Introduction to Computer Graphics

Module 1 – Part2

SUPRIYA
Dept of CSE
Sahyadri College of Engineering & Management

OpenGL Line-Attribute Functions

- **Line Width**

  glLineWidth (width);

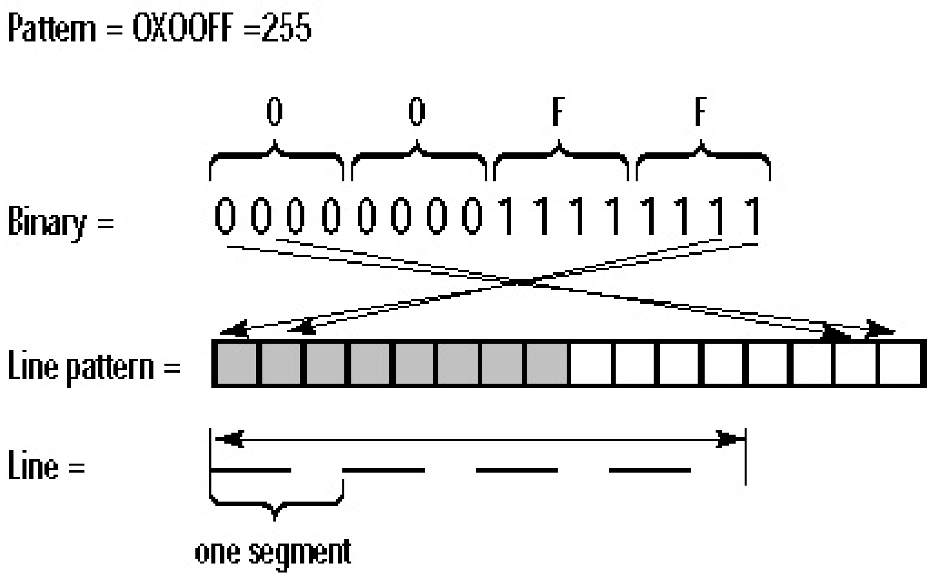
  - **width is floating-point value rounded to the nearest nonnegative integer**

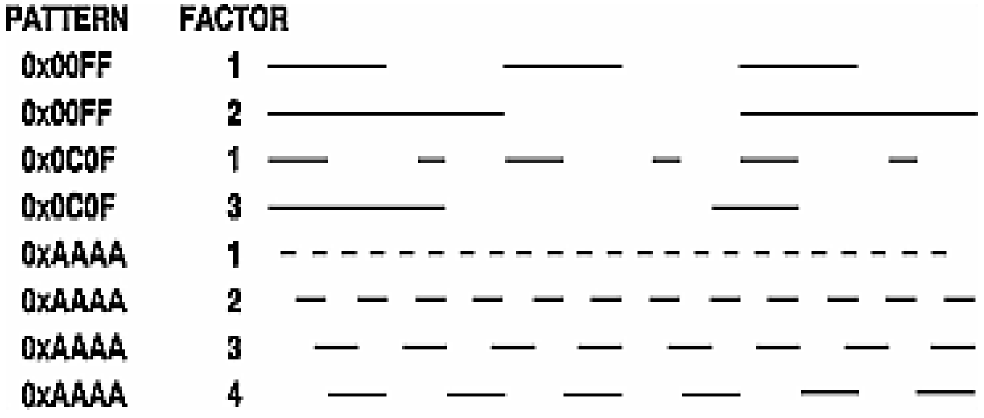
- **Line Style**

  glLineStipple (repeatFactor, pattern)

  - **Sets the current display style for lines**
  - **By default, a straight-line segment is displayed as a solid line**

- **pattern** is used to reference a 16-bit integer that describes how the line should be displayed.
  - A1 bit in the pattern denotes an "on" pixel position, and a 0 bit indicates an "off" pixel position.
  - The pattern is applied to the pixels along the line path starting with the low-order bits in the pattern.
  - The default pattern is 0xFFFF (each bit position has a value of 1), which produces a solid line.
- **repeatFactor** specifies how many times each bit in the pattern is to be repeated before the next bit in the pattern is applied.
  - The default repeat value is 1.





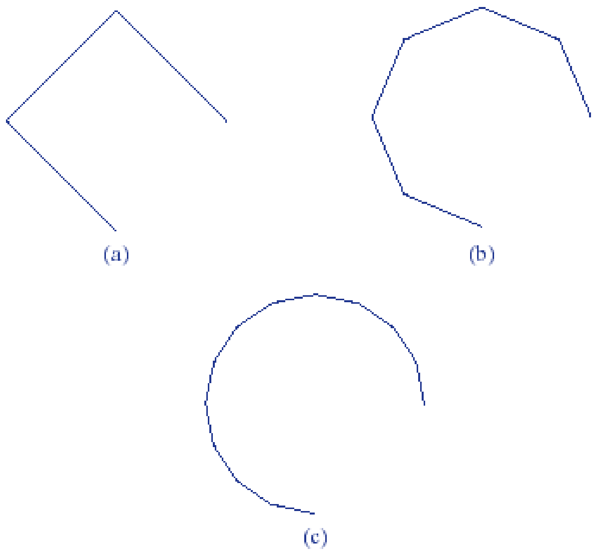
- **glLineStipple(1, 0x3F07);**
  - **Pattern 0x3F07 translates to 0011111100000111**
  - **Line is drawn with 3 pixels on, 5 off, 6 on, and 2 off**
- **glLineStipple(2, 0x3F07);**
  - **Factor is 2**
  - **Line is drawn with 6 pixels on, 10 off, 12 on, and 4 off**

- **glEnable (GL_LINE_STIPPLE);**
  - **activate the line-style feature of OpenGL**
- **glDisable (GL_LINE_STIPPLE);**
  - **replaces the current line-style pattern with the default pattern (solid lines).**

Curve Attributes

- Curves can be displayed with varying colors, widths, dot-dash patterns, and available pen or brush options
- Routines for generating basic curves, such as circles and ellipses, are not included as primitive functions in the OpenGL core library
- curved segments can be drawn using splines.
  - These can be drawn using OpenGL evaluator functions, or by using functions from the OpenGL Utility (GLU) library which draw b-splines.
    - Using rational B-splines, circles, ellipses, and other two-dimensional quadrics can be displayed
    - GLU library has routines for three-dimensional quadrics, such as spheres and cylinders
    - A curve can also be drawn using polyline

- **Curves can be generated by approximating using a polyline**



A circular arc approximated with (a) three straight-line segments,

(b) six line segments, and

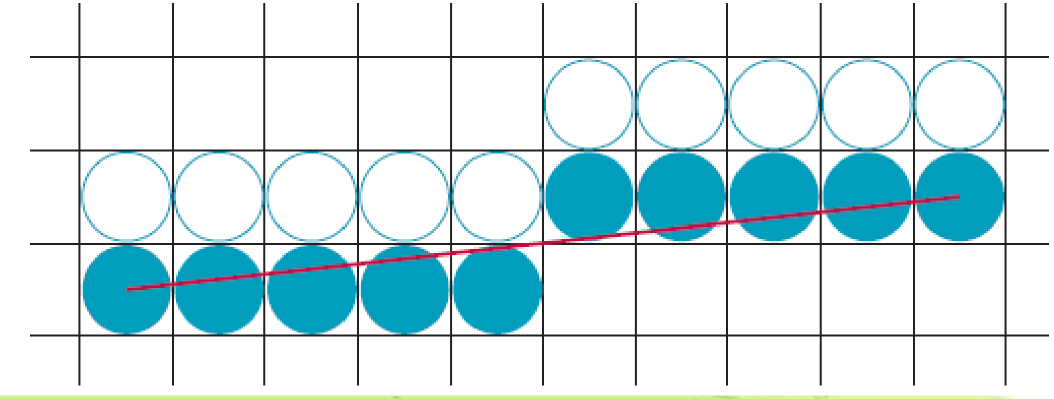
(c) twelve line segments.

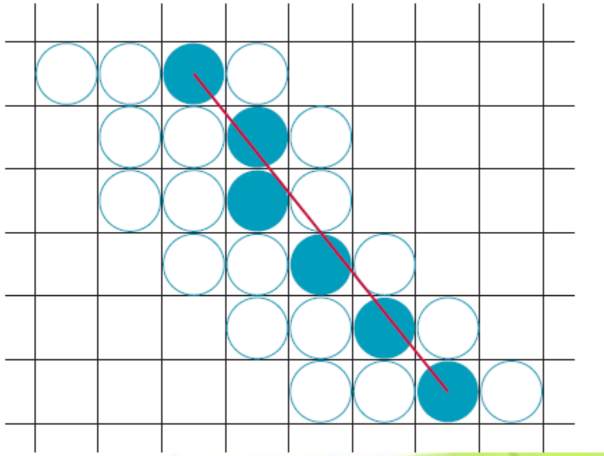
- **Curves can also be generated by defining curve generation functions based on the algorithms**

Line-Drawing Algorithms

- A straight-line segment in a scene is defined by the coordinate positions for the endpoints of the segment.
- To display the line on a raster monitor, the graphics system must first project the endpoints to integer screen coordinates and determine the nearest pixel positions along the line path between the two endpoints
  - A line position of (10.48, 20.51) is converted to (10, 21).
- Then 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

- **The rounding of coordinate values to integers causes all but horizontal and vertical lines to be displayed with a stair-step appearance (known as "the jaggies")**
  - **The stair-step shape of raster lines is noticeable on systems with low resolution**

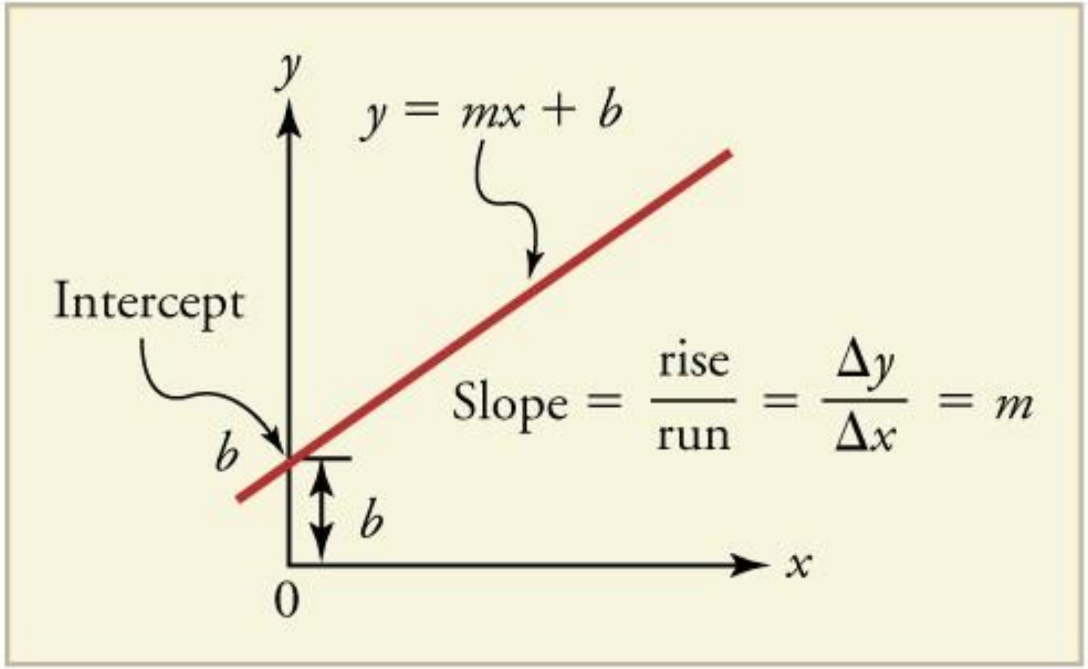




Line Equations

- The pixel positions along a straight-line path can be derived from the geometric properties of the line.

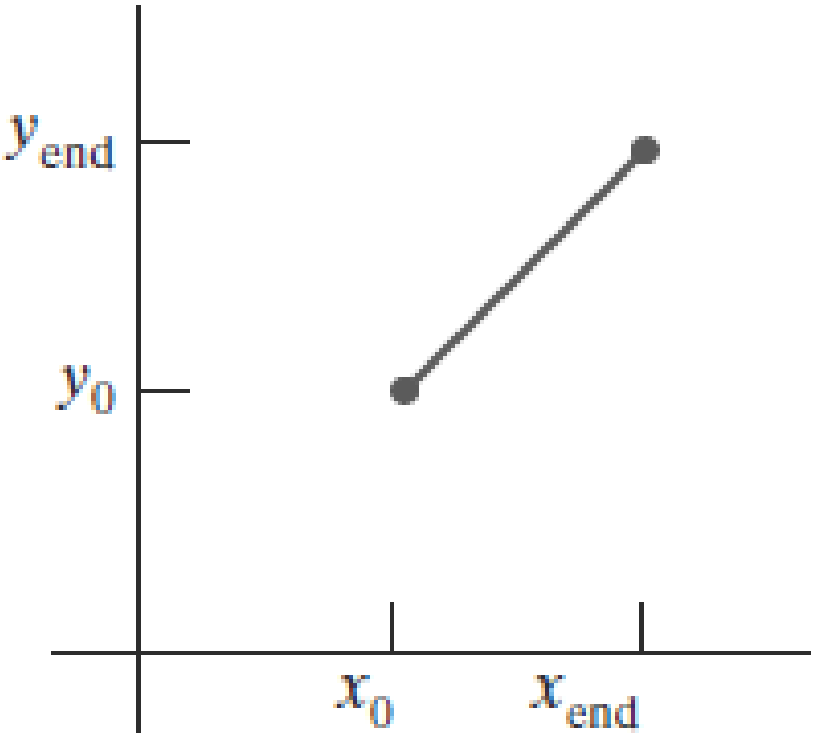
- The Cartesian slope-intercept equation for a straight line is
  - $y=mx+b$
    - With m as the slope of the line and b as the y intercept



- Given that the two endpoints of a line segment are specified at positions (x_0, y_0) and (x_{end}, y_{end}), then

$$m = \frac{y_{end} - y_o}{x_{end} - x_o}$$

$$b = y_0 - m x_0$$



- For any given x interval δx along a line, the corresponding y interval, δy can be computed as

$$\delta y = m.\delta x$$

- Similarly the x interval δx, for any corresponding y interval δy is given by

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

DDA Algorithm

- The **digital differential analyzer (DDA)** is a scan-conversion line algorithm based on calculating either **δy** or **δx**.
- A line is sampled at unit intervals in **one coordinate** and the corresponding **integer values** nearest the line path are determined for the **other coordinate**.

- **Let assume that lines are to be processed from the left endpoint to the right endpoint**
- Considering a line with positive slope, if the slope<=1, sample at unit x intervals ($\delta x=1$) and compute successive y values as

$$y_{k+1} = y_k + m \quad \text{and} \quad x_{k+1} = x_k + 1$$

  - k takes integer values starting from 0, for the first point, and increases by 1 until the final endpoint is reached

- **m** can be any real number between 0.0 and 1.0
- Each calculated y value must be rounded to the nearest integer corresponding to a screen pixel position in the x column
- For lines with a positive slope greater than 1.0, the roles of x and y are reversed.
  - Sample at unit y intervals (δy = 1) and calculate consecutive x values as

$$x_{k+1} = x_k + \frac{1}{m} \quad \text{and} \quad y_{k+1} = y_k + 1$$

- **When the lines are processed from right to left**
  - If $m<=1$
    - $\delta x=-1$
    - $y_{k+1} = y_k + m$ and $x_{k+1} = x_k - 1$
  - If $m>1$
    - $\delta y=-1$

$$x_{k+1} = x_k - \frac{1}{m} \quad \text{and} \quad y_{k+1} = y_k - 1$$

DDA Algorithm

Steps

1. **Get the input of two end points (X_0,Y_0) and (X_1,Y_1).**
2. **Calculate the difference between two end points.**
   - $dx = X_1 - X_0$
   - $dy = Y_1 - Y_0$
3. **Based on the calculated difference in step-2, identify the number of steps to put pixel.**
   - This value is the number of pixels that must be drawn beyond the starting pixel
   - If dx > dy, then more steps are needed in x coordinate; otherwise in y coordinate.

```
if dx > dy
    Steps = absolute(dx);
else
   Steps = absolute(dy);
```

4. Calculate the increment in x coordinate and y coordinate.

   Xincrement = dx /   steps;

   Yincrement = dy /   steps;

   - if Steps=dy, then Xincrement =1/m;
   - if Steps=dx, then Yincrement =m;

5. **Put the pixel by successfully incrementing x and y coordinates accordingly and complete the drawing of the line.**

```
x=Xo and y=Yo
for(int v=0; v < Steps; v++)
 {
    x = x + Xincrement;
   y = y + Yincrement;
    putpixel(Round(x), Round(y));
}
```

An Example

1. Let the end points of line are (X_0, Y_0) = (10,20) & (X_1, Y_1) =(20,25)
2. Find dx and dy
   - dx= 20-10=10 and dy=25-20=5
3. Find the number of steps
   - dx>dy, so steps=dx=10
4. Calculate xincr and yincr
   - xincr=10/10 =1 and yincr=5/10=0.5
5. Draw the starting pixel at position (x0, y0), and Compute the next pixel's position and draw it.
   - x=X_0+xincr and y=Y_0+yncr

x=10+1=11, y=20+0.5=20.5 put pixel at (11,21)

x=11+1=12, y=20.5+0.5=21 put pixel at (12,21)

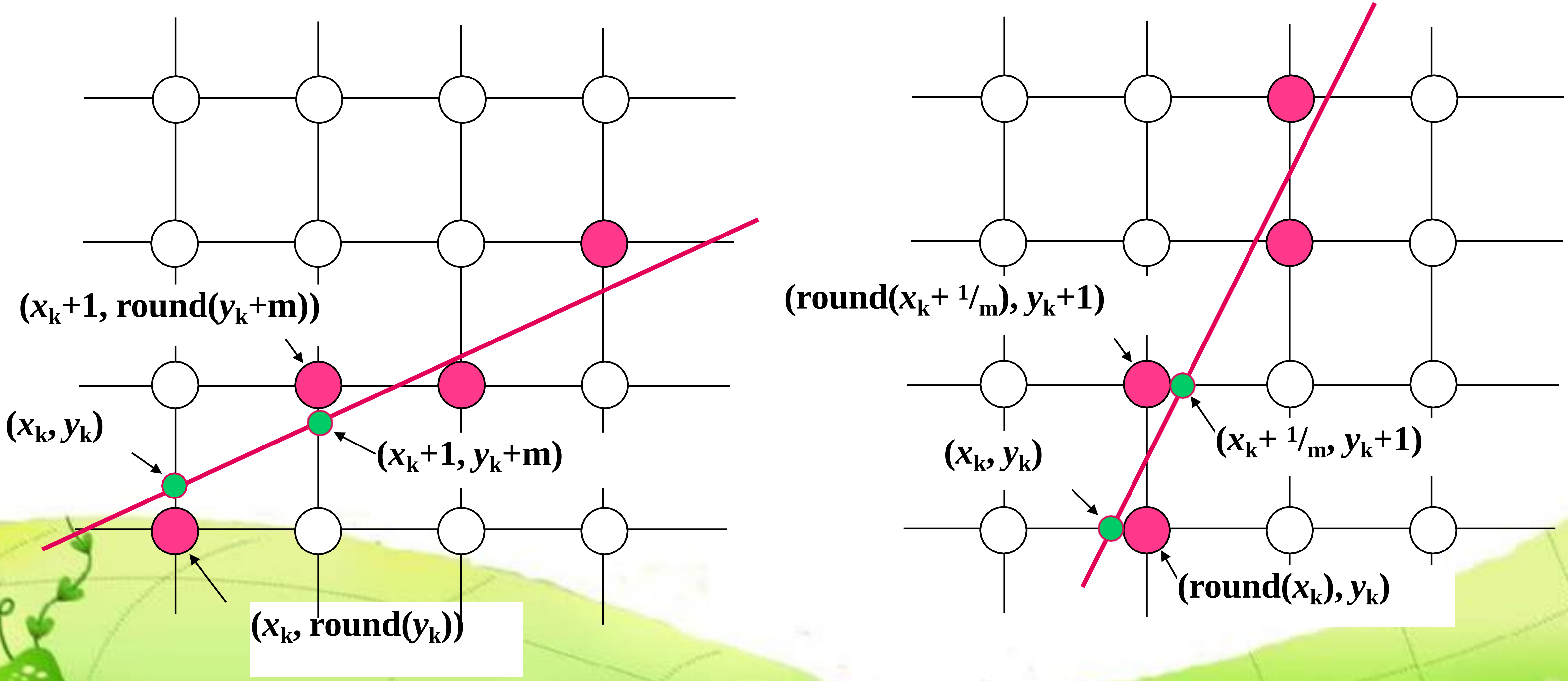
x=12+1=13, y=21+0.5=21.5 put pixel at (13,22)

.

.

.

X=19+1=20, y=24.5+0.5=25 put pixel at (20,25)



$(x_k+1, \text{round}(y_k+m))$
(x_k, y_k)
(x_k+1, y_k+m)
$(x_k, \text{round}(y_k))$
$(\text{round}(x_k+ {}^1/_m), y_k+1)$
(x_k, y_k)
$(x_k+ {}^1/_m, y_k+1)$
$(\text{round}(x_k), y_k)$

Points to remember

- If the **magnitude of dx is greater than the magnitude of dy** and **x0 is less than xEnd**, the values for the increments in the x and y directions are **1 and m**, respectively.

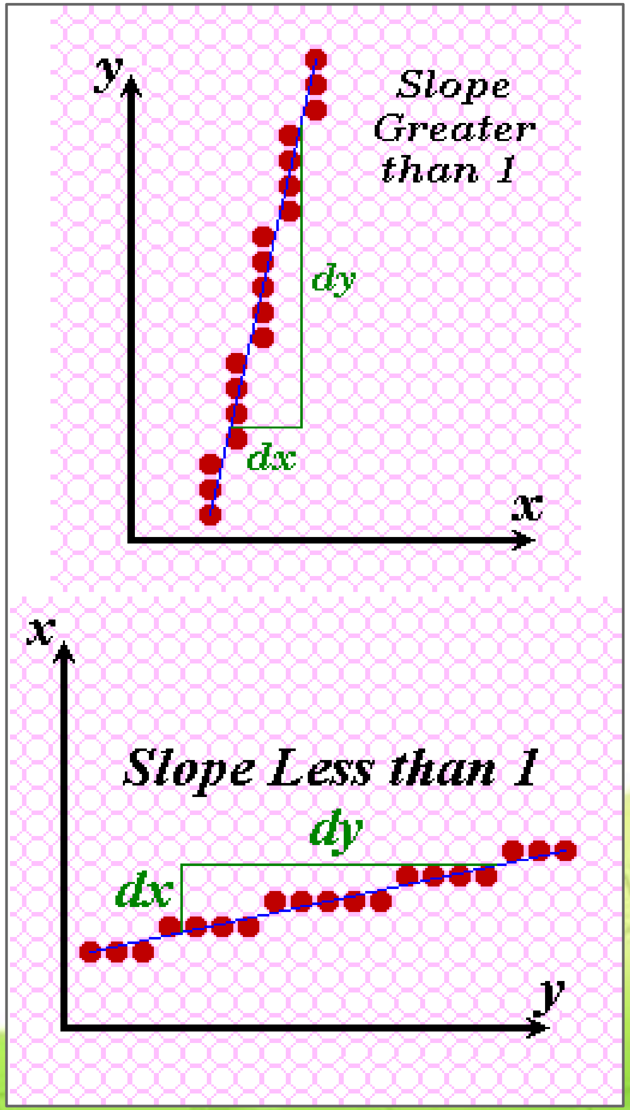
- If the **greater change is in the x direction**, but **x0 is greater than xEnd**, then the decrements **−1 and −m** are used to generate each new point on the line.

- Otherwise, an unit increment (or decrement) is used in the y direction and an x increment (or decrement) of 1 / m

```
#include <stdlib.h>
#include <math.h>
inline int round (const float a) { return int (a + 0.5); }
void lineDDA (int x0, int y0, int xEnd, int yEnd)
{
    int dx = xEnd -  x0, dy = yEnd -  y0, steps, k;
    float xIncrement, yIncrement, x = x0, y = y0;
    if (fabs (dx) > fabs (dy))
        steps = fabs (dx);
    else
        steps = fabs (dy);
```

```
xIncrement = float (dx) /   float (steps);
    yIncrement = float (dy) /   float (steps);
    setPixel (round (x), round  (y));
    for (k = 0;  k < steps; k++) {
        x += xIncrement;
        y +=  yIncrement;
    setPixel  (round (x), round  (y));         }
}
```

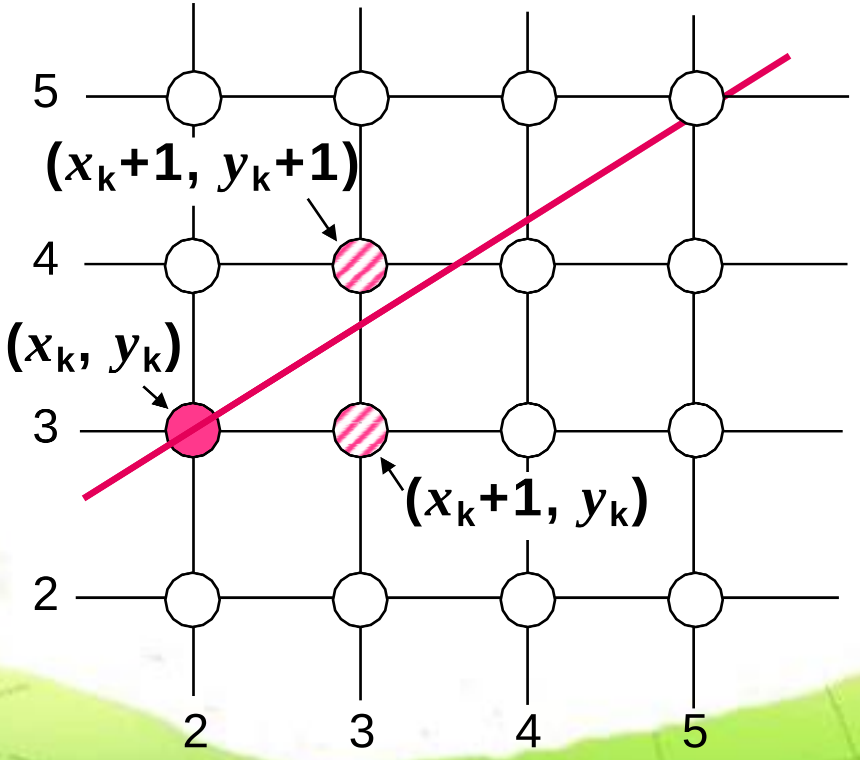
- **Advantages**
  - **Faster calulation of pixel positions**
    - **No longer any multiplications involved**
- **Disadvantages**
  - **The rounding operation & floating point arithmetic are time consuming procedures.**
  - **Round-off error can cause the calculated pixel position to drift away from the true line path for long line segment.**



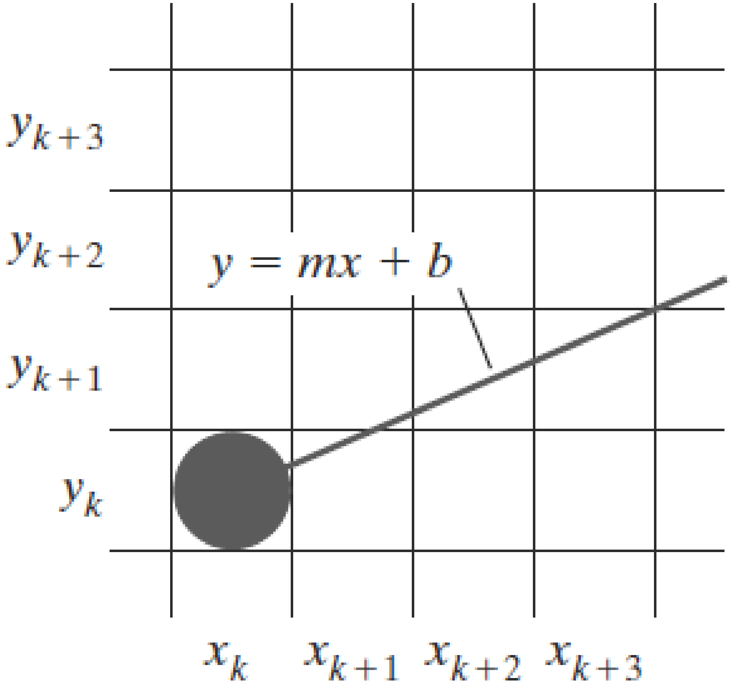
Bresenham’s Line Algorithm

- **Accurate and efficient raster line-generating algorithm**
- **Can be easily adapted to display circles and other curves**
- **it uses only integer calculations**

- **Move across the x axis in unit intervals and at each step choose between two different y coordinates**
  - Eg: from position (2, 3) we have to choose between (3, 3) and (3, 4)

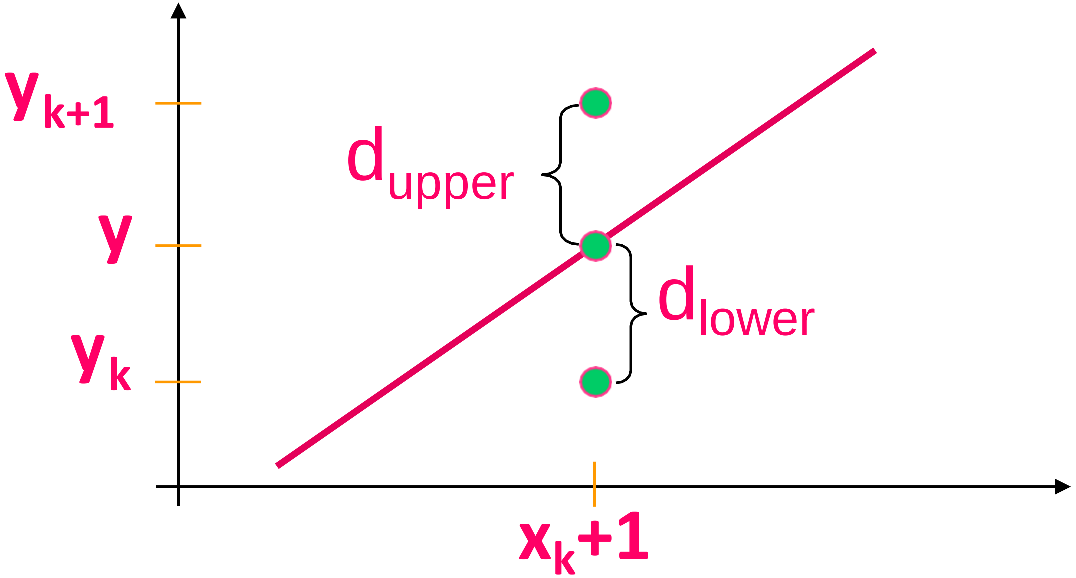


- scan-conversion process for lines with positive slope less than 1.0.
  - Pixel positions along a line path are then determined by sampling at unit x intervals.
    - Starting from the left endpoint (x_o,y_o) of a given line, each successive column (x position) is determined
    - pixel is ploted whose scan-line **y value** is closest to the line path.



Deriving The Bresenham Line Algorithm

- Assume the pixel at (x_k, y_k) is displayed.
- Then, decide pixel to plot in column $x_k + 1$.
  - choices are the pixels at positions (x_k+1, y_k) and (x_k+1, y_k+1)
- At sample position x_k+1, the vertical separations from the mathematical line are labelled d_{upper} and d_{lower}



- **The y coordinate on the mathematical line at x_k+1 is:**

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

- **So, d_{upper} and d_{lower} are given as follows:**

$$d_{lower} = y - y_k$$
$$= m(x_k + 1) + b - y_k$$

and

$$d_{upper} = (y_k + 1) - y$$
$$= y_k + 1 - m(x_k + 1) - b$$

- To determine which of the two pixels is closest to the line path, an efficient test that is based on the difference between the two pixel separations is used

$$d_{lower} - d_{upper} = 2m(x_k + 1) - 2y_k + 2b - 1$$

- Let's substitute m with $\Delta y / \Delta x$ where Δx and Δy are the differences between the end-points

$$d_{lower} - d_{upper} = 2\frac{\Delta y}{\Delta x}(x_k + 1) - 2y_k + 2b - 1$$

- To avoid floating point number let multiply the difference (d_{lower} - d_{upper}) by and Δx (d_{lower} - d_{upper}) is known as decision parameter p

$$\Delta x(d_{lower} - d_{upper}) = \Delta x(2\frac{\Delta y}{\Delta x}(x_k + 1) - 2y_k + 2b - 1)$$

$$= 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + 2\Delta y + \Delta x(2b - 1)$$

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

- **A decision parameter p_k for the k^{th} step along a line is given by**

$$p_k = \Delta x(d_{lower} - d_{upper})$$

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

- The sign of the decision parameter p_k is the same as that of $d_{lower} - d_{upper}$
- If p_k is negative, then we choose the lower pixel, otherwise we choose the upper pixel
- Parameter c is constant
  - has the value $2\Delta y + \Delta x(2b - 1)$, which is independent of the pixel position
  - Value will be eliminated in the recursive calculations for p_k .
  - If the pixel at y_k is "closer" to the line path than the pixel at y_{k+1} (that is, $d_{lower} < d_{upper}$), then decision parameter p_k is negative.
    - In that case, the lower pixel is plotted;
    - Otherwise, upper pixel is plotted.

- Coordinate changes along the line occur in unit steps in either the x or y direction.
- At step **k+1** the decision parameter is given as:

$$p_{k+1} = 2\Delta y \cdot x_{k+1} - 2\Delta x \cdot y_{k+1} + c$$

- Subtracting $\mathbf{p_k}$ from this we get:

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

➢But, x_{k+1} is the same as x_k+1 and value of y_{k+1} - y_k is either 0 or 1 depending on the sign of p_k

$$p_{k+1} - p_k = 2\Delta y\underbrace{(x_{k+1} - x_k)}_{1} - 2\Delta x\underbrace{(y_{k+1} - y_k)}_{0 \text{ or } 1}$$

1

0 or 1

depending on sign of p_k

$P_k<0$, $y_{k+1}=y_k$
$P_k>0$, $y_{k+1}=y_k+1$

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

- The recursive calculation of decision parameters is performed at each integer x position, starting at the left coordinate endpoint of the line.
- The first decision parameter p_0 is evaluated at (x_0, y_0) from equation

$$P_k=2\Delta y.x_k-2\Delta x.y_k+c$$

where $c=2\Delta y+\Delta x(2b-1)$ and $m=\Delta y/\Delta x;$

$$P_0=0-0+c$$

$$=2\Delta y+\Delta x(2*0-1)$$

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

The Bresenham Line Algorithm

BRESENtfAM'S LINE DRAWING ALGORITtfM
(for $|m| < 1.0$)

1. Input the two line endpoints, storing the left endpoint in (x_0, y_0)
2. Plot the point (x_0, y_0)
3. Calculate the constants Δx, Δy, $2\Delta y$, and $(2\Delta y - 2\Delta x)$ and get the first value for the decision parameter as:

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

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

The Bresenham Line Algorithm (cont…)

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

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

5. Repeat step 4 $(\Delta x - 1)$ times

➢**The algorithm and derivation above assumes slopes are less than 1. for other slopes we need to adjust the algorithm slightly**

Digitize the line with end points (30,20) and (40,28)

- Plot the first point i.e. (30,20)
- Calculate the constants Δx, Δy, $2\Delta y$, and $(2\Delta y - 2\Delta x)$

  $\Delta x=40-30=10$, $\Delta y=28-20=8$, $2\Delta y=16$, $2\Delta x=20$,

- Compute the starting value for the decision parameter

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

  $= 16-10= 6 > 0$

- Compute the increments for calculating successive decision parameters

  $2\Delta y=16$

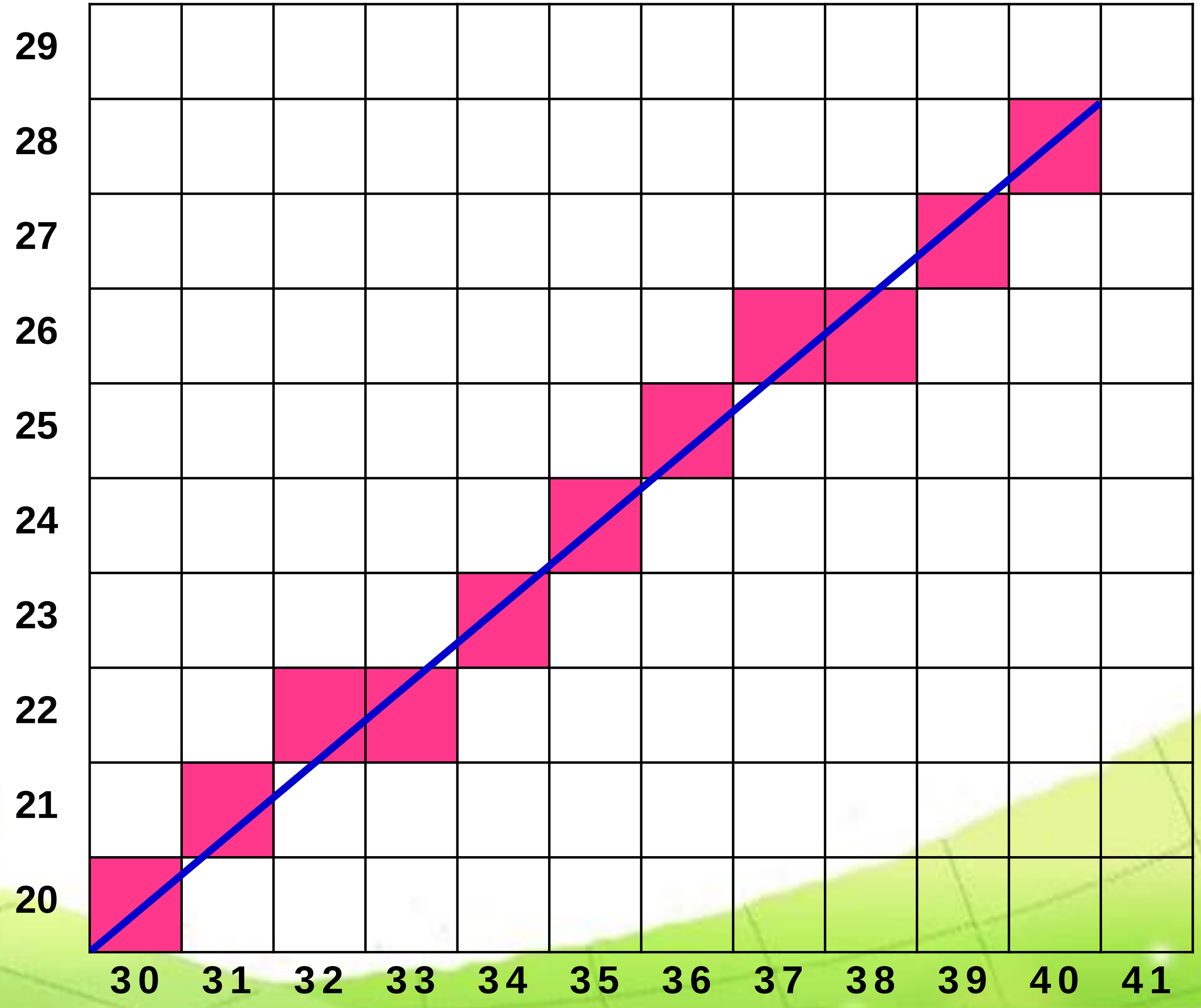
  $2\Delta y - 2\Delta x= 16-20$

  $=-4$

- $p_0>0$, compute (X_{k+1}, Y_{k+1}) and p_{k+1}
  - $(X_{k+1}, Y_{k+1}) = (31,21)$
  - $p_{k+1}= p_k+2\Delta y-2\Delta x= 6+16-20 =2 >0$
- $p_0>0$, compute (X_{k+1}, Y_{k+1}) and p_{k+1}
  - $(X_{k+1}, Y_{k+1}) = (32,22)$
  - $p_{k+1}= p_k+2\Delta y-2\Delta x= 2+16-20 =-2 <0$
- $p_0<0$, compute (X_{k+1}, Y_k) and p_{k+1}
  - $(X_{k+1}, Y_k) = (33,22)$
  - $P_{k+1}= p_k+2\Delta y =-2+16 =14 >0$
- $p_0>0$, compute (X_{k+1}, Y_{k+1}) and p_{k+1}
- Continue the procedure $\Delta x-1$ times

- Compute the successive pixel positions along the line path from the decision parameter p_0

K	P_k	(X_{k+1}, Y_{k+1})
0	6	31,21
1	2	32,22
2	-2	33,22
3	14	34,23
4	10	35,24
5	6	36,25
6	2	37,26
7	-2	38,26
8	14	39,27
9	10	40,28



Bresenham line-drawing procedure for 0<m<1.0

```
#include <stdlib.h>
#include <math.h>
void lineBres (int x0, int y0, int xEnd, int yEnd)
{
    int dx = fabs (xEnd - x0), dy = fabs(yEnd - y0);
    int p = 2 * dy - dx;
    int twoDy = 2 * dy, twoDyMinusDx = 2 * (dy - dx);
    int x, y;
    /* Determine which endpoint to use as start position.*/
    if (x0 > xEnd) {
    x = xEnd;
    y = yEnd;
    xEnd = x0;
}
```

```
setPixel (x, y);
while (x < xEnd)
{
    x++;
    if (p < 0)
      p += twoDy;
   else
   {
     y++;
     p += twoDyMinusDx;
   }
  setPixel (x, y);
  }
}
```