Computer Graphics

Line, Circle, Ellipse Drawing

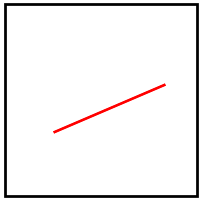
And

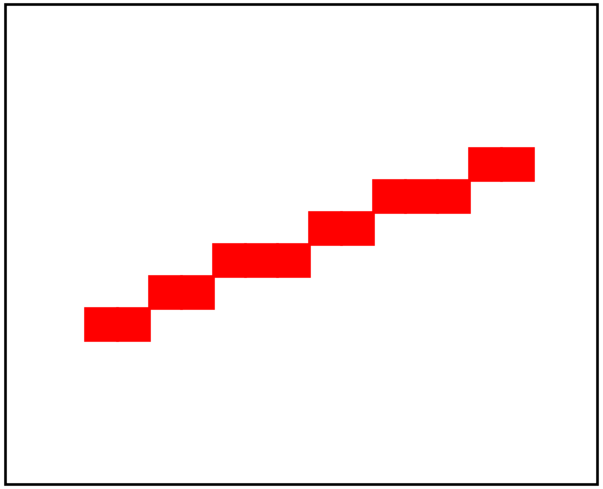
Fill Area Algorithms

Line Drawing

- Line drawing is fundamental to computer graphics.
- We must have fast and efficient line drawing functions.

Rasterization Problem: Given only the two end points, how to compute the intermediate pixels, so that the set of pixels closely approximate the ideal line.



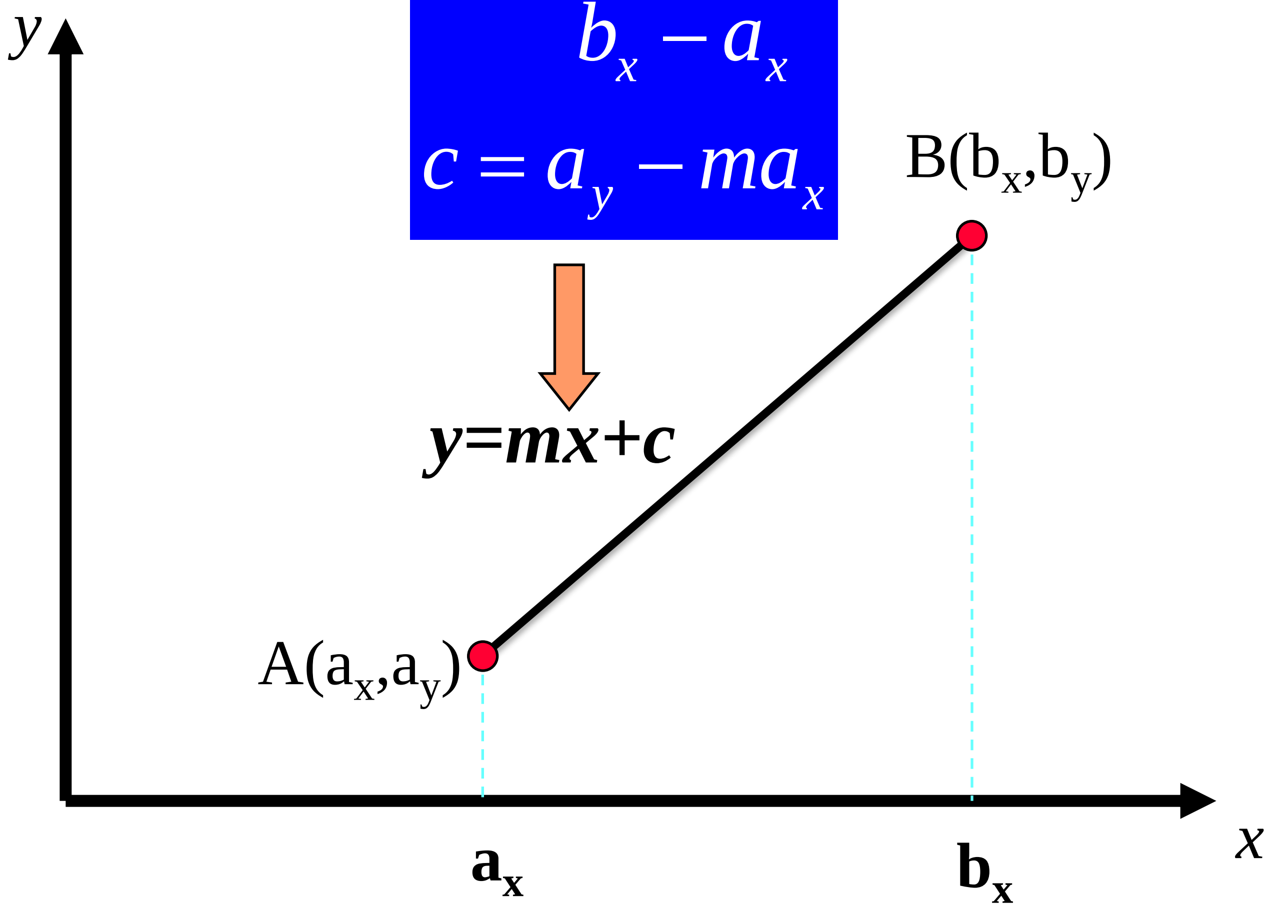


Line Drawing - Analytical Method

$$m = \frac{b_y - a_y}{b_x - a_x}$$

$$c = a_y - m a_x$$

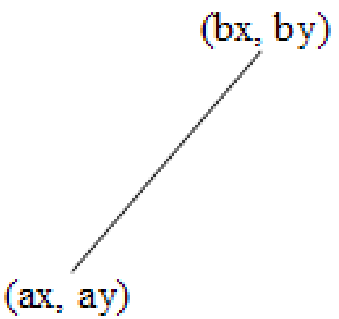
$$y = mx + c$$



Line Drawing - Analytical Method

```
//Given (ax, ay) and (bx, by)

double m = (double)(by-ay)/(bx-ax);
double c = ay - m*ax;
double y;
int    iy;
for (int x=ax; x<=bx; x++) {
    y = m*x + c;
    iy = round(y);
    setPixel(x, iy);
}
```



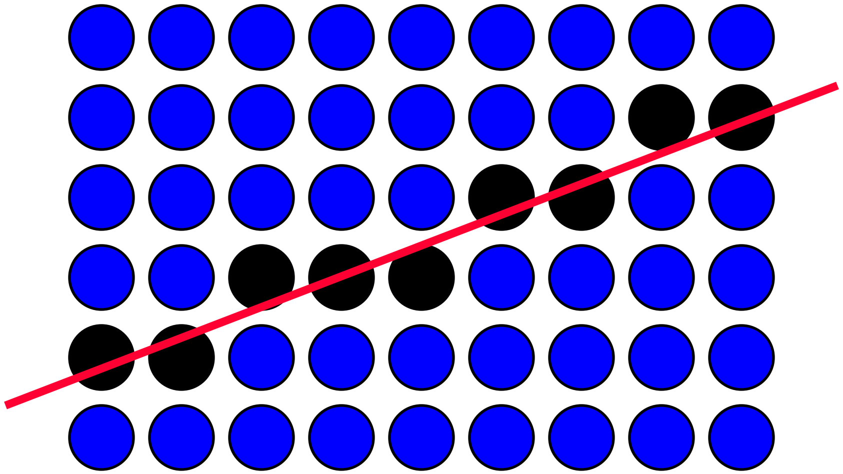
- Directly based on the analytical equation of a line.
- Involves floating point multiplication and addition
- Requires round-off function.

Incremental Algorithms

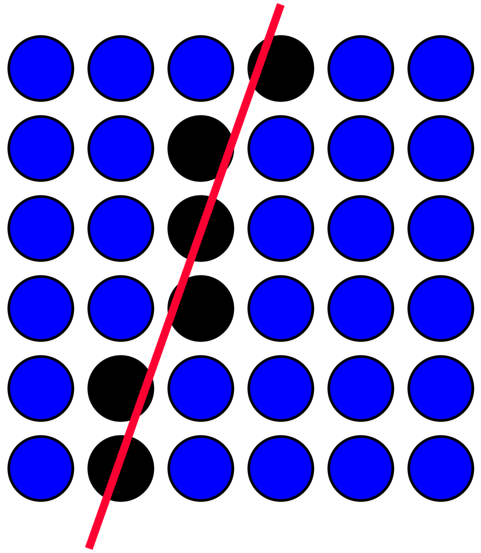
I have got a pixel on the line (Current Pixel).
How do I get the next pixel on the line?

Compute one point based on the previous point:

(x_0, y_0)…………………(x_k, y_k) (x_{k+1}, y_{k+1}) …….

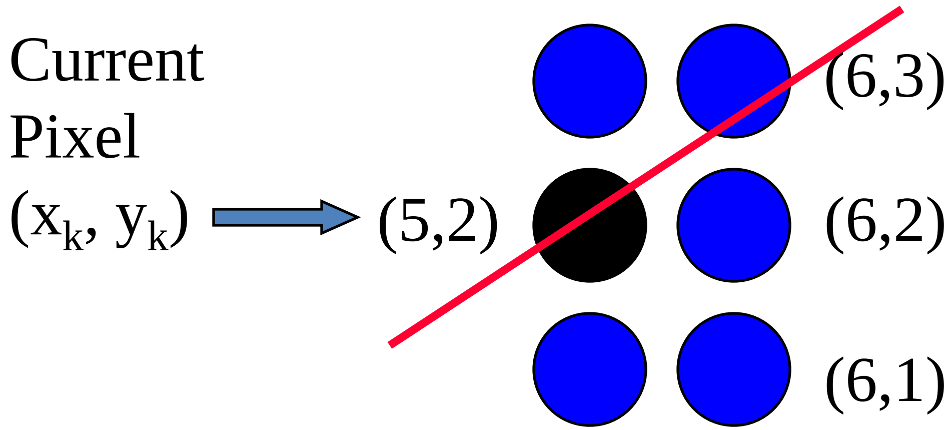


Next pixel on next column (when slope is small)



Next pixel on next row (when slope is large)

Incrementing along x



To find $(x_{k+1}, y_{k+!})$:

$\mathbf{x_{k+1} = x_k+1}$

$\mathbf{y_{k+1} = \ ?}$

- Assumes that the next pixel to be set is on the next column of pixels (Incrementing the value of x !)
- Not valid if slope of the line is large

Line Drawing - DDA

Digital Differential Analyzer Algorithm is an incremental algorithm.

Assumption: Slope is less than 1 (Increment along x).

Current Pixel = (x_k, y_k).

(x_k, y_k) lies on the given line. → $y_k = m.x_k + c$

Next pixel is on next column. → $x_{k+1} = x_k+1$

Next point (x_{k+1}, y_{k+1}) on the line → $y_{k+1} = m.x_{k+1} + c$

$$= m(x_k+1) + c$$

$$= y_k + m$$

Given a point (x_k, y_k) on a line, the next point is given by

$$\mathbf{x_{k+1} = x_k+1}$$

$$\mathbf{y_{k+1} = y_k + m}$$

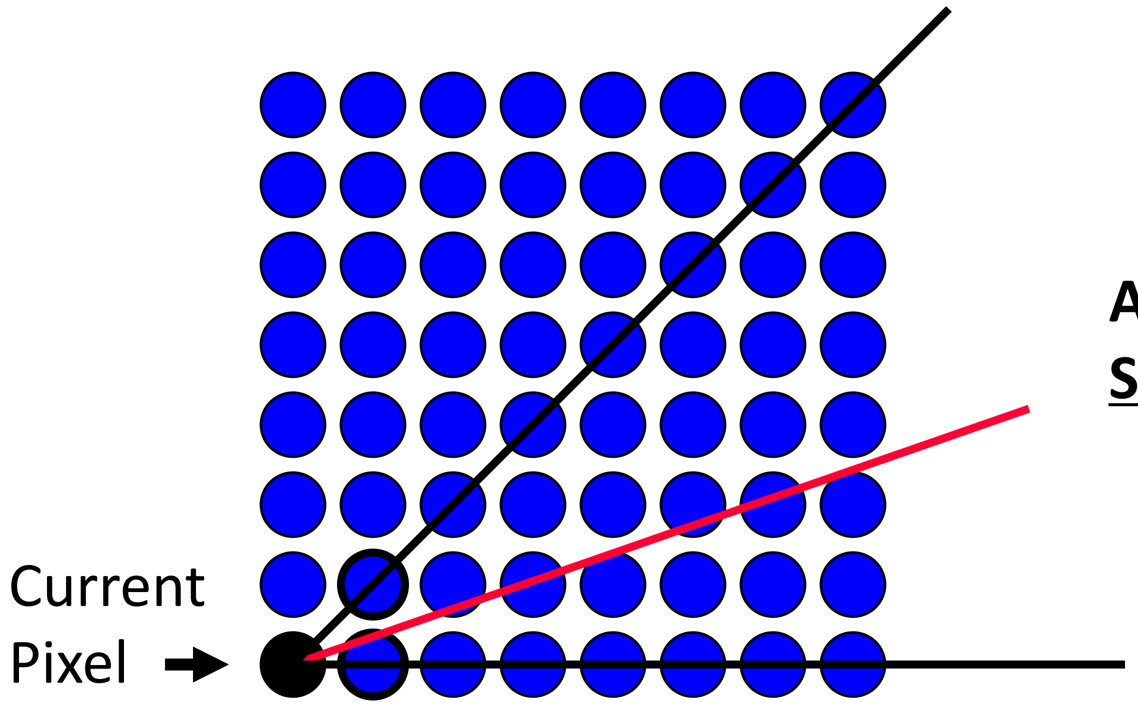
Line Drawing - DDA

```
double m = (double) (by-ay)/(bx-ax);
double y = ay;
int    iy;
for (int x=ax; x<=bx; x++) {
    iy = round(y);
    putPixel(x, iy, RED);
    y += m;
}
```

- Does not involve any floating point multiplication
- Involves floating point addition.
- Requires round-off function

Midpoint Algorithm

Midpoint algorithm is an incremental algorithm

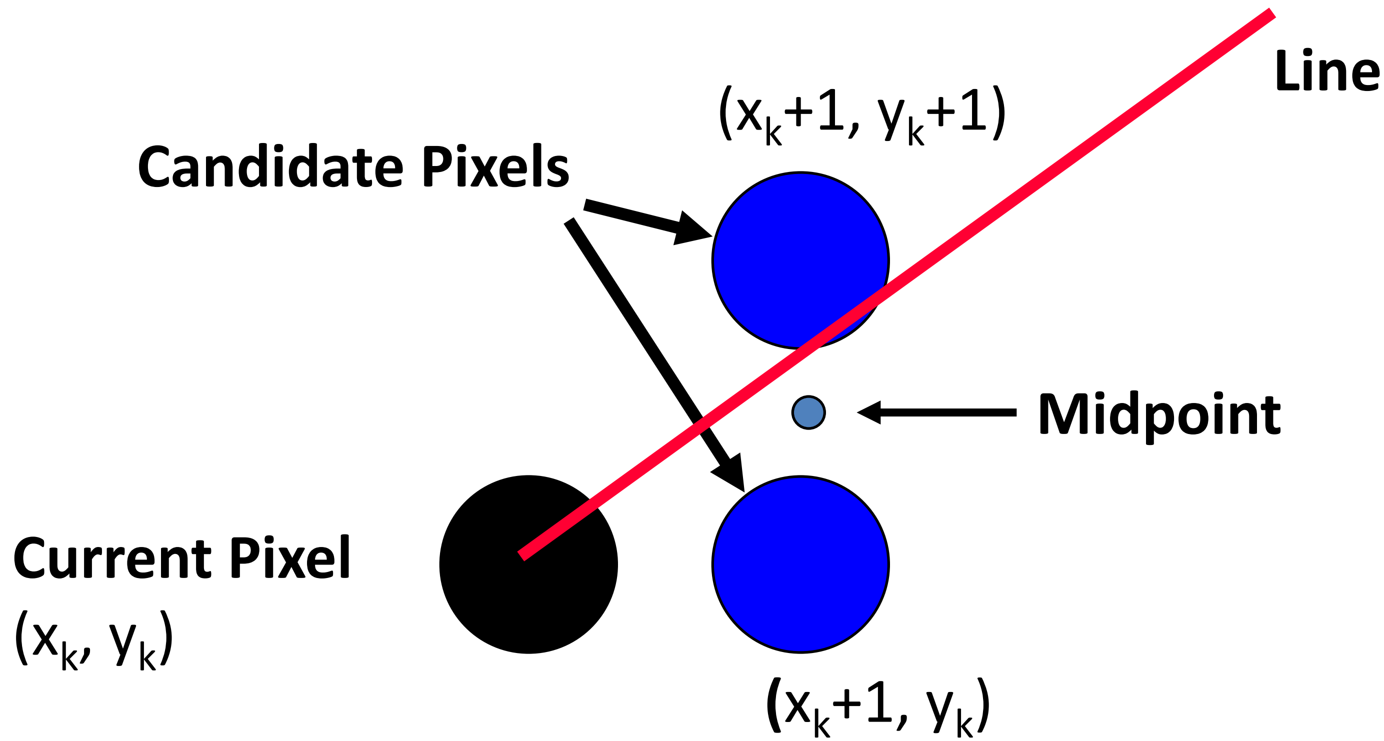


Assumption:
Slope < 1

$x_{k+1} = x_k + 1$

$y_{k+1} = \text{Either } y_k \text{ or } y_k + 1$

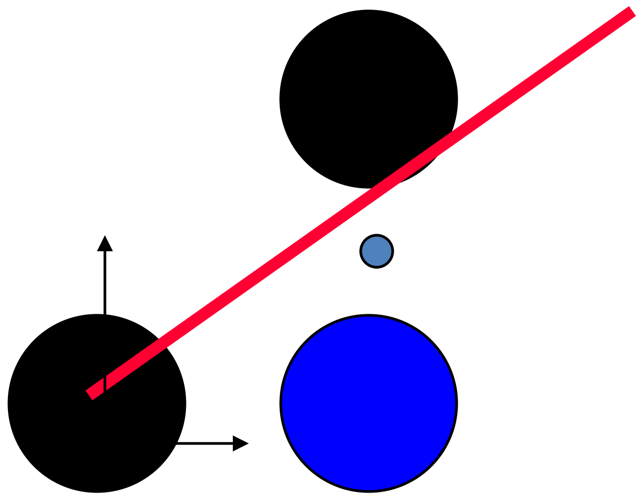
Midpoint Algorithm - Notations



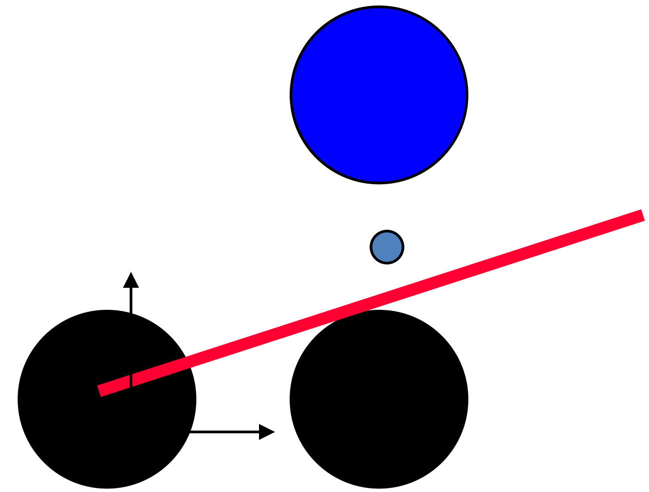
Coordinates of Midpoint = $(x_k+1,\ y_k+½)$

Midpoint Algorithm:

Choice of the next pixel



Midpoint Below Line



Midpoint Above Line

- If the midpoint is below the line, then the next pixel is **(x_k+1, y_k+1)**
- If the midpoint is above the line, then the next pixel is **(x_k+1, y_k)**

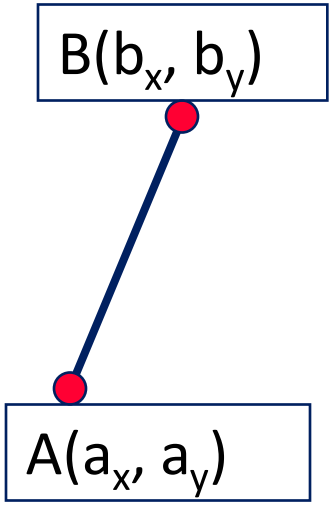
Equation of a line revisited

Equation of the line: $$\frac{y-a_y}{b_y-a_y}=\frac{x-a_x}{b_x-a_x}$$

Let $w = b_x - a_x$, and $h = b_y - a_y$

Then, $h\,(x - a_x) - w\,(y - a_y) = 0$

(h, w, a_x, a_y are all integers)

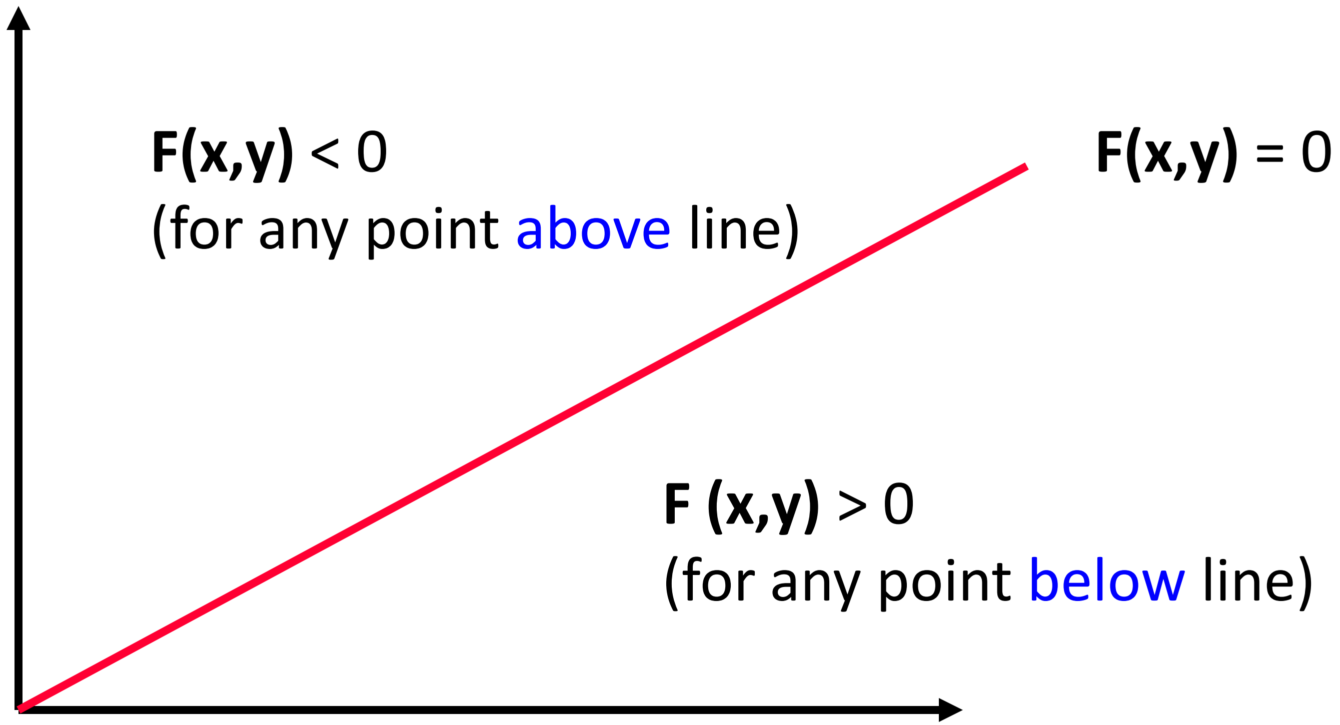


In other words, every point (x, y) on the line satisfies the equation $F(x, y) = 0$, where

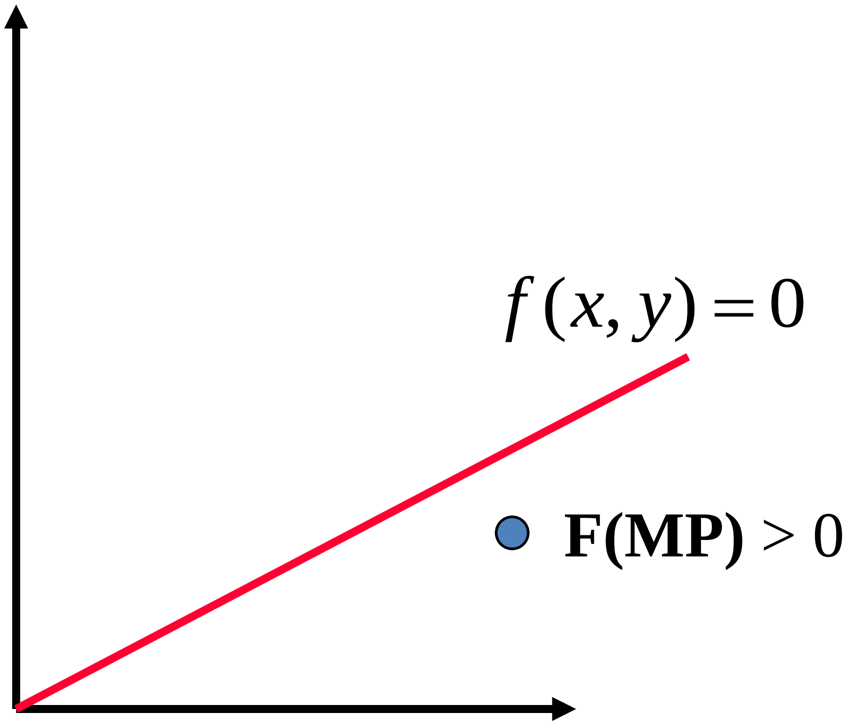
$$\mathbf{F(x, y) = h\,(x - a_x) - w\,(y - a_y)}$$

Midpoint Algorithm:

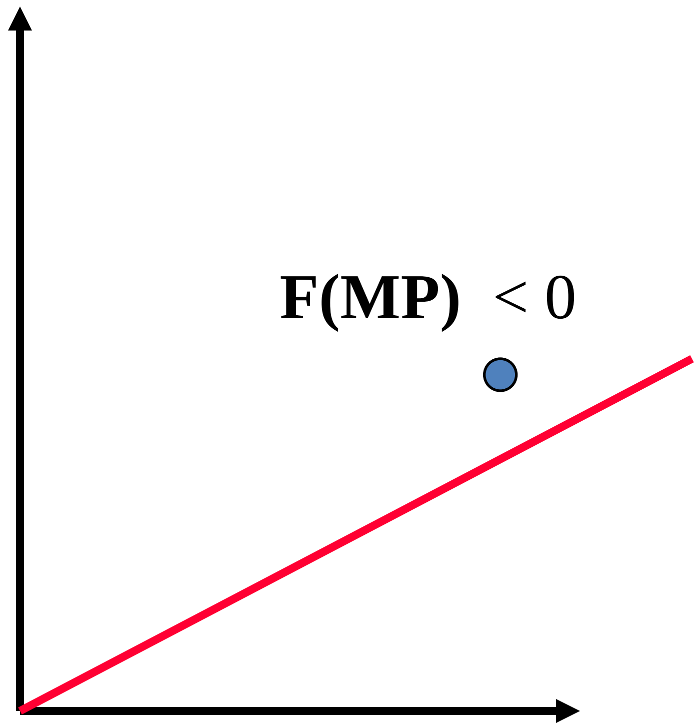
Regions below and above the line



Midpoint Algorithm:
Decision Criteria



Midpoint below line



Midpoint above line

Midpoint Algorithm

Decision Criteria

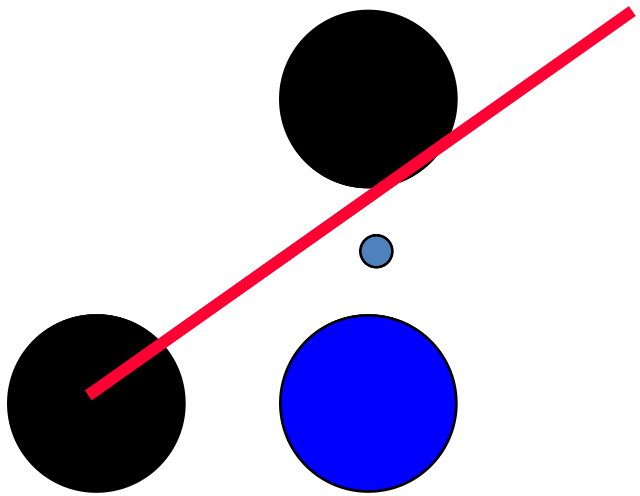
Decision Parameter

$\mathbf{F(MP) = F(x_k+1,\ y_k+\tfrac{1}{2})\ =\ F_k}$ *(Notation)*

If $F_k < 0$: The midpoint is above the line. So the next pixel is $(x_k+1,\ y_k)$

If $F_k \geq 0$: The midpoint is below or on the line. So the next pixel is $(x_k+1,\ y_k+1)$

Midpoint Algorithm – Story so far

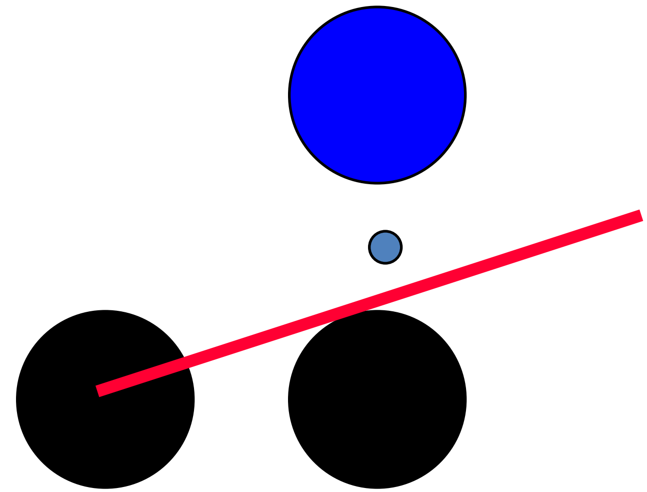


Midpoint Below Line

$$F_k > 0$$

$$y_{k+1} = y_k + 1$$

Next pixel $= (x_k+1, y_k+1)$



Midpoint Above Line

$$F_k < 0$$

$$y_{k+1} = y_k$$

Next pixel $= (x_k+1, y_k)$

Midpoint Algorithm:

Update Equation

$$F_k = F(x_k+1, y_k+ ½) = h\,(x_k+1 - a_x) - w\,(y_k+½ - a_y) \longrightarrow 1$$

$$F_{k+1} = F(x_{k+1}+1, y_{k+1}+ ½) = h\,(x_{k+1}+1 - a_x) - w\,(y_{k+1}+½ - a_y) \longrightarrow 2$$

$$2 - 1 \Rightarrow \quad F_{k+1} - F_k = h\,(x_{k+1} - x_k) - w\,(y_{k+1} - y_k)$$

$$F_{k+1} - F_k = h\,(x_k + 1 - x_k) - w\,(y_{k+1} - y_k)$$

$$F_{k+1} = F_k + h - w\,(y_{k+1} - y_k)$$

Thus, given $F_k < 0$, $y_{k+1} = y_k$ **then** $F_{k+1} = F_k + h$

given $F_k \geq 0$, $y_{k+1} = y_k+1$ **then** $F_{k+1} = F_k + h - w$

$$F_0 = h\,(a_x+1 - a_x) - w\,(a_y+½ - a_y) = h - w/2$$

Midpoint Algorithm:

Update Equation

Update Equation

Summary:

$$F_{k+1} = F_k + h - w\,(y_{k+1} - y_k)$$

given $F_k < 0$, $y_{k+1} = y_k$ then $F_{k+1} = F_k + h$

given $F_k \geq 0$, $y_{k+1} = y_k + 1$ then $F_{k+1} = F_k + h - w$

$$F_0 = h - w/2$$

Midpoint Algorithm

```
int   h = by-ay;
int   w = bx-ax;
float F = h-w/2;
int   y = ay;
for (int x=ax; x<=bx; x++) {
   putPixel(x, y);

   if(F < 0)
       F += h;
   else {
       F += h-w;
       y++;
   }
}
```

Bresenham's Algorithm

(Improved Midpoint Algorithm)

```
int   h = by-ay;
int   w = bx-ax;
int   F = 2*h-w;
int   y = ay;
for (int x=ax; x<=bx; x++) {
   putPixel(x, y);

   if(F < 0)
      F += 2*h;
   else {
      F += 2*(h-w);
      y++;
   }
}
```

Bresenham’s Line Algorithm

- An **accurate**, **efficient** raster line drawing algorithm developed by Bresenham, scan converts lines using only *incremental integer* calculations that can be adapted to display circles and other curves.

- Keeping in mind the symmetry property of lines, lets derive a more efficient way of drawing a line.

  Starting from the left end point (x_0,y_0) of a given line , we step to each successive column (x position) and plot the pixel whose scan-line y value closest to the line path

  Assuming we have determined that the pixel at (x_k,y_k) is to be displayed, we next need to decide which pixel to plot in column x_{k+1}.

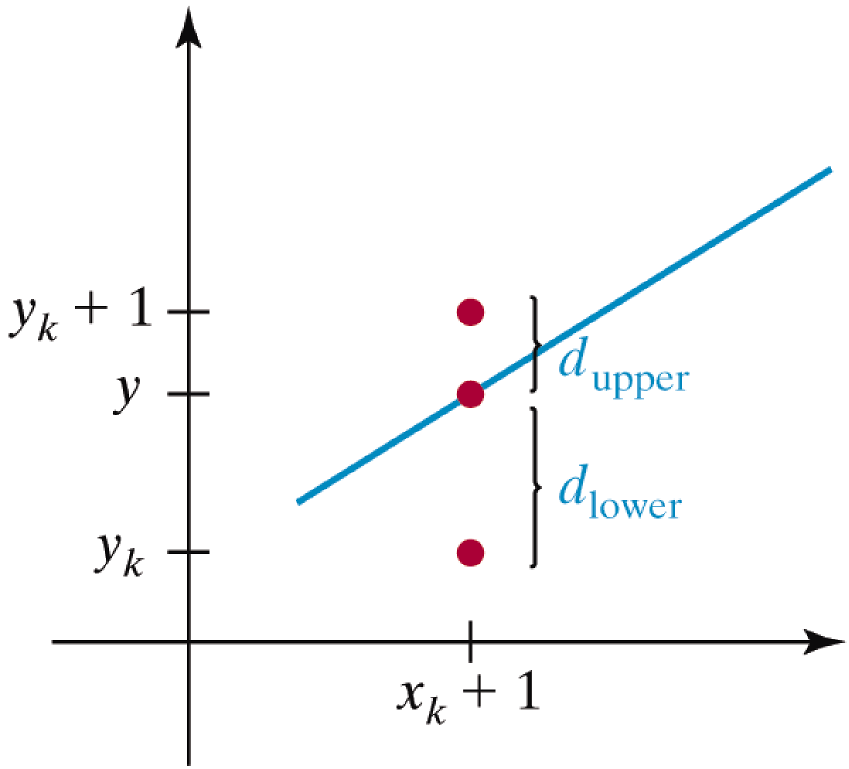


Figure 3-11

Vertical distances between pixel positions and the line y coordinate at sampling position $x_k + 1$.

Bresenham Line Algorithm (cont)

Choices are $(x_k +1, y_k)$ and (x_k+1, y_K+1)

$$d_1 = y - y_k = m(x_k + 1) + b - y_k$$

$$d_2 = (y_k + 1) - y = y_k + 1 - m(x_k + 1) - b$$

- The difference between these 2 separations is

  d1-d2 = 2m(xk + 1) – 2 yk + 2b – 1

- A decision parameter p_k for the k^{th} step in the line algorithm can be obtained by rearranging above equation so that it involves only *integer calculations*

Bresenham's Line Algorithm

- Define

$$P_k = \Delta x\ (d_1 - d_2) = 2\Delta y x_k - 2\ \Delta x y_k + c$$

- The sign of P_k is the same as the sign of d_1-d_2, since $\Delta x > 0$.

  Parameter c is a constant and has the value $2\Delta y + \Delta x(2b-1)$

  (independent of pixel position)

- If *pixel at* y_k is closer to line-path than pixel at $y_k + 1$

  (i.e, if $d_1 < d_2$*)* then p_k is negative. We plot lower pixel in such a case. Otherwise , upper pixel will be plotted.

Bresenham's algorithm (cont)

- At step *$k + 1$*, the decision parameter can be evaluated as,

  $p_k+1 = 2\Delta y x_k+1 - 2\Delta x y_k+1 + c$

- Taking the difference of p_{k+1} and p_k we get the following.

  $p_{k+1} – p_k = 2\Delta y(x_{k+1}- x_k)-2\Delta x(y_{k+1} – y_k)$

- But, $x_{k+1} = x_k +1$, so that

  $p_{k+1} = p_k + 2\Delta y - 2 \Delta x(y_{k+1} – y_k)$

- Where the term $y_{k+1}-y_k$ is either 0 or 1, depending on the sign of *parameter* p_k

Bresenham's Line Algorithm

- The first parameter p_0 is directly computed

  $p_0 = 2\ \Delta y x_k - 2\ \Delta x y_k + c = 2\ \Delta y x_k – 2\ \Delta y + \Delta x\ (2b\text{-}1)$

- Since (x_0,y_0) satisfies the line equation , we also have

  $y_0 = \Delta y/\ \Delta x * x_0 + b$

- Combining the above 2 equations , we will have

  $p_0 = 2\Delta y – \Delta x$

  The constants $2\Delta y$ and $2\Delta y\text{-}2\Delta x$ are calculated once for each time to be scan converted

Bresenham's Line Algorithm

- So, the arithmetic involves only integer addition and subtraction of 2 constants

Input the two end points and store the left end point in (x_0, y_0)

Load (x_0, y_0) into the frame buffer ***(plot the first point)***

Calculate the constants Δx, Δy, $2\Delta y$ and $2\Delta y - 2\Delta x$ and obtain the starting value for the decision parameter as

$$p_0 = 2\Delta y - \Delta x$$

Bresenham's Line Algorithm

At each x_k along the line, starting at k=0, perform the following test:

If $p_k < 0$, the next point is (x_k+1, y_k) and

$$p_{k+1} = p_k + 2\Delta y$$

Otherwise

Point to plot is (x_k+1, y_k+1)

$$p_{k+1} = p_k + 2\Delta y - 2\Delta x$$

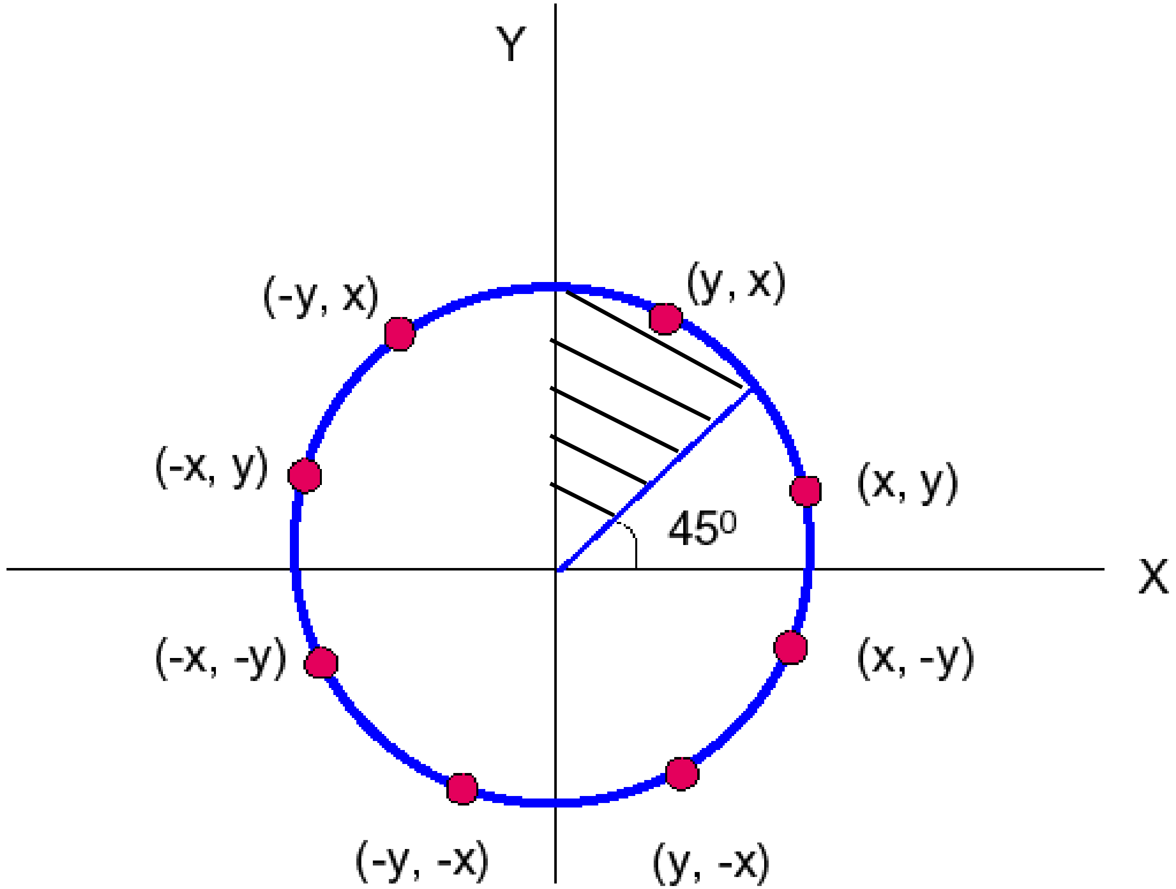
Repeat step 4 (above step) Δx times

Midpoint Circle Drawing Algorithm

- To determine the closest pixel position to the specified circle path at each step.
- For given radius r and screen center position (x_c, y_c), calculate pixel positions around a circle path centered at the coodinate origin (0,0).
- Then, move each calculated position (x, y) to its proper screen position by adding **x_c to x** and **y_c to y**.
- Along the circle section from x=0 to x=y in the first quadrant, the gradient varies from 0 to -1.

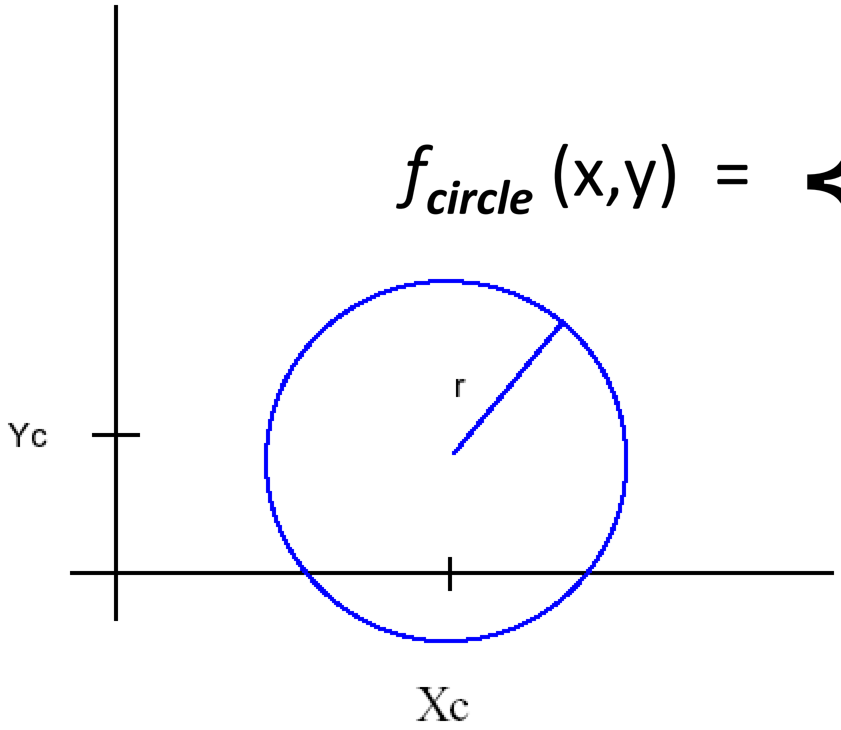
Midpoint Circle Drawing Algorithm

- 8 segments of octants for a circle:



Midpoint Circle Drawing Algorithm

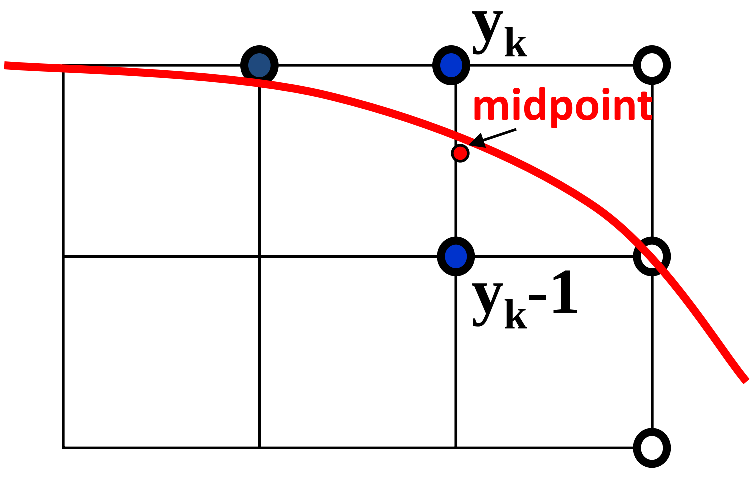
- Circle function: $\boxed{f_{circle}(x,y) = x^2 + y^2 - r^2}$

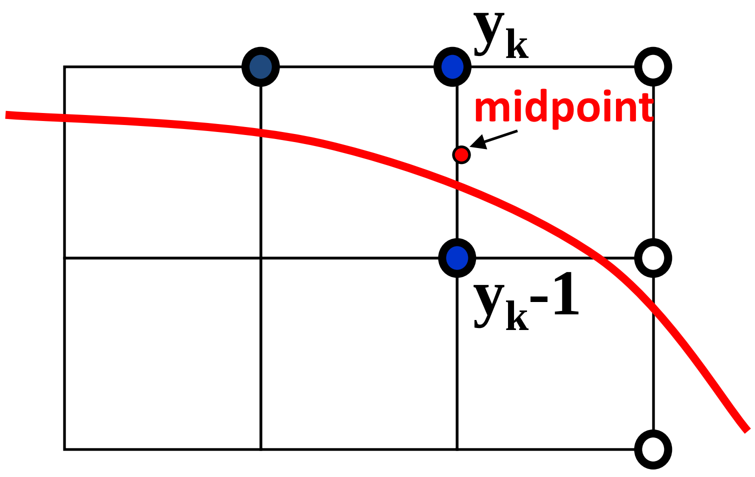


$$f_{circle}(x,y) = \begin{cases} > 0, & (x,y) \text{ outside the circle} \\ < 0, & (x,y) \text{ inside the circle} \\ = 0, & (x,y) \text{ is on the circle boundary} \end{cases}$$

Gaurav Raj, Lovely professional University, Punjab

Midpoint Circle Drawing Algorithm





$F_k < 0$	$F_k >= 0$
$y_{k+1} = y_k$	$y_{k+1} = y_k - 1$
Next pixel = (x_k+1, y_k)	Next pixel = (x_k+1, y_k-1)

Midpoint Circle Drawing Algorithm

We know $x_{k+1} = x_k+1$,

$$F_k = F(x_k+1, y_k - ½)$$
$$F_k = (x_k +1)^2 + (y_k - ½)^2 - r^2 \qquad \text{-------- (1)}$$

$$F_{k+1} = F(x_{k+1}+1, y_{k+1} - ½)$$
$$F_{k+1} = (x_k +2)^2 + (y_{k+1} - ½)^2 - r^2 \qquad \text{-------- (2)}$$

(2) – (1)

$$\boxed{F_{k+1} = F_k + 2(x_k+1) + (y^2_{k+1} – y^2_k) - (y_{k+1} – y_k) + 1}$$

If $F_k < 0$, $\boxed{F_{k+1} = F_k + 2x_{k+1}+1}$

If $F_k >= 0$, $\boxed{F_{k+1} = F_k + 2x_{k+1}+1 – 2y_{k+1}}$

Midpoint Circle Drawing Algorithm

For the initial point, $(x_0 , y_{0)} = (0, r)$

$$f_0 = f_{circle}\,(1, r\text{-}½\ )$$

$$= 1 + (r\text{-}½\ )^2 - r^2$$

$$= \frac{5}{4} - r$$

$$\approx 1 - r$$

Midpoint Circle Drawing Algorithm

Example:

Given a circle radius = 10, determine the circle octant in the first octant from x=0 to x=y.

Solution:

$$f_0 = \frac{5}{4} - r$$

$$= \frac{5}{4} - 10$$

$$= -8.75$$

$$\approx -9$$

Midpoint Circle Drawing Algorithm

Initial (x_0, y_0) = (1,10)
Decision parameters are: $2x_0$ = 2, $2y_0$ = 20

k	F_k	*x*	*y*	$2x_{k+1}$	$2y_{k+1}$
0	-9	1	10	2	20
1	-9+2+1=-6	2	10	4	20
2	-6+4+1=-1	3	10	6	20
3	-1+6+1=6	4	9	8	18
4	6+8+1-18=-3	5	9	10	18
5	-3+10+1=8	6	8	12	16
6	8+12+1-16=5	7	7	14	14

Midpoint Circle Drawing Algorithm

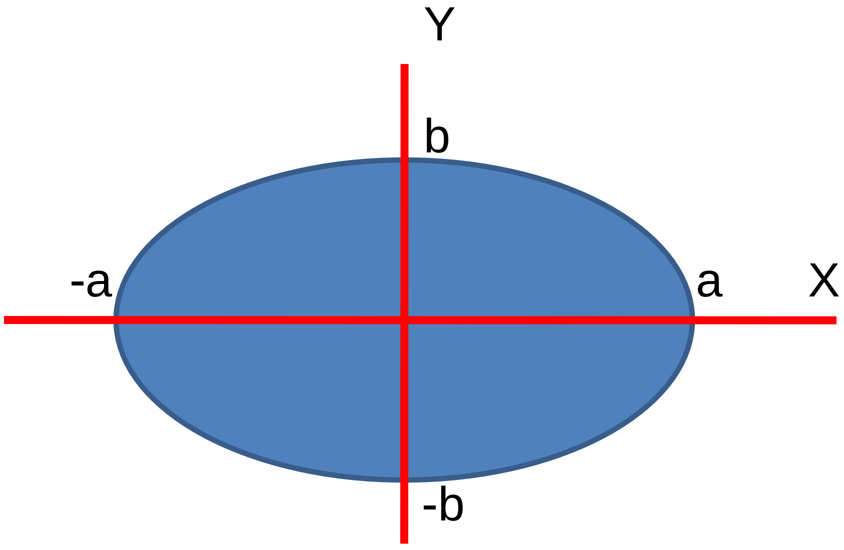
```
void circleMidpoint(int xCenter, int yCenter, int radius)
{
   int x = 0;
   Int y = radius;
   int f = 1 - radius;

    circlePlotPoints(xCenter, yCenter, x, y);
    while (x < y) {
      x++;
      if (f < 0)
      f += 2*x + 1;
      else {
         y--;
         f += 2*(x-y)+1;
      }
      circlePlotPoints(xCenter, yCenter, x, y);
   }
}
```

Midpoint Circle Drawing Algorithm

```
void circlePlotPoints( int xCenter, int yCenter,
                       int x, int y)
{
      setPixel (xCenter + x, yCenter + y);
      setPixel (xCenter – x, yCenter + y);
      setPixel (xCenter + x, yCenter – y);
      setPixel (xCenter – x, yCenter – y);
      setPixel (xCenter + y, yCenter + x);
      setPixel (xCenter – y, yCenter + x);
      setPixel (xCenter + y, yCenter – x);
      setPixel (xCenter – y, yCenter – x);
}
```

Ellipse Drawing

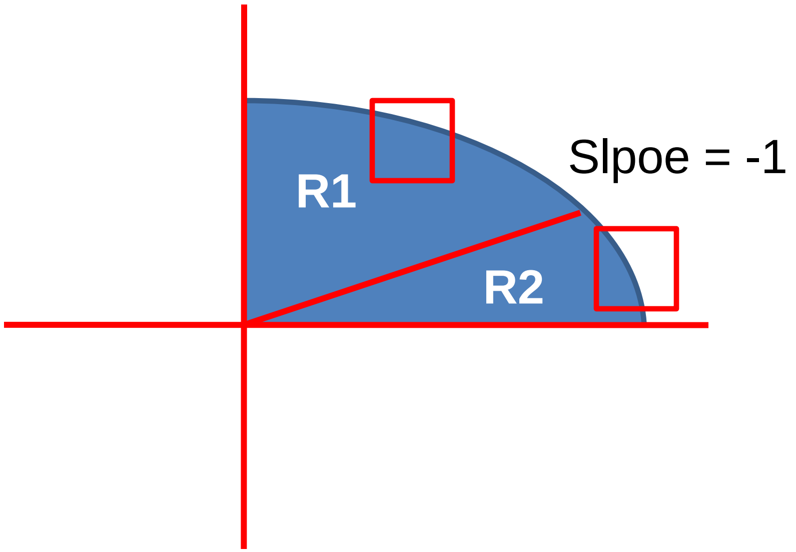


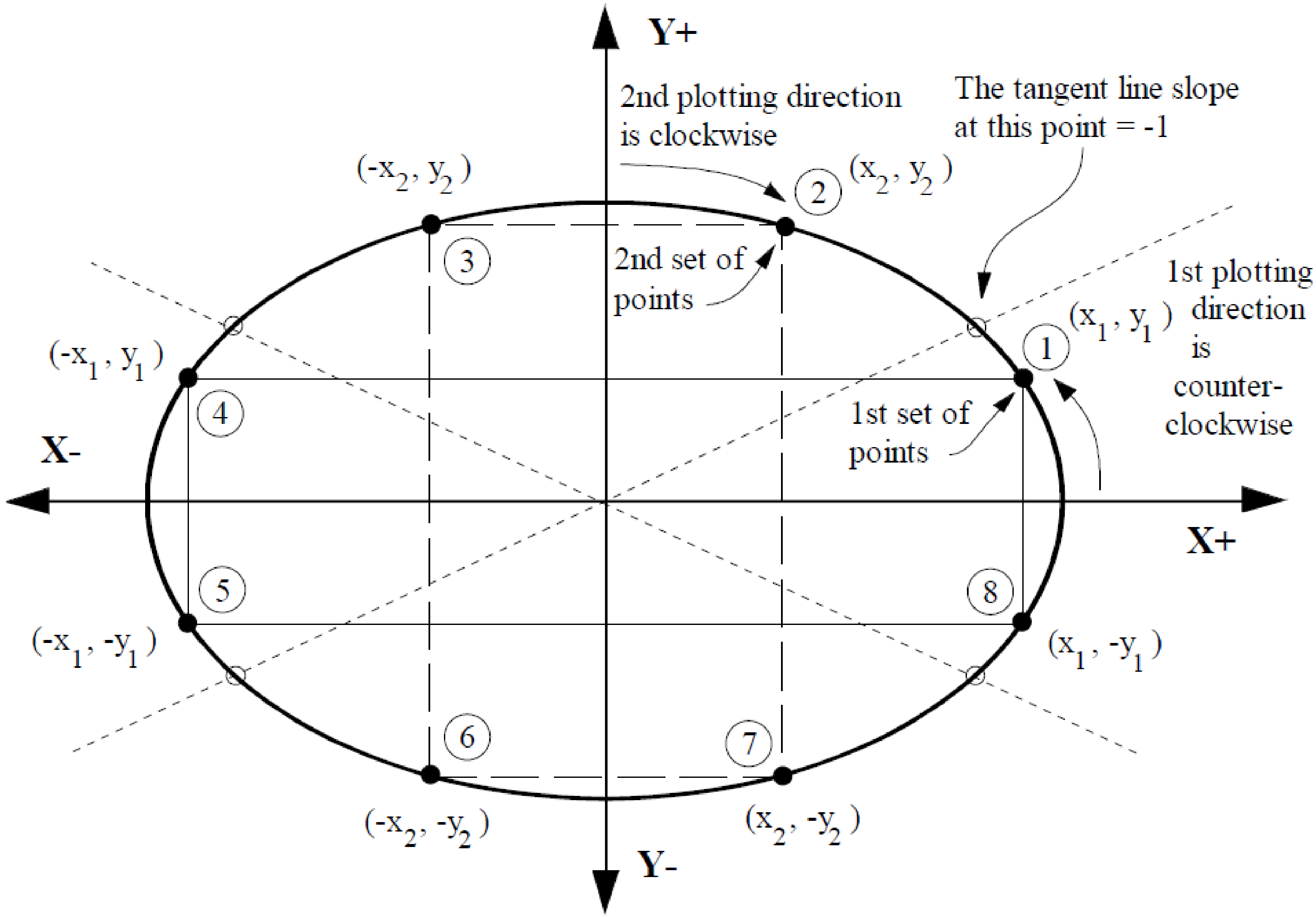
Equation of ellipse :

$$F(X,Y) = b^2X^2 + a^2Y^2 - a^2b^2 = 0$$

Length of major axis: **2a and 2b**

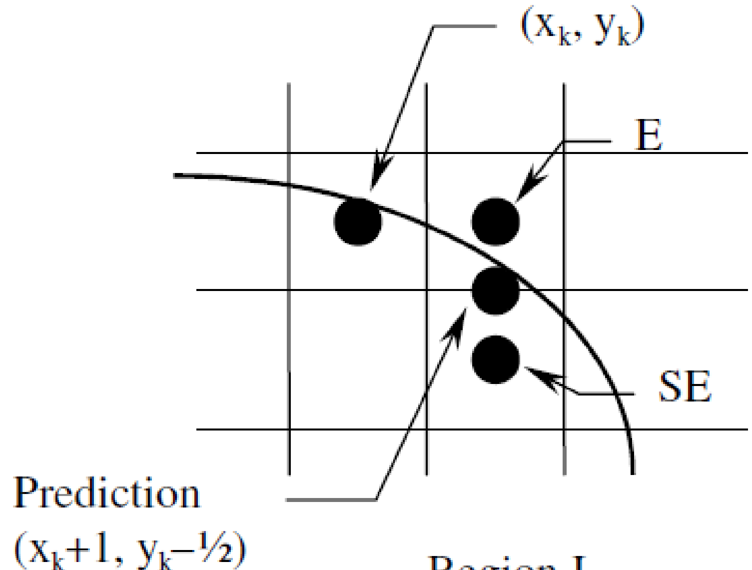
- We need to obtain points on the contour where the slope of the curve is -1.
- This help to demarcate region R1 and R2.
- Choice of pixels in Region R1 is between **E and SE**, Where in R2 , it is **S and SE**.





This figure indicates the two sets of points in the first quadrant that get plotted. The plotting algorithm uses two sets with 4-point symmetry.

In region I (dy/dx > –1),



Region I

x is always incremented in each step, i.e. $x_{k+1} = x_k + 1$.

$y_{k+1} = y_k$ if E is selected, or $y_{k+1} = y_k - 1$ if SE is selected.

In order to make decision between S and SE, a prediction $(x_k+1, y_k-½)$ is set at the middle between the two candidate pixels. A prediction function P_k can be defined as follows:

$$
\begin{aligned}
P_k &= f(x_k+1,\ y_k-½) \\
&= b^2(x_k+1)^2 + a^2(y_k-½)^2 - a^2b^2 \\
&= b^2(x_k^{\ 2} + 2x_k + 1) + a^2(y_k^{\ 2} - y_k + ¼) - a^2b^2
\end{aligned}
$$

If $P_k < 0$, select E :

$$
\begin{aligned}
P_{k+1}^{\ E} &= f(x_k+2,\ y_k-½) \\
&= b^2(x_k+2)^2 + a^2(y_k-½)^2 - a^2b^2 \\
&= b^2(x_k^{\ 2} + 4x_k + 4) + a^2(y_k^{\ 2} - y_k + ¼) - a^2b^2
\end{aligned}
$$

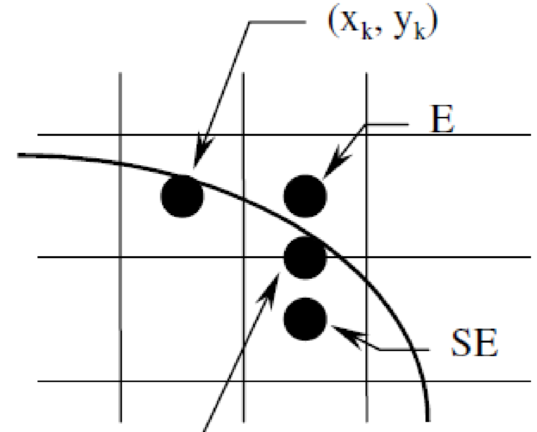
Change of $P_k^{\ E}$ is: $\Delta P_k^{\ E} = P_{k+1}^{\ E} - P_k = b^2(2x_k + 3)$

If $P_k > 0$, select SE :

$$
\begin{aligned}
P_{k+1}^{\ SE} &= f(x_k+2,\ y_k-3/2) \\
&= b^2(x_k+2)^2 + a^2(y_k-3/2)^2 - a^2b^2 \\
&= b^2(x_k^{\ 2} + 4x_k + 4) + a^2(y_k^{\ 2} - 3y_k + 9/4) - a^2b^2
\end{aligned}
$$

Change of $P_k^{\ SE}$ is $\Delta P_k^{\ SE} = P_{k+1}^{\ SE} - P_k = b^2(2x_k + 3) - 2a^2(y_k - 1)$

In region I $(dy/dx > -1)$,



Calculate the changes of ΔP_k:

If E is selected,

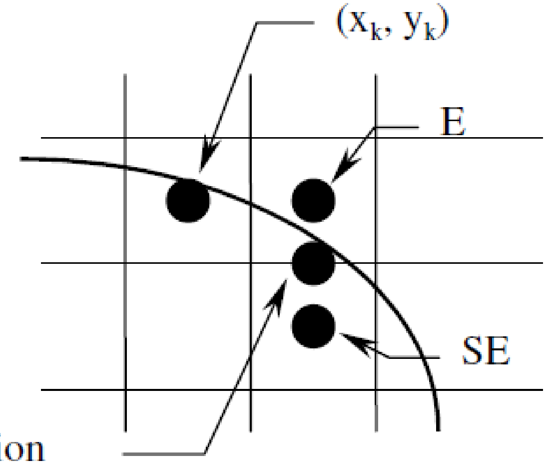
$$\Delta P_{k+1}^{E} = b^2(2x_k + 5)$$

$$\Delta^2 P_k^{E} = \Delta P_{k+1}^{E} - \Delta P_k^{E} = 2b^2$$

$$\Delta P_{k+1}^{SE} = b^2(2x_k + 5) - 2a^2(y_k - 1)$$

$$\Delta^2 P_k^{SE} = \Delta P_{k+1}^{SE} - \Delta P_k^{SE} = 2b^2$$

In region I ($dy/dx > -1$),



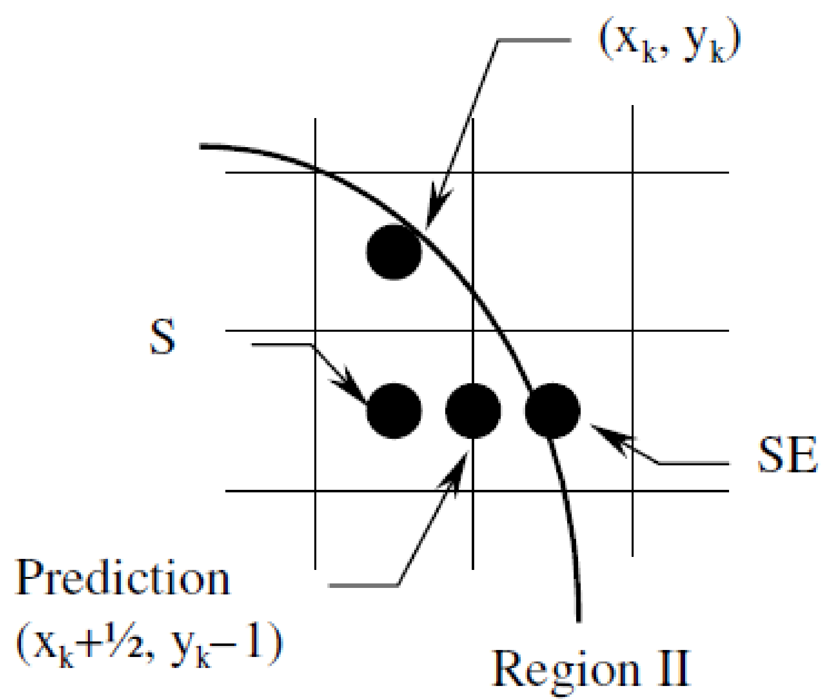
If SE is selected,

$$\Delta P_{k+1}^{E} = b^2(2x_k + 5)$$
$$\Delta^2 P_k^{E} = \Delta P_{k+1}^{E} - \Delta P_k^{E} = 2b^2$$

$$\Delta P_{k+1}^{SE} = b^2(2x_k + 5) - 2a^2(y_k - 2)$$
$$\Delta^2 P_k^{SE} = \Delta P_{k+1}^{SE} - \Delta P_k^{SE} = 2(a^2 + b^2)$$

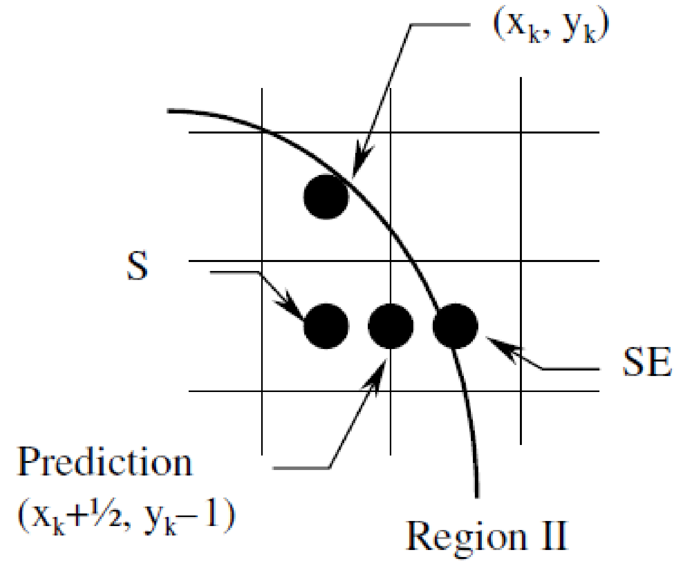
Initial values:

$$x_0 = 0,\ y_0 = b,\ P_0 = b^2 + \tfrac{1}{4}a^2(1 - 4b)$$
$$\Delta P_0^{E} = 3b^2,\ \Delta P_0^{SE} = 3b^2 - 2a^2(b - 1)$$



y is always decremented in each step, i.e. $y_{k+1} = y_k - 1$.
$x_{k+1} = x_k$ if S is selected, or $x_{k+1} = x_k + 1$ if SE is selected.

$$\begin{aligned} P_k &= f(x_k+½, y_k-1) \\ &= b^2(x_k+½)^2 + a^2(y_k-1)^2 - a^2b^2 \\ &= b^2(x_k^2 + x_k + ¼) + a^2(y_k^2 - 2y_k + 1) - a^2b^2 \end{aligned}$$



If $P_k > 0$, select S :

$$
\begin{aligned}
P_{k+1}^{S} &= f(x_k+½, y_k-2) \\
&= b^2(x_k+½)^2 + a^2(y_k-2)^2 - a^2b^2 \\
&= b^2(x_k^2 + x_k + ¼) + a^2(y_k^2 - 4y_k + 4) - a^2b^2
\end{aligned}
$$

Change of P_k^S is: $\Delta P_k^S = P_{k+1}^S - P_k = a^2(3 - 2y_k)$

If Pk < 0, select SE :

$$
\begin{aligned}
P_{k+1}^{SE} &= f(x_k+3/2, y_k-2) \\
&= b^2(x_k+3/2)^2 + a^2(y_k-2)^2 - a^2b^2 \\
&= b^2(x_k^2 + 3x_k + 9/4) + a^2(y_k^2 - 4y_k + 4) - a^2b^2
\end{aligned}
$$

Change of P_k^{SE} is $\Delta P_k^{SE} = P_{k+1}^{SE} - P_k = 2b^2(x_k + 1) + a^2(3 - 2y_k)$

Calculate the changes of ΔP_k:

If S is selected,

$$\Delta P_{k+1}{}^{S} = a^2(5 - 2y_k)$$

$$\Delta^2 P_k{}^{S} = \Delta P_{k+1}{}^{S} - \Delta P_k{}^{S} = 2a^2$$

$$\Delta P_{k+1}{}^{SE} = 2b^2(x_k + 1) + a^2(5 - 2y_k)$$

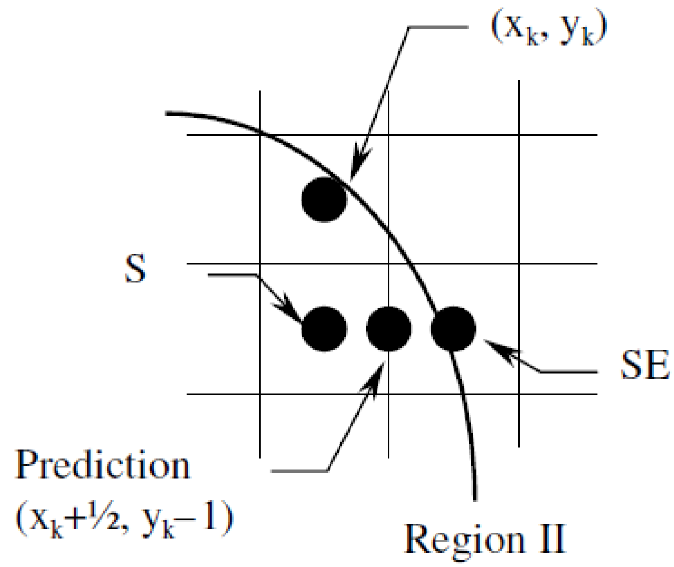
$$\Delta^2 P_k{}^{SE} = \Delta P_{k+1}{}^{SE} - \Delta P_k{}^{SE} = 2a^2$$

If SE is selected,

$$\Delta P_{k+1}{}^{S} = a^2(5 - 2y_k)$$

$$\Delta^2 P_k{}^{S} = \Delta P_{k+1}{}^{S} - \Delta P_k{}^{S} = 2a^2$$

$$\Delta P_{k+1}{}^{SE} = 2b^2(2x_k + 2) - a^2(5 - 2y_k)$$



Determine the boundary between region I and II:

Set $f(x, y) = 0$, $\dfrac{dy}{dx} = \dfrac{-bx}{a^2\sqrt{1 - x^2/a^2}}$.

When $dy/dx = -1$, $x = \dfrac{a^2}{\sqrt{a^2 + b^2}}$ and $y = \dfrac{b^2}{\sqrt{a^2 + b^2}}$.

At region I, $dy/dx > -1$, $x < \dfrac{a^2}{\sqrt{a^2 + b^2}}$ and $y > \dfrac{b^2}{\sqrt{a^2 + b^2}}$, therefore

$$\Delta P_k^{SE} < b^2\left(\frac{2a^2}{\sqrt{a^2 + b^2}} + 3\right) - 2a^2\left(\frac{b^2}{\sqrt{a^2 + b^2}} - 1\right) = 2a^2 + 3b^2.$$

Initial values at region II:

$$x_0 = \frac{a^2}{\sqrt{a^2 + b^2}} \text{ and } y_0 = \frac{b^2}{\sqrt{a^2 + b^2}}$$

- x_0 and y_0 will be the accumulative results from region I at the boundary.
- It is not necessary to calculate them from values of a and b.

$$P_0 = P_k^{I} - ¼[a^2(4y_0 - 3) + b^2(4x_0 + 3)]$$

where P_k^I is the accumulative result from region I at the boundary.

$$\Delta P_0^{E} = b^2(2x_0 + 3)$$
$$\Delta P_0^{SE} = 2a^2 + 3b^2$$

- The algorithm described above shows how to obtain the pixel coordinates in the first quarter only.
- The ellipse centre is assumed to be at the origin.
- In actual implementation, the pixel coordinates in other quarters can be simply obtained by use of the symmetric characteristics of an ellipse.
- For a pixel (x, y) in the first quarter, the corresponding pixels in other three quarters are (x, –y), (–x, y) and (–x, –y) respectively.
- If the centre is at (xC, yC), all calculated coordinate (x, y) should be adjusted by adding the offset (xC, yC). For easy implementation, a function PlotEllipse() is defined as follows:

```
PlotEllipse (xC, yC, x, y)
    putpixel(xC+x, yC+y)
    putpixel(xC+x, yC–y)
    putpixel(xC–x, yC+y)
    putpixel(xC–x, yC–y)
end PlotEllipse
```

The function to draw an ellipse is described in the following pseudo-codes:

DrawEllipse (x_C, y_C, a, b)

Declare integers x, y, P, ΔP^E, ΔP^S, ΔP^{SE}, $\Delta^2 P^E$, $\Delta^2 P^S$ and $\Delta^2 P^{SE}$

// Set initial values in region I

Set x = 0 and y = b

$P = b^2 + (a^2(1 - 4b) - 2)/4$ // Intentionally –2 to round off the value

$\Delta P^E = 3b^2$

$\Delta^2 P^E = 2b^2$

$\Delta P^{SE} = \Delta P^E - 2a^2(b - 1)$

$\Delta^2 P^{SE} = \Delta^2 P^E + 2a^2$

// Plot the pixels in region I

PlotEllipse(x_C, y_C, x, y)

while $\Delta P^{SE} < 2a^2 + 3b^2$
    if $P < 0$ then // Select E
        $P = P + \Delta P^{E}$
        $\Delta P^{E} = \Delta P^{E} + \Delta^2 P^{E}$
        $\Delta P^{SE} = \Delta P^{SE} + \Delta^2 P^{E}$
    else // Select SE
        $P = P + \Delta P^{SE}$
        $\Delta P^{E} = \Delta P^{E} + \Delta^2 P^{E}$
        $\Delta P^{SE} = \Delta P^{SE} + \Delta^2 P^{SE}$
        decrement y
    end if
    increment x
    PlotEllipse(x_C, y_C, x, y)
end while

// Set initial values in region II

$P = P - (a^2(4y - 3) + b^2(4x + 3) + 2) / 4$

// Intentionally +2 to round off the value

$\Delta P^{S} = a^2(3 - 2y)$

$\Delta P^{SE} = 2b^2 + 3a^2$

$\Delta^2 P^{S} = 2a^2$

```
// Plot the pixels in region II
while y > 0
        if P > 0 then          // Select S
                P = P + ΔP^E
                ΔP^E = ΔP^E + Δ^2P^S
                ΔP^SE = ΔP^SE + Δ^2P^S
        else              // Select SE
                P = P + ΔP^SE
                ΔP^E = ΔP^E + Δ^2P^S
                ΔP^SE = ΔP^SE + Δ^2P^SE
                increment x
        end if
        decrement y
        PlotEllipse(x_C, y_C, x, y)
end while
end DrawEllipse
```

Conic Sections

- In general, we can describe a conic section (or conic) with the second-degree equation:

$$Ax^2 + By^2 + Cxy + Dx + Ey + F = 0$$

- where values for parameters *A, B, C, D, E, and F determine the kind of curve we* are to display. Give11 this set of coefficients, we can determine the particular conic that will be generated by evaluating the discriminant B^2 *- 4AC:*

$$B^2 - 4AC \begin{cases} < 0, & \text{generates an ellipse (or circle)} \\ = 0, & \text{generates a parabola} \\ > 0, & \text{generates a hyperbola} \end{cases}$$

we get the circle equation *when*

$A = B = 1, C = 0, D = -2x_c, E = -2y_c,$

and $F = x_c^2 + y_c^2 - r^2.$

Polynomials and Spline Curves

A polynomial function of nth degree in *x is defined as*

$$y = \sum_{k=0}^{n} a_k x^k$$

$$= a_0 + a_1 x + \cdot \cdot \cdot + a_{n-1} x^{n-1} + a_n x^n$$

- where n is a nonnegative integer and the a, are constants, with $a_n \neq 0$. *We get a* quadratic
- when *n = 2; a cubic polynomial*
- *when n = 3; a quartic*
- *when n = 4; and* so forth.
- And we have a straight line when n = 1.
- Polynomials are useful in a number of graphics applications, including the design of object shapes, the specification of animation paths, and the graphing of data trends in a discrete set of data points.

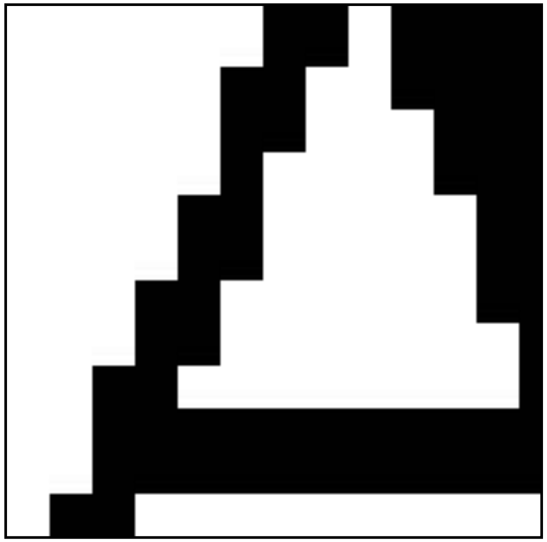
Anti-aliasing

Anti-aliasing is a technique used to diminish the jagged edges of an image or a line, so that the line appears to be smoother; by changing the pixels around the edges to intermediate colors or gray scales.

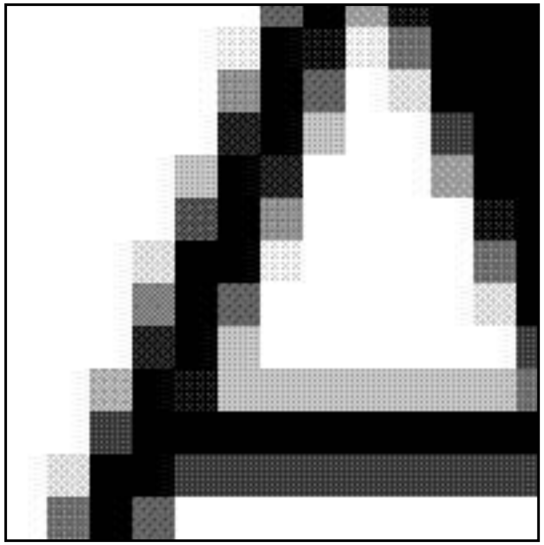
E.g. Anti-aliasing disabled:

E.g. Anti-aliasing enabled:

Anti-aliasing (OpenGL)



Anti-aliasing disabled



Anti-aliasing enabled

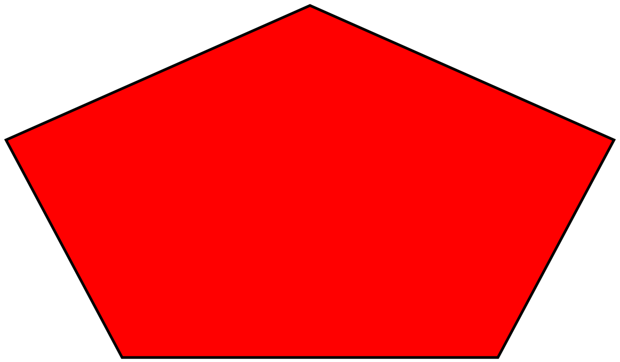
Setting anti-aliasing option for lines:

glEnable (GL_LINE_SMOOTH);

Fill Area Algorithms

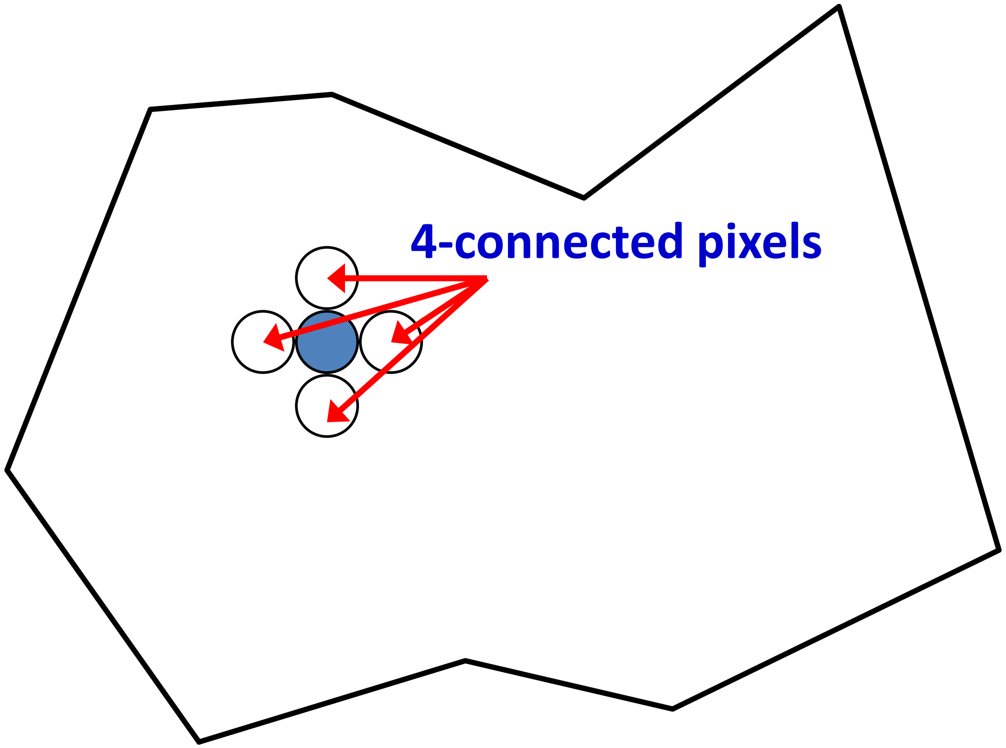
Fill Area Algorithms

- Fill-Area algorithms are used to fill the interior of a polygonal shape.
- Many algorithms perform fill operations by first identifying the interior points, given the polygon boundary.



Basic Filling Algorithm

The basic filling algorithm is commonly used in interactive graphics packages, where the user specifies an interior point of the region to be filled.



Basic Filling Algorithm

[1] Set the user specified point.

[2] Store the four neighboring pixels in a stack.

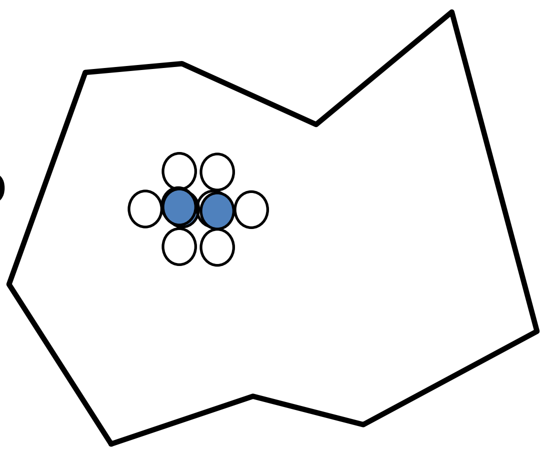
[3] Remove a pixel from the stack.

[4] If the pixel is not set,

- Set the pixel
- Push its four neighboring pixels into the stack

[5] Go to step 3

[6] Repeat till the stack is empty.

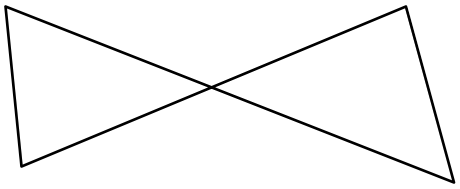


Basic Filling Algorithm (Code)

```
void fill(int x, int y) {
  if(getPixel(x,y)==0){
     setPixel(x,y);
     fill(x+1,y);
     fill(x-1,y);
     fill(x,y+1);
     fill(x,y-1);
  }
}
```

Basic Filling Algorithm: Conditions

- Requires an interior point.
- Involves considerable amount of stack operations.
- The boundary has to be closed.
- Not suitable for self-intersecting polygons



Types of Basic Filling Algorithms

- Boundary Fill Algorithm
  - For filling a region with a single boundary color.
  - Condition for setting pixels:
    - Color is not the same as border color
    - Color is not the same as fill color

- Flood Fill Algorithm
  - For filling a region with multiple boundary colors.
  - Condition for setting pixels:
    - Color is same as the old interior color

Boundary Fill Algorithm (Code)

```
void boundaryFill(int x, int y,
            int fillColor, int borderColor)
{
  getPixel(x, y, color);
  if ((color != borderColor)
            && (color != fillColor)) {
     setPixel(x,y);
     boundaryFill(x+1,y,fillColor,borderColor);
     boundaryFill(x-1,y,fillColor,borderColor);
     boundaryFill(x,y+1,fillColor,borderColor);
     boundaryFill(x,y-1,fillColor,borderColor);
  }
}
```

Flood Fill Algorithm (Code)

```
void floodFill(int x, int y,
               int fillColor, int oldColor)
{
  getPixel(x, y, color);
  if (color == oldColor)
  {
     setPixel(x,y);
     floodFill(x+1, y, fillColor, oldColor);
     floodFill(x-1, y, fillColor, oldColor);
     floodFill(x, y+1, fillColor, oldColor);
     floodFill(x, y-1, fillColor, oldColor);
  }
}
```