

Computer Graphics: Rasterization Methods

**Dr. Michael Laszlo
Nova Southeastern University
School of Computer and Information Sciences
2002**

Rasterization

- *Rasterization*, also known as *scan conversion*, is the process of representing a continuous mathematical object as a pixelmap (a matrix of pixels). At its simplest, the problem is this:
 - Given a mathematical object such as a line, a polygon, or an ellipse, what pixels should be painted to best represent the object?
- Rasterization is a critical part of computer graphics, for it allows the user to think and work at a higher level of abstraction: in terms of graphical objects, rather than in terms of pixels.
- Rasterization methods are generally implemented in hardware and on graphics cards in today's systems.
- We will look at three rasterization methods:
 - Line scan-conversion: Bresenham's algorithm
 - Circle scan-conversion: The midpoint circle method
 - Filled-polygon scan-conversion: The scan-line method

Rasterizing Lines

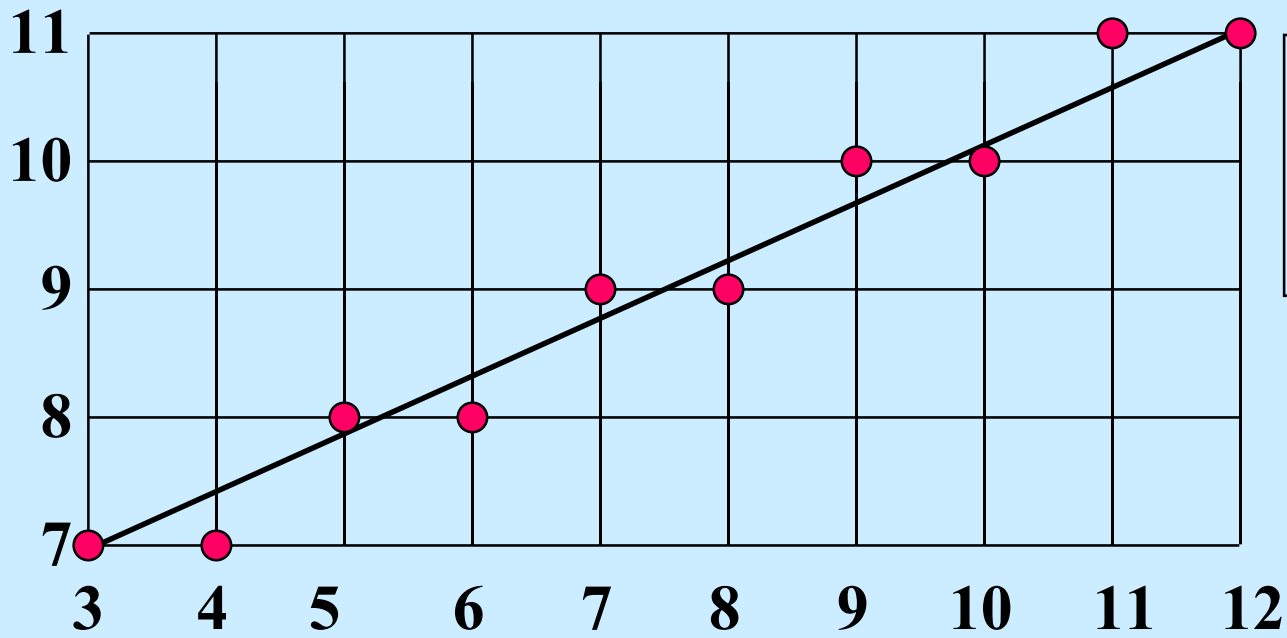
- **Objective:** Given endpoints p_0 and p_1 , rasterize the line from p_0 to p_1 .
- Here are the ideal properties of a line rasterization method:
 - the method is fast
 - produces lines of constant brightness
 - produces lines of constant thickness
 - produces straight lines
 - produces anti-aliased lines
 - produces the same line whenever it draws between the same two endpoints

Rasterizing Lines: A Simple Algorithm

- Draw the line from $p_0 = (x_0, y_0)$ to $p_1 = (x_1, y_1)$
- Assumption: Where m is the slope of line, $-1 \leq m \leq 1$. (We can swap the roles of x and y to ensure that this holds.)

```
Line(x0, y0, x1, y1, color) {  
    dy = y1 - y0;  
    dx = x1 - x0;  
    m = dy / dx;  
    y = y0;  
    for x = x0 to x1 {  
        PaintPixel(x, Round(y), color);  
        y = y + m;  
    }  
}
```

- The next slide shows an example of this method.



Simple
line drawing
from (3,7)
to (12,11)

$m=.4444$

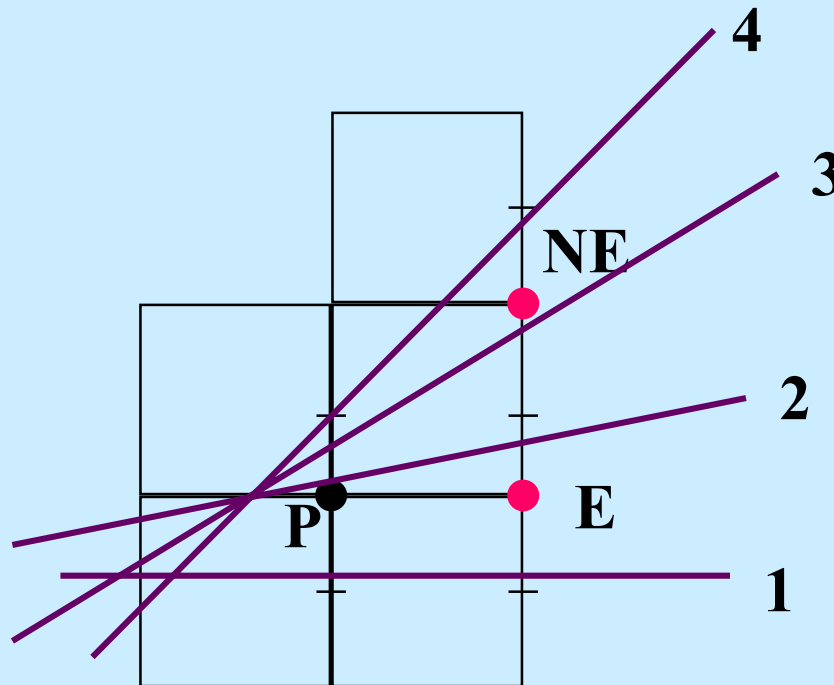
<u>x</u>	<u>y</u>	<u>point</u>
3	7.0	(3,7)
4	7.4444	(4,7)
5	7.8888	(5,8)
6	8.3333	(6,8)
7	8.7777	(7,9)
8	9.2222	(8,9)
9	9.6666	(10,10)
10	10.1111	(9,10)
11	10.5555	(11,11)
12	11.0	(12,11)

Rasterizing Lines: Problems with the Simple Line Algorithm

- This simple line-drawing method has some problems:
 - It is slow. It requires floating point calculations.
 - It is inaccurate. Errors accumulate in y due to floating point calculations.
- Bresenham's algorithm overcomes these problems. This algorithm:
 - Devised by J. Bresenham in 1965.
 - Solves these problems by using only integer arithmetic.
 - Same idea extended by Bresenham to the scan conversion of circles in 1977.

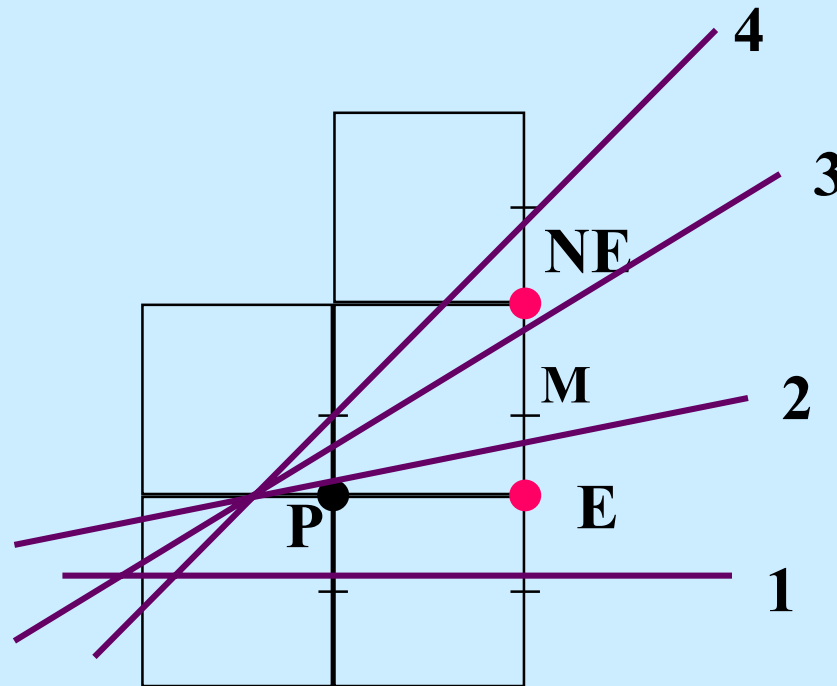
Rasterizing Lines: Bresenham's Algorithm

- Assumption: Where m is the line's slope, $0 \leq m \leq 1$.
- Given that the current pixel P is lit, we must decide whether pixel E (to its east) or pixel NE (to its northeast) gets lit next.



- Here pixel E is lit in the cases of lines 1 and 2, and pixel NE is lit in the cases of lines 3 and 4.

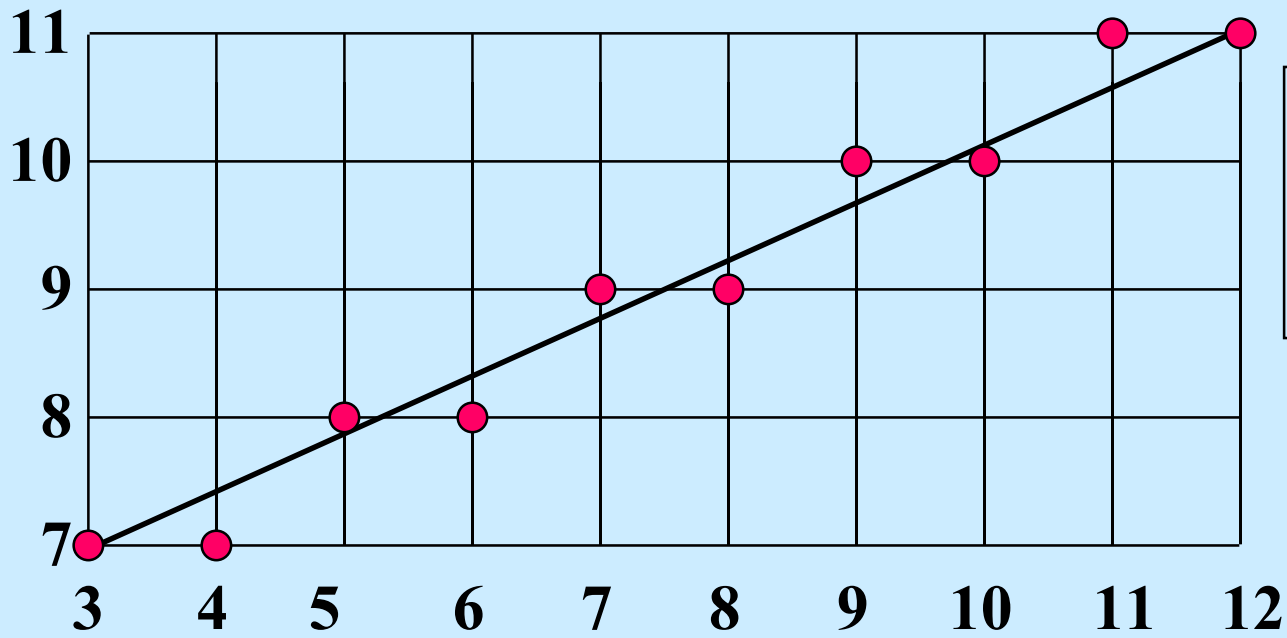
Rasterizing Lines: Bresenham's Algorithm



- Lines 1 and 2 cause pixel E to be lit because they pass below the midpoint M; lines 3 and 4 cause the pixel NE to be lit because they pass above the midpoint M.
- This algorithm uses a *decision variable* d , whose sign is used to classify the midpoint M with respect to the line being drawn.

Rasterizing Lines: Bresenham's Algorithm

```
Bresenham(x0, y0, x1, y1, color) {
    dx = x1 - x0;
    dy = y1 - y0;
    d = 2 * dy - dx;
    incE = 2 * dy;
    incNE = 2 * (dy - dx);
    x = x0;    y = y0;
    PaintPixel(x, y, val);
    while (x < x1) {
        if (d <= 0) {
            d = d + incE;
            x = x + 1;
        } else {
            d = d + incNE;
            x = x + 1;
            y = y + 1;
        }
        PaintPixel(x, y, val);
    }
}
```



**Bresenham's
method from
(3,7) to
(12,11)**

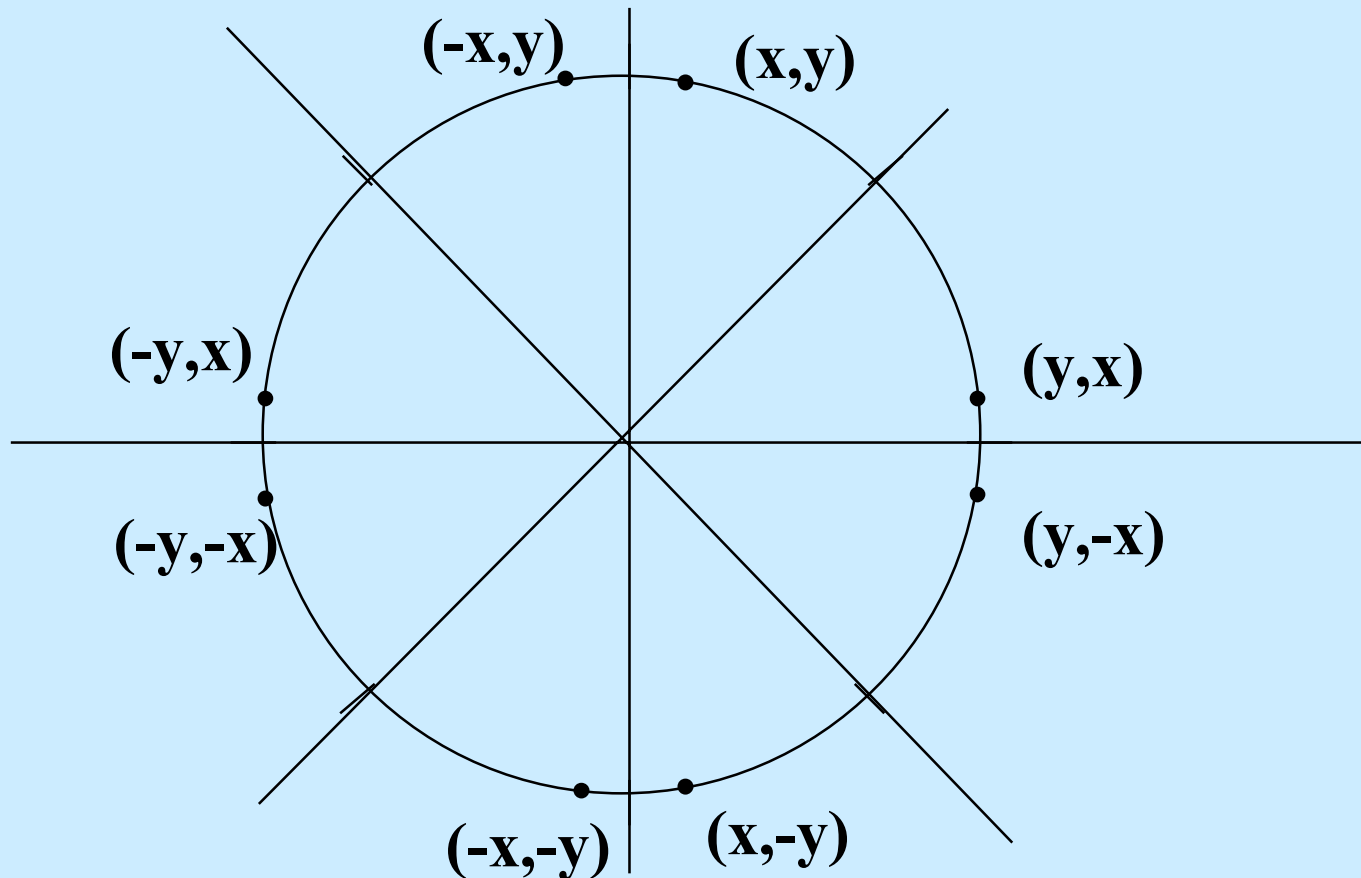
incE = 8

incNE = -10

<u>x</u>	<u>y</u>	<u>d</u>	<u>point</u>
3	7	-1	(3,7)
4	7	7	(4,7)
5	8	-3	(5,8)
6	8	5	(6,8)
7	9	-5	(7,9)
8	9	3	(8,9)
9	10	-7	(9,10)
10	10	1	(10,10)
11	11	-9	(11,11)
12	11	-1	(12,11)

Scan-Converting Circles

- Let us scan convert a circle with center $(0,0)$ and radius R .
- It is sufficient to scan convert a circular arc of 45 degrees, and reflect each point across the lines $x=0$, $y=0$, $x=y$, and $x=-y$.



Scan-Converting Circles: A Simple Algorithm

- We wish to scan convert the 45 degree arc extending from $(0,R)$ to (x,y) where $x = y = \sqrt{R^2/2}$.
- We use the fact that the y -coordinate of points along this arc is given by $y = \sqrt{R^2 - x^2}$.
- In the following, `CirclePoints` reflects a single point to eight points, as described in the previous slide.

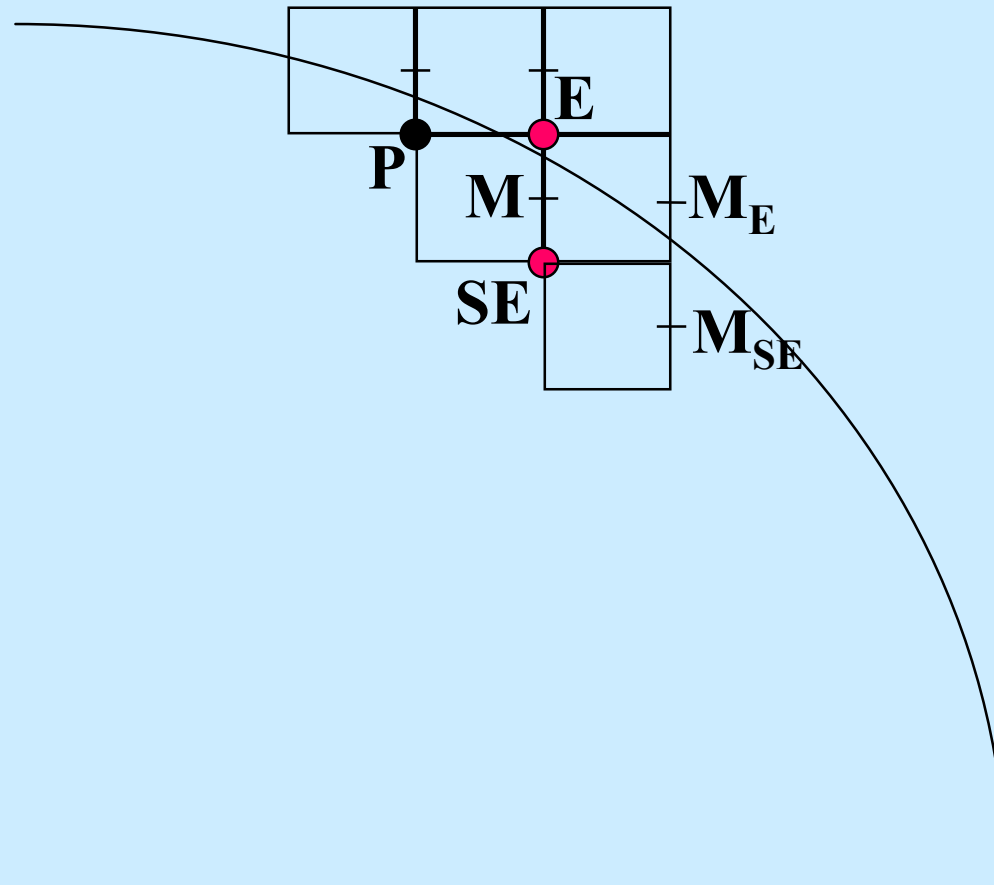
```
circle( R, color) {  
    y = R;  
    x = 0;  
    while (y > x) {  
        CirclePoints(x, Round(y), color);  
        y = sqrt(R2 - x2);  
        x = x + 1;  
    }  
}
```

Scan-Converting Circles: The Midpoint Circle Method

- **Problems with the simple algorithm:**
 - It is slow: floating point and sqrt operations are slow
 - It is inaccurate: floating point errors accumulate
- **Another algorithm: Midpoint Circle Algorithm**
 - Similar in spirit to Bresenham's line algorithm.
 - Integer arithmetic yields speed and accuracy

Scan-Converting Circles: The Midpoint Circle Method

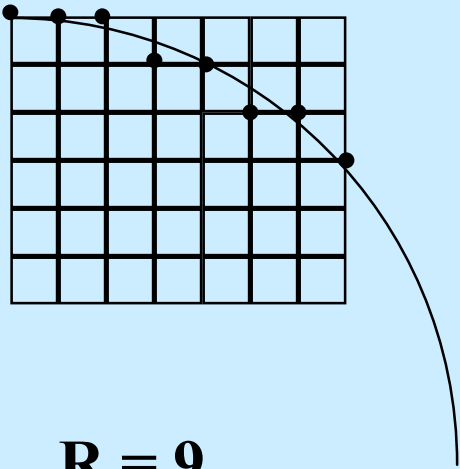
Given current pixel p , a decision variable d is used to decide which of the pixels E and SE is to be painted next.



Scan-Converting Circles: The Midpoint Circle Method

```
MidpointCircle(R, color) {
    x = 0;    y = R;
    d = 1 - R; deltaE = 3;    deltaSE = -2 * R + 5;
    CirclePoints(x, y, color);
    while (y > x) {
        if (d < 0) { /* select pixel E */
            d = d + deltaE;
            deltaE = deltaE + 2;
            deltaSE = deltaSE + 2;
            x = x + 1;
        } else { /* select pixel SE */
            d = d + deltaSE;
            deltaE = deltaE + 2;
            deltaSE = deltaSE + 4;
            x = x + 1;
            y = y - 1;
        }
        CirclePoints(x, y, color);
    }
}
```

The Midpoint Circle Method

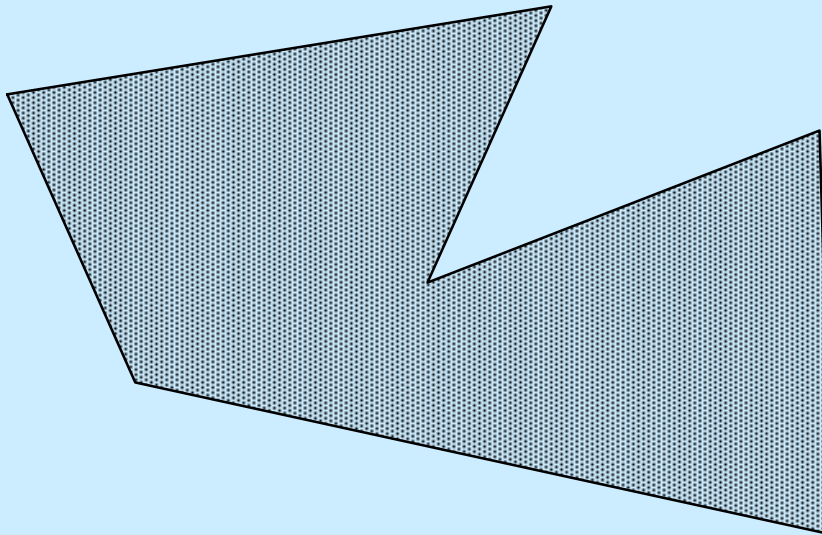


R = 9

x	y	d	deltaE	deltaSE
0	9	-8	3	-13
1	9	-5	5	-11
2	9	0	7	-9
3	8	-9	9	-5
4	8	0	11	-3
5	7	-3	13	1
6	7	10	15	3
7	6	13	17	7

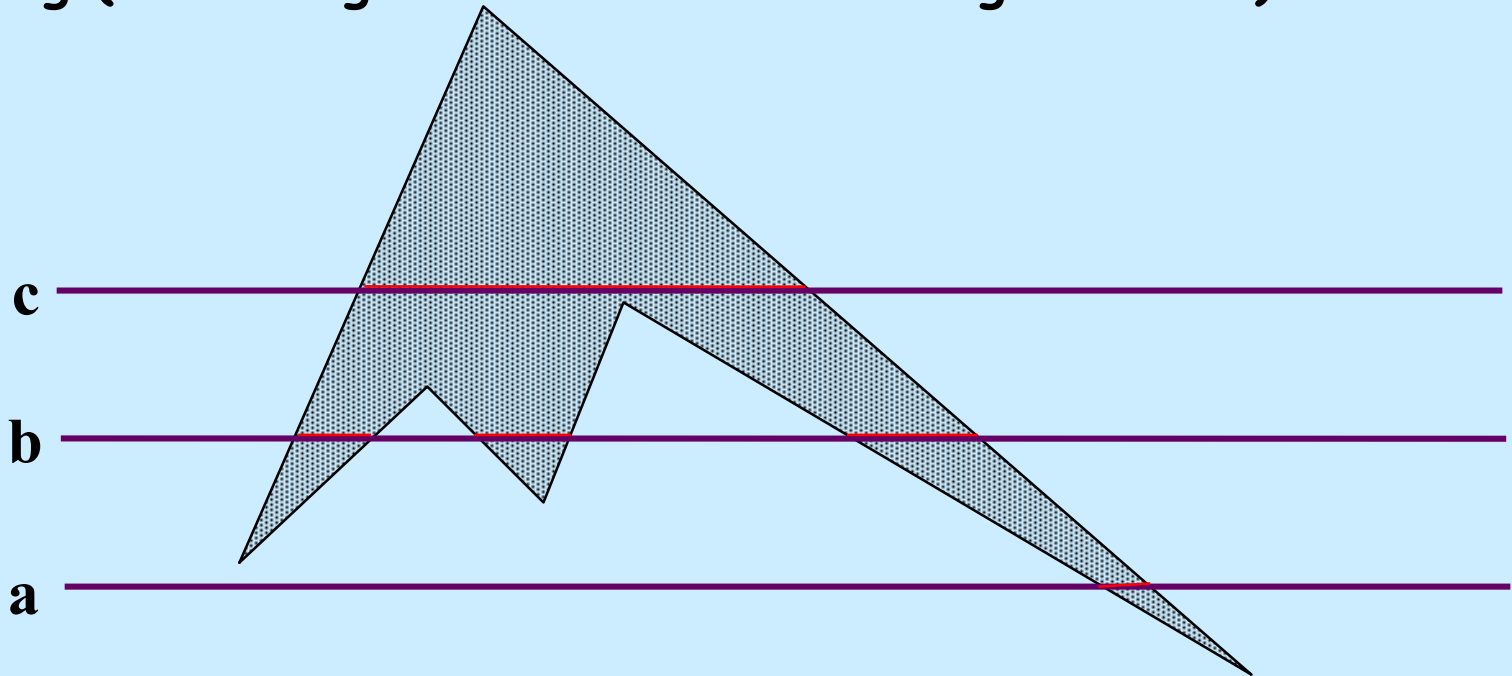
Scan Converting Filled Polygons

- A *polygon* is a non-self-intersecting cycle of line segments called *edges*. The endpoints of the edges are *vertices*.
- A *filled polygon* consists of a polygon together with the bounded region it encloses.
- We can represent a polygon (or filled polygon) by a circular list of its vertices.



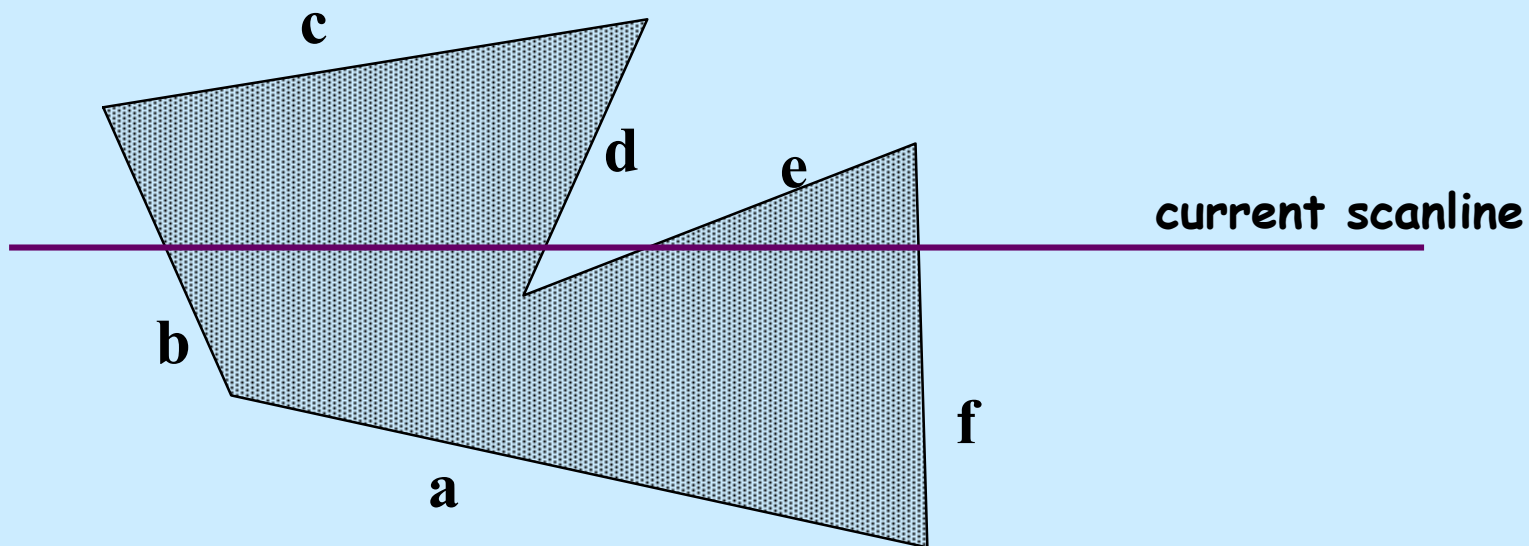
Scan Converting Filled Polygons

- We scan convert a filled polygon using a horizontal *sweep*line which sweeps across the polygon in increments, stopping at each *scanline* (row of pixels).
- As we traverse any given scanline left-to-right, the parity of number of crossings is odd whenever we enter the polygon and even whenever we exit the polygon.
- Draw each segment connecting an odd crossing to the next even crossing (drawn segments are red in the figure below).



Scan Converting Filled Polygons: Data Structures

- Active Edge Table (AET): A list of *active edges* (i.e., those that the sweepline crosses), ordered by increasing x coordinate.
- Edge Table (ET): A list of all edges, ordered by increasing y coordinate of their lower endpoints.



AET = (b, d, e, f) for the current scanline

ET = (a, f, b, d, e, c)

Scan Converting Filled Polygons

```
FilledPolygon(Polygon P) {
    create edge table ET for P;
    y = minimum y in ET;
    create empty active edge table AET;
    repeat {
        insert each edge with ymin = y into AET;
        fill pixels in current scanline y, using AET;
        remove each edge with ymax = y from AET;
        y = y + 1;
        for each edge of AET, update x for new y;
    } while (AET ≠ ∅);
}
```