Introduction to Computer Graphics

6. Rasterization

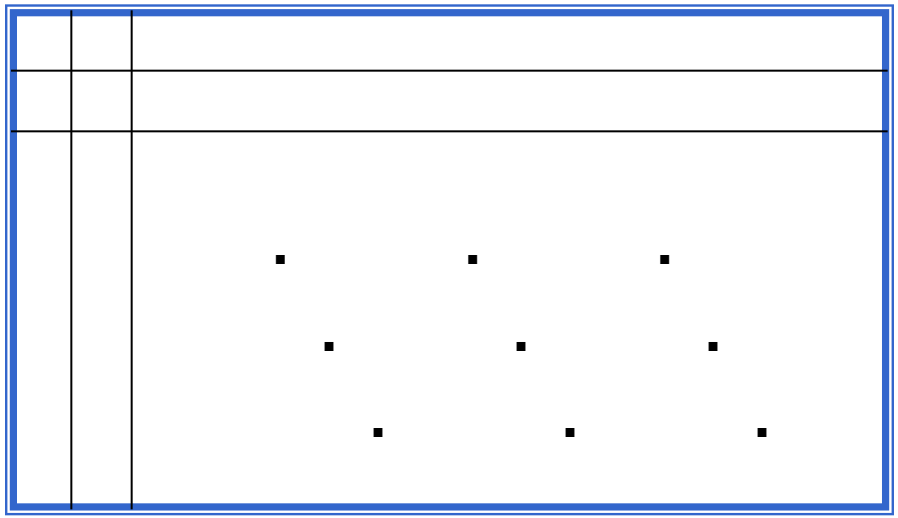
National Chiao Tung Univ, Taiwan
By: I-Chen Lin, Assistant Professor

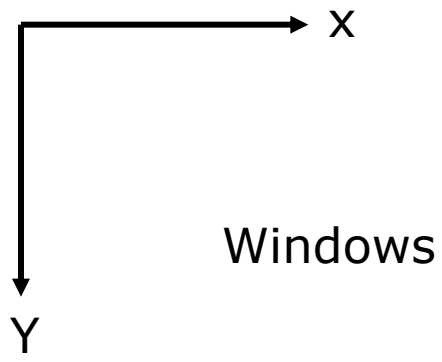
Textbook: E.Angel, Interactive Computer Graphics, 5th Ed., Addison Wesley
Ref:Hearn and Baker, Computer Graphics, 3rd Ed., Prentice Hall

Outline

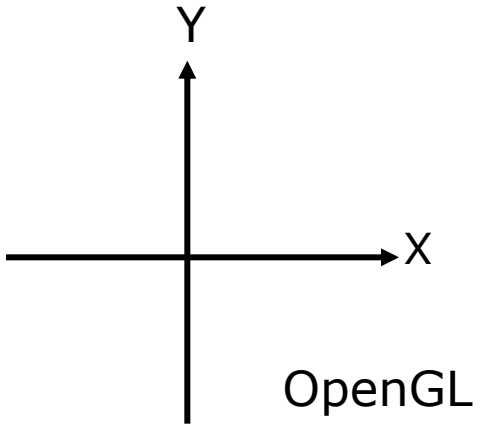
- 2D graphics primitives
  - Line drawing
  - Circle drawing
  - ……
- Area filling
  - Polygons
  - ……

Discrete Video Screen





- Assigning pixel values by
  - Functions:
    - e.g. SetPixel(x, y, color)
  - Buffer or arrary:
    - e.g. FrameBuf[x][y] = color



How to Draw Primitives ?

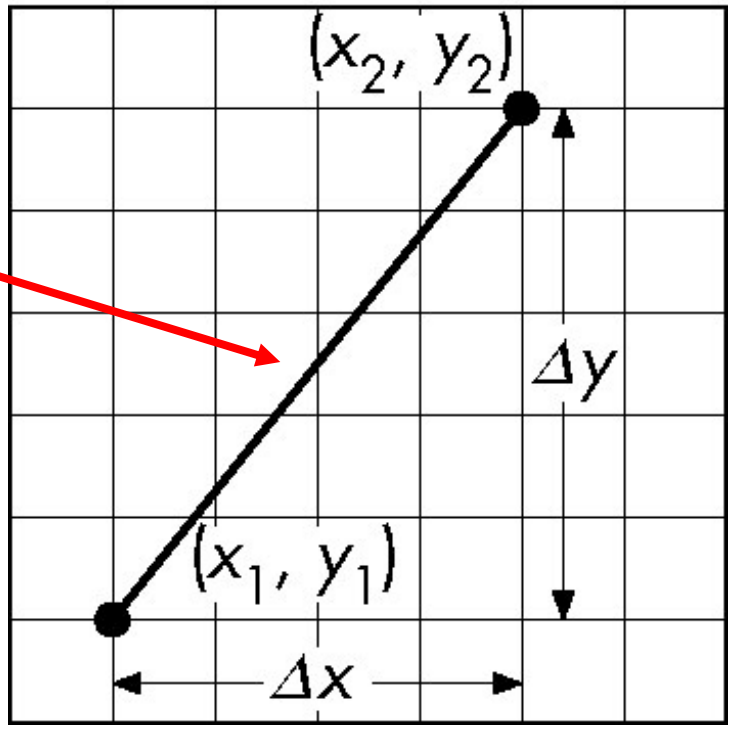
- From math representation to screen.
- In addition to “brute-force”, how to improve the efficiency of computation or memory usage.
- Primitives
  - Lines
  - Circles
  - Curves
  - ……

Line-Drawing Algorithms

- Start with line segment in window coordinates with integer values for endpoints

$$y = mx + h$$

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



DDA Algorithm

- Digital Differential Analyzer
  - Line *y=mx+ h* satisfies differential equation.

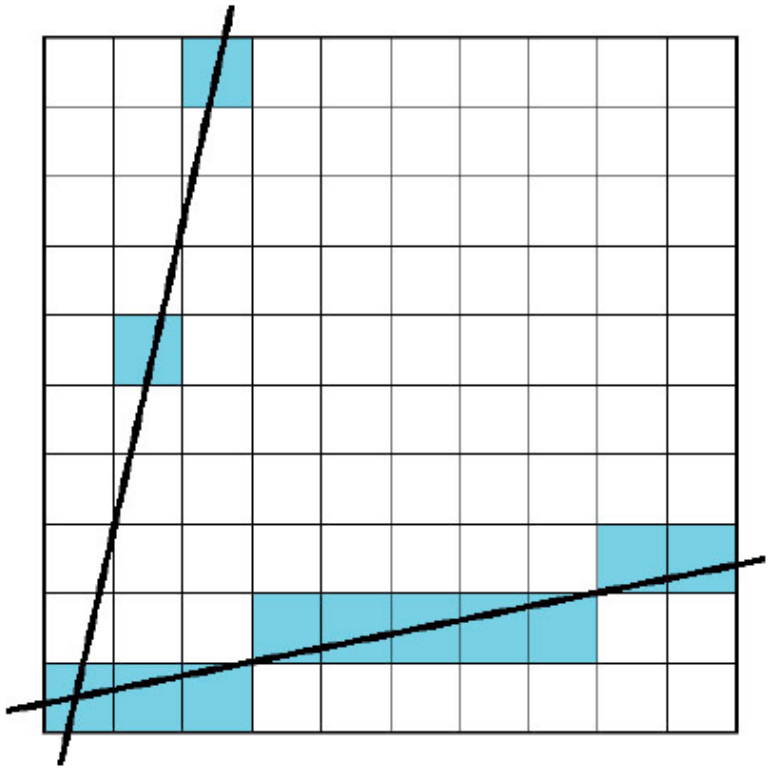
$$\frac{dy}{dx} = m = \frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1}$$

- Along scan line Δx = 1

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

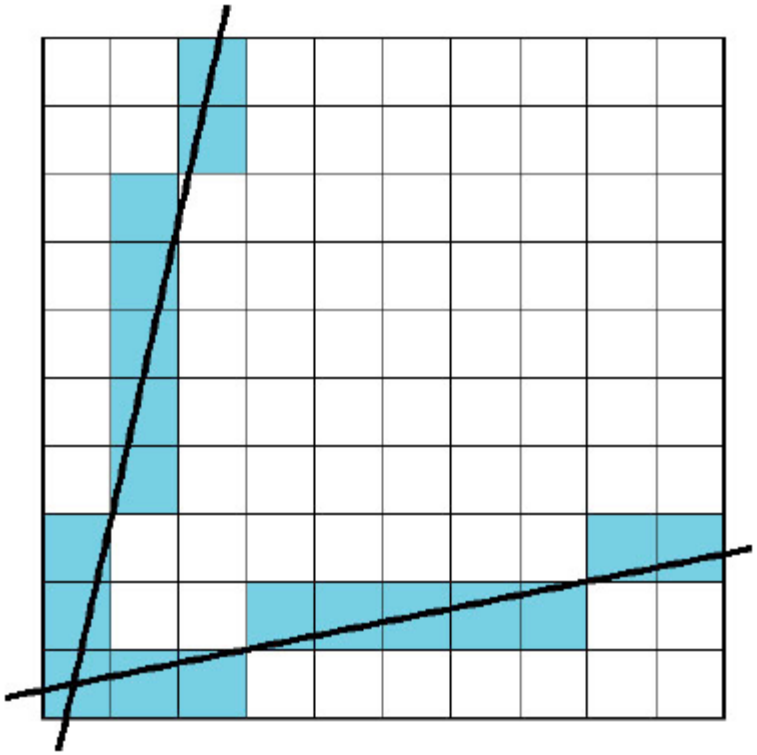
Problem

- 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 roles of x and y
  - For each y, plot closest x



Bresenham's Algorithm

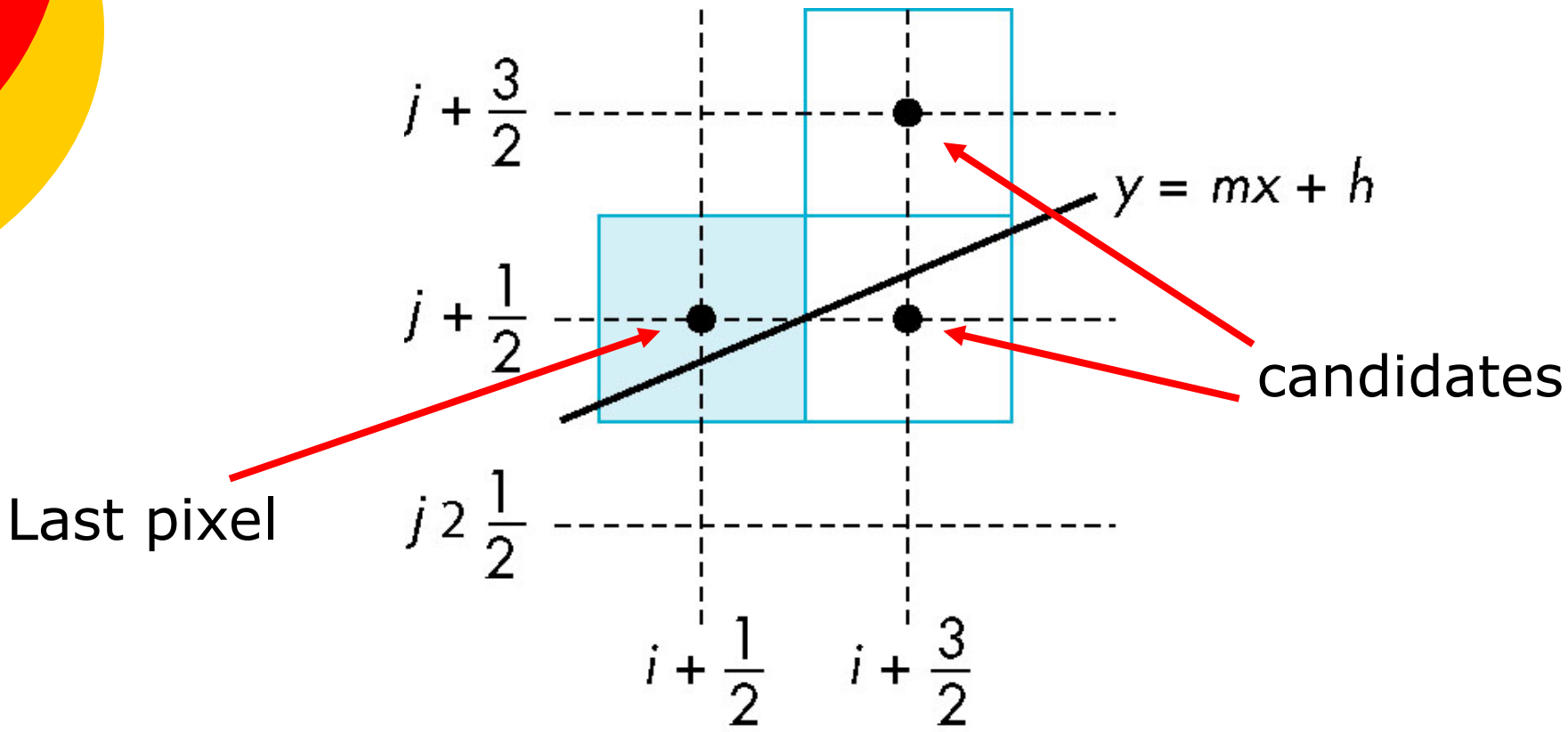
- DDA requires one floating point addition per step.
- Bresenham's algorithm eliminates all fp.

- Consider only $1 \geq m \geq 0$
  - Other cases by symmetry
- Assume pixel centers are at half integers.

- Characteristics:
  - If we start at a pixel that has been written, there are only two candidates for the next pixel

Candidate Pixels

- $1 \geq m \geq 0$

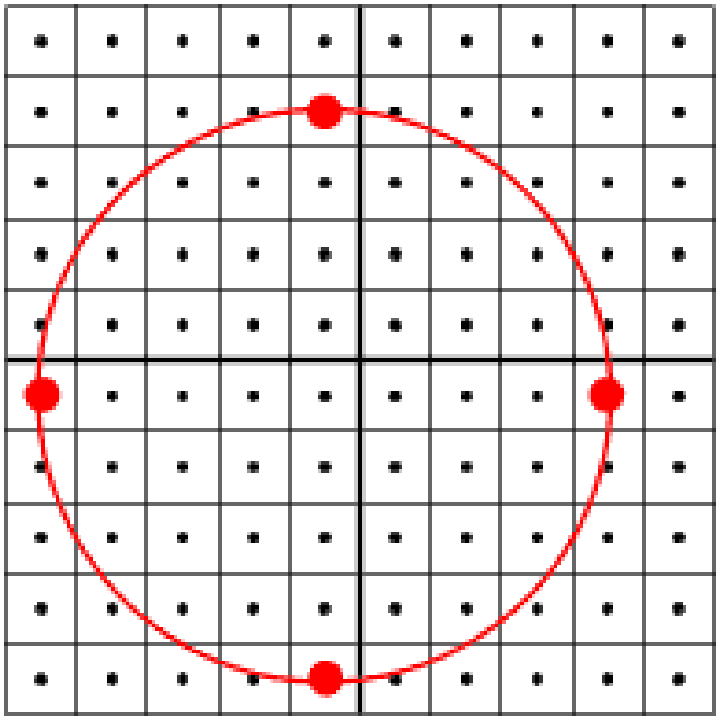


Bresenham's Algorithm

```
function line(x0, x1, y0, y1)
    int deltax := abs(x1 - x0)
    int deltay := abs(y1 - y0)
    real error := 0
    real deltaerr := deltay ÷ deltax
    int y := y0
    for x from x0 to x1
    plot(x,y)
    error := error + deltaerr
    if error ≥ 0.5
          y := y + 1
          error := error - 1.0
```

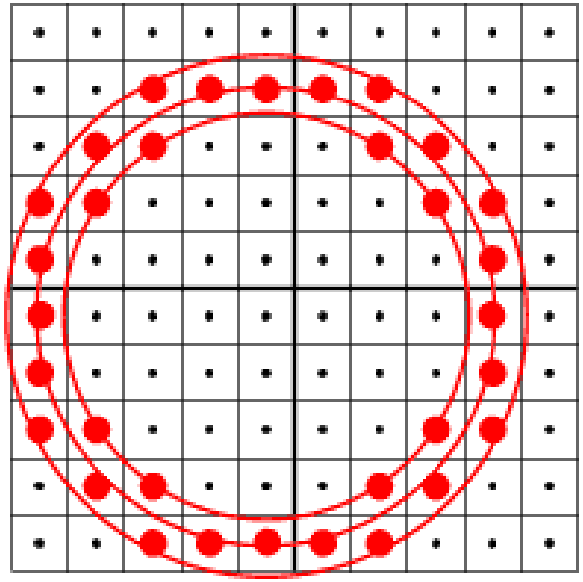
```
function line(x0, x1,y0, y1)
int deltax := abs(x1 - x0)
int deltay := abs(y1 - y0)
int error := 0
int deltaerr := deltay
int y := y0
for x from x0 to x1
    plot(x,y)
    error := error + deltaerr
    if 2×error ≥ deltax
          y := y + 1
          error := error - deltax
```

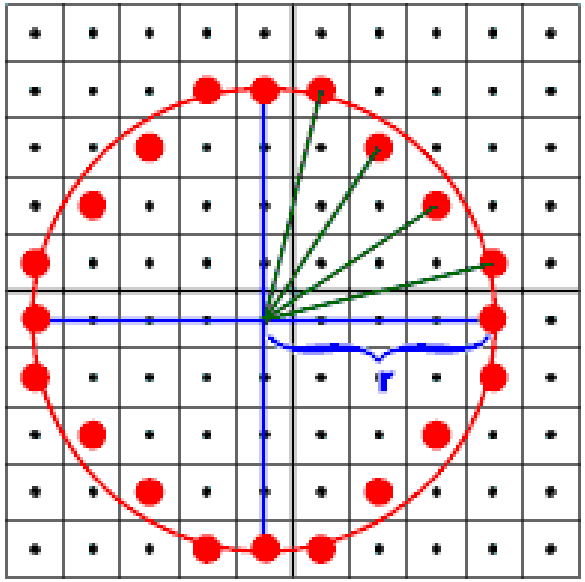
Circle-drawing Algorithms



Ref: http://www.cs.umbc.edu/~rheingan/435/index.html

Circle-drawing Algorithms



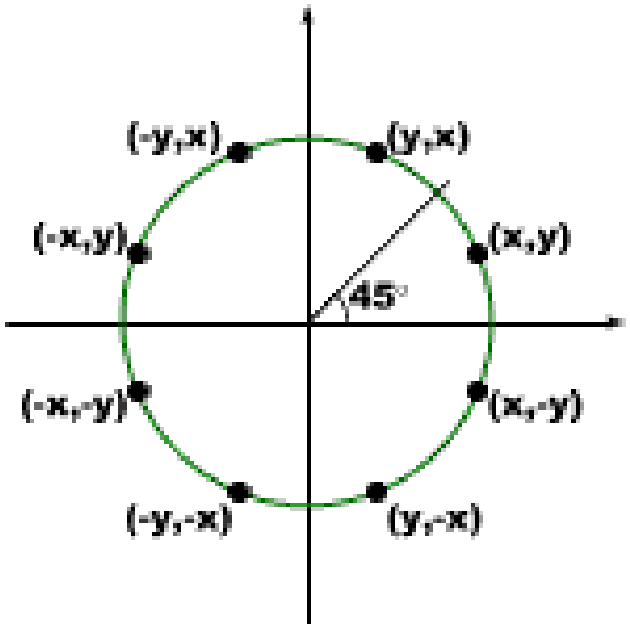



```
for each x, y
    if | x² + y² - r² | <= ε
    SetPixel ( x, y )
```

```
for θ in [0~360 degree ]
    x = r cos(θ)
    y = r sin(θ)
    SetPixel ( x, y )
```

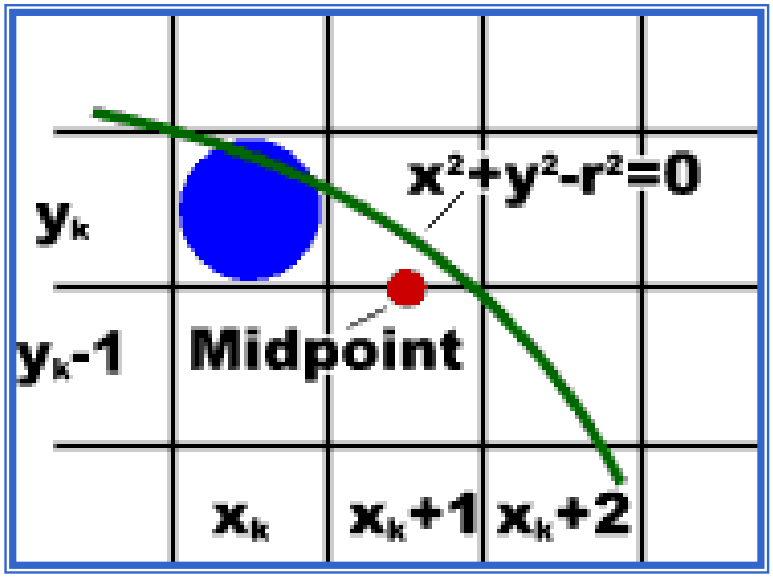
Midpoint Circle Algorithm

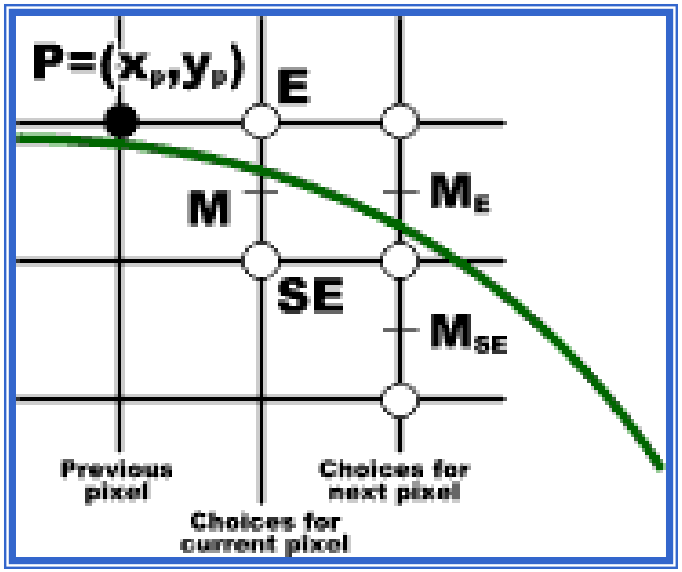
- Can we utilize the similar idea in Bresenham's line-drawing algorithm ?
  - Check only the next candidates.
  - Use symmetry and simple decision rules.



Symmetry of a Circle

Midpoint Circle Algorithm (cont.)

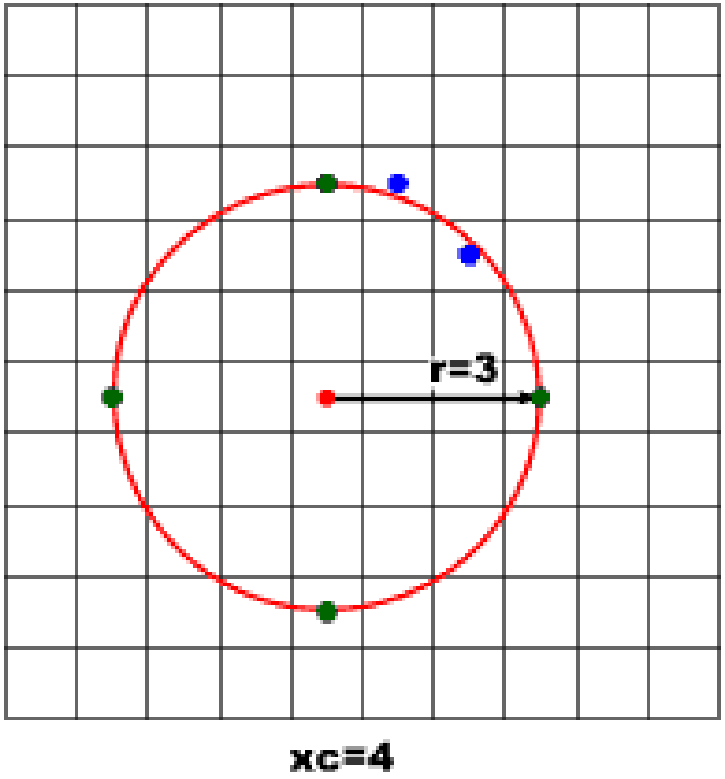


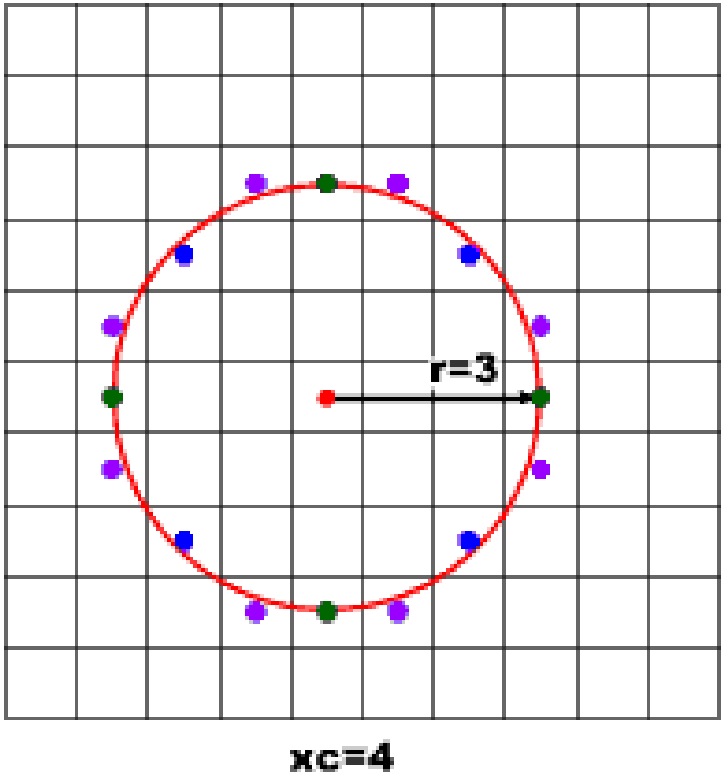



```
f(x,y) = x² + y² - R²
f(x,y) > 0 => point outside circle
f(x,y) < 0 => point inside circle
```

$$P_k = f_{circ}(x_k + 1, y_k - \tfrac{1}{2})$$

Midpoint Circle Algorithm (cont.)





Midpoint Circle Algorithm (cont.)

- Given the starting point (0,r), the computation is more efficient.

$P_0 = 5/4 - r$

At each x position,

if(pk < 0)

the next point is (x_{k+1}, y_k)

$p_{k+1} = p_k + 2x_{k+1} + 1$

else

the next point is $(x_{k+1}, y_k - 1)$

$p_{k+1} = p_k + 2x_{k+1} + 1 - 2y_{k+1}$

Other Primitives

- Ellipse, polynomials, splines, etc.
- How to draw such primitives ?
- How about the computation cost ?

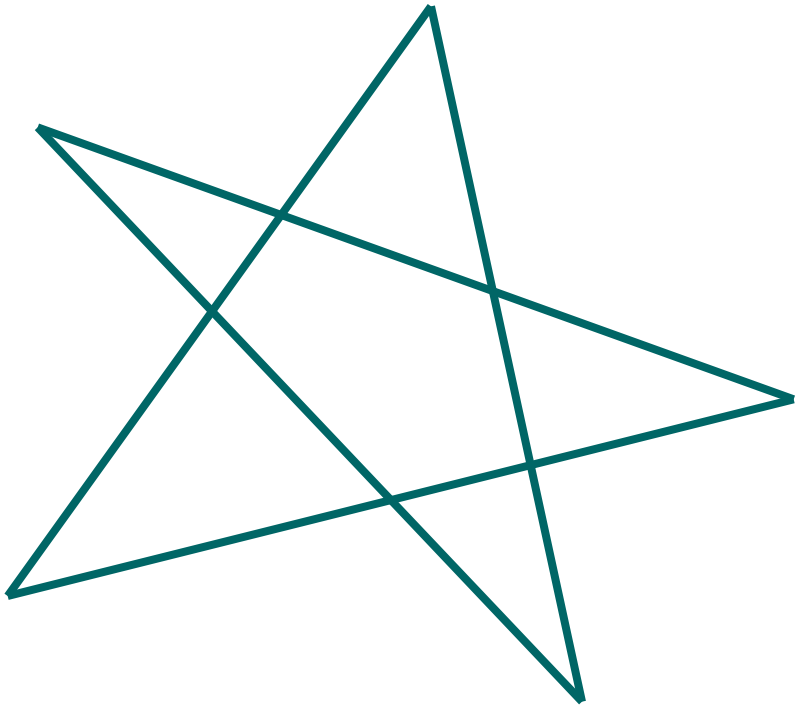
2D Polygon Filling

- Recall:
  - In computer graphics, we usually use polygons to approximate complex surfaces.

- Let’s focus on the polygon filling !

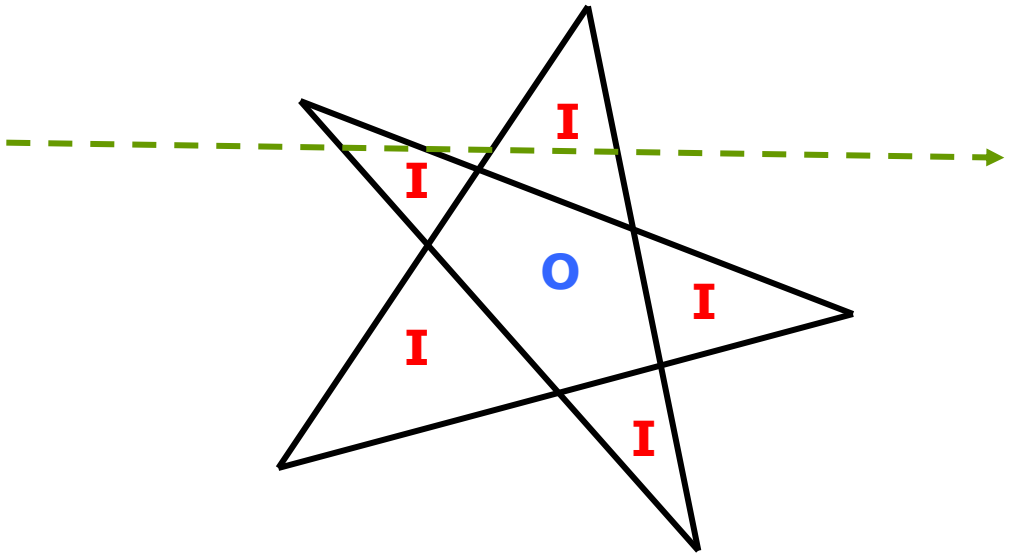
General Polygons

- Inside or Outside are not obvious
  - It's not obvious when the polygon intersects itself.



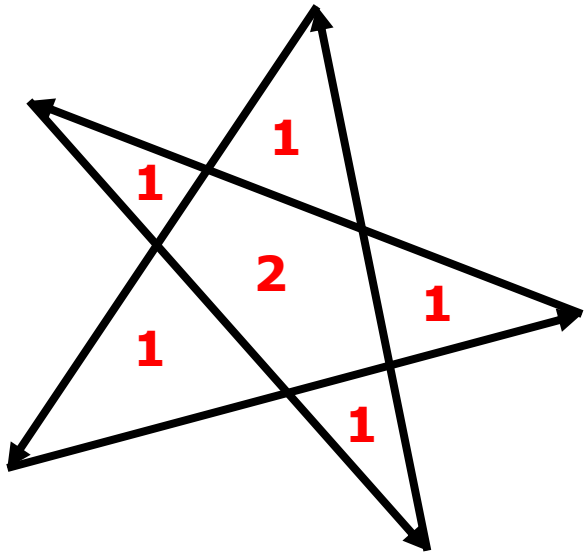
Inside or Outside

- Odd-even rule :
  - Draw a ray to infinity and count the number of edges that cross it.
  - Even → outside; odd → inside
  - usually used for polygon rasterization



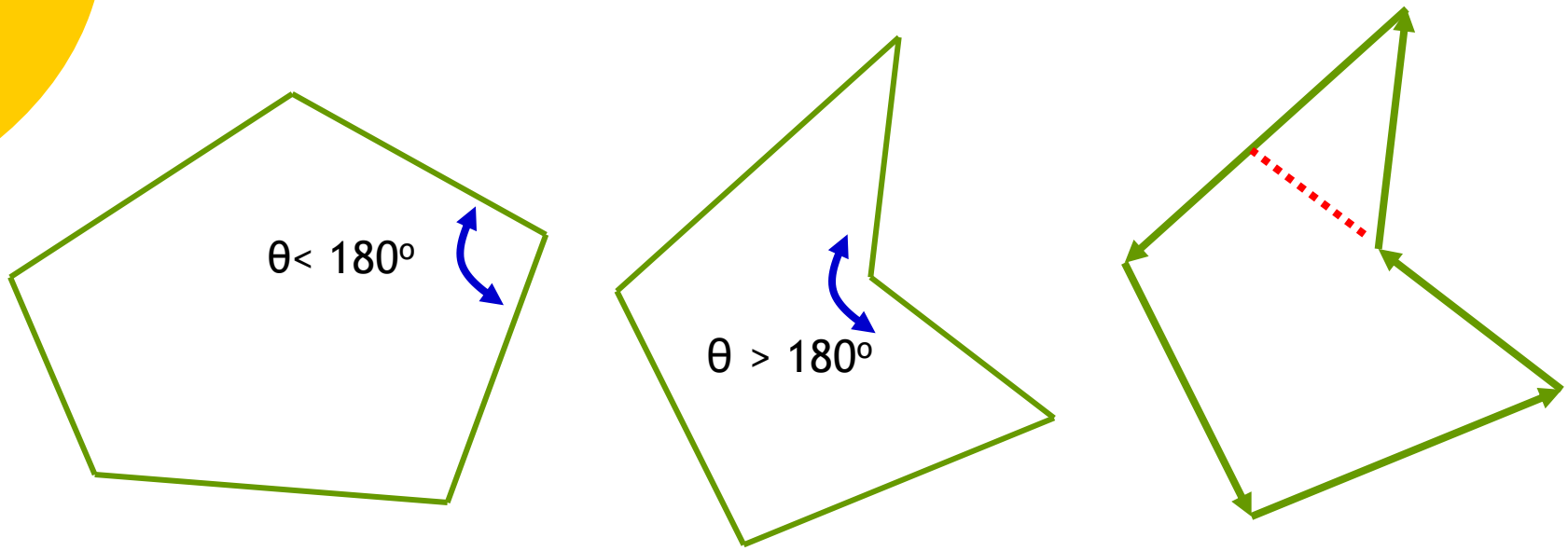
Inside or Outside

- Non-zero winding rule
  - trace around the polygon, count the number of times the point is circled (+1 for clockwise, -1 for counter clockwise).
  - Non-zero winding counts = inside



Concave vs. Convex

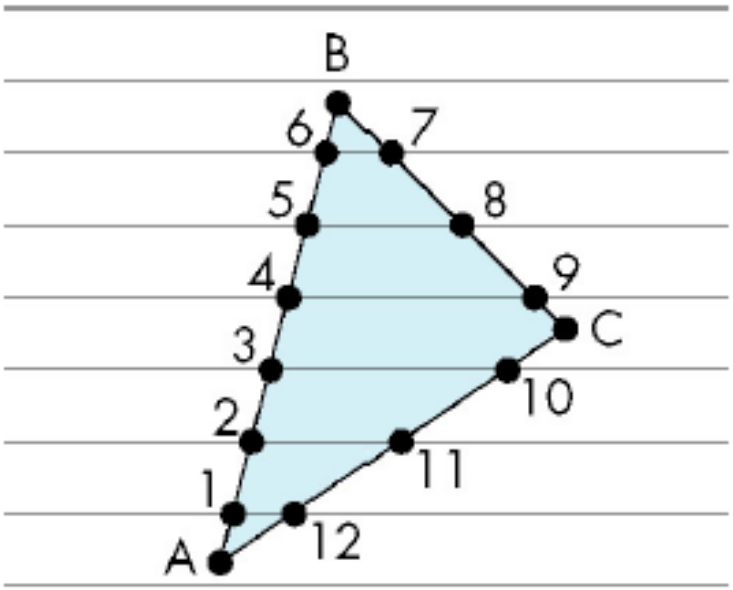
- We prefer dealing with “simpler” polygons.
- Convex (easy to break into triangles)

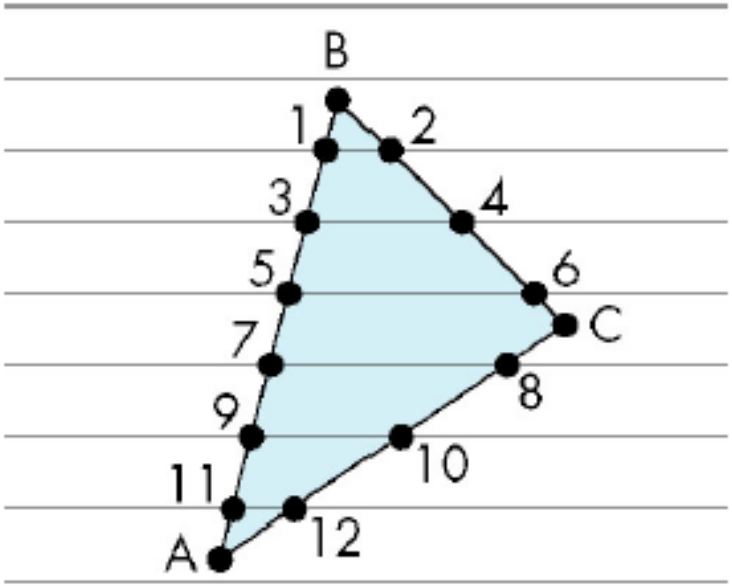


convex concave

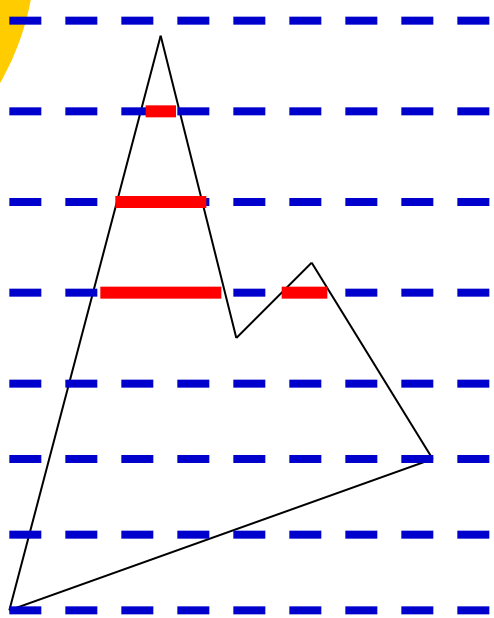
Polygon Filling by Scan Lines

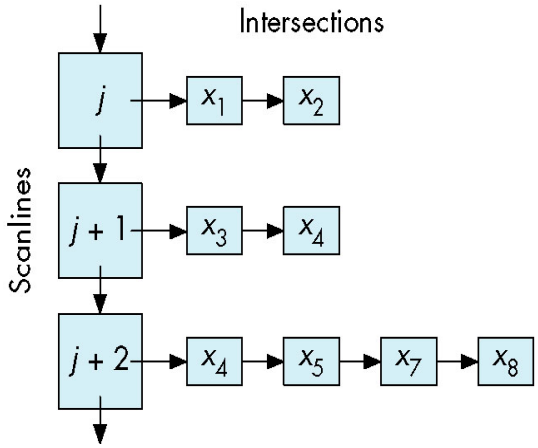
- Fill by maintaining a data structure of all intersections of polygons with scan lines
  - Sort the scan lines
  - Fill each span





Data Structure





Applying the odd-even rule