Hochiminh city University of Technology
Faculty of Computer Science and Engineering

COMPUTER GRAPHICS

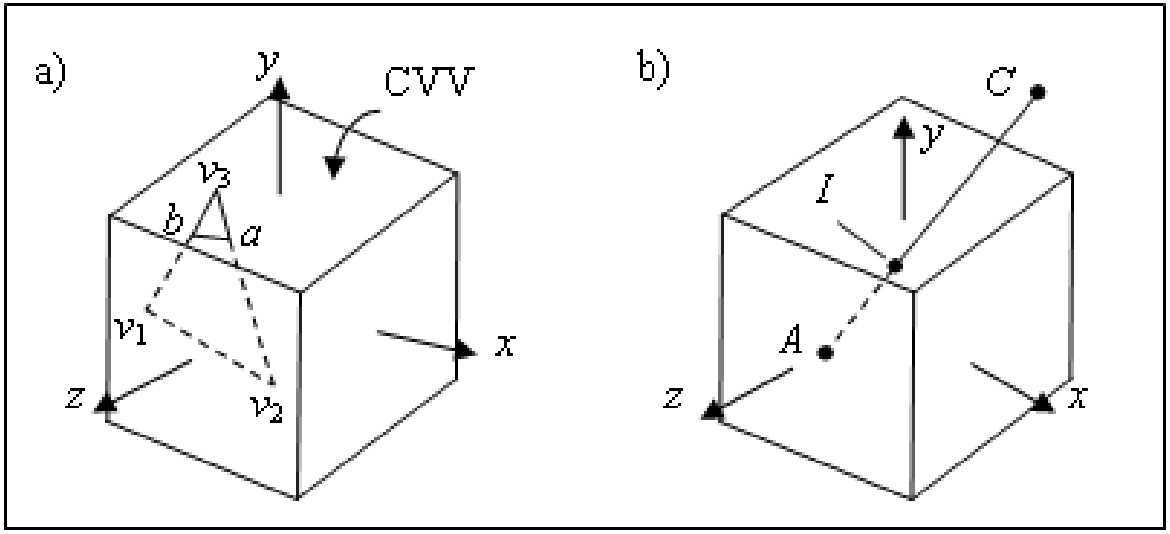
CHAPTER 07:

Rasterization

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

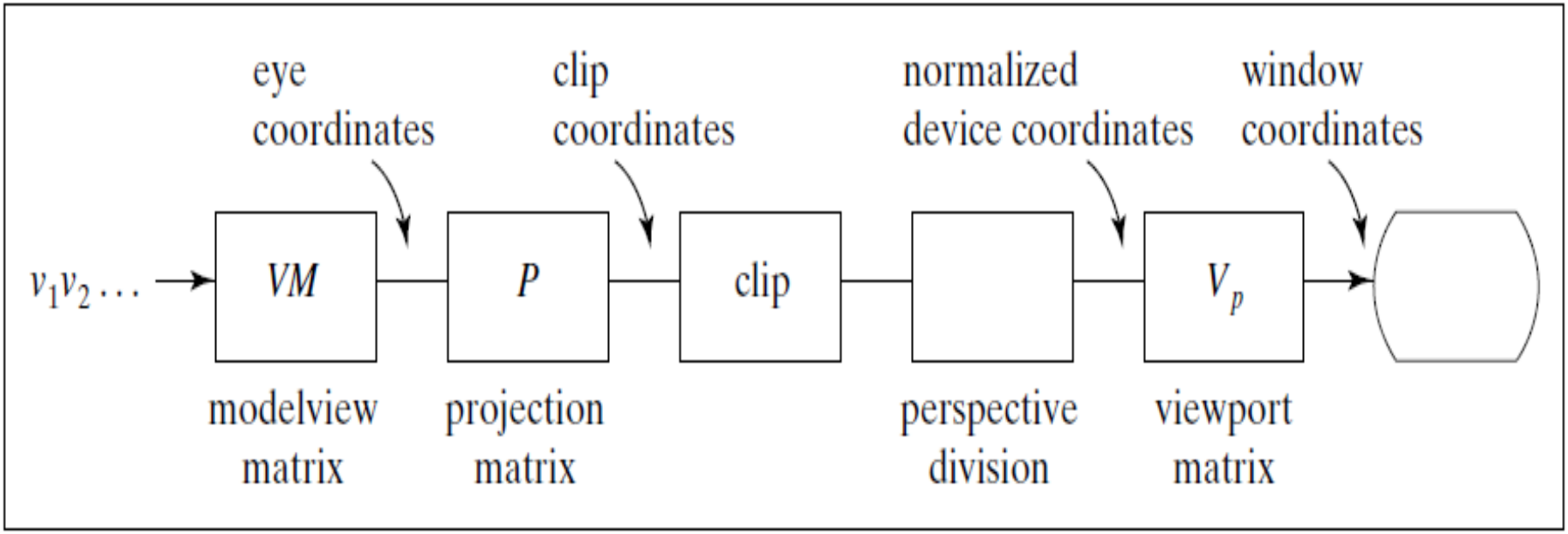
Overview



Required Tasks

- Modeling
- Geometric processing
  - Projection
  - Primitive assembly
  - Clipping
  - Shading
- Rasterization
- Fragment Processing
  - Texture, Blending color

Required Tasks

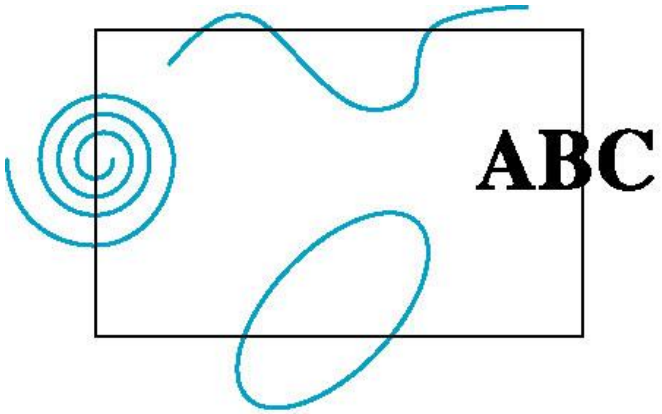


Rasterization Meta Algorithms

- 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

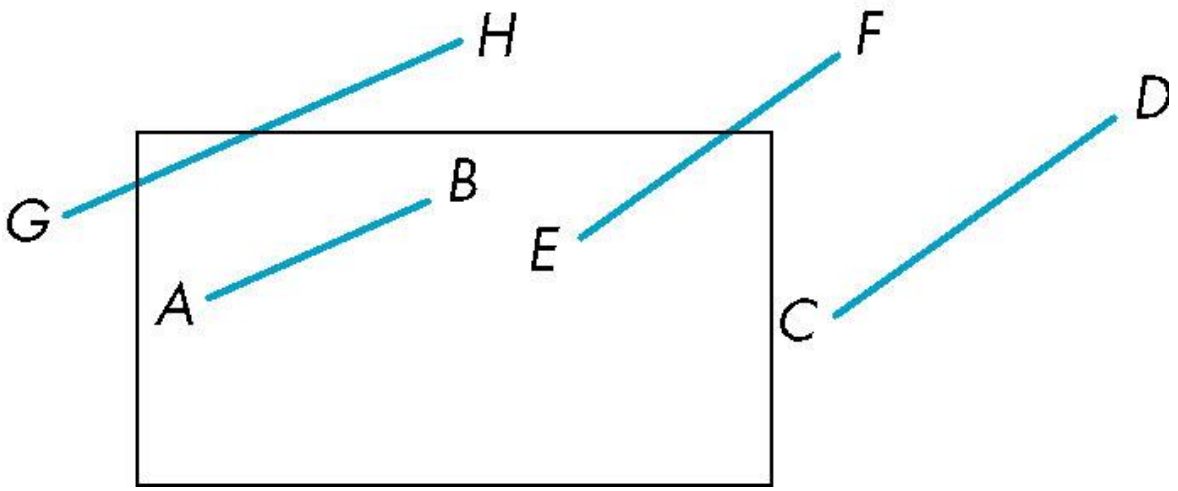
Clipping

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



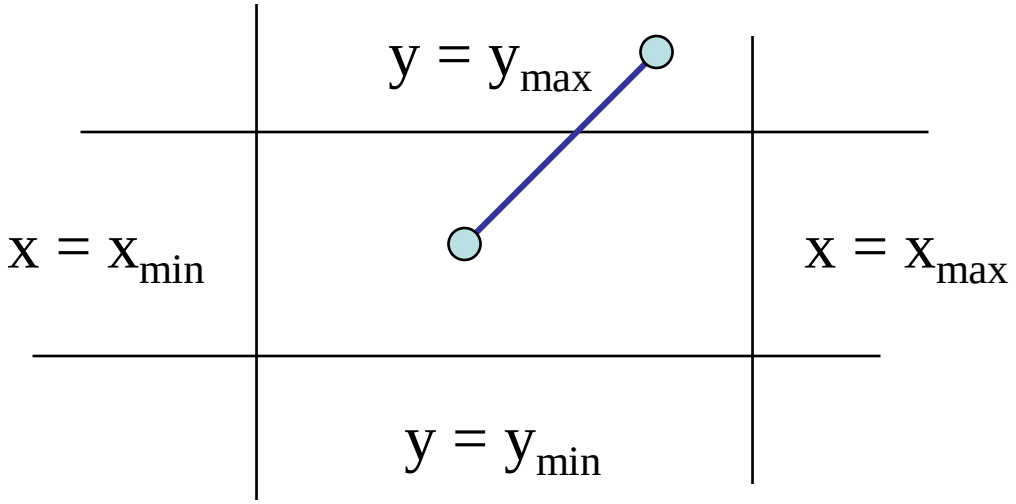
Clipping 2D Line Segments

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



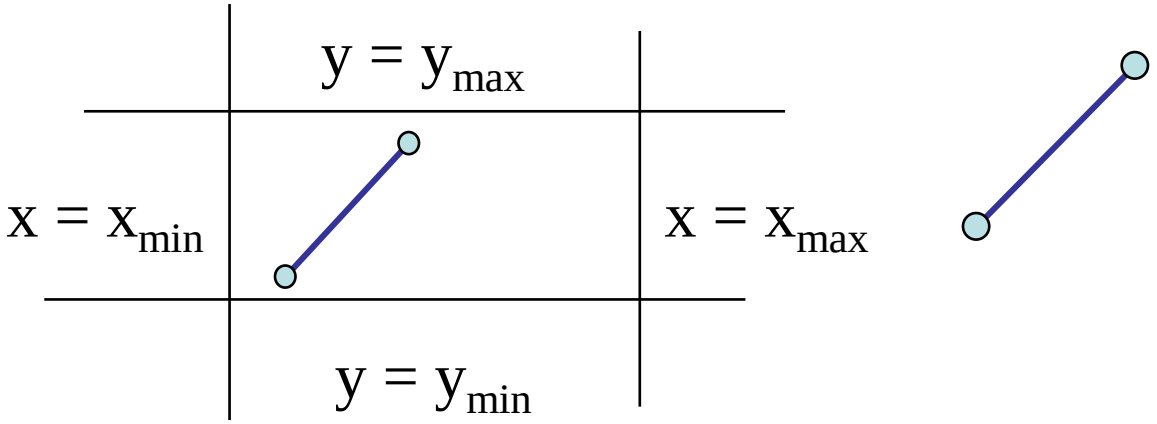
Cohen-Sutherland Algorithm

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



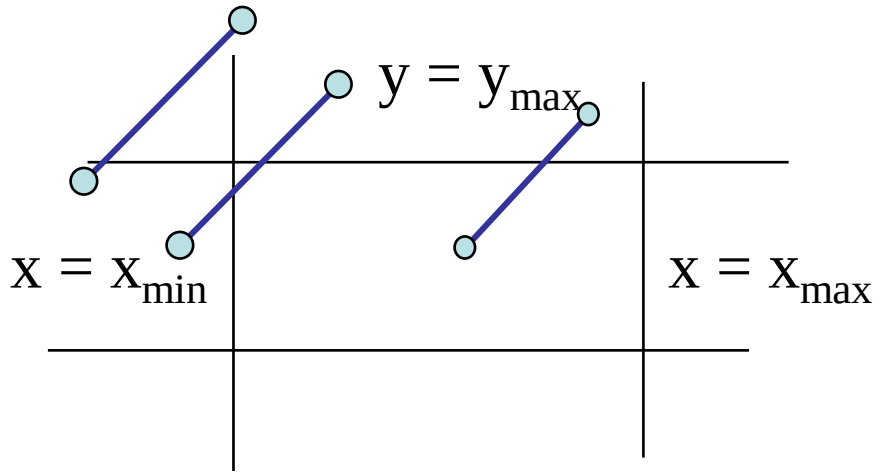
The Cases

- 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

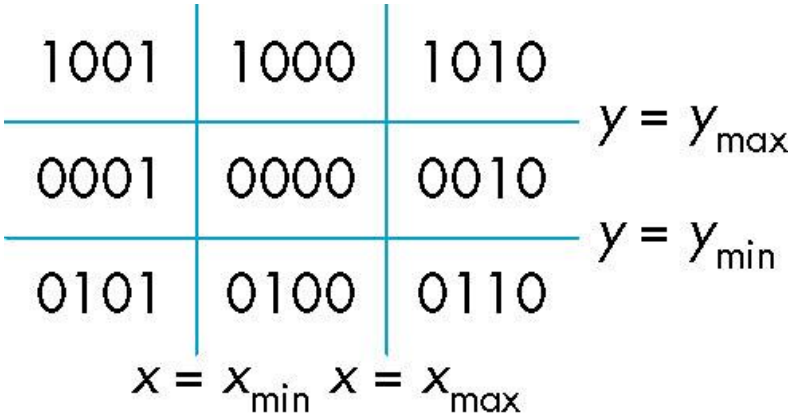


Defining Outcodes

- For each endpoint, define an outcode

$$b_0 b_1 b_2 b_3$$

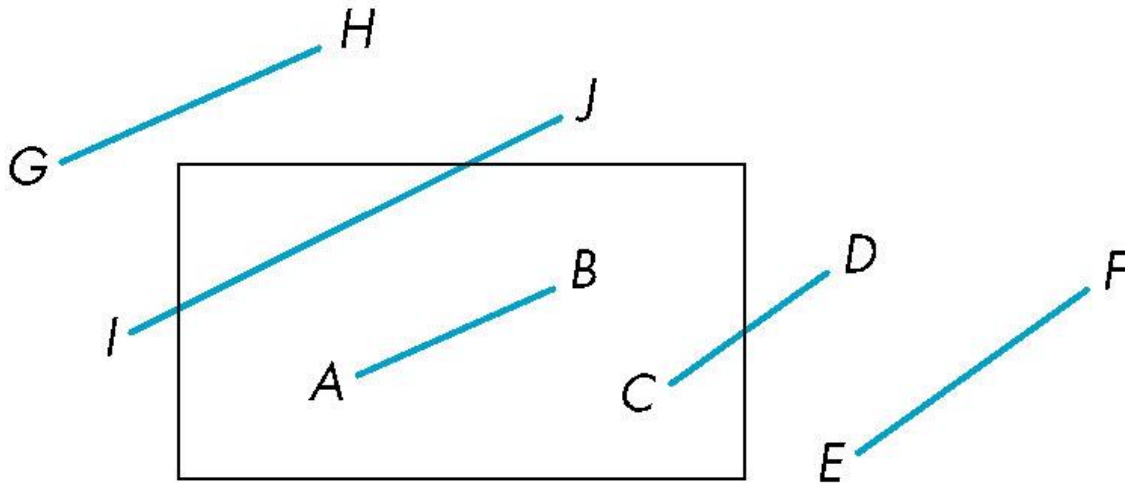
$b_0 = 1$ if $y > y_{max}$, 0 otherwise
$b_1 = 1$ if $y < y_{min}$, 0 otherwise
$b_2 = 1$ if $x > x_{max}$, 0 otherwise
$b_3 = 1$ if $x < x_{min}$, 0 otherwise



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

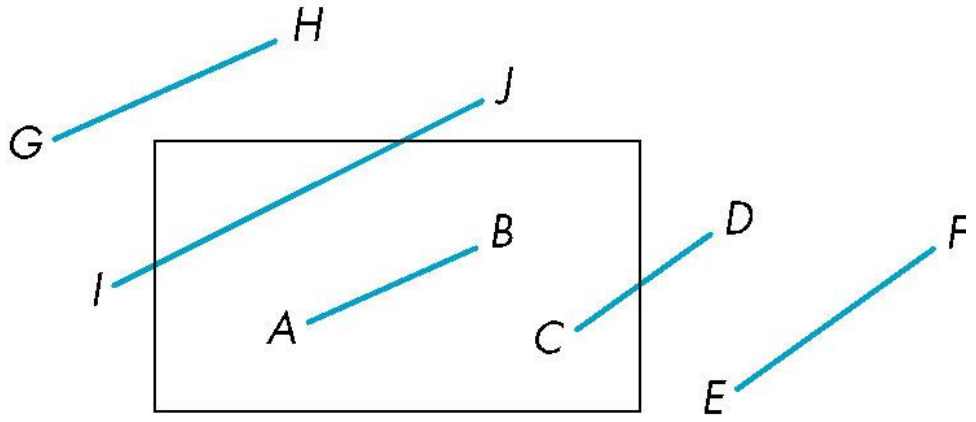
Using Outcodes

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



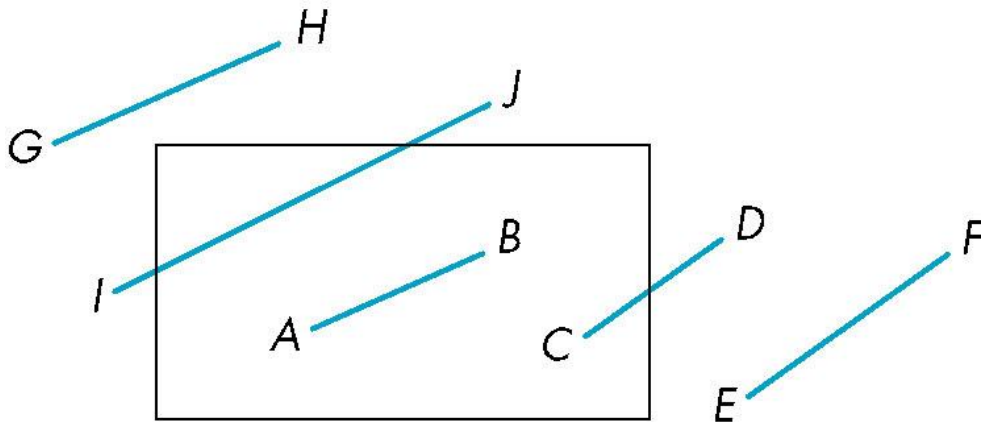
Using Outcodes

- CD: outcode (C) = 0 (0000), outcode(D) (0010) ≠ 0
  - Compute intersection
  - Location of 1 in outcode(D) determines which edge to intersect with
  - Note if there were a segment from C to a point in a region with 2 ones in outcode, we might have to do two interesections



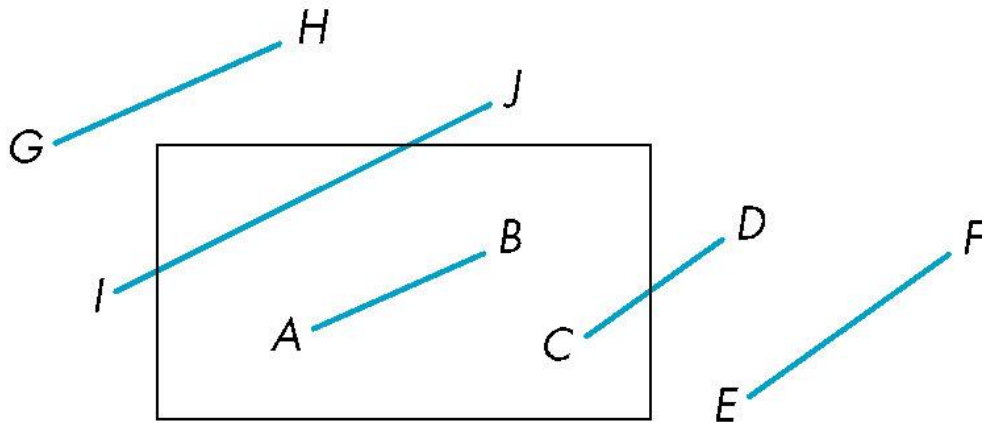
Using Outcodes

- EF: outcode(E) (0010) logically ANDed with outcode(F) (0010) (bitwise) ≠ 0
  - Both outcodes have a 1 bit in the same place
  - Line segment is outside of corresponding side of clipping window
  - reject



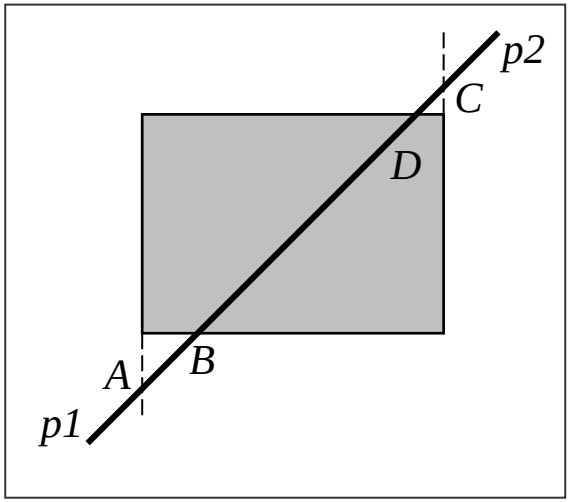
Using Outcodes

- GH and IJ: same outcodes, neither zero but logical AND yields zero. G, I(0001), H, J(1000)
- Shorten line segment by intersecting with one of sides of window
- Compute outcode of intersection (new endpoint of shortened line segment)
- Reexecute algorithm



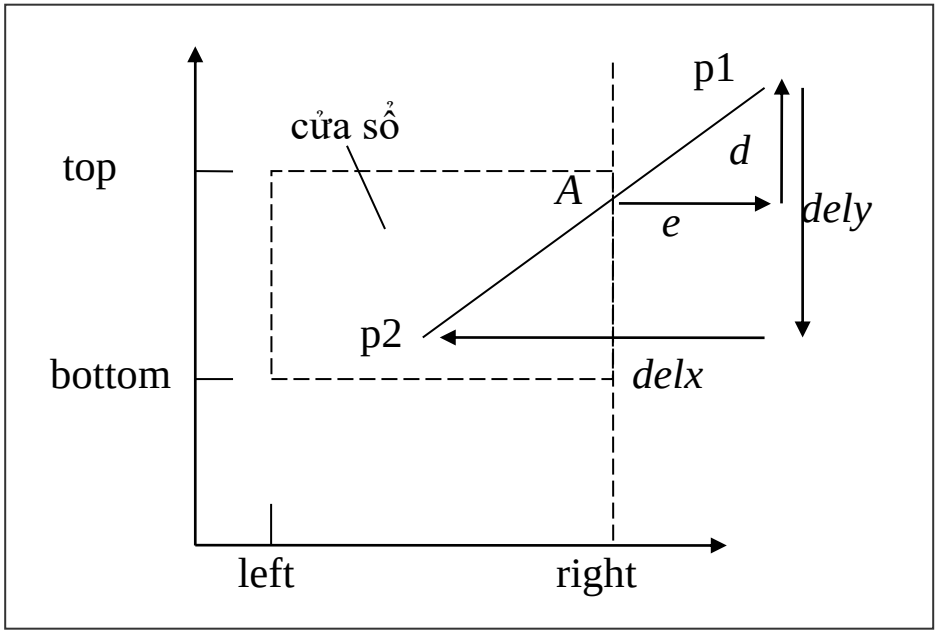
Using Outcodes

```
int clipSegment(Point2& p1, Point2& p2, RealRect W)
{
      do{
         if (trivial accept)  return 1;
         if (trivial reject)    return 0;
         if (p1 nằm ngoài)
         {
             if (p1 nằm bên trên) cắt xén p1 với cạnh trên
             else if (p1 nằm bên dưới) cắt xén p1 với cạnh dưới
             else if (p1 nằm phải) cắt xén p1 với cạnh phải
             else if (p1 nằm trái) cắt xén p1 với cạnh trái
          }
          else        //p2 nằm ngoài
          {
             if (p2 nằm bên trên) cắt xén p2 với cạnh trên
             else if (p2 nằm bên dưới) cắt xén p2 với cạnh dưới
             else if (p2 nằm phải) cắt xén p2 với cạnh phải
             else if (p2 nằm trái) cắt xén p2 với cạnh trái
          }
      }while(1);
}
```



Using Outcodes

$$\frac{d}{dely} = \frac{e}{delx}$$



e = p1.x - W.right;
delx = p2.x - p1.x;
dely = p2.y - p1.y;

p1.y = p1.y + (W.right - p1.x)*dely/delx

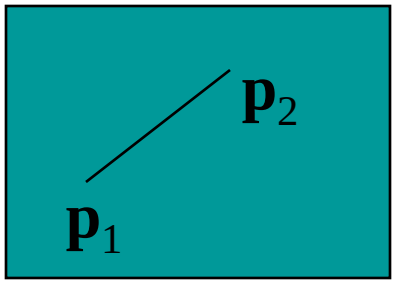
Efficiency

- 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

Liang-Barsky Clipping

- Consider the parametric form of a line segment

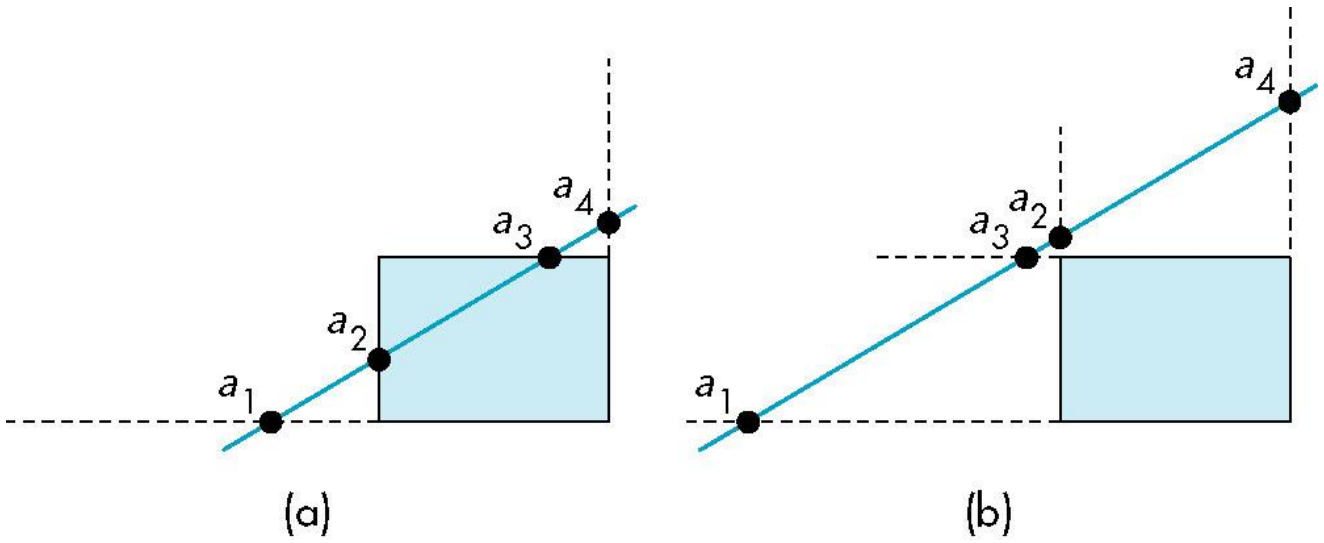
$$\mathbf{p}(\alpha) = (1-\alpha)\mathbf{p}_1 + \alpha\mathbf{p}_2 \quad 1 \geq \alpha \geq 0$$



- 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): $1>\alpha_4 > \alpha_3 > \alpha_2 > \alpha_1>0$
  - Intersect right, top, left, bottom: shorten
- In (b): $1>\alpha_4 > \alpha_2 > \alpha_3 > \alpha_1>0$
  - Intersect right, left, top, bottom: reject



Liang-Barsky Clipping

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

$\mathbf{x}(\alpha) = (1- \alpha)\mathbf{x}_1 + \alpha\, \mathbf{x}_2$
$\mathbf{y}(\alpha) = (1- \alpha)\mathbf{y}_1 + \alpha\, \mathbf{y}_2$

$$\alpha = \frac{y_{\max} - y_1}{y_2 - y_1}$$

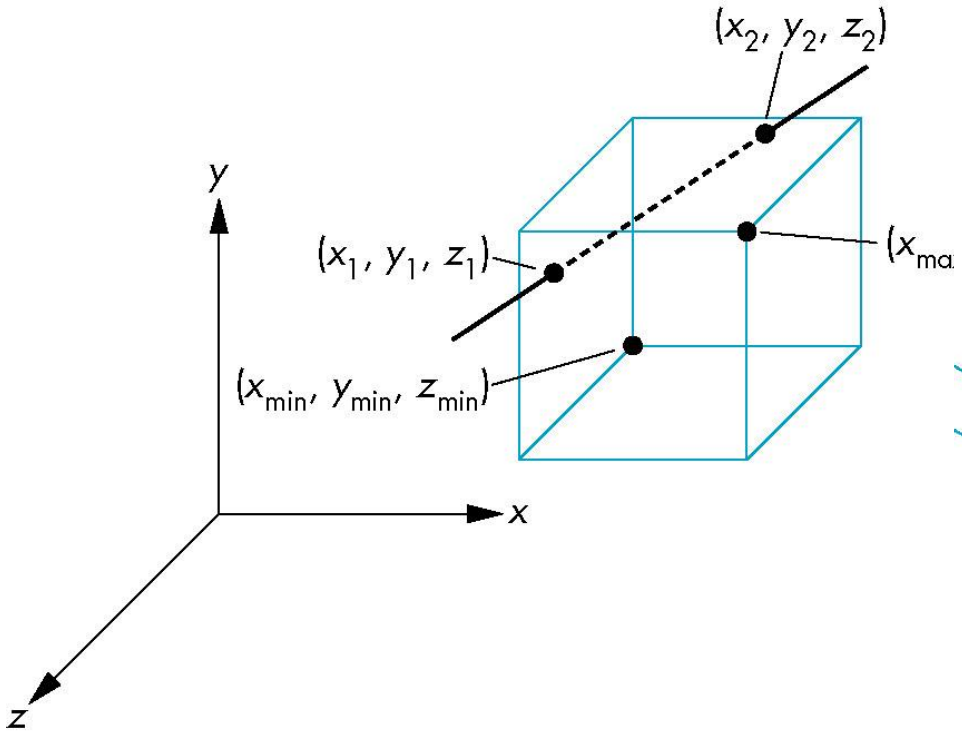
$$\alpha(y_2 - y_1) = \alpha\Delta y = y_{\max} - y_1 = \Delta y_{\max}$$

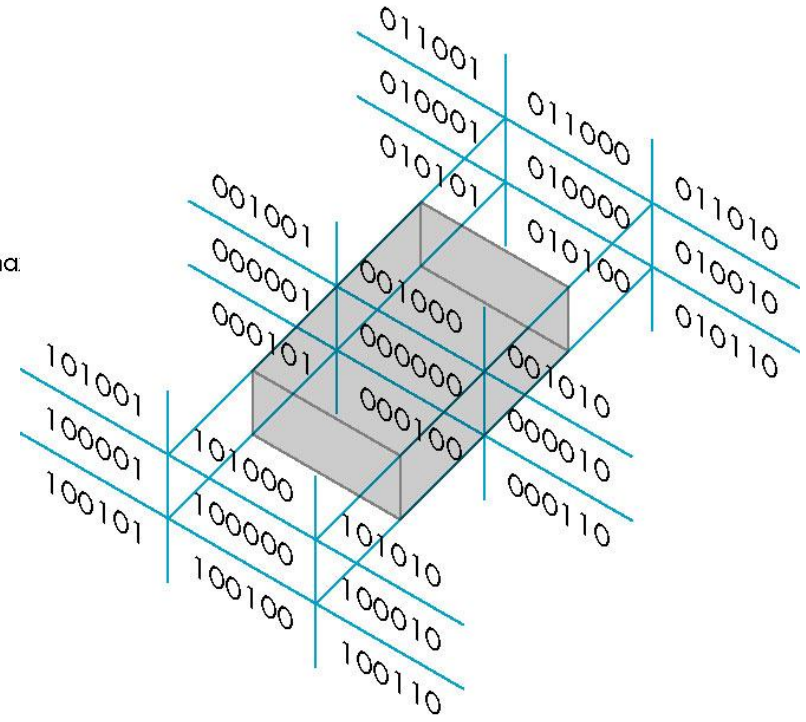
Advantages

- 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

Cohen Sutherland in 3D

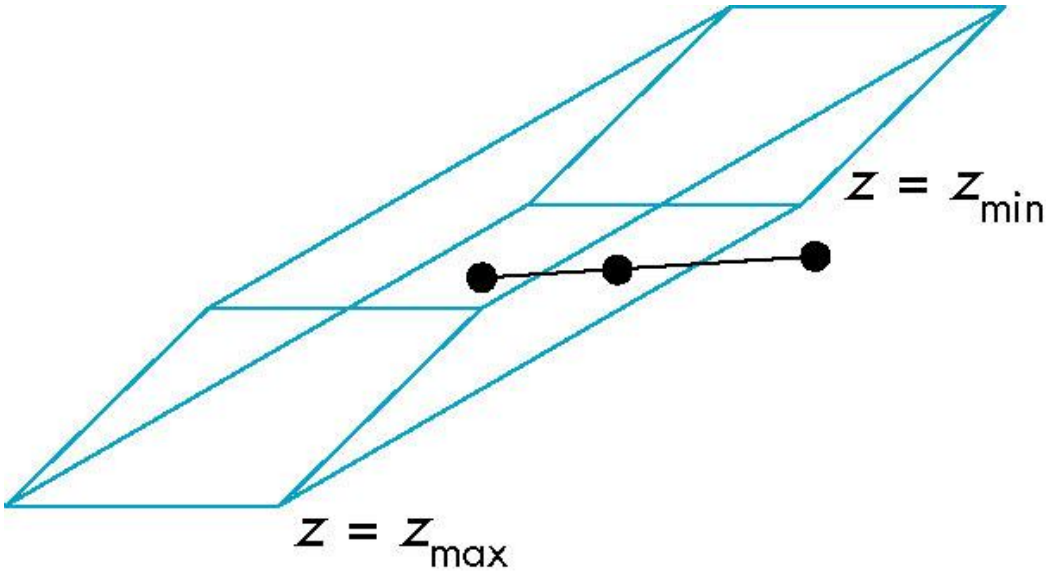
- Use 6-bit outcodes
- When needed, clip line segment against planes



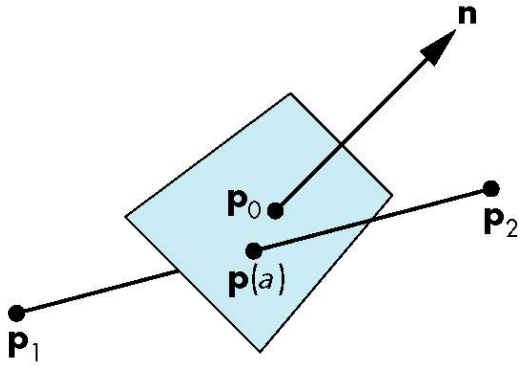


Clipping and Normalization

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



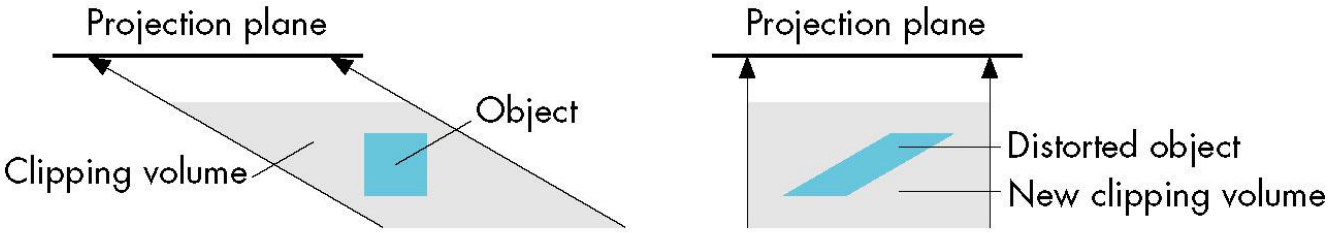
Plane-Line Intersections



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

Normalized Form

top view

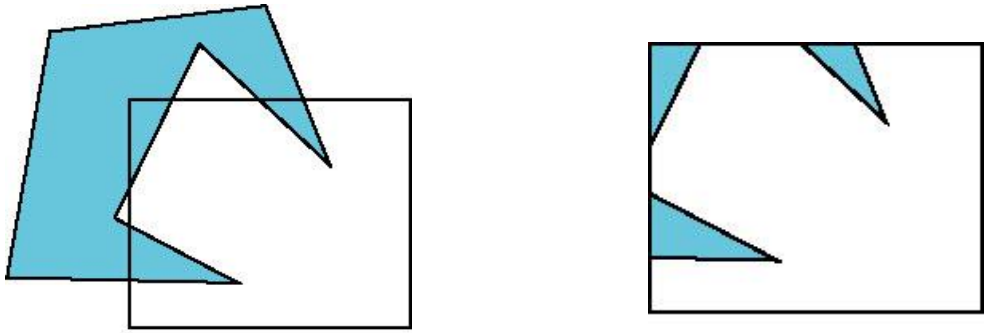


before normalization after normalization

- 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 > xmax ?

Polygon Clipping

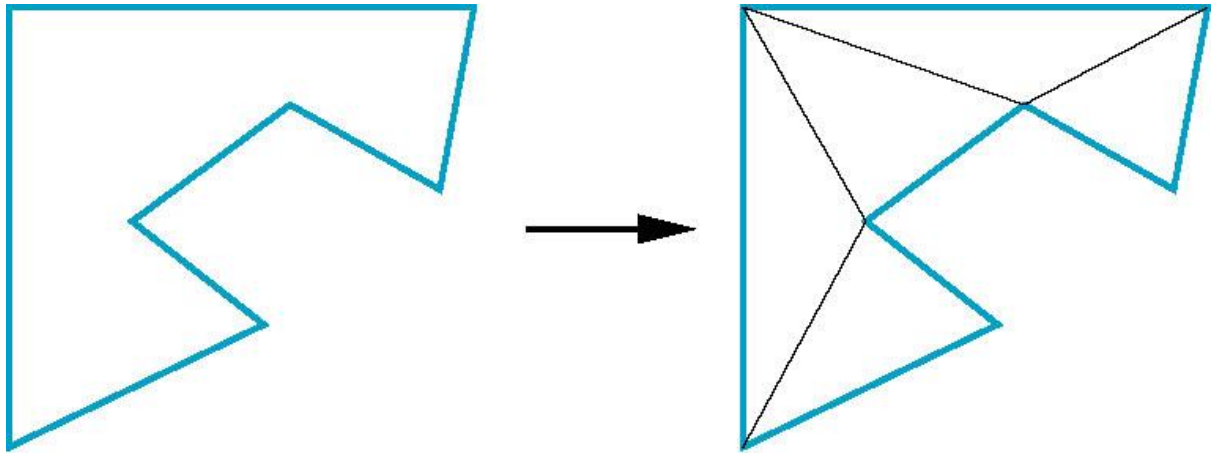
- Not as simple as line segment clipping
  - Clipping a line segment yields at most one line segment
  - Clipping a polygon can yield multiple polygons



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

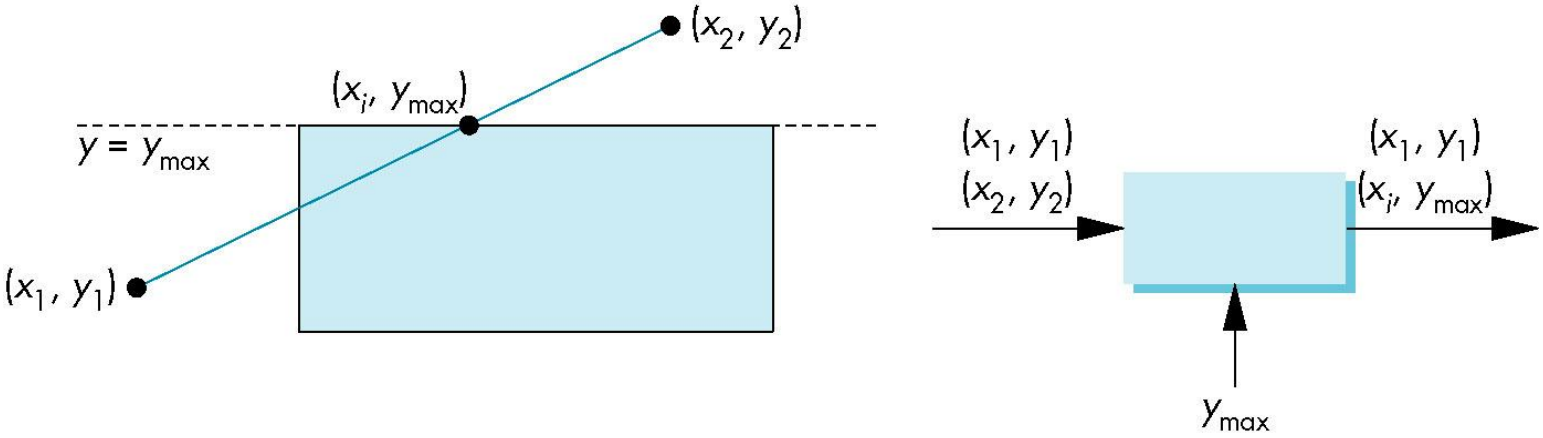
Tessellation and Convexity

- 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



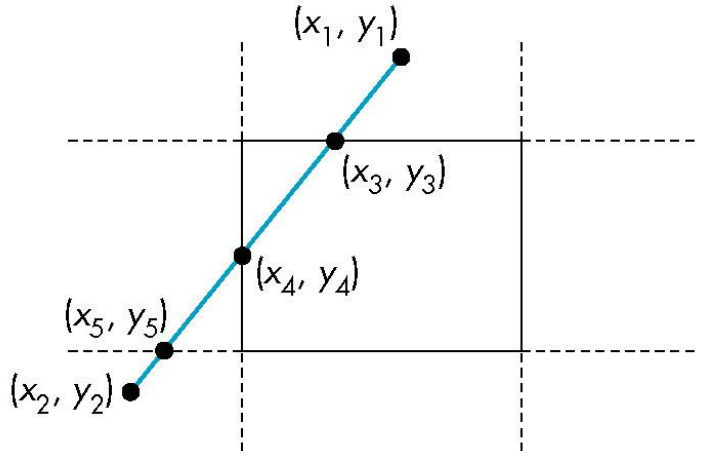
Clipping as a Black Box

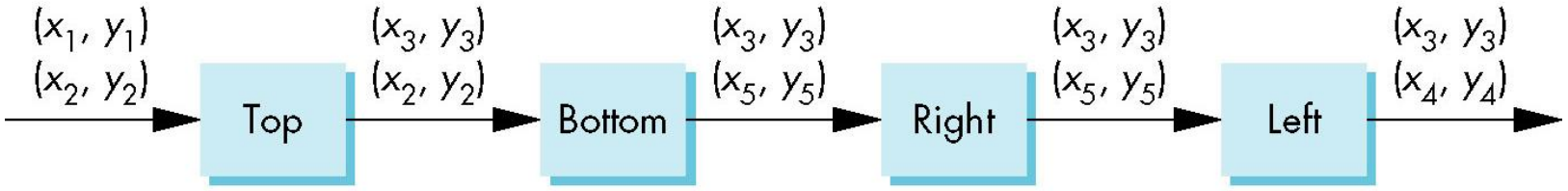
- Can consider line segment clipping as a process that takes in two vertices and produces either no vertices or the vertices of a clipped line segment



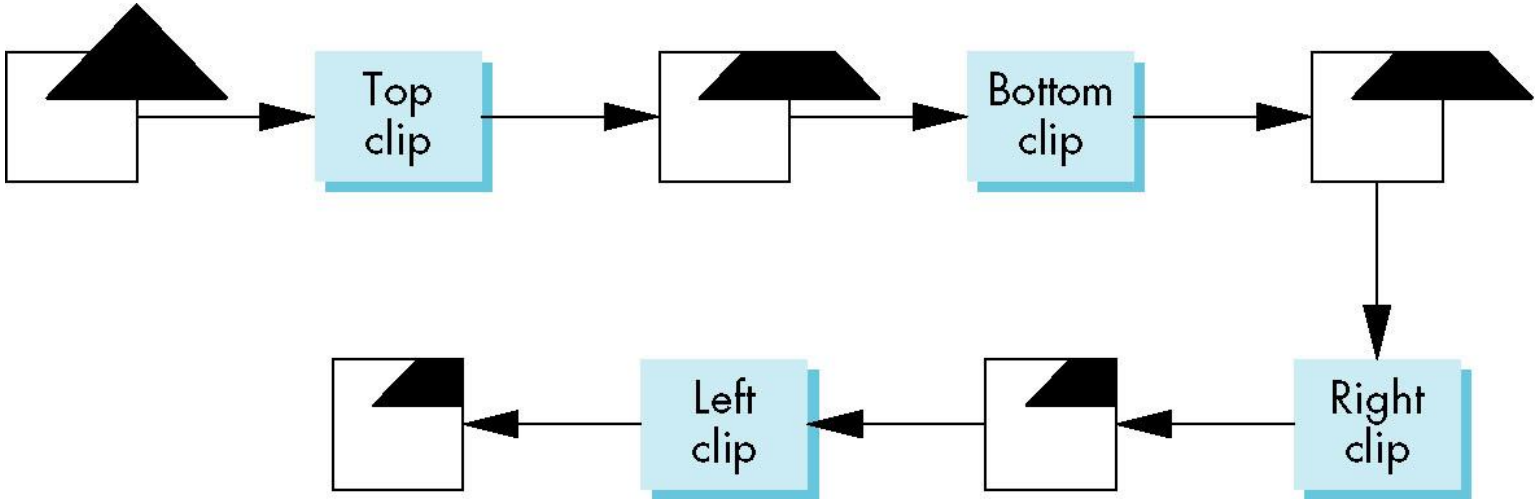
Pipeline Clipping of Line Segments

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





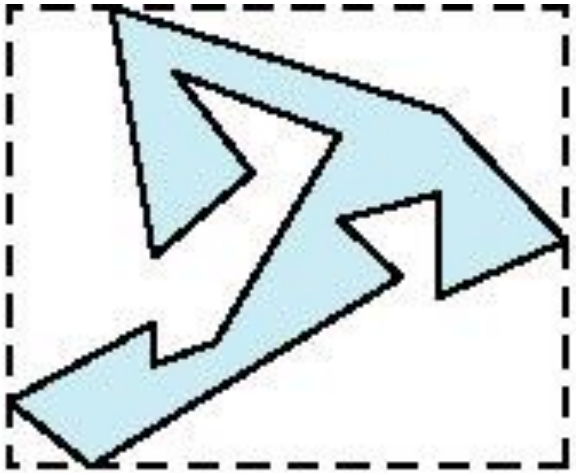
Pipeline Clipping of Polygons



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

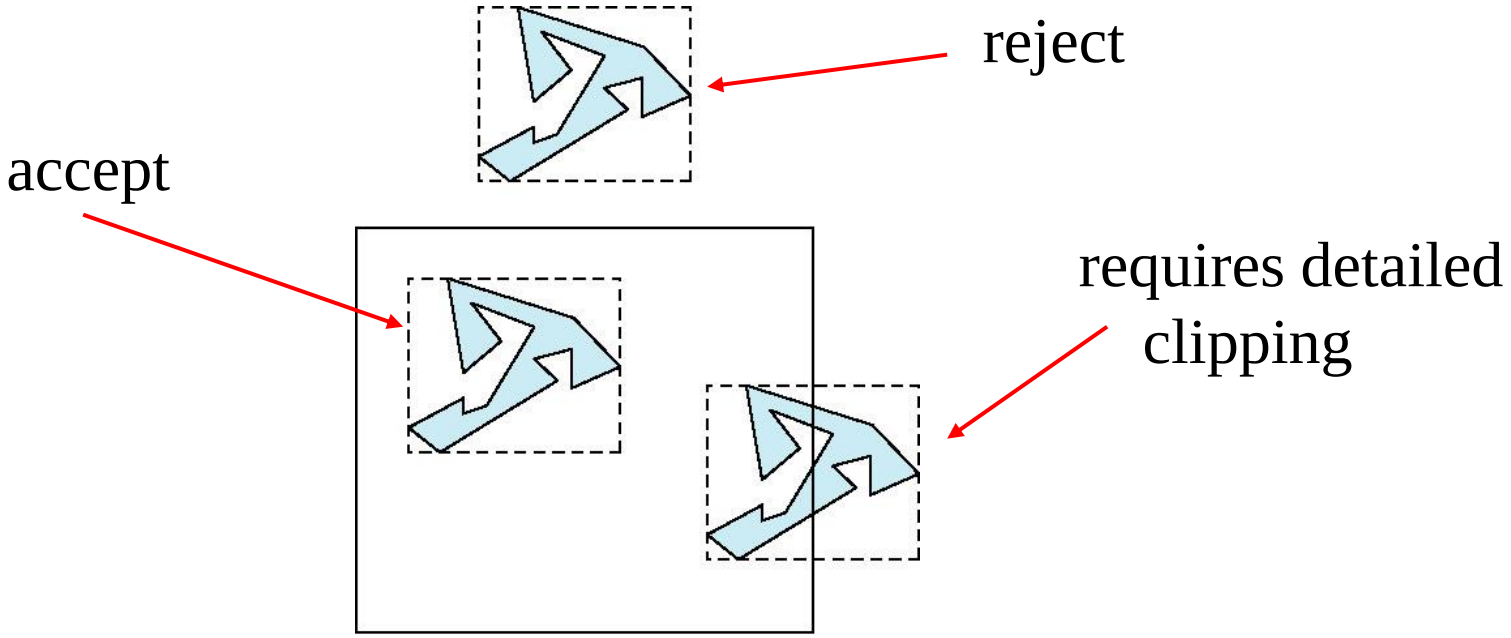
Bounding Boxes

- 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



Rasterization

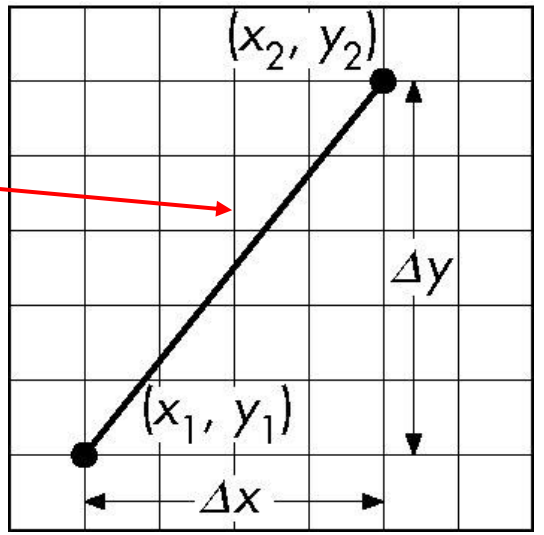
- Rasterization (scan conversion)
  - Determine which pixels that are inside primitive specified by a set of vertices
  - Produces a set of fragments
  - Fragments have a location (pixel location) and other attributes such color and texture coordinates that are determined by interpolating values at vertices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

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

y = mx + h



Scan Conversion of Line Segments

- Requirements
  - Pass 2 end points
  - Independent of start point
  - Smooth, equal thickness
  - Independent of number of call

DDA Algorithm

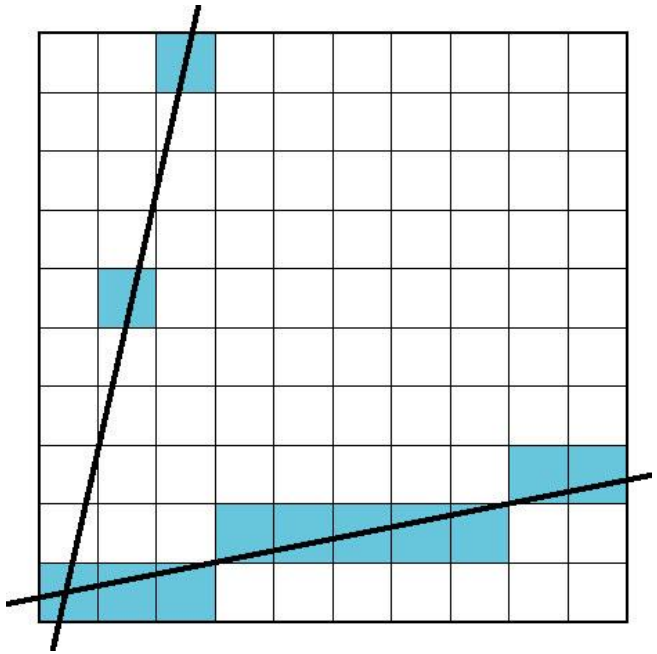
- Digital Differential Analyzer
  - DDA was a mechanical device for numerical solution of differential equations
  - Line $y = mx + h$ satisfies differential equation

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

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

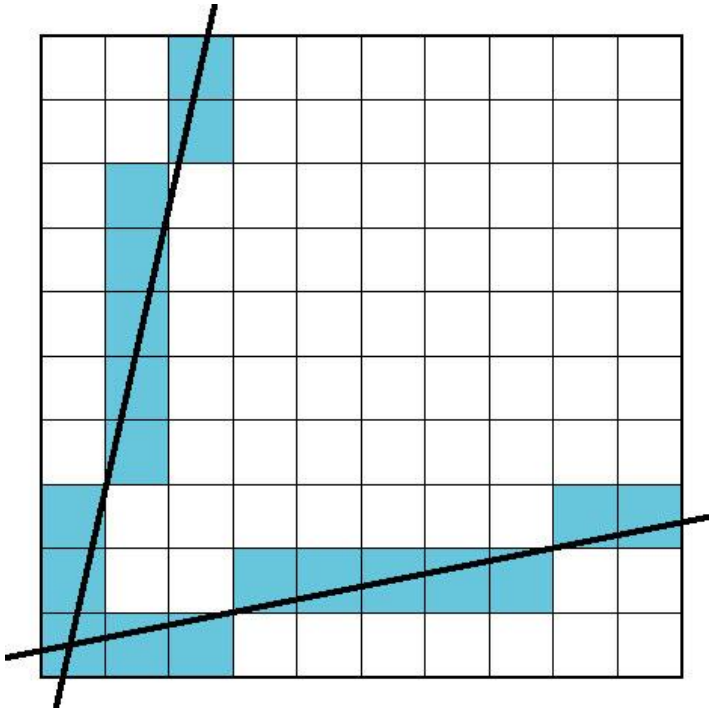
Problem

- In some case, line segment does not pass both end point
- Slow because using round().
- Problems for steep lines



Problem

- 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
- Line segment with end point: (x_a , y_a) and (x_b , y_b),
- Line equation:

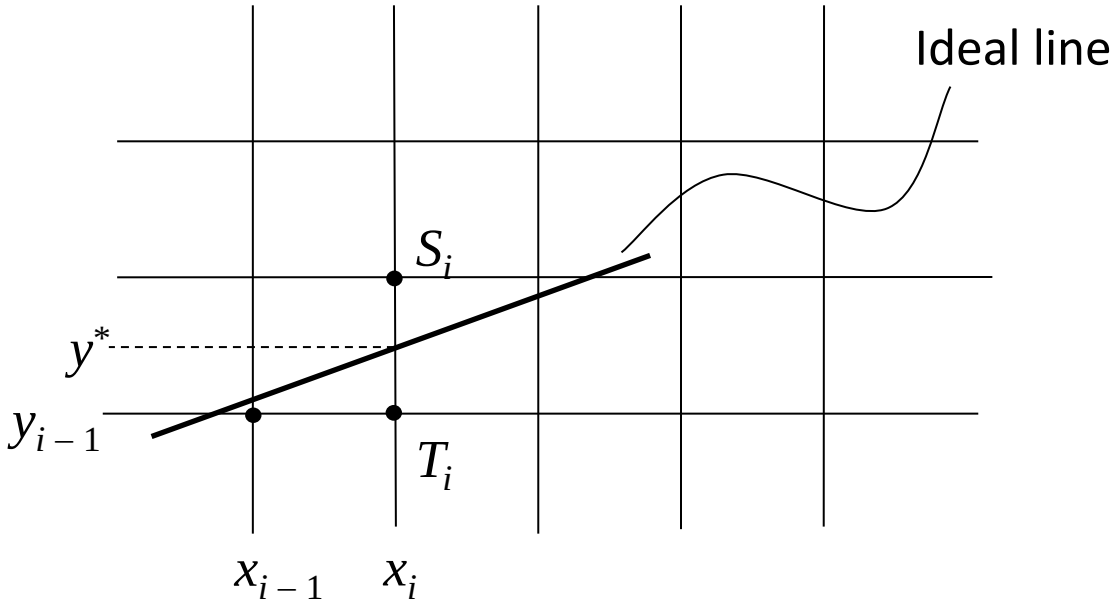
$$y = m(x - x_a) + y_a \qquad m = \frac{\Delta y}{\Delta x} \qquad \Delta y = y_b - y_a, \quad \Delta x = x_b - x_a$$

- (Without loss of generality), $x_a < x_b$ and $0 < m < 1$
  - Other cases by symmetry
- 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

Bresenham's Algorithm

- Generate the pixel by incremental Algorithm
- Error defined as follow:

$$e(T_i) = y^* - y_{i-1}$$
$$e(S_i) = (y_{i-1} + 1) - y^*$$



Choose T_i iff $e(T_i) - e(S_i) < 0$

Bresenham's Algorithm

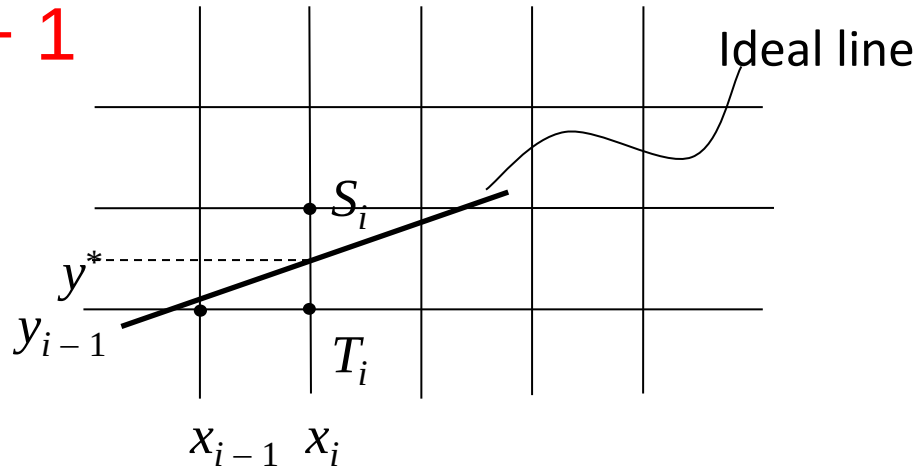
- $e(T_i) - e(S_i) = 2m(x_i - x_a) + 2(y_a - y_{i-1}) - 1$
- *Defined*

$$e_i = \Delta x \cdot (e(T_i) - e(S_i))$$
$$= 2(\Delta y)(x_i - x_a) + 2(\Delta x)(y_a - y_{i-1}) - \Delta x \qquad (*)$$

-The rule:

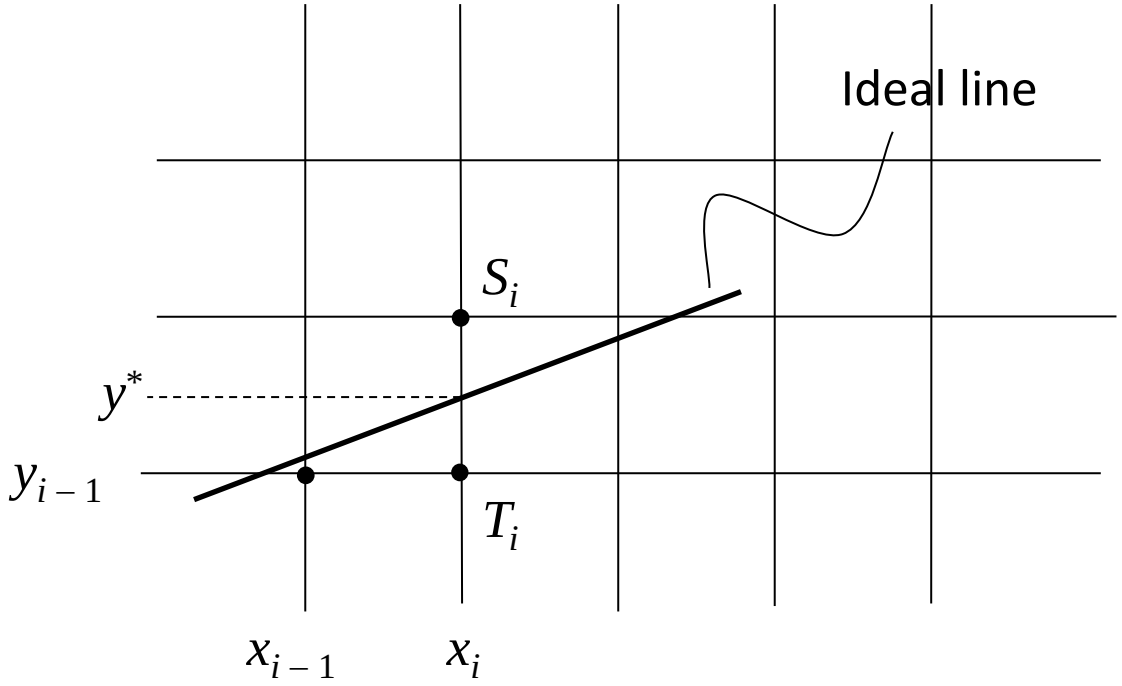
If $e_i < 0$ choose $y_i = y_{i-1}$,

else choose $y_i = y_{i-1} + 1$



Bresenham's Algorithm

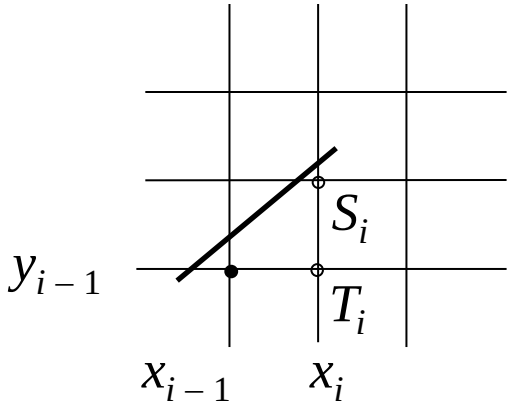
- Idea line segment pass the middle of S_i and T_i



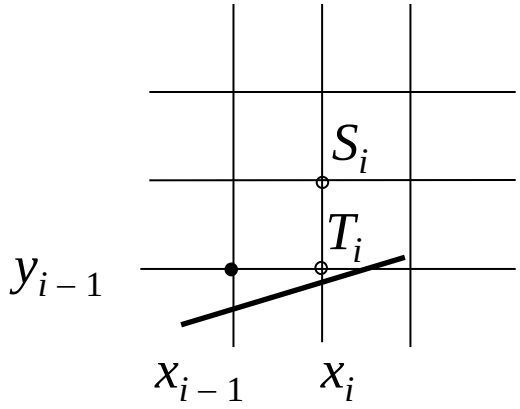
$e(T_i) = y^* - y_{i-1}$

$e(S_i) = (y_{i-1} + 1) - y^*$

Bresenham's Algorithm



$e(T_i) >= 0$
$e(S_i) <= 0$



$e(T_i) <= 0$
$e(S_i) >= 0$

$$e(T_i) = y^* - y_{i-1}$$
$$e(S_i) = (y_{i-1} + 1) - y^*$$
$$e_i = \Delta x \cdot (e(T_i) - e(S_i))$$

The rule:

If $e_i < 0$ choose $y_i = y_{i-1}$,

else choose $y_i = y_{i-1} + 1$

Bresenham's Algorithm

$e_i = 2(\Delta y)(x_i - x_a) + 2(\Delta x)(y_a - y_{i-1}) - \Delta x$

$\rightarrow e_{i+1} = 2(\Delta y)(x_{i+1} - x_a) + 2(\Delta x)(y_a - y_i) - \Delta x$

$\rightarrow e_{i+1} = e_i + 2(\Delta y)(x_{i+1} - x_i) - 2(\Delta x)(y_i - y_{i-1})$

The rule:

If $e_i < 0$ choose $y_i = y_{i-1}$,

else choose $y_i = y_{i-1} + 1$

- *According the rule*
  - if $e_i < 0$ choose $y_i = y_{i-1}$
    - $e_{i+1} = e_i + 2\Delta y$

    else choose $y_i = y_{i-1} + 1$
    - $e_{i+1} = e_i + 2\Delta y - 2\Delta x$

Bresenham's Algorithm

- Start:
  - $x_0 = x_a$, $y_0 = y_a$
  - $e_1 = 2(\Delta y) - \Delta x \quad (x_1 - x_a = 1)$

    (Use $e_i = 2(\Delta y)(x_i - x_a) + 2(\Delta x)(y_a - y_{i-1}) - \Delta x$*)*

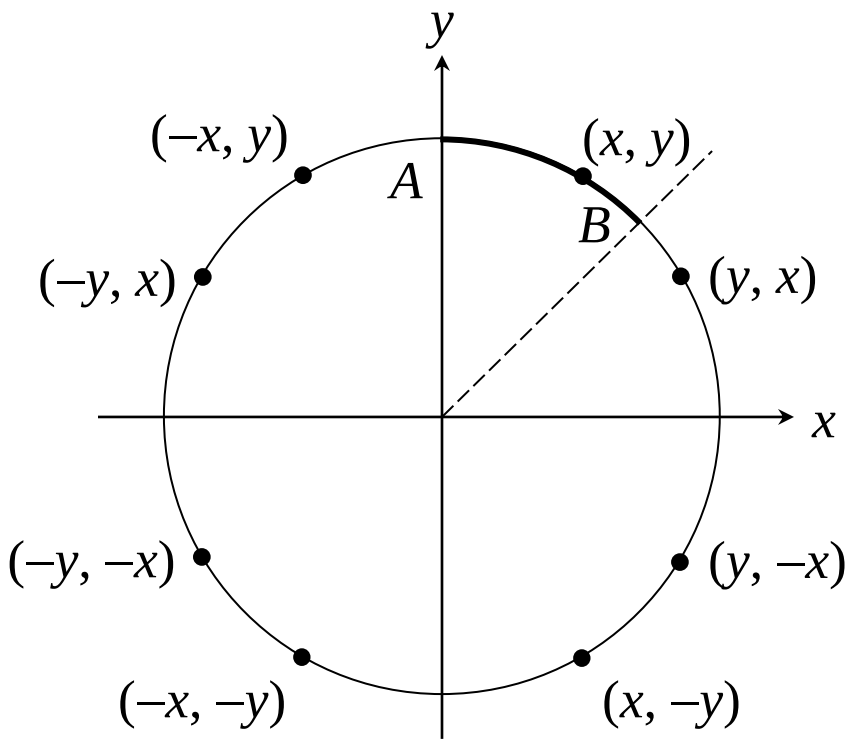
Bresenham's Algorithm

```
procedure Bresenham(xa, xb : col; ya, yb : row;  col_val : color);
begin
    y := ya; dx := xb - xa; dy := yb - ya;
    e_noinc := dy + dy;                          {2dy}
    e       := e_noinc - dx;                     {e1=2dy-dx}
    e_inc   := e - dx;                           {2dy-2dx}
    for x := xa to xb do
        begin
            SetPixel(x, y, col_val);
            if e < 0 then e := e + e_noinc;
            else begin
                    y := y + 1;
                    e := e + e_inc
            end;
        end;
end; {Bresenham}
```

Bresenham's Algorithm

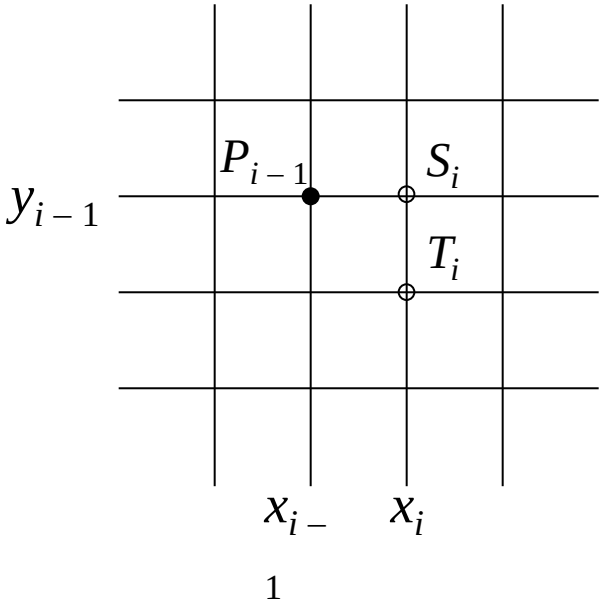
- $x_a > x_b$ (change the points)
- $m > 1$ (change the role of x and y)
- $-1 < m < 0$ (replace dy by –dy)
- m < -1 (replace dx by –dx, change the role of x and y)
- Vertical and horizontal line

Circle Drawing Algorithms



Circle Drawing Algorithms

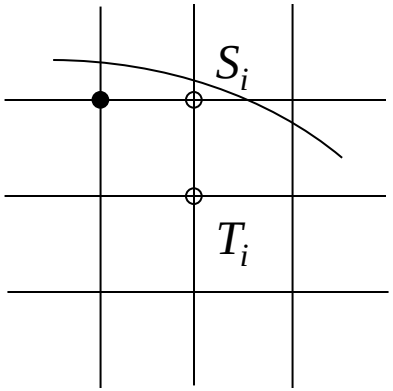
- *Suppose pixel at step i* - 1 is $P_{i-1} = (x_{i-1}, y_{i-1})$
- Define
  - $S_i = (x_{i-1} + 1, y_{i-1})$
  - $T_i = (x_{i-1} + 1, y_{i-1} - 1)$
- Define the error
  - $e(P) = (x^2 + y^2) - R^2$
- Define the decision function
  - $d_i = e(S_i) + e(T_i)$
  - if $d_i < 0$ choose S_i
    else choose T_i



Circle Drawing Algorithms

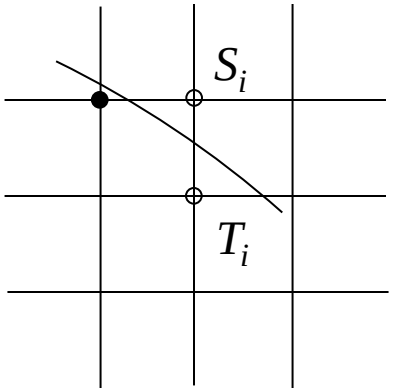
$d_i = e(S_i) + e(T_i)$

$d_i < 0$ choose S_i



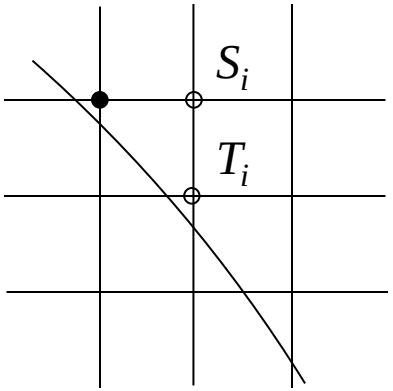
$e(S_i) < 0$
$e(T_i) < 0$

OK!



$e(S_i) > 0$
$e(T_i) < 0$

OK



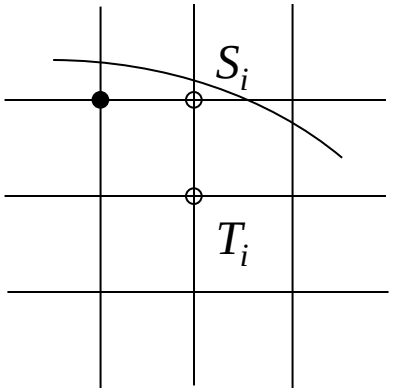
$e(S_i) > 0$
$e(T_i) > 0$

imposible

Circle Drawing Algorithms

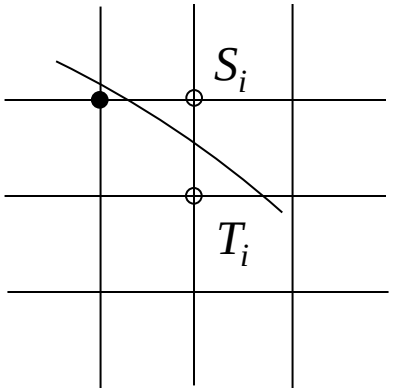
$d_i = e(S_i) + e(T_i)$

$d_i >= 0$ choose T_i



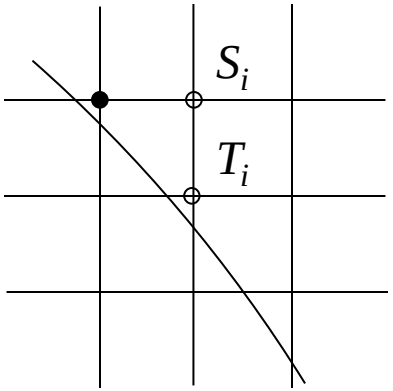
$e(S_i) < 0$
$e(T_i) < 0$

Impossible



$e(S_i) > 0$
$e(T_i) < 0$

OK



$e(S_i) > 0$
$e(T_i) > 0$

OK

Circle Drawing Algorithms

$e(P) = (x^2 + y^2) - R^2$

$d_i = e(S_i) + e(T_i)$

$d_{i+1} = d_i + 4\,x_{i-1} + 6 + 2(y_i^2 - y_{i-1}^2) - 2(y_i - y_{i-1})$

if $d_i < 0 \rightarrow y_i = y_{i-1}$, choose T_i

$d_{i+1} = d_i + 4\,x_{i-1} + 6$

else $y_i = y_{i-1} - 1$, choose S_i

$d_{i+1} = d_i + 4(x_{i-1} - y_{i-1}) + 10$

- Start:

$x_0 = 0, y_0 = R$

$\rightarrow S_1 = (1, R)$ and $T_1 = (1, R - 1)$,

$\rightarrow d_1 = e(S_1) + e(T_1) = (1^2 + R^2 - R^2) + (1^2 + (R-1)^2 - R^2) = 3 - 2R$

Circle Drawing Algorithms

```
procedure MichCirc(xc : col; yc : row; Rad : integer; value : color);
begin
    x := 0; y := Rad;   d := 3 - 2*Rad;
    while x <= y do begin
        SetPixel(xc + x, yc + y, value);        {draw 8 points }
        SetPixel(xc - x, yc + y, value);        {based on (x, y)}
        ……
        if d < 0 then d := d + 4*x + 6          {update error term}
        else begin
                d := d + 4*(x - y) + 10;
                y := y - 1
        end;
        x := x + 1
    end
end;
```

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);
    }
}
```

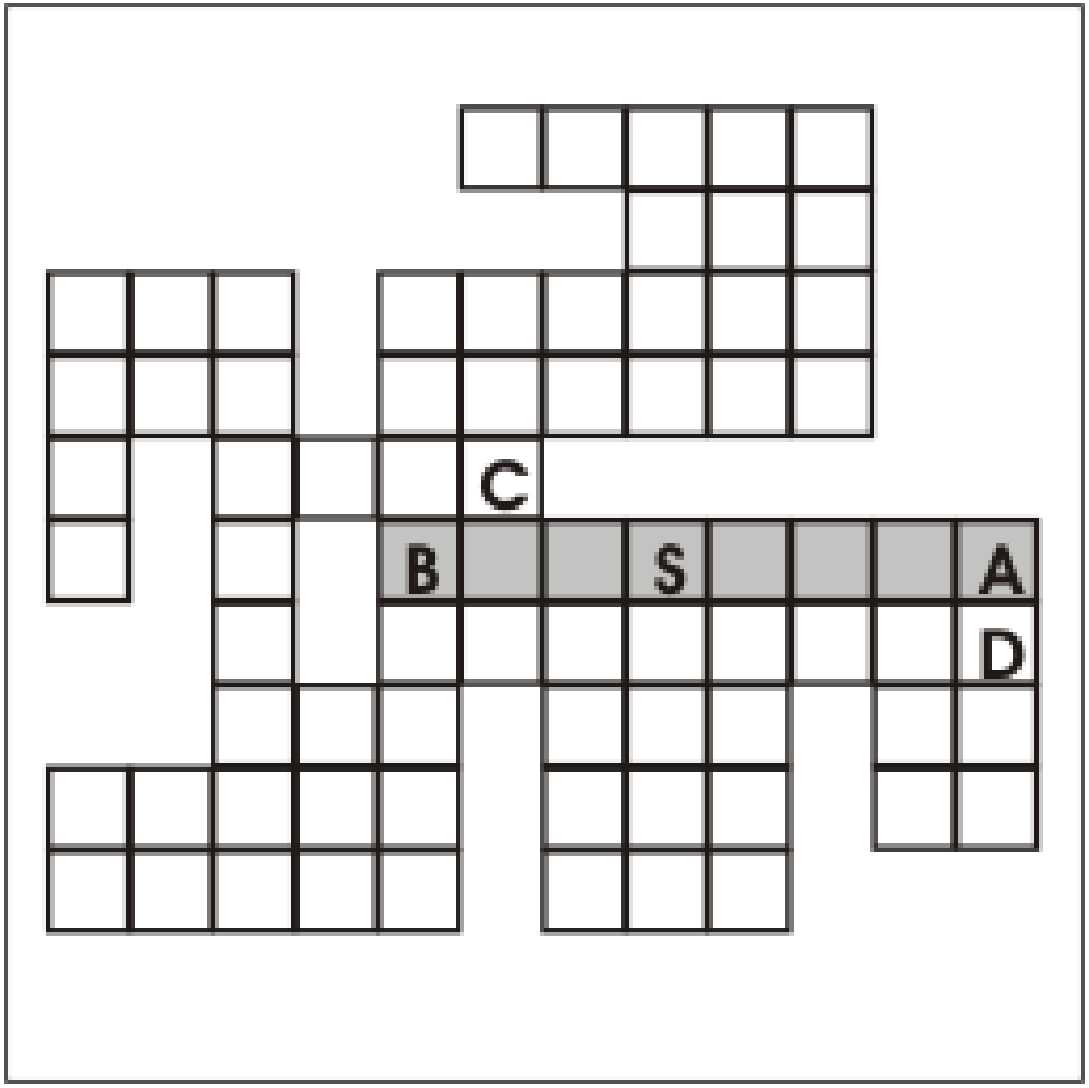
Flood Fill

- Repeat many times
- overflow

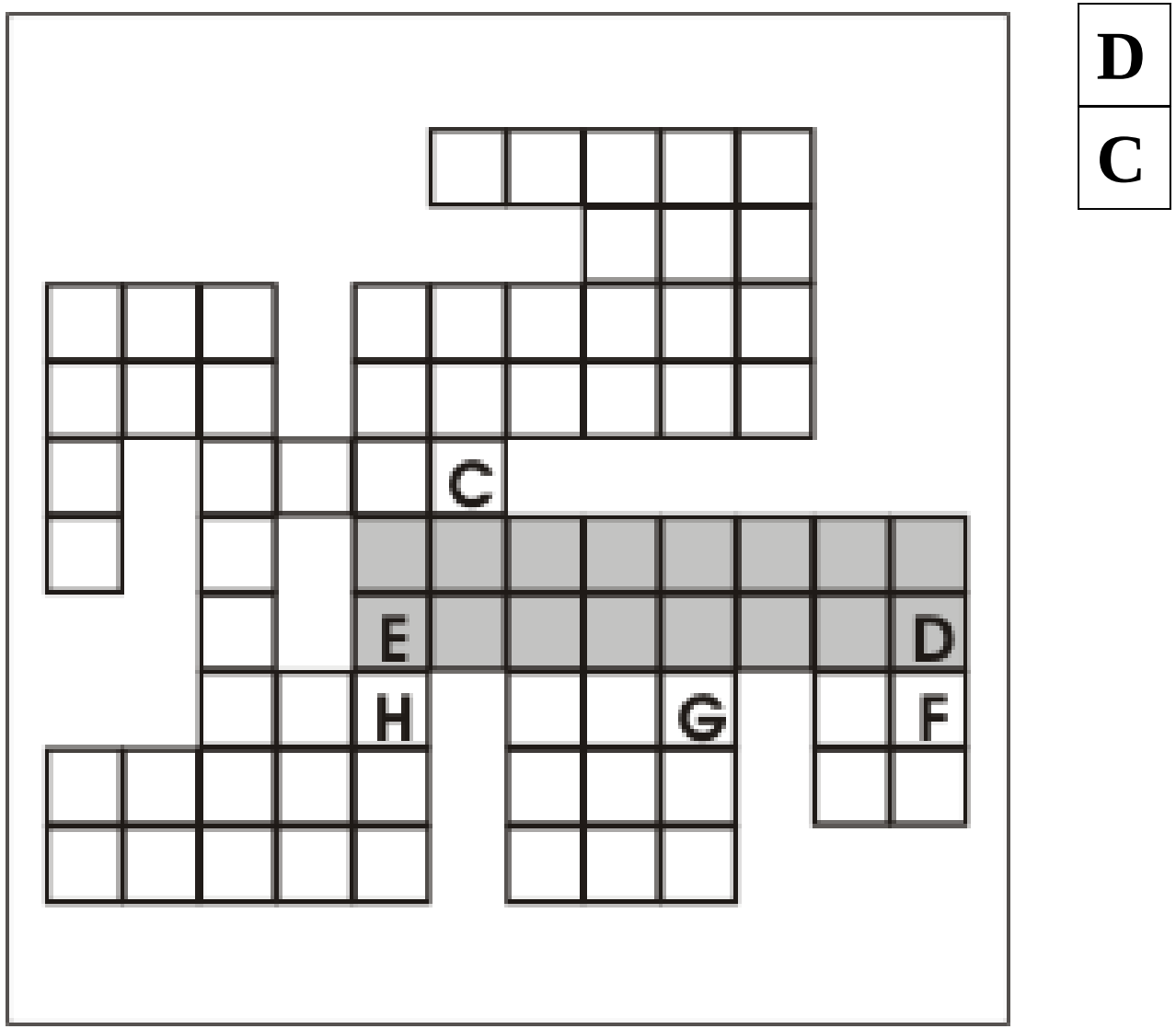
Flood Fill

```
procedure FloodFill(x : col; y : row; int_color, new_color :
    color);
begin
    if (GetPixel(x, y) >< bound_color) and (GetPixel(x, y) ><
    new_color) then begin
        SetPixel(x, y, new_color);
        FloodFill(x – 1, y, int_color, new_color);
        FloodFill(x + 1, y, int_color, new_color);
        FloodFill(x, y + 1, int_color, new_color);
        FloodFill(x, y – 1, int_color, new_color);
    end
end;
```

Run of Pixels

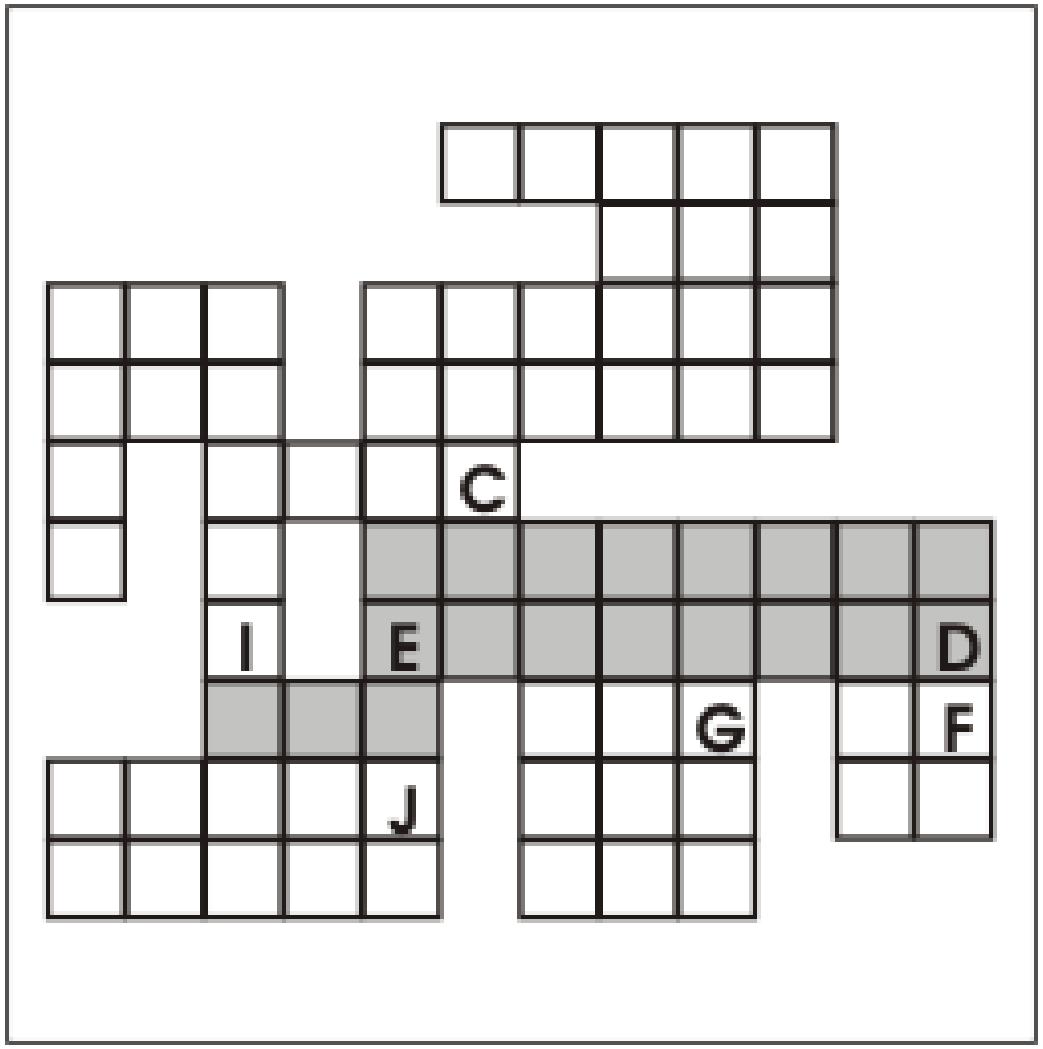


Run of Pixels



D
C

Run of Pixels

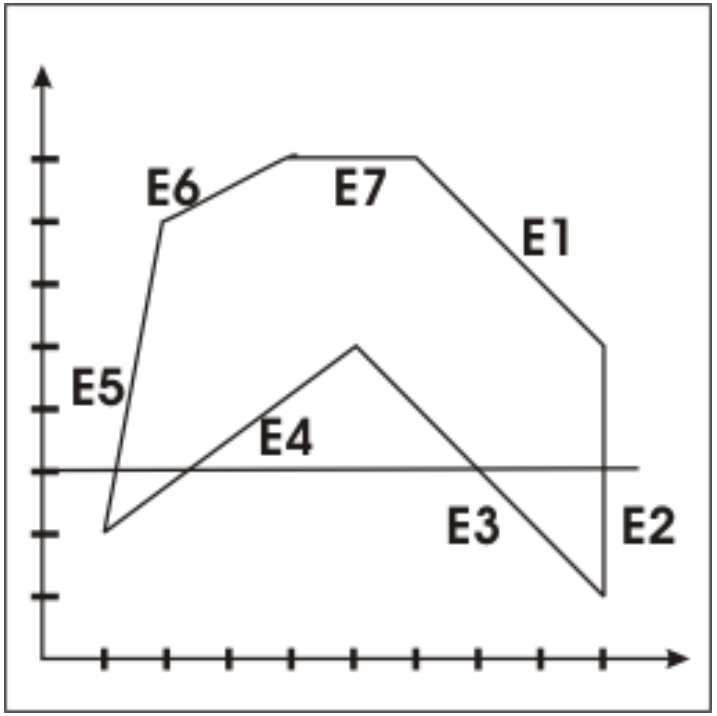


J
I
G
F
C

Run of Pixels

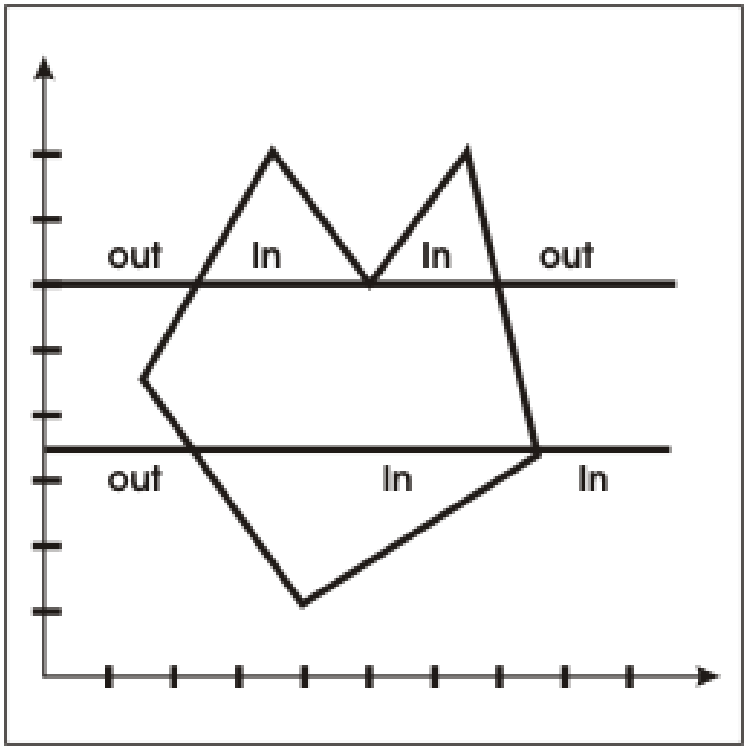
```
Push address of seed pixel on the stack;
   while stack not empty do
      Pop the stack to provide the next seed;
      Fill in the run defined by the seed;
      Examine the row above for runs reachable
            from this run;
      Push the address of the rightmost pixels
            of each such run;
      Do the same for the row below the current run;
   end;
```

Fill Polygon-defined Region



- With each scan line
  - Find intersections with the edge of polygon
  - Sort the intersections
  - Fill each span

Fill Polygon-defined Region



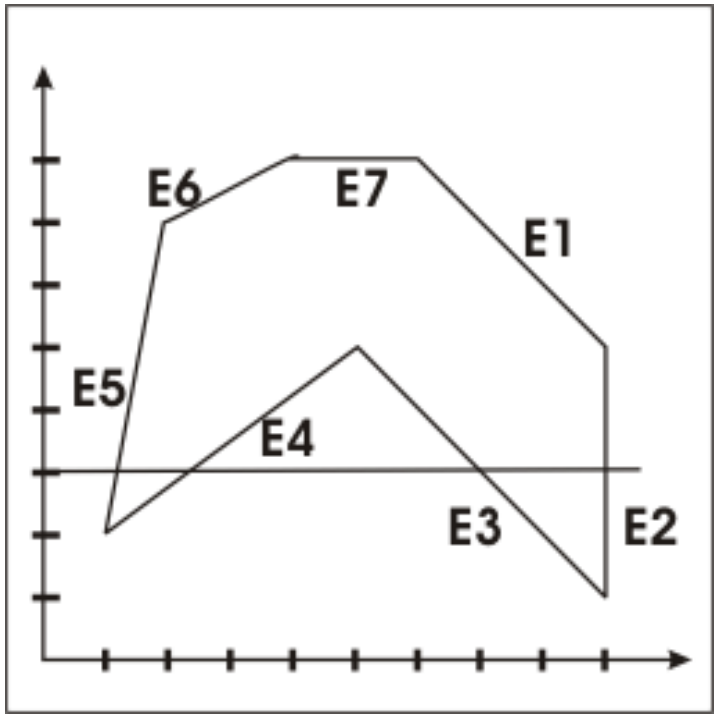
- Ignore horizontal edge
- Inside-outside test not true at the vertex which is not local maximum, minimum
  - Move down the point of the edge 1 pixel

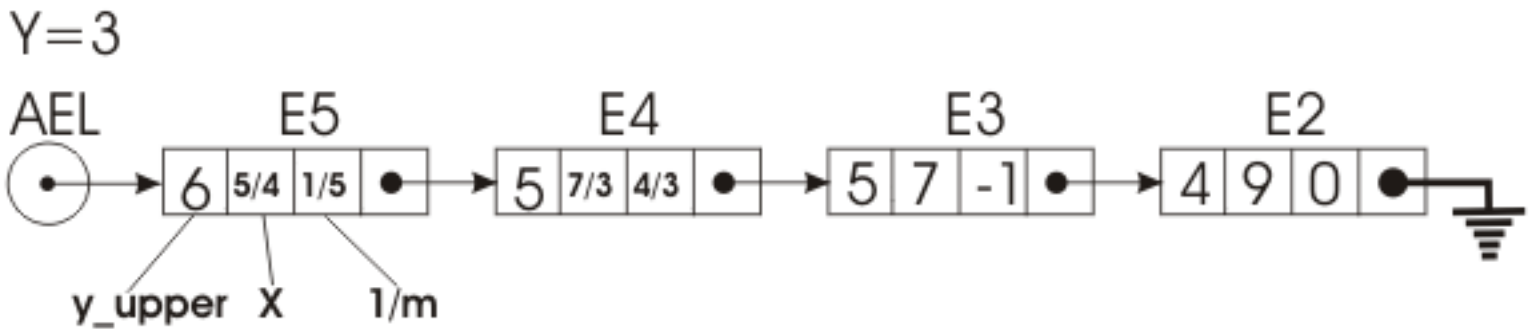
Fill Polygon-defined Region

- Active Edge List (AEL)
  - The edges intersected by scanline y, mostly intersected by scanline y+1.
  - Can predict the value of intersection
- **type**

```
edge_ptr = ^edge_info
edge_info = record
            y_upper      : row;
            x_int        : real;
            recip_slope : real;
            next          : egde_ptr;
end;
```

Fill Polygon-defined Region



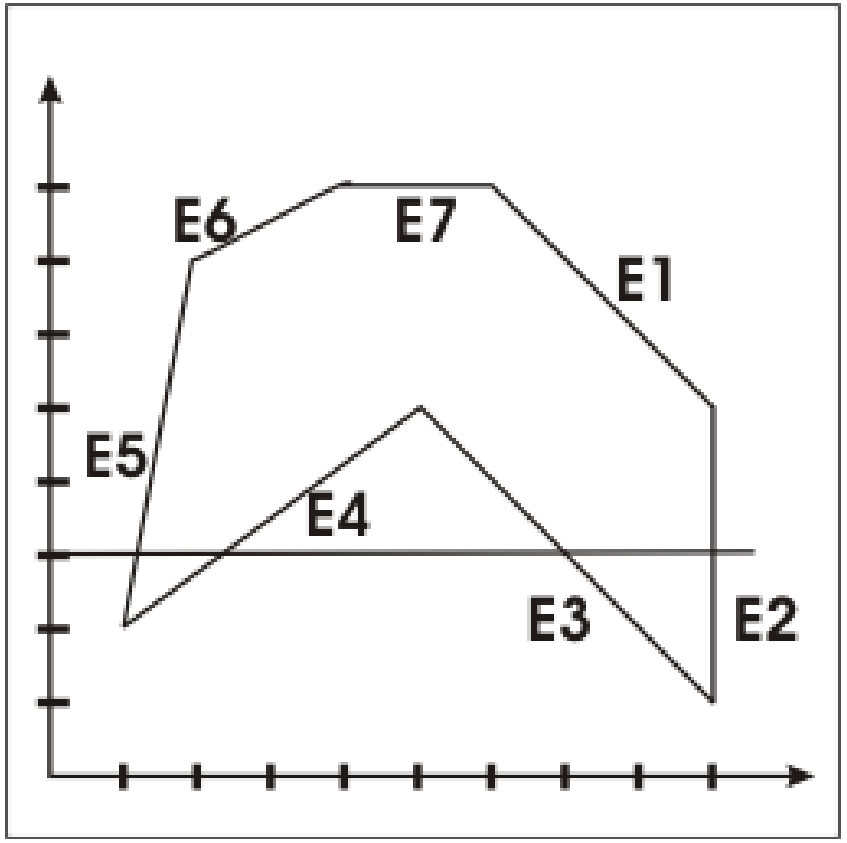


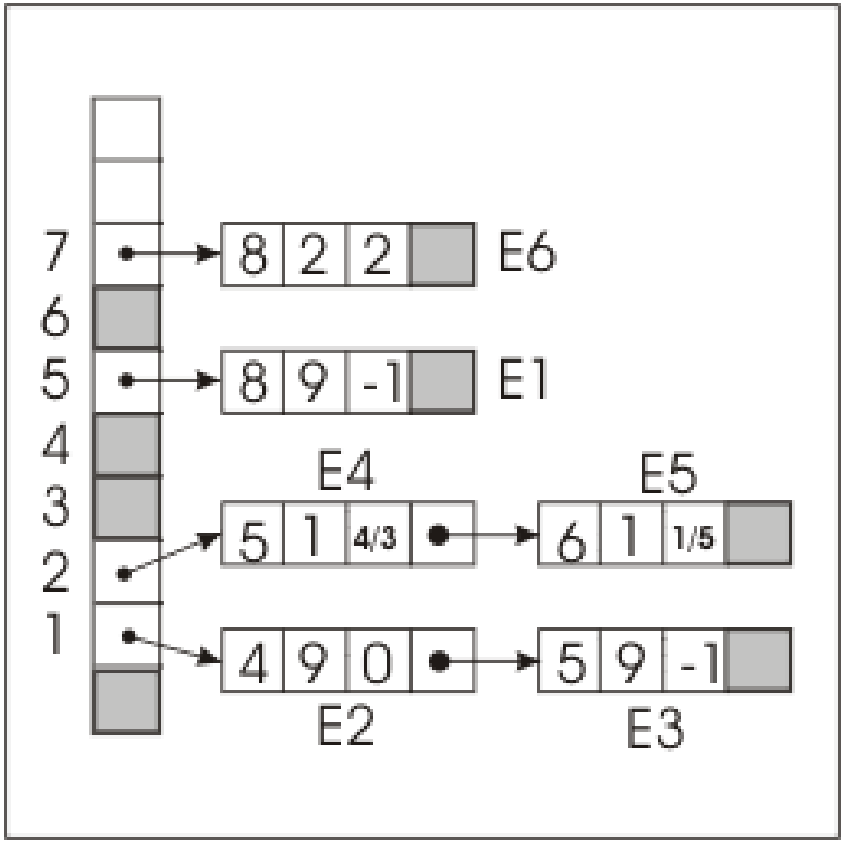
Fill Polygon-defined Region

- update AEL for scanline y+1
  - Delete all the edges with y_upper < new_y
  - Change x_value by plus recip_slope
  - Add new edge with y_lower = new_y
  - Sort the intersection

Fill Polygon-defined Region

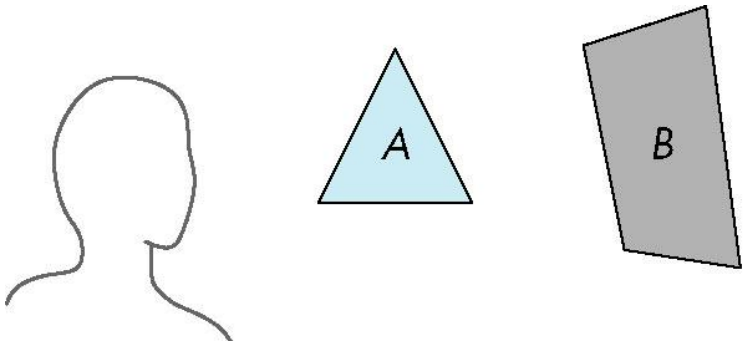
- Build edge_table: **array** [row] of edge_ptr



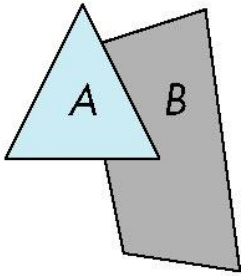


Depth sort & Painter's Algorithm

- Object-space with lower polygon count
- Render polygons a back to front order so that polygons behind others are simply painted over



B behind A as seen by viewer

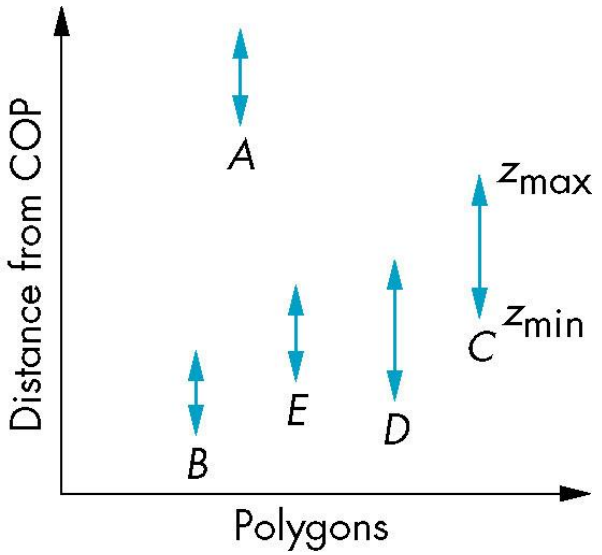


Fill B then A

Depth sort & Painter's Algorithm

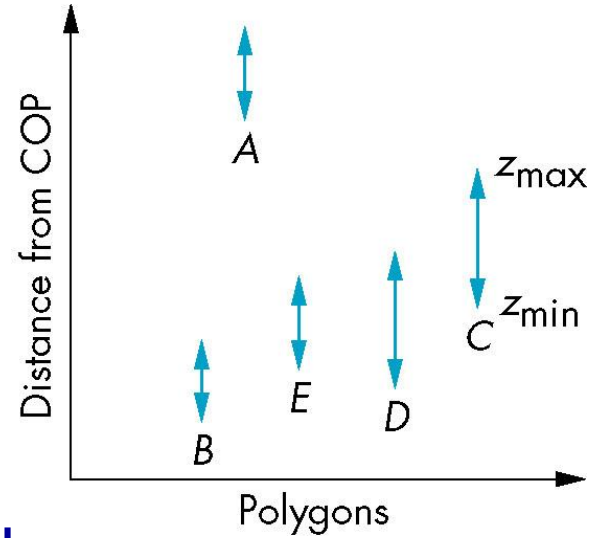
- Requires ordering of polygons first
  - O(n log n) calculation for ordering
  - Not every polygon is either in front or behind all other polygons
- Order polygons and deal with easy cases first, harder later

Polygons sorted by distance from COP

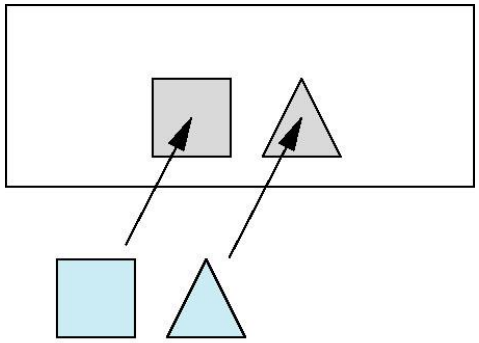


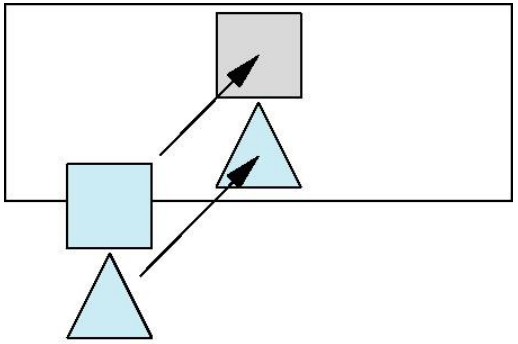
Depth sort & Painter's Algorithm

- A lies behind all other polygons
  - Can render

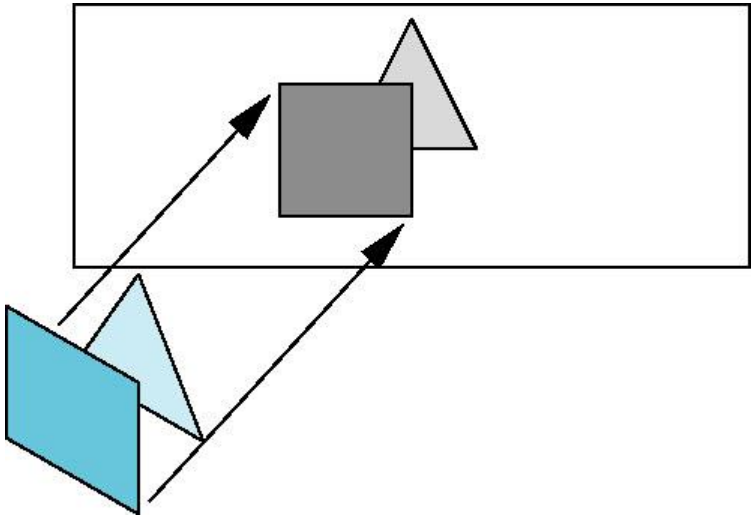


- Polygons overlap in z but not in either x or y
  - Can render independently

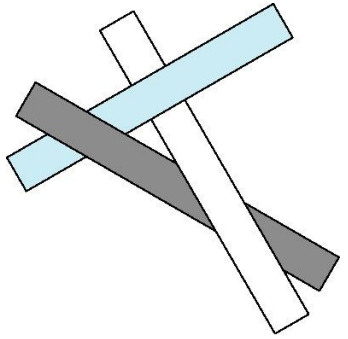




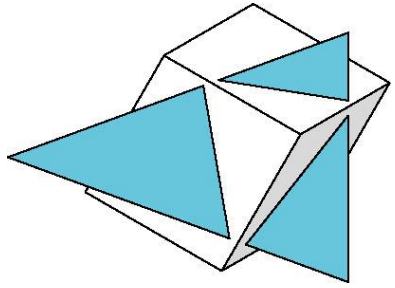
Depth sort & Painter's Algorithm



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



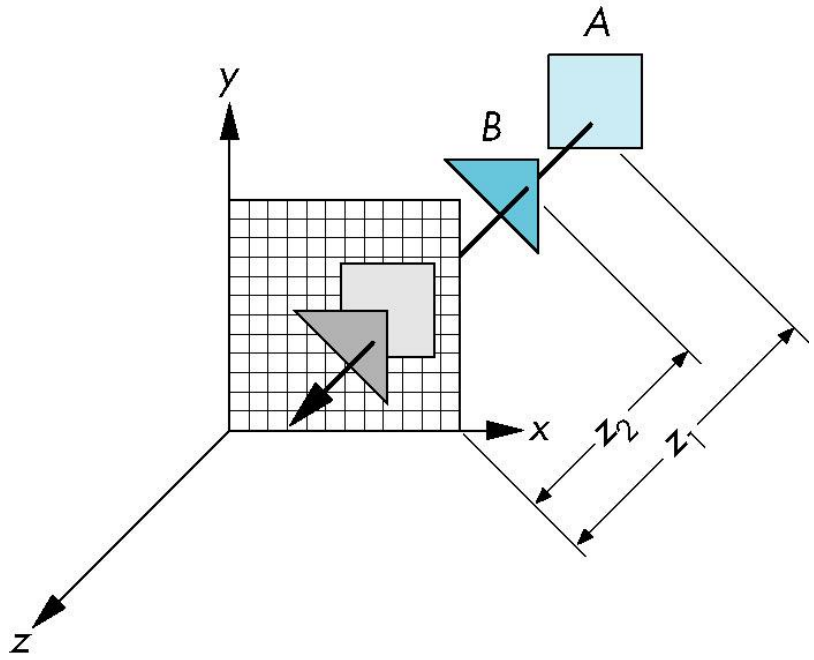
cyclic overlap



penetration

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

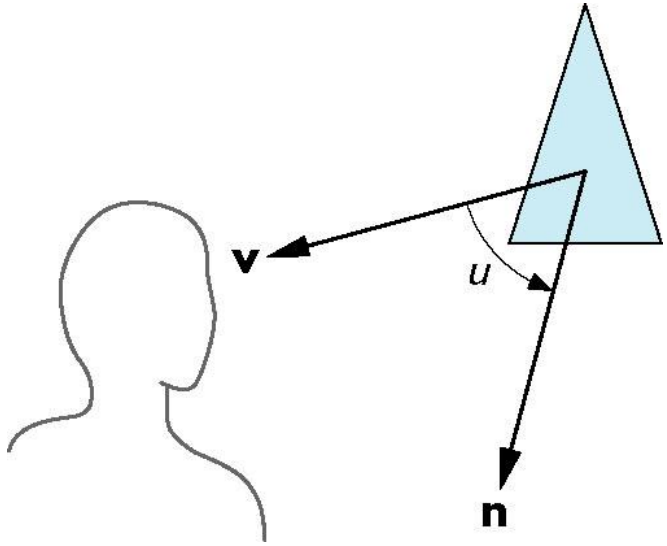


z-Buffer Algorithm

- It must be
  - Requested in **`main()`**
    - **`glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH)`**
  - Enabled
    - **`glEnable(GL_DEPTH_TEST)`**
  - Cleared in the display callback
    - **`glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)`**

Back-Face Removal (Culling)

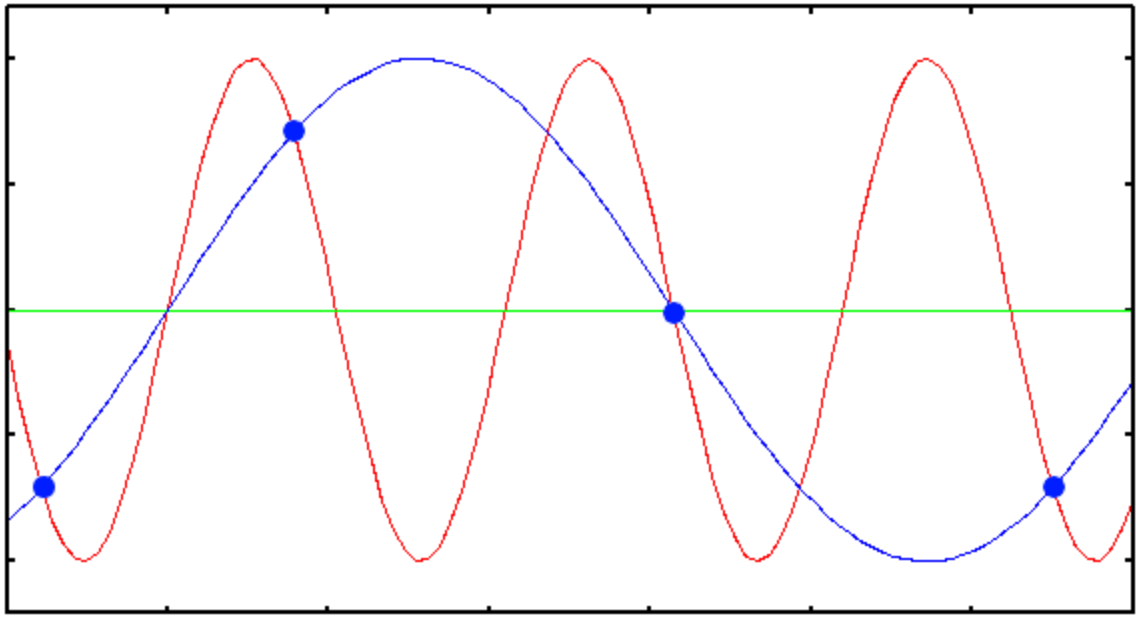
•face is visible iff $90 \geq \theta \geq -90$
equivalently $\cos \theta \geq 0$
or $\mathbf{v} \cdot \mathbf{n} \geq 0$



•Compute the area of polygon

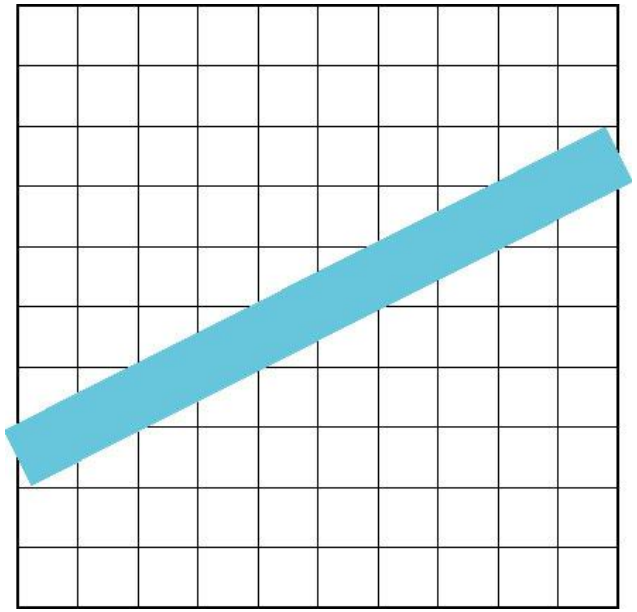
Aliasing

- One pattern of pixels ←→ many different continous line
- Pixel locations are fixed
- Pixels have a fixed size and shape



Aliasing

- Ideal rasterized line should be 1 pixel wide

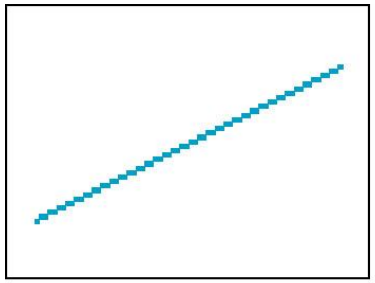


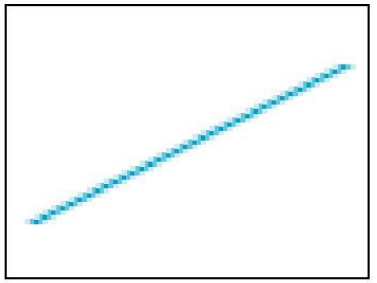
- Choosing best y for each x (or visa versa) produces aliased raster lines

Antialiasing by Area Averaging

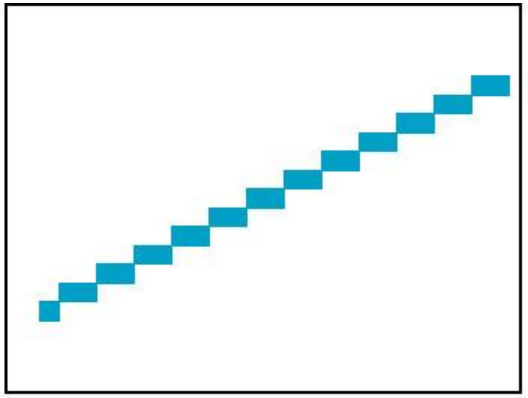
- Color multiple pixels for each x depending on coverage by ideal line

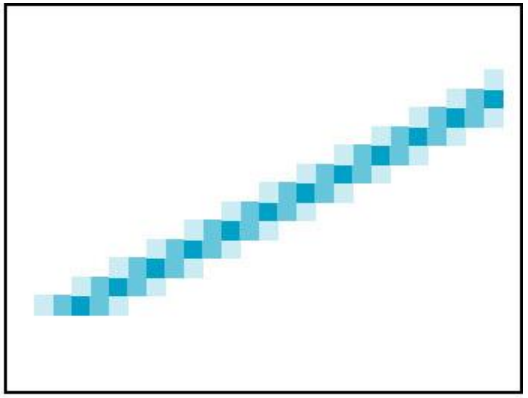
original





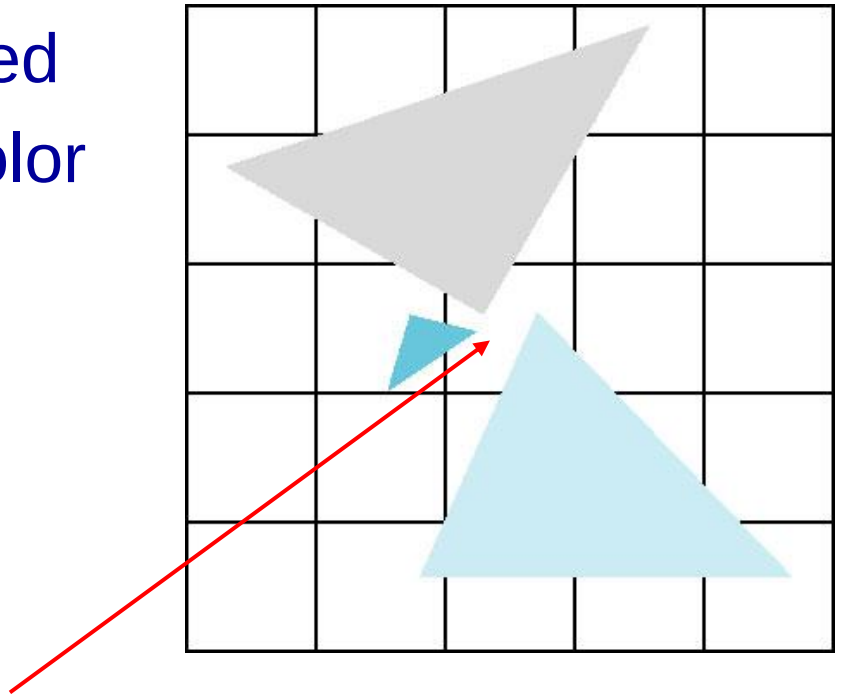
antialiased





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

Further Reading

- "**Interactive Computer Graphics: A Topdown Approach Using OpenGL**", *Edward Angel*
  - Chapter 7: From Vertices to Fragments
- "Đồ họa máy tính trong không gian ba chiều", Trần Giang Sơn
  - Chương 5: Xây dựng công cụ cho đồ họa raster