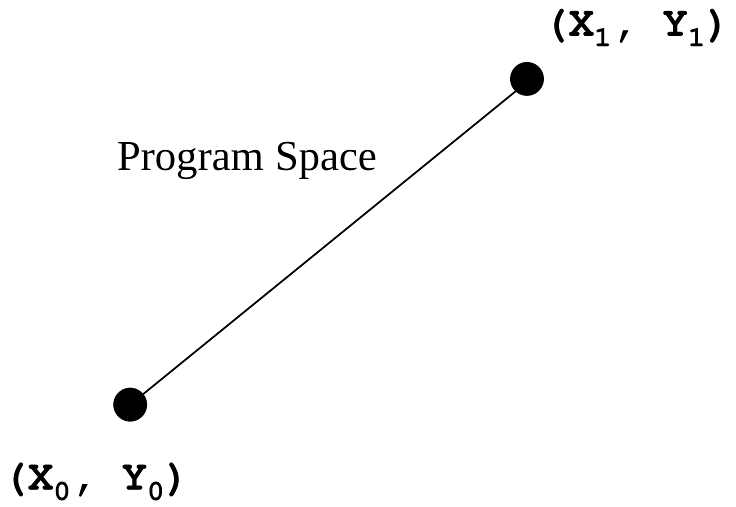
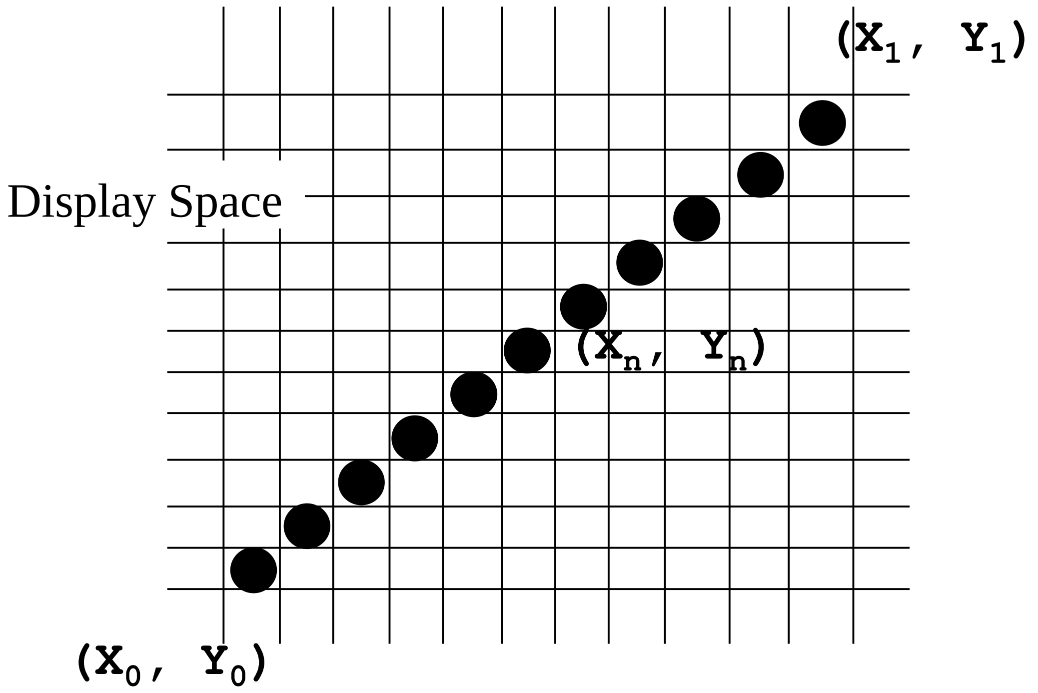
Scan Conversion

- Also known as *rasterization*
- In our programs objects are represented symbolically
  - 3D coordinates representing an object's position
  - Attributes representing color, motion, etc.
- But, the display device is a 2D array of pixels (unless you're doing holographic displays)
- Scan Conversion is the process in which an object's 2D symbolic representation is converted to pixels

Scan Conversion

- Consider a straight line in 2D
  - Our program will [most likely] represent it as two end points of a line segment which is not compatible with what the display expects

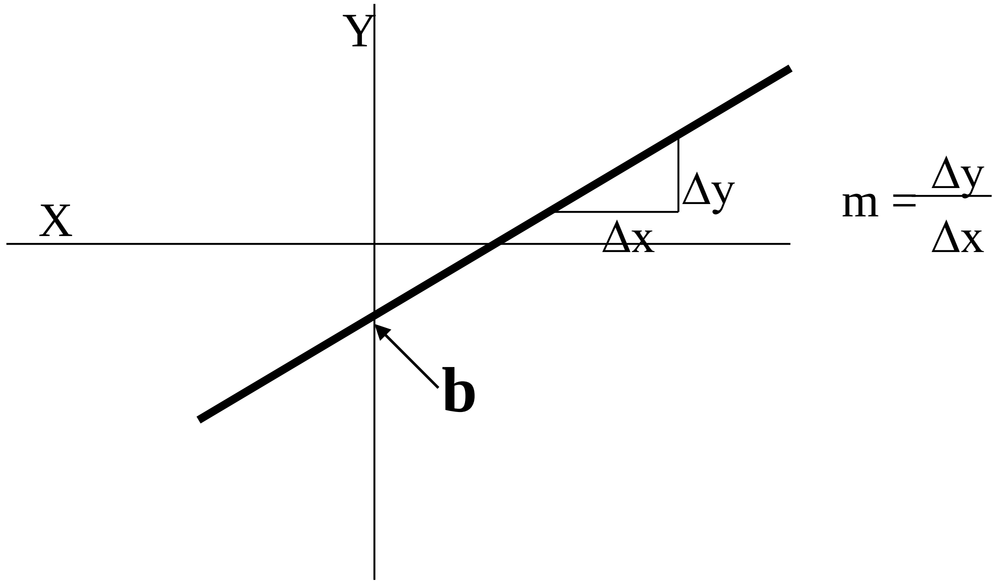




  - We need a way to perform the conversion

Line Drawing Algorithms

- Equations of a line
  - Point-Slope Form: $y = mx + b$ (Implicit form)
  - Also, $f(x,y) = Ax + By + C = 0$ (Explicit form)



As it turns out, this is not very convenient for scan converting a line

Drawing Lines

- Equations of a line
  - Two Point Form:

$$y = \frac{y_1 - y_0}{x_1 - x_0}(x - x_0) + y_0$$

  - Directly related to the point-slope form but now it allows us to represent a line by two points – which is what most graphics systems use

Drawing Lines

- Equations of a line
  - Parametric Form:

$$x = x_0 + (x_1 - x_0)t$$
$$y = y_0 + (y_1 - y_0)t$$

  - By varying *t* from 0 to 1we can compute all of the points between the end points of the line segment – which is really what we want

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 you 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 to draw individual pixels more than once
  - That’s wasting valuable time

Midpoint/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*
- All simple integer calculations
  - Addition, subtraction, multiplication by 2 (shifts)
- Guarantees connected (gap-less) lines where each point is drawn exactly 1 time
- Also known as the *midpoint line algorithm*

Bresenham's Line Algorithm

- Consider the two point-slope forms:

$$F(x, y) = Ax + By + C = 0 \qquad \text{Eqn. (1)}$$

$$y = \frac{\Delta y}{\Delta x} x + b$$

algebraic manipulation yields:

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

where:

$$\Delta y = (y_1 - y_0); \Delta x = (x_1 - x_0)$$

yielding:

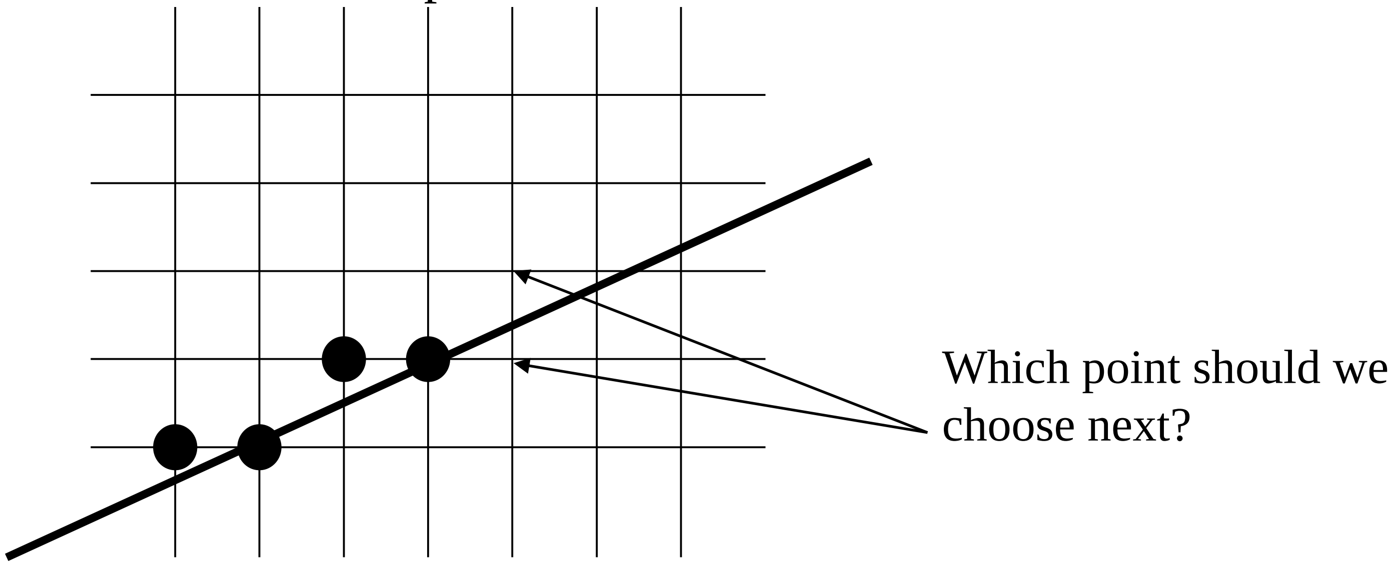
$$A = \Delta y; B = -\Delta x$$

Bresenham’s Line Algorithm

- Assume that the slope of the line segment is

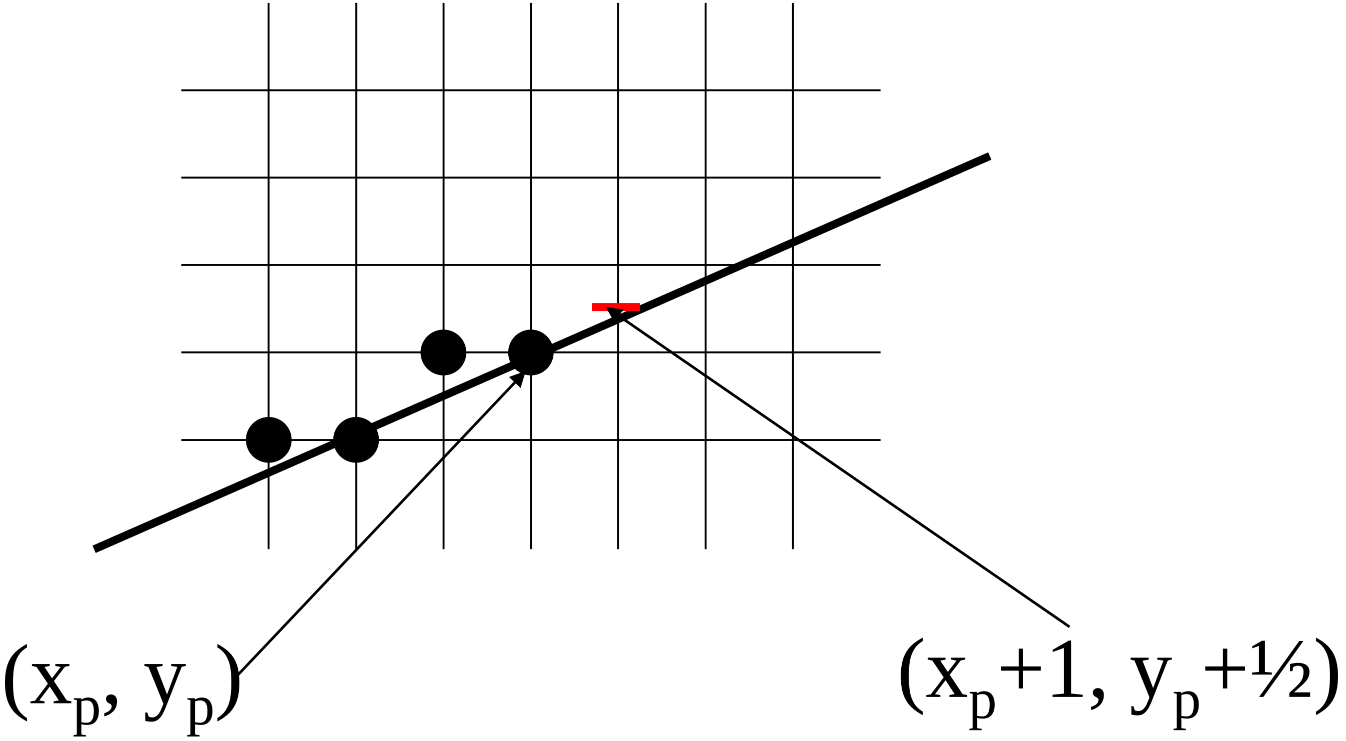
$$0 \leq m \leq 1$$

- Also, we know that our selection of points is restricted to the grid of display 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



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



Bresenham's Line Algorithm

- Define *F(m)* = *d* as the *discriminant or* *deviation*
  - (derive from the equation of a line *F(x,y)* = *Ax* + *By* + *C*)

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

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

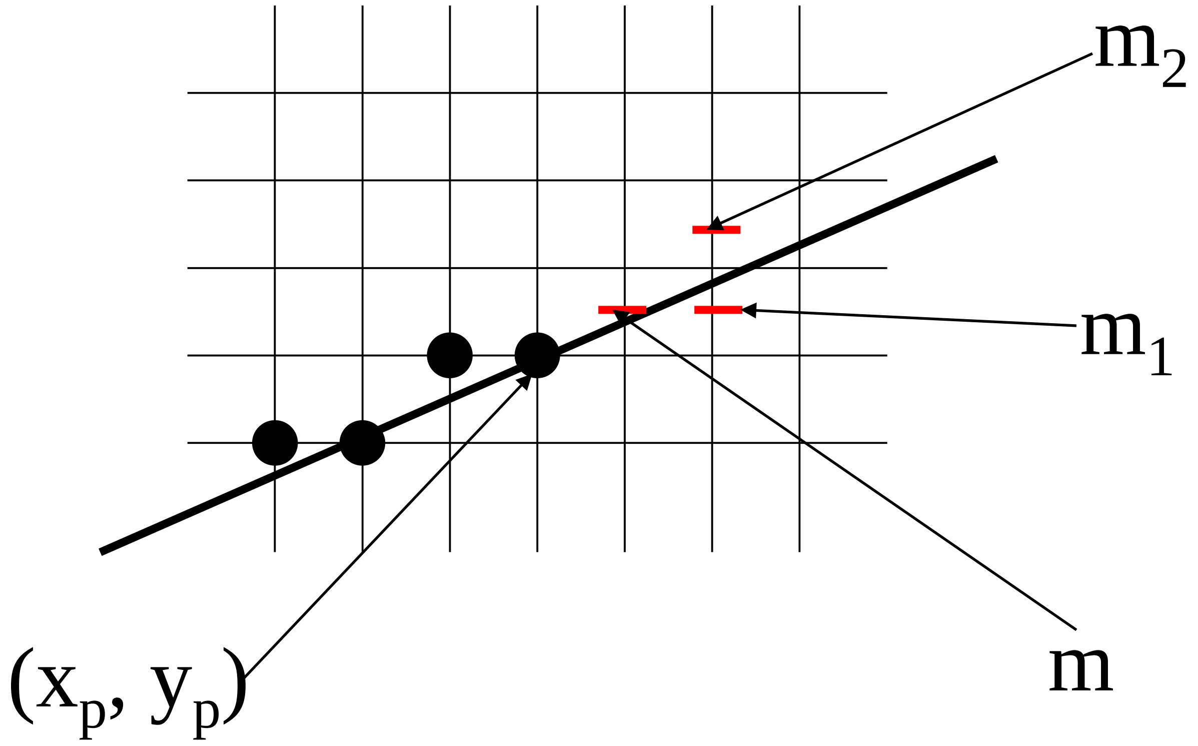
- if (d_m > 0) choose the upper point, otherwise choose the lower point

Bresenham’s Line Algorithm

- But this is yet another relatively complicated computation for every point
- Bresenham’s “trick” is to compute the discriminant *incrementally* rather than from scratch for every point

Bresenham’s Line Algorithm

- Looking one point ahead we have:



Bresenham's Line Algorithm

- If d > 0 then discriminant is: (diagonal movement)

$$F(x_p+2, y_p+\frac{3}{2}) = d_{m2} = A\cdot(x_p+2)+B\cdot(y_p+\frac{3}{2})+C$$

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

$$F(x_p+2, y_p+\frac{1}{2}) = d_{m1} = A\cdot(x_p+2)+B\cdot(y_p+\frac{1}{2})+C$$

- 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 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 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

$$d_{initial} = A + \frac{B}{2} = \Delta y - \frac{\Delta x}{2}$$

Bresenham's Line Algorithm

- Finally, to do this incrementally we need to know the differences between the current discriminant (d_m) and the two possible next discriminants (d_{m1} and d_{m2})

Bresenham's Line Algorithm

We know:

$$d_{current} = F(x_p + 1, y_p + \frac{1}{2}) = A \cdot (x_p + 1) + B \cdot (y_p + \frac{1}{2}) + C$$

$$d_{m1} = F(x_p + 2, y_p + \frac{1}{2}) = A \cdot (x_p + 2) + B \cdot (y_p + \frac{1}{2}) + C$$

$$d_{m2} = F(x_p + 2, y_p + \frac{3}{2}) = A \cdot (x_p + 2) + B \cdot (y_p + \frac{3}{2}) + C$$

We need (if we choose m1):

$$d_{m1} - d_{current} = A = (y_1 - y_0) = \Delta y = \Delta E$$

or (if we choose m2):

$$d_{m2} - d_{current} = A + B = (y_1 - y_0) - (x_1 - x_0) = \Delta y - \Delta x = \Delta NE$$

Bresenham's Line Algorithm

- Notice that *ΔE* and *ΔNE* both contain only constant, known values!

$$d_{m1} - d_{current} = A = (y_1 - y_0) = \Delta E$$

$$d_{m2} - d_{current} = A + B = (y_1 - y_0) - (x_1 - x_0) = \Delta NE$$

- Thus, we can use them incrementally – we need only add one of them to our current discriminant to get the next discriminant!

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 (*let* $d = d_{initial}$) is less than/equal to 0 then increment x only, leaving y alone, and increment the discriminant by ΔE ($d += \Delta E$)
  - If the discriminant is greater than 0 then increment x, increment y, and increment the discriminant by ΔNE ($d += \Delta NE$)
- 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 you'll see implementations that the discriminant, ΔE, and ΔNE 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
  - But, be careful because much of the code only does the slope < 1 case, so you’ll have to add the additional code so that the algorithm is applicable for any slope.