Introduction to Computer Graphics

7. Rasterization 光柵掃描法

要把primitive中間塞滿

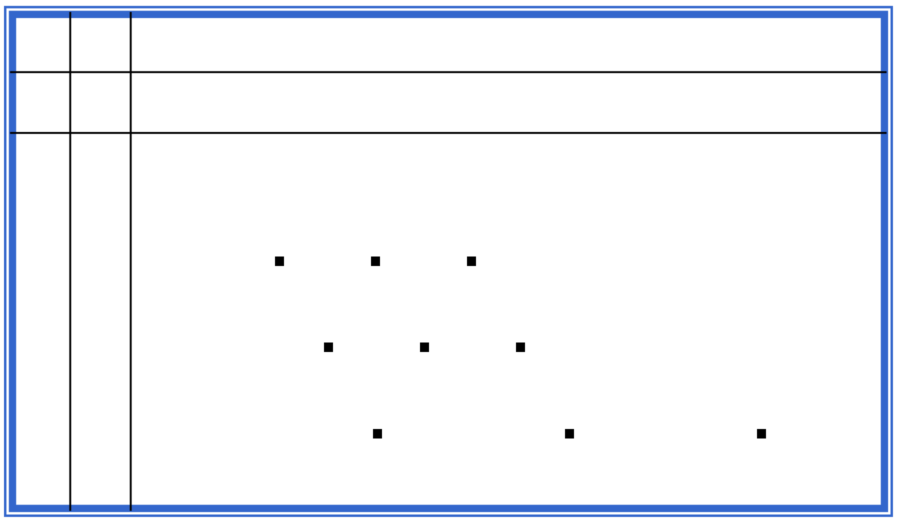
I-Chen Lin
National Chiao Tung University

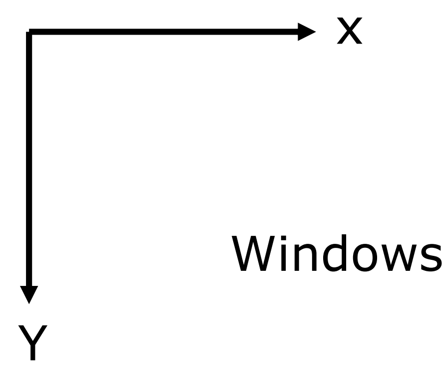
Textbook: E.Angel, D. Shreiner Interactive Computer Graphics, 6th Ed., Pearson
Ref: D.D. Hearn, M. P. Baker, W. Carithers, Computer Graphics with OpenGL, 4th Ed., Pearson

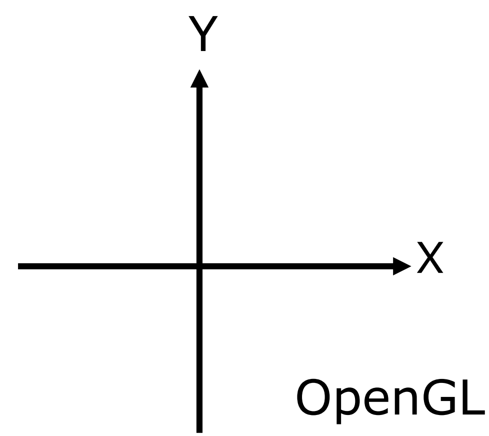
Outline

- Draw primitives in discrete screen space.
- 2D graphics primitives
  - Line drawing
  - Circle drawing
- Area filling
  - Polygons

Discrete Video Screen







- Assigning pixel values by
  - Functions: 離散地放pixel在螢幕上，每次都要syscall =>速度慢、且可能會interrupt
    - e.g. SetPixel(x, y, color)
  - Buffer or arrary: 這個方法比較好，array化較有效率
    - 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
  - ……

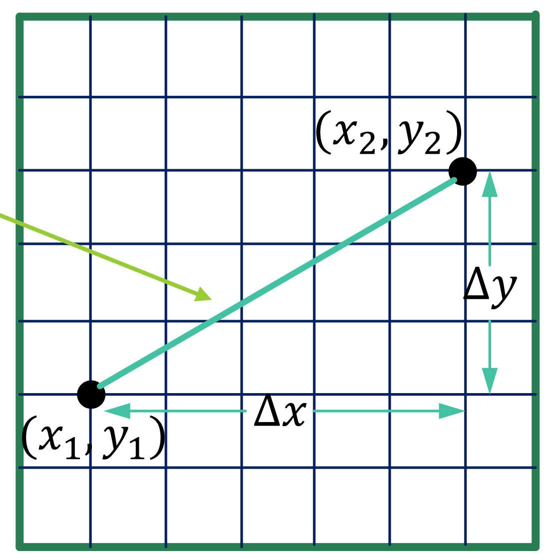
Line-Drawing Algorithms 離散化畫在畫面上

- Start with line segment in window coordinates with integer values for endpoints. *有人定義點要放在格線上、有人定義點要放格子間

$$\mathbf{y = mx + h}$$

填滿中間pixel畫出直線

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



DDA Algorithm

round(四捨五入) : plot(x, int(y+0.5))
=>儘量用整數計算

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

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

*橫線直線直接for loop塗滿中間

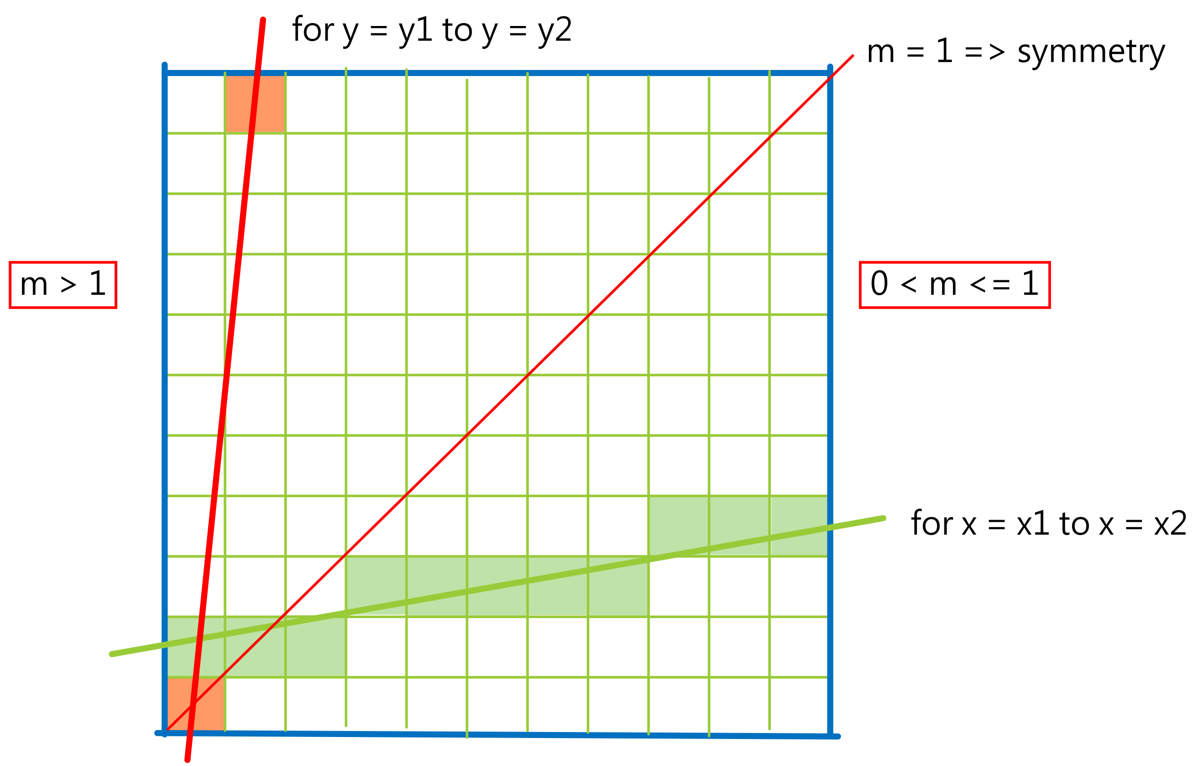
- Along scan line Δx = 1

$\boxed{y=mx+h}$

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

Problem

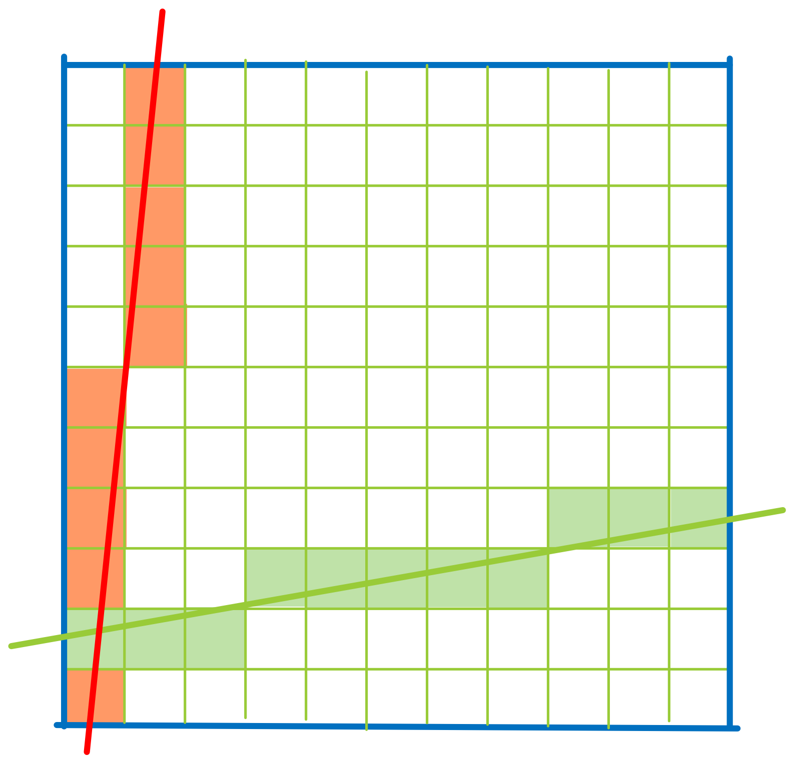
- DDA = for each *x* plot pixel at closest *y.*
  - Problems for steep lines



*直接看delta x 和delta y誰大，決定for loop要由誰決定

Using Symmetry

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



其他演算法：
主要著色的地方顏色較深，
旁邊的部分會用較淺的顏色

純整數運算法

Bresenham's Algorithm

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

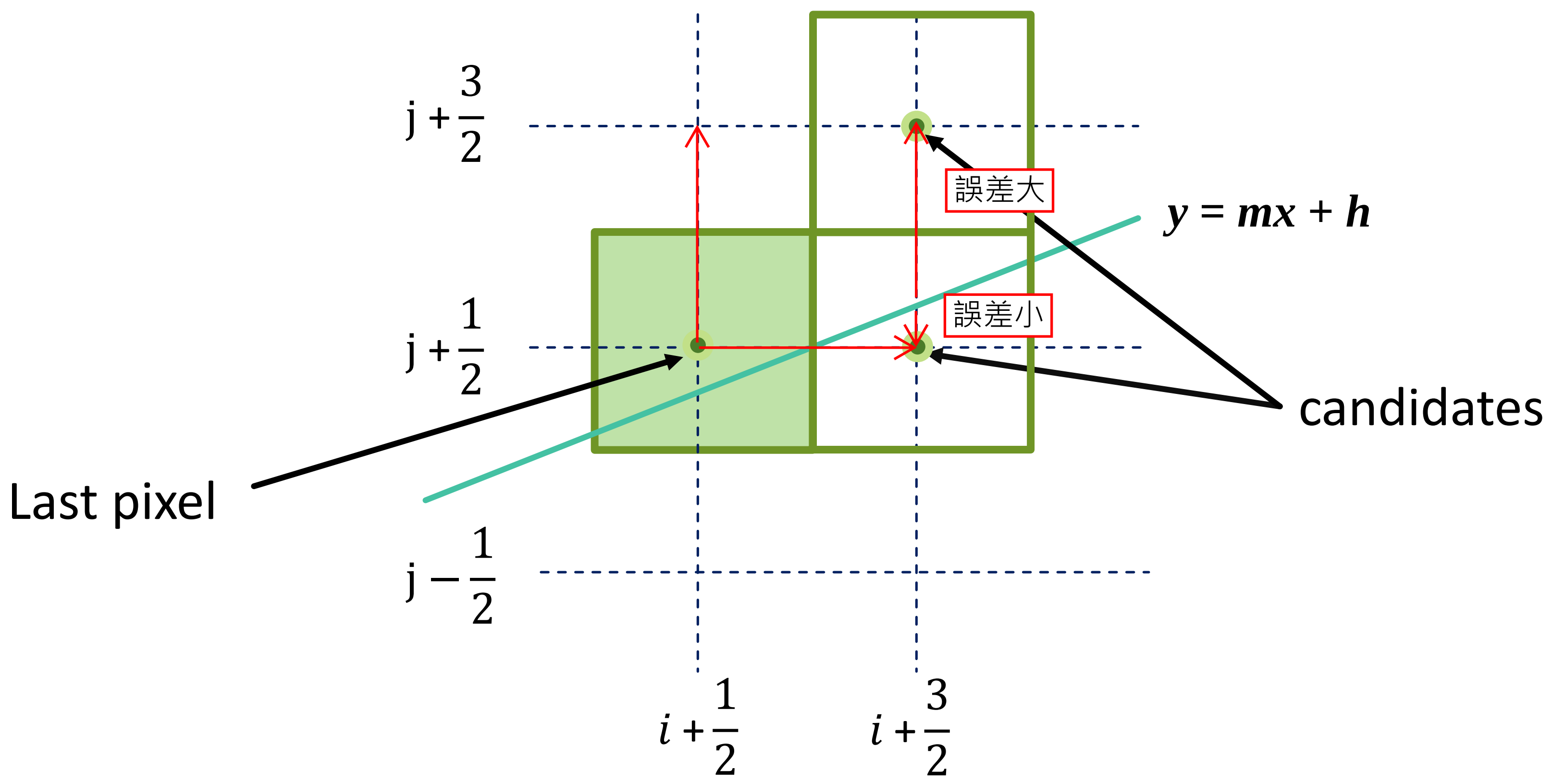
- Consider only $1 \geq m \geq 0$ EX：delta x = 1 => delta y <= 1
  - Handing other cases by symmetry ：只討論其中一段，另外一段翻過來就好
- Assume pixel centers are at half integers.

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

每次都在點的位置二選一

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
```

四捨五入後的誤差

每次往右動的時候會增加一個deltaerr (deltaerr = m)

誤差 >= 0.5時選上面那顆比較好

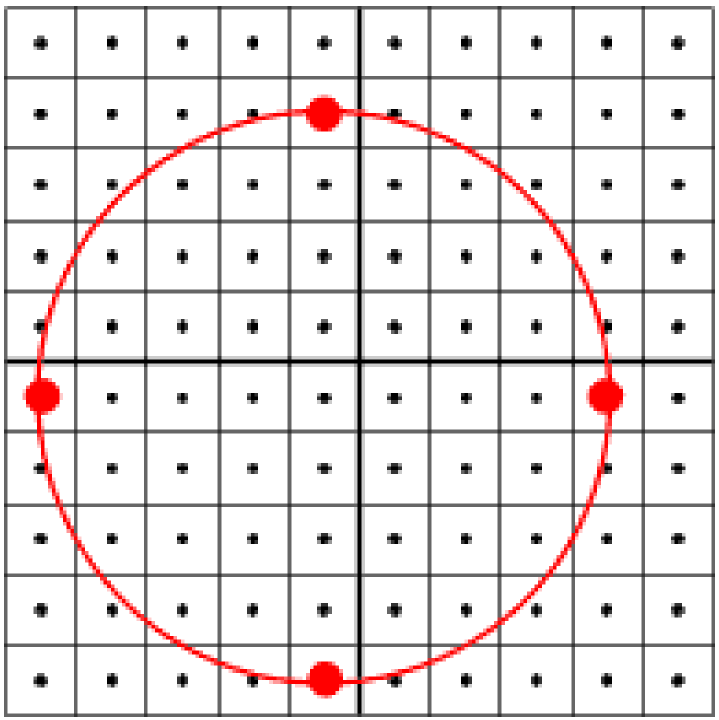
```
function line(x0, x1,y0, y1)
int deltax := abs(x1 - x0)
int deltay := abs(y1 - y0)
int error := 0
int deltaerr := deltay
int y := y0
for x from x0 to x1
    plot(x,y)
    error := error + deltaerr
    if 2×error ≥ deltax
        y := y + 1
        error := error - deltax
```

tilda err = delta x * delta err

減少浮點數出現

Circle-drawing Algorithms

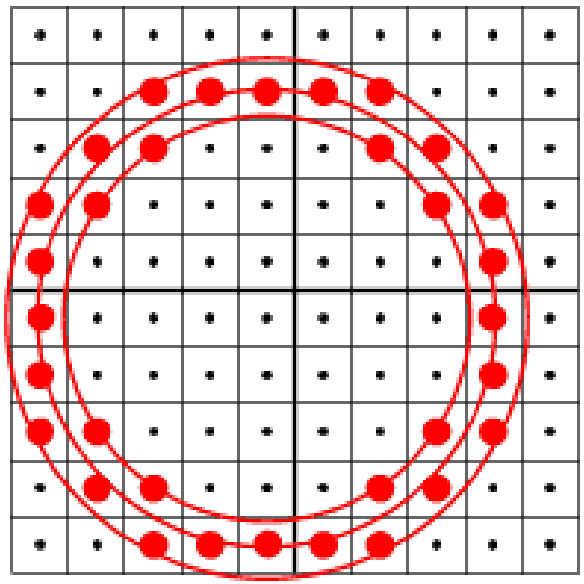
給圓心和半徑，要去填滿圖形；
可以用鋸齒或顏色漸進變化去讓它漂亮一點

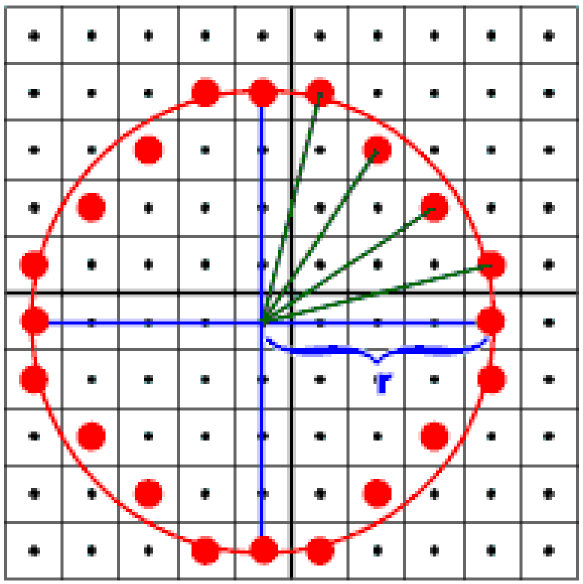


Ref: http://www.cs.umbc.edu/~rheingan/435/index.html

Circle-drawing Algorithms

因為都取整數點，所以可能圓中間會有洞(顏色塗不滿)





for each x, y
  if | $x^2 + y^2 - r^2$ | <= ε
  SetPixel (x, y)

sigma : 容錯誤差(tolerance)、很難決定取值
O(n^2)且中間很多無意義的檢測(可能取到重複的線)

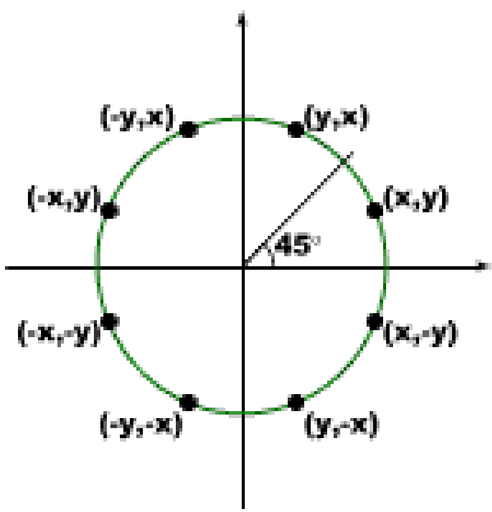
for θ in [0~360 degree]
  x = r cos(θ)
  y = r sin(θ)
  SetPixel (x, y)

theta難決定 : 可能在某個角度擠了
一大堆格子、或跳格子(中間沒畫到)

Midpoint Circle Algorithm

先把圓切成八等分=>symmetry
確保每次只畫一格、避免redundant

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



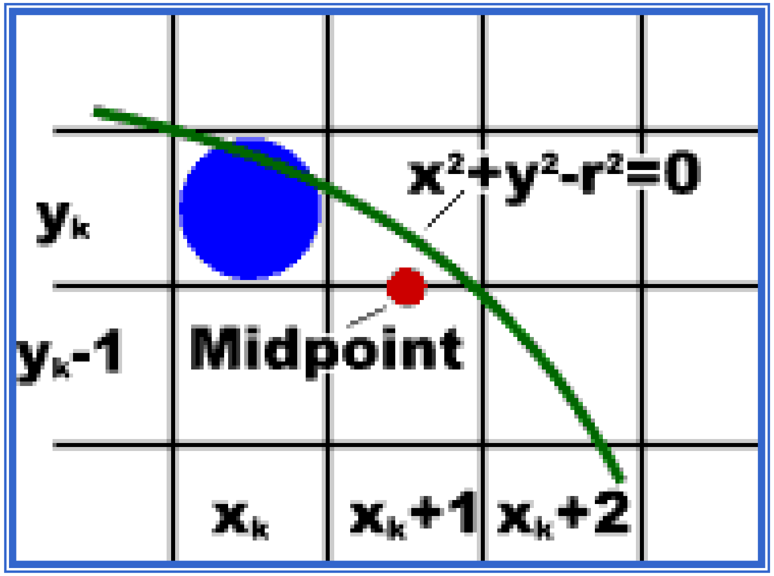
Symmetry of a Circle

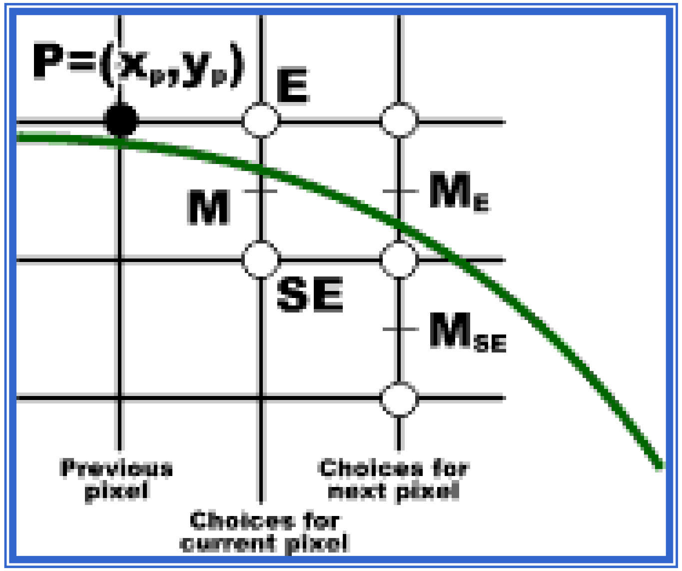
mapping table由不同的半徑決定
上下左右翻一翻

二選一的原因：避免中間斷掉(可能會跳躍取點)
use symmetic for loop

Midpoint Circle Algorithm (cont.)

往右走，做midpoint檢測
比曲線大就往下走；
比曲線小就往上走
=>真正的弧線在點上面





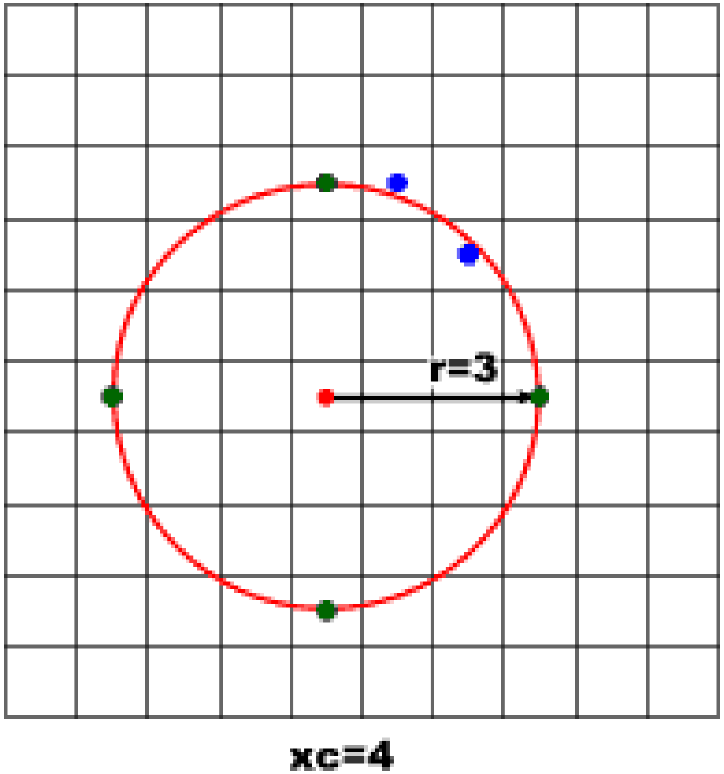
f(x,y) = $x^2 + y^2 - R^2$

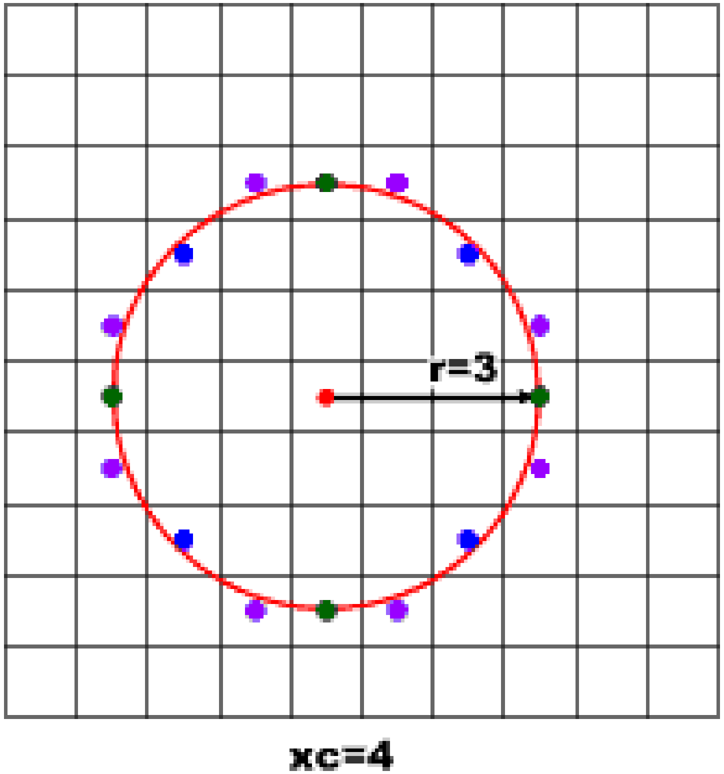
f(x,y) > 0 => point outside circle
f(x,y) < 0 => point inside circle

f(M) > 0 => choose SE
f(M) < 0 => choose E

$$P_k = f_{circ}\boxed{(x_k + 1, y_k - ½)}$$ =>midpoint

Midpoint Circle Algorithm (cont.)





(xk + 1, yk - 1/2)

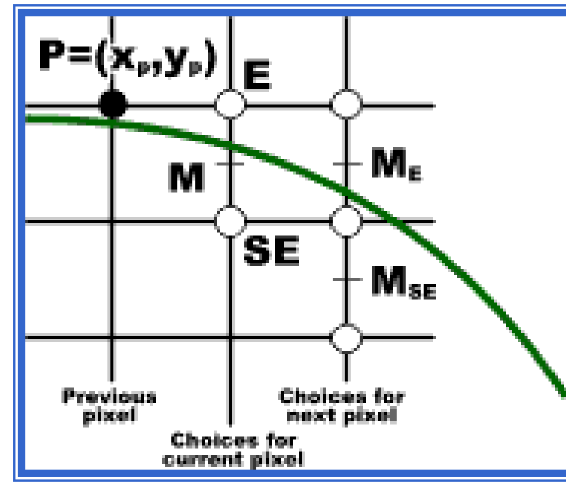
Midpoint Circle Algorithm

點會重複計算=>想要簡化計算、減少重複平方項

x^2 + y^2 = r^2

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

$$p_0 = f_{circle}(1, r-1/2)$$
$$= 1+ (r-1/2)^2-r^2$$
$$= \boxed{5/4 - r}$$



- For each x position,

$$p_k = f_{circle}(x_k +1, y_k-1/2) = (x_k +1)^2 + (y_k -1/2)^2 - r^2,$$

圈圈內 If $p_k < 0$, choose E, $(x_{k+1} = x_k+1,\ y_{k+1} = y_k)$

$$p_{k+1} = f_{circle}(x_{k+1}+1, y_{k+1}-1/2) = [(x_k+1)+1]^2 + (y_k -1/2)^2 - r^2$$
$$\boxed{= p_k + 2x_k + 3 = p_k + 2x_{k+1} + 1}$$

圈圈外 If $p_k > 0$, choose SE, $(x_{k+1} = x_k+1,\ y_{k+1} = y_k-1)$

記後面等號的

$$p_{k+1} = f_{circle}(x_{k+1}+1, y_{k+1}-1/2) = [(x_k+1)+1]^2 + (y_k -1/2-1)^2 - r^2$$
$$\boxed{= p_k + 2x_k - 2y_k + 5 = p_k + 2x_{k+1} - 2y_{k+1} +1}$$

Midpoint Circle Algorithm (cont.)

Summary of the algorithm:

- Given the starting point (0,r),

  Initialization,

  $P_0 = 5/4 - r$ =>float

  At each x position, 二選一，選下一個point

  if(pk < 0)

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

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

  else

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

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

Other Primitives

- The same concept can be extended to other primitives.
  - Ellipse, polynomials, splines, etc.

2D Polygon Filling

畫出邊線後要光柵化把中間格子填滿

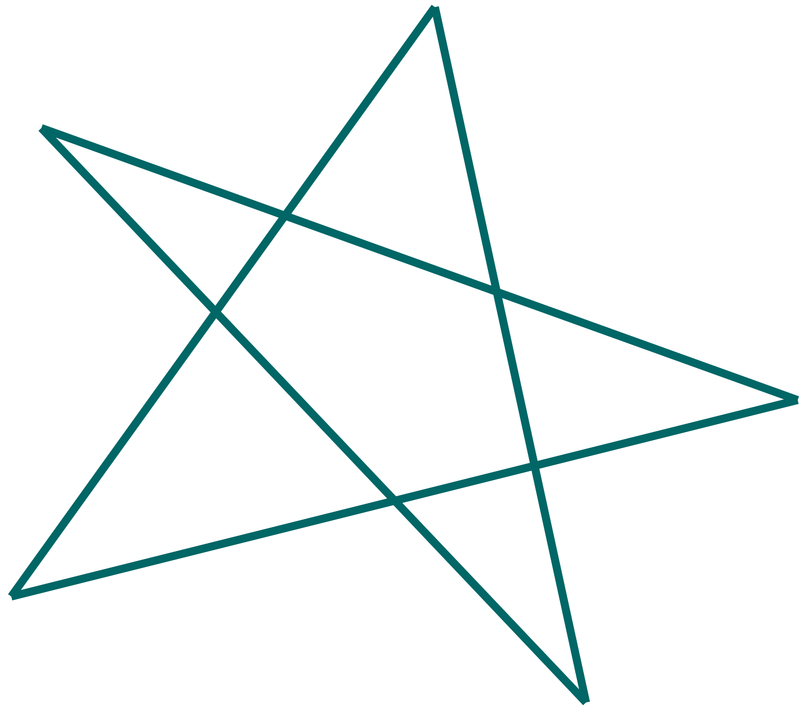
- Recall:
  - In computer graphics, we usually use polygons to approximate complex surfaces.
- Let's focus on the polygon filling !

General Polygons

畫出點之後會自動幫你連線，可能還會幫你補點(圖形可能會歪掉)

打斷convex(凸多邊形)法 : 三角形太細長不好
中間(扇形)打斷法 : 三角形數量會太多

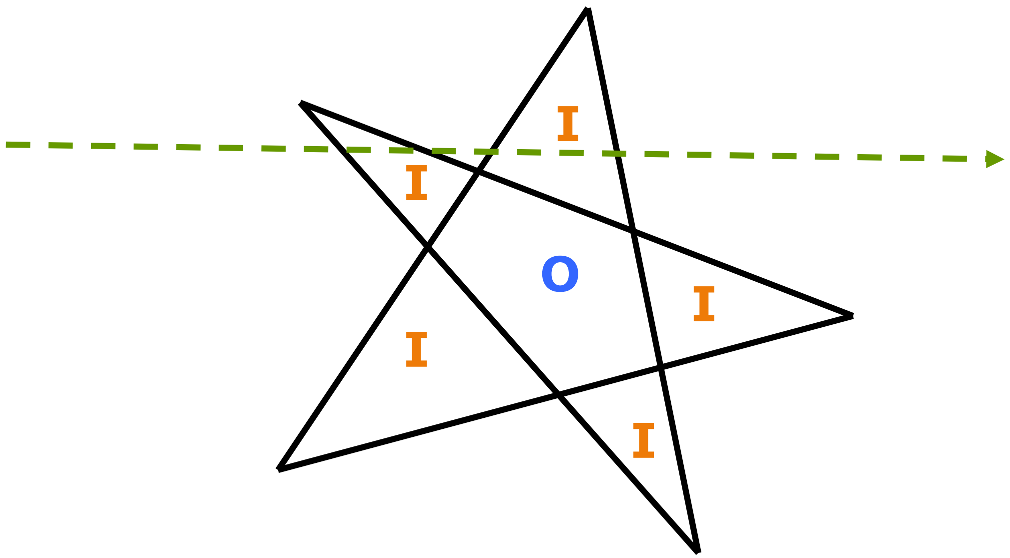
- Inside or Outside are not obvious
  - It’s not obvious when the polygon intersects itself.



Inside or Outside

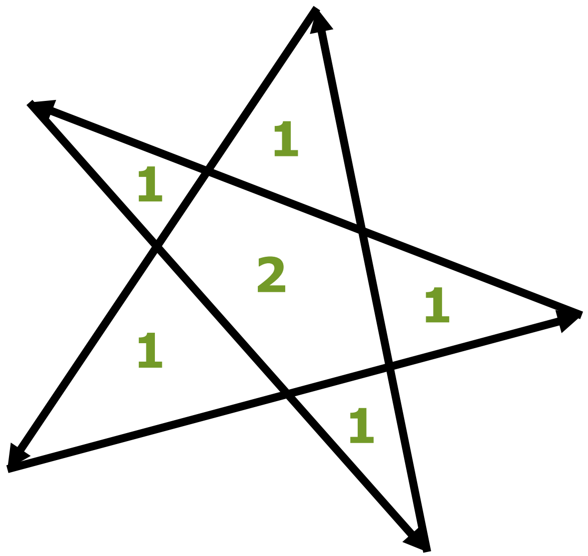
一條線經過，會經過幾個邊界，適用中間中空的圖形

- 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 判斷某個點是否在圖形內 : 外面有沒有點繞過他 這種計算方式會有太多重複運算的東西
  - 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

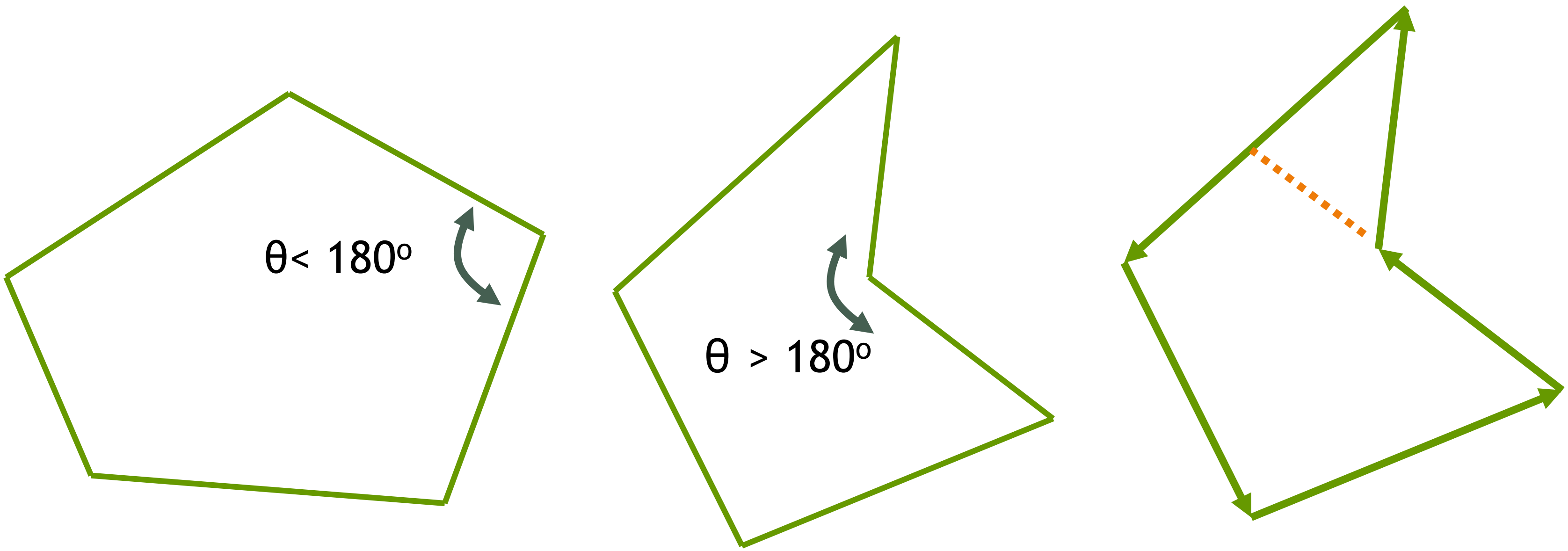


凹多邊形 凸多邊形

Concave vs. Convex

凹多邊形的例外太多
=>通常用凸多邊形，以三四五邊形為主

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

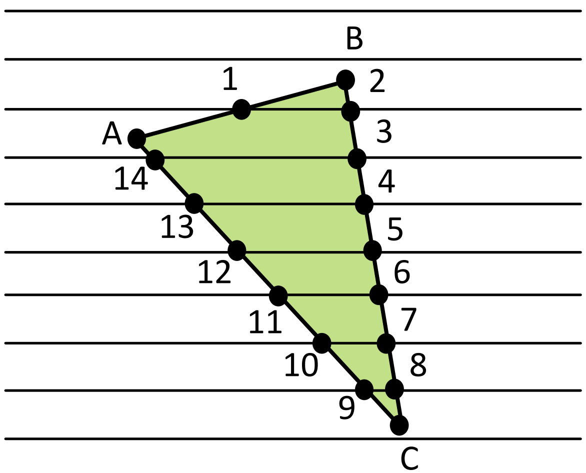


convex concave

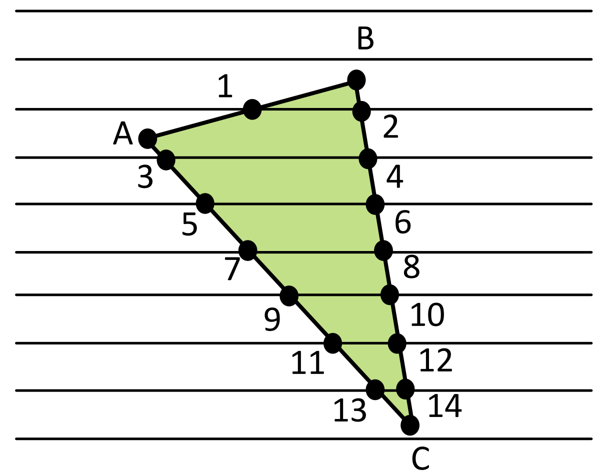
Polygon Filling by Scan Lines

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

先確定ABC三個端點，從最高點(B)開始
從B出發到A(第二高)，到底了就換線
從B出發到C(找第三高的線)
=>照高度去做顏色或normal內插



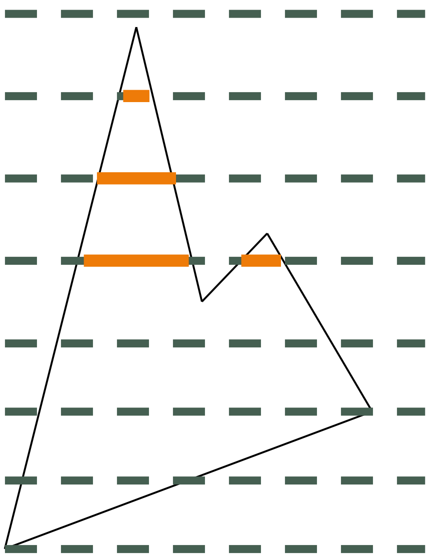
vertex order generated by vertex list

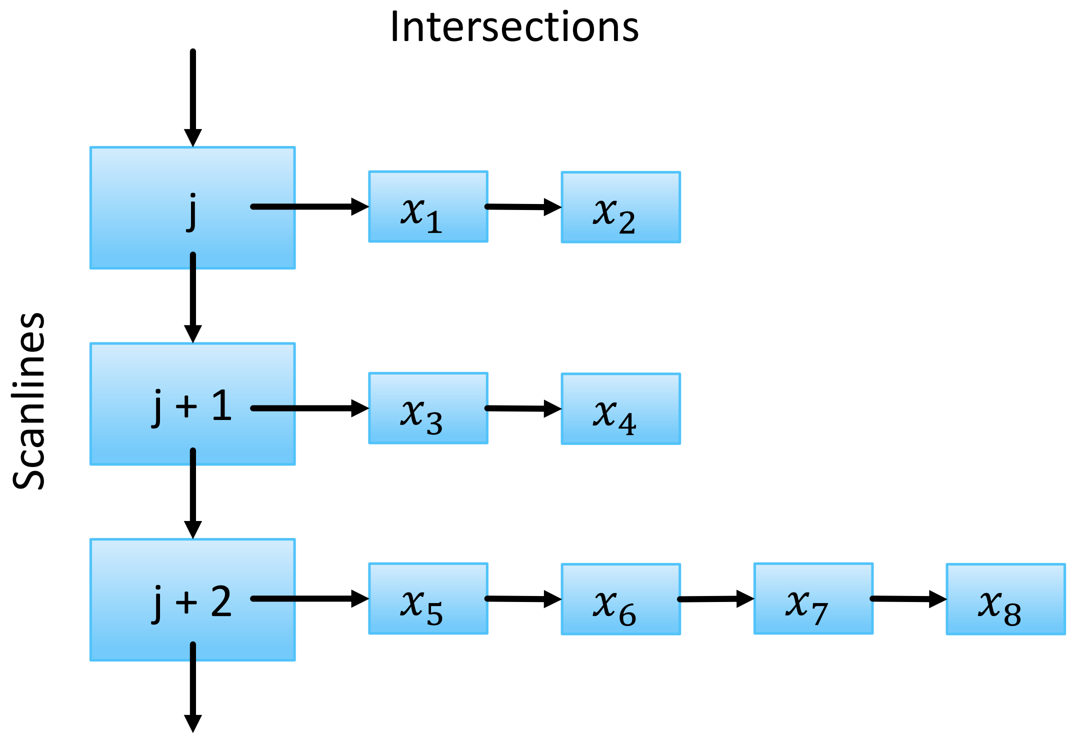


desired order

Data Structure for General Cases

先準備好點=>sorting=>畫線打點
=>同一個高度上會有哪些點





Applying the odd-even rule