Bresenham's
Line drawing Algorithm

Issues With Line Drawing

- Line drawing is such a fundamental algorithm that it must be done fast
  - Use of floating point calculations does not facilitate speed
- Furthermore, lines must be drawn without gaps
  - Gaps look bad, also create problem in continuity searching cases.
  - If we try to fill a polygon made of lines with gaps the fill will leak out into other portions of the display
  - Eliminating gaps through direct implementation of any of the standard line equations is difficult
- Finally, we don't want pixel redundancy, i.e., to draw a line pixel more than once
  - That's wasting valuable time

Bresenham's Line Drawing Algorithm

- Jack Bresenham addressed these issues with the *Bresenham Line Scan Convert* algorithm
  - This was back in 1965 in the days of *pen-plotters*
- Features:
  - All simple integer calculations
  - Addition, subtraction, multiplication by 2 (shifts)
  - Guarantees connected (gap-less) lines where each point is drawn 1 time (no redundancy)
- However, the algorithm is very much slope dependent.
- Also known as the *midpoint line algorithm*

Bresenham's Line Algorithm

- Explicit form of a line:

$$y = mx + c \qquad \dots (1)$$

- Implicit form of the same line

$$f(x, y) = mx - y + c = 0$$

$Assuming\ m = \frac{\Delta y}{\Delta x},$

Algebraic manipulation yields:

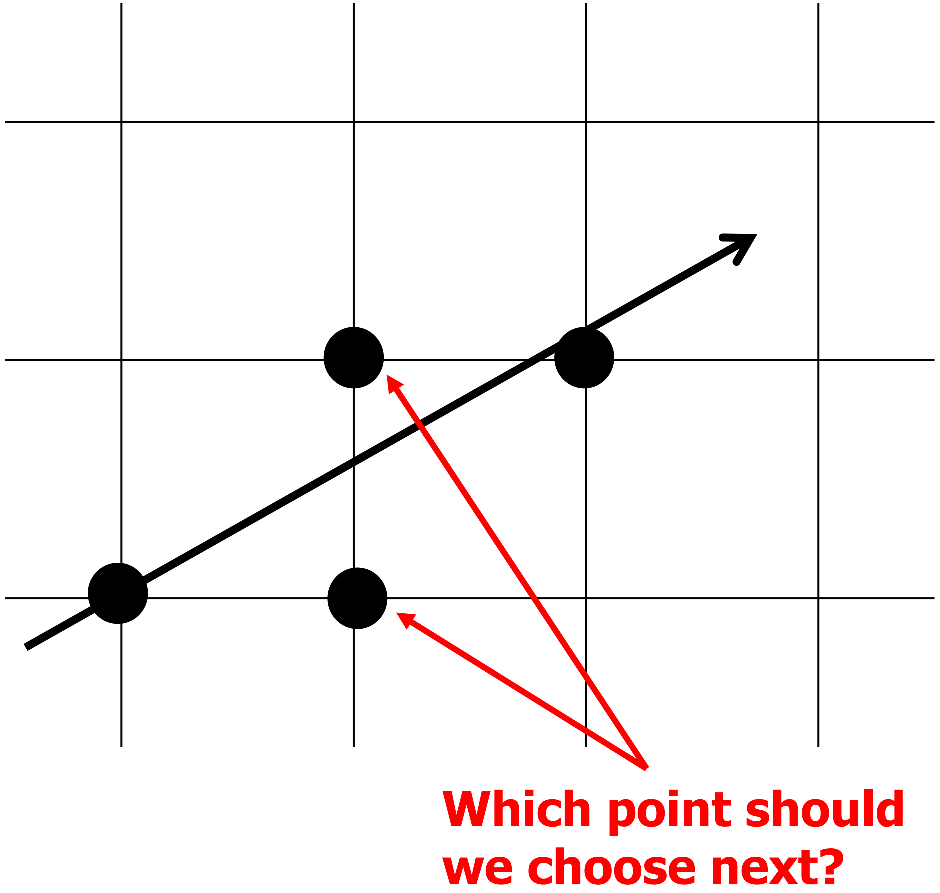
$$\Delta y \cdot x - \Delta x \cdot y + \Delta x \cdot c = 0$$

$$f(x, y) = Ax + By + C = 0 \quad \dots (2)$$

where: $A = \Delta y;\ \ and\ \ B = -\Delta x$

Bresenham's Line Algorithm

- As we know that our selection of line points is restricted to the grid of pixels
- The problem is now reduced to a decision of which grid point to draw at each step along the line
  - We have to determine how to make our steps (let's consider the lines with slope $0 \leq m \leq 1$.
- Equation (2) returns 0 value only when a point (x, y) is exactly on the line, otherwise it will return a value (say) *d* or deviation.
- Bresenham's algorithm utilizes the sign of d to determine the next pixel of the line



Bresenham's Line Algorithm

- What it comes down to is computing how close the midpoint (between the two grid points) is to the actual line.
- If we put the coordinate values of midpoint 'm' in equation 2:

$$F(m) = d_m = F(x_p + 1, y_p + \frac{1}{2})$$

$$F(m) = d_m = A \cdot (x_p + 1) + B \cdot (y_p + \frac{1}{2}) + C$$

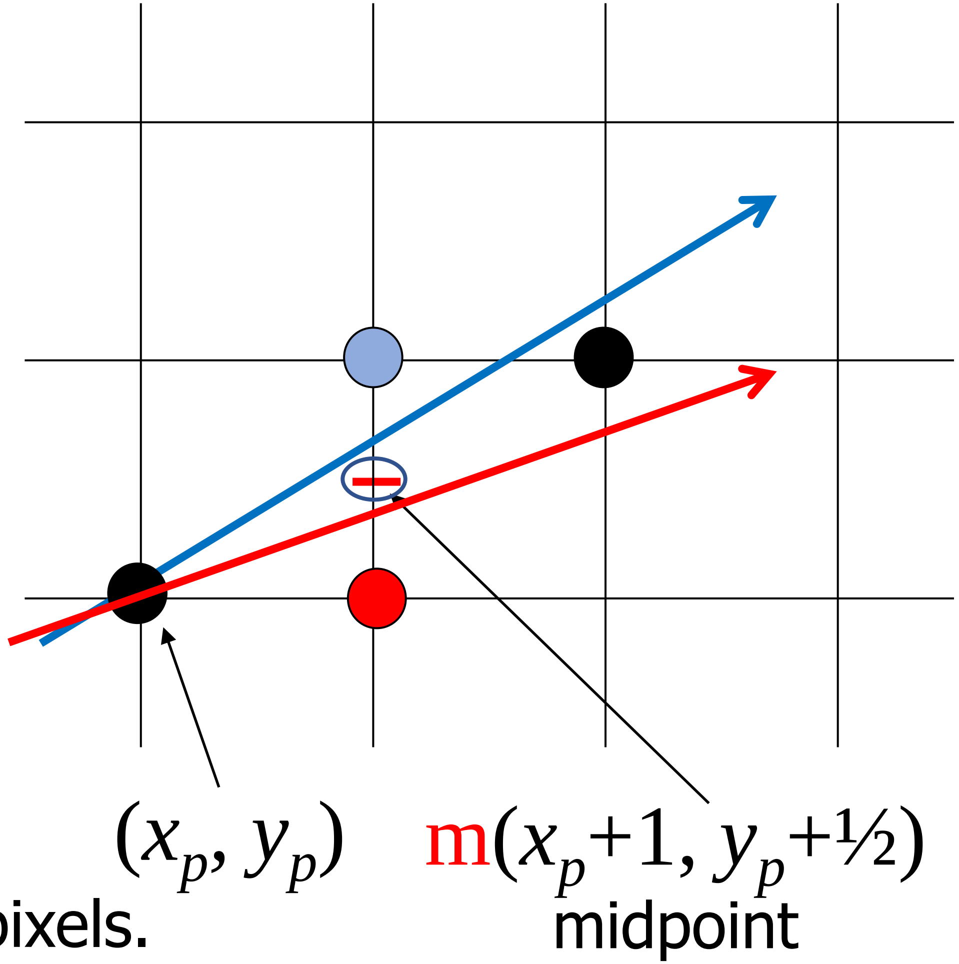
  - For red line, if the deviation 'd_m' is –ve, definitely it will be +ve for blue line.
  - That is,

if $d_m < 0$

the next line-pixel is red (x_p+1, y_p)

else

the next line-pixel is blue (x_p+1, y_p+1)



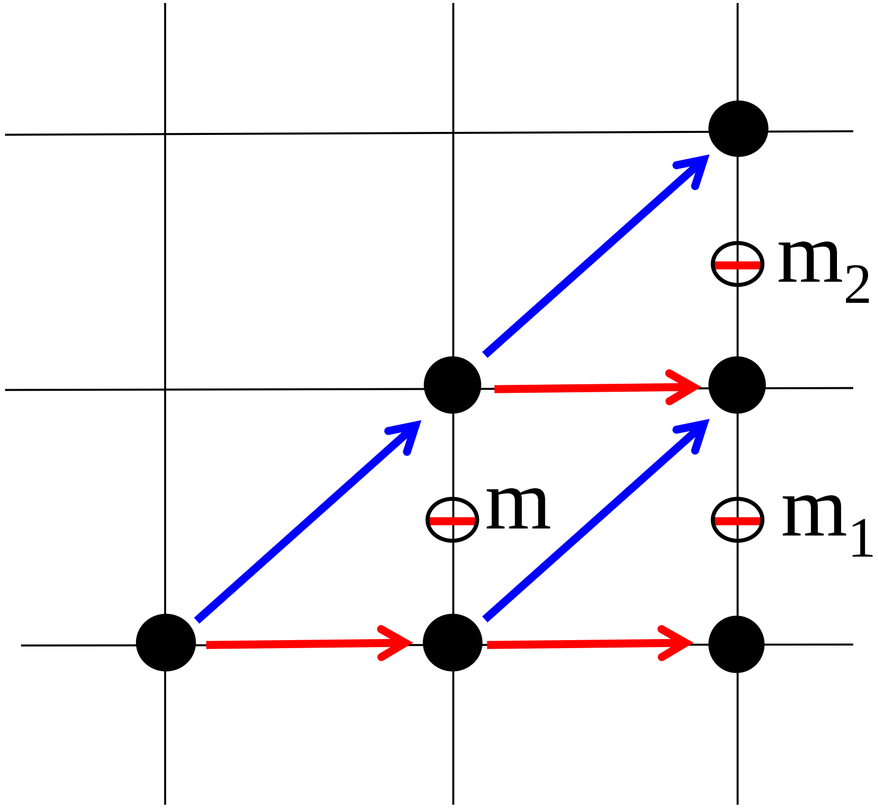
- Continue with the same logic for the successive line pixels.

Bresenham's Line Algorithm

- But this is yet another relatively complicated computation for every point
- Bresenham's "trick" is to compute the deviation *incrementally* rather than from scratch for every point
- That is, new_deviation = current_deviation + rate of change
- Looking one point ahead we have:

For horizontal movement (red-line),
new_deviation = current_deviation + $(d_{m1} - d_m)$

For diagonal movement (blue-line),
new_deviation = current_deviation + $(d_{m2} - d_m)$



Bresenham's Line Algorithm

- For horizontal movement (red-line),

  Rate of change of d = $\Delta E = d_{m1} - d_m$

- For diagonal movement (blue-line),

  Rate of change of d = $\Delta NE = d_{m2} - d_m$

- The coordinates of the midpoints are:

  $m = (x_p+1, y_p+ \frac{1}{2})$, $m_1 = (x_p+2, y_p+ \frac{1}{2})$

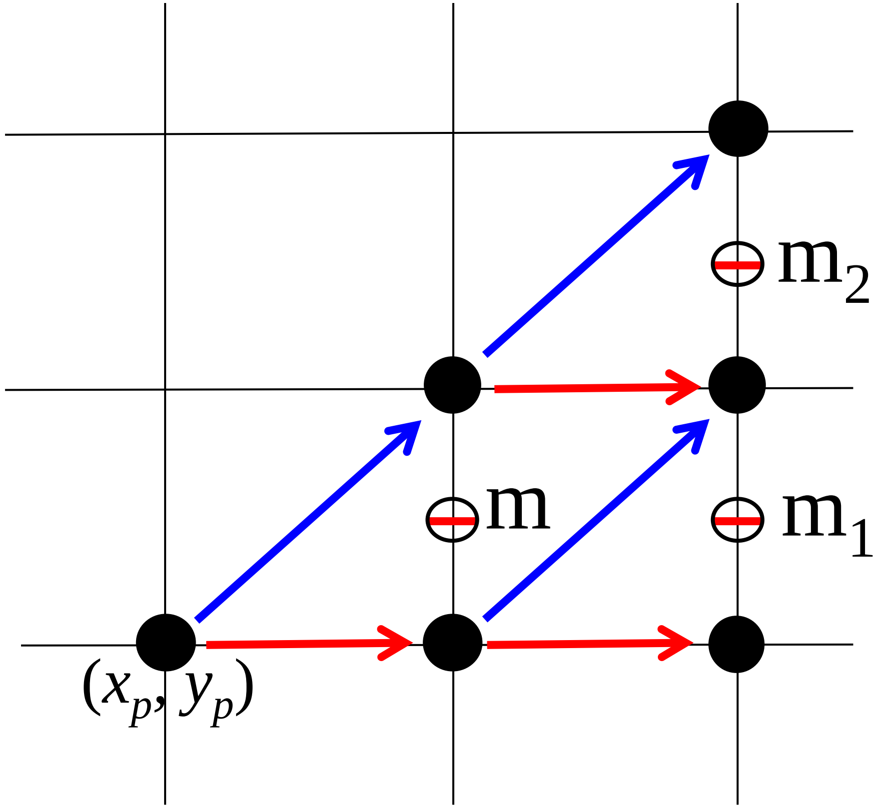
  and $m_2 = (x_p+2, y_p+ {}^{3}/_{2})$

- Deviations:

$$F(m) = d_m = A(x_p + 1) + B(y_p + \tfrac{1}{2}) + C$$

$$F(m_1) = d_{m1} = A(x_p + 2) + B(y_p + \tfrac{1}{2}) + C$$

$$F(m_2) = d_{m2} = A(x_p + 2) + B(y_p + {}^{3}/_{2}) + C$$



Bresenham's Line Algorithm

- For horizontal movement (red-line),

  Rate of change of d = $\Delta E = d_{m1} - d_m$

$$F(m_1) = d_{m1} = A(x_p + 2) + B(y_p + ½) + C$$
$$-F(m) = -d_m = -A(x_p + 1) - B(y_p + ½) - C$$

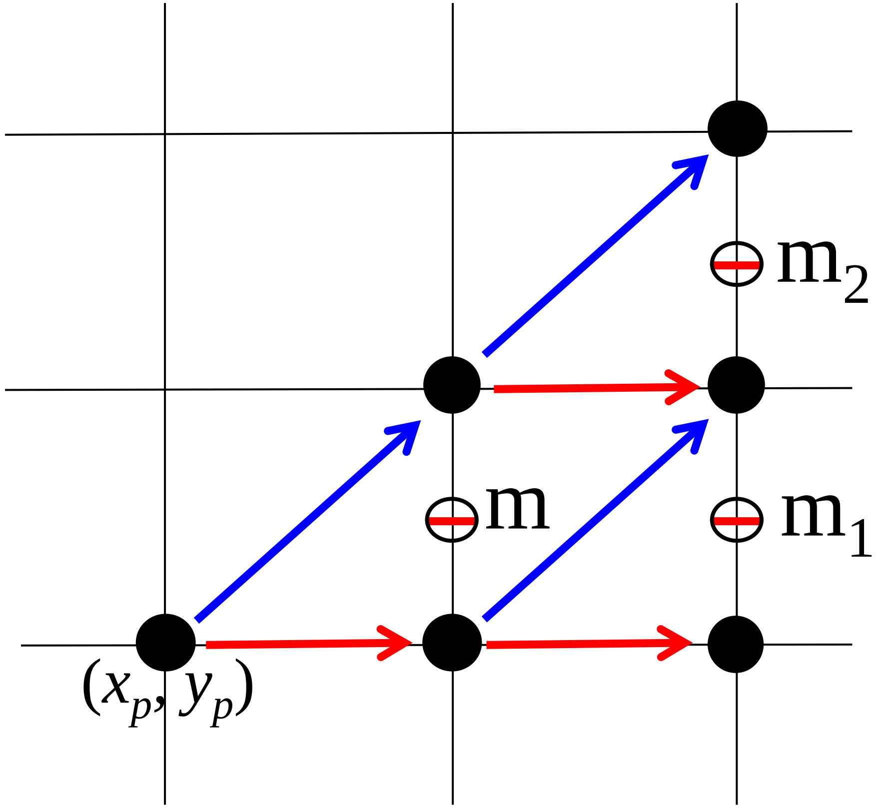
$$\boxed{\Delta E = d_{m1} - d_m = A = dy \qquad \ldots (3)}$$

- For diagonal movement (blue-line),

  Rate of change of d = $\Delta NE = d_{m2} - d_m$

$$F(m_2) = d_{m2} = A(x_p + 2) + B(y_p + {}^3/_2) + C$$
$$-F(m) = -d_m = -A(x_p + 1) - B(y_p + ½) - C$$

$$\boxed{\Delta NE = d_{m2} - d_m = A + B = dy - dx \qquad \ldots (4)}$$



Bresenham's Line Algorithm

- If $d < 0$ then the next deviation is: (horizontal movement)

$$d_{new} = d_{current} + \Delta E \qquad (\text{or } d = d + dy)$$

- If $d \geq 0$ then the next discriminant is: (diagonal movement)

$$d_{new} = d_{current} + \Delta NE \quad (\text{or } d = d + (dy - dx))$$

- These can now be computed incrementally given the proper starting value

Bresenham's Line Algorithm

- Initial point is (x_0, y_0) and we know that it is exactly on the line so

  $F(x_0, y_0)$ = must be 0, i.e., $Ax_0+By_0+C=0$
- Initial midpoint is $(x_0+1, y_0+½)$
- Initial deviation/discriminant is discriminant at $(x_0+1, y_0+½)$

$$\begin{aligned} F(x_0+1, y_0+½) &= A(x_0+1) + B(y_0+½) + C \\ &= (Ax_0+By_0+C) + A + B/2 \\ &= F(x_0, y_0) + A + B/2 \end{aligned}$$

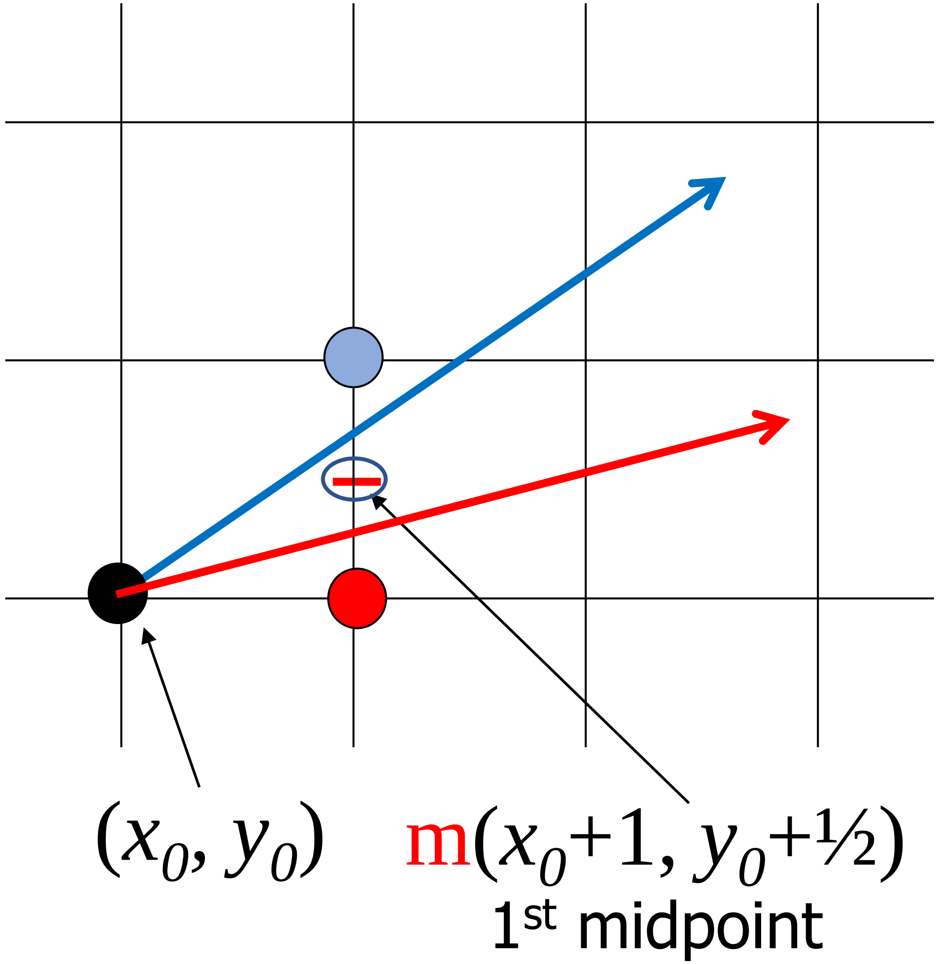
- And we know that $F(x_0, y_0) = 0$,
- So, $\boxed{d_{\text{initial}} = A + \frac{B}{2} = dy - \frac{dx}{2} \quad \ldots (5)}$

Bresenham's Line Algorithm

- Let's see the situation numerically, $\mathrm{d}_{\mathrm{initial}} = dy - \frac{dx}{2}$ … (5)

For red-line, if dx is 1.0, dy will be < 0.5
so, $d_{initial}$ is < 0.

For blue-line, if dx is 1.0, dy will be > 0.5
so, $d_{initial}$ is > 0.



Bresenham's Line Algorithm

- The algorithm then loops through x values in the range of $x_0 \leq x \leq x_1$ computing a y for each x – then plotting the point (x, y)
- Update step
  - If the discriminant/deviation (*let* $d = d_{initial}$) is less than 0 then increment x only, leaving y alone, and d will be updated by adding ΔE, *i.e.*, $d \mathrel{+}= \Delta E$
  - If the discriminant is greater than/equal to 0 then increment x, increment y, and d will be updated by adding ΔNE, *i.e.*, $d \mathrel{+}= \Delta NE$
  - This is for slopes less than or equal to 1
- If the slope is greater than 1 then loop on y (loop controller) and reverse the increments of x's and y's
- Sometimes we'll see implementations that, d, ΔE, and ΔNE *are multiplied* by 2 (to get rid of the floating point, initial divide by 2)
- And that is Bresenham's algorithm

Summary

- Why did we go through all this?
- Because it's an extremely important algorithm
- Because the problem demonstrates the "need for speed" in computer graphics
- Because it relates mathematics to computer graphics (and the math is simple algebraic manipulations)
- Because it presents a nice programming assignment…

Implementation

- Implement Bresenham's approach
  - You can search the web for this code – it’s out there, (I am giving it in the next page)
  - But, be careful because much of the code only does the slope [$0 \leq m \leq 1$] case (zone-0),
  - So you’ll have to think about the additional ways so that the algorithm is applicable for any slope.
  - Please learn to implement Bresenham’s algorithm for the cases of other slopes.

Code for implementing Bresenham's Algorithm

```
def drawLine_0(x0, y0, x1, y1):
    dx = x1 - x0
    dy = y1 - y0
    delE = 2 * dy
    delNE = 2 * (dy - dx)
    d = 2 * dy - dx
    x = x0
    y = y0
    print(f"x = {x} and y = {y}")
    while x < x1:
        if d < 0:
            d += delE
            x += 1
        else:
            d += delNE
            x += 1
            y += 1
        print(f"x = {x} and y = {y}")
drawLine_0(-5, -3, 5, 3)
```

Output

x = -5 and y = -3
x = -4 and y = -2
x = -3 and y = -2
x = -2 and y = -1
x = -1 and y = -1
x = 0 and y = 0
x = 1 and y = 1
x = 2 and y = 1
x = 3 and y = 2
x = 4 and y = 2
x = 5 and y = 3