Scan Conversion

CMP 477 Computer Graphics

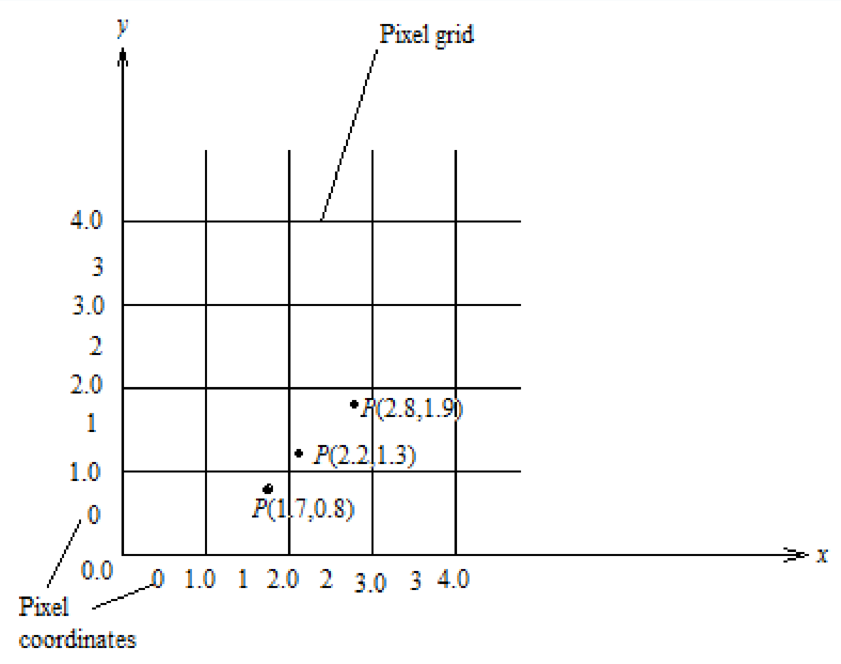
S. A. Arekete

What is Scan-Conversion?

- 2D or 3D objects in real world space are made up of graphic primitives such as points, lines, circles and filled polygons.
- These picture components are often defined in a contiguous space at a higher level of abstraction than individual pixels in the discrete image space.
- For instance, a line is defined by its two endpoints and the line equation while a circle is defined by its radius, centre position, and the circle equation.
- It is the responsibility of the graphics system or the application program to convert each primitive from its geometric definition into a set of pixels that makes up the primitive in the image space.
- This conversion task is generally referred to as scan-conversion or rasterization.

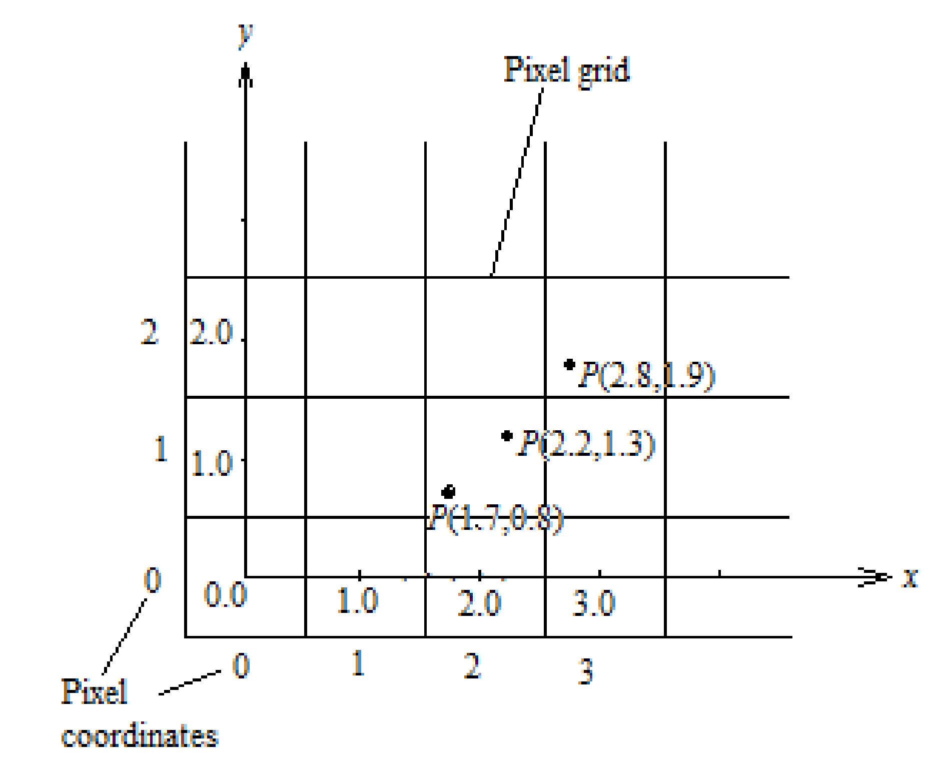
Scan-Converting a Point

- A mathematical point (x,y) where x and y are real numbers within the image area, needs to be converted to a pixel location (x',y').
- This can be done by making x' to be the integer part of x, and y' the integer part of y.
- In other words, $x' = Floor(x)$ and $y' = Floor(y)$, where function *Floor* returns the largest integer that is less than or equal to the argument.
- Doing so in essence places the origin of a continuous coordinate system (x,y) at the lowest left corner of the pixel grid in the image space
- All points that satisfy $x' \leq x < x' + 1$ and $y' \leq y < y' + 1$ are mapped to pixel (x', y')
- For example, point $P_1(1.7,0.8)$ is represented by pixel (1,0), points $P_2(2.2,1.3)$ and $P_3(2.8,1.9)$ are both represented by pixel (2,1)



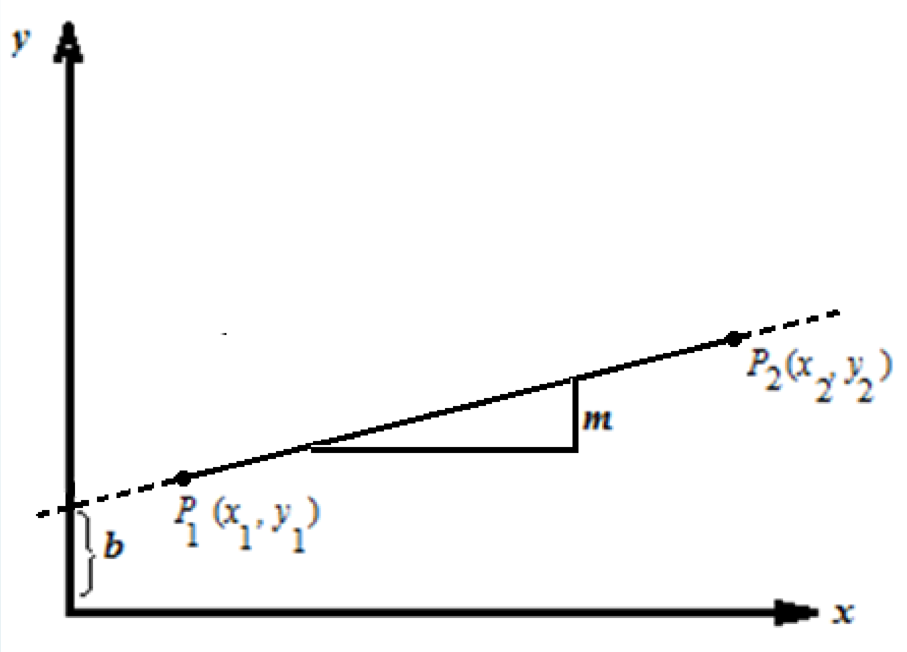
Scan-Converting a Point

- Another approach is to align the integer values in the coordinate system for (x,y) with the pixel coordinates
- Here we scan (x,y) by making, $x' = Floor(x + 0.5)$ and $y' = Floor(y + 0.5)$.
- This essentially places the origin of a coordinate system (x,y) at the centre of the pixel (0,0).
- 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')
- This means that points $P_1(1.7,0.8)$ and $P_2(2.2,1.3)$ are now both represented by pixel (2,1), whereas $P_3(2.8,1.9)$ is represented by pixel (3,2)



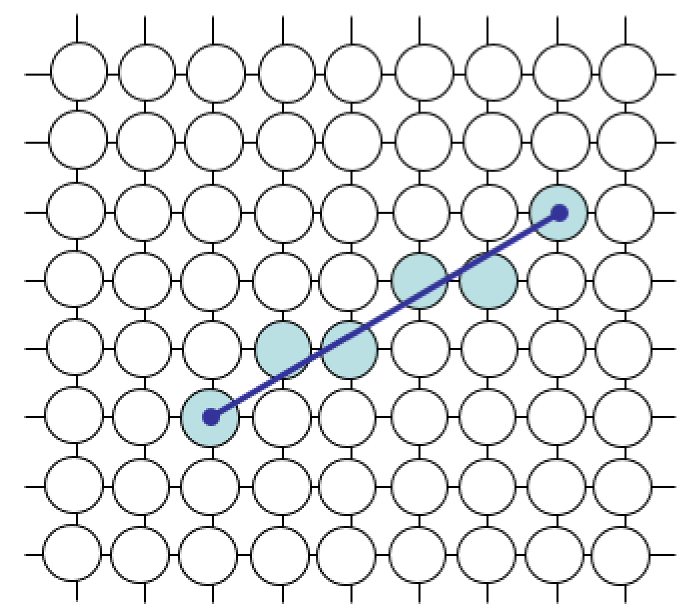
Scan-Converting a Line

- A line in computer graphics typically refers to a line segment – a portion of a straight line which extends indefinitely in opposite directions
- It is defined by the endpoints and the line equation: $y = mx + b$
  - Where ***m*** is the slope of the line and ***b*** is the *y* intercept
- NB: The slope-intercept equation is not suitable for vertical lines.
- Horizontal, vertical and diagonal lines |m|=1 are special cases which are often mapped into the image space specially for execution efficiency



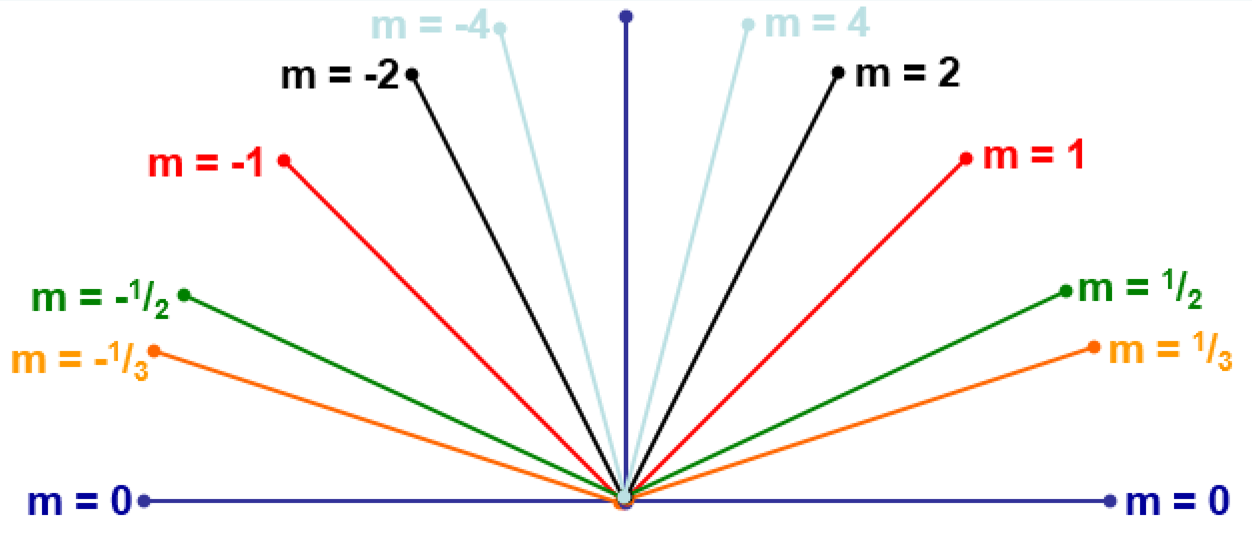
Consideration for Scan-Conversion of a Line

- But what happens when we try to draw line on a pixel based display?
- How do we choose which pixels to turn on?
- The line has to look good
  - Avoid jaggies
  - The drawing has to be very fast!
- How many lines need to be drawn in a typical scene?
- This is going to come back to bite us again and again



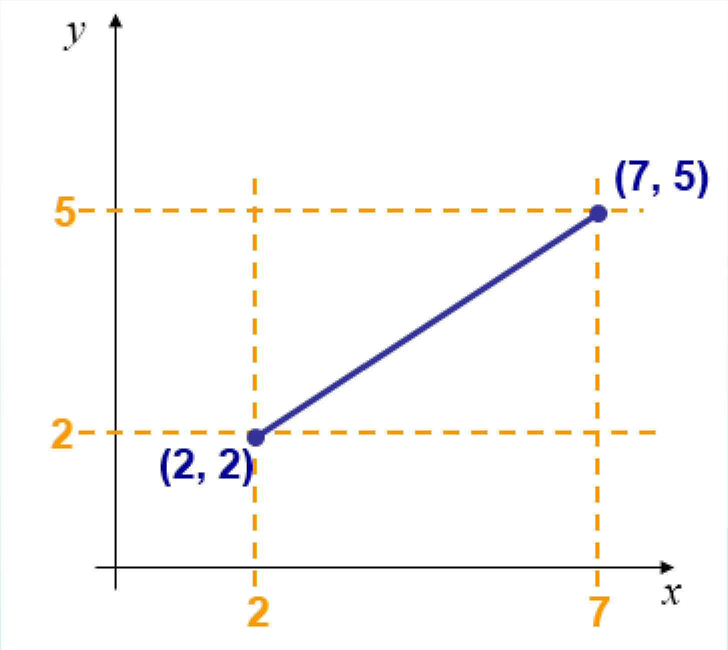
Lines and Slopes

- The slope of a line (m) is defined by its start and end coordinates
- The diagram below shows some examples of lines and their slopes

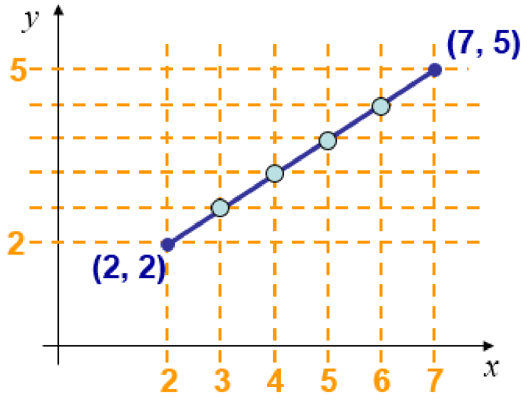


An Example of Direct Line Equation Method

- We could simply work out the corresponding *y* coordinate for each unit *x* coordinate
- Let's consider the following example:



Direct Line Equation Method..



First work out m and b:

$$m = \frac{5-2}{7-2} = \frac{3}{5}$$

$$b = 2 - \frac{3}{5} * 2 = \frac{4}{5}$$

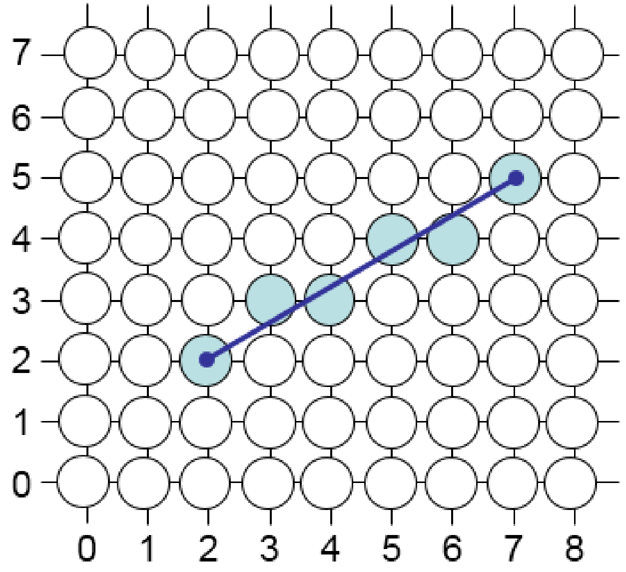
Now for each x value work out the y value:

$$y(3) = \frac{3}{5} \cdot 3 + \frac{4}{5} = 2\frac{3}{5} \qquad y(4) = \frac{3}{5} \cdot 4 + \frac{4}{5} = 3\frac{1}{5}$$

$$y(5) = \frac{3}{5} \cdot 5 + \frac{4}{5} = 3\frac{4}{5} \qquad y(6) = \frac{3}{5} \cdot 6 + \frac{4}{5} = 4\frac{2}{5}$$

Direct Line Equation Method..

Now just round off the results and turn on these pixels to draw our line



$$y(3) = 2\frac{3}{5} \approx 3$$

$$y(4) = 3\frac{1}{5} \approx 3$$

$$y(5) = 3\frac{4}{5} \approx 4$$

$$y(6) = 4\frac{2}{5} \approx 4$$

Limitations of the Direct Line Equation Method

- However, this approach is just way too slow as mentioned earlier
- In particular look out for:
  - The equation $y = mx + b$ requires the multiplication of m by x
  - Rounding off the resulting y coordinates
- We need a faster solution

The DDA Algorithm..

- The *digital differential analyzer* (DDA) algorithm takes an incremental approach in order to speed up scan conversion
- Simply calculate y_{k+1} based on y_k
- Consider the list of points that we determined for the line in our previous example:
- (2, 2), (3, $2^3/_5$), (4, $3^1/_5$), (5, $3^4/_5$), (6, $4^2/_5$), (7, 5)
- Notice that as the x coordinates go up by one, the y coordinates simply go up by the slope of the line
- This is the key insight in the DDA algorithm

The DDA Algorithm..

- When the slope of the line is between -1 and 1 begin at the first point in the line and, by incrementing the x coordinate by 1, calculate the corresponding y coordinates as follows:

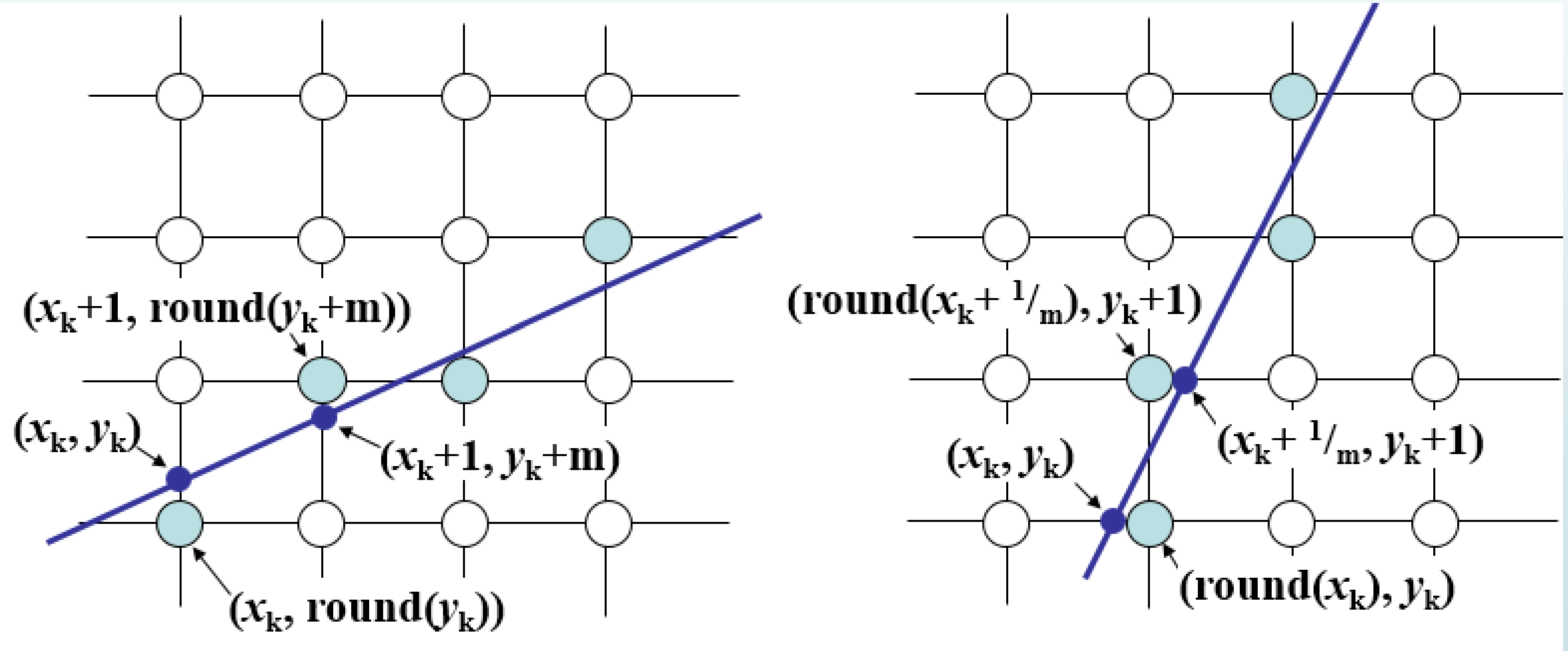
$$y_{k+1} = y_k + m$$

- When the slope is outside these limits, increment the y coordinate by 1 and calculate the corresponding x coordinates as follows:

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

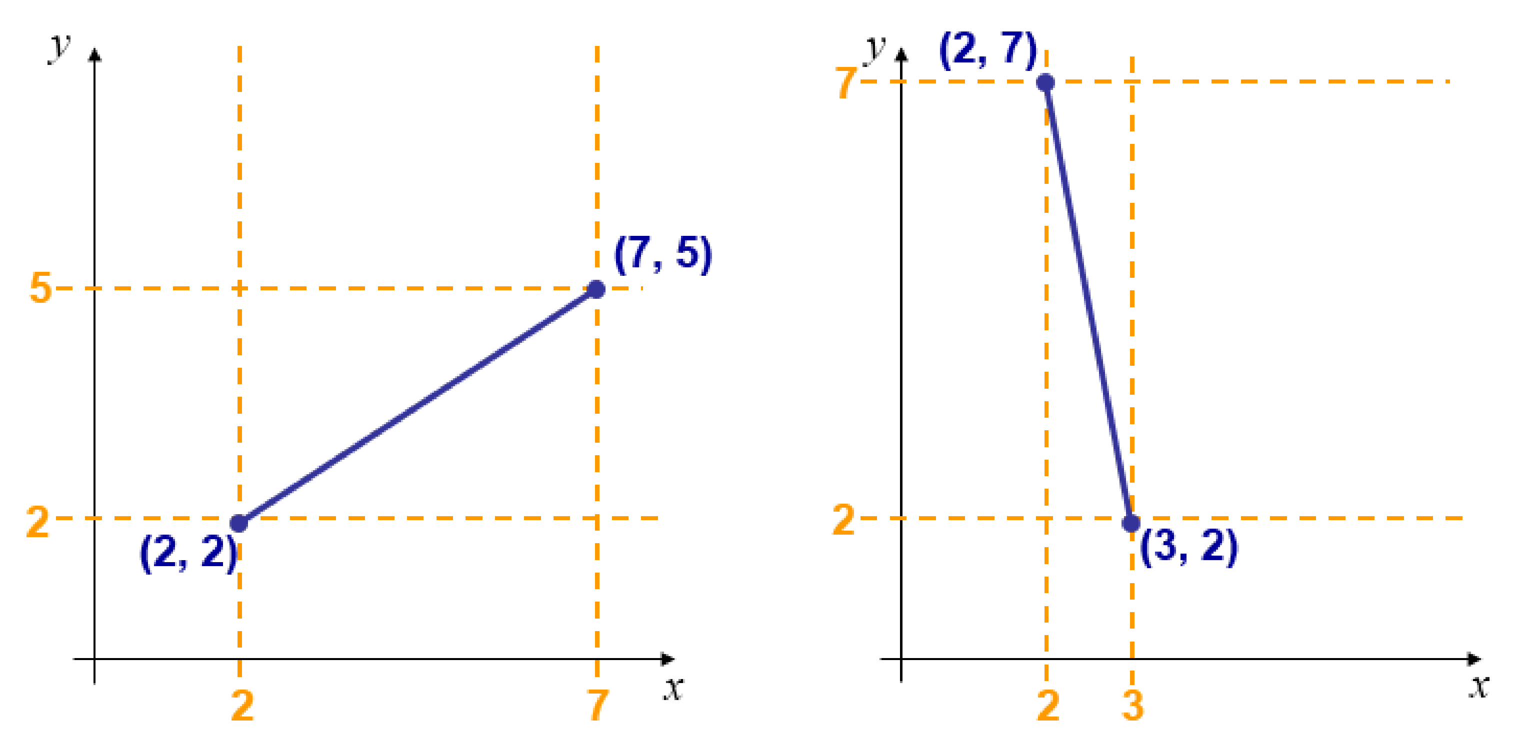
- **Limitation of the DDA**: The values calculated by the equations used by the DDA algorithm must be rounded to match pixel values

The DDA Algorithm..

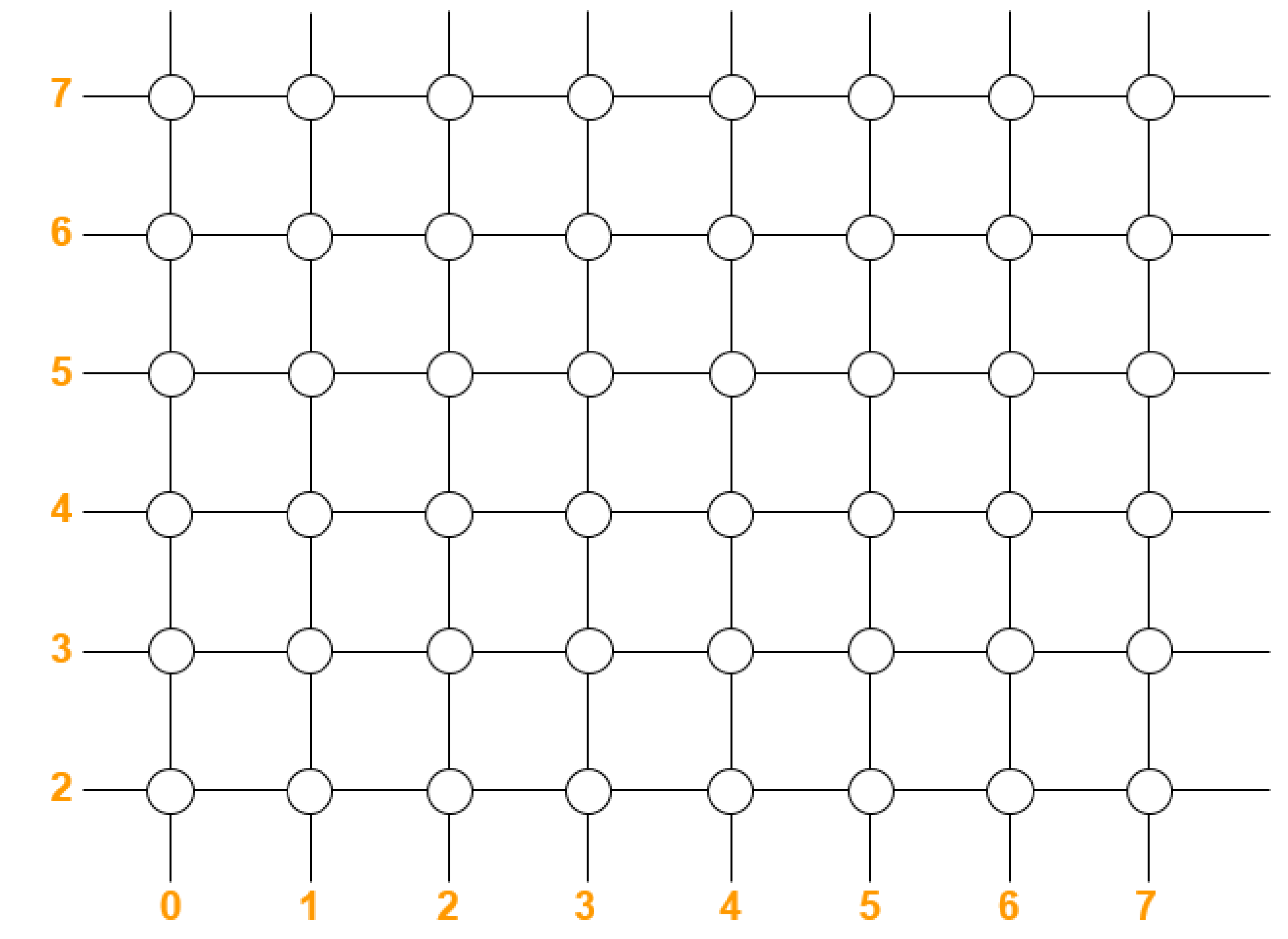


DDA Algorithm Example

- Let's try out the following examples:



DDA Algorithm Example..

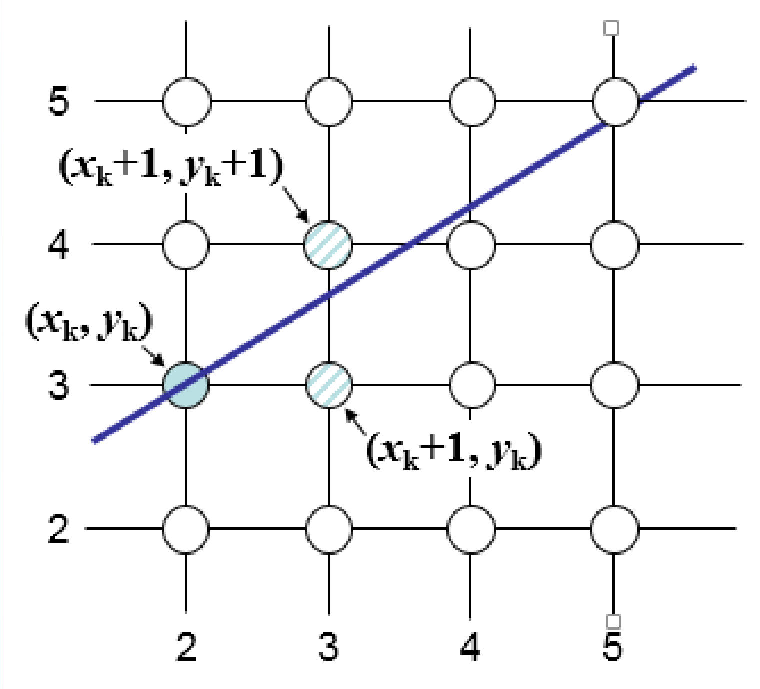


The DDA Algorithm Summary

- The DDA algorithm is much faster than our previous attempt
  - In particular, there are no longer any multiplications involved
- However, there are still two big issues:
  - Accumulation of round-off errors can make the pixelated line drift away from what was intended
  - The rounding operations and floating point arithmetic involved are time consuming

The Bresenham's Line Algorithm

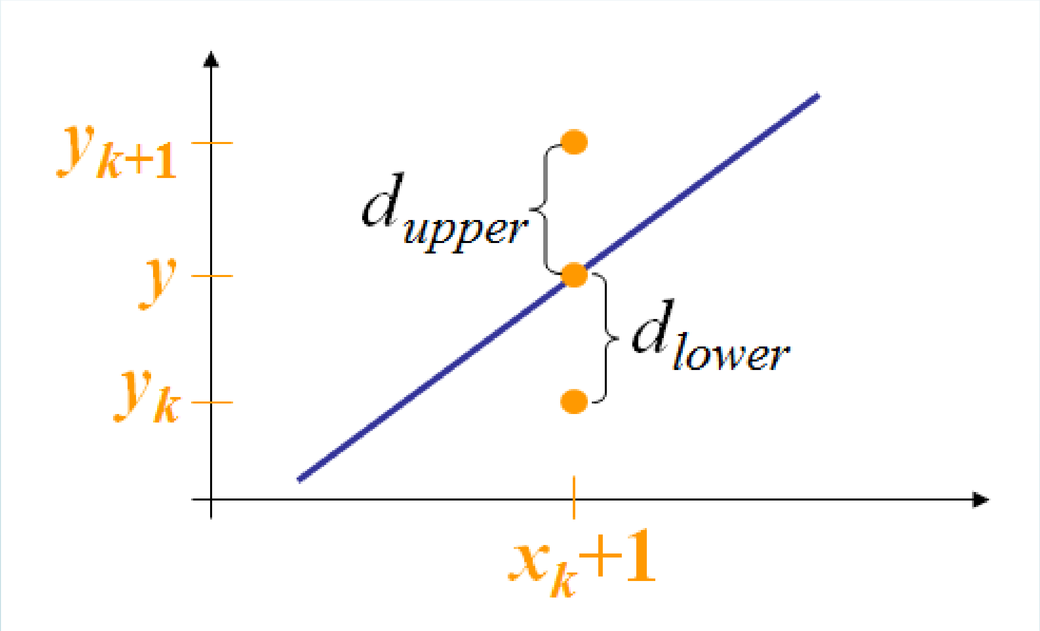
- The Bresenham algorithm is another incremental scan conversion algorithm
- The big advantage of this algorithm is that it uses only integer calculations
- Move across the x axis in unit intervals and at each step choose between two different y coordinates
- For example, from position (2, 3) we have to choose between (3, 3) and (3, 4)
- We would like the point that is closer to the original line



The Bresenham's Line Algorithm..

- 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:

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



The Bresenham's Line Algorithm..

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

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

- and

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

- We can use these to make a simple decision about which pixel is closer to the mathematical line

The Bresenham's Line Algorithm..

- This simple decision is based on the difference between the two pixel positions:

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

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

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

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

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

The Bresenham's Line Algorithm..

- So, a decision parameter p_k for the kth step along a line is given by:

$$\begin{aligned} p_k &= \Delta x(d_{lower} - d_{upper}) \\ &= 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + c \end{aligned}$$

- 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

The Bresenham's Line Algorithm..

- Remember coordinate changes occur along the x axis in unit steps so we can do everything with integer calculations
- 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 p_k from this we get:

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

The Bresenham's Line Algorithm..

But, x_{k+1} is the same as x_k+1 so:

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

where y_{k+1} - y_k is either 0 or 1 depending on the sign of p_k

The first decision parameter p_0 is evaluated at (x_0, y_0) is given as:

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

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

   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

N.B.: The algorithm and derivation above assumes slopes are less than 1. For other slopes we need to adjust the algorithm slightly

An Example on Bresenham's Line Algorithm

- Let's have a go at this:
- Let's plot the line from (20, 10) to (30, 18)
- First off calculate all of the constants:

  Δx: 10

  Δy: 8

  $2\Delta y$: 16

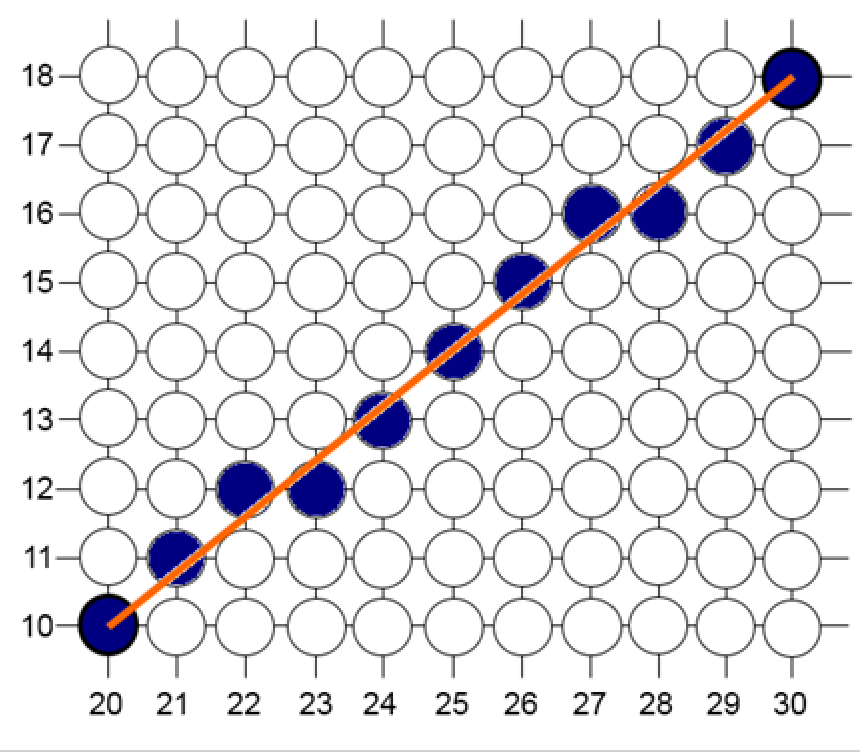
  $2\Delta y - 2\Delta x$: -4

- Calculate the initial decision parameter p_0:

$$p_0 = 2\Delta y - \Delta x = 6$$

An Example on Bresenham's Line Algorithm..

- Go through the steps of the Bresenham line drawing algorithm for a line going from (21,12) to (29,16)



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

Bresenham Line Algorithm Summary

- The Bresenham's line algorithm has the following advantages:
  - A fast incremental algorithm
  - Uses only integer calculations
- Comparing this to the DDA algorithm, DDA has the following problems:
  - Accumulation of round-off errors can make the pixelated line drift away from what was intended
  - The rounding operations and floating point arithmetic involved are time consuming