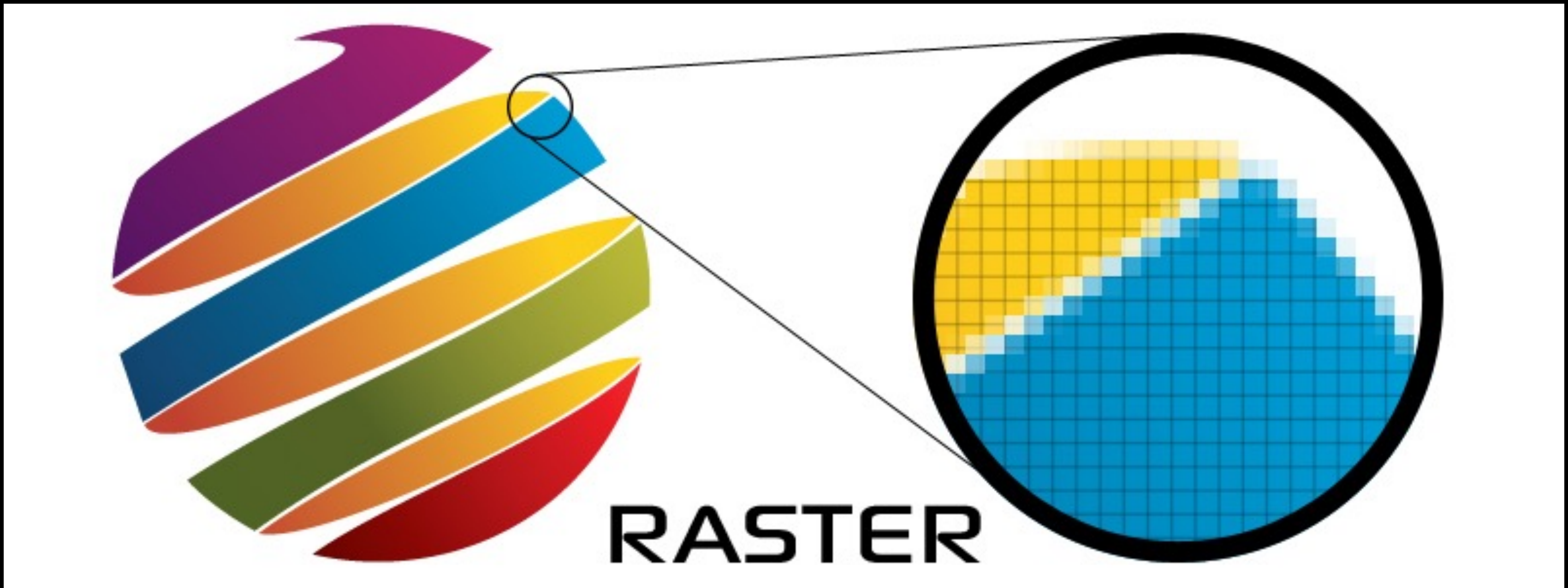
Basic Raster Graphics Algorithms for Drawing 2D Primitives

Prof. Lizhuang Ma & Ran Yi
Shanghai Jiao Tong University

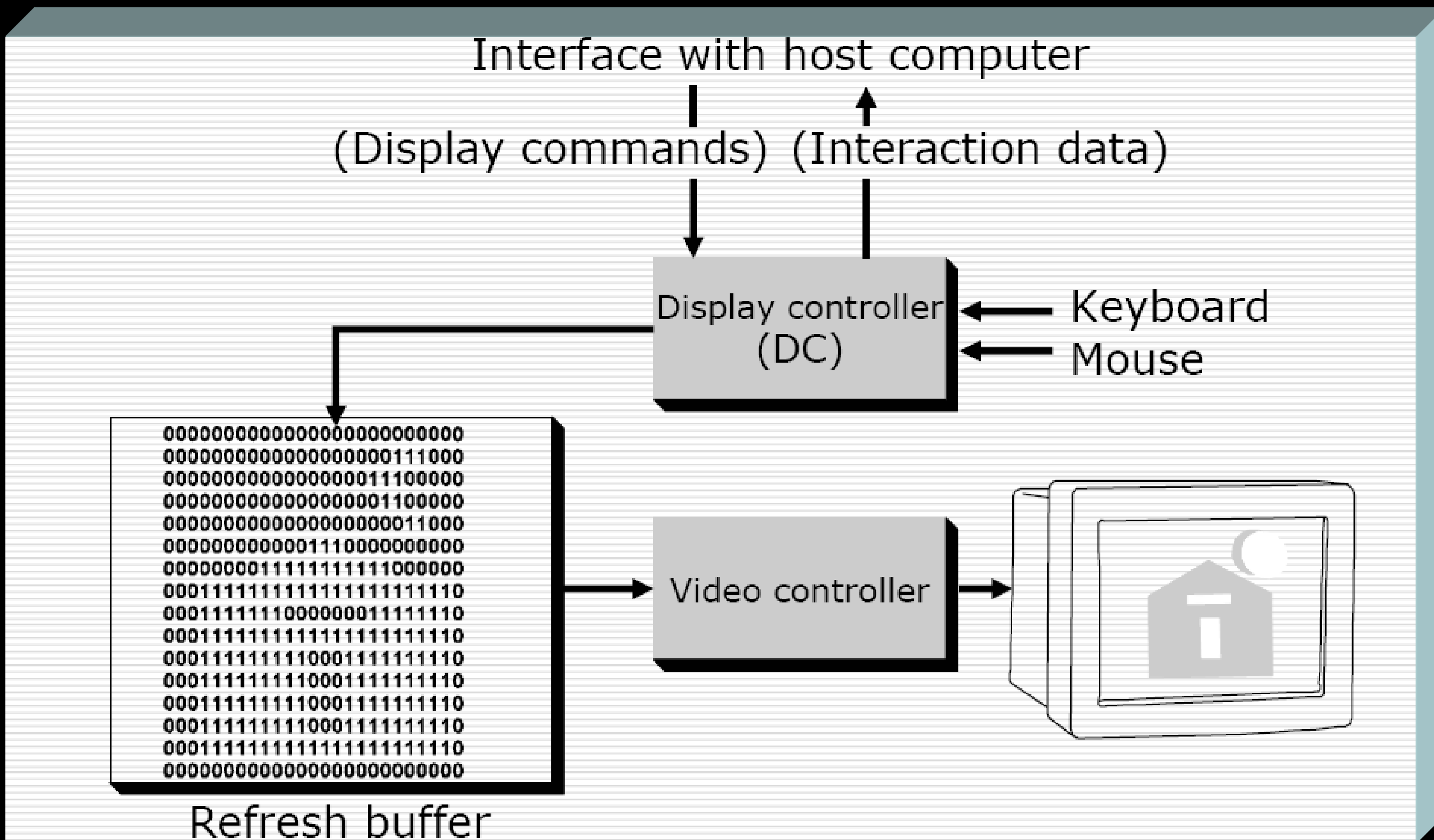
Contents

Raster Scan Display

- Raster: a rectangular collection of dots plotted
- An image subdivided into various horizontal lines (scan lines), then further divided into different pixels

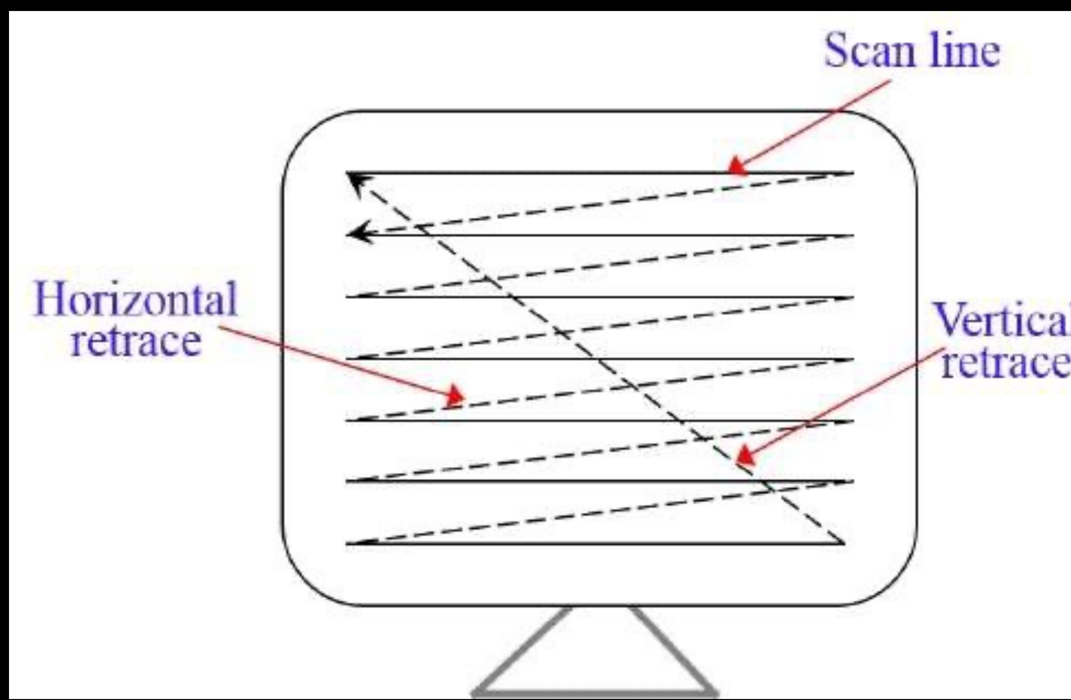


Architecture of a Raster Display



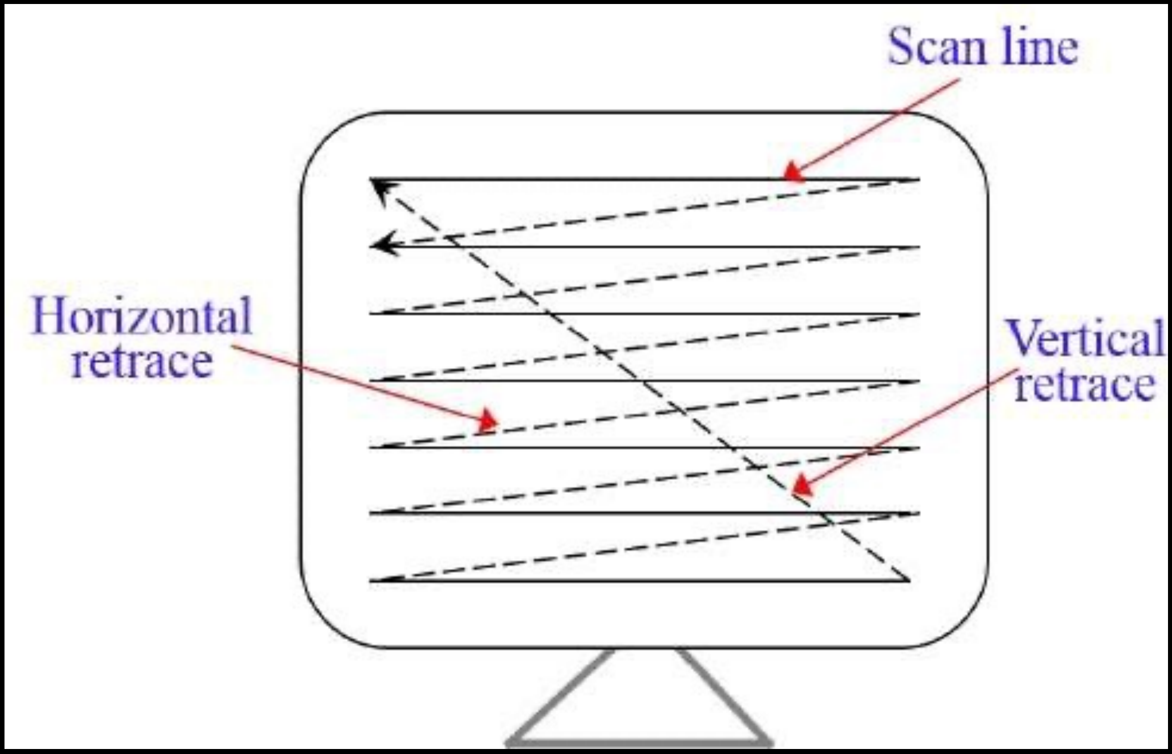
Basic working of Raster Scan

- A beam of an electron (电子束) is moved across the screen. It moves from top to bottom considering one row at a time
- As the beam of electron moves through each row, its intensity is alternatively turned on and off, which helps to create a pattern of spots that are illuminated



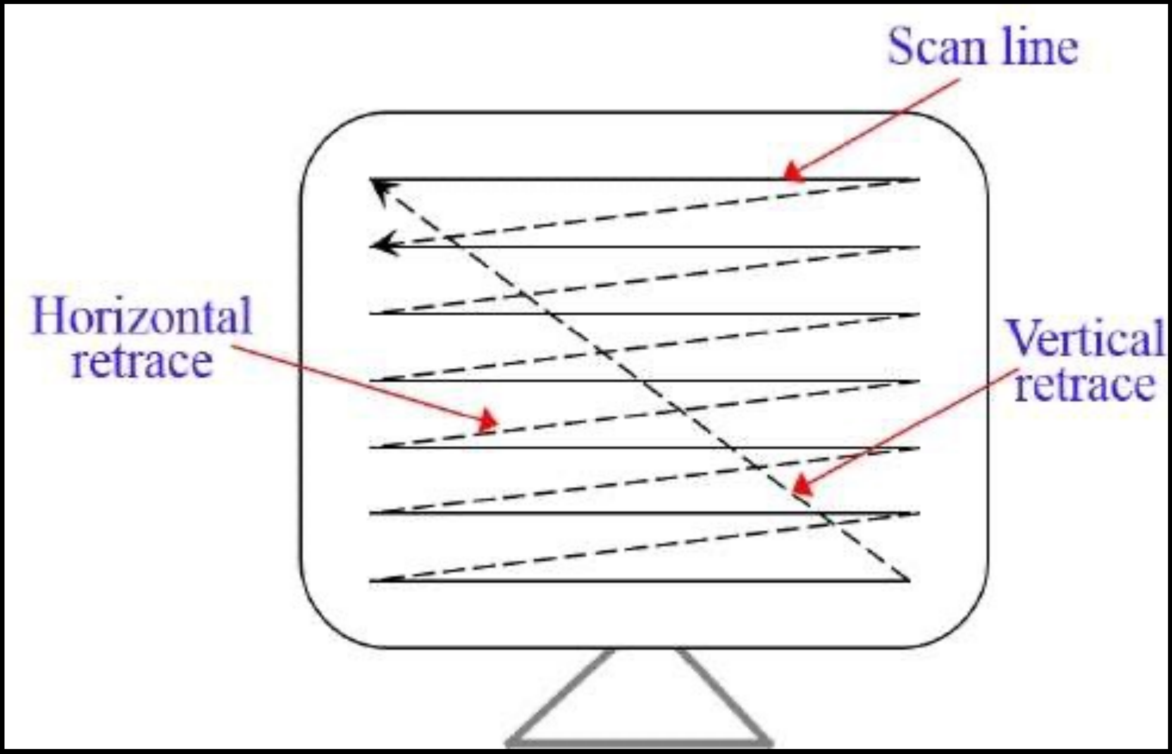
Basic working of Raster Scan

- When each scan of the line is refreshed it returns to the left side of the screen —— **Horizontal retrace(水平回扫)**
- As a particular frame ends, the beam of electron moves to the left top corner of the screen to move to another frame —— **Vertical retrace(垂直回扫)**



Basic working of Raster Scan

- The **frame buffer** in a raster scan is that area that is responsible for containing intensity of the various points on the screen.
- The values stored in the buffer are then fetched and traced over scan lines one by one on the screen.



Definitions

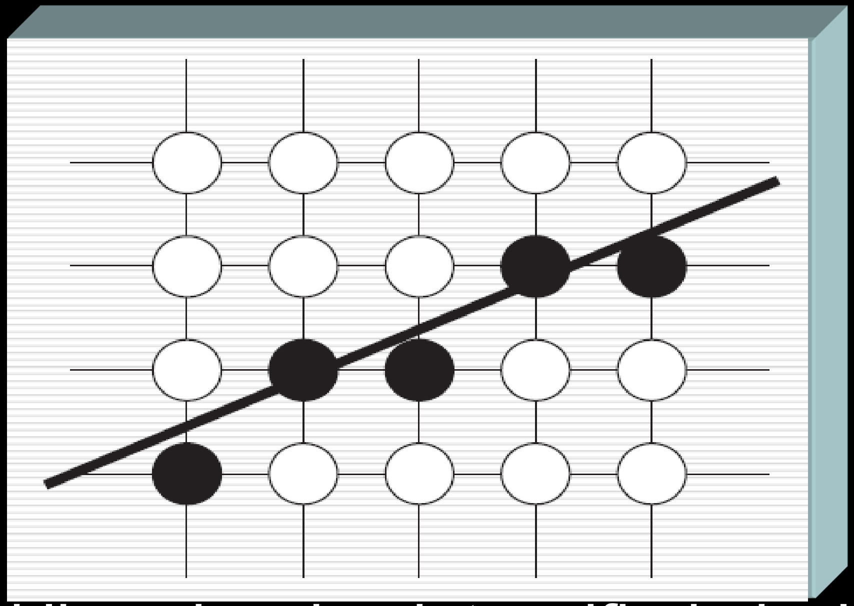
- Pixel: a screen consists of N x M pixels
- Bilevel = monochrome (2 color image), 1 bit / pixel
- Color: RGB/CYK/LUV…
- Bitmap / pixmap
- Frame buffer: an array of data in memory mapped to screen

Scan Converting Algorithms

- Approximate mathematical “ideal” primitives, by sets of pixels on a raster display
- Scan converting primitives to pixels
- Want efficiency & speed

Scan Converting Line

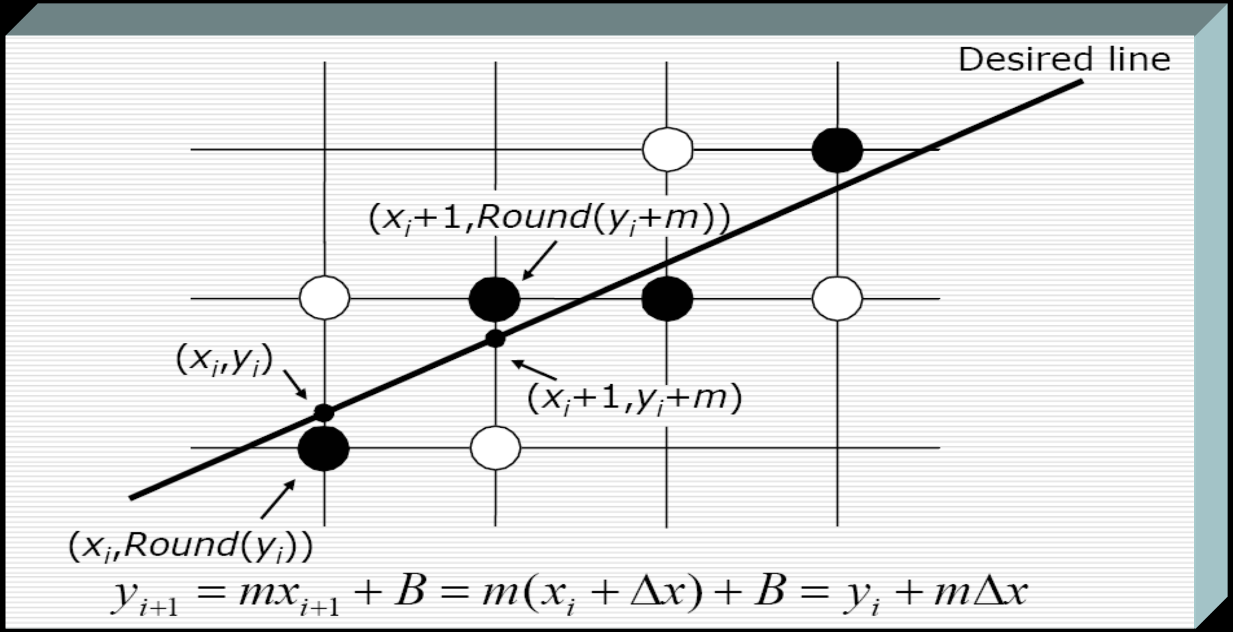
- Approximate a line by sets of pixels
- Idea: compute the coordinates of pixels that lies on or near a line
- Assumptions: 1) endpoints coordinates are integers; 2) pixel on or off (2 states); 3) slope $|m| \leq 1$



A scan-converted line showing intensified pixels as black circles

The Basic Incremental Algorithm

- Line slope intercept form(斜截式): $y=mx+B$
- Idea: start from x_0, increment x by 1 and calculate $y_i = mx_i + B$
- $x_{i+1} = x_i + 1$, $y_{i+1} = mx_{i+1} + B = y_i + m$
- Pixel (xi, round(yi)) turned on



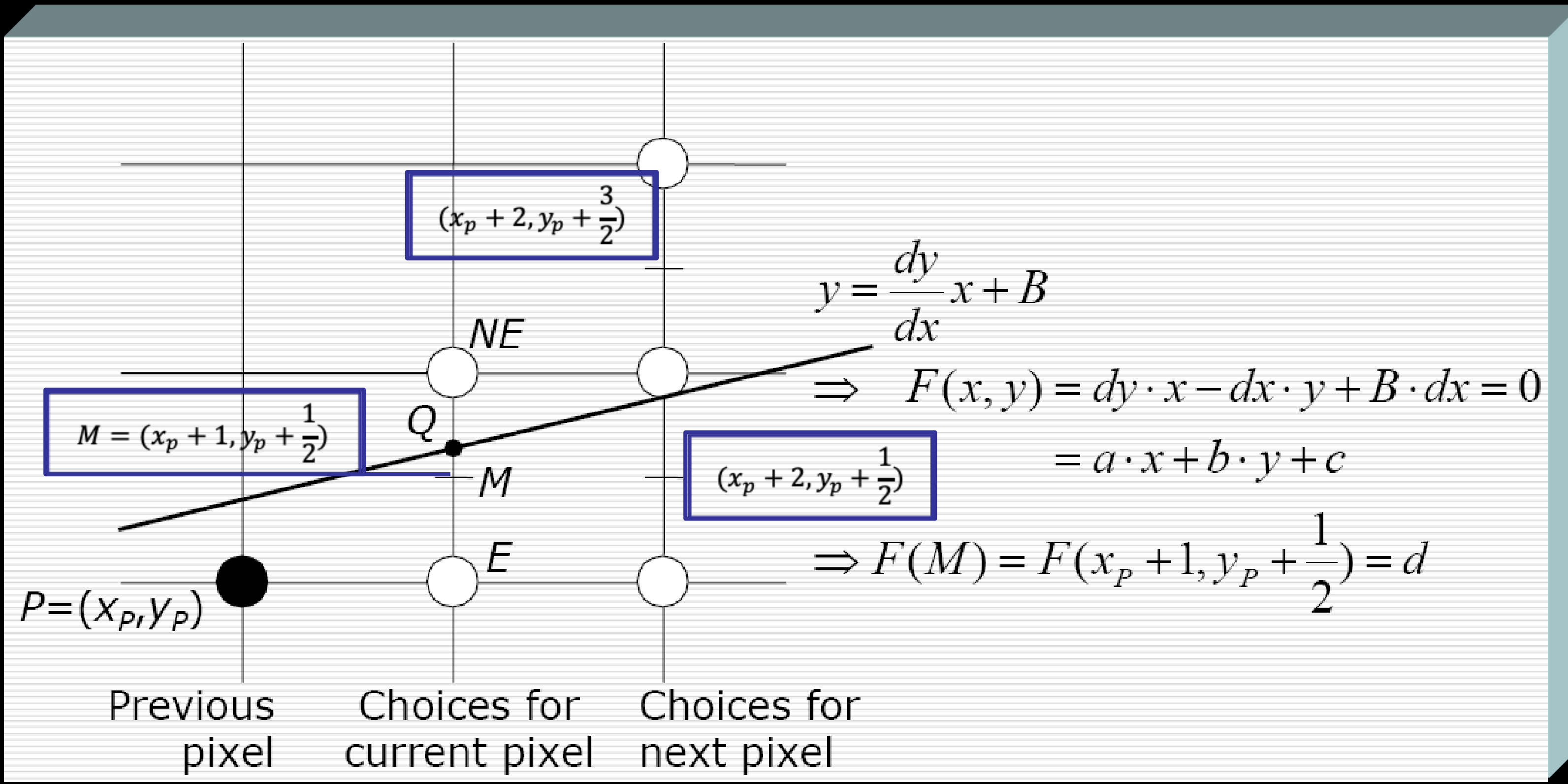
The Basic Incremental Algorithm

start point   end point

```
void Line (int x0, int y0, int x1, int y1, int value)
    {
        int x;
        float dy, dx, y, m;
        dy = y1 - y0;
        dx = x1 - x0;
        m = dy / dx;
        y = y0;
        for(x=x0; x<=x1; x++) {
            WritePixel(x, (int)floor(y+0.5), value);
            y += m;
        }
    }
```

round(y)

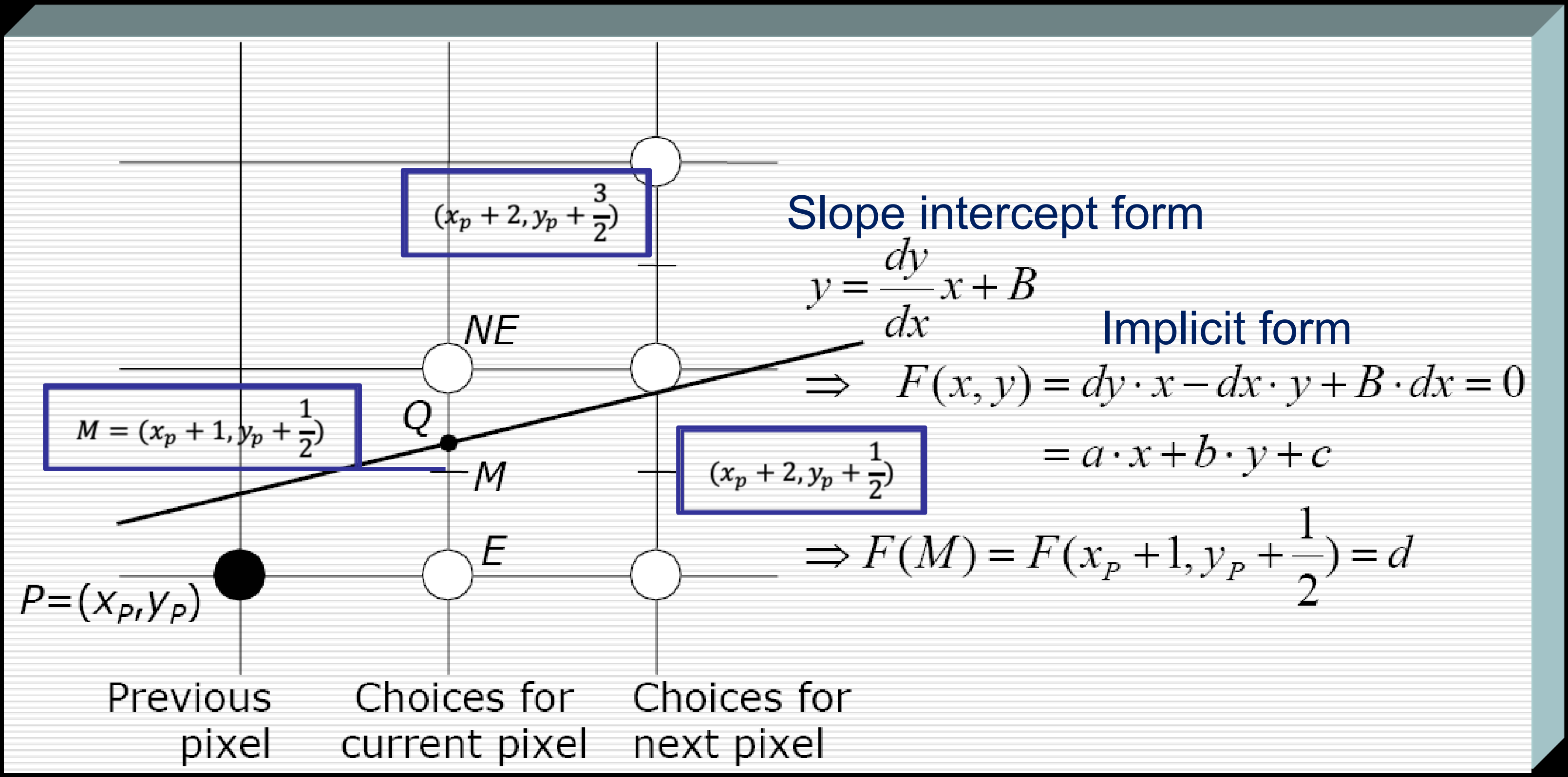
Midpoint Line Algorithm



- If Q > M, pick NE
- If Q < M, pick E
- If Q = M, pick either one

Error will be ≤ 1/2

Midpoint Line Algorithm



- Line implicit form F(x,y)=ax+by+c. a=dy, b=-dx, c=Bdx
- F(x,y)=0 on line; <0 above line; >0 below line
- F(M)=d, if d>0, M<Q, pick NE; d<0, pick E; d=0 either

Midpoint Line Algorithm

How to compute d iteratively?

$$d_{old} = F(x_P + 1, y_P + \frac{1}{2}) = a(x_P + 1) + b(y_P + \frac{1}{2}) + c$$

$$d_{new} = \begin{cases} F(x_P + 2, y_P + \frac{1}{2}) = a(x_P + 2) + b(y_P + \frac{1}{2}) + c & forE \\ F(x_P + 2, y_P + \frac{3}{2}) = a(x_P + 2) + b(y_P + \frac{3}{2}) + c & forNE \end{cases}$$

$$d_{new} = \begin{cases} d_{old} + a & forE \\ d_{old} + a + b & forNE \end{cases}$$

At the beginning:

$d = F(x_0+1, y_0+\frac{1}{2})$
$= F(x_0, y_0) + a + b/2$
$= a + b/2$
$= dy - dx/2$

Change d=F(x,y) to 2F(x,y) to avoid floating point operation

Midpoint Line Algorithm

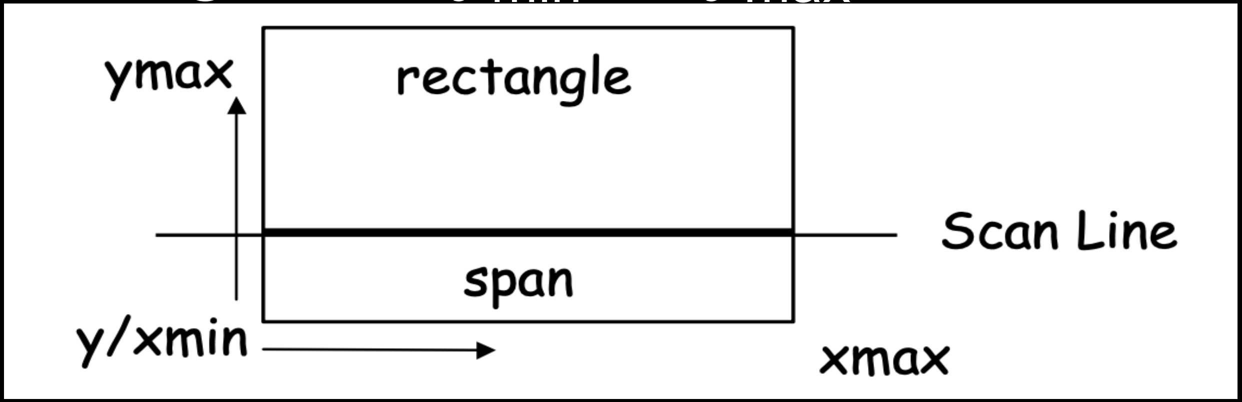
```
void MidpointLine(int x0, int y0, int x1, int y1, int value){
    int dx, dy, incrE, incrNE, d, x, y;
    dy = y1 - y0;
    dx = x1 - x0;
    d = dy * 2 - dx;
    incrE = dy * 2;
    incrNE = (dy - dx) * 2;
    x = x0; y = y0;
    WritePixel(x, y, value);
    while(x<x1) {
        if(d<=0) {
            d += incrE;
            x++;
        }
        else {
            d += incrNE;
            x++;
            y++;
        }
        WritePixel(x, y, value);
    }
}
```

Filling Algorithms

- Decide what pixels to fill
- Decide what value to fill them
- Primitives: Rectangles / Polygons —— scan line algorithms
- Regions of pixels: filling algorithms

Filling Rectangles

- Fill each span from x_{min} to x_{max}, while traveling from y_{min} to y_{max}



for(y from y_{min} to y_{max} of the rectangle) {

  for(x from x_{min} to x_{max}) {

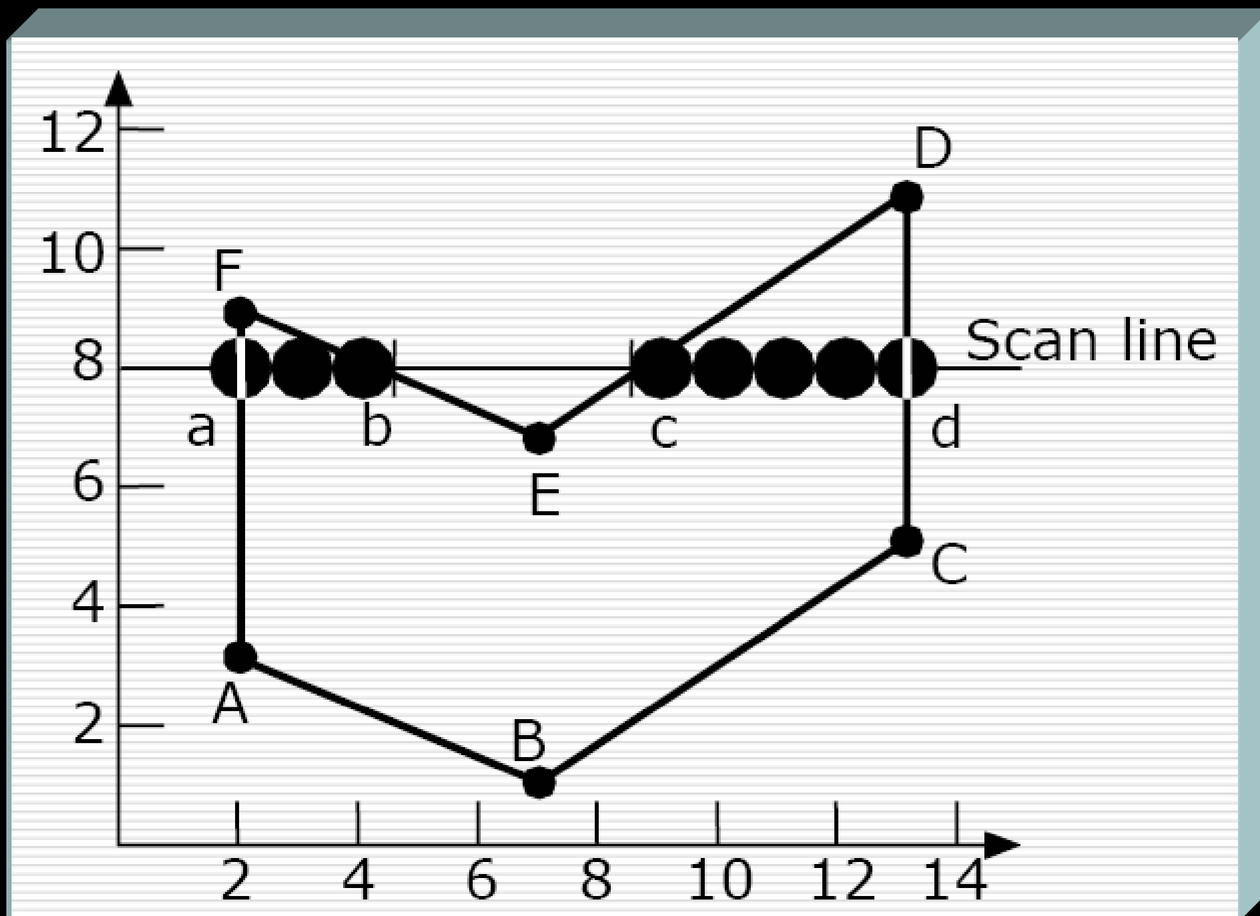
    WritePixel(x, y, $value$);

  }

}

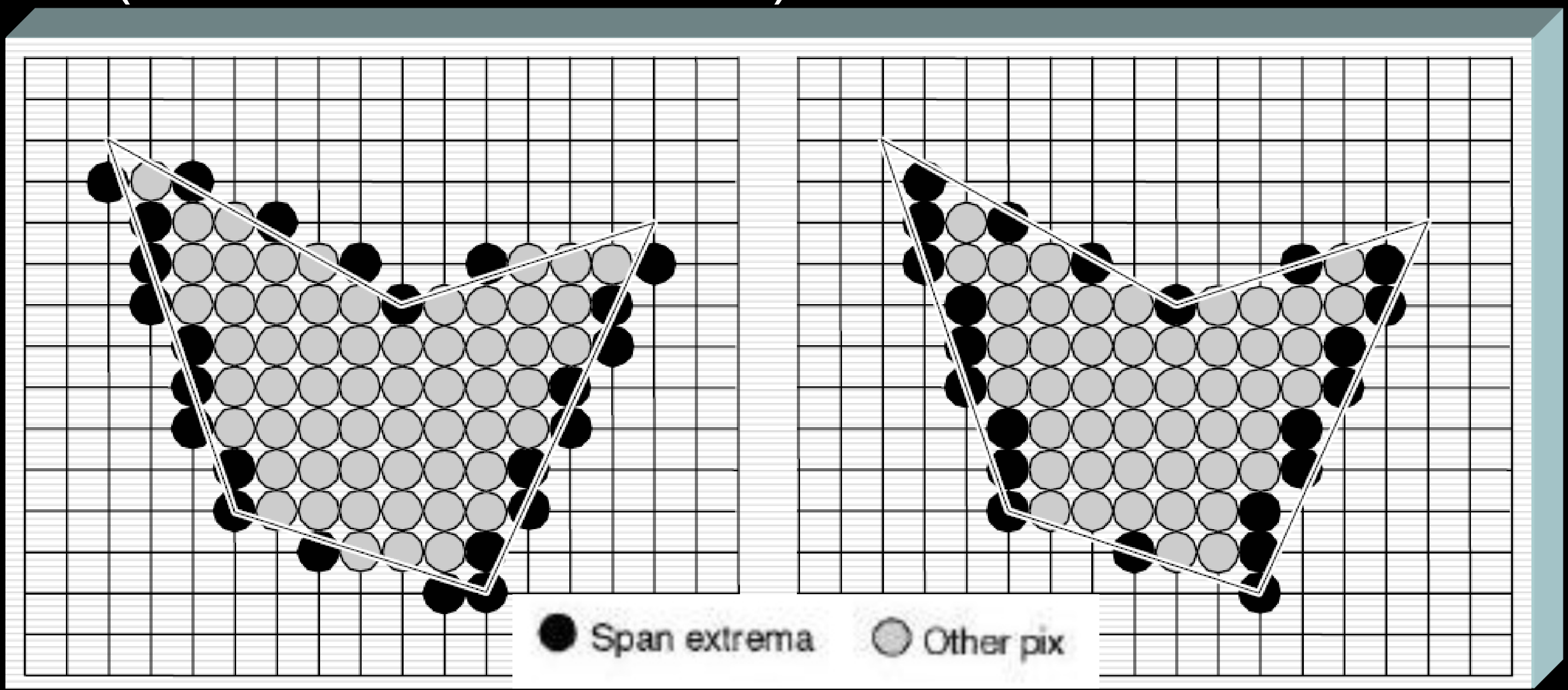
Filling Polygons

- Intersect the polygon with consecutive scan-lines and check for points of intersection



Filling Polygons

- Could determine span extrema (outermost pixels of a span), using midpoint algorithm, but watch out for extrema outside of polygon (want to fill the interior)



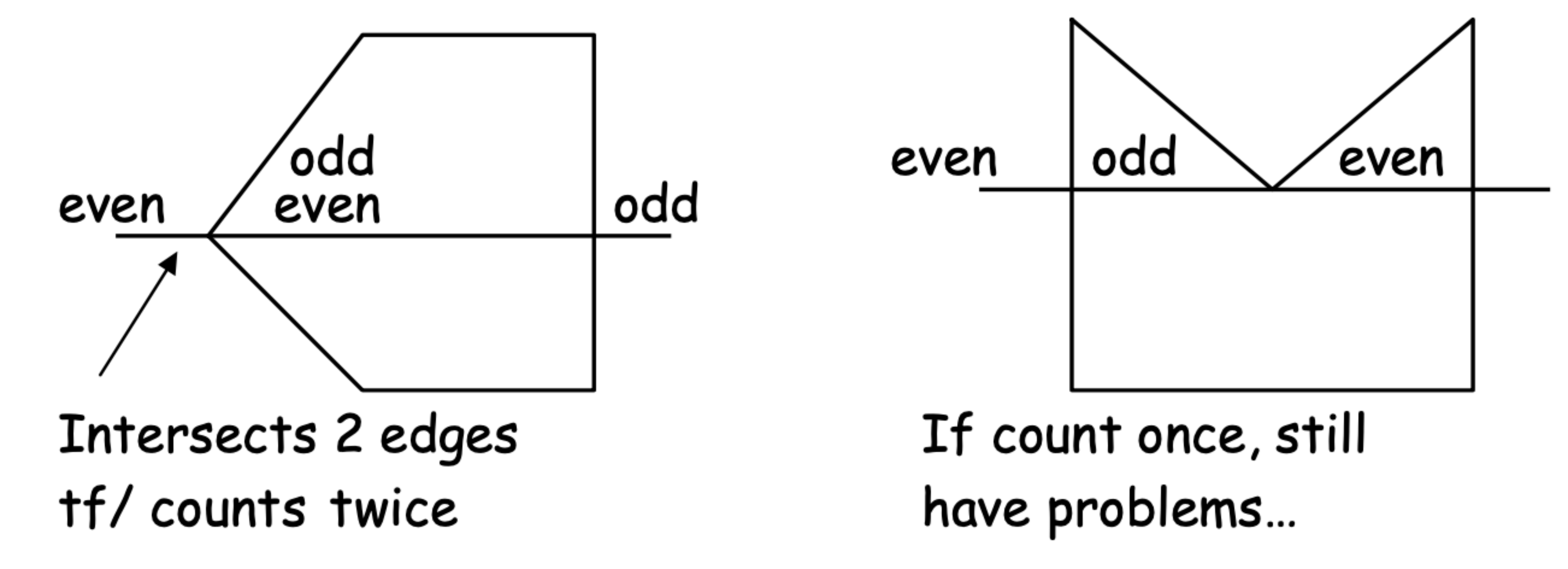
Filling Polygons

Incremental Algorithm:

1. Find the intersections of the scan line with all edges of the polygon
2. Sort the intersections by increasing *x* coordinate
3. Fill in all pixels between pairs of intersections that lie interior to the polygon

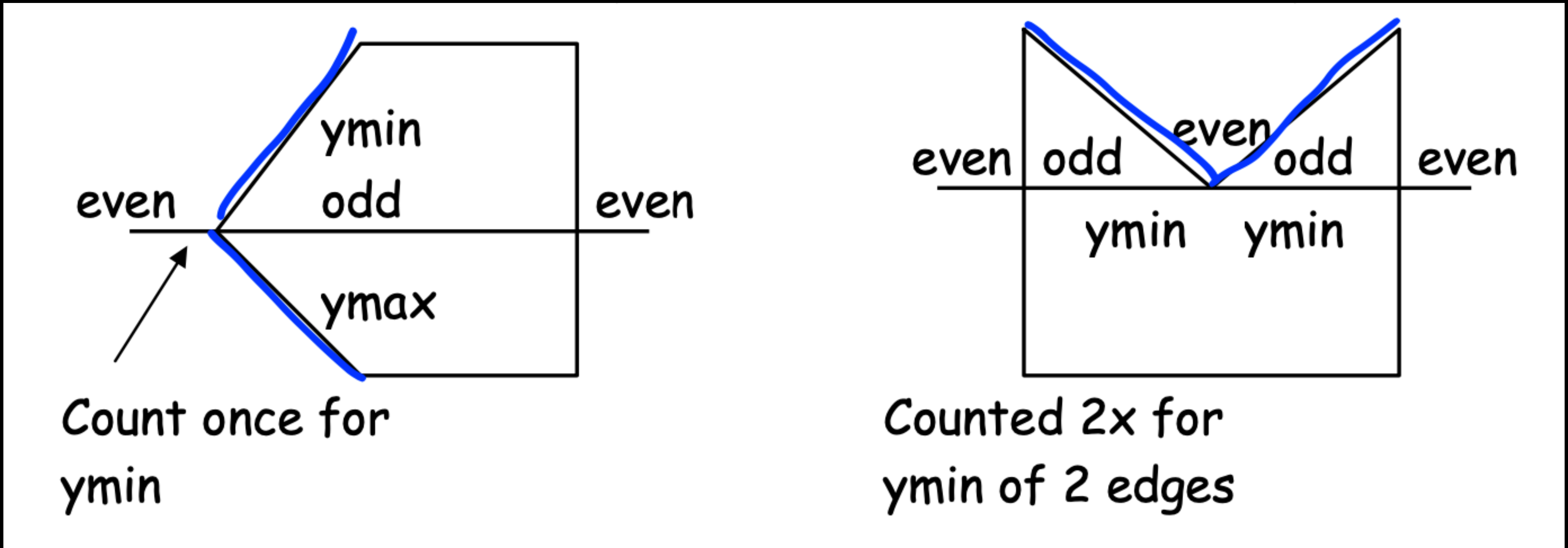
Filling Polygons

- How to decide interior: odd-parity rule
- Parity: initially even, each intersection inverts the parity. Draw when odd only

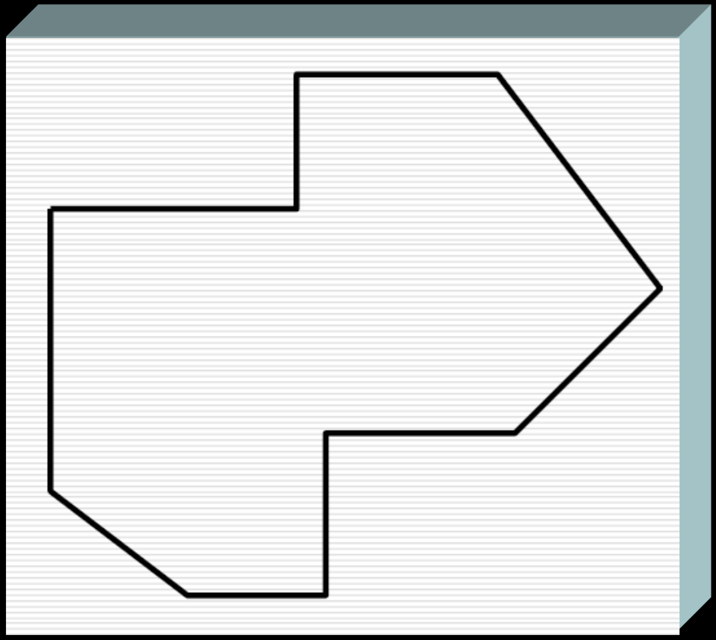


Filling Polygons

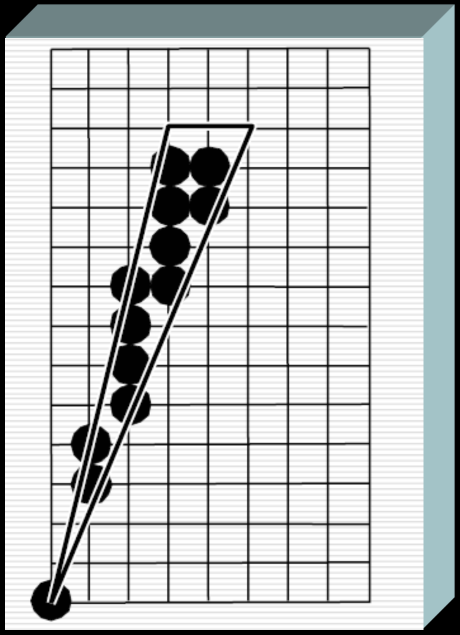
- How to decide interior: odd-parity rule
- Solution: count ymin intersection point of edges, but not ymax intersection



Special Cases



Horizontal Edges



Slivers：

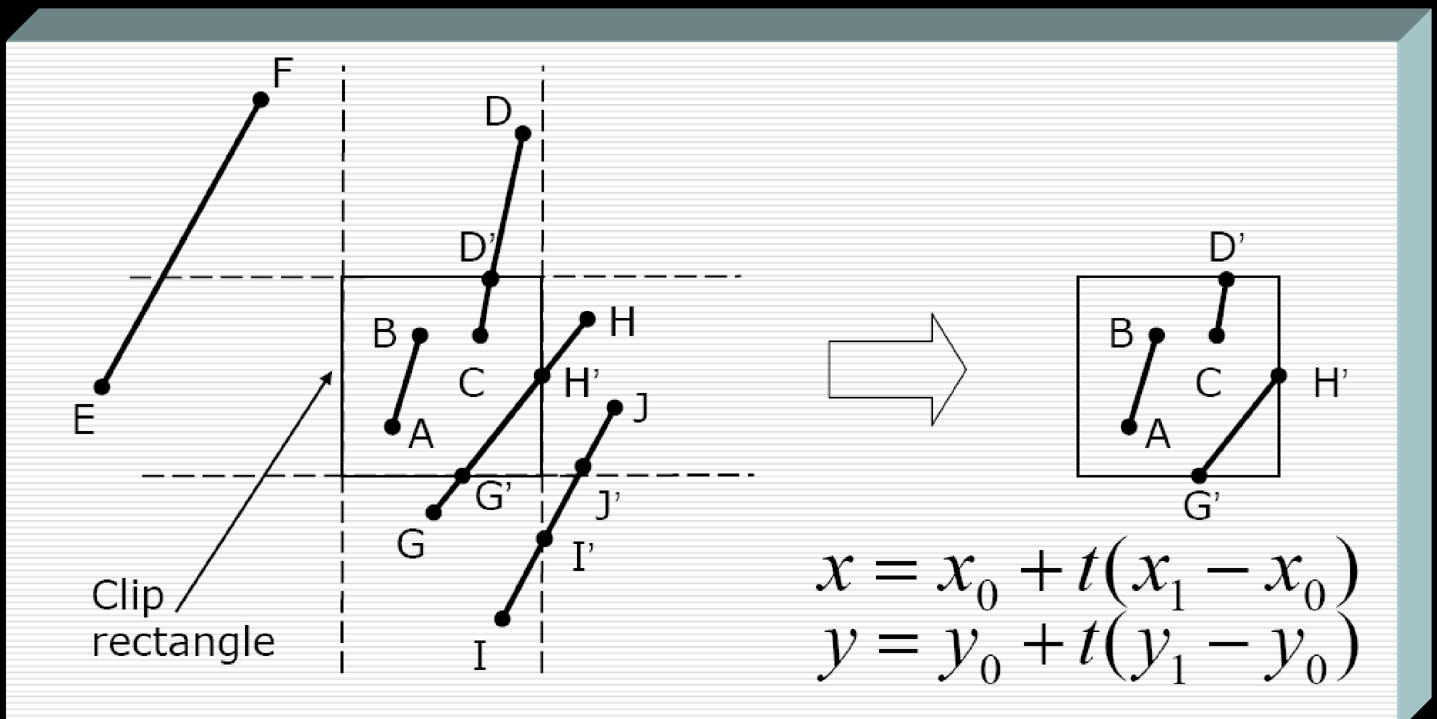
The edges lie so close together that the area does not contain a single pixel

Clipping

- "Clip" objects being drawn that protrude past the bounds of the display/window
- Why clipping: waste of time drawing objects that aren't going to be visible (inefficient)
- When to clip: before / during / after scan conversion

Clipping Lines

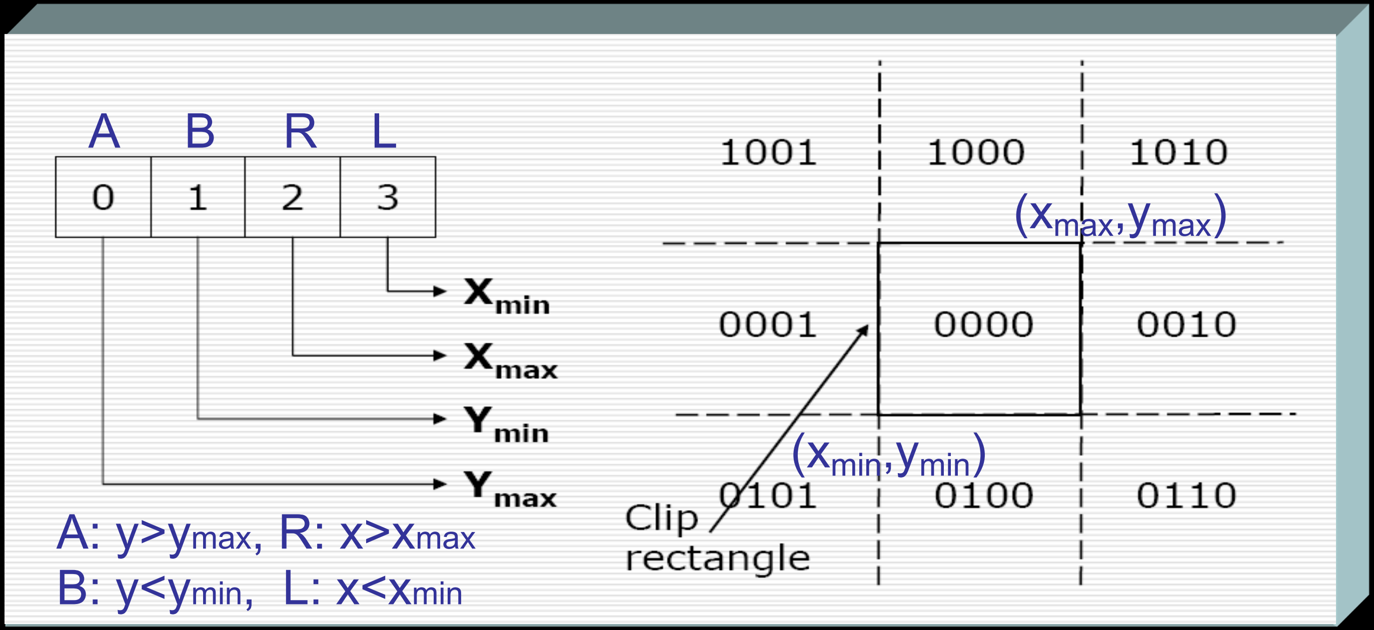
- Lines are clipped into line segments
- Can check whether a line needs to be clipped by looking at its endpoints



$$x = x_0 + t(x_1 - x_0)$$
$$y = y_0 + t(y_1 - y_0)$$

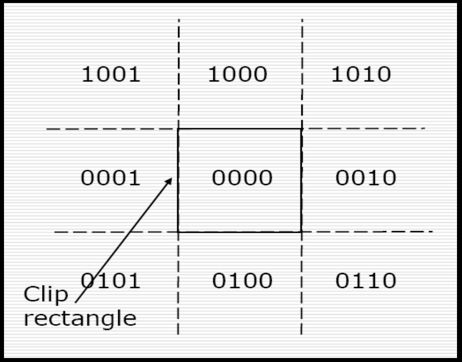
The Cohen-Sutherland Algorithm

- Determines whether intersection calculations can be avoided by performing "region checks"
- Assign a 4-bit code to the clipping rectangle AND the surrounding regions
- Code ABRL (Above, Below, Right, Left)

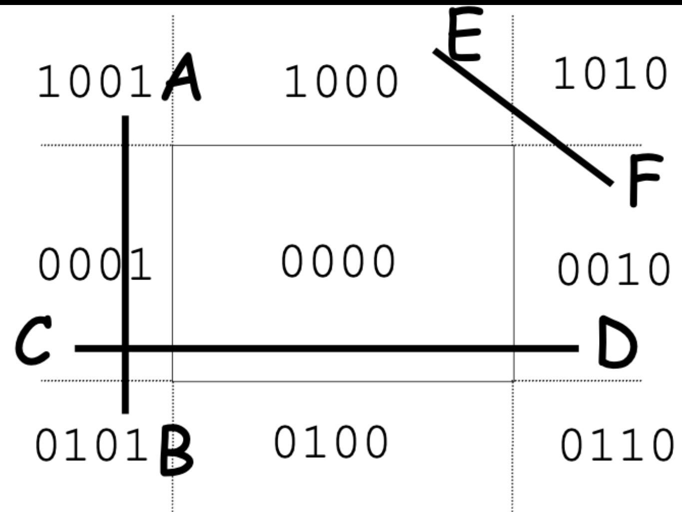


The Cohen-Sutherland Algorithm

- Steps：
- 1. Calculate the code of the 2 line endpoints
- 2. Check for trivial acceptance (both codes = 0000)
- 3. Perform bitwise AND of codes
- 4. Trivially reject if result ≠ 0000 (both endpoints above/below/right/left of the clipping rectangle)
- 5. Choose an endpoint outside the clipping rectangle. Test its code to determine which clip edge was crossed and find the intersection of the line and that clip edge (test the edges in a consistent order).
- 6. Replace endpoint (selected above) with intersection point
- 7. Repeat



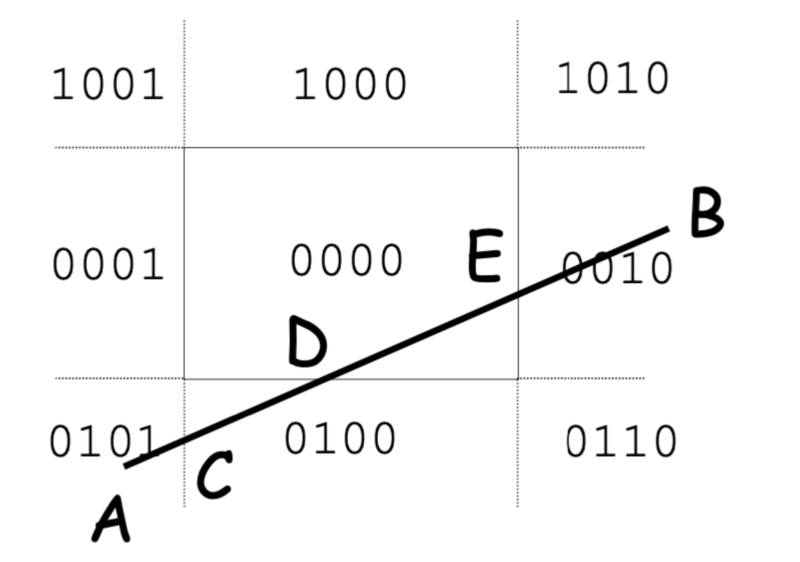
The Cohen-Sutherland Algorithm



A	1001	C	0001
B	0101	D	0010
	0001 Reject		0000 Intersect

E	1000
F	0010
	0000 Intersect

The Cohen-Sutherland Algorithm



A 0101

B 0010

0000 Intersection

- A is outside, intersection test yields C, replace A by C

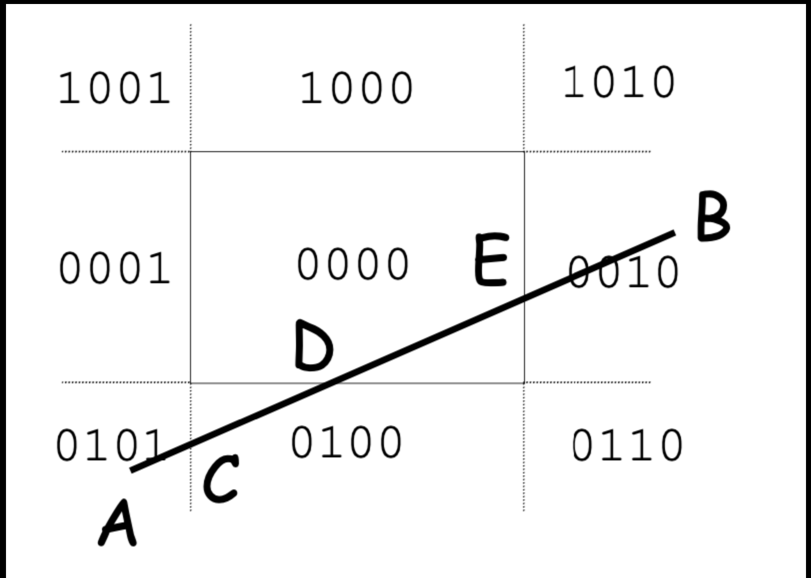
C 0100

B 0010

0000 Intersection

- C replaced by D

The Cohen-Sutherland Algorithm



D 0000

B 0010

0000 Intersection

- B replaced by E

E 0000

D 0000 Accept DE

Liang-Barsky Algorithm

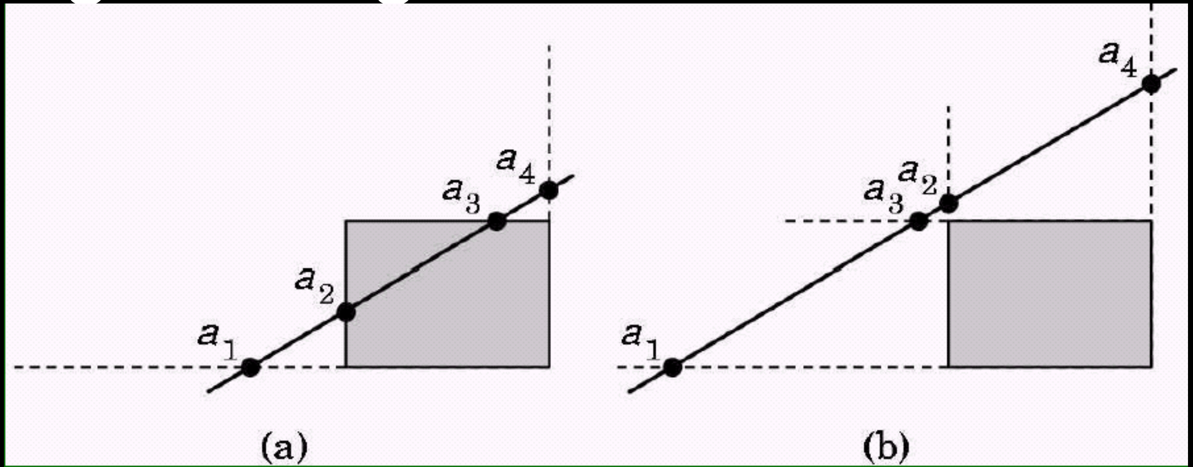
- Represent line in parametric form

$$P(\alpha) = (1-\alpha)P_1 + \alpha P_2, \qquad 0 \le \alpha \le 1$$

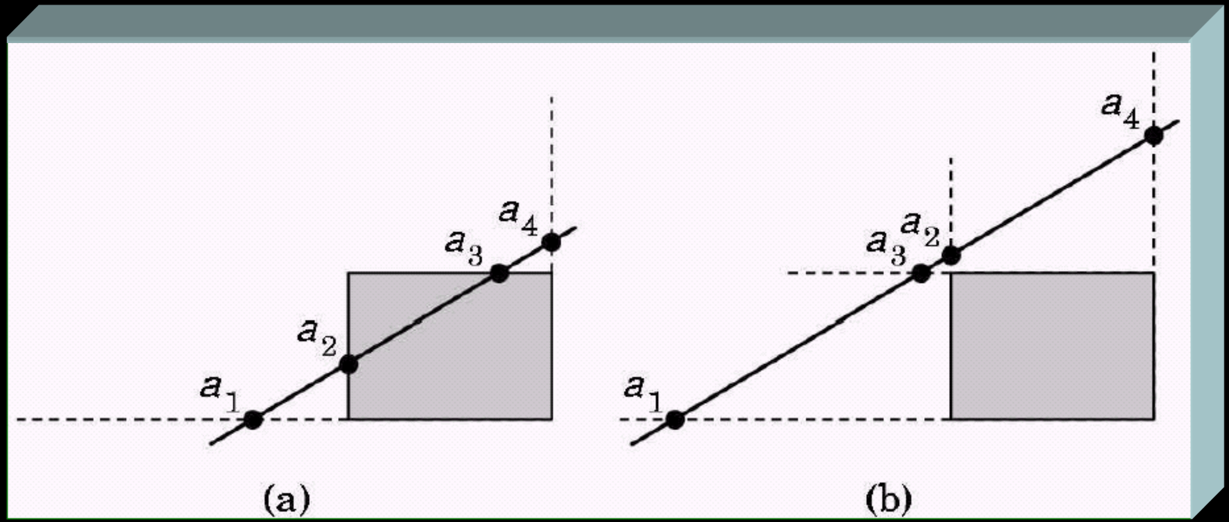
$$x(\alpha) = (1-\alpha)x_1 + \alpha x_2$$

$$y(\alpha) = (1-\alpha)y_1 + \alpha y_2$$

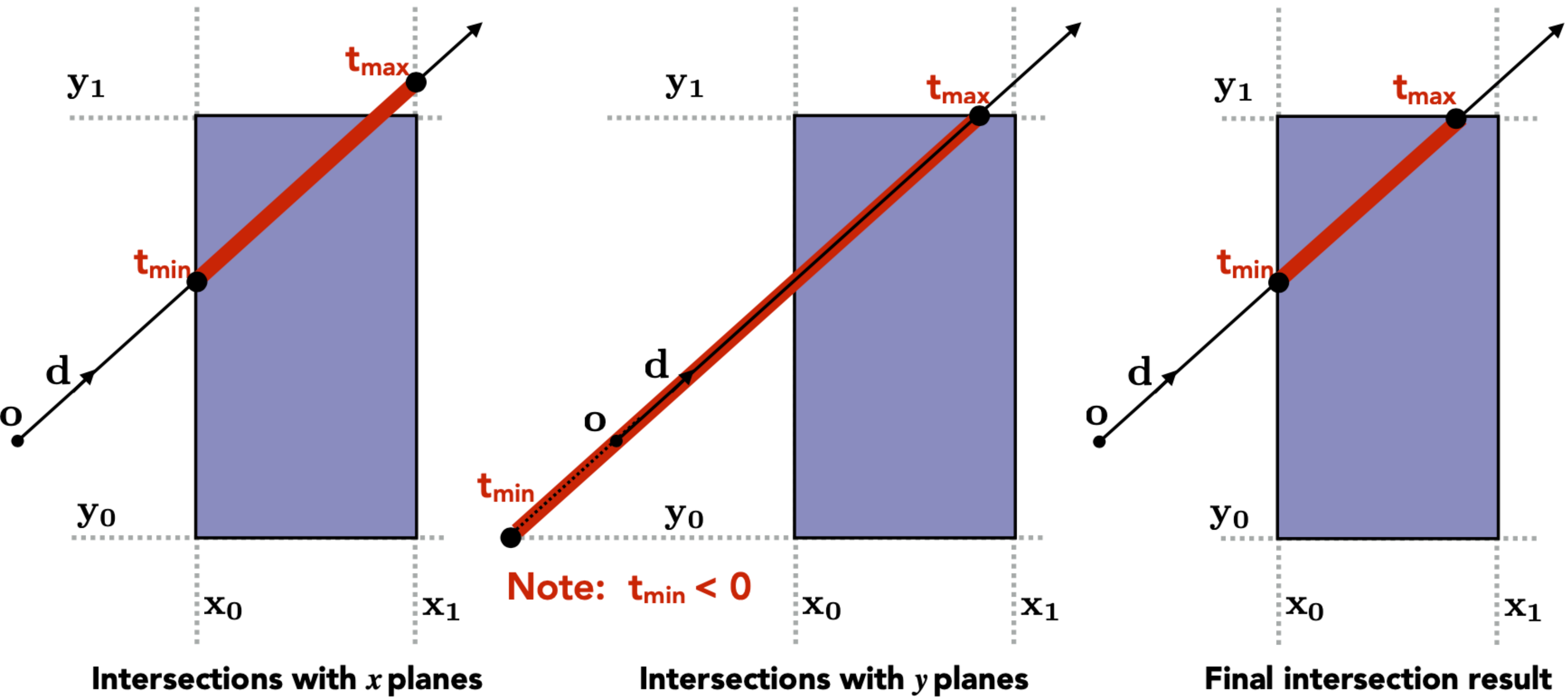
- Compute four intersections with extended clipping rectangle



Ordering of intersection points



- x: $\alpha 2$(enter), $\alpha 4$(exit);
- y: $\alpha 1$(enter), $\alpha 3$(exit)
- Order the intersection points
- Figure (a): $1 > \alpha 4 > \alpha 3 > \alpha 2 > \alpha 1 > 0$, intersect
- Figure (b): $1 > \alpha 4 > \alpha 2 > \alpha 3 > \alpha 1 > 0$, no intersection



Slides from GAMES101

- Key ideas
  - The ray enters the box **only when** it enters all pairs of slabs
  - The ray exits the box **as long as** it exits any pair of slabs
- For each pair, calculate the t_{min} and t_{max}

$$t_{enter} = \mathbf{max}\{t_{min}, 0\},\ t_{exit} = \mathbf{min}\{t_{max}, 1\}$$

- If $t_{enter} < t_{exit}$, we know the ray **stays a while** in the box (so they must intersect!)

Slides from GAMES101

Liang-Barsky Efficiency Improvements

- Efficiency improvement 1:
  - Compute intersections one by one
  - Often can reject before all four are computed
- Efficiency improvement 2:
  - Equations for α_3, α_2

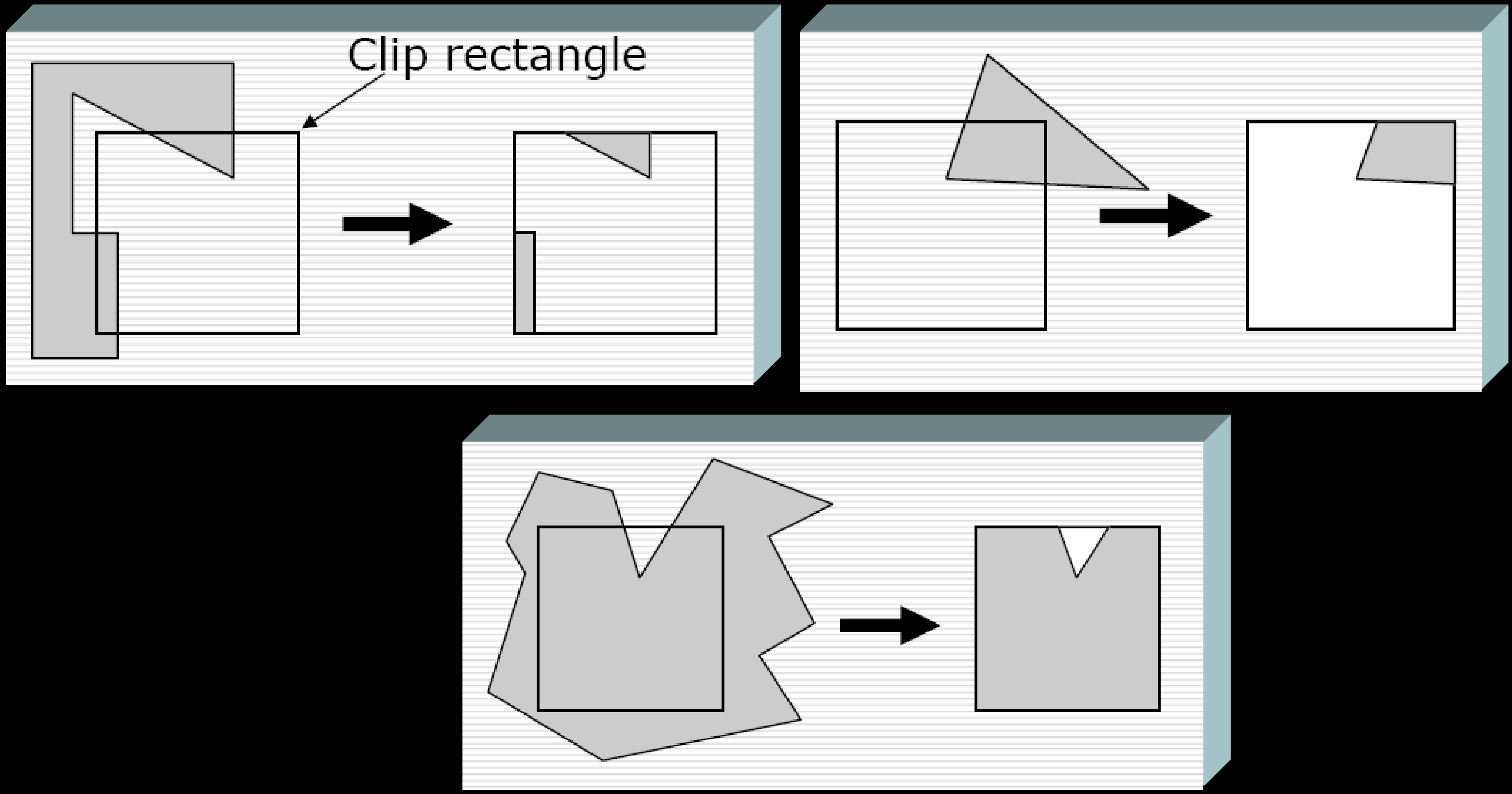
$$y_{\max} = (1-\alpha_3)y_1 + \alpha_3 y_2$$

$$x_{\min} = (1-\alpha_2)x_1 + \alpha_2 x_2$$

$$\alpha_3 = \frac{y_{\max} - y_1}{y_2 - y_1} \qquad \alpha_2 = \frac{x_{\min} - x_1}{x_2 - x_1}$$

  - Compare α_3, α_2 without floating-point division

Clipping Polygons

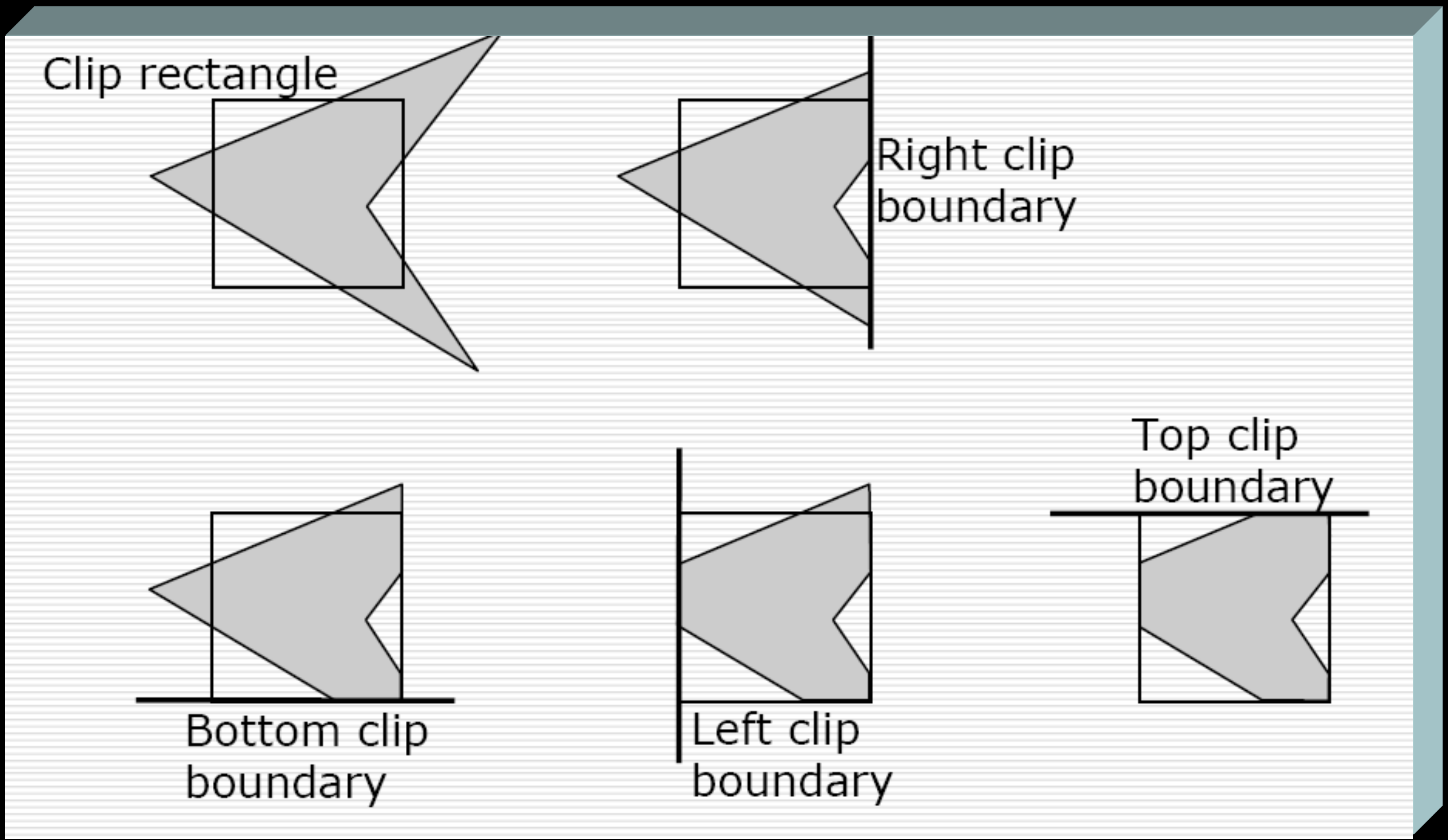


The Sutherland-Hodgman Polygon-Clipping Algorithm

- Divide & Conquer strategy
  - subproblem: clip polygon against 1 clip edge
  - at each step, the partially clipped polygon is clipped against the next edge and so on...

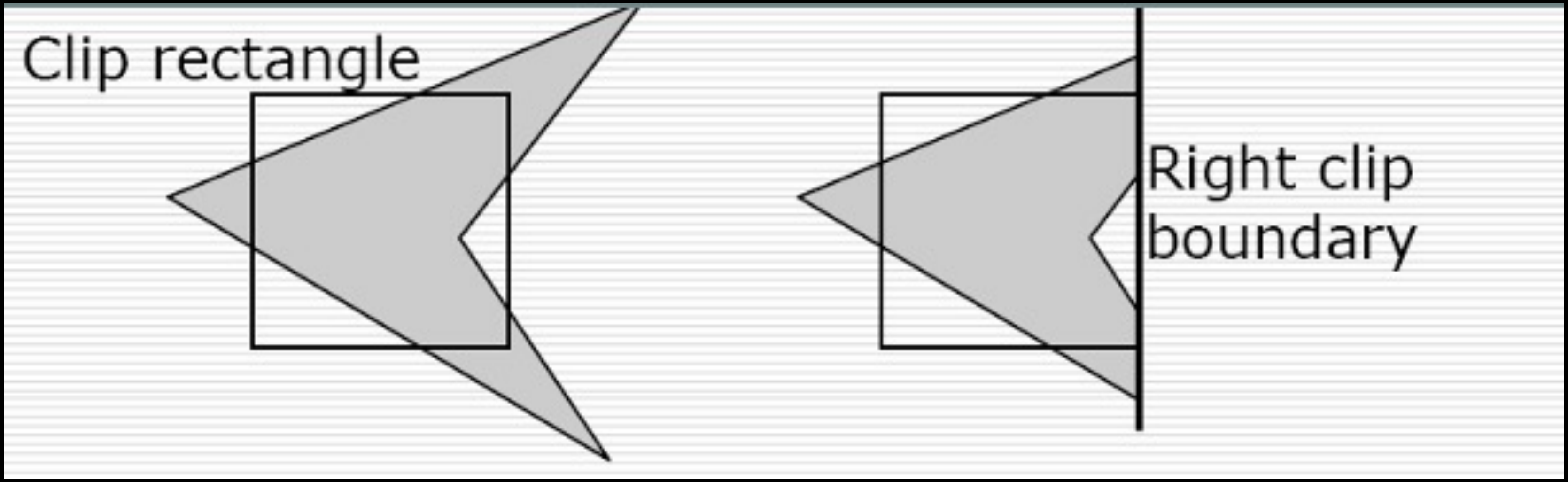
The Sutherland-Hodgman Polygon-Clipping Algorithm

Each time clip polygon against 1 clip edge



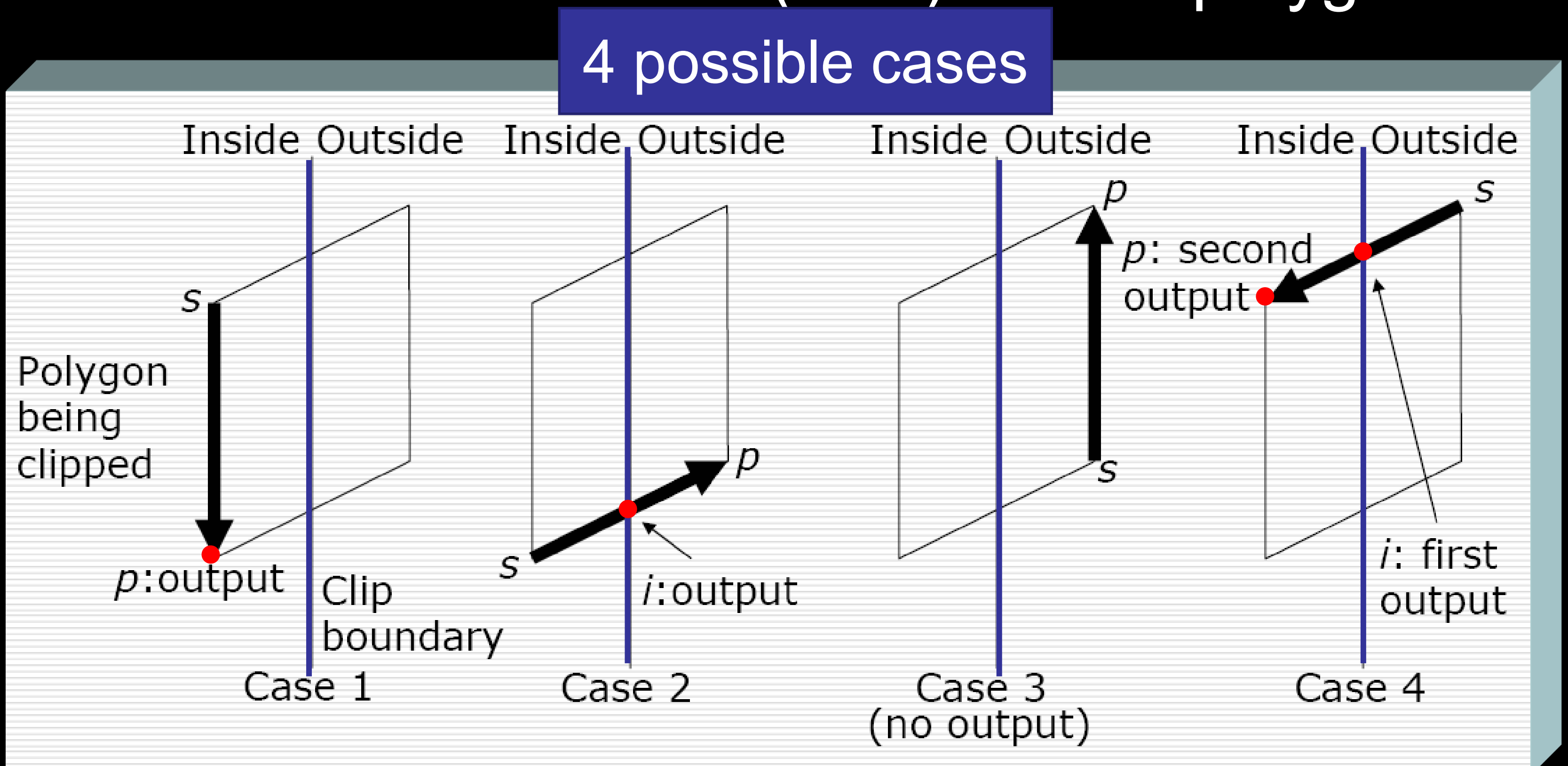
The Sutherland-Hodgman Polygon-Clipping Algorithm

- for each clipping edge
  - for each polygon edge
    - clip polygon edge to clip edge
    - store vertices (new) to new polygon



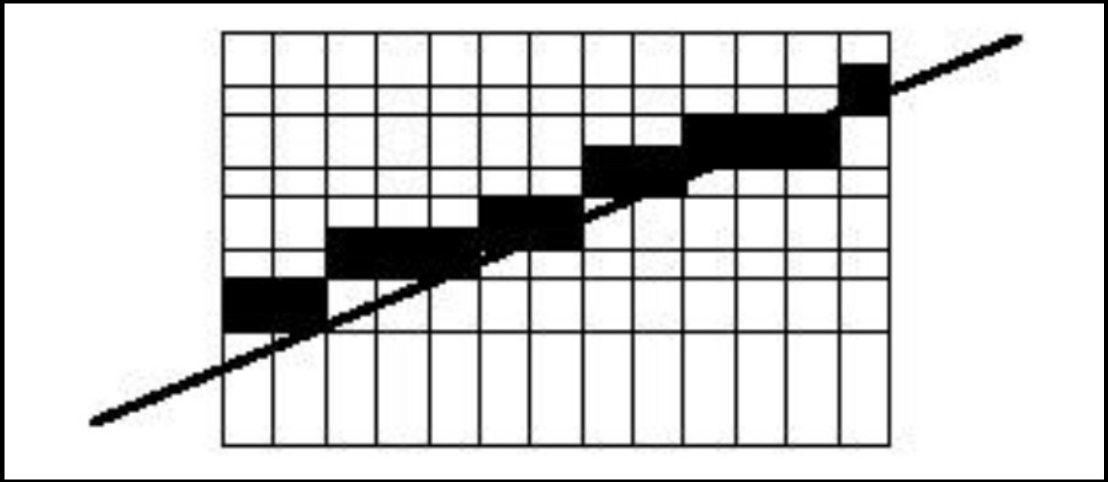
The Sutherland-Hodgman Polygon-Clipping Algorithm

- for each clipping edge
  - for each polygon edge
    - clip polygon edge to clip edge
    - store vertices (new) to new polygon



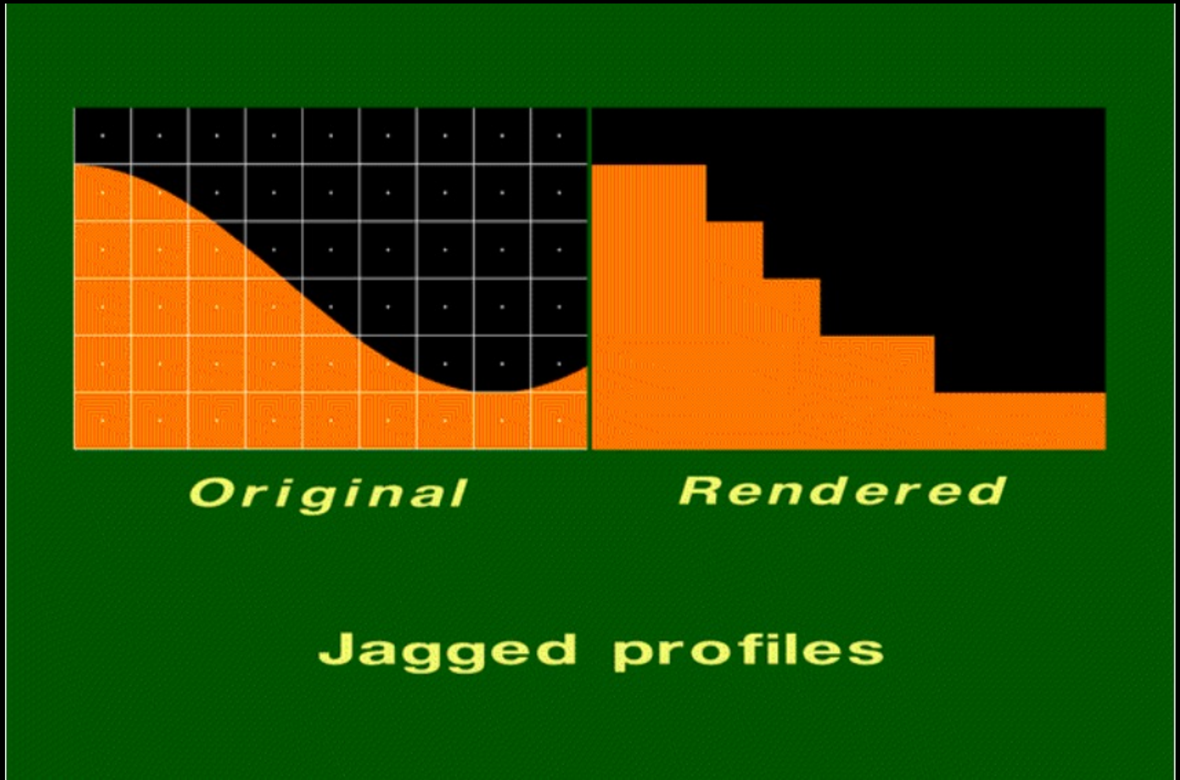
Aliasing

- Inherent property of raster displays
- Due to discrete sampling of a continuous primitive
- Problem:
  - staircases or jaggies
  - result of “all or nothing” approach (pixels are ON or OFF)



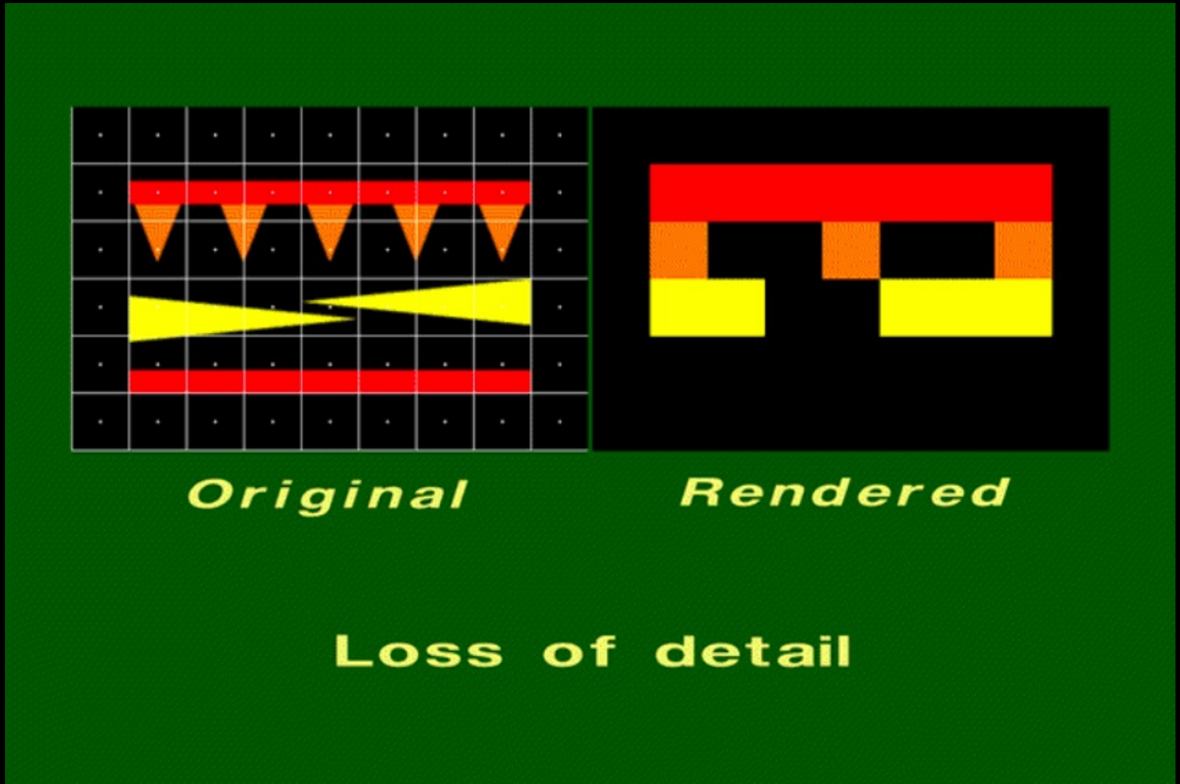
Errors caused by Aliasing

- Jagged Profiles
- Improperly Rendered Detail
- Disintegrating textures



Errors caused by Aliasing

- Jagged Profiles
- Improperly Rendered Detail
- Disintegrating textures



Errors caused by Aliasing

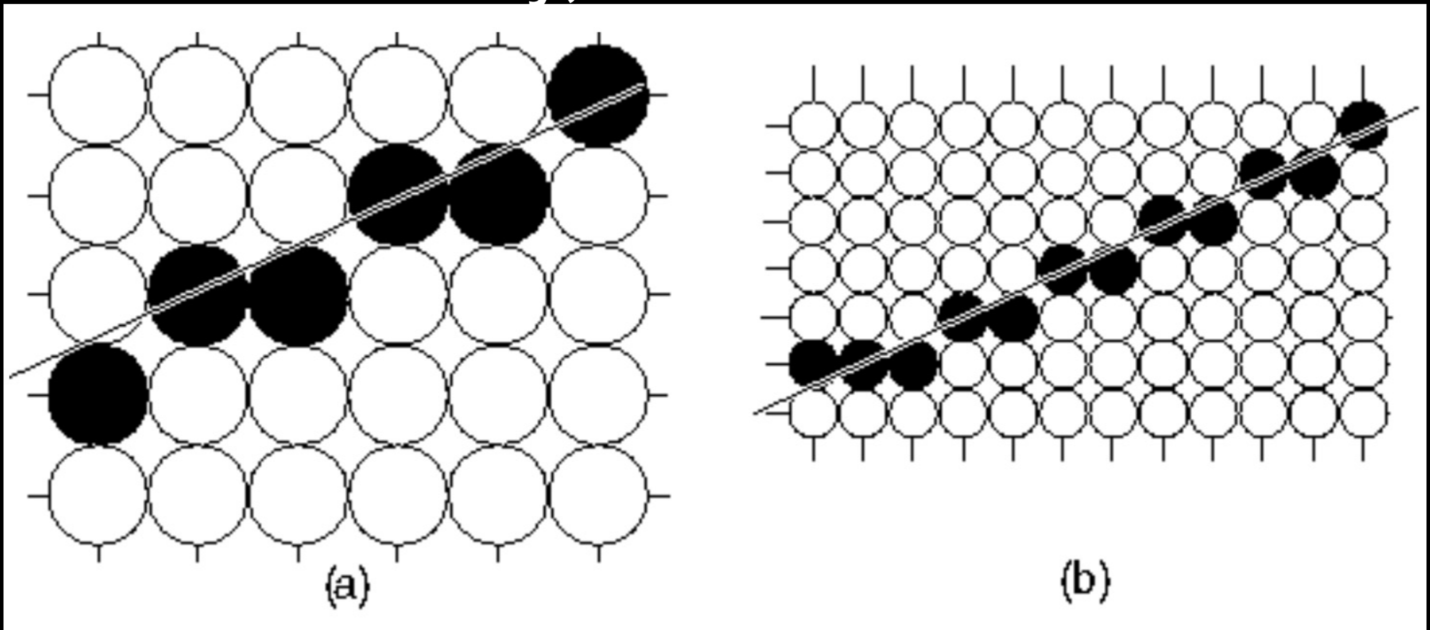
- Jagged Profiles
- Improperly Rendered Detail
- Disintegrating textures

Antialiasing

- “Blurring” the edges to “smooth” the image
- Three ways of anti-aliasing:
- 1. increase resolution
- 2. pre-filtering
- 3. post-filtering

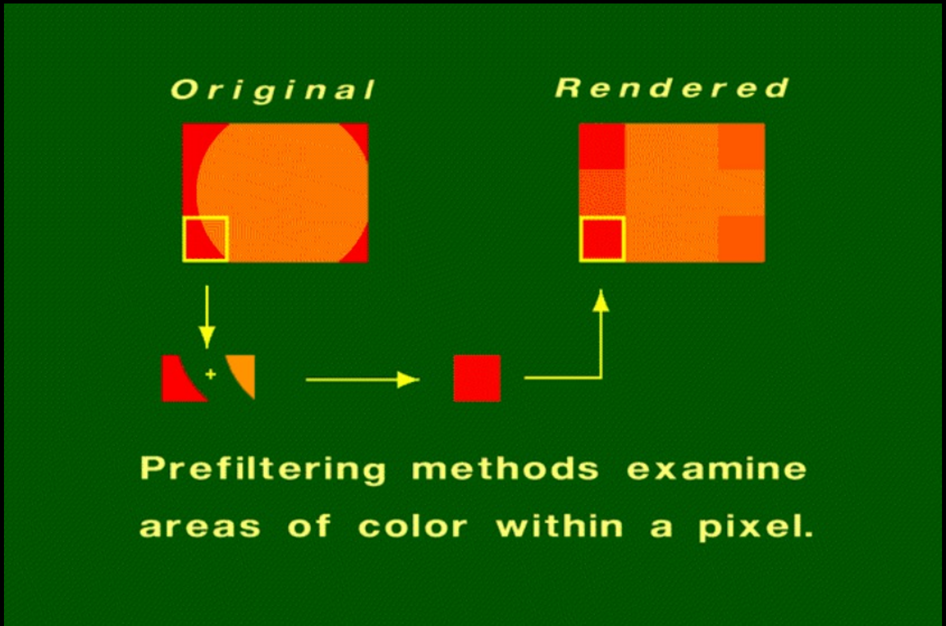
Antialiasing

- 1. Increase resolution
- twice as many jaggies, but all are 1⁄2 the size so they are less noticeable
- cost: 4x memory, 2x scan conversion time



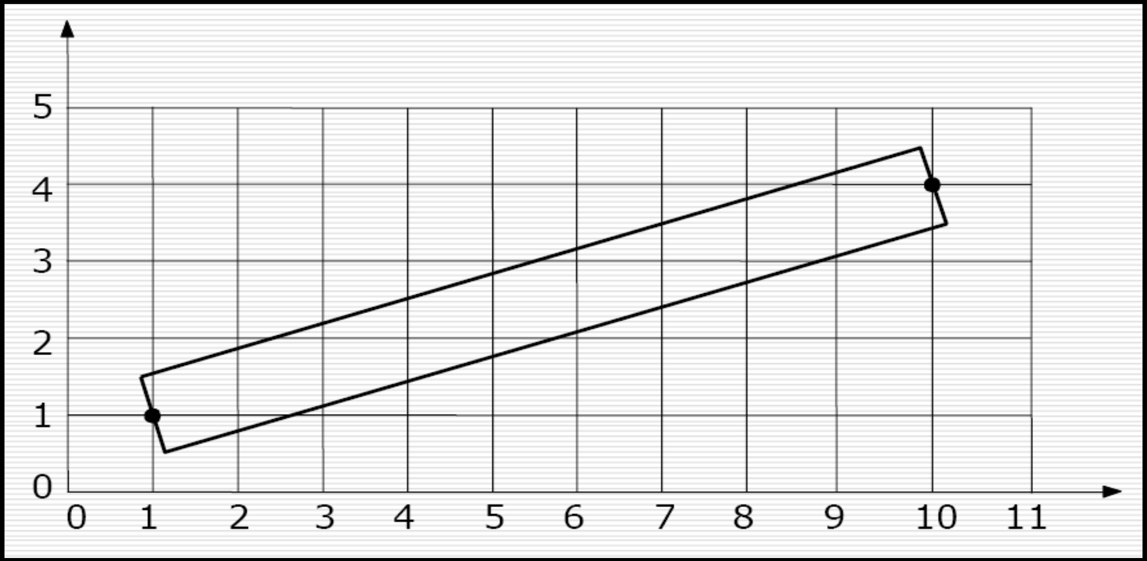
Antialiasing

- 2. Pre-filtering
- treat a pixel as an area
- compute pixel color based on the overlap of the scene's objects with a pixel's area



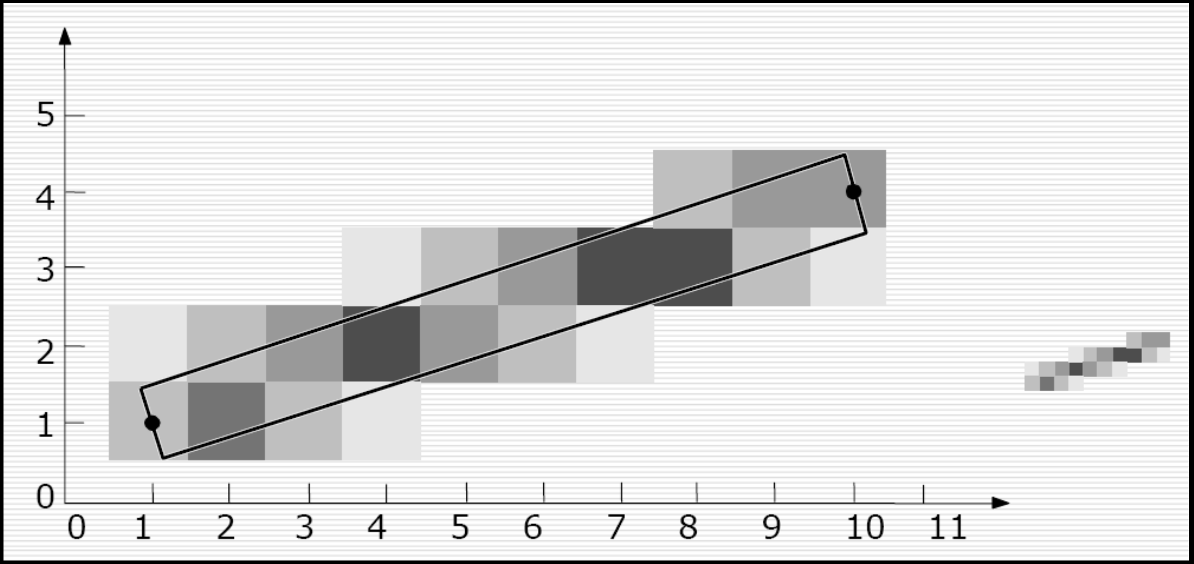
Antialiasing

- 2. Pre-filtering
- a line treated as a rectangle with width
- the intensity of a pixel is proportional to how much is covered by the line



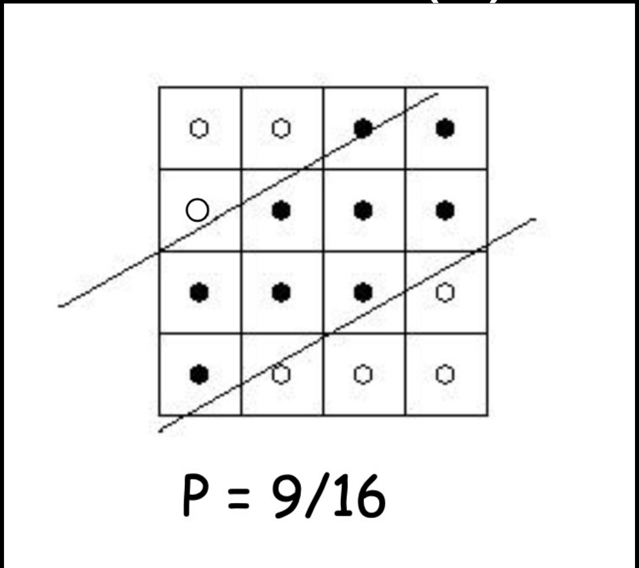
Antialiasing

- 2. Pre-filtering
- determine the percentage of a pixel that is covered



Antialiasing

- 2. Pre-filtering
- can approximate the area of overlap by subdividing a pixel into n rectangular subpixels, then counting the number of subpixels inside the line (k), area ≈ k/n



Antialiasing

- 2. Pre-filtering

Original Image:

Pre-filtered Image:

Antialiasing

- 3. Post-filtering
- computes each display pixel as a "weighted" average of an appropriate set of neighboring
- a filter is used to weight each neighboring sample

Antialiasing

- 3. Post-filtering

$$\begin{bmatrix} \frac{1}{16} & \frac{1}{16} & \frac{1}{16} \\ \frac{1}{16} & \frac{1}{2} & \frac{1}{16} \\ \frac{1}{16} & \frac{1}{16} & \frac{1}{16} \end{bmatrix}$$

5	12	31	36	9	11
18	27	28	16	4	
31	53	42	30	17	
50	52	53	60	62	

$$\frac{1}{2}(30)+\frac{1}{16}(28+16+4+42+17+53+60+62)=32.625$$

Antialiasing

- 3. Post-filtering
- Filters

$$\frac{1}{8}\begin{bmatrix} 0 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \qquad \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Antialiasing

- 3. Post-filtering
- Filters

$$\frac{1}{81}\begin{bmatrix} 1 & 2 & 3 & 2 & 1 \\ 2 & 4 & 6 & 4 & 2 \\ 3 & 6 & 9 & 6 & 3 \\ 2 & 4 & 6 & 4 & 2 \\ 1 & 2 & 3 & 2 & 1 \end{bmatrix}$$