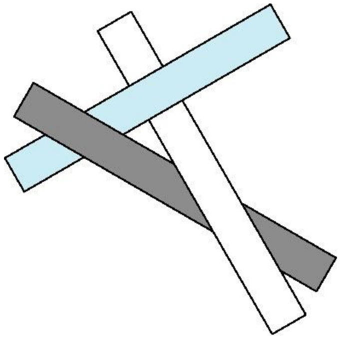




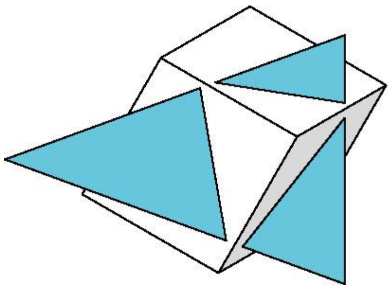
Hard Cases



Overlap in all directions but can one is fully on one side of the other

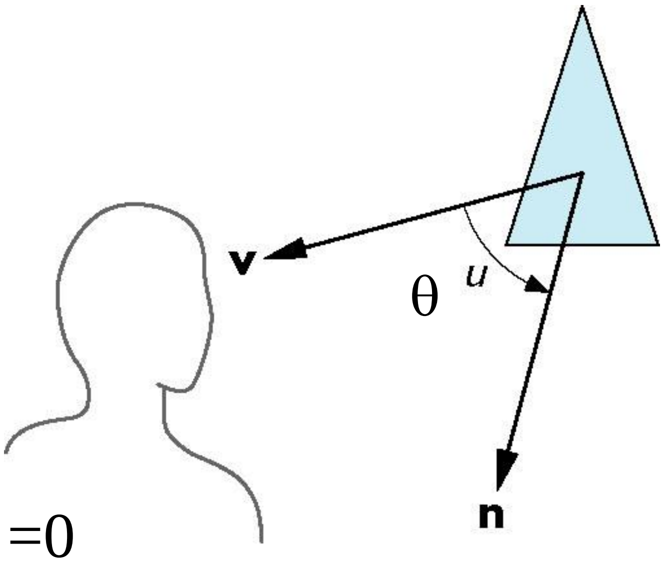


cyclic overlap



penetration

Back-Face Removal (Culling)



•face is visible iff $90 \geq \theta \geq -90$
equivalently $\cos\theta \geq 0$
or $\mathbf{v} \cdot \mathbf{n} \geq 0$

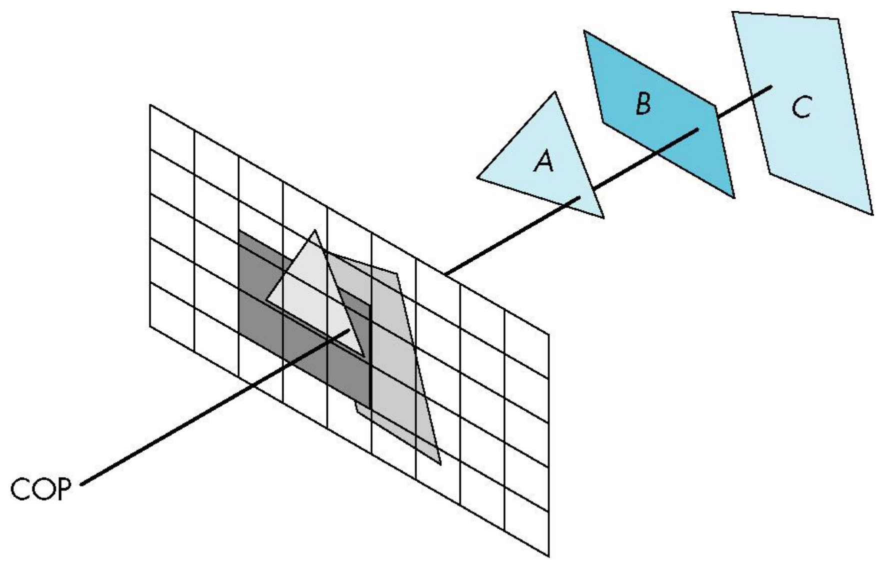
•plane of face has form $ax + by + cz + d = 0$
but after normalization $\mathbf{n} = (\ 0\ 0\ 1\ 0)^T$

•need only test the sign of c

•In OpenGL we can simply enable culling
but may not work correctly if we have nonconvex objects

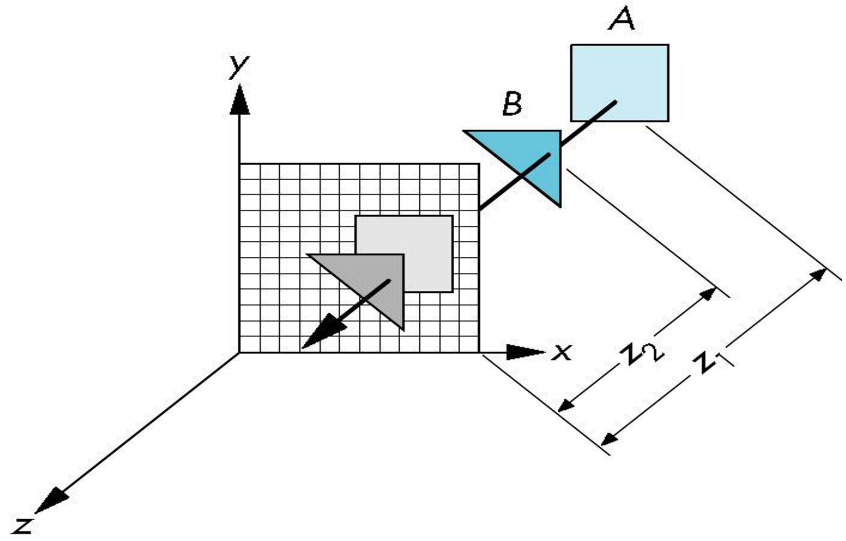
Image Space Approach

- Look at each projector (nm for an n x m frame buffer) and find closest of k polygons
- Complexity O(nmk)
- Ray tracing
- z-buffer



z-Buffer Algorithm

- Use a buffer called the z or depth buffer to store the depth of the closest object at each pixel found so far
- As we render each polygon, compare the depth of each pixel to depth in z buffer
- If less, place shade of pixel in color buffer and update z buffer



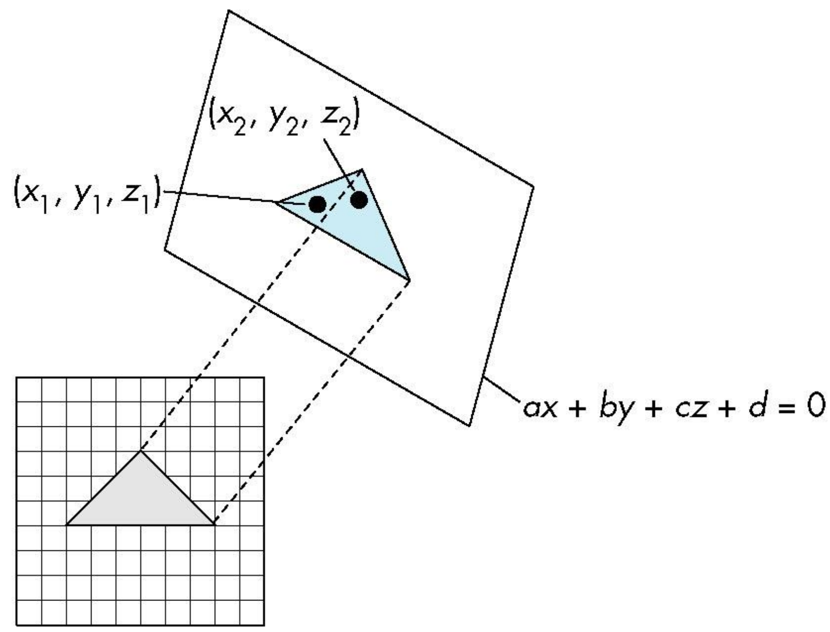
Efficiency

- If we work scan line by scan line as we move across a scan line, the depth changes satisfy $a\Delta x+b\Delta y+c\Delta z=0$

Along scan line

$$\Delta y = 0$$

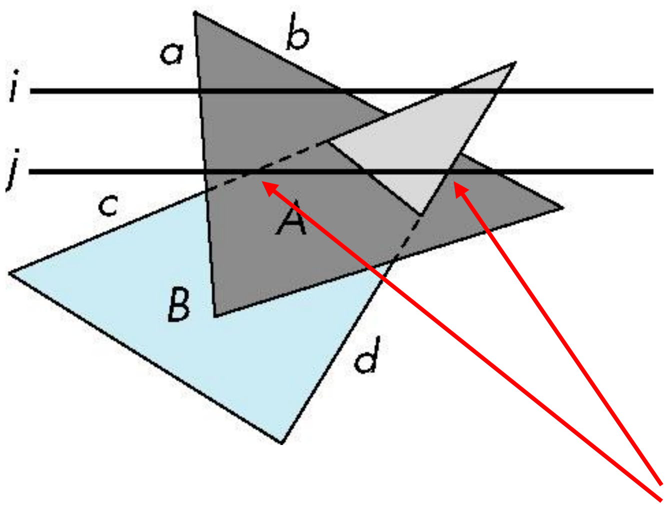
$$\Delta z = -\frac{a}{c}\Delta x$$



In screen space $\Delta x = 1$

Scan-Line Algorithm

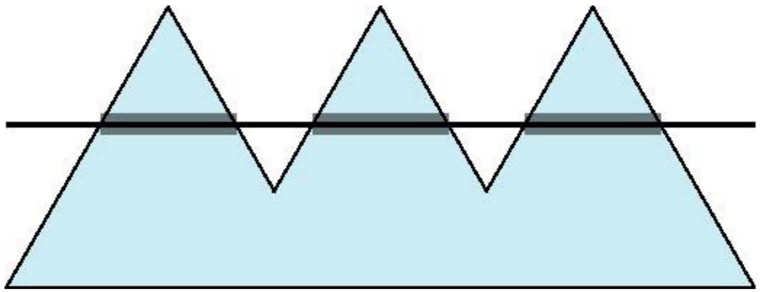
- Can combine shading and hsr through scan line algorithm



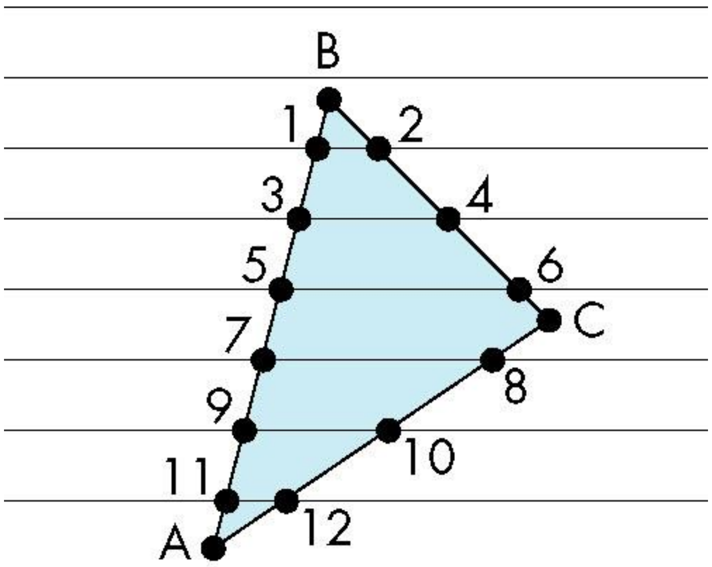
scan line i: no need for depth information, can only be in no or one polygon

scan line j: need depth information only when in more than one polygon

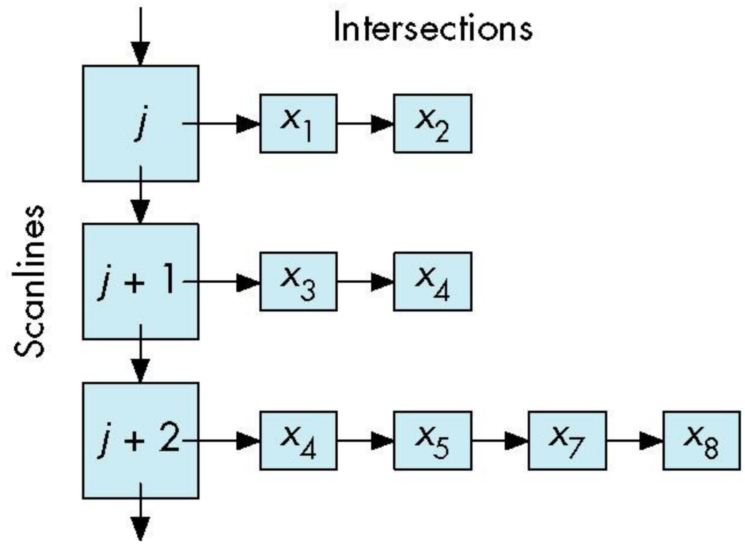
Scan-Line Algorithms



Polygon with spans



Polygon generated by vertex list



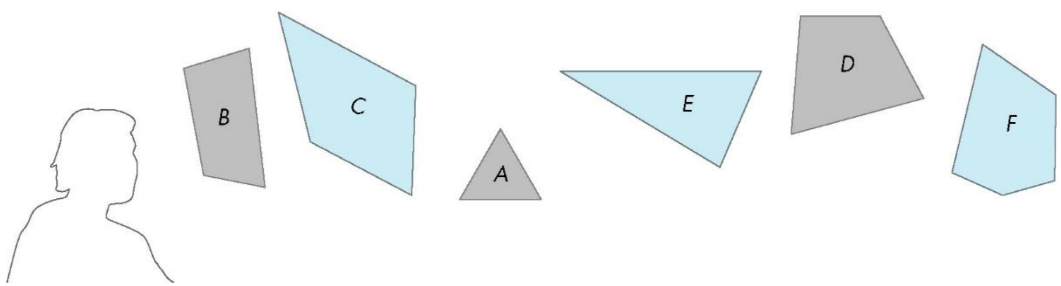
Data structure for *y*-*x* algorithm

Implementation

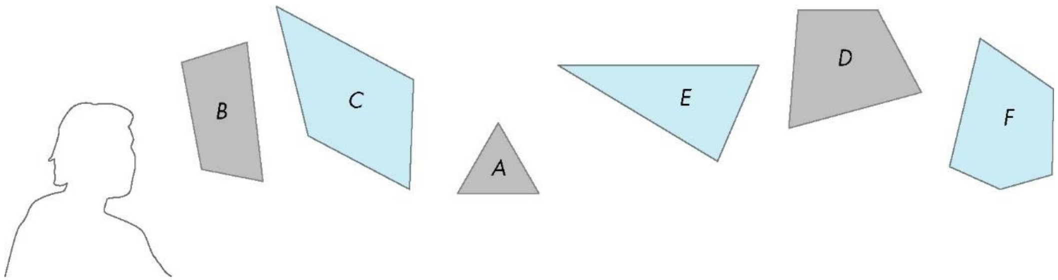
- Need a data structure to store
  - Flag for each polygon (inside/outside)
  - Incremental structure for scan lines that stores which edges are encountered
  - Parameters for planes

Visibility Testing

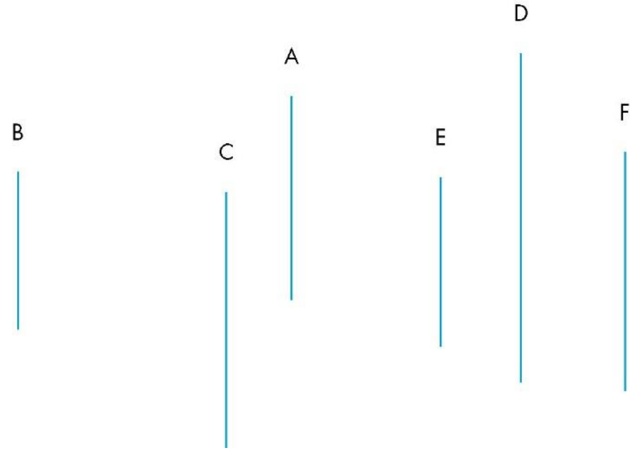
- In many realtime applications, such as games, we want to eliminate as many objects as possible within the application
  - –Reduce burden on pipeline
  - –Reduce traffic on bus
- Partition space with Binary Spatial Partition (BSP) Tree



Simple Example



consider 6 parallel polygons

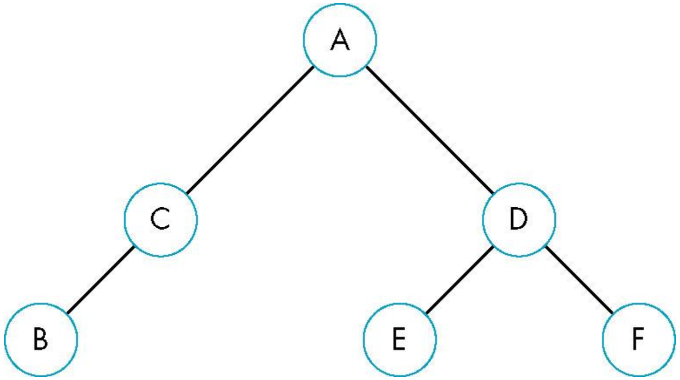


top view

The plane of A separates B and C from D, E and F

BSP Tree

- Can continue recursively
  - –Plane of C separates B from A
  - –Plane of D separates E and F
- Can put this information in a BSP tree
  - –Use for visibility and occlusion testing



Implementation III

Objectives

- Survey Line Drawing Algorithms
  - DDA
  - Bresenham

Rasterization

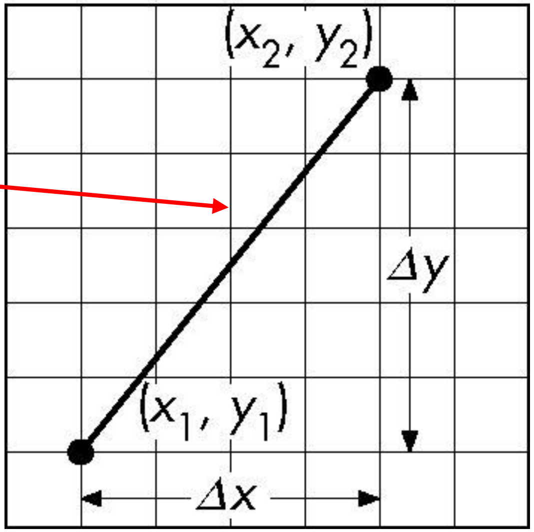
- Rasterization (scan conversion)
  - Determine which pixels that are inside primitive specified by a set of vertices
  - Produces a set of fragments
  - Fragments have a location (pixel location) and other attributes such color and texture coordinates that are determined by interpolating values at vertices
- Pixel colors determined later using color, texture, and other vertex properties

Scan Conversion of Line Segments

- Start with line segment in window coordinates with integer values for endpoints
- Assume implementation has a **`write_pixel`** function

y = mx + h

$$m = \frac{\Delta y}{\Delta x}$$



DDA Algorithm

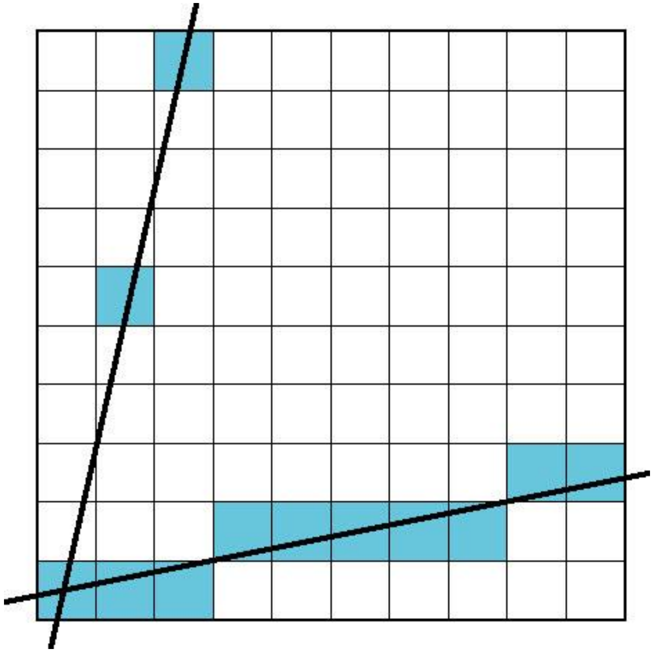
- Digital Differential Analyzer
  - DDA was a mechanical device for numerical solution of differential equations
  - Line y=mx+ h satisfies differential equation

    $dy/dx = m = \Delta y/\Delta x = y_2\text{-}y_1/x_2\text{-}x_1$
- Along scan line $\Delta x = 1$

```
For(x=x1; x<=x2,ix++) {
    y+=m;
   write_pixel(x, round(y), line_color)
}
```

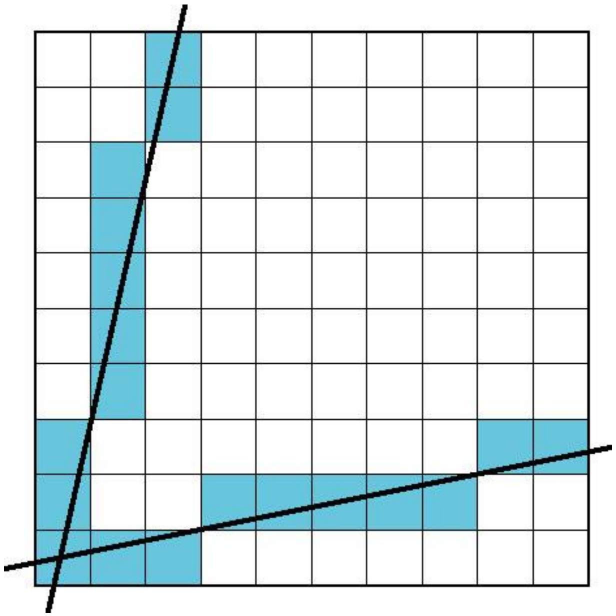
Problem

- DDA = for each x plot pixel at closest y
  - –Problems for steep lines



Using Symmetry

- Use for $1 \geq m \geq 0$
- For $m > 1$, swap role of x and y
  - For each y, plot closest x

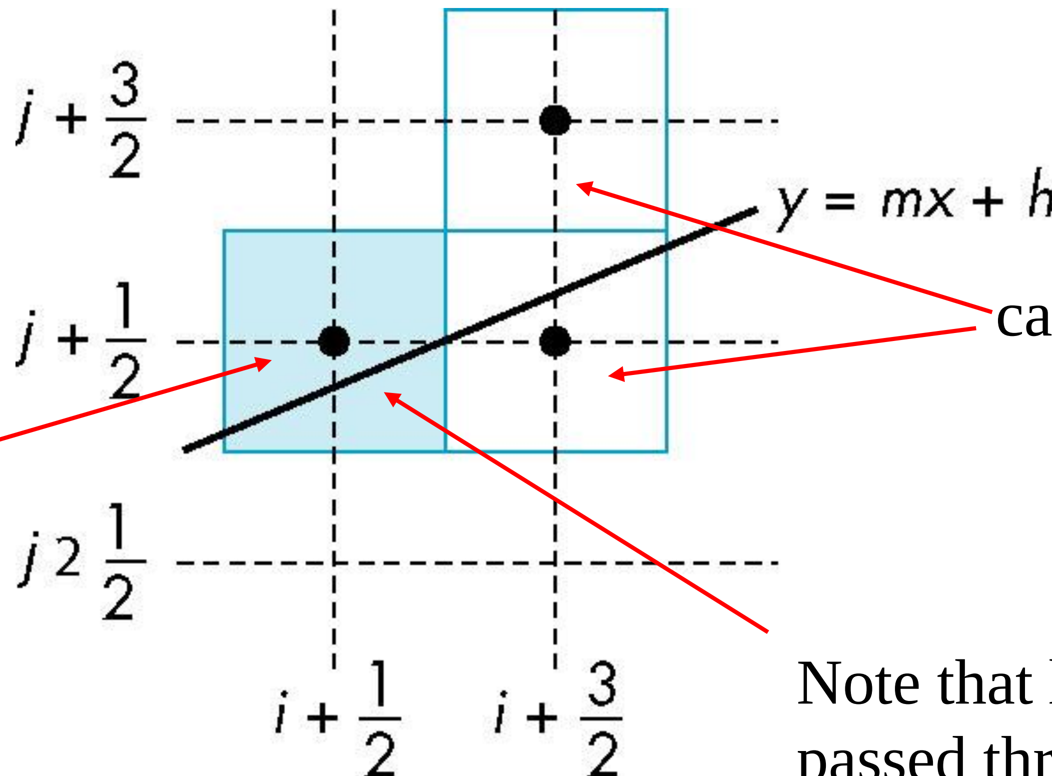


Bresenham's Algorithm

- DDA requires one floating point addition per step
- We can eliminate all fp through Bresenham's algorithm
- Consider only $1 \geq m \geq 0$
  - Other cases by symmetry
- Assume pixel centers are at half integers
- If we start at a pixel that has been written, there are only two candidates for the next pixel to be written into the frame buffer

Candidate Pixels

$1 \geq m \geq 0$



candidates

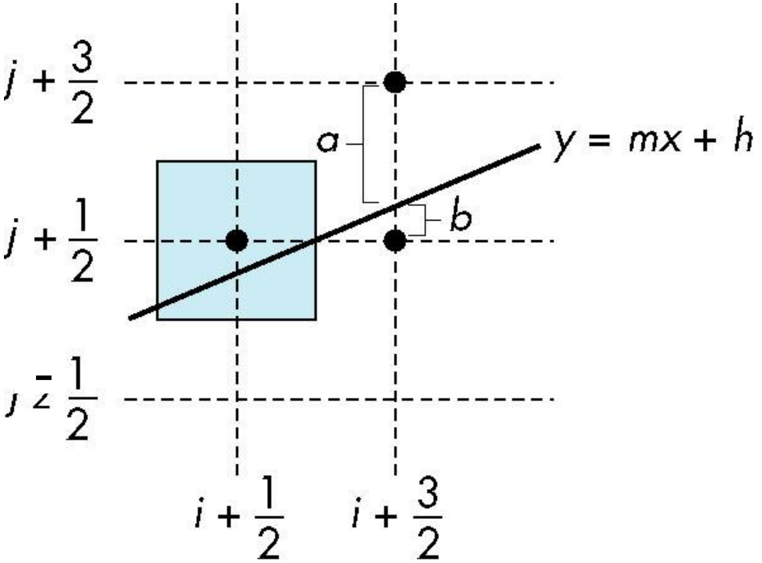
last pixel

Note that line could have passed through any part of this pixel

Decision Variable

$d = \Delta x(b-a)$

d is an integer
d > 0 use upper pixel
d < 0 use lower pixel



Incremental Form

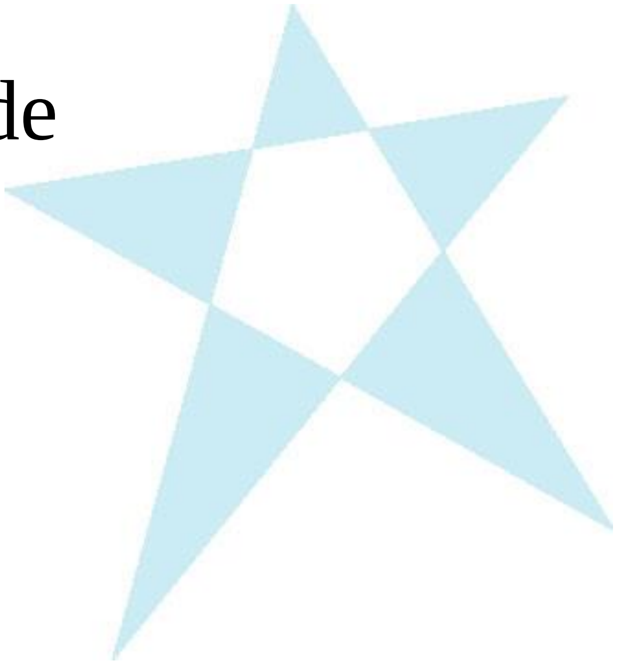
- More efficient if we look at d_k, the value of the decision variable at $x = k$

$$d_{k+1} = d_k - 2Dy, \quad \text{if } d_k < 0$$
$$d_{k+1} = d_k - 2(Dy - Dx), \quad \text{otherwise}$$

- For each x, we need do only an integer addition and a test
- Single instruction on graphics chips

Polygon Scan Conversion

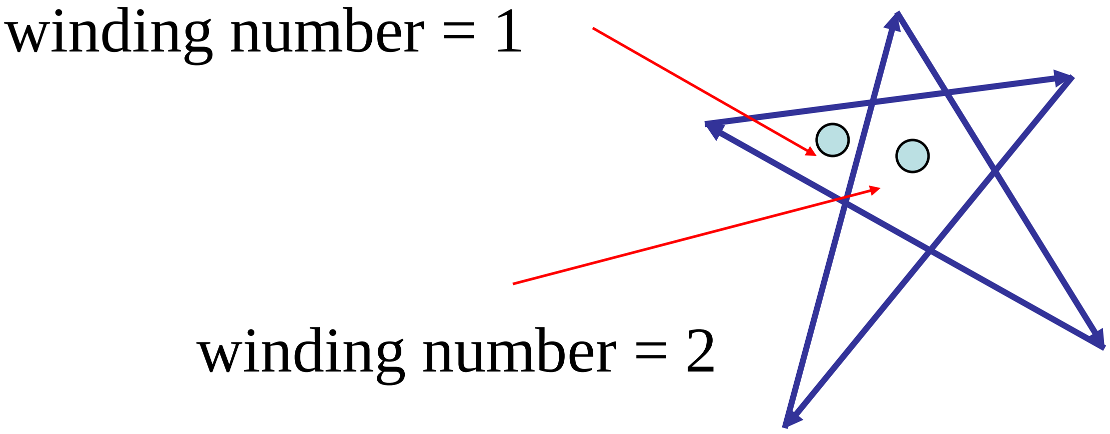
- Scan Conversion = Fill
- How to tell inside from outside
  - –Convex easy
  - –Nonsimple difficult
  - –Odd-even test
    - Count edge crossings
  - –Winding number



odd-even fill

Winding Number

- Count clockwise encirclements of point



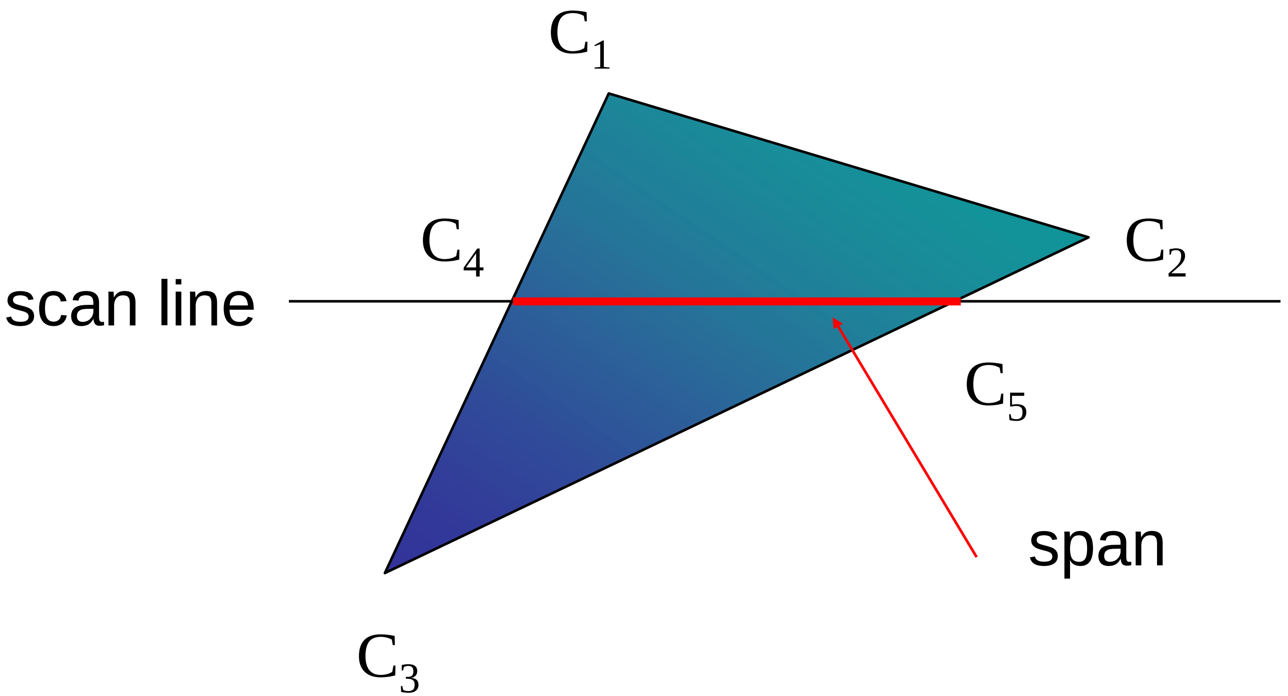
- Alternate definition of inside: inside if winding number $\neq 0$

Filling in the Frame Buffer

- Fill at end of pipeline
  - Convex Polygons only
  - Nonconvex polygons assumed to have been tessellated
  - Shades (colors) have been computed for vertices (Gouraud shading)
  - Combine with z-buffer algorithm
    - March across scan lines interpolating shades
    - Incremental work small

Using Interpolation

C_1 C_2 C_3 specified by **`glColor`** or by vertex shading
C_4 determined by interpolating between C_1 and C_2
C_5 determined by interpolating between C_2 and C_3
interpolate between C_4 and C_5 along span



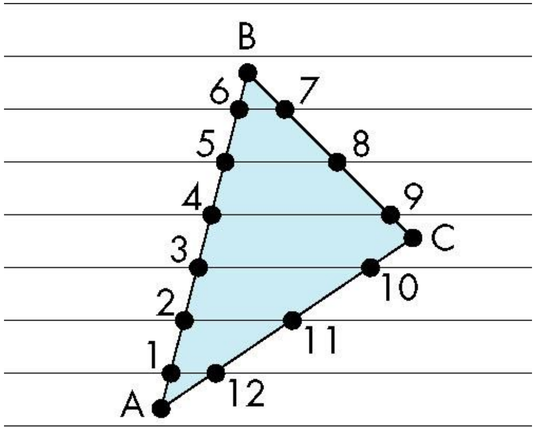
Flood Fill

- Fill can be done recursively if we know a seed point located inside (WHITE)
- Scan convert edges into buffer in edge/inside color (BLACK)

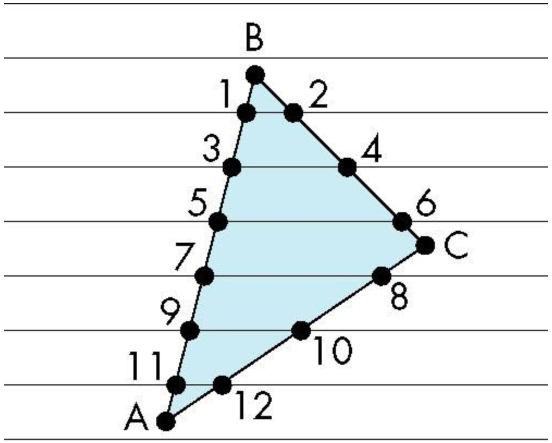
```
flood_fill(int x, int y) {
    if(read_pixel(x,y)= = WHITE) {
        write_pixel(x,y,BLACK);
        flood_fill(x-1, y);
        flood_fill(x+1, y);
        flood_fill(x, y+1);
        flood_fill(x, y-1);
} }
```

Scan Line Fill

- Can also fill by maintaining a data structure of all intersections of polygons with scan lines
  - Sort by scan line
  - Fill each span

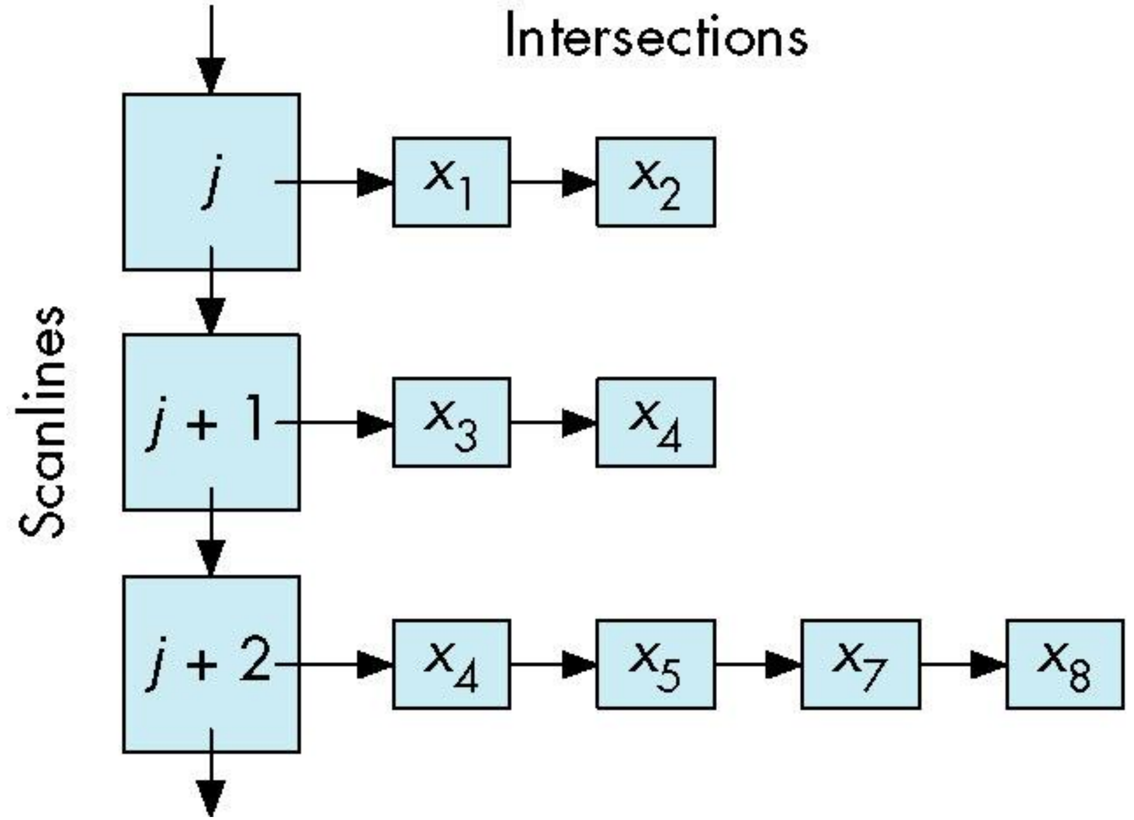


vertex order generated by vertex list



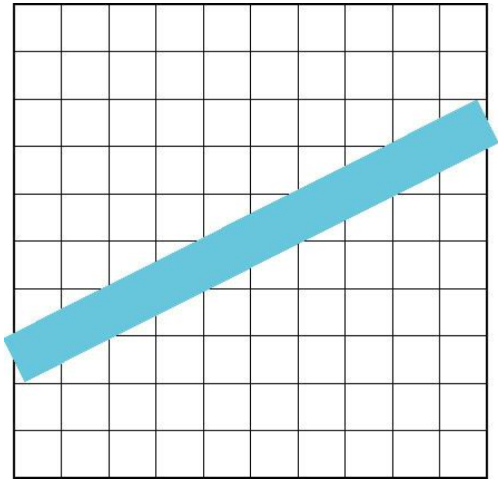
desired order

Data Structure



Aliasing

- Ideal rasterized line should be 1 pixel wide

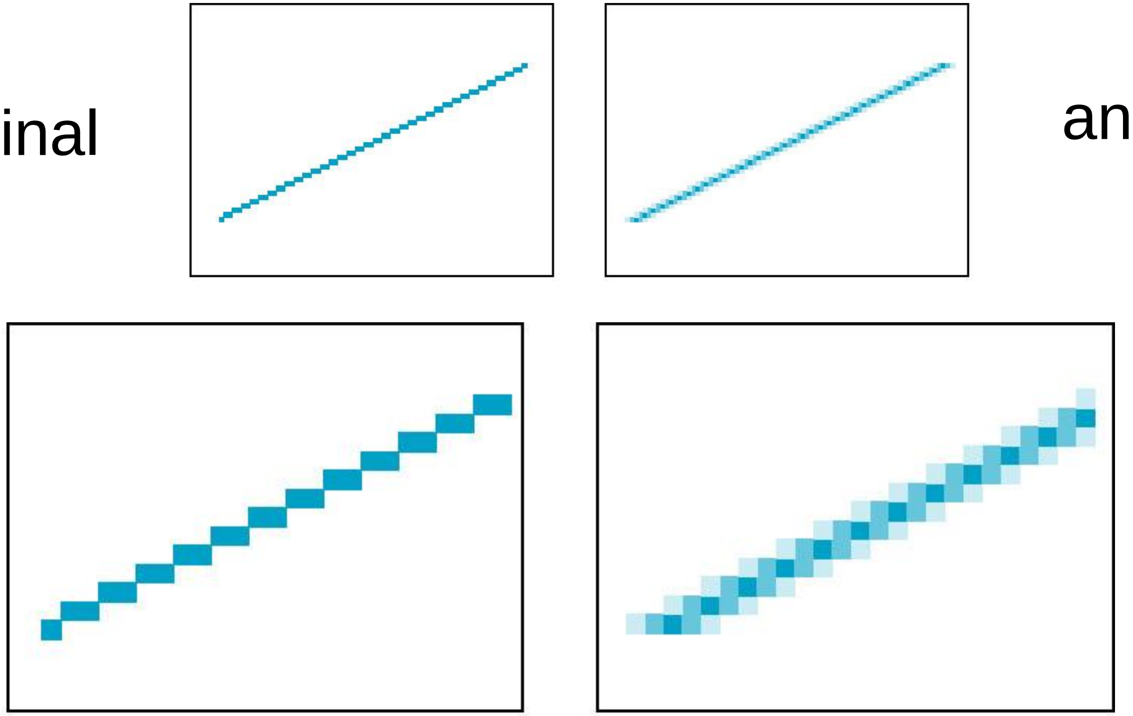


- Choosing best y for each x (or visa versa) produces aliased raster lines

Antialiasing by Area Averaging

- Color multiple pixels for each x depending on coverage by ideal line

original

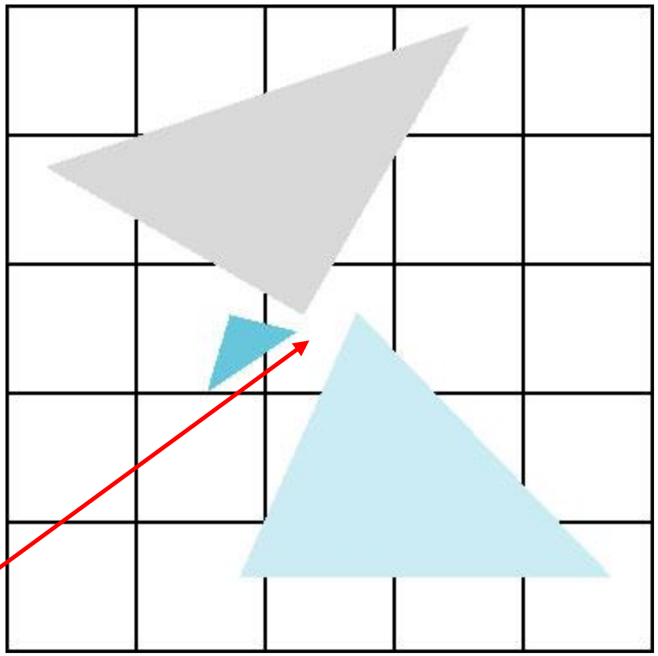


antialiased

magnified

Polygon Aliasing

- Aliasing problems can be serious for polygons
  - Jaggedness of edges
  - Small polygons neglected
  - Need compositing so color of one polygon does not totally determine color of pixel



All three polygons should contribute to color