COMPUTER GRAPHICS

CH2 – Simple Drawing Algorithms

Outline

- Introduction – Output Primitives
- Line Drawing and Attributes
- Polygon Drawing
- Circle Drawing
- Ellipse Drawing
- Character Generation and Attributes

Introduction – Output Primitives

- **Shapes and colors** of objects can be described internally with sets of ***basic geometric structures*** such as straight line segments and polygon color areas.
- A scene can be **displayed** by loading pixel arrays into the ***frame buffer*** or by ***scan converting*** basic geometric structure specifications into ***pixel patterns***.
- Typically, **graphics programming packages** provide functions
  - To describe a scene/picture/model in terms of these ***geometric structures***, referred to as ***output primitives***, and
  - To group sets of ***output primitives*** into more complex structures.

Introduction – Output Primitives

- **Output Primitives** are basic geometric structures used to describe scenes.
  - They can also be grouped into **more complex structures.**
  - Each one is specified with input coordinate data and other information about the way that object is to be displayed.
  - Examples of **output primitives** can include:
    - Points,
    - Straight line segment,
    - Circles and other conic sections,
    - Quadric surfaces,
    - Spline curve and surfaces,
    - Polygon color areas, and
    - Character strings
  - These **picture components** are often defined in a ***continuous space***.

Cont….

- A **polygon** in continuous space is **a series of line segments joined at their endpoints,** thus conversion of a polygon from continuous to **discrete space** is completely determined by the method applied to convert **the continuous line segments to line segments** composed of **pixels in discrete space.**

Introduction – Output Primitives

- **In digital representation:**
  - Display screen is divided into ***scan lines*** and ***columns***.
  - ***Pixels positions*** are referenced according to **scan line number** and **column number** (columns across scan lines).
  - **Scan lines start** from **0 at screen bottom**, and **columns start from 0** at the screen left side.
- **Screen locations (or pixels)** are referenced with ***integer values***.
  - In order to draw the primitive objects, one has to **first scan convert** the object.
- **Scan conversion** refers to the operation of
  - finding out the location of pixels to be intensified and then setting the values of **corresponding bits, in the graphic memory**, to the desired intensity code.
- The ***frame buffer*** stores the intensities temporarily, and the video controller reads from the frame buffer and plots the screen pixels.

Introduction – Output Primitives

Picture or scene descriptions

- In **Raster display:**
  - Is completely specified by the set of intensities for the pixels positions in the display.
  - Shapes and colors are described with pixel arrays.
  - The picture or scene is displayed by loading ***pixels array*** into the ***frame buffer***.
- In **Vector Display:**
  - Picture or scene description is set of complex objects positioned at specified coordinate locations within the **scene**.
  - **Shapes and colors** are described with sets of basic ***geometric structures***.
  - The picture or scene is displayed by ***scan converting*** the **geometric-structure specifications** into **pixel patterns**.

Line Drawing and Attributes

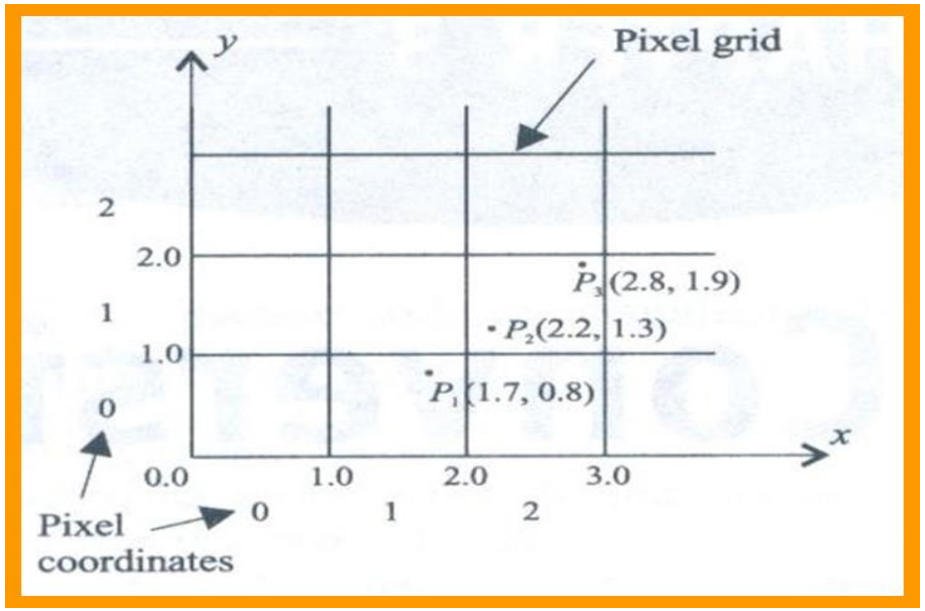
Point Drawing

- **Point drawing** is accomplished by **converting** a single coordinate position furnished by an application program into appropriate operation for the **output device** in use.
- In **raster-scan system:**
  - **Black-white:** setting the bit value corresponding to a specified screen position within the frame buffer to 1.
  - **RGB:** loading the frame buffer with the color codes for the **intensities** that are to be displayed at the screen pixel positions.
- In **random-scan (vector) system:**
  - Stores **point-plotting instructions** in the display list, and coordinate values in these instructions are converted to deflection voltages that position the electron beam at the screen locations to be plotted during each refresh cycle.

Line Drawing and Attributes

Point Drawing

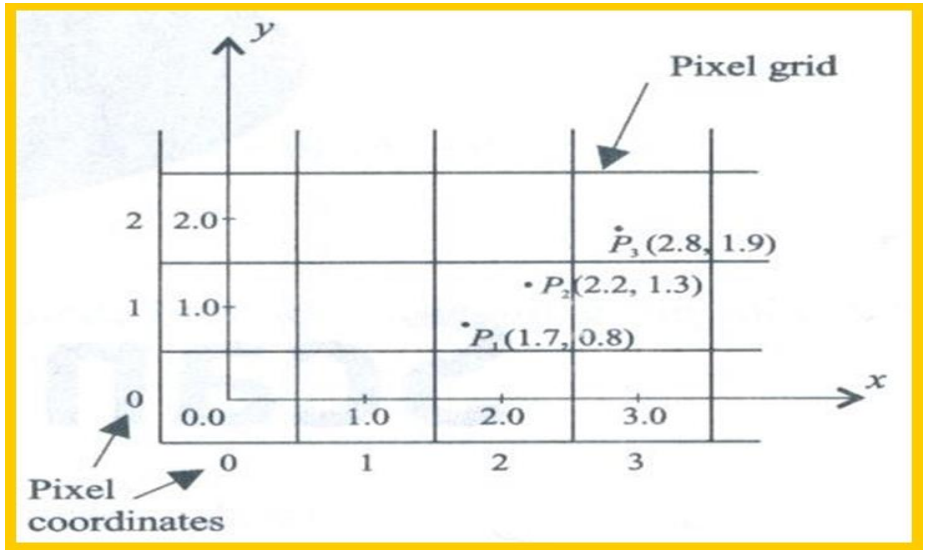
- A mathematical point (x', y') needs to be scan converted to a pixel at location (x, y): $x = Round(x')$ and $y = Round(y')$.
  - All points that satisfy $(x \leq x' < x + 1)$ and $(y \leq y' < y + 1)$ are mapped to pixel (x, y).



Line Drawing and Attributes

Point Drawing

- **A better way:** A mathematical point (x', y') needs to be **scan converted** to a pixel at location (x, y): $x = Round(x' + 0.5)$ and $y = Round(y' + 0.5)$.
  - All points that satisfy $(x - 0.5 \leq x' < x + 0.5)$ and $(y - 0.5 \leq y' < y + 0.5)$ are mapped to pixel (x, y).



Line Drawing and Attributes

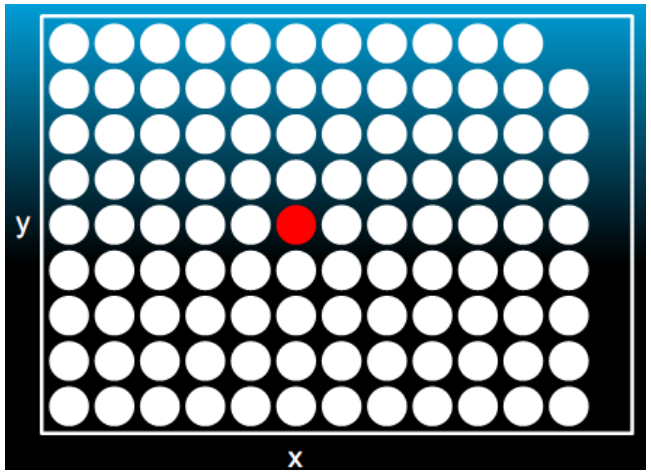
Point Drawing

- Pixel positions are referenced by ***scan-line number*** and ***column number.***
  - To load a specified color into the frame buffer at a position corresponding to ***column x*** along ***scan line y,*** we will assume we have available a low-level procedure of the form

$$putPixel(x, y, COLOR)$$

  - We also want to be able to retrieve the current frame-buffer intensity setting for a specified **location** using the **low-level** function

$$getPixel(x, y)$$



Line Drawing

- **Line drawing** Is accomplished by **calculating** intermediate positions along the line path between two specified ***endpoint positions***.
  - An **output device** is then directed to fill in these positions between the **endpoints.**
- **Digital devices display** a straight line segment by **plotting discrete** points between the two endpoints.
  - Discrete coordinate positions along the line path are **calculated** from the **equation** of the **line.**
  - For a raster video display, the line color (**intensity**) is then loaded into the **frame buffer** at the **corresponding pixel coordinates.**
  - **Reading** from the frame buffer, the video controller then "**plots**" the screen **pixels.**
  - Screen locations are referenced with **integer values,** so plotted positions may only approximate **actual line positions** between two specified endpoints.

Line Drawing

- For example:
  - A computed line position of **(10.48, 20.51)**, would be **converted to pixel position (10, 21).**
  - Thus rounding of coordinate values to integers causes lines to be displayed with aliasing effect or stair-case appearance ("the jaggies").
  - These stair-case characteristic shape of raster lines is particularly noticeable on systems with **low resolution**, and we can improve their appearance somewhat by displaying them on **high resolution** systems.
  - More effective techniques for **smoothing** raster lines are based on **adjusting pixel intensities** along the line paths (**anti-aliasing**).

Line Drawing Algorithms

- The Cartesian ***slope-intercept equation*** for a straight line with $\boldsymbol{m}$ representing the slope of the line and $\boldsymbol{b}$ as the y-intercept is:

$$\boldsymbol{y = m \cdot x + b} \qquad \boldsymbol{(2.1)}$$

- Given that the two endpoints of a line segment are specified at positions (x_1, y_1) and (x_2, y_2), we can determine values for the slope m and y-intercept b with the following calculations:

$$\boldsymbol{m = \frac{y_2 - y_1}{x_2 - x_1}} \qquad \boldsymbol{(2.2)}$$

$$\boldsymbol{b = y_1 - m x_1} \qquad \boldsymbol{(2.3)}$$

- Algorithms for displaying straight lines are based on the line equation (2.1) and the calculations given in equations (2.2) and (2.3).
- For any given x interval Δx along a line, we can compute the corresponding y interval Δy from equation (2.2) as:

$$\boldsymbol{\Delta y = m \Delta x} \qquad \boldsymbol{(2.4)}$$

- Similarly, we can obtain the x interval Δx corresponding to a specified Δy as:

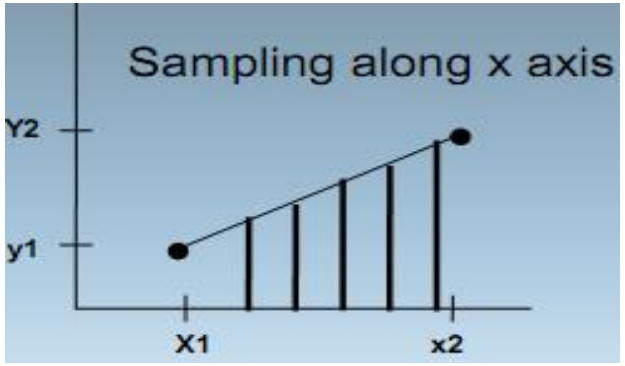
$$\boldsymbol{\Delta x = \frac{\Delta y}{m}} \qquad \boldsymbol{(2.5)}$$

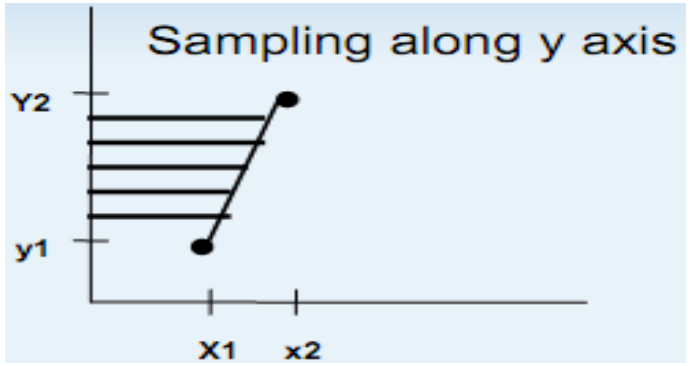
Line Drawing Algorithms

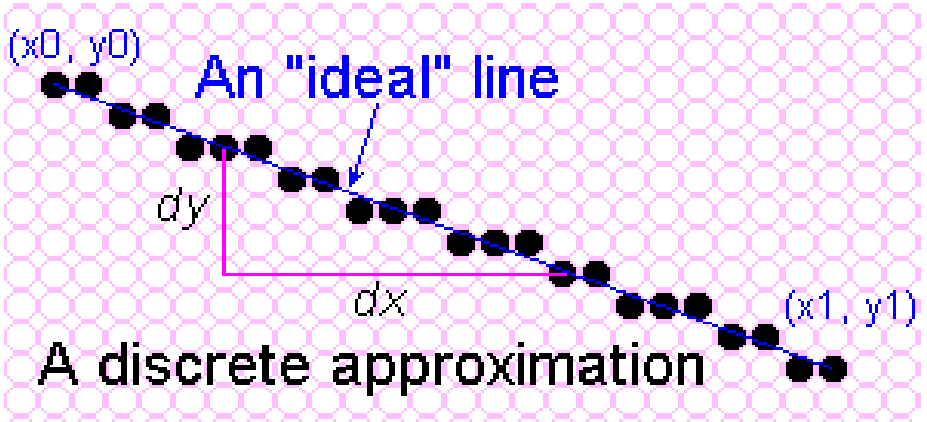
- These equations form the basis for determining **deflection voltages** in analog devices.
- For lines with slope magnitudes $|\boldsymbol{m}| < \mathbf{1}$,
  - $\boldsymbol{\Delta x}$ can be set proportional to small horizontal deflection voltage and the corresponding vertical deflection is then set proportional to Δy as calculated from equation (2.4).
- For lines whose slopes have magnitudes $|m| > 1$,
  - Δy can be set proportional to a small vertical deflection voltage with the corresponding horizontal deflection voltage set proportional to Δx, calculated from equation (2.5).
- For lines with $m = 1$,
  - $\Delta x = \Delta y$ and the **horizontal** and **vertical** deflections voltages are **equal**.
- In each case, a **smooth line** with **slope $\boldsymbol{m}$** is generated between the specified **endpoints**.

Line Drawing Algorithms

- On ***raster systems***, lines are plotted with ***pixels***, and **step sizes** in the horizontal and vertical directions are constrained by ***pixel separations***,
  - i.e., we must "**sample**" a line at discrete positions and determine the ***nearest pixel*** to the line at each **sampled position**.
  - ***Scan conversion*** process ***samples*** a line at discrete positions and determine the nearest pixel to the line at each **sampled position**.







Line Drawing Algorithms

Digital Differential Analyzer (DDA) Algorithm

- The **DDA** algorithm is a scan-conversion algorithm based on calculating either Δy or Δx using equation (2.4) or equation (2.5).
- We sample the line at unit intervals in one coordinate and determine corresponding integer values nearest the line path for the other **coordinate**.
- Let us consider first a line with **positive slope**.
  - If the slope is **less than or equal to 1**, we sample at unit x intervals $(\Delta x = 1)$ and compute each successive y values as:

$$\boldsymbol{y_{k+1} = y_k + m} \qquad \boldsymbol{(2.6)}$$

  - Subscript k takes integer values starting from 1, for the first point, and increases by 1 until the final endpoint is reached.
  - Since $\boldsymbol{m}$ can be any real number between 0 and 1, the calculated y values must be rounded to the nearest integer.
  - For lines with a **positive slope greater** than 1, we reverse the roles of x and y, i.e., we sample at unit y intervals $(\Delta y = 1)$ and calculate each succeeding x value as:

$$\boldsymbol{x_{k+1} = x_k + \frac{1}{m}} \qquad \boldsymbol{(2.7)}$$

Digital Differential Analyzer (DDA) Algorithm

- Equations (2.6) and (2.7) are based on the **assumption** that lines are to be processed from the left endpoint to the right endpoint.
- If this **processing is reversed**, so that the **starting endpoint** is at the right, then either we have $\Delta x = -1$ and

$$y_{k+1} = y_k - m \qquad (2.8)$$

or (when the slope is greater than 1) we have $\Delta y = -1$ with

$$x_{k+1} = x_k - \frac{1}{m} \qquad (2.9)$$

Digital Differential Analyzer (DDA) Algorithm

- Equations (2.6) through (2.9) can also be used to calculate pixel positions along a line with **negative slope.**
- If $|m| \leq 1$ and the start endpoint is at the left,
  - We set $\Delta x = 1$ and calculate y values with equations (2.6).

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

  - When the start endpoint is at the right (for the same slope), we set $\Delta x = -1$ and obtain y positions from equation (2.8).

$$\boldsymbol{y_{k+1} = y_k - m}$$

- Similarly, when $|m| > 1$,
  - We use $\Delta y = -1$ and equation (2.9),

$$\boldsymbol{x_{k+1} = x_k - \frac{1}{m}}$$

  - Or we use $\Delta y = 1$ and equation (2.7),

$$\boldsymbol{x_{k+1} = x_k + \frac{1}{m}}$$

Digital Differential Analyzer (DDA) Algorithm

- The DDA algorithm is summarized in the following procedure, which accepts as input the two endpoint **pixel positions** (x_a, y_a) and (x_b, y_b).
  - Horizontal and vertical differences between the endpoint positions are assigned to parameters $\boldsymbol{dx}$ and $\boldsymbol{dy}$.
    - The difference with the **greater magnitude determines** the value of parameter ***steps***.
  - Starting with pixel position (x_a, y_a), we determine the offset needed at each step to generate the **next pixel position along** the line path.
  - We loop through this process **steps** times.
    - If the $|\boldsymbol{dx}|$ is **greater than** $|\boldsymbol{dy}|$ and $\boldsymbol{x_a}$ **is less than** $\boldsymbol{x_b}$, the values of the **increments** in the $\boldsymbol{x}$ **and** $\boldsymbol{y}$ **directions** are $\boldsymbol{1}$ **and** $\boldsymbol{m}$, respectively.
    - If the **greater change** is in the $\boldsymbol{x}$ **direction, but** $\boldsymbol{x_a}$ **is greater than** $\boldsymbol{x_b}$, then the decrements -1 and $-m$ are used to generate each new point on the line.
    - Otherwise, we use a unit increment (or decrement) in the $\boldsymbol{y}$ **direction and an** $\boldsymbol{x}$ **increment** (or decrement) of $\frac{\boldsymbol{1}}{\boldsymbol{m}}$.

DDA Algorithm


```
#include "device.h"
#define ROUND(a)    ((int) (a + 0.5))
void lineDDA (int xa, int ya, int xb, int yb)
{
      int dx = xb - xa, dy = yb - ya, steps, k;
      float xIncrement, yIncrement, x = xa, y = ya;
      if (abs(dx) > abs(dy))
            steps = abs(dx ) ;
      else
            steps = abs(dy);
      xIncrement = dx / (float)steps;
      yIncrement = dy / (float)steps
      setpixel(ROUND(x), ROUND(y));
      for(k = O; k < steps; k++){
            x += xIncrment;
            y += yIncrement;
            setpixel(ROUND(x), ROVND(y))
       }
}
```

(x_1, y_1) (x_2, y_2)

$(2, 2)$ $(9, 2)$

$\Delta x = 9 - 2 = 7$

$\Delta y = 2 - 2 = 0$

$m = \frac{\Delta y}{\Delta x} = \frac{0}{7} = 0$ steps $= 7$

$x_{inc} = \frac{7}{7} = 1$ $y_{inc} = \frac{0}{7} = 0$

x	y
2	2
3	2
4	2
5	2
6	2
7	2
8	2
9	2

DDA Line Drawing Algorithm

(x_1, y_1) (x_2, y_2)

$(2, 5)$ $(2, 12)$

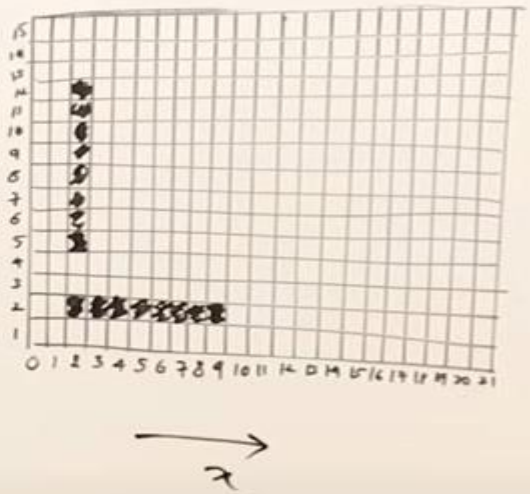
$\Delta x = 2 - 2 = 0$

$\Delta y = 12 - 5 = 7$

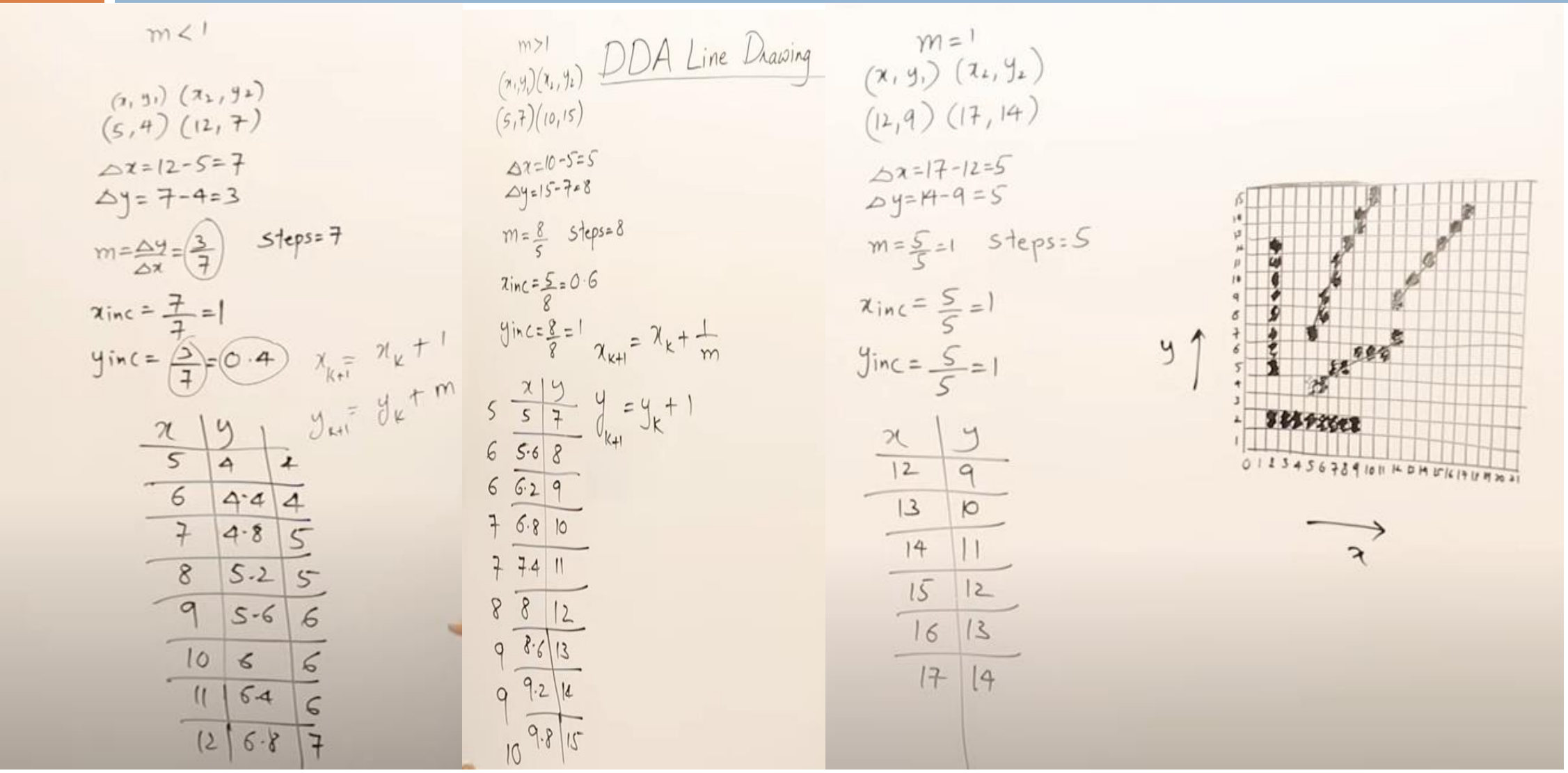
$m = \frac{\Delta y}{\Delta x} = \frac{7}{0} = \infty$ steps $= 7$

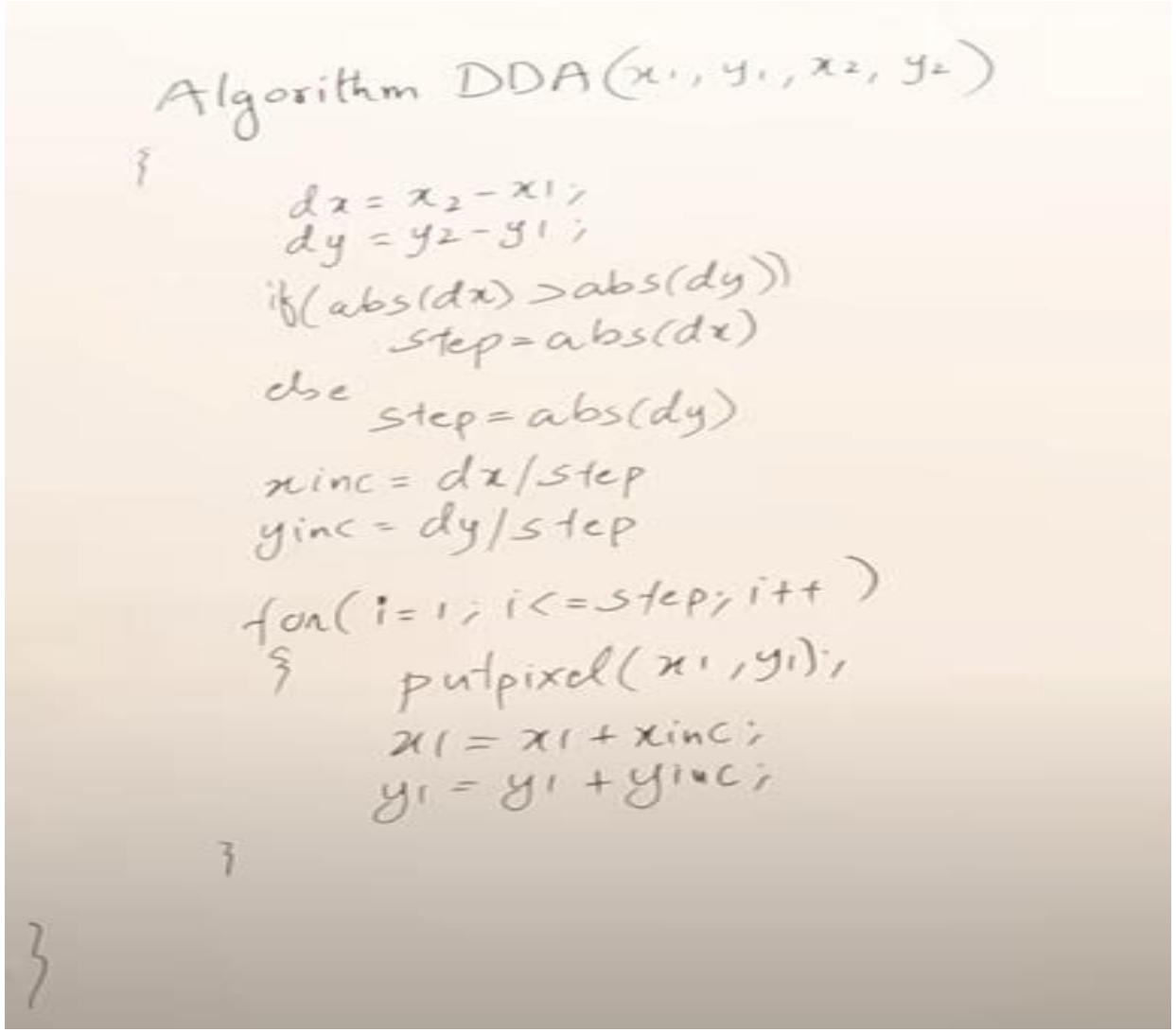
$x_{inc} = \frac{0}{7} = 0$ $y_{inc} = \frac{7}{7} = 1$

x	y
2	5
2	6
2	7
2	8
2	9
2	10
2	11
2	12



DDA Line Drawing Algorithm





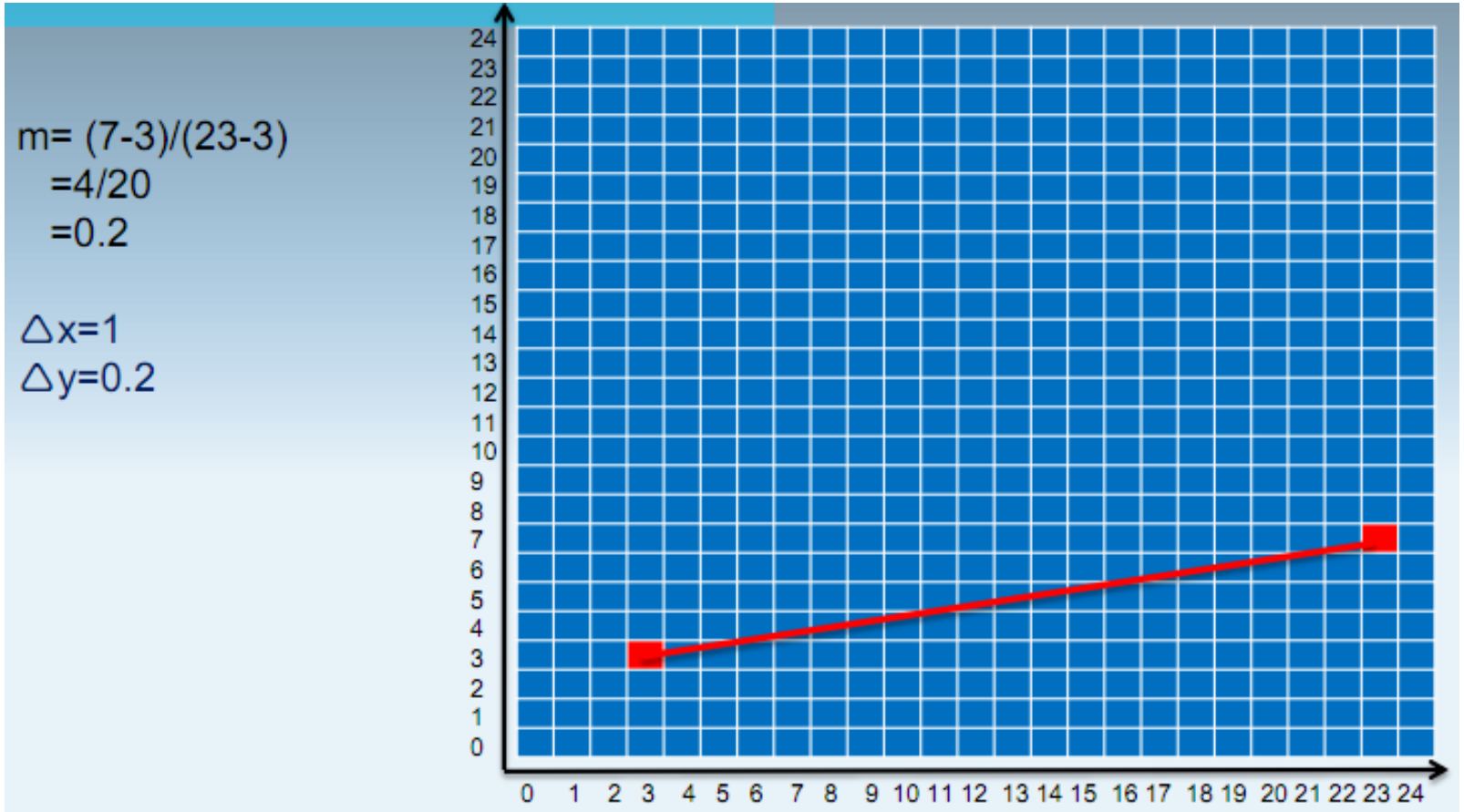
```
Algorithm DDA(x1, y1, x2, y2)
{
    dx = x2 - x1;
    dy = y2 - y1;
    if(abs(dx) > abs(dy))
        step = abs(dx)
    else
        step = abs(dy)
    xinc = dx/step
    yinc = dy/step
    for(i = 1; i <= step; i++ )
    {
        putpixel(x1, y1);
        x1 = x1 + xinc;
        y1 = y1 + yinc;
    }
}
```

Digital Differential Analyzer (DDA) Algorithm

Example 1: Scan convert a *line having end points* $(3,2)$ and $(4,7)$ using the **DDA algorithm?**

Digital Differential Analyzer (DDA) Algorithm

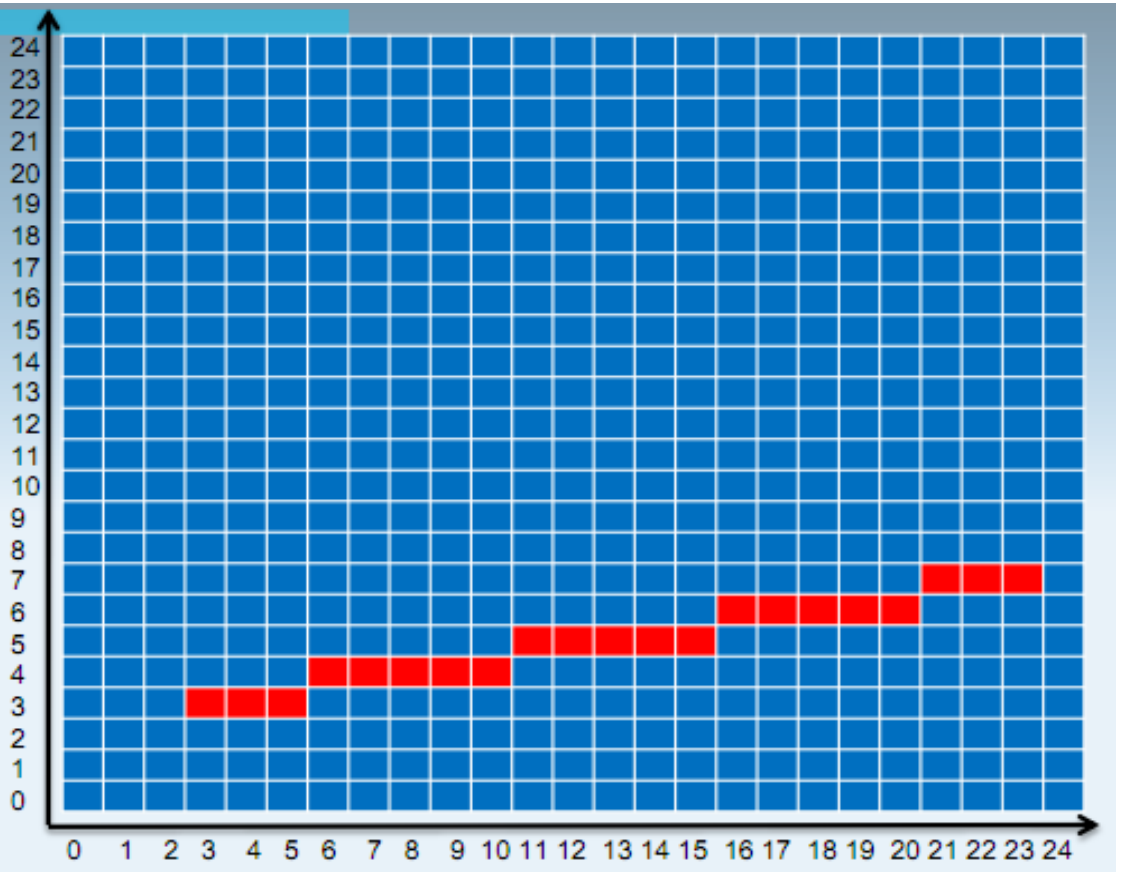
Example 2: Describe the line segment which starts at $(3,3)$ and ends at $(23,7)$.



Digital Differential Analyzer (DDA) Algorithm

Example 1: Describe the line segment which starts at $(3,3)$ and ends at $(23,7)$.

x	y	actual point	pixel position
3	3	(3,3)	(3,3)
4	3.2	(4,3.2)	(4,3)
5	3.4	(5,3.4)	(5,3)
6	3.6	(6,3.6)	(6,4)
7	3.8	(7,3.8)	(7,4)
8	4	(8,4)	(8,4)
9	4.2	(9,4.2)	(9,4)
10	4.4	(10,4.4)	(10,4)
11	4.6	(11,4.6)	(11,5)
12	4.8	(12,4.8)	(12,5)
13	5	(13,5)	(13,5)
14	5.2	(14,5.2)	(14,5)
15	5.4	(15,5.4)	(15,5)
16	5.6	(16,5.6)	(16,6)
17	5.8	(17,5.8)	(17,6)
..	..	..	..
..	..	..	..
22	6.8	(22,6.8)	(22,7)
23	7	(23,7)	(23,7)



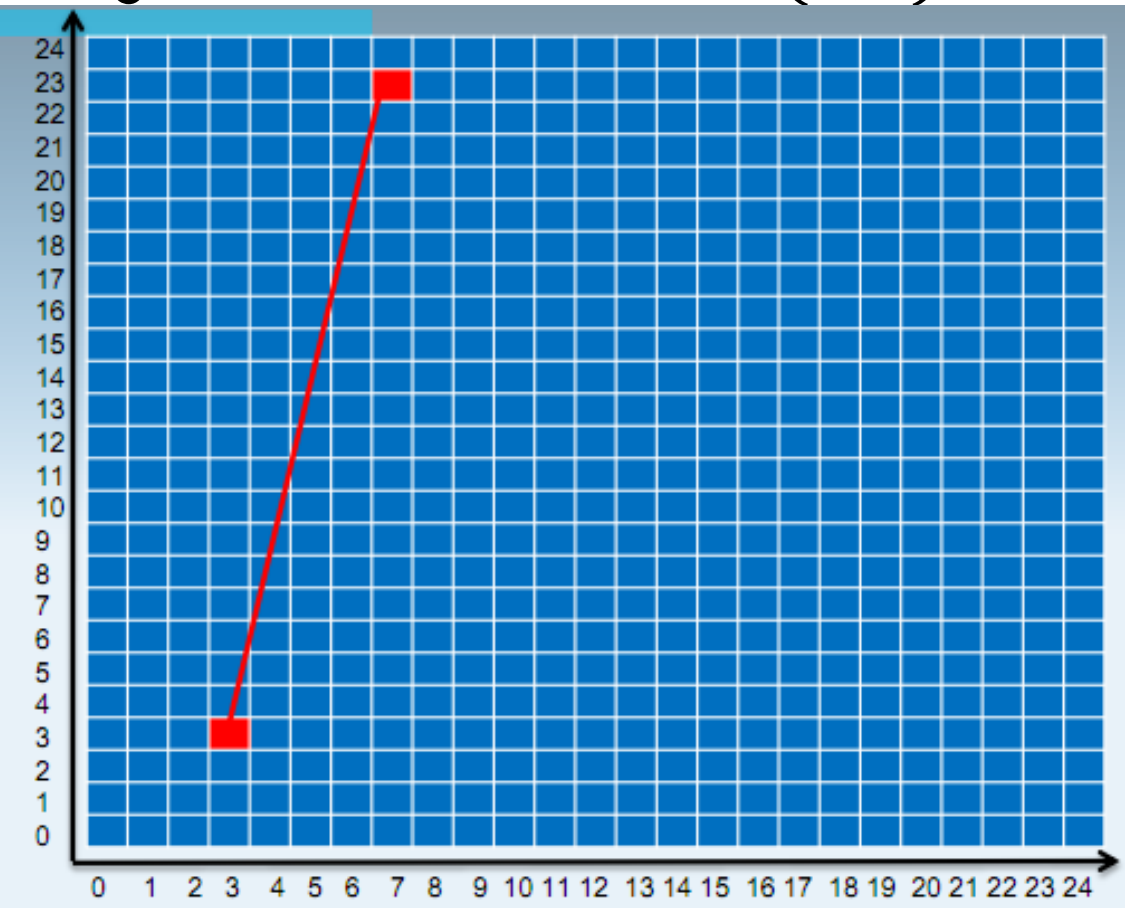
Digital Differential Analyzer (DDA) Algorithm

Example 3: Describe the line segment which starts at $(3,3)$ and ends at $(7,23)$.

m= (23-3)/(7-3)
= 20/4
= 5

△x =1/m
=1/5
=0.2

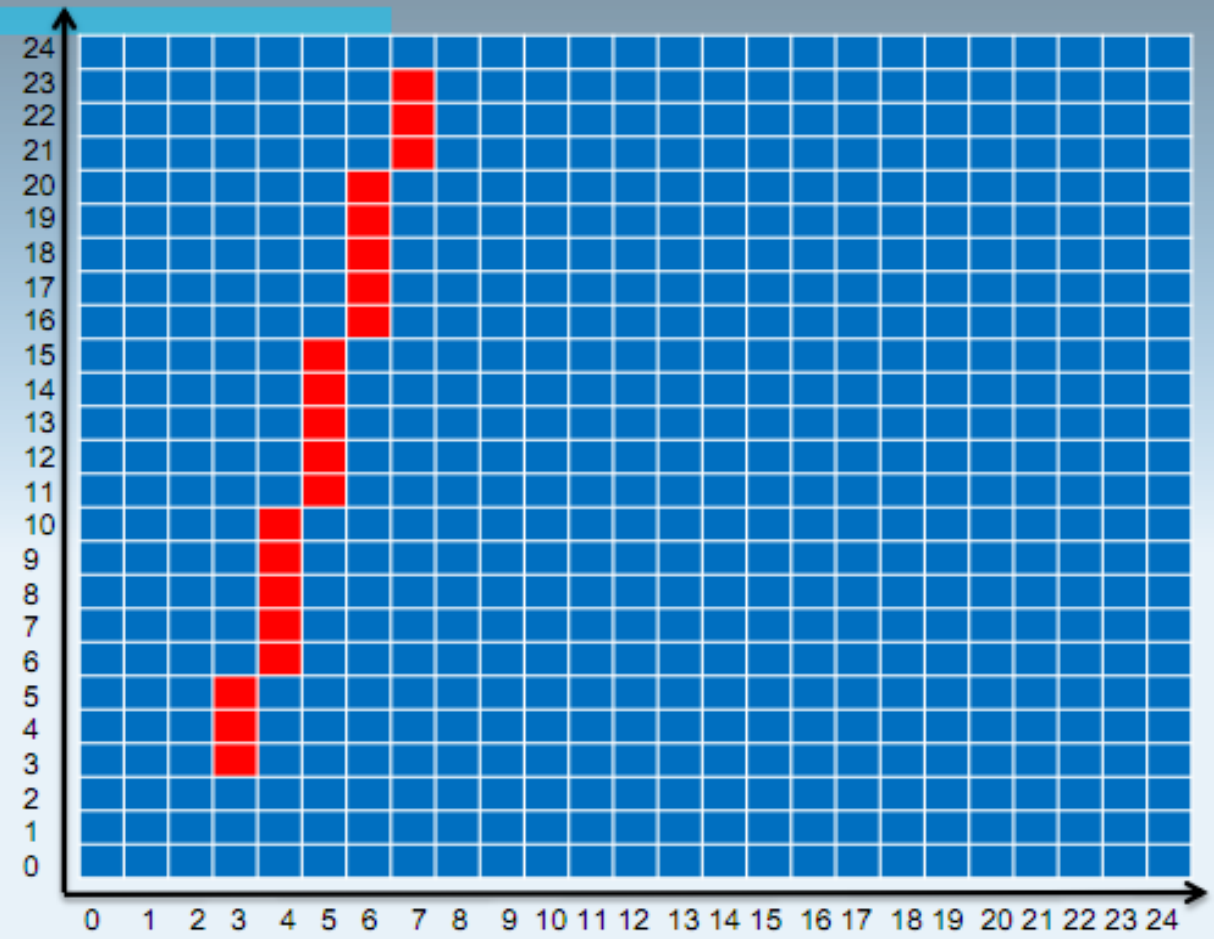
△y =1



Digital Differential Analyzer (DDA) Algorithm

Example 2: Describe the line segment which starts at $(3,3)$ and ends at $(7,23)$.

y	x	actual point	pixel position
3	3	(3,3)	(3,3)
4	3.2	(3.2,4)	(3,4)
5	3.4	(3.4,5)	(3,5)
6	3.6	(3.6,6)	(4,6)
7	3.8	(3.8,7)	(4,7)
8	4	(4,8)	(4,8)
9	4.2	(4.2,9)	(4,9)
10	4.4	(4.4,10)	(4,10)
11	4.6	(4.6,11)	(5,11)
12	4.8	(4.8,12)	(5,12)
13	5	(5,13)	(5,13)
14	5.2	(5.2,14)	(5,14)
15	5.4	(5.4,15)	(5,15)
16	5.6	(5.6,16)	(6,16)
17	5.8	(5.8,17)	(6,17)
..	..	..	..
..	..	..	..
22	6.8	(6.8,22)	(7,22)
23	7	(7,23)	(7,23)



Exercise

Use DDA Algorithm to draw a line from (2,3) to (9,8).

Digital Differential Analyzer (DDA) Algorithm

- The **DDA algorithm** is a faster method for **calculating pixel positions** than the **direct use** of equation (2.1).
  - It **eliminates the multiplication** in equation (2.1) by making use of **raster characteristics**, so that **appropriate increments** are applied in the x **or** y direction to step to pixel positions along the **line path**.
- The accumulation of ***round off error in successive*** additions of the **floating-point increment**, however, can cause the ***calculated pixel positions to drift*** away from the true line path for *long line segments*.
- Furthermore, the **rounding operations** and **floating-point arithmetic** in procedure **lineDDA** are still **time-consuming**.
  - We can improve the performance of the DDA algorithm by separating the increments m and $\frac{1}{m}$ into integer and fractional parts so that all calculations are reduced to integer operations.

Bresenham's Line Drawing Algorithm

- An **accurate** and **efficient raster** line-generating algorithm is developed by Jack Elton Bresenham, in 1962 at IBM.
  - This algorithms **scan converts lines** using only ***incremental integer calculations*** that can be adapted to **display circles** and other **curves.**

Working of Bresenham's Algorithm:

- Let us say we want to **scan convert** a line with a positive slope less than 1 ($0 < m < 1$, i.e., $\Delta y < \Delta x$.
  - We start with pixel $P_1'(x_1', y_1')$, then select subsequent pixels in the horizontal direction towards $P_2'(x_2', y_2')$.
  - Once a **pixel** is chosen at any step, the **next pixel** is either the **one** to its right or the one to its **right and up** due to the **limit** on m.
  - The line is best approximated by those pixels that fall the **least distance** from its **true path** between P_1' and P_2'.

Bresenham's Line Drawing Algorithm

Working of Bresenham's Algorithm:

- The coordinates of the **last chosen pixel** upon entering ***step i*** are (xi, y_i).
- The task is to choose the **next one between** the bottom pixel **S** and top pixel **T**.
  - If **S** is chosen, we have: $x_{i+1} = x_i + 1$ and $y_{i+1} = y_i$
  - If **T** is chosen, we have: $x_{i+1} = x_i + 1$ and $y_{i+1} = y_i + 1$
- The actual y coordinate of line at $x = x_{i+1}$ is:

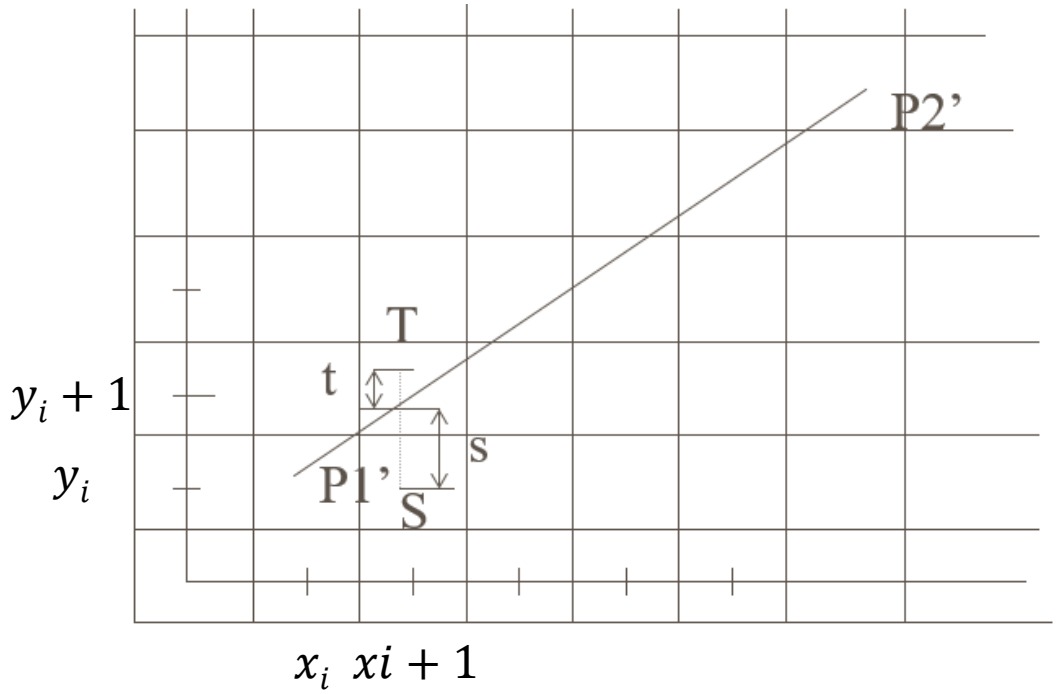
$$y = mx + b = m\,x_{i+1} + b$$
$$= m(xi + 1) + b \quad \ldots\ldots\ldots\ldots \text{(eq. 1)}$$

- The distance from **S** to actual line in y direction is:

$$s = y - y_i$$

- The distance from **T** to actual line in y direction is:

$$t = (y_{i+1}) - y$$



Bresenham's Line Algorithm

- It determines the points of an n-dimensional raster that should be selected in order to form a close approximation to a straight line between two points.

- Efficient=>B/c it only integer add, multiplication & subtraction
- The operations performed rapidly so line can be generated quickly.
- Bresenham's line drawing algorithm is an accurate and efficient line drawing algorithm.
- Bresenham's line drawing algorithm converts line only using incremental integer calculations.

Algorithm: - start coordinate (x_0, y_0)
End coordinate (x_n,Y_n)

Step 1: Calculate $\Delta x \; and \; \Delta y$

- $\Delta x = xn - x0$
- $\Delta y = yn - y0$

Step 2: Calculate decision parameter –it is used to find exact point to draw line

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

Step 3: suppose current point (x_k , y_k), next point ((x_{k+1},y_{k+1}), find next point depending on value of decision parameter $Pk = 2\Delta y - \Delta x$

Here we use two case

Case 1

$if \;\; pk < 0$

- Next pk and points
- Pk+1 = pk + $2\Delta y$
- Xk+1 = xk+1
- Yk+1= yk

Case 2

Pk >= 0

- Pk+1 =pk + $2\Delta y$- $2\Delta x$
- Xk+1 = xk+1
- yk+1 = yk+1
- Step 4: repeat step 3 until end point is reached

Example

Plot a line using Bresenham's Line drawing algorithm for the endpoints (19,38) and (28,45).

Solution:

Step 1: Calculate slope: $|m| = \Delta y/\Delta x$

$|m| = (45-38)/(28-19) = 7/9 < 1$

Step 2: Plot Left Endpoint as first endpoint (x0,y0).

Step 3: Calculate 4 constants:

$\Delta x = 9$

$\Delta y = 7$

$2\Delta y = 14$

$2\Delta y - 2\Delta X = 14-18 = -4$

Step 4: Calculate initial decision parameter $P_0 = 2\Delta y - \Delta X = 14-9 = 5$

Since P0 > 0,

Next point to plot = $(x_k+1, y_k+1) = (20,39)$

	P_k	x	y
		19	38
P_0	5	20	39
P_1			
P_2			
P_3			
P_4			
P_5			
P_6			
P_7			
P_8			
P_9			

Step 5: Repeat Step 4 for the next decision parameter

Since P0 > 0,

Next point to plot = $(x_k+1, y_k+1) = (20,39)$

& $P_1 = P_0 + (2\Delta y - 2\Delta X) = 5 + (-4) = 1$

Since P1 > 0,

Next point to plot = $(x_k+1, y_k+1) = (21,40)$

& $P_2 = P_1 + (2\Delta y - 2\Delta X) = 1 + (-4) = -3$

Since P2 < 0,

Next point to plot = $(x_k+1, y_k) = (22,40)$

& $P_3 = P_2 + (2\Delta y) = -3 + 14 = 11$

Since P3 > 0,

Next point to plot = $(x_k+1, y_k+1) = (23,41)$

& $P_4 = P_3 + (2\Delta y - 2\Delta X) = 11 + (-4) = 7$

Since P4 > 0,

Next point to plot = $(x_k+1, y_k+1) = (24,42)$

& $P_5 = P_4 + (2\Delta y - 2\Delta X) = 7 + (-4) = 3$

	P_k	x	y
		19	38
P_0	5	20	39
P_1	1	21	40
P_2	-3	22	40
P_3	11	23	41
P_4	7	24	42
P_5	3		
P_6			
P_7			
P_8			
P_9			

Since P5 > 0,

Next point to plot = $(x_k+1, y_k+1) = (25,43)$

& $P_6 = P_5 + (2\Delta y - 2\Delta X) = 3 + (-4) = -1$

Since P6 < 0,

Next point to plot = $(x_k+1, y_k) = (26,43)$

& $P_7 = P_1 + (2\Delta y) = -1 + (14) = 13$

Since P7 > 0,

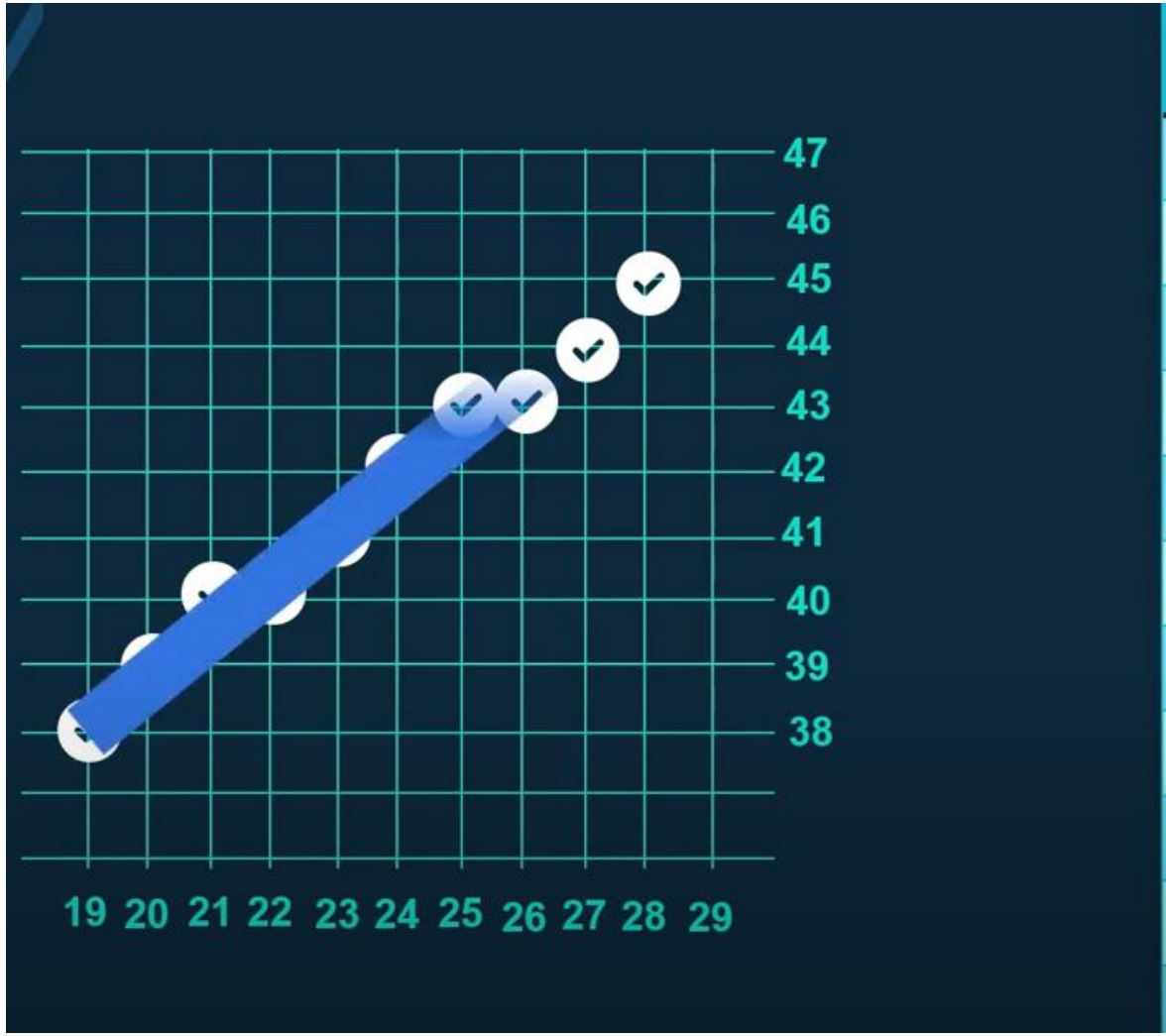
Next point to plot = $(x_k+1, y_k+1) = (27,44)$

& $P_8 = P_7 + (2\Delta y - 2\Delta X) = 13 + (-4) = 9$

Since P8 > 0,

Next point to plot = $(x_k+1, y_k+1) = (28,45)$

	P_k	x	y
		19	38
P_0	5	20	39
P_1	1	21	40
P_2	-3	22	40
P_3	11	23	41
P_4	7	24	42
P_5	3	25	43
P_6	-1	26	43
P_7	13	27	44
P_8	9	28	45
P_9			



	P_k	x	y
		19	38
P_0	5	20	39
P_1	1	21	40
P_2	-3	22	40
P_3	11	23	41
P_4	7	24	42
P_5	3	25	43
P_6	-1	26	43
P_7	13	27	44
P_8	9	28	45
P_9			

Example

- Starting coordinate (9,18) and Ending coordinate (14,22)

So, Plot a line using bresenham's line drawing algorithm

- Calculate the points between the starting coordinates (9, 18) and ending coordinates (14, 22).

Using Bresenham's algorithm, generate the coordinates of the pixels that lie on a line segment having the endpoints (2, 3) and (5, 8).

Draw a line from (1,1) to (8,7) using Bresenham's Line Algorithm.

- Calculate the points between the starting coordinates (9, 18) and ending coordinates (14, 22) DDA and BLA using

Draw a line from (1,1) to (8,7) using DDA and BLA algorithms.

Example -3: Draw a line from (0,0) to (7,7) using DDA Algorithm

Bresenham's Line Drawing Algorithm

Working of Bresenham's Algorithm:

- The coordinates of the **last chosen pixel** upon entering ***step i*** are (xi, y_i).
- The task is to choose the **next one between** the bottom pixel **S** and top pixel **T**.
  - If **S** is chosen, we have: $x_{i+1} = x_i + 1$ and $y_{i+1} = y_i$
  - If **T** is chosen, we have: $x_{i+1} = x_i + 1$ and $y_{i+1} = y_i + 1$
- The actual y coordinate of line at $x = x_{i+1}$ is:

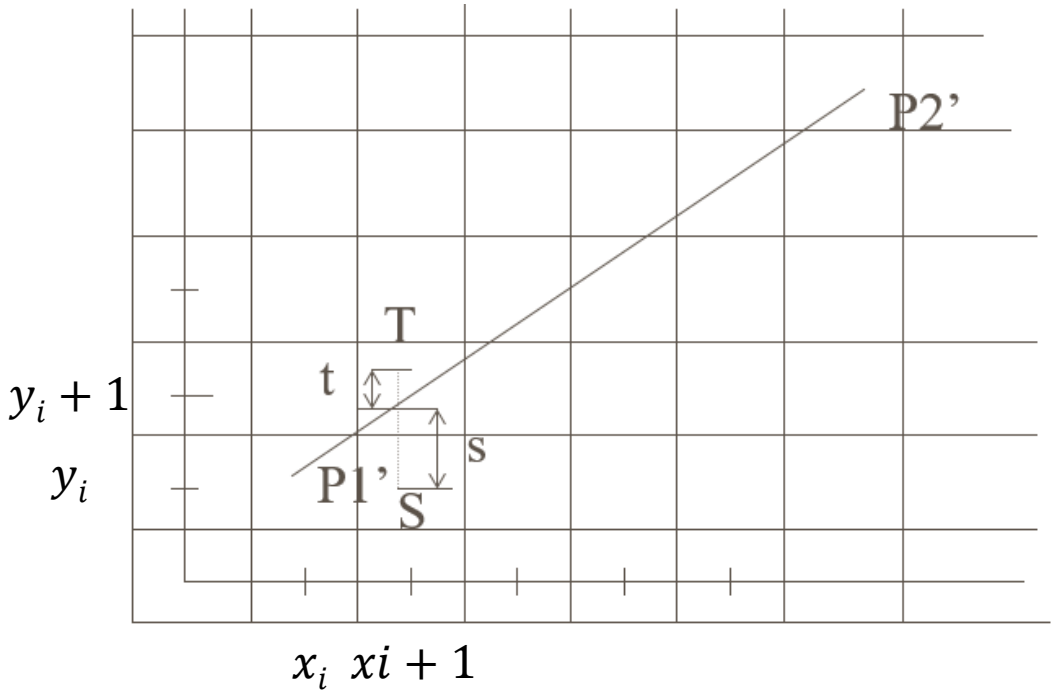
$$y = mx + b = m\,x_{i+1} + b$$
$$= m(xi + 1) + b \quad \ldots\ldots\ldots\ldots \text{(eq. 1)}$$

- The distance from **S** to actual line in y direction is:

$$s = y - y_i$$

- The distance from **T** to actual line in y direction is:

$$t = (y_{i+1}) - y$$



Bresenham's Line Drawing Algorithm

Working of Bresenham's Algorithm:

- Now consider the difference between these two distance values $s - t$.
  - When $s - t < 0$ we have $s < t$ and the closest pixel is **S**.
  - When $s - t > 0$ we have $s > t$ and the closest pixel is **T**.
  - We also choose **T** when $s - t = 0$.
- The difference is:-

$$s - t = (y - y_i) - [(y_i + 1) - y] = y - y_i - y_i - 1 + y = 2y - 2yi - 1$$

- From (**eq. 1**) above $\Rightarrow y = m(x_i + 1) + b$

$$s - t = 2\,(m(x_i + 1) + b\,) - 2yi - 1$$

$$= 2m\,(xi + 1) + 2b - 2yi - 1 \quad \ldots\ldots\ldots\ldots\ldots\ldots \textbf{(eq. 2)}$$

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

$$s - t = 2\left(\frac{\Delta y}{\Delta x}\right)(x_i + 1) + 2b - 2yi - 1$$

$$\Delta x(s - t) = 2\Delta y(xi + 1) + (2b - 2yi - 1)\Delta x$$

Bresenham's Line Drawing Algorithm

Working of Bresenham's Algorithm:

- Now, take decision variable

$$d_i = \Delta x(s - t) \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \text{ (eq. 3)}$$

$$\begin{aligned} d_i &= 2\Delta y(x_i + 1) + (2b - 2yi - 1)\Delta x \\ &= 2\Delta y x_i + 2\Delta y + (2b - 2yi - 1)\Delta x \\ &= 2\Delta y x_i + 2\Delta y + 2b\Delta x - 2yi\Delta x - \Delta x \\ &= 2\Delta y x_i - 2yi\Delta x + 2\Delta y + 2b\Delta x - \Delta x \\ &= 2\Delta y x_i - 2yi\Delta x + 2\Delta y + (2b - 1)\Delta x \\ &= 2\Delta y x_i - 2\Delta x y_i + C \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \text{ (eq. 4)} \end{aligned}$$

where, $C = 2\Delta y + (2b - 1)\Delta x$

- Similarly,

$$d_{i+1} = 2\Delta y x_{i+1} - 2\Delta x\, y_{i+1} + C \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \text{ (eq. 5)}$$

Bresenham's Line Drawing Algorithm

Working of Bresenham's Algorithm:

- Now subtract (**eq. 4**) from (**eq. 5**), we get

$$d_{i+1} - d_i = 2\Delta y x_{i+1} - 2\Delta x y_{i+1} + C - 2\Delta y x i + 2\Delta x y i - C$$

- Put: $x_{i+1} = xi + 1$

$$\begin{aligned} d_{i+1} - di &= 2\Delta y(x_i + 1) - 2\Delta x y_{i+1} + C - 2\Delta y x_i + 2\Delta x y_i - C \\ &= 2\Delta y(x_i + 1) - 2\Delta x y_{i+1} - 2\Delta y x_i + 2\Delta x y_i \\ &= 2\Delta y(x_i + 1) - 2\Delta y x_i - 2\Delta x y_{i+1} + 2\Delta x y_i \\ &= 2\Delta y(x_i + 1 - xi) - 2\Delta x(y_{i+1} - y_i) \\ &= 2\Delta y - 2\Delta x(y_{i+1} - y_i) \end{aligned}$$

$$d_{i+1} = di + 2\Delta y - 2\Delta x(y_{i+1} - y_i) \ldots\ldots\ldots\ldots\ldots\ldots \textbf{(eq. 6)}$$

Bresenham's Line Drawing Algorithm

Working of Bresenham's Algorithm:

- Now if choose pixel ***T***, it means that $s - t > 0$

$$d_i > 0 \quad \text{as} \quad d_i = (s-t)\Delta x$$

  - then, $y_{i+1} = y_i + 1$ putting this in **(eq. 6)**

$$d_{i+1} = di + 2\Delta y - 2\Delta x(y_i + 1 - y_i)$$

$$= di + 2\Delta y - 2\Delta x$$

$$= di + 2(\Delta y - \Delta x)$$

- Now if choose pixel **S**, it means that $s - t < 0$

$$d_i < 0 \quad \text{as} \quad d_i = (s-t)\Delta x$$

  - then, $y_{i+1} = y_i$ putting this in **(eq. 6)**

$$d_{i+1} = di + 2\Delta y - 2\Delta x(y_i - y_i)$$

$$= d_i + 2\Delta y$$

Bresenham's Line Drawing Algorithm

Working of Bresenham's Algorithm:

- Thus we have, $d_{i+1} = \begin{cases} d_i + 2(\Delta y - \Delta x) & if \quad d_i > 0 \\ d_i + 2\Delta y & if \quad d_i < 0 \end{cases}$
- Now we calculate d_i from the original value of d_i
  - From (**eq. 3**) $\Rightarrow d_i = \Delta x(s - t)$
  - From (**eq. 2**) $\Rightarrow = \Delta x(2m(x_i + 1) + 2b - 2yi - 1)$

$$d_1 = \Delta x(2m(x_1 + 1) + 2b - 2y_1 - 1)$$
$$= \Delta x[2(mx_1 + b - y_1) + 2m - 1)$$

  - But $y = mx + b \Rightarrow mx + b - y = 0$
  $\Rightarrow mx_1 + b - y_1 = 0$
  - Thus $d_1 = \Delta x[2(0) + 2m - 1]$

$$d_1 = \Delta x[2m - 1]$$
$$= \Delta x[2\frac{\Delta y}{\Delta x} - 1]$$
$$d_1 = 2\Delta y - \Delta x$$

Bresenham's Line Drawing Algorithm

Bresenham's Line-Drawing Algorithm for $|m| < 1$

1. Input the **two line endpoints** and store the left endpoint in (x_0, y_0).
2. Load **(x_0, y_0)** into the frame buffer; that is, plot the first point.
3. Calculate constants Δx, Δy, $2\Delta y$, and $2\Delta y - 2\Delta x$, and obtain the starting value for the decision parameter as

$$d_0 = 2\Delta y - \Delta x$$

4. At each x_i along the line, starting at $i = 0$, perform the following test:

   If $d_i < 0$, the **next point** to plot is (x_{i+1}, y_i) and

$$d_{i+1} = di + 2\Delta y$$

   Otherwise (if $d_i > 0$), *the next point to plot is* (x_{i+1}, y_{i+1}) *and*

$$d_{i+1} = d_i + 2\Delta y - 2\Delta x$$

5. Repeat **step 4** Δx times.

Bresenham's Line Drawing Algorithm

Bresenham's Line-Drawing Algorithm for $|m| < 1$

Algorithm: Algorithm for scan converting a line from $P_1'(x_1', y_1')$ to $P_2'(x_2', y_2')$ with $x_1' < x_2'$ and $0 < m < 1$.

```
1)  int x  = x1' and y = y1'
2)  int dx = x2' - x1', dy = y2' - y1, dT = 2(dy - dx), dS = 2dy
3)  int d = 2dy - dx
4)  setpixel(x, y)
5)  while(x < x2')
    {
        x++
        if(d < 0)
            d = d + dS
        else
        {
            y++
            d = d + dT
        }
        setpixel(x, y)
    }
```

Bresenham's Line Drawing Algorithm

Example 1: Scan convert a line from $P_1(1,1)$ and $P_2(8,5)$ with $0 < m < 1$.

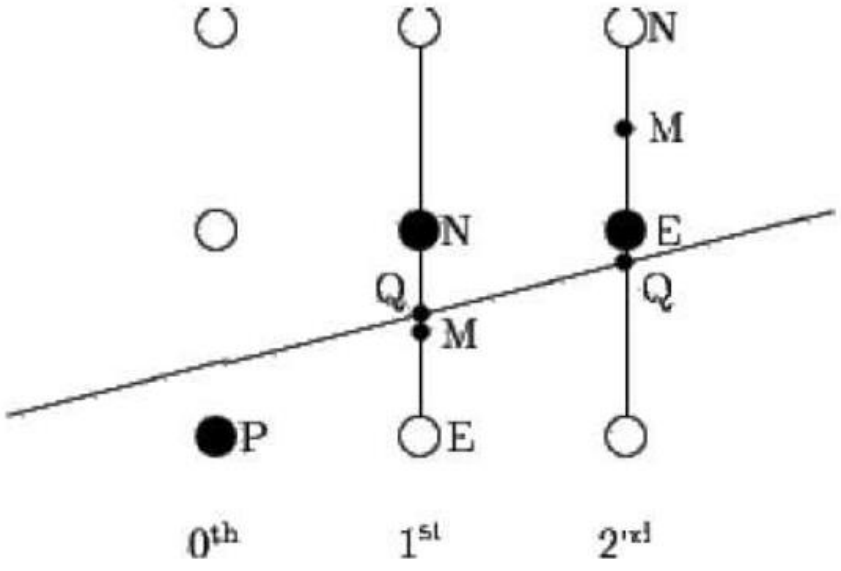
$$x_1 = 1, \quad y_1 = 1, \quad x_2 = 8, \quad y_2 = 5$$

$x = x_1$	$y = y_1$	$dx = x_2 - x_1$	$dy = y2 - y_1$	$dT = 2(dy - dx)$	$dS = 2dy$	$d = 2dy - dx$	Plot
1	1	7	4	-6	8	1	1 , 1
2	2					-5	2 , 2
3						3	3 , 2
4	3					-3	4 , 3
5						5	5 , 3
6	4					-1	6 , 4
7						7	7 , 4
8	5					1	8 , 5

Mid-Point Algorithm

- **Mid-point algorithm** is due to Bresenham which was modified by Pitteway and Van Aken.

- Assume that you have already put the point **P** at (x, y) coordinate and the slope of the line is $0 \leq k \leq 1$ as shown in the illustration.
- Now you need to decide whether to ***put the next*** point at E or N.
  - This can be chosen by identifying the **intersection point *Q*** closest to the point ***E* or *N***.
  - If the intersection point Q is closest to the point N then N is considered as the next point; otherwise E.



Mid-Point Algorithm

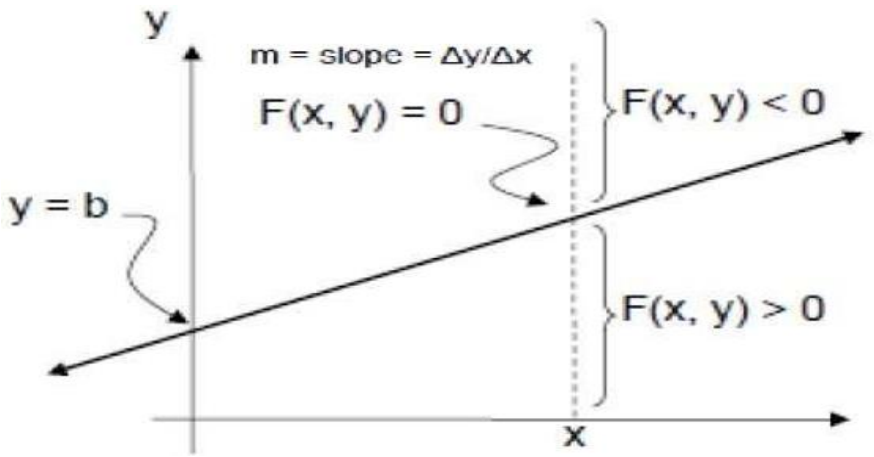
- To determine which point to choose, first calculate the mid-point

$$\boldsymbol{M(x+1, y\ +\ ½)}$$

  - If the **intersection point Q** of the line with the vertical line connecting E and N is below M, then take E as the next point; otherwise take N as the next point.
  - In order to check this, we need to consider the implicit line equation:

$$\boldsymbol{F(x, y) = mx + b - y}$$

- For a positive slope m at any given x,
  - If y is on the line, then $F(x, y) = 0$
  - If y is above the line, then $F(x, y) < 0$
  - If y is below the line, then $F(x, y) > 0$



Circle Drawing – Circle Generating Algorithms

- A **circle** is defined as the **set of points** that are all at a given **distance** r from a center position (xc, y_c).
- This **distance relationship** is expressed by the Pythagorean theorem in Cartesian coordinate as:

$$(x - xc)^2 + (y - y_c)^2 = r^2 \qquad (eq.1)$$

- We could use this equation to calculate the position of points on a circle circumference by stepping along the x axis in unit steps from $x_c - r$ to $x_c + r$ and calculating the corresponding y values at each position as:

$$y = yc \pm \sqrt{(x - xc) - r^2} \qquad (eq.2)$$

- **But this** is not the best method for generating a **circle.**
- One problem with this **approach** is that it involves considerable **computation at each step.**
- Moreover, the **spacing between plotted pixel positions** is **not uniform.**
  - The spacing can be adjusted by interchanging x **and** y (stepping through y values and calculating x values) whenever the **absolute** value of the **slope** of the circle is greater than 1.
  - But, this **simply increases** the **computation and processing required** by the algorithm.

Circle Drawing – Circle Generating Algorithms

- Another way to eliminate the ***unequal spacing*** is to calculate points along the circular boundary using polar coordinates ***r and θ*** $(eq.3)$

$$x = xc + r\cos\theta, \quad y = y_c + r\sin\theta \qquad (eq.3)$$

- When a display is generated with these ***equations*** using a fixed angular step size, a circle is plotted with equally spaced points along the ***circumference***.
- Computation can be ***reduced*** by considering the ***symmetry of circles***.
- The shape of the circle is similar in each **quadrant**.
  - We can generate the circle section in the **second quadrant** of the $xy\ plane$ by noting that the **two circle** sections are symmetric with respect to the y **axis.**
  - And circle sections in the third and fourth quadrants can be obtained from sections in the first and second quadrants by considering symmetry about the x axis.
- Determining pixel positions along a circle circumference using either $(eq.1)$ or equations $(eq.3)$ requires a good deal of computation time.
  - The Cartesian equation $(eq.1)$ involves **multiplications** and **square-root calculations,** while the parametric equations contain multiplications and trigonometric calculations.

Circle Drawing – Circle Generating Algorithms

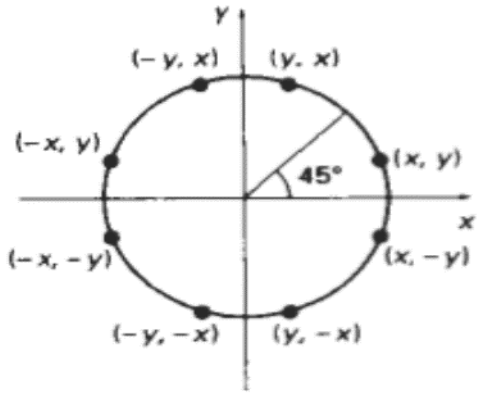
- More efficient circle algorithms are based on incremental calculation of decision-parameters, as in the **Bresenham line algorithm,** which involves only integer operations.
- A method for direct distance comparison is to test the halfway position between two pixels to determine **if this midpoint is inside or outside the circle** boundary.
- This method is **more easily applied** to other conics; and for an integer circle radius, the midpoint approach generates the **same pixel positions** as the Bresenham circle algorithm.
- Also, the error involved in locating pixel positions along any conic section using the midpoint test is limited to one-half the pixel separation.

Circle Drawing – Midpoint Circle Algorithm

- As in the raster line algorithm, we sample at unit intervals and determine the closest pixel position to the specified circle path at each step.
  - For a given radius r and screen center position (xc, y_c), we can first set up our algorithm to calculate pixel positions around a circle path centered at the coordinate origin $(0, 0)$.
  - Then each calculated position $(\boldsymbol{x}, \boldsymbol{y})$ is moved to its **proper screen position** by adding $\mathbf{x_c}$ **to** $\mathbf{x}$ **and** $\mathbf{y_c}$ **to** $\mathbf{y}$.

Circle Drawing – Midpoint Circle Algorithm

- Along the circle section from $x = 0$ to $x = y$ in the first quadrant, the slope of the curve varies from 0 to -1.
  - Therefore, we can take unit steps in the positive x direction over this octant and use a decision parameter to determine which of the two possible y positions is closer to the circle path at each step.
  - Positions in the other seven octants are then obtained by symmetry.



Circle Drawing – Midpoint Circle Algorithm

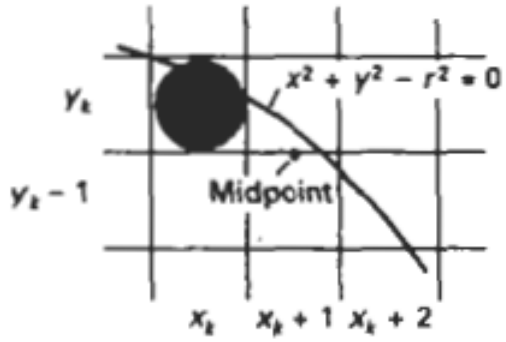
- To apply the midpoint method, we define a circle function:

$$f_{circle}(x, y) = x^2 + y^2 - r^2 \qquad (eq. 4)$$

  - Any point (x, y) on the boundary of the circle with radius r satisfies the equation $f_{circle}(x, y) = 0$.
  - If the point is in the interior of the circle, the circle function **is negative.**
  - And if the point is outside the circle, the circle function is **positive.**
- The circle-function tests are performed for the mid positions between pixels near the circle path at each ***sampling step.***
  - Thus, the circle function is the ***decision parameter*** in the midpoint algorithm, and we can set up ***incremental calculations*** for this function as we did in the line algorithm.

Circle Drawing – Midpoint Circle Algorithm

- Figure below shows the midpoint between the two candidate pixels at sampling position $x_k + 1$.
- Assuming we have just plotted the pixel at (x_k, y_k), we next need to determine whether the pixel at position $(x_k + 1,\ y_k)$ or the one at position $(x_k + 1,\ y_k - 1)$ is closer to the circle.



- Our decision parameter is the circle function evaluated at the midpoint els:

$$p_k = f_{circle}(x_k + 1, y_k - \frac{1}{2}) = (x_k + 1)^2 + (y_k - \frac{1}{2})^2 - r^2 \qquad (eq.5)$$

  - If $p_k < 0$, this midpoint is inside the circle and the pixel on scan line y_k is closer to the circle boundary.
  - Otherwise, the mid position is outside or on the circle boundary, and we select the pixel on scan line $y_k - 1$.
- Successive decision parameters are obtained using incremental calculations.

Circle Drawing – Midpoint Circle Algorithm

- We obtain a recursive expression for the next decision parameter by evaluating the circle function at sampling position $x_{k+1} + 1 = x_k + 2$:

$$p_{k+1} = f_{circle}\left(x_{k+1} + 1, y_{k+1} - \frac{1}{2}\right) = [(x_k + 1) + 1]^2 + (y_{k+1} - \frac{1}{2})^2 - r^2$$

or

$$p_{k+1} = p_k + 2(x_k + 1) + (y_{k+1}^2 - y_k^2) - (y_{k+1} - y_k) + 1 \qquad (eq.6)$$

where y_{k+1} is either y_k or $y_k - 1$, depending on the sign of p_k.

- Increments for obtaining p_{k+1} are either $2x_{k+1} + 1$ (**if p_k is negative**) or $2x_{k+1} + 1 - 2x_{k+1}$.
  - Evaluation of the terms $2x_{k+1}$ and $2y_{k+1}$ can also be done incrementally as

$$2x_{k+1} = 2xk + 2$$
$$2y_{k+1} = 2yk - 2$$

- At the start position $(0, r)$, these two terms have the values 0 and $2r$, respectively.
- Each successive value is obtained by adding 2 to the previous value $2x$ and subtracting 2 from the previous value of $2y$.

Circle Drawing – Midpoint Circle Algorithm

- The initial decision parameter is obtained by evaluating the circle function at the start position $(x_0, y_0) = (0, r)$:

$$p_0 = f_{circle}(1, r - \frac{1}{2}) = 1 + (r - \frac{1}{2})^2 - r^2$$

or

$$p_0 = \frac{5}{4} - r \qquad (eq.7)$$

  - If the radius is specified as an integer, we can simply round p_0 to $1 - r$ since all increments are integers.
- As in Bresenham's **line algorithm**, the midpoint method calculates pixel positions along the circumference of a circle using integer additions and subtractions, assuming that the circle parameters are specified in integer screen coordinates.

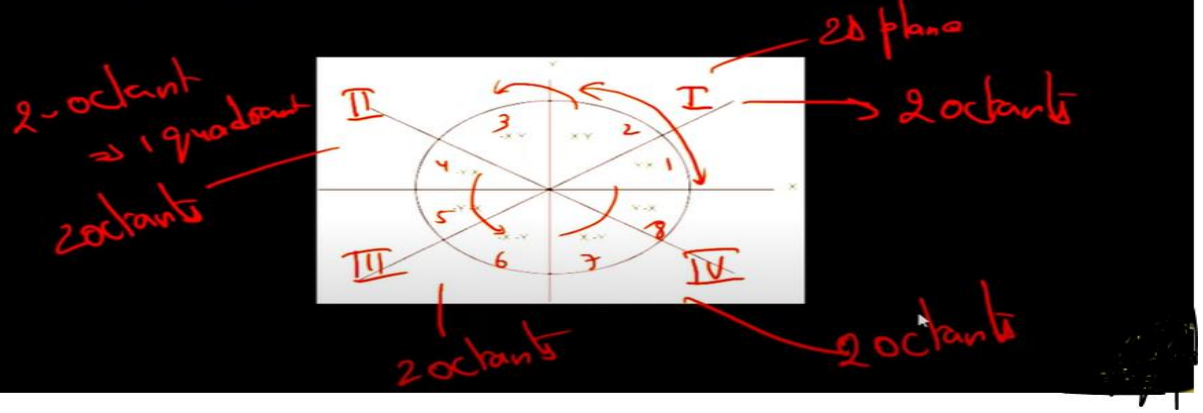
Circle Drawing – Midpoint Circle Algorithm

Introduction:

We all know that a circle is defined by its center and radius. It is not easy to display an arc on the computer screen, because the screen is made up of pixels which are organized in the form of matrix. So, for drawing a circle on screen we need to consider the nearest pixels from a printed pixels.

The main property of circle is its symmetry, we have to find points of circle only for one octant, the other octants can be derived easily.

Let's consider a circle and divide into 8 octants on 2D plane, We are going to calculate pixel locations for the octant x=0 to x=y.



- Mid point circle algorithm is used to determine the points needed for rasterizing a circle.
- It is similar to mid point line generation algorithm but only the boundary condition is different.
- It calculate all the perimeter points of the circle in the first octant and then print them along with their mirror points in the other octants.

Algorithm:

Step 1: Consider a center coordinates (X1, Y1) as
X1 = 0;
Y1 = r;

Step 2: Calculate the starting decision parameter d1 :
d1 = 1 – r;

Step 3: Let us assume, starting coordinates as = (Xk, Yk)
So, the next coordinates are (Xk+1, Yk+1)
Finding the next point on first octant based on the value of the decision parameter (dk).

Step 4: Consider

```
=> Case 1: If dk<0
          dk+1=dk+2x+1
          Xn=X+1
          Yn=Y

=> Case 2: if dk>0
          dk+1=dk+2X+2Y
          Xn=X+1
          Yn=Y-1
```

- If $p_k < 0$, the next point along the circle centered on $(0, 0)$ is (x_{k+1}, y_k) and

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

- Otherwise, the next point along the circle is (x_{k+1}, y_{k-1}) and

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

where $2x_{k+1} = \mathbf{2xk + 2}$ and $2y_{k+1} = \mathbf{2yk - 2}$

Step 5: If the center coordinate point (X1, Y1) is not at the origin (0, 0)
Then finding the points as follow:

For x coordinate = Xc + X1;

For y coordinate = Yc + Y1
{Xc and Yc contains the current values of X and Y}

Step 6: Repeat step 4 and 5 till we get X>=Y.

Advantages of midpoint circle drawing algorithm:

- It is an efficient algorithm.
- It is easy to implement.
- It is used to create curves on a raster display.

Disadvantages of midpoint circle drawing algorithm:

- Time-consuming algorithm.
- Sometimes the points of the circle are not accurate.

Example:

Draw a circle using mid-point circle algorithm centred at origin with radius 15.

Solution:

=> Centre Coordinates are (X0, Y0) = (0, 0), and radius = 15

=> Consider the starting coordinates are (X0, Y0) as-

X0 = 0

Y0 = 0

r =15

=> The value of initial decision parameter d0 is:

d0 = 1 – r

= 1- 15

= -14

=> Case 1: If dk<0
dk+1=dk+2x+1
Xn=X+1
Yn=Y

=> Case 2: if dk>0
dk+1=dk+2X+2Y
Xn=X+1
Yn=Y-1

=> As dk is less than 0, Xn=1,Yn=15
=> dk=-14+2*1+1=-11 // again less than 0 therefore Xn=2,Yn=15
=> dk=-11+2*2+1=-6 // again less than 0 therefore Xn=3,Yn=15
=> dk=-6+2*3+1=1 // here greater than 0 therefore Xn=4,Yn=14

=> dk = 1+2*4+1-2*14 = -18 // here less than 0 therefore Xn=5,Yn=14
=> dk = -18+2*5+1 = -7 // here less than 0 therefore Xn=6,Yn=14
=> dk = -7+2*6+1= 6 // here greater then 0 therefore Xn=7,Yn=13
=> dk = 6+2*7+1-2*13 = -5 // here less than 0 therefore Xn=8,Yn=13
=> dk = -5+2*8+1 = 12 // here greater than 0 therefore Xn=9,Yn=12
=> dk = 12+2*9+1-2*12 = 7 // here greater than 0 therefore Xn=10,Yn=11
=> (dk=7+2*10+1-2*11=6 // here greater than 0 therefore Xn=11,Yn=10)
=> Here we will stop because
X is greater than Y.

=> So, as we got all the points of X and Y
then just draw it as per points.

- Draw a circle where center is (5,7) and diameter is 12 using Midpoint Algorithm.

Solution:

Given, Center (X,Y) = (5,7) & Radius r = 12 ÷ 2 = 6

Now, initial point (x,y)= (0, 6)

Calculation table:

p	x = 0	y = 6	(x_{plot}, y_{plot})
1 – 6 = -5	0 + 1 = 1	6	(5+1,7+6) = (6,13)
-5 + 2×0 + 3 = -2	1 + 1 = 2	6	(5+2,7+6) = (7,13)
-2 + 2×1 + 3 = 3	2 + 1 = 3	6 – 1 = 5	(5+3,7+5) = (8,12)
3 + 2(2-6) + 5 = 0	3 + 1 = 4	5 – 1 = 4	(5+4,7+4) = (9,11)
0 + 2(3-5) + 5 = 1	4 + 1 = 5	4 – 1 = 3	(5+5,7+3) = (10,10)
1 + 2(4-4) + 5 = 6	5 + 1 = 6	3 – 1 = 2	(5+6,7+2) = (11,9)
6 + 2(5-3) + 5 = 15	6 + 1 = 7	2 – 1 = 1	(5+7,7+1) = (12,8)
15 + 2(6-2) + 5 = 28	7 + 1 = 8	1 – 1 = 0	(5+8,7+0) = (13,7)
28 + 2(7-1) + 5 = 45	8 + 1 = 9	0 – 1 = -1	(5+9,7-1) = (14,6)

Circle Drawing – Midpoint Circle Algorithm

Midpoint Circle Algorithm

1. Input radius r and circle center (x_c, y_c), and obtain the first point on the circumference of a circle centered on the origin as $(x_0, y_0) = (0, r)$.
2. Calculate the initial value of the decision parameter as: $p_0 = \frac{5}{4} - r$
3. At each x_k position, starting at $\boldsymbol{k = 0}$, perform the following test:
   - If $\boldsymbol{p_k < 0}$, the next point along the circle centered on $(0, 0)$ is (x_{k+1}, y_k) and
     $$p_{k+1} = p_k + 2x_{k+1} + 1$$
   - Otherwise, the next point along the circle is (x_{k+1}, y_{k-1}) and
     $$p_{k+1} = p_k + 2x_{k+1} + 1 - 2y_{k+1}$$
     where $2x_{k+1} = \boldsymbol{2xk + 2}$ and $2y_{k+1} = \boldsymbol{2yk - 2}$
4. Determine symmetry points in the other seven octants
5. Move each calculated pixel position (x, y) onto the circular path centered on (x_c, y_c) and plot the coordinate values: $x = x + x_c, \quad y = y + y_c.$
6. Repeat **steps 3** through **5** until $\boldsymbol{x = y}$.

Circle Drawing – Midpoint Circle Algorithm

- A C++ implementation of midpoint circle algorithm is given below:

```
void circleMidpoint(int xCenter, int yCenter, int radius)
{
    int x = 0, y = radius;
    int p = 1 - radius;
    circlePlotPoints(xCenter, yCenter, x, y);
    while (x < y)
    {
        x++;
        if (p < 0)
            p += 2 * x + 1;
        else
        {
            y--;
            p += 2 * (x - y) + 1;
        }
        circlePlotPoints(xCenter, yCenter, x, y);
    }
}
```

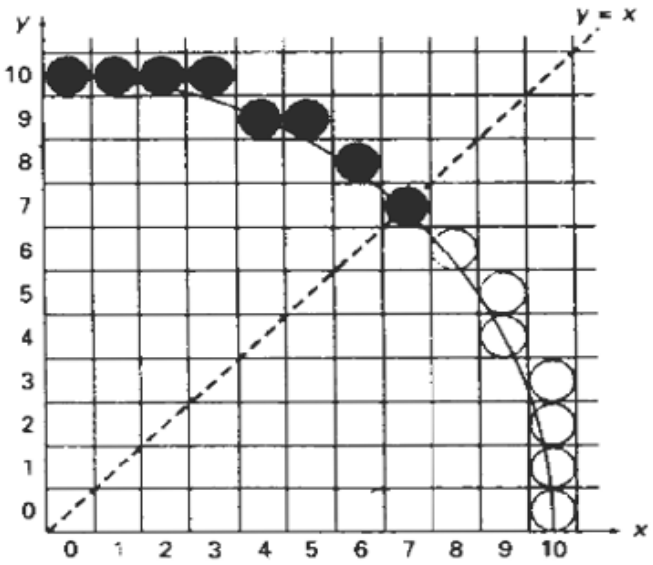
```
void circlePlotPoints(int xCenter, int yCenter, int x, int y)
{
    putpixel(xCenter + x, yCenter + y, WHITE);
    putpixel(xCenter - x, yCenter + y, WHITE);
    putpixel(xCenter + x, yCenter - y, WHITE);
    putpixel(xCenter - x, yCenter - y, WHITE);
    putpixel(xCenter + y, yCenter + x, WHITE);
    putpixel(xCenter - y, yCenter + x, WHITE);
    putpixel(xCenter + y, yCenter - x, WHITE);
    putpixel(xCenter - y, yCenter - x, WHITE);
}
```

Circle Drawing – Midpoint Circle Algorithm

Example: Given a circle radius $r = 10$, we demonstrate the midpoint circle algorithm by determining positions along the circle octant in the first quadrant from $x = 0$ to $x = y$.

- The initial value of the **decision parameter** is: $p_0 = 1 - r = -9$.
- The initial point is $(x_0, y_0) - (0, 10)$, and initial increment terms for calculating the decision parameters are: $2x_0 = 0,\ 2y_0 = 20.$
- Successive decision parameter values and positions along the circle path are calculated using the midpoint method as:

k	p_k	(x_{k+1}, y_{k+1})	$2x_{k+1}$	$2y_{k+1}$
0	-9	(1, 10)	2	20
1	-6	(2,10)	4	20
2	-1	(3,10)	6	20
3	6	(4,9)	8	18
4	-3	(5,9)	10	18
5	8	(6,8)	14	16
6	5	(7,7)	16	14



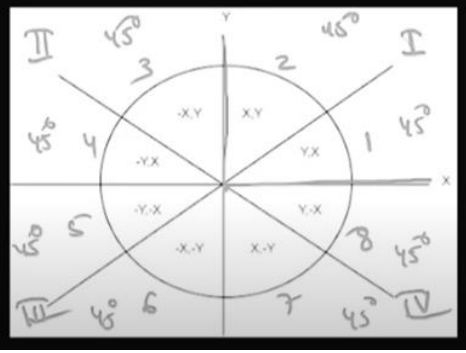
Bresenham's Circle Drawing Algorithm

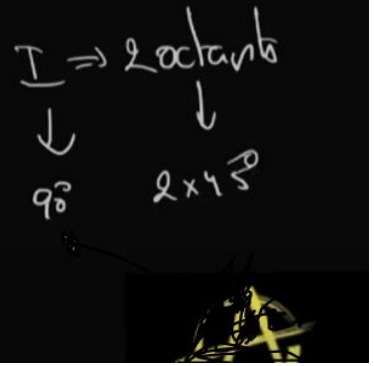
Introduction:

We all know that a circle is defined by its center and radius. It is not easy to display an arc on the computer screen, because the screen is made up of pixels which are organized in the form of matrix. So, for drawing a circle on screen we need to consider the nearest pixels from a printed pixels.

The main property of circle is its symmetry, we have to find points of circle only for one octant, the other octants can be derived easily.

Let's consider a circle and divide into 8 octants on 2D plane, We are going to calculate pixel locations for the octant x=0 to x=y.





Bresenham's Circle Drawing Algorithm:

- It is used for scan conversion.
- It attempts to generate the points of one octant and the point of other octants are generated using the eight symmetry property.
- This algorithm perform the calculation faster when compare to other algorithms.
- It select the closest pixel position to complete the arc.

Algorithm:

Step 1: Let the starting coordinates (X1, Y1) as:

$X1 = 0$;

$Y1 = r$;

Step 2: Calculate the initial decision parameter,

$d0 = 3 - 2r$;

Step 3: Let the initial coordinates are (Xk, Yk),
Next coordinates are (Xk+1, Yk+1)
Now, calculate the next point of the first octant
based on the decision parameter (dk).

Step 4 : Consider

Case 1: If dk < 0,
then Xk+1 =Xk + 1
Yk+1 =Yk
dk+1 = dk + 4Xk+1 + 6

Case 2: If dk >= 0,
then Xk+1 =Xk + 1
Yk+1 =Yk —1
dk+1 = dk + 4(Xk+1 — Yk+1)+ 10

Step 5: If the center coordinates (X1, Y1) is not at the origin (0, 0), the points generated are:

X coordinate = Xc + X1;
y coordinate = Yc + Y1;
{Xc and Yc are the current value of x and y coordinate}

Step 6: We repeat step 4 and 5 till
We get x >= y

Advantages of Bresenham's circle drawing algorithm:

- It is a simple and less time consuming algorithm.
- It can easily implemented because it uses integer arithmetic which makes the implementation less complex.
- Accuracy is high as compare to other circle drawing.

Disadvantages of Bresenham's circle drawing algorithm:

- It is not suitable for complex and high graphic images.
- There is a problem of accuracy while generating points.

Example:

Draw a circle using Bresenham's circle algorithm with center point (5,8) and radius r=9.

Solution:

=> Centre Coordinates are (X0, Y0) = (5, 8), and radius = 9

=> Consider the starting coordinates are (Xk, Yk) as-

Xk = 0;

Yk = 9;

=> The value of initial decision parameter pk is:

Pk = 3-2 * r

= 3-2*9

= -15

=> Case 1: If Pk<0

Xn=Xk+1

Yn=Yk

Pk+1=Pk+4Xk+1+6

=> Case 2: if Pk≥0

Xn=Xk+1

Yn=Yk-1

=> Initial parameter Pk<0

=> So, Xk+1= 0 + 1 = 1

Yk+1= Yk = 9

Pk+1 = Pk +4Xk+1 + 6

= -15 +(4*1) + 6

= -5

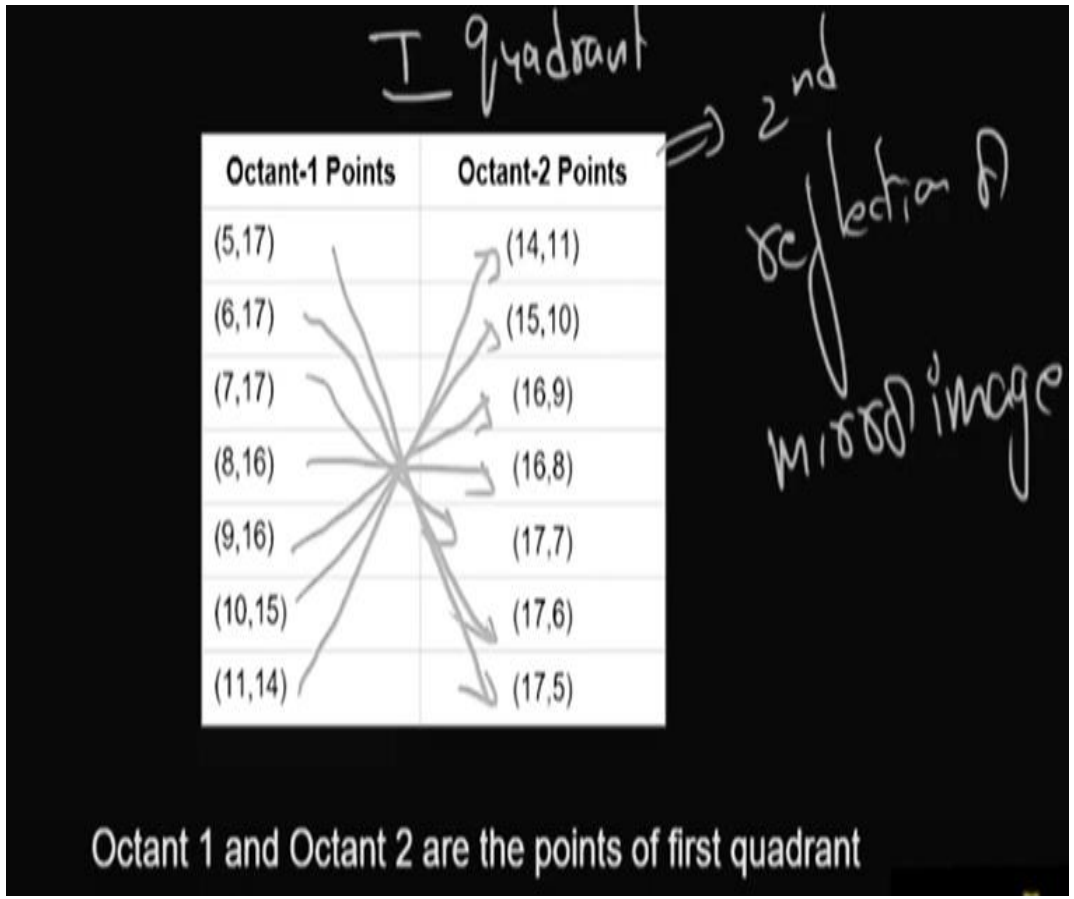
=> Continue the process until we get X>=Y.

Pk	Pk+1	(Xk+1, Yk+1)	(X plot, Y plot)
		(0,9)	(5,17)
-15	-5	(1,9)	(6,17)
-5	9	(2,9)	(7,17)
9	-1	(3,8)	(8,16)
-1	21	(4,8)	(9,16)
21	23	(5,7)	(10,15)
23	33	(6,6)	(11,14)

These are the points for first octant, second octant can be occurre
using mirror effect by swapping x and y coordinates.

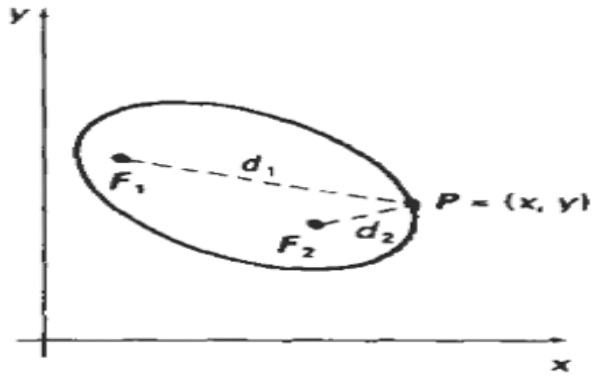
Octant-1 Points	Octant-2 Points
(5,17)	(14,11)
(6,17)	(15,10)
(7,17)	(16,9)
(8,16)	(16,8)
(9,16)	(17,7)
(10,15)	(17,6)
(11,14)	(17,5)

Octant 1 and Octant 2 are the points of first quadrant

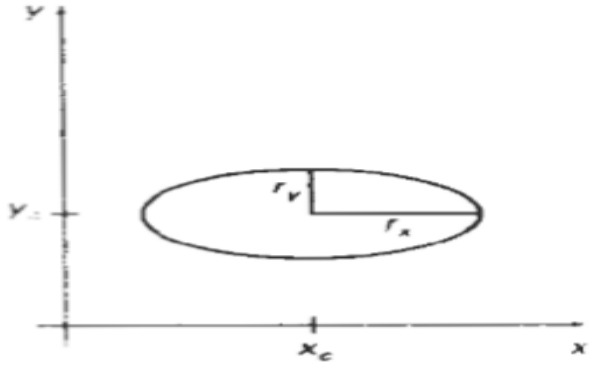


Ellipse – Generating Algorithms

- An ellipse is defined as the set of points such that the sum of the distances from two fixed positions (foci) is the same for all points.
- Therefore, elliptical curves can be generated by modifying circle-drawing procedures to take into account the different dimensions of an ellipse along the **major and minor axes.**



(a) Ellipse generated about foci $\mathbf{F_1}$ and $\mathbf{F_2}$.



(b) Ellipse centered at (x_c, y_c) with semimajor axis r_x, and semiminor axis r_y.

Ellipse – Generating Algorithms

- If the distances to the two foci from any point $P = (x, y)$ on the ellipse are labeled d_1 and d_2, then the general equation of an ellipse can be stated as:

$$d_1 + d_2 = constant \qquad (eq.1)$$

- Expressing distances d_1 and d_2 in terms of the focal coordinates $F_1 = (x_1, y_1)$ and $F_2 = (x_2, y_2)$, we have:

$$\sqrt{(x - x_1)^2 + (y - y_1)^2} + \sqrt{(x - x_2)^2 + (y - y_2)^2} = constant \qquad (eq.2)$$

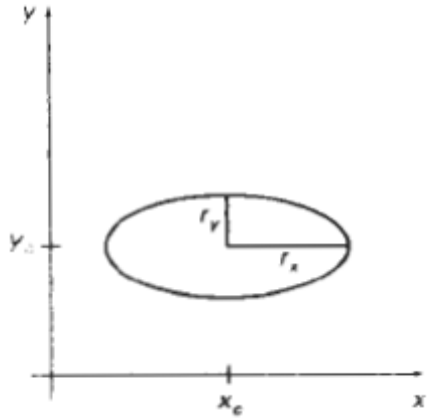
- By squaring this equation, isolating the remaining radical, and then squaring again, we can rewrite the general ellipse equation in the form:

$$Ax^2 + By^2 + Cxy + Dx + Ey + F = 0 \qquad (eq.3)$$

where the coefficients A, B, C, D, E and F are evaluated in terms of the focal coordinates and the dimensions of the major and minor axes of the ellipse.

Ellipse – Generating Algorithm

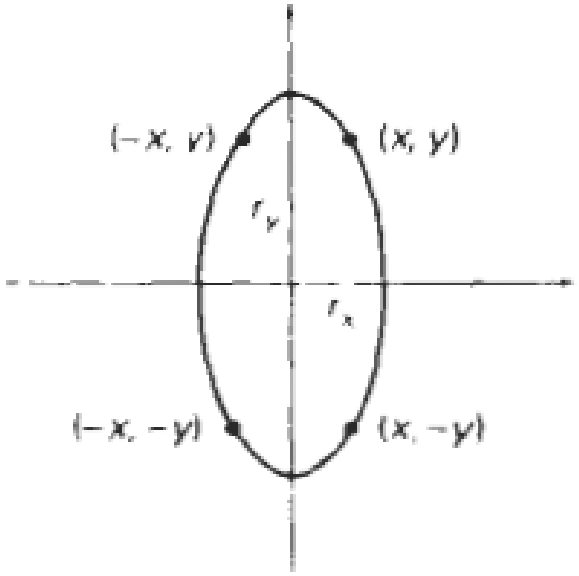
- The **major axis** is the straight line segment extending from one side of the ellipse to the other through the foci.
- The **minor axis** spans the shorter dimension of the ellipse, bisecting the major axis at the halfway position (ellipse center) between the two foci.
- Ellipse equations are greatly simplified if the major and minor axes are oriented to align with the coordinate axes.
  - Ellipse centered at (x_c, y_c) with semimajor axis r_x and semiminor axis r_y.
    - The equation of the ellipse in terms of the ellipse center coordinates and parameters r_x and r_y as:



$$\left(\frac{x-x_c}{r_x}\right)^2 + \left(\frac{y-y_c}{r_y}\right)^2 = 1 \qquad (eq.4)$$

Ellipse – Generating Algorithm

- Symmetry considerations can be used to further reduce computations.
- An ellipse in standard position is symmetric between quadrants, but unlike a circle, it is not symmetric between the two octants of a quadrant.



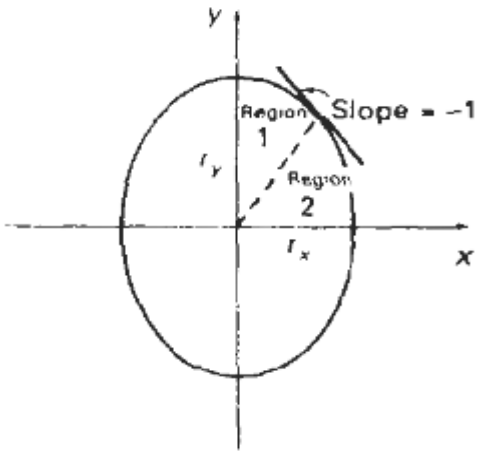
- Thus, we must calculate pixel positions along the elliptical arc throughout one quadrant, then we obtain positions in the remaining three quadrants by symmetry.

Ellipse – Generating Algorithm Midpoint Ellipse Algorithm

- Our approach here is similar to that used in displaying a raster circle.
- Given parameters r_x, r_y, and (x_c, y_c), we determine points (x, y) for an ellipse in standard position centered on the origin, and then we shift the points so the ellipse is centered at (x_c, y_c).
- If we wish also to display the ellipse in nonstandard position, we could then rotate the ellipse about its center coordinates to reorient the major and minor axes.
- For the present, we consider only the display of ellipses in standard position.

Ellipse – Generating Algorithm Midpoint Ellipse Algorithm

- The midpoint ellipse method is applied throughout the first quadrant in two parts.
- The figure below shows the division of the first quadrant according to the slope, of an ellipse with $r_x < r_y$.
- We process this quadrant by
  - taking unit steps in the x direction where the slope of the curve has a magnitude less than 1, and
  - taking unit steps in the y direction where the slope has a magnitude greater than 1.



- Ellipse processing regions:
  - Over region 1, the magnitude of the ellipse slope is less than 1;
  - Over region 2, the magnitude of the slope is greater than 1.

Ellipse – Generating Algorithm Midpoint Ellipse Algorithm

- Regions 1 and 2 can be processed in various ways.
  - We can start at position $(0, r_y)$ and step clockwise along the elliptical path in the first quadrant, shifting from unit steps in x to unit steps in y when the slope becomes less than 1.
  - Alternatively, we could start at $(r_x, 0)$ and select points in a counterclockwise order, shifting from unit steps in y to unit steps in x when the slope becomes greater than –1.
- With parallel processors, we could calculate pixel positions in the two regions simultaneously.
- As an example of a sequential implementation of the midpoint algorithm, we take the start position at $(0, r_y)$ and step along the ellipse path in clockwise order throughout the first quadrant.

Ellipse – Generating Algorithm Midpoint Ellipse Algorithm

- We define an ellipse function from $(eq.4)$ with $(x_c, y_c) = (0, 0)$ as

$$f_{ellipse}(x, y) = r_y^2 x^2 + r_x^2 y^2 - r_x^2 r_y^2 \qquad (eq.5)$$

which has the following properties:

$$f_{ellipse}(x, y) \begin{cases} < 0 & \text{if } (x, y) \text{ is inside the ellipse boundary} \\ = 0 & \text{if } (x, y) \text{ is on the ellipse boundary} \\ > 0 & \text{if } (x, y) \text{ is outside the ellipse boundary} \end{cases} \qquad (eq.6)$$

- Thus, the ellipse function $f_{ellipse}$ serves as the decision parameter in the **midpoint algorithm.**
- At each sampling position, we select the next pixel along the ellipse path according to the sign of the ellipse function evaluated at the midpoint between the **two candidate pixels.**

Ellipse – Generating Algorithm Midpoint Ellipse Algorithm

- Starting at $(0, r_y)$, we take unit steps in the x **direction until** we reach the boundary between **Region 1 and Region 2.**
- Then we switch to unit steps in the y direction over the **remainder of the curve** in the first quadrant.
- At each step, we need to test the value of the ***slope of the curve***.
  - The ellipse slope is calculated from $(eq.5)$ as

$$\frac{dy}{dx} = -\frac{2r_y^2 x}{2r_x^2 y} \qquad (eq.7)$$

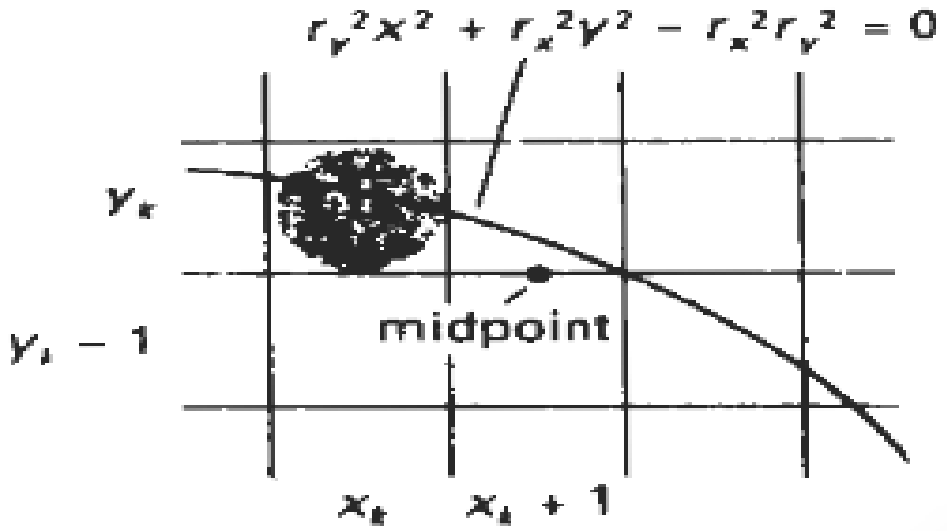
  - At the boundary between Region 1 and Region 2, $\frac{dy}{dx} = -1 \quad and \quad 2r_y^2 x = 2r_x^2 y$
  - Therefore, we move out of Region 1 whenever

$$(eq.8)$$

$$2r_y^2 x \geq 2r_x^2 y$$

Ellipse – Generating Algorithm Midpoint Ellipse Algorithm

- The figure below shows the **midpoint** between the two candidate pixels at sampling position $x_k + 1$ in the first region.



  - Midpoint between candidate **pixels at sampling** position $x_k + 1$ along an elliptical path.

Ellipse – Generating Algorithm Midpoint Ellipse Algorithm

- Assuming position (x_k, y_k) has been selected at the previous step,
  - we determine the **next position** along the ellipse path by **evaluating** the decision parameter (that is, the **ellipse function** $(eq.5)$) at this **midpoint**:

$$p1_k = f_{\boldsymbol{ellipse}}\left(x_k + 1, y_k - \frac{1}{2}\right)$$
$$= \boldsymbol{r_y^2(x_k + 1)^2 + r_x^2\left(y_k - \frac{1}{2}\right)^2 - r_x^2 r_y^2} \qquad (eq.9)$$

  - If $\boldsymbol{p1_k < 0}$, the **midpoint** is inside the **ellipse** and the **pixel** on scan line y_k is closer to the **ellipse boundary**.
  - Otherwise, the mid-position is outside or on the **ellipse boundary**, and we select the pixel on scan line $\boldsymbol{y_k} - 1$.

Ellipse – Generating Algorithm
Midpoint Ellipse Algorithm

- At the next sampling position $(xk_{+1+1} = x_k + 2)$, the decision parameter for region 1 is evaluated as

$$p1_{k+1} = f_{ellipse}\left(x_{k+1}+1, y_{k+1}-\frac{1}{2}\right)$$

$$= r_y^2[(x_k+1)+1]^2 + r_x^2\left(y_{k+1}-\frac{1}{2}\right)^2 - r_x^2 r_y^2$$

or

$$p1_{k+1} = p1_k + 2r_y^2(x_k+1) + r_y^2 + r_x^2\left[\left(y_{k+1}-\frac{1}{2}\right)^2 - \left(y_k-\frac{1}{2}\right)^2\right] \qquad (eq.10)$$

where y_{k+1} is either y_k or $y_k - 1$ depending on the sign of $p1_k$.

  - Decision parameters are incremented by the following amounts:

$$increment = \begin{Bmatrix} 2r_y^2 x_{k+1} + r_y^2, if p1_k < 0 \\ 2r_y^2 x_{k+1} + r_y^2 - 2r_x^2 y_{k+1}, if p1_k \geq 0 \end{Bmatrix}$$

Ellipse – Generating Algorithm
Midpoint Ellipse Algorithm

- As in the circle algorithm, increments for the decision parameters can be calculated using only addition and subtraction, since values for the terms and can also be obtained incrementally.
- At the initial position $(0, r_y)$, the two terms evaluate to

$$2r_y^2 x = 0 \qquad (eq.11)$$

$$2r_x^2 y = 2r_x^2 r_y \qquad (eq.12)$$

  - As x and y are incremented, updated values are obtained by adding to $(eq.11)$ and subtracting from $(eq.12)$.
  - The updated values are compared at each step, and we move from region 1 to region 2 when condition $(eq.8)$ is satisfied.

Ellipse – Generating Algorithm
Midpoint Ellipse Algorithm

- In region 1, the initial value of the **decision parameter** is obtained by evaluating the ellipse function at the start position $(x_0, y_0) = (0, r_y)$:

$$p1_0 = f_{ellipse}\left(1, r_y - \frac{1}{2}\right)$$

$$= r_y^2 + r_x^2\left(r_y - \frac{1}{2}\right)^2 - r_x^2 r_y^2$$

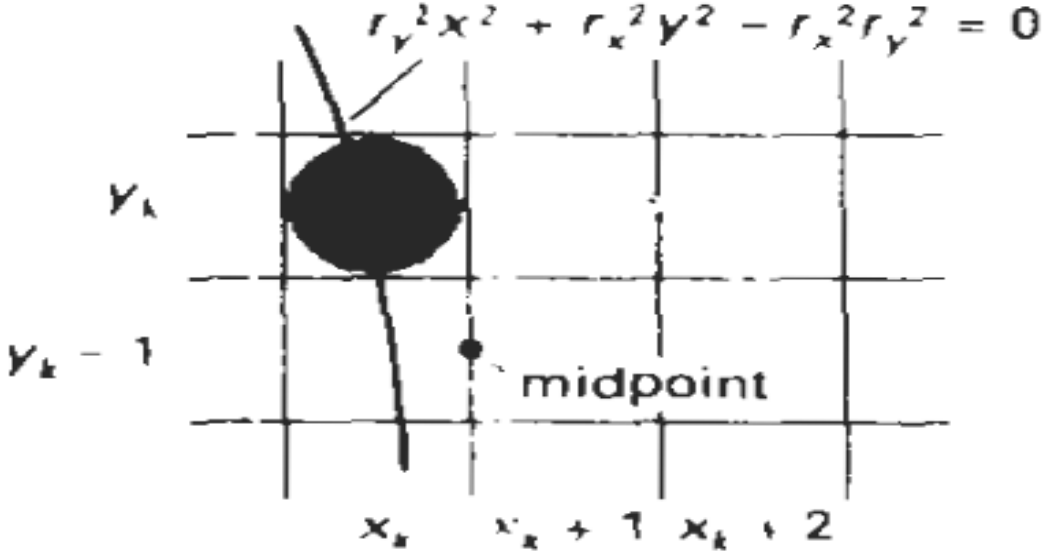
or

$$p1_0 = r_y^2 - r_x^2 r_y + \frac{1}{4}r_x^2 \qquad (eq.13)$$

Ellipse – Generating Algorithm
Midpoint Ellipse Algorithm

- Over region 2, we sample at unit steps in the negative y direction, and the midpoint is now taken between horizontal pixels at each step.
- For this region, the decision parameter is evaluated as

$$p2_k = f_{ellipse}\left(x_k + \frac{1}{2}, y_k - 1\right)$$
$$= r_y^2\left(x_k + \frac{1}{2}\right)^2 + r_x^2(y_k - 1)^2 - r_x^2 r_y^2 \qquad (eq.14)$$



- **Midpoint** between candidate pixels at sampling position $y_k - 1$ along an elliptical path.

Ellipse – Generating Algorithm
Midpoint Ellipse Algorithm

- If $p2_k > 0$, the mid-position is outside the ellipse boundary, and we select the pixel at x_k.
- If $p2_k \leq 0$, the midpoint is inside or on the ellipse boundary, and we select pixel position x_{k+1}.
- To determine the relationship between successive decision parameters in region 2, we evaluate the ellipse function at the next sampling step $y_{k+1} - 1 = y_k - 2$:

$$p2_{k+1} = f_{ellipse}\left(x_{k+1} + \frac{1}{2}, y_{k+1} - 1\right)$$
$$= r_y^2\left(x_{k+1} + \frac{1}{2}\right)^2 = r_x^2[(y_k - 1) - 1]^2 - r_x^2 r_y^2 \qquad (eq.15)$$

or

$$p2_{k+1} = p2_k + 2r_x^2(y_k - 1) + r_x^2 + r_y^2\left[\left(x_{k+1} + \frac{1}{2}\right)^2 - \left(x_k + \frac{1}{2}\right)^2\right] \qquad (eq.16)$$

with $x_k + 1$ set either to x_k or to x_{k+1}, depending on the sign of $p2_k$.

Ellipse – Generating Algorithm
Midpoint Ellipse Algorithm

- When we enter region 2, the initial position (x_0, y_0) is taken as the last position selected in region 1 and the initial decision parameter in region 2 is then

$$p2_0 = \left(x_0 + \frac{1}{2}, y_0 - 1\right) = r_y^2 \left(x_0 + \frac{1}{2}\right)^2 + r_x^2 (y_0 - 1)^2 - r_x^2 r_y^2 \qquad (eq.17)$$

  - To simplify the calculation of $p20$ we could select pixel positions in counterclockwise order starting at $(r_x, 0)$.
  - Unit steps would then be taken in the positive y direction up to the last position selected in region 1.

Ellipse – Generating Algorithm
Midpoint Ellipse Algorithm

- The midpoint algorithm can be adapted to generate an ellipse in nonstandard position using the ellipse function $(eq.5)$ and calculating pixel positions over the entire elliptical path.
  - Assuming r_x, r_y, and the ellipse center are given in integer screen coordinates, we only need incremental integer calculations to determine values for the decision parameters in the midpoint ellipse algorithm.
- The increments, r_x^2, r_y^2, $2r_x^2$ and $2r_y^2$ are evaluated once at the beginning of the procedure.
- A summary of the midpoint ellipse algorithm is listed in the following steps:

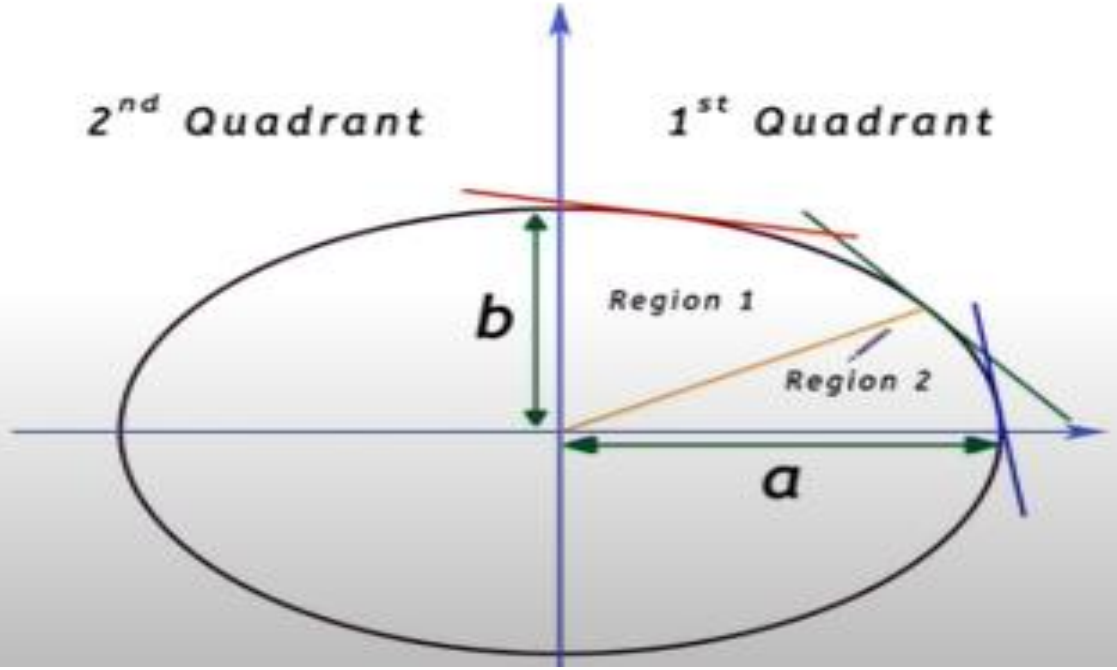
Midpoint Ellipse Drawing Algorithm

Midpoint Ellipse Drawing Algorithm

In this algorithm an ellipse will be drawn. The center of the ellipse at (0, 0)

To draw an ellipse, we will solve the algorithm for the first quadrant. Points on other coordinate will be mirrored from the first quadrant.

The first quadrant has two regions. If we draw a tangent at any point on the ellipse at region 1, the slope of the tangent must be less than 1 (m < 1). In the



Similarly, if we draw a tangent on
the ellipse at any point of region

2, the slope of the tangent must be greater than one (m > 1). In the diagram the blue tangent has a slope > 1
The green tangent is separating each region. The slope of this tangent is m = -1

The equation of an ellipse is $(x^2/a^2) + (y^2/b^2) = 1$
or, $x^2b^2 + y^2a^2 - a^2b^2 = 0 = f(x, y)$ ✓
we know the slope of any line is $m = dy/dx$
the partial derivative of f(x, y) w.r.t x = $f_x = 2xb^2$
the partial derivative of f(x, y) w.r.t y = $f_y = 2ya^2$
$dy / dx = - (f_x / f_y) = (- 2xb^2 / 2ya^2)$
So when $2xb^2 \geq 2ya^2$ calculation for region 1 is stopped and for region 2 will start.

For the Region 1:
The slope of tangent ($m < 1$)
The x increases at unit interval so, $x_{k+1} = x_k + 1$
the y value will be either y_k or $y_k - 1$

So the next point will be either $(x_k + 1, y_k)$ or $(x_k + 1, y_k - 1)$
the midpoint = (x_m, y_m)
$x_m = \{(x_k + 1) + (x_k + 1)\}/2 = x_k + 1$

$y_m = \{y_k + (y_k - 1)\}/2 = y_k - ½$

$(x_m, y_m) = (x_k + 1, y_k - ½)$

Putting the value of midpoint in the ellipse equation we get the decision parameter P_{1k}

$P_{1k} = (x_k + 1)^2b^2 + (y_k - ½)^2a^2 - a^2b^2$ [Here P_{1k} is P_K for region 1]

$P_{1k+1} = (x_{k+1} + 1)^2b^2 + (y_{k+1} - ½)^2a^2 - a^2b^2$

or $P_{1k+1} = \{(x_k + 1) + 1\}^2b^2 + (y_{k+1} - ½)^2a^2 - a^2b^2$

now,

$P_{1k+1} - P_{1k} = [\{(x_k + 1) + 1\}^2b^2 + (y_{k+1} - ½)^2a^2 - a^2b^2] - [(x_k + 1)^2b^2 + (y_k - ½)^2a^2 - a^2b^2]$

or $P_{1k+1} - P_{1k} = (x_k + 1)^2b^2 + b^2 + 2(x_k + 1)b^2 + (y_{k+1})^2a^2 + ¼\ a^2 - y_{k+1}a^2 - a^2b^2 -$

$(x_k + 1)^2b^2 - (y_k)^2a^2 - ¼\ a^2 + y_ka^2 + a^2b^2$

or $P_{1k+1} - P_{1k} = b^2 + 2(x_k + 1)b^2 + a^2\{(y_{k+1})^2 - (y_k)^2\} - a^2(y_{k+1} - y_k)$

$P_{1k+1} = P_{1k} + b^2 + 2(x_k + 1)b^2 + a^2\{(y_{k+1})^2 - (y_k)^2\} - a^2(y_{k+1} - y_k)$ [Decision parameter for first region]

The ellipse starts from (0, b), therefore putting (0, b) in P_{1k} we get,

$P_{1k} = (0+1)^2b^2 + (b - ½\ )^2a^2 - a^2b^2$

or $P_{1k} = b^2 + b^2a^2 + ¼\ a^2 - a^2b - a^2b^2$

or **$P_{1k} = b^2 + ¼\ a^2 - a^2b$ [Initial Decision parameter for first region]**

Now, if $P_{1k} \geq 0$ then the next coordinate is $(x_k + 1, y_k - 1)$

else if $P_{1k} < 0$ then the next coordinate is $(x_k + 1, y_k)$

For the Region 2:

The slope of tangent ($m > 1$)

The y decreases at unit interval so, $y_{k+1} = y_k - 1$

the x value will be either x_k or $x_k + 1$

So the next point will be either $(x_k, y_k - 1)$ or $(x_k + 1, y_k - 1)$

the midpoint = (x_m, y_m)

$x_m = \{(x_k + (x_k + 1)\}/2 = x_k + ½$

$y_m = \{(y_k - 1) + (y_k - 1)\}/2 = y_k - 1$

$(x_m, y_m) = (x_k + ½ , y_k - 1)$

Putting the value of midpoint in the ellipse equation we get the decision parameter P_{2k}

$P_{2k} = (x_k + ½)^2b^2 + (y_k - 1)^2a^2 - a^2b^2$ [Here P_{2k} is P_K for region 2]

$P_{2k+1} = (x_{k+1} + ½)^2b^2 + (y_{k+1} - 1)^2a^2 - a^2b^2$

or $P_{2k+1} = (x_{k+1} + ½)^2b^2 + \{(y_k - 1) - 1\}^2a^2 - a^2b^2$

now,

$P_{2k+1} - P_{2k} = [(x_{k+1} + ½)^2b^2 + \{(y_k - 1) - 1\}^2a^2 - a^2b^2] - [(x_k + ½)^2b^2 + (y_k - 1)^2a^2 - a^2b^2]$

or $P_{2k+1} - P_{2k} = (x_{k+1})^2b^2 + ¼\, b^2 + x_{k+1}b^2 + a^2(y_k - 1)^2 + a^2 - 2(y_k - 1)a^2 - a^2b^2$

$- (x_k)^2b^2 - x_kb^2 - ¼\, b^2 - (y_k - 1)^2a^2 + a^2b^2$

$P_{2k+1} = P_{2k} + a^2 - 2(y_k - 1)a^2 + b^2\{(x_{k+1})^2 - (x_k)^2\} + b^2(x_{k+1} - x_k)$ [Decision parameter for second region]

The initial decision parameter value will be calculated after completing the first region.

Now, if $P_{2k} \geq 0$ then the next coordinate is $(x_k, y_k - 1)$

else if $P_{2k} < 0$ then the next coordinate is $(x_k + 1, y_k - 1)$

Ellipse – Generating Algorithm Midpoint Ellipse Algorithm

Algorithm:

1. Input r_x, r_y, and ellipse center (x_c, y_c), and obtain the first point on an ellipse centered on the origin as:

   $(x_0, y_0) = (0, r_y)$

2. Calculate the initial value of the decision parameter in **region 1** as

$$p1_0 = r_y^2 - r_x^2 r_y + \frac{1}{4} r_x^2$$

3. At each x_k position in region 1, starting at $k = 0$, perform the following test: If $p1_k < 0$, the next point along the ellipse centered on (0, 0) is (x_{k+1}, y_k) and

$$p1_{k+1} = p1_k + 2r_y^2 x_{k+1} + r_y^2$$

Otherwise, the next point along the ellipse is $(x_k + 1, y_k - 1)$ and

$$p1_{k+1} = p1_k + 2r_y^2 x_{k+1} - 2r_x^2 y_{k+1} + r_y^2$$

with

$$2r_y^2 x_{k+1} = 2r^2 x_k + 2r_y^2, \qquad 2r_x^2 y_{k+1} = 2r_x^2 y_k - 2r_k^2$$

and continue until .

$$2r_y^2 x \geq 2r_x^2 y$$

Ellipse – Generating Algorithm
Midpoint Ellipse Algorithm

Algorithm:

4. Calculate the initial value of the decision parameter in **region 2** using the **last point (x_0, y_0)** calculated in region 1 as

$$p2_0 = r_y^2\left(x_0 + \frac{1}{2}\right)^2 + r_x^2(y_0 - 1)^2 - r_x^2 r_y^2$$

5. At each y_k position in region 2, starting at $k = 0$, perform the following test: If $p2_k > 0$, the next point along the ellipse centered on (0, 0) is $(x_k, y_k - 1)$ and

$$p2_{k+1} = p2_k - 2r_x^2 y_{k+1} + r_x^2$$

Otherwise, the next point along the ellipse is $(x_k + 1, y_k - 1)$ and

$$p2_{k+1} = p2_k + 2r_y^2 x_{k+1} - 2r_x^2 y_{k+1} + r_x^2$$

using the same incremental calculations for x and y as in region 1.

6. Determine symmetry points in the other three quadrants.

Move each calculated pixel position (x, y) onto the elliptical path centered on (x_c, y_c) and plot the coordinate values:

$$x = x + x_c, \qquad y = y + y_c$$

7. Repeat the steps for region 1 until .

$$2r_y^2 x < 2r_x^2 y$$

Ellipse – Generating Algorithm
Midpoint Ellipse Algorithm

Example: Midpoint Ellipse Drawing

- Given input ellipse parameters $r_x = 8$ and $r_y = 6$, we illustrate the steps in the midpoint ellipse algorithm
  - by determining raster positions along the ellipse path in the first quadrant.
  - Initial values and increments for the decision parameter calculations are

$$2r_x^2 = 0 \qquad \text{(with increment} \quad 2r_y^2 = 72 \quad \text{)}$$
$$2r_x^2 y = 2r_x^2 r_y \quad \text{(with increment} \quad -2r_x^2 = -128 \text{)}$$

- For region 1: The initial point for the ellipse centered on the origin is $(x_0, y_0) = (0, 6)$, and the initial decision parameter value is

$$p1_0 = r_y^2 - r_x^2 r_y + \frac{1}{4} r_x^2 = -332$$

Ellipse – Generating Algorithm
Midpoint Ellipse Algorithm

Example: Midpoint Ellipse Drawing

- Successive decision parameter values and positions along the ellipse path are calculated using the midpoint method as

k	$p1_k$	(x_{k+1}, y_{k+1})	$2r_y^2 x_{x+1}$	$2r_x^2 y_{x+1}$
0	-332	(1, 6)	72	768
1	-224	(2, 6)	144	768
2	-44	(3, 6)	216	768
3	208	(4, 5)	288	640
4	-108	(5, 5)	360	640
5	288	(6, 4)	432	512
6	244	(7, 3)	504	384

- We now move out of region 1, since

Ellipse – Generating Algorithm
Midpoint Ellipse Algorithm

Example: Midpoint Ellipse Drawing

- For region 2, the initial point is $(x_0, y_0) = (7, 3)$ and the initial decision parameter is

$$p2_0 = f\left(7 + \frac{1}{2}, 2\right) = -151$$

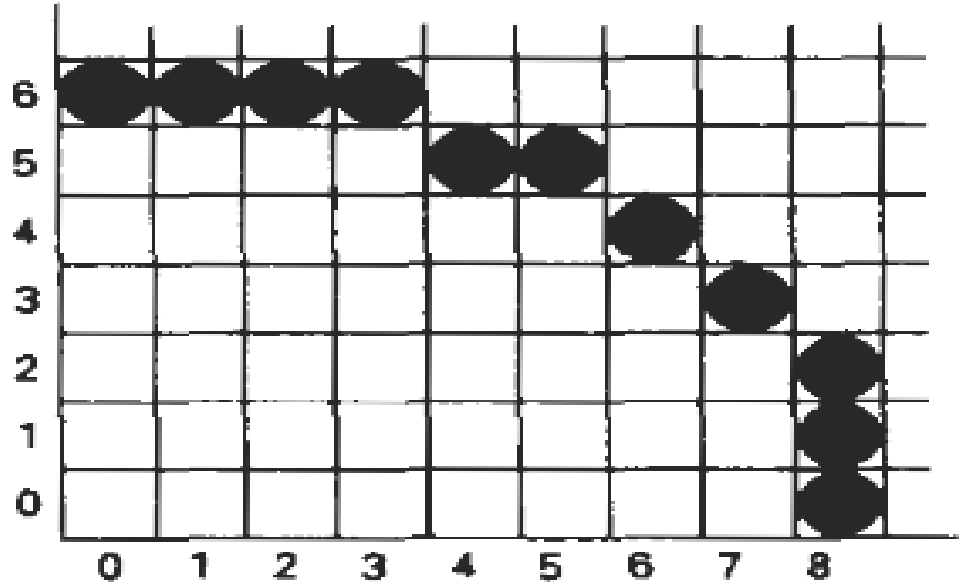
- The remaining positions along the ellipse path in the first quadrant are then calculated as

k	$p2_k$	(x_{k+1}, y_{k+1})	$2r_y^2 x_{x+1}$	$2r_x^2 y_{x+1}$
0	-151	(8, 2)	576	256
1	233	(8, 1)	576	128
2	745	(8, 0)	--	--

Ellipse – Generating Algorithm
Midpoint Ellipse Algorithm

Example: Midpoint Ellipse Drawing

- A plot of the selected positions around the ellipse boundary within the first quadrant is shown in figure below.



- Positions along an elliptical path centered on the origin with $r_x = 8$ and $r_y = 6$ using
- the midpoint algorithm to calculate pixel addresses in the first quadrant.

Polygon Drawing – Filled Area Primitives

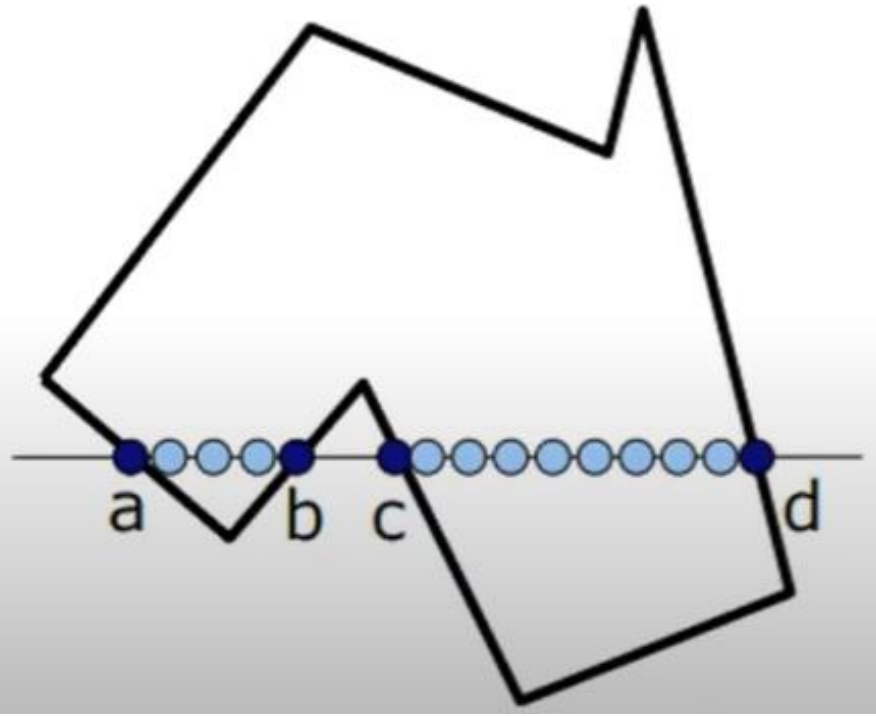
Polygon Drawing – Filled Area Primitives

- **Polygons** are the areas enclosed by **single closed loops of line segments,** where the **line segments** are specified by the **vertices** at their endpoints.

- Polygons are typically drawn with the pixels in the interior filled in, but you can also draw them as **outlines** or a set of points.

- A standard output primitive in general graphics packages is a solid-color or patterned ***polygon area***.

- This algorithm works by intersecting scanline with polygon edges and fills the polygon between pairs of intersections.



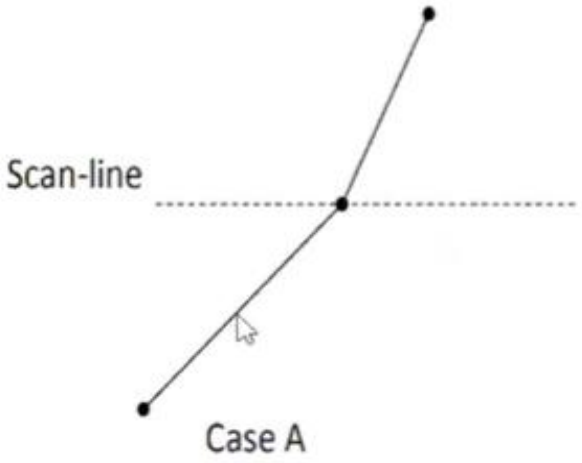
For each scan-line:

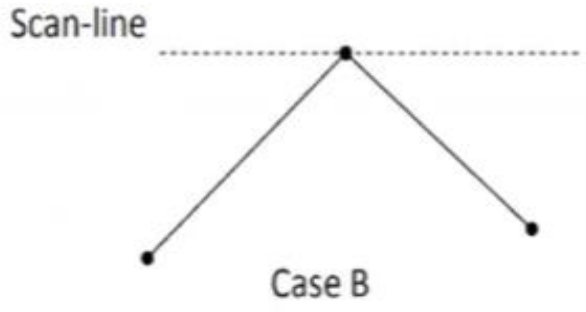
1. Locate the intersection of the scan-line with the edges.
2. Sort the intersection points from left to right.
3. Draw the interiors intersection points pairwise i.e. (a-b), (c-d)

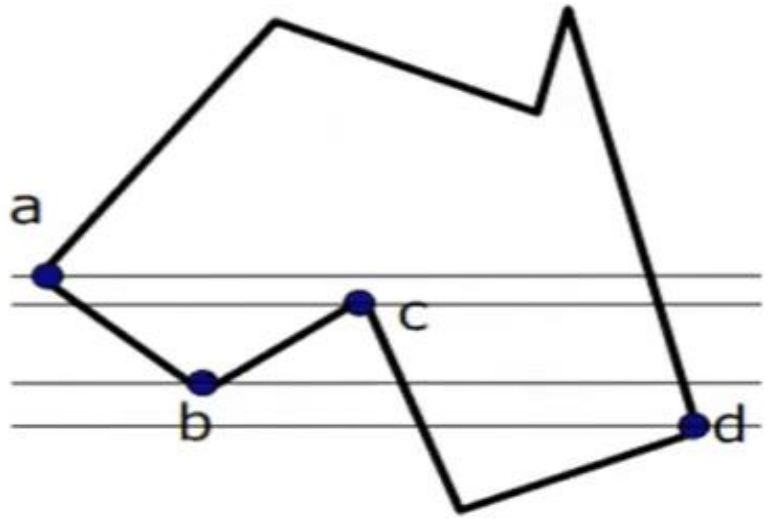
Scan Line Polygon fill approach

when the scan line passes from the vertex consider the following rules:

1. If the edges of the polygon lie on different side, count it only once.
2. If edges of the polygon lie on the same side, count it twice







As shown in above figure,

- Point a , b , c and d are intersected by 2 line segments each.
- Count b,c twice but a and d once.

Algorithm steps

1. Assume scan line start from the left and is outside the polygon.
2. When intersect an edge of polygon, start to color each pixel (because now we're inside the polygon), when intersect another edge, stop coloring .
3. Odd number of edges: inside
4. Even number of edges: outside

Steps to perform:

- For Scan line polygon filling there are three steps to perform in the following order:

1. Find the intersections of the scan line with all edges of the polygon.
2. Sort the intersections by increasing x-coordinate i.e. from left to right.
3. Make pairs of the intersections and fill in color within all the pixels inside the pair.

Steps to perform:

- For Scan line polygon filling there are three steps to perform in the following order:

1. Find the intersections of the scan line with all edges of the polygon.
2. Sort the intersections by increasing x-coordinate i.e. from left to right.
3. Make pairs of the intersections and fill in color within all the pixels inside the pair.

Scan line polygon filling algorithm is used for solid color filling in polygons.

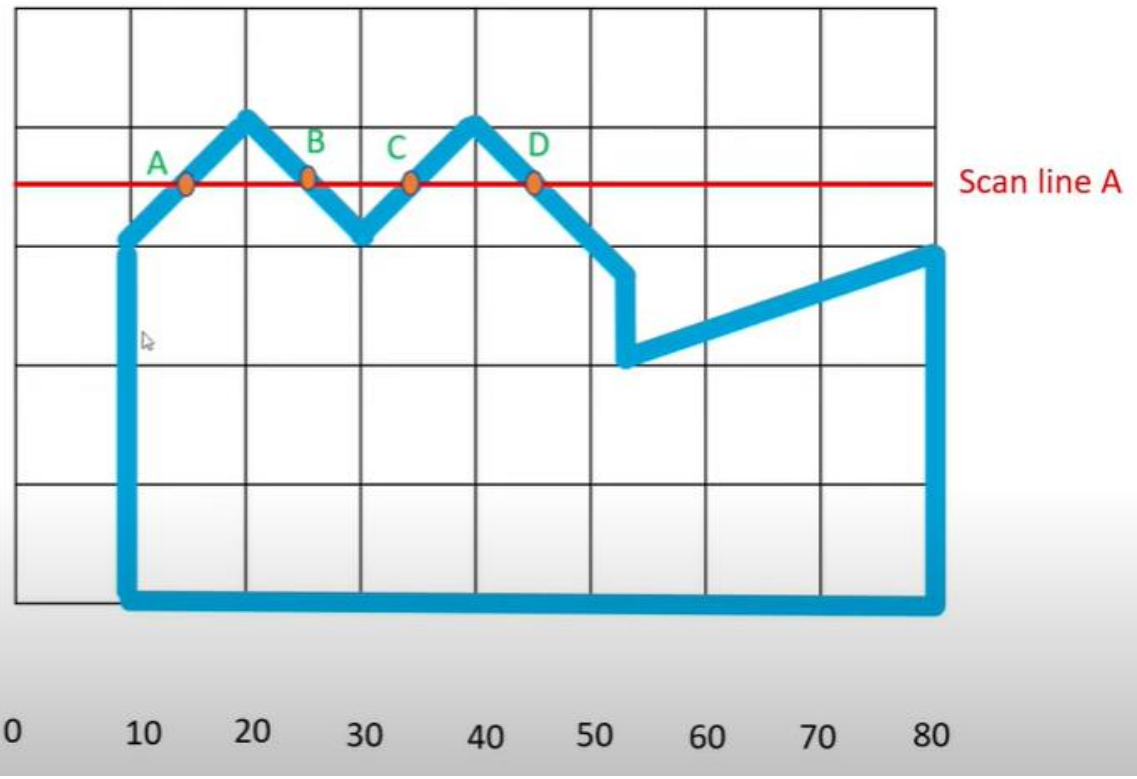
SCAN LINE A

SCAN LINE B

SCAN LINE C

SCAN LINE D

SCAN LINE E



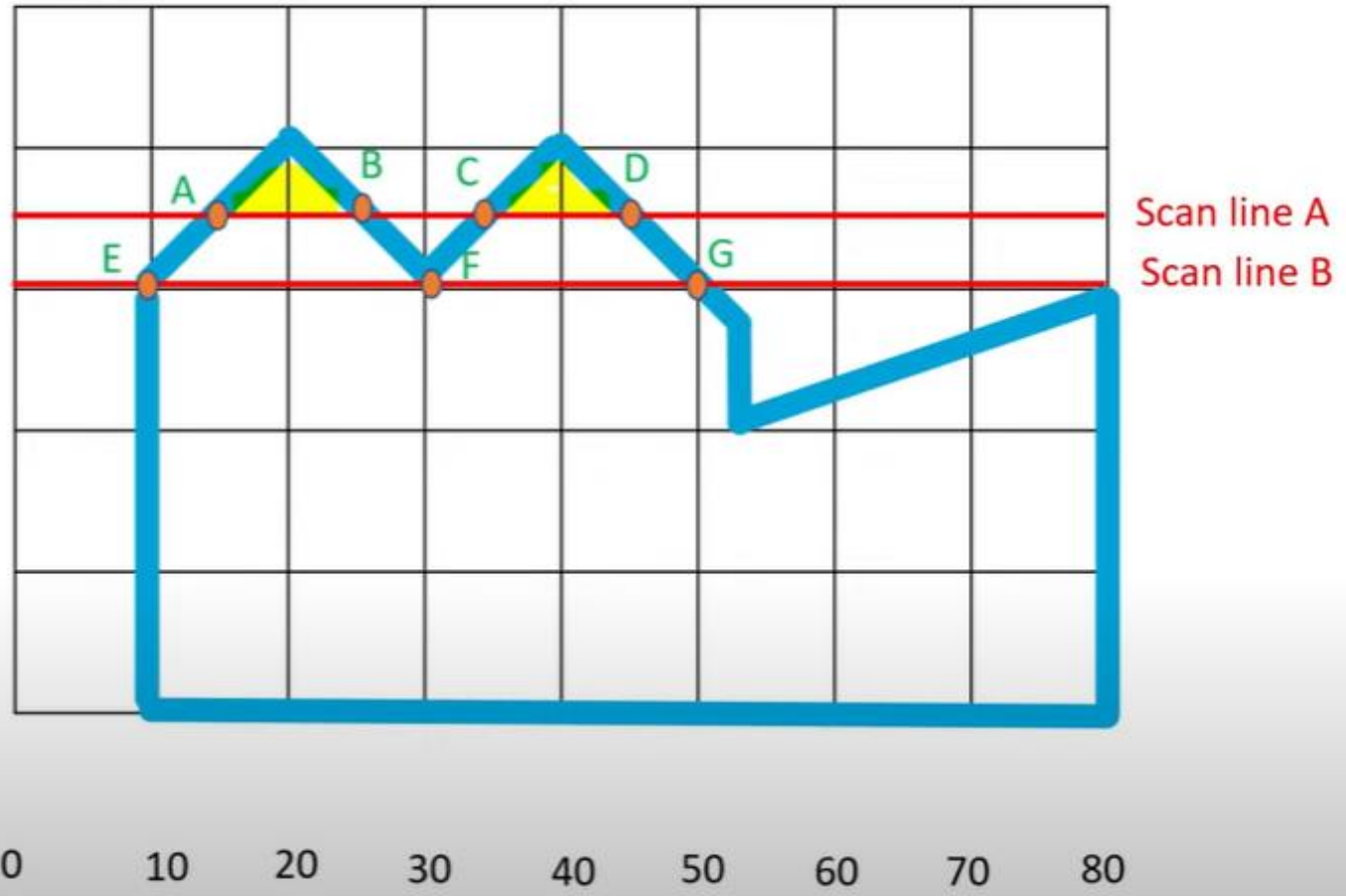
SCAN LINE A {A , B} {C , D}

SCAN LINE B {E , F} {F, G}.

SCAN LINE C

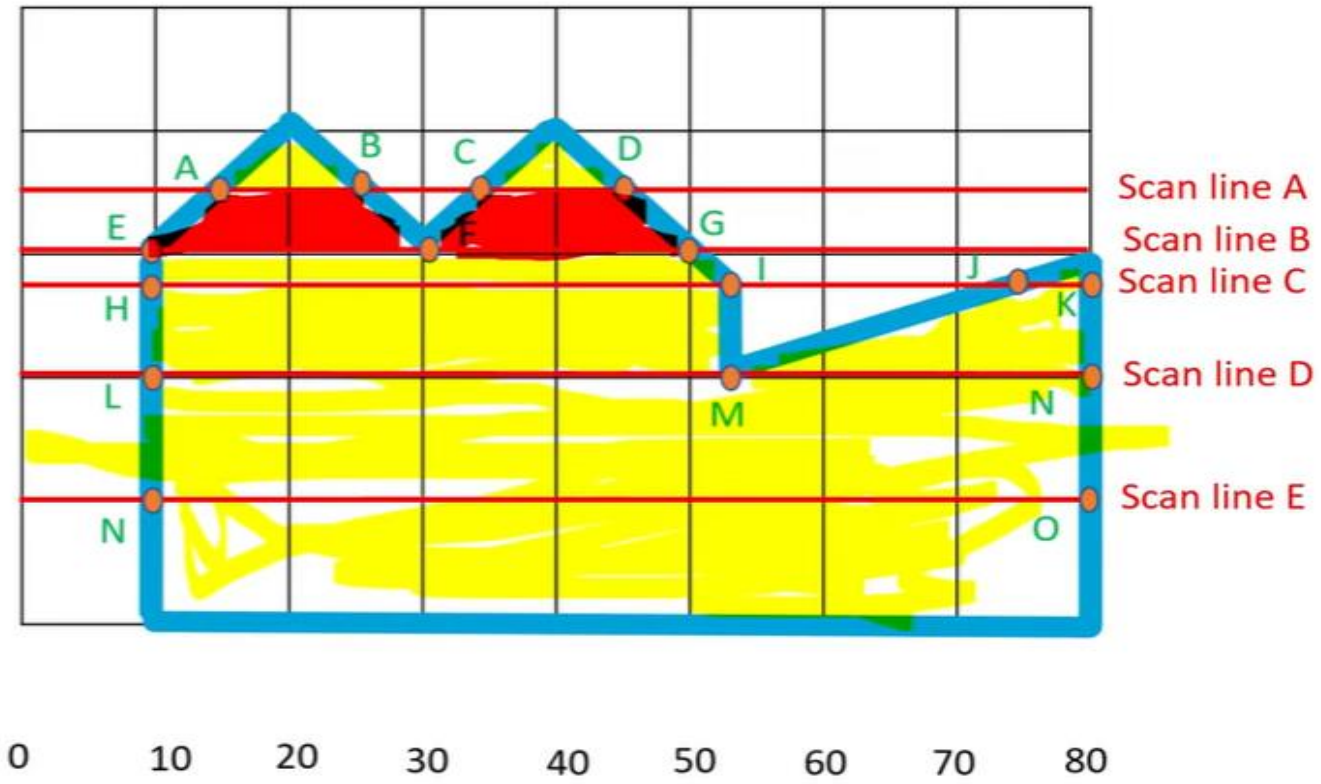
SCAN LINE D

SCAN LINE E



Scan line polygon filling algorithm is used for solid color filling in polygons.

SCAN LINE A {A , B} {C , D}
SCAN LINE B {E , F} {F, G}
SCAN LINE C {H, I} {J , K}
SCAN LINE D {L , M} {M, N}
SCAN LINE E {N , O}



Polygon Drawing – Filled Area Primitives

- There are two basic approaches to area filling on raster systems:

  1. To determine the overlap intervals for scan lines that cross the area.

  2. To start from a given interior position and paint outward from this point until encounter the specified boundary conditions.

- The scan-line approach is typically used in general graphics packages to fill polygons, circles, ellipses, and other simple curves.

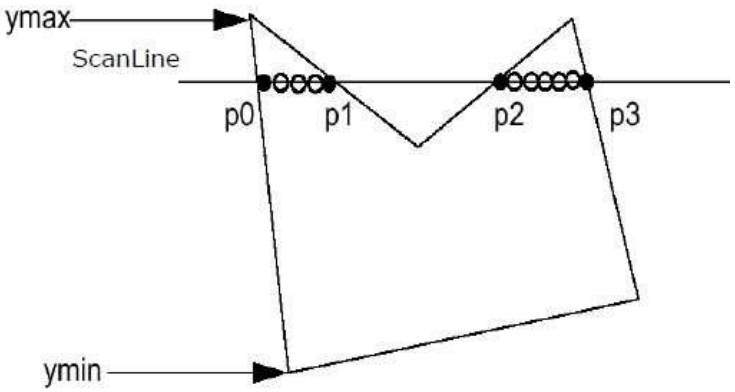
Polygon Drawing – Filled Area Primitives

Filling Polygons – Inside-outside test

- An important issue that arises when filling polygons is that of deciding whether a particular point is interior or exterior to a polygon.
  - A rule called the ***odd-parity*** (or the ***odd-even rule***) is usually applied to test whether a point is **interior or not**.
  - A half-line starting from the particular point and extending to infinity is drawn in any direction such that no polygon vertex intersects with the line.
  - The point is considered to be interior if the number of intersections between the line and the polygon edges is odd.

Polygon Drawing
The scan-line polygon filling algorithm

- The **scan-line polygon filling algorithm** involves
  - the horizontal scanning of the polygon from its **lowermost** to its **topmost** vertex,
  - identifying which edges intersect the scan-line, and
  - finally drawing the interior horizontal lines.



- The algorithm is specified as:
  - For each horizontal scan-line:
    1. List all the points that intersect with the horizontal scan-line.
    2. Sort the intersection points in ascending order of the x coordinate.
    3. Fill in all the interior pixels between pairs of successive intersections.

Polygon Drawing
The scan-line polygon filling algorithm

- The third step accepts a sorted list of points and connects them according to the ***odd-parity rule***.
  - For example, given the list $[p_1; p_2; p_3; p_4; \ldots; p_{2n-1}; p_{2n}]$, it draws the lines $p_1 \rightarrow p_2; p_3 \rightarrow p_4; \ldots; p_{2n-1} \rightarrow p_{2n}$.
  - A decision must be taken as to whether the edges should be displayed or not: given that $p_1 = (x_1, y)$ and $p_2 = (x_2, y)$, should we display the line (x_1, y, x_2, y) or just the interior points (x_{1+1}, y, x_{2-1}, y)?
- Step 1 can be optimized by making use of a ***sorted edge table***.
  - Entries in the edge table are sorted on the basis of their **lower y value**.
  - Next, edges sharing the **same low y** value are sorted on the basis of their higher y value.

Polygon Drawing
The scan-line polygon filling algorithm

- A pair of markers are used to denote the range of ***'active'*** edges in the table that need to be considered for a particular scan-line.
  - This range starts at the top of the table, and moves progressively downwards as higher scan-lines are processed.
  - Given a scan-line $y = s$ and a non-horizontal edge with end-points $(x_1, y_1), (x_2, y_2), y_1 < y_2$, an intersection between the two exists if $y_1 \leq y \leq y_2$.
  - The point of intersection is $(\frac{s-c}{m}, s)$ where $m = \frac{y_2 - y_1}{x_2 - x_1}$ and $c = y_1 - mx_1$.

Polygon Drawing

The scan-line polygon filling algorithm

An Example: Consider the polygon in Right. The edge table and edge list for such a polygon would be:

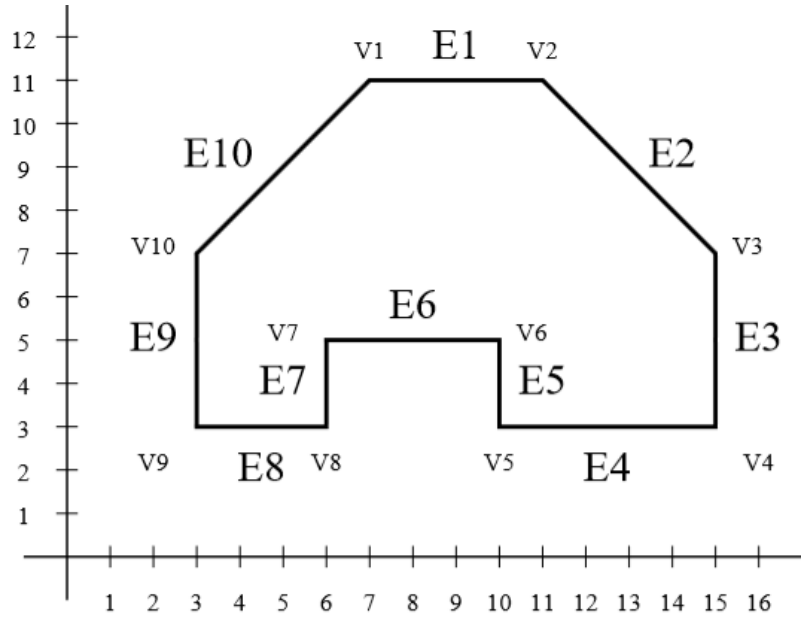
Edge Table

Edge	Y_{min}	Y_{max}	X of Y_{min}	X of Y_{max}	$\frac{1}{m}$
E1	11	11	6	11	0
E2	7	11	15	11	-1
E3	3	7	15	15	0
E4	3	3	10	15	-
E5	3	5	10	10	0
E6	5	5	10	6	-
E7	3	5	6	6	0
E8	3	3	3	6	-
E9	3	7	3	3	0
E10	7	11	3	7	1

Edge List

Scan-line	Edge number
11	-
10	-
9	-
8	-
7	2, 10
6	-
5	-
4	-
3	3, 5, 7, 9

Given Polygon



Polygon Drawing

The scan-line polygon filling algorithm

An Example: Cont….

- Note that in the above table the horizontal lines are not added to the edge list.
- The reason for this is discussed below.
  - The active edges for scan-line 3 would be 3, 5, 7, 9, these are sorted in order of their x values, in this case 9, 7, 5, 3.
  - The polygon fill routine would proceed to fill the intersections between $(3,3)$ $(E9)$ and $(6,3)$ $(E7)$ and $(10,3)$ $(E5)$ to $(15,3)$ $(E3)$.
  - The next scan-line (4) is calculated in the same manner.
  - In this the values of x do not change (since the line is vertical; it is incremented by 0).
  - The active edge at scan-line 7 are 10 and 2 (correct order).

Polygon Drawing
General-purpose filling algorithms

Boundary-fill Algorithm

- Sometimes we come across an object where we want to fill the area and its boundary with different colors.
- This makes use of coherence properties of the boundary of a primitive/figure:
  - given a point inside the region the algorithm recursively plots the surrounding pixels until the primitive boundary is reached.
- Given the ***FillColor***, the ***BoundaryColor*** and a point inside the boundary, the following algorithm recursively sets the four adjacent pixels (2 horizontal and 2 vertical) to the FillColor.

Polygon Drawing
General-purpose filling algorithms

Boundary-fill Algorithm: 4-connected

```
void boundaryFill4(int x, int y, int fill, int boundary)
{
      int current;
      current = getpixel(x, y);
      if ((current != boundary) && (current != fill))
      {
            setcolor(fill);
            setpixel(x, y):
            boundaryFill4(x+1, y, fill, boundary);
            boundaryFill4(x-1, y, fill, boundary);
            boundaryFill4(x, y+1, fill, boundary);
            boundaryFill4(x, y-1, fill, boundary);
      }
}
```

- Regions which can be completely filled with this algorithm are called **4-connected regions**.
- Some regions cannot be filled using this algorithm.
  - Such regions are called **8-connected** and algorithms filling such areas consider the four diagonally adjacent pixels as well as the horizontal and vertical ones.

Polygon Drawing
General-purpose filling algorithms

Boundary-fill Algorithm: 8-connected

```
void boundaryFill8(int x, int y, int fill, int boundary)
{
      int current;
      current = getpixel(x, y);
      if ((current != boundary) && (current != fill))
      {
            setcolor(fill);
            setpixel(x, y):
            boundaryFill8(x+1, y, fill, boundary);
            boundaryFill8(x-1, y, fill, boundary);
            boundaryFill8(x, y+1, fill, boundary);
            boundaryFill8(x, y-1, fill, boundary);
            boundaryFill8(x+1, y+1, fill, boundary);
            boundaryFill8(x+1, y-1, fill, boundary);
            boundaryFill8(x-1, y+1, fill, boundary);
            boundaryFill8(x-1, y-1, fill, boundary);
      }
}
```

- Care must be taken to ensure that the boundary does not contain holes, which will cause the fill to 'leak'.
- The 8-connected algorithm is particularly vulnerable.

Polygon Drawing
General-purpose filling algorithms

Flood-fill Algorithm:

- Sometimes we come across an object where we want to fill the area and its boundary with different colors.
- The flood-fill algorithm is used to fill a region which has the same color and whose boundary may have more than one color.

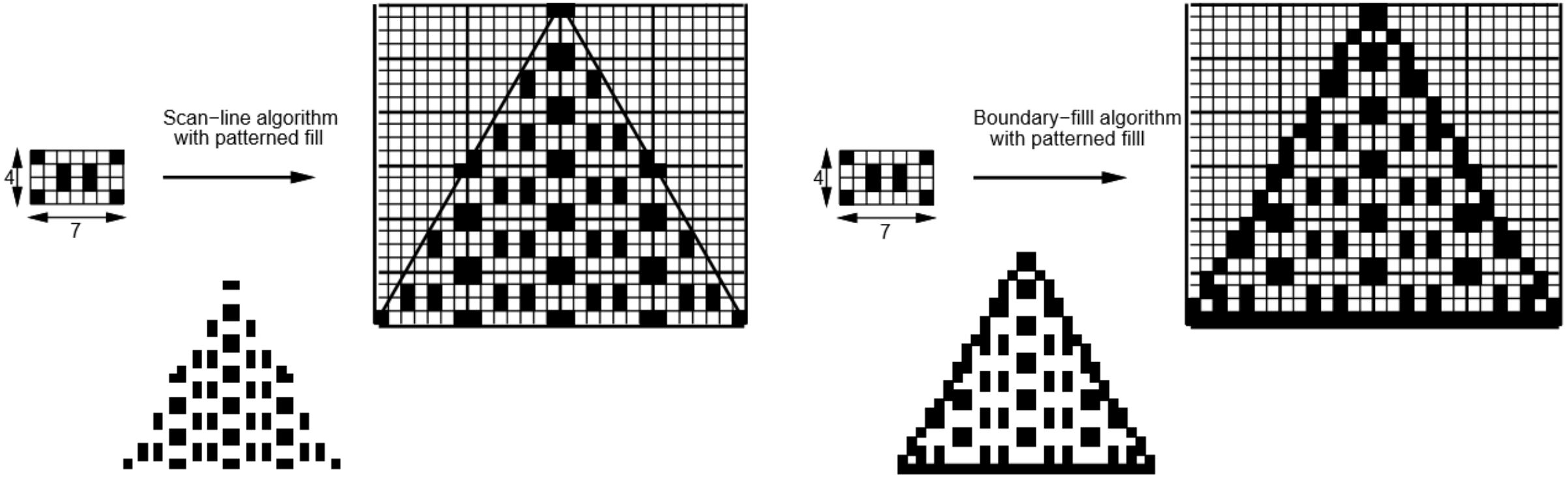
```
void floodFill4(int x, int y, int fillcolor, int oldcolor)
{
      if(getpixel(x, y) == oldcolor)
      {
            setcolor (fillcolor);
            setpixel (x, y):
            floodFill4 (x+1, y, fillColor, oldColor):
            floodfill4 (x-1, y, fillcolor, oldcolor);
            floodPill4 (x, y+1, fillcolor, oldcolor);
            floodFill4 (x, y-1, fillColor, oldcolor);
      }
}
```

Polygon Drawing

General-purpose filling algorithms

Pattern Filling

- Any of the filling algorithms discussed thus far can be modified to use a given pattern when filling.
  - The effects of using this with the scan-line and boundary fill algorithms respectively is shown below:



Polygon Drawing
General-purpose filling algorithms

Pattern Filling

- Patterned fills can be achieved by changing $plotpixel(x, y)$ statements into $setpixel\ (x, y, pixmap[x\ mod\ m, y\ mod\ n])$, where $pixmap$ an m by n matrix that defines the fill pattern.

End of Ch2

Thank You!!