

Output Primitives

Output Primitives

- **Points**
- **Lines**
 - DDA Algorithm
 - Bresenham's Algorithm

A picture can be described as:

- The set of intensities for the pixel position in the display.

OR

- As a set of complex objects, such as trees and terrain or furniture and walls, positioned at specified coordinate locations within the scene.

Shapes and colors of the objects can be described:

- As internally with pixel arrays

OR

- Sets of basic geometric structures, such as straight line segments and polygon color areas.

The scene is converted by:

- Loading the pixel arrays into frame buffer.

OR

- Scan converting the basic geometric-structure specifications into pixel patterns.

- Output primitives - Basic geometric structures.
- A scene is described in such a way that the graphics programming packages provide functions to describe a scene in terms of basic geometric structures called output primitives and to group sets of o/p primitives into more complex structures
- Each output primitive is specified with the input coordinate data and other information about the way that object is to be displayed.
- Points and straight line segments are the simplest geometric components of pictures.

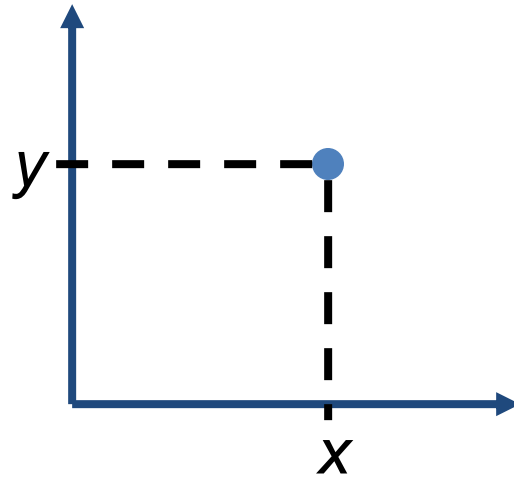
- Additional output primitives that can be used to construct a picture include circles and other conic sections, quadric surfaces, spline curves and surfaces, polygon color areas, and character strings.

Points- Point plotting is accomplished by converting a single coordinate position in an application program into appropriate operations for the output device in use.

- A random-scan 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.
- In raster scan system, point is plotted by setting the bit value corresponding to a specified screen position within the frame buffer to 1. Then, as the electron beam sweeps across each horizontal scan line, it emits a burst of electrons whenever a value of 1 is encountered in the frame buffer.

- **Single Coordinate Position**

- Set the bit value(color code) corresponding to a specified screen position within the frame buffer



setPixel (x, y)

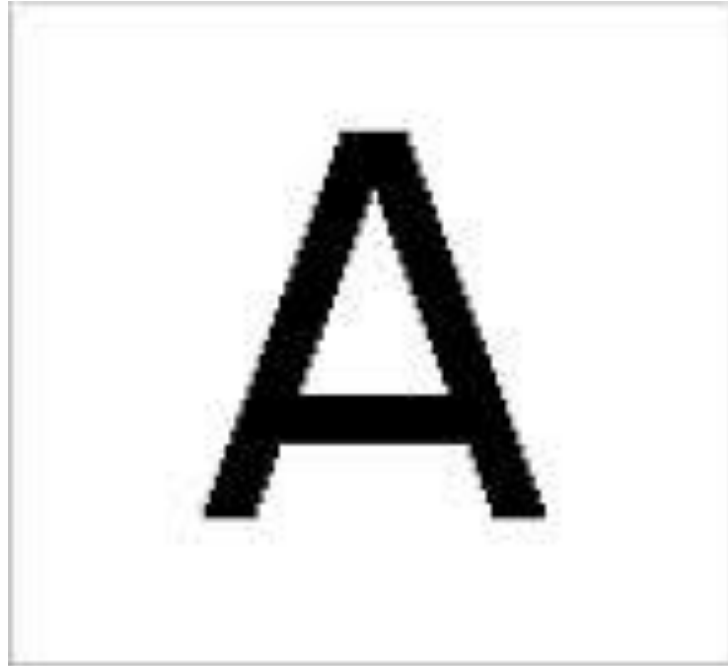
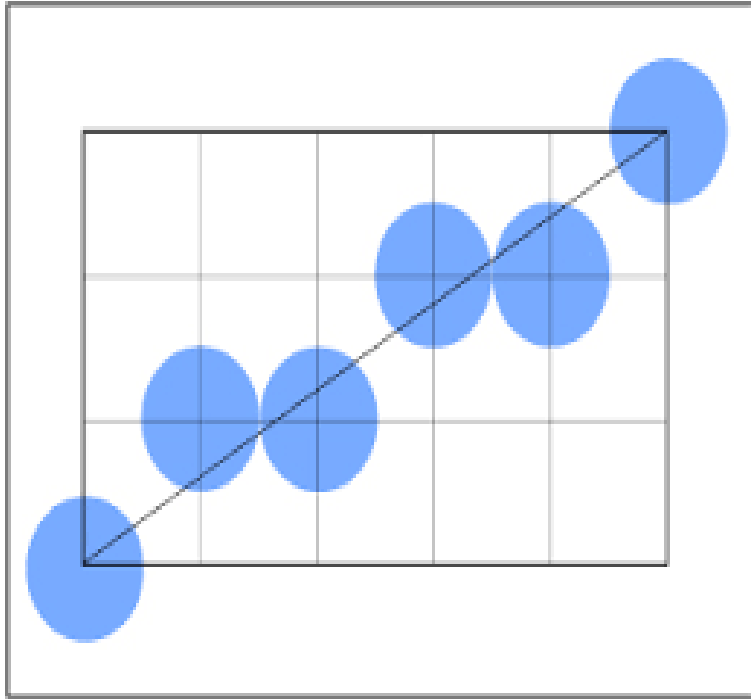
Line drawing is accomplished by calculating intermediate positions along the line path between two specified endpoint positions.

An output device is directed to fill in these positions between the endpoints.

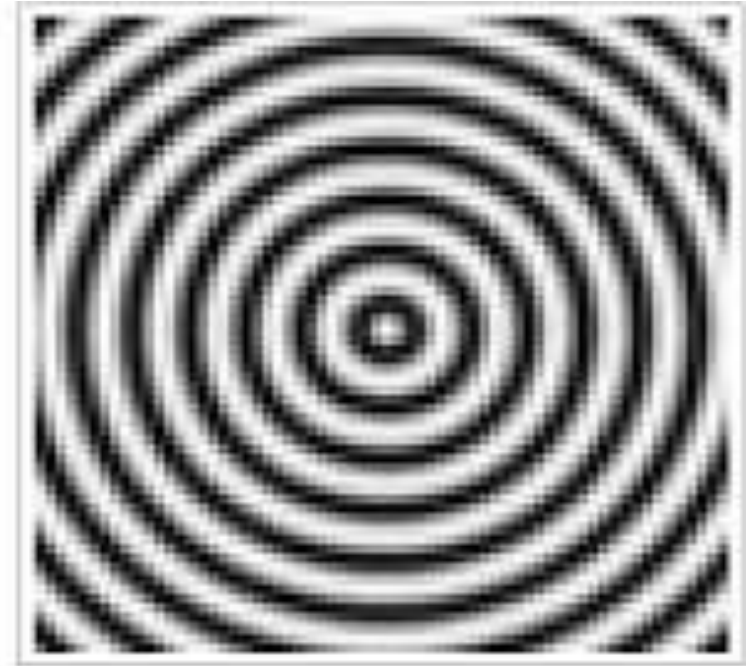
- For analog devices,
 - ✓ a straight line can be drawn smoothly from one end point to the other.
 - ✓ Linearly varying horizontal and vertical deflection voltages are generated that are proportional to the required changes in the x and y directions to produce the smooth line.

- For digital devices,
 - ✓ a straight line segment is displayed by plotting discrete points between two end points.
 - ✓ Discrete coordinate positions along the line path are calculated from the equation of line.
 - ✓ For a raster video display, the line color is loaded into the frame buffer at the corresponding pixel coordinates.
 - ✓ Reading from the frame buffer, the video controller plots the screen pixels.
 - ✓ Screen locations are referenced with integer values, so plotted positions may only approximate actual line positions between two endpoints.
 - ✓ A computed line positions, for example (10.48, 20.51) would be converted to pixel position (10,21).

- ✓ This rounding of coordinate values to integers causes lines to be displayed with a stairstep appearance “the jaggies”.



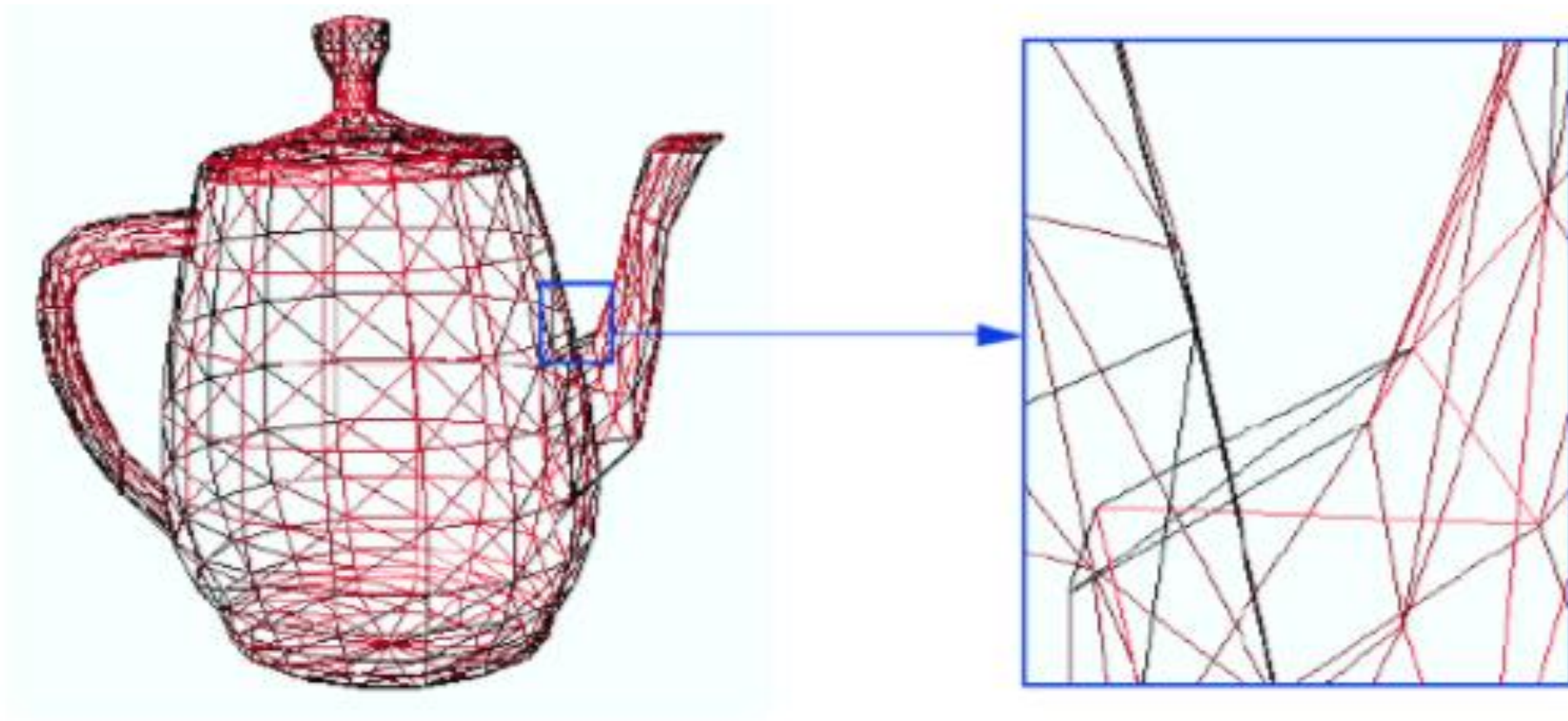
(a)



(b)

- ✓ The characteristic stairstep shape of raster lines is noticeable on systems with low resolution.
- ✓ This can be improved somewhat by displaying them on high-resolution systems.

Line Drawing Algorithms



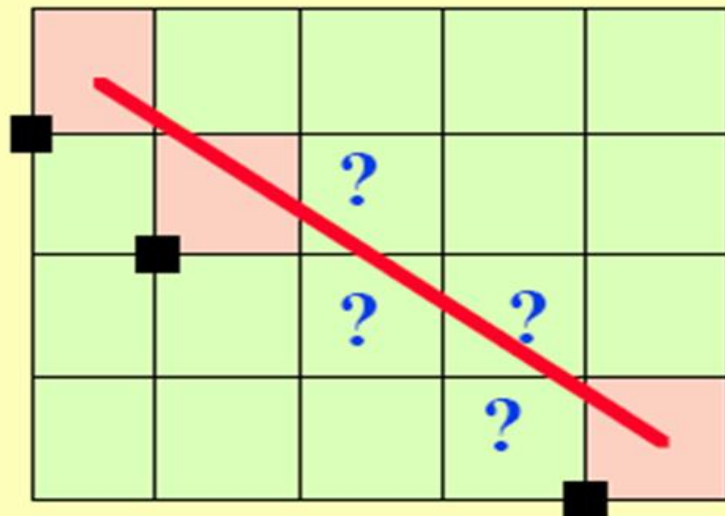
The lines of this object
appears continuous

However, they are
made of pixels

We are going to analyze how this process is achieved.

Some useful definitions

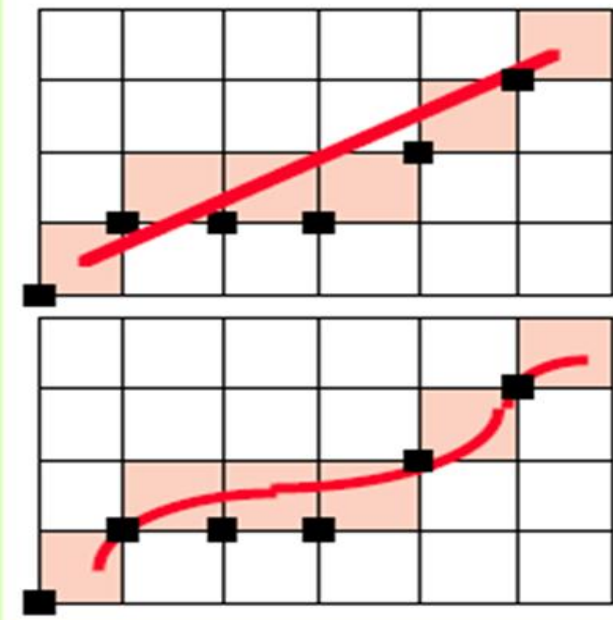
Rasterization: Process of determining which pixels provide the best approximation to a desired line on the screen.



Scan Conversion: Combination of rasterization and generating the picture in scan line order.

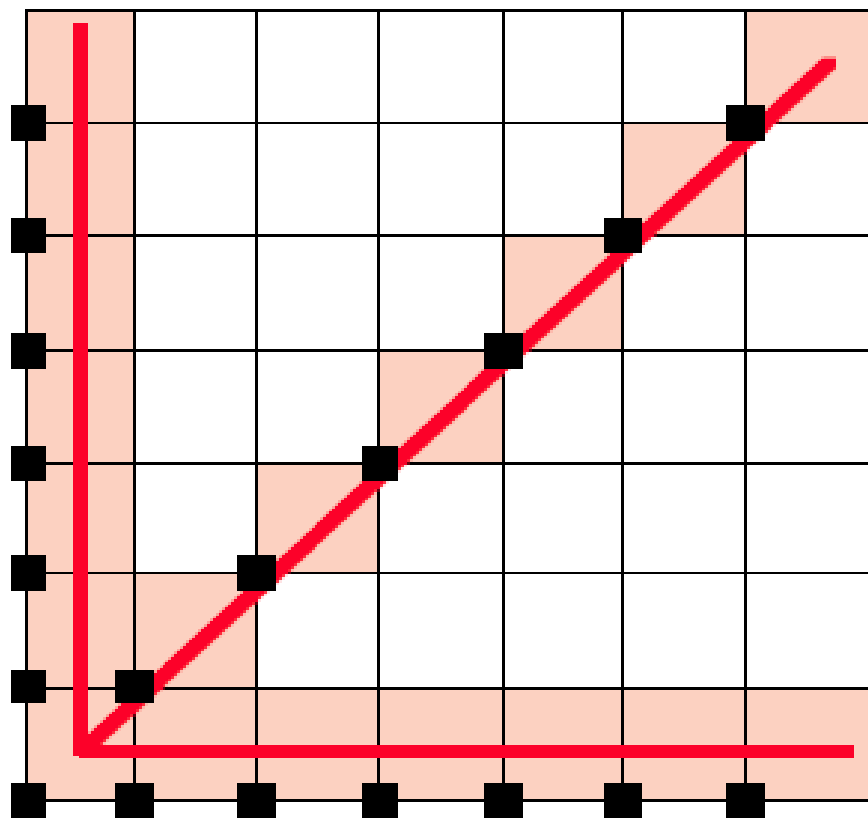
General requirements

- **Straight lines** must appear as **straight lines**.



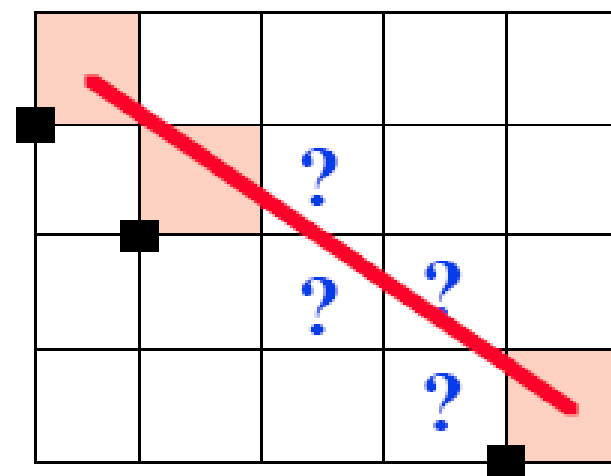
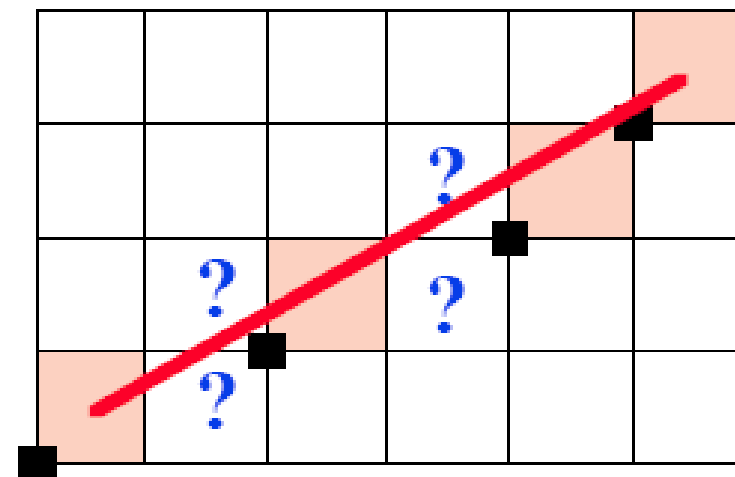
- They must **start** and **end accurately**
- Lines should have **constant brightness** along their length
- Lines should be drawn rapidly

For horizontal, vertical and 45° lines, the choice of raster elements is obvious. This lines exhibit constant brightness along the length:



© 2001 Andrés Iglesias. See: <http://personales.unican.es/iglesias>

For any other orientation the choice is more difficult:



The equation of a straight line is given by: $y = m.x + b$

Algorithm 1: Direct Scan Conversion

1. Start at the pixel for the left-hand endpoint x_l
2. Step along the pixels horizontally until we reach the right-hand end of the line, x_r
3. For each pixel compute the corresponding y value
4. round this value to the nearest integer to select the nearest pixel

```
x = xl;
```

```
while (x <= xr) {
```

```
    ytrue = m*x + b;
```

```
    y = Round (ytrue);
```

```
    PlotPixel (x, y);
```

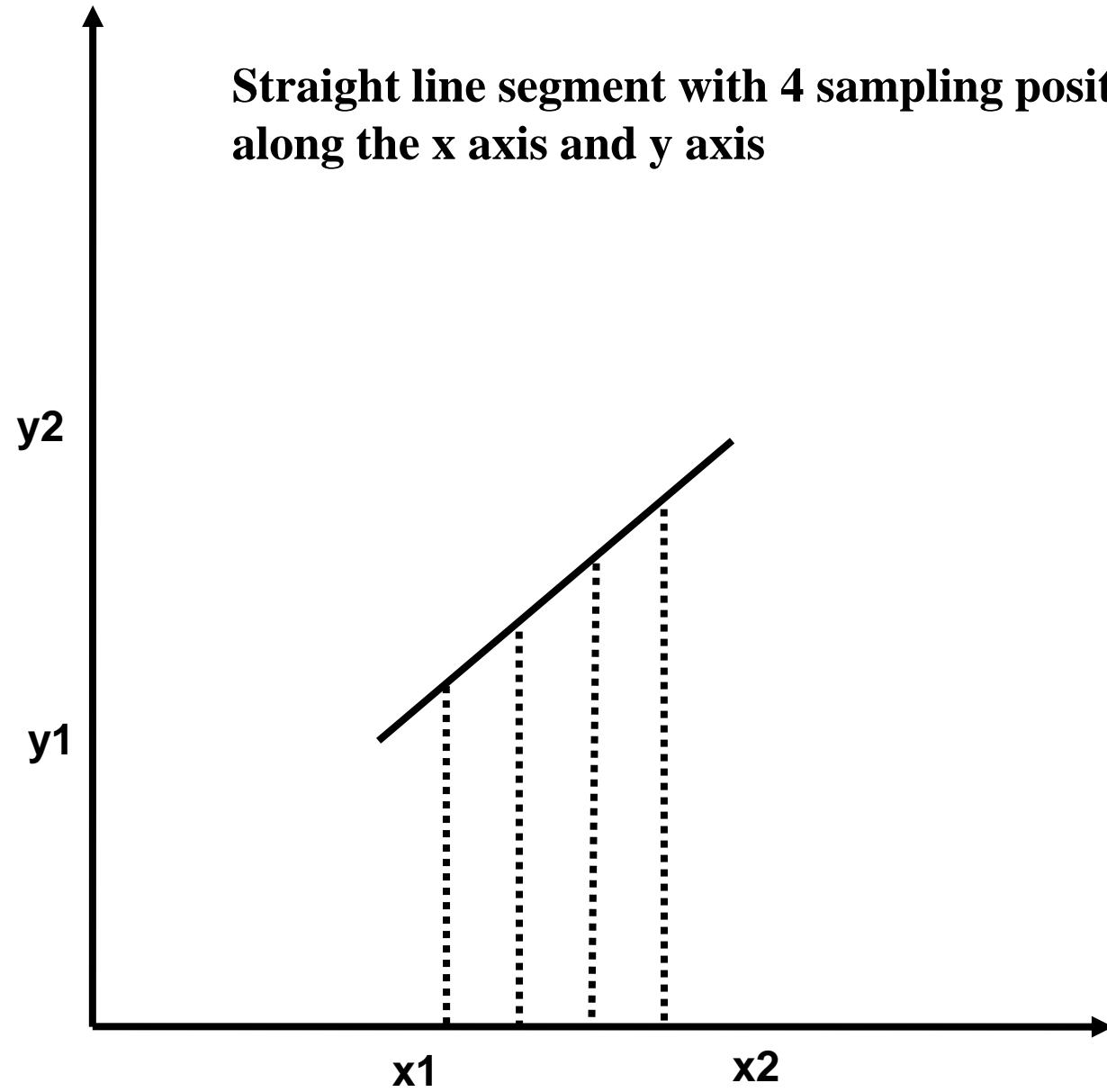
```
    /* Set the pixel at (x,y) on */
```

```
    x = x + 1;
```

```
}
```

The algorithm performs a **floating-point multiplication for every step in x** . This method therefore requires an enormous number of floating-point multiplications, and is therefore **expensive**.

**Straight line segment with 4 sampling positions
along the x axis and y axis**



- The Cartesian slope intercept equation for straight line is

$$y = m x + b$$

where m is slope of the line and b as the y intercept

- Let (x_1, y_1) & (x_2, y_2) be the two endpoints of the line, then

$$m = (y_2 - y_1) / (x_2 - x_1)$$

$$b = y_1 - m x_1$$

$$(y_2 - y_1) = m (x_2 - x_1)$$

$$m = (y_2 - y_1) / (x_2 - x_1) = \Delta y / \Delta x$$

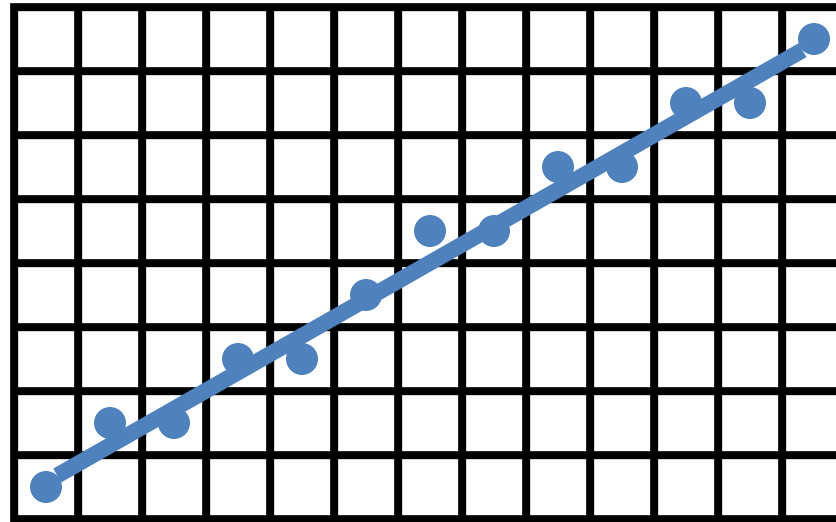
- For any given x interval Δx along a line, compute the corresponding y interval Δy as $\Delta y = m \Delta x$
- Similarly the x interval is obtained as

$$\Delta x = \Delta y / m$$

- The above equations form the basis for determining deflection voltages.
 - $|m| < 1$, Δx can be set proportional to a small horizontal deflection voltage and corresponding vertical deflection is set proportional to Δy
 - $|m| > 1$, Δy can be set proportional to a small vertical deflection voltage and corresponding horizontal deflection is set proportional to Δx
 - $|m| = 1$, $\Delta x = \Delta y$ and the horizontal and vertical deflection voltages are equal, which leads to a smooth line with slope m.

Digital Differential Analyzer (DDA) Algorithm

- **Intermediate Positions between Two Endpoints**
 - DDA, Bresenham's line algorithms



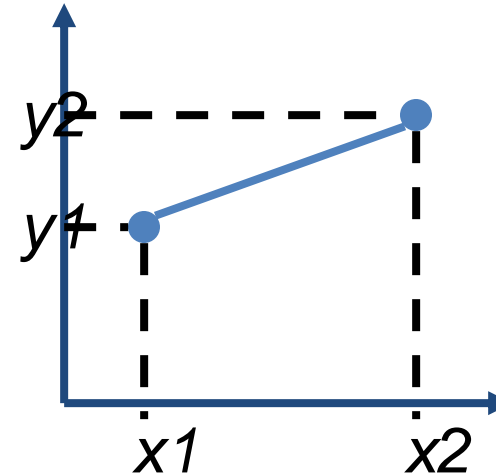
**Jaggies
= Aliasing**

DDA Algorithm

- **Digital Differential Analyzer**

$0 < \text{Slope} \leq 1$

- Unit x interval = 1

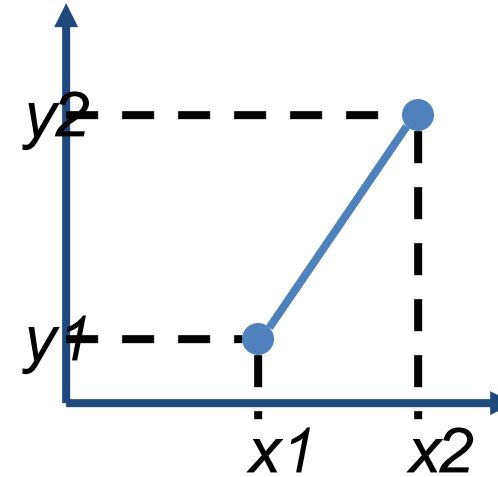


$$y_{k+1} = y_k + m$$

- **Digital Differential Analyzer**

Slope > 1

- Unit y interval = 1

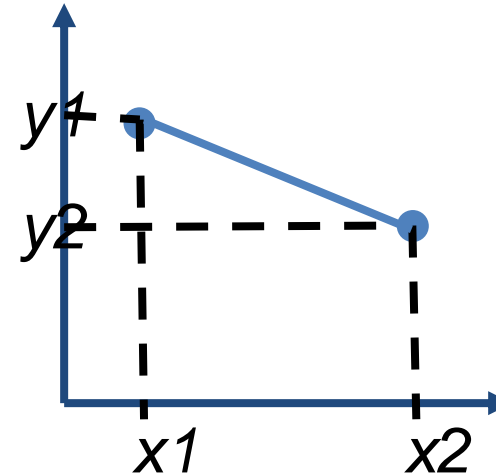


$$x_{k+1} = x_k + \frac{1}{m}$$

- **Digital Differential Analyzer**

$-1 \leq \text{Slope} < 0$

- Unit x interval = -1

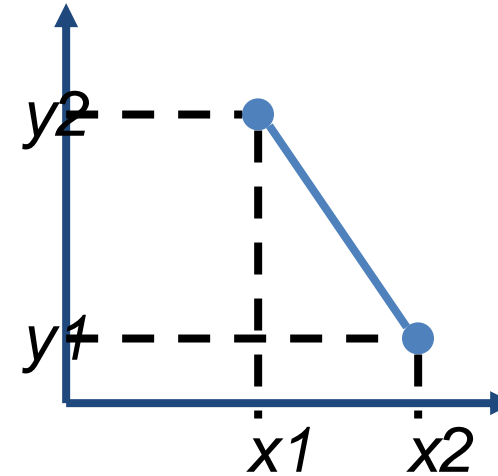


$$y_{k+1} = y_k - m$$

- **Digital Differential Analyzer**

Slope < -1

- Unit y interval = -1



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

Digital Differential Algorithm(DDA)

- Scan conversion algorithm based on calculating either Δx or Δy .
- Sample the line at unit intervals in one coordinate and determine the corresponding integer values nearest the line path for the other coordinate.

Case 1

Line with positive slope less than or equal to 1, we sample at unit x intervals ($\Delta x = 1$) and determine the successive y value as

$$Y_{k+1} = Y_k + m$$

$$\Delta y = m \Delta x$$

$$Y_{k+1} - Y_k = m \Delta x$$

$$Y_{k+1} - Y_k = m \quad (\Delta x = 1)$$

- **k** takes integer values **starting from 1** and **increases by 1** until the final point is reached.
- **m** can be any number between **0** and **1**.
- Calculated **y** values must be rounded off to the nearest integer.
- For lines with positive slope greater than 1, sample at unit **y** intervals (**$\Delta y = 1$**) and calculate each successive **x** value as

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

- The above eqns are based on the assumption that lines are processed from left to right. If the processing is reversed then

- $\Delta x = -1$ & $y_{k+1} = y_k - m$ \longrightarrow Slope >1
- $\Delta y = -1$ & $x_{k+1} = x_k - 1/m$ \longrightarrow Slope <1

Steps

1. **P1 (x_a, y_a) and P2 (x_b, y_b)** are the two end points.
2. Horizontal and vertical differences between the endpoint positions are computed & assigned to two parameters namely **dx** and **dy** .
3. The difference with greater magnitude determines the '**value**' of increments to be done. That means the number of times sampling has to be done. This value is assigned to a parameter called '**steps**'.

4. Starting from the 1st pixel ,we determine the offset needed at each step to generate the next pixel position along the line path. Call the offset value as $x_{\text{increment}}$ and $y_{\text{increment}}$.

The starting points are x_a and y_a .

Assign $x = x_a$ and $y = y_a$

$$x = x + x_{\text{incr}} \ \& \ y = y + y_{\text{incr}}$$

5. Loop through the process steps times, till the last point x_b, y_b is reached.



Steps for DDA Algorithm

Step 1: Accept as input the 2 end point pixel position.

$P1(x_a, y_a)$

$P2(x_b, y_b)$

Step 2: Horizontal and vertical differences between the endpoint positions are computed & assigned to two parameters namely dx and dy.

$dx = x_b - x_a$

$dy = y_b - y_a$

Step 3: If $\text{abs}(\text{dx}) > \text{abs}(\text{dy})$

$\text{steps} = \text{abs}(\text{dx})$

else

$\text{steps} = \text{abs}(\text{dy})$

Step 4: $\text{x}_{\text{incr}} = \text{dx}/\text{steps}$

$\text{y}_{\text{incr}} = \text{dy}/\text{steps}$

(ie $\text{dy}/\text{dx} = \text{m}$)

Assign $\text{x} = \text{x}_a$ and $\text{y} = \text{y}_a$

$\text{x} = \text{x} + \text{x}_{\text{incr}}$ & $\text{y} = \text{y} + \text{y}_{\text{incr}}$

Step 5: Loop through the process steps times.

DDA Example

- Suppose we want to draw a line starting at pixel (2,3) and ending at pixel (12,8).
- What are the values of the variables x and y at each timestep?
- What are the pixels colored, according to the DDA algorithm?

$$\text{numsteps} = 12 - 2 = 10$$

$$\text{xinc} = 10/10 = 1.0$$

$$\text{yinc} = 5/10 = 0.5$$

t	x	y	R(x)	R(y)
0	2	3	2	3
1	3	3.5	3	4
2	4	4	4	4
3	5	4.5	5	5
4	6	5	6	5
5	7	5.5	7	6
6	8	6	8	6
7	9	6.5	9	7
8	10	7	10	7
9	11	7.5	11	8
10	12	8	12	8

Advantages of DDA

- It calculates the pixel positions faster than the calculations performed by using the equation $y=mx+b$
- Multiplication is eliminated as the x and y increments are used to determine the position of the next pixel on a line.

Disadvantages of DDA

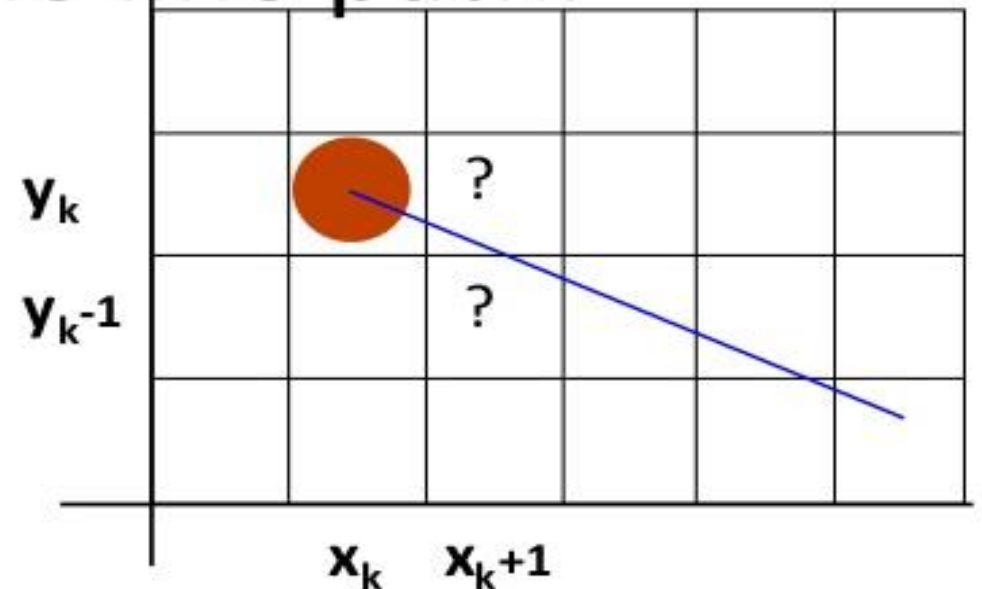
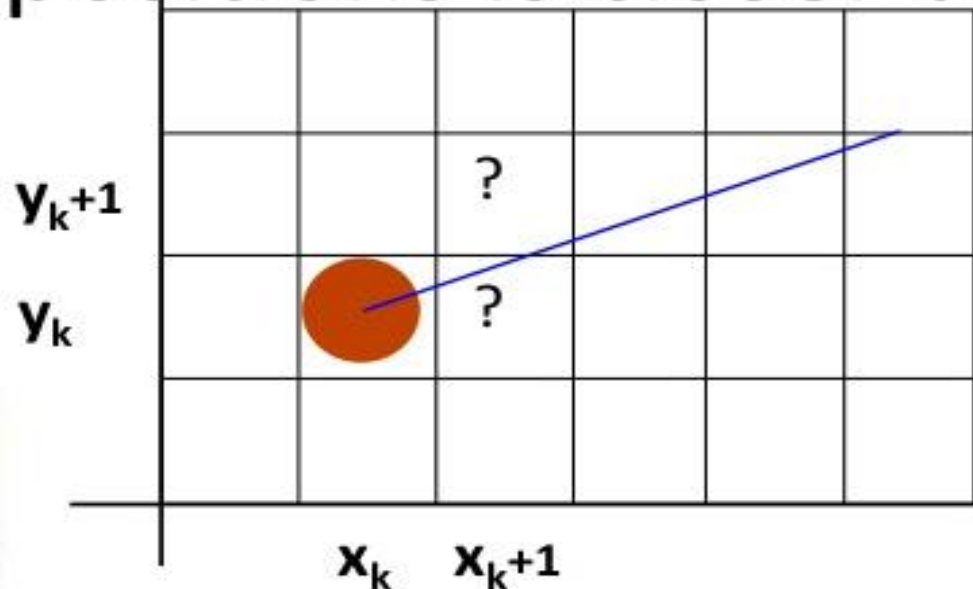
- The rounding and floating point operations are time consuming.
- The round off error which results in each successive addition leads to the drift in pixel position which is already calculated.

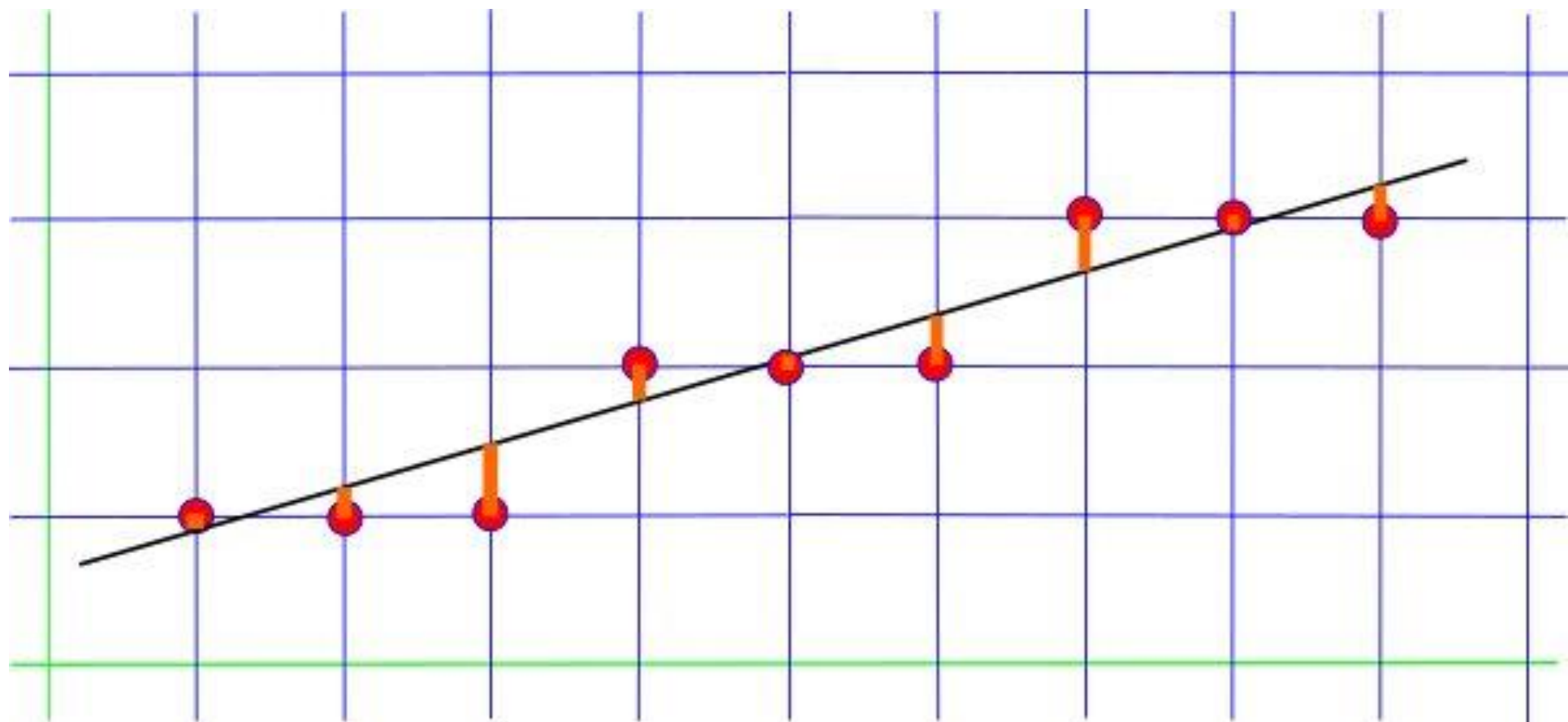
Exercise:

Draw a line using DDA Algorithm between (10, 100) and (50, 20)

Bresenham's Line Algorithm

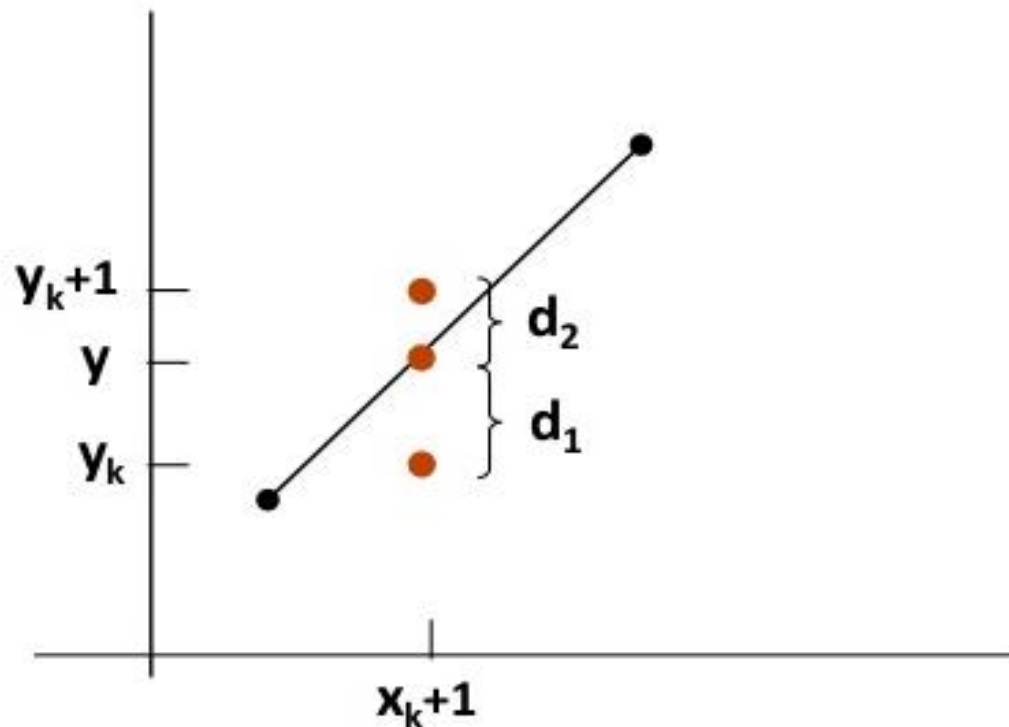
- Developed by Bresenham.
- Efficient than DDA:
 - Only integer calculations are involved.
- Determines which of two possible pixel positions is closer to the line path.





Method:

- Test the sign of an *integer parameter* :
 - Whose value is proportional to the difference between the separations of the two pixel positions from the actual line path.



If $d_1 < d_2$:

- $d_1 - d_2 < 0$
- y_k is closer to the line.
- Plot y_k

Else:

- $d_1 - d_2 > 0$
- y_{k+1} is closer to the line.
- Plot y_{k+1}

If $0 < m < 1$:

If $x_k + 1$ is on the line $y = mx + b$,

$$\Rightarrow y = m(x_k + 1) + b$$

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

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

$$\Rightarrow d_1 - d_2 = 2m(x_k + 1) - 2y_k + 2b - 1$$

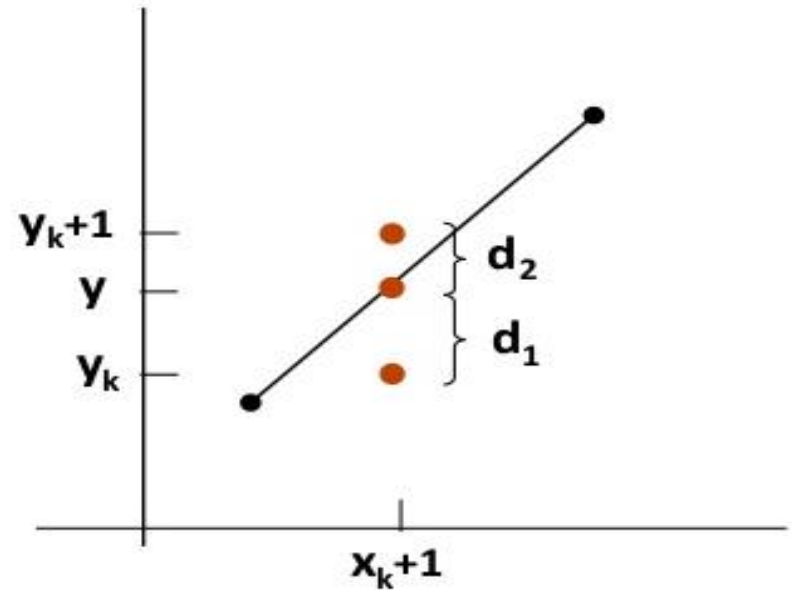
$$= 2 \frac{dy}{dx} (x_k + 1) - 2y_k + 2b - 1$$

$$dx(d_1 - d_2) = 2dy(x_k + 1) - dx2y_k + dx(2b - 1)$$

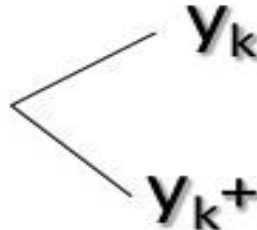
$$dx(d_1 - d_2) = 2dy.x_k - 2dx.y_k + c \text{ ; where } c = 2dy + dx(2b - 1)$$

Let, $p_k = dx(d_1 - d_2)$; Sign of p_k is the same as the sign of $d_1 - d_2$ since $dx > 0$

$$\Rightarrow p_k = 2dy.x_k - 2dx.y_k + c$$



Now:

– y_{k+1}  y_k
 y_{k+1}^+ ; will be determined by the sign of p_k .

– Write:

$$p_{k+1} = 2dy.x_{k+1} - 2dx.y_{k+1} + c$$

$$p_{k+1} = 2dy(x_k + 1) - 2dx.y_{k+1} + c; \because x_{k+1} = x_k + 1$$

$$\Rightarrow p_{k+1} - P_k = 2dy - 2dx(y_{k+1} - y_k)$$

$$p_{k+1} = P_k + 2dy - 2dx(y_{k+1} - y_k)$$

- Term($y_{k+1} - y_k$) is either 0 or 1, depending on the sign of p_k .
- If $p_k < 0$ then $(y_{k+1} - y_k) = 0 \Rightarrow y_{k+1} = y_k$
 - if $p_k > 0$ then $(y_{k+1} - y_k) = 1 \Rightarrow y_{k+1} = y_k + 1$

Bresenham Line Algorithm

(For +ve slope and $0 < m < 1$)

BRESENHAM'S LINE DRAWING ALGORITHM (for $|m| < 1.0$)

1. Input the two line end-points, storing the left end-point in (x_0, y_0)
2. Plot the point (x_0, y_0)
3. Calculate the constants dx , dy , $2dy$, and $(2dy - 2dx)$ and get the first value for the decision parameter as:

$$p_0 = 2dy - dx$$

4. At each x_k along the line, starting at $k = 0$, perform the following test. If $p_k < 0$, the next point to plot is $(x_k + 1, y_k)$ and:

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

Otherwise, the next point to plot is (x_k+1, y_k+1) and:

$$p_{k+1} = p_k + 2dy - 2dx$$

5. Repeat step 4 $(dx - 1)$ times

Bresenham Line Algorithm

(For +ve slope and $m > 1$)

BRESENHAM'S LINE DRAWING ALGORITHM (for $|m| > 1.0$)

1. Input the two line end-points, storing the left end-point in (x_0, y_0)
2. Plot the point (x_0, y_0)
3. Calculate the constants dx , dy , $2dx$, and $(2dx - 2dy)$ and get the first value for the decision parameter as:

$$p_0 = 2dx - dy$$

4. At each y_k along the line, starting at $k = 0$, perform the following test. If $p_k < 0$, the next point to plot is $(x_k, y_k + 1)$ and:

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

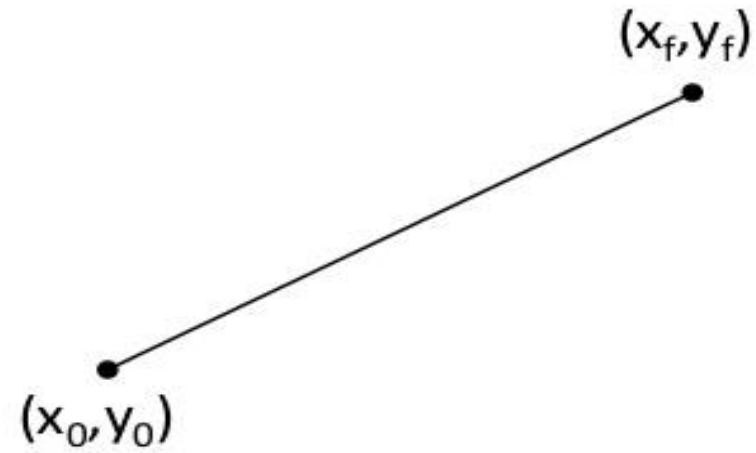
Otherwise, the next point to plot is (x_k+1, y_k+1) and:

$$p_{k+1} = p_k + 2dx - 2dy$$

5. Repeat step 4 $(dy - 1)$ times

- Implementation:

```
setPixel (x0,y0)  
dx=xf-x0; dy=yf-y0  
dx2=2*dx; dy2=2*dy  
P0 = dy2 - dx  
k = 0 ; pk = p0 ; x=x0 ; y=y0  
Do while (x<xf)  
    { x=x++;  
      if (pk<0)  
          pk=pk+dy2  
      else  
          { pk=pk+dy2-dx2  
            y=y++ }  
      setPixel (x,y)  
    }
```



– Apply BLA to $(x_1, y_1) = (20, 10)$, $(x_2, y_2) = (30, 18)$.

$$dx = x_2 - x_1 = 30 - 20 = 10$$

$$dy = y_2 - y_1 = 18 - 10 = 8$$

$$dx2 = 2 * 10 = 20$$

$$dy2 = 2 * 8 = 16$$

$$p_0 = dy2 - dx = 16 - 10 = 6$$

$$dy2 - dx2 = 16 - 20 = -4$$

– If $p_k < 0$

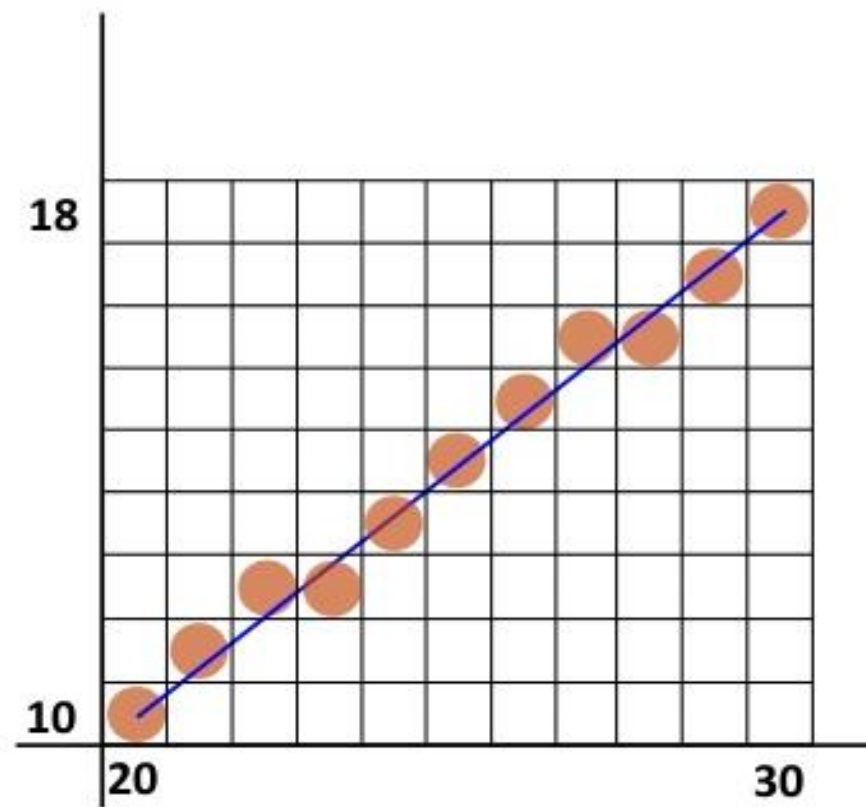
$$p_k = p_k + dy2 = p_k + 16$$

else

$$p_k = p_k + dy2 - dx2 = p_k - 4$$

k	p_k	x_{k+1}	y_{k+1}
0	$6 > 0$	21	11
1	$2 > 0$	22	12
2	$-2 < 0$	23	12
3	$14 > 0$	24	13
4	$10 > 0$	25	14
5	$6 > 0$	26	15
6	$2 > 0$	27	16
7	$-2 < 0$	28	16
8	$14 > 0$	29	17
9	$10 > 0$	30	18

k	p_k	x_{k+1}	y_{k+1}
0	$6 > 0$	21	11
1	$2 > 0$	22	12
2	$-2 < 0$	23	12
3	$14 > 0$	24	13
4	$10 > 0$	25	14
5	$6 > 0$	26	15
6	$2 > 0$	27	16
7	$-2 < 0$	28	16
8	$14 > 0$	29	17
9	$10 > 0$	30	18



DDA vs. Bresenham Algorithm

- DDA is the simplest line drawing algorithm
 - Not very efficient
 - Round operation is expensive
- Optimized algorithms typically used.
 - Integer DDA
- Bresenham algorithm
 - Incremental algorithm: current value uses previous value
 - Integers only: avoid floating point arithmetic

Exercise:

Draw a line using Bresenham's Algorithm between (10, 20) and (50, 100)

Draw a line using Bresenham's Algorithm between (10, 20) and (50, 40)