CSD2341

Computer Graphics Programming




Line Drawing

Third Edition

- Hearn and Baker 3.5

Fourth Edition

- Hearn, Baker & Carithers 6.1

Line Drawing – Why?

- Can't we just go
  - g.draw(line) ? (Java graphics API)
  - gl.gl_begin(GL.GL_LINES) etc? (OpenGL)

- Sure, but
  - Knowing how these methods work helps you understand their properties and limitations
  - You might have to write your own (e.g. for a custom embedded system)
  - The same algorithm has other purposes – e.g. computing the path of a projectile for a simulation or game.
  - The ideas behind these algorithms can be used to solve other problems.

What is a line?

- We really mean a line segment, i.e. all the points on the shortest path between two end points.
- For a random-scan device, this is easy
- For a raster-scan device, we have to figure out which pixels to set to which colours
- This means calculating an approximation to the real line, called
  - Digitizing
  - Rasterization
  - Scan-conversion

Aims for a good line drawing algorithm:

- Accuracy – pixels closest to true position
- Speed – calculated quickly
- Continuous appearance
  - No gaps
  - No “jaggies”
- Uniform thickness and brightness
- Consistency (start and end points)

Math review: What is a line?

- Equation of a line in 2D in terms of (x,y) coordinates:

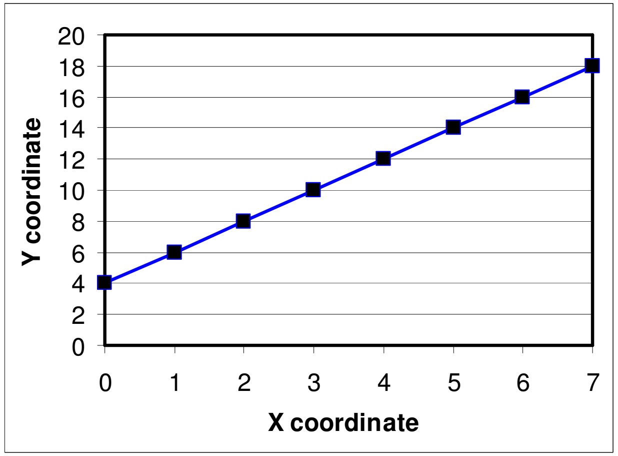
$$y = mx + c$$

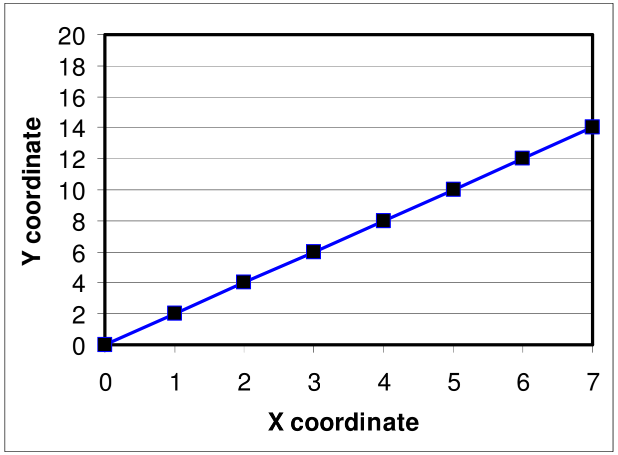
- m = slope of line
- c = y-axis intercept
- If we know m and c, we can work out a y co-ordinate for each x co-ordinate

What is a line?

$y = mx + c$

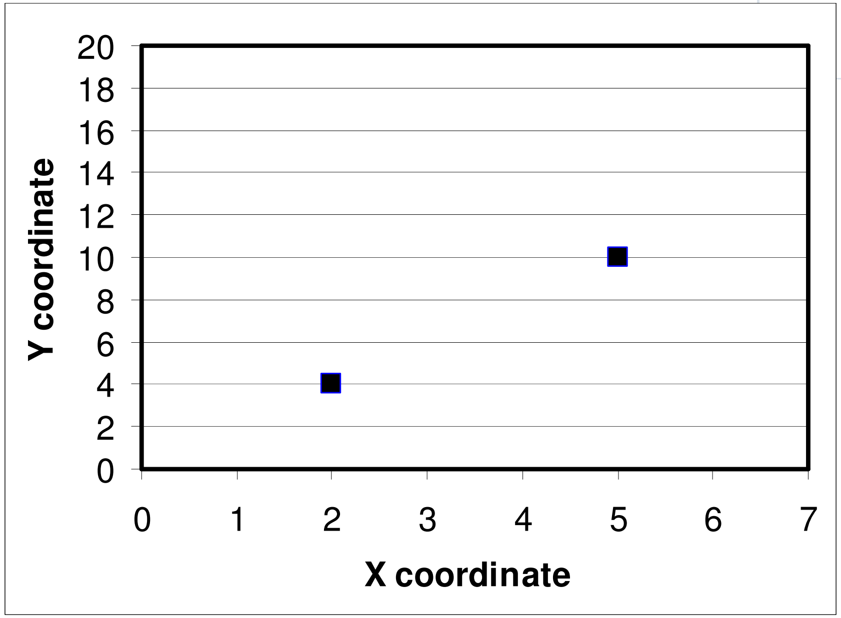
- m is the rate that y increases with respect to x, (how much y increases when x increases by 1)
  - called the gradient, slope, and rise/run
- What is the slope (m) of these two lines? – m = 2 for both.





What is a line?

- $y = mx + c$
- What is the slope (m) of the line joining these two points?
- P1 $(x1,y1) = (2,4)$
- P2 $(x2,y2) = (5,10)$



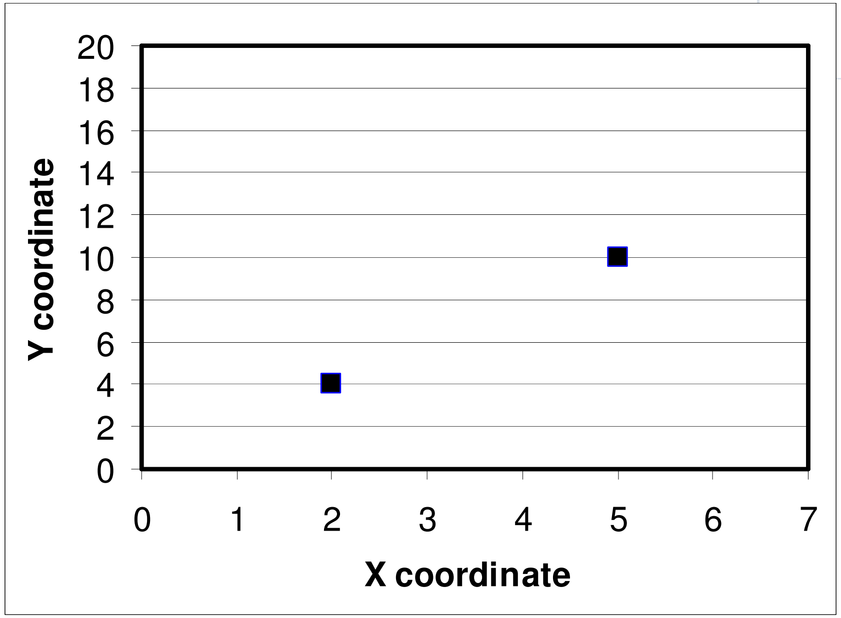
What is a line?

$y = mx + c$

- What is the slope (m) of the line joining these two points?
- x increases by 3
- y increases by 6
- So m = (y2-y1)/(x2-x1)

  = 6/3

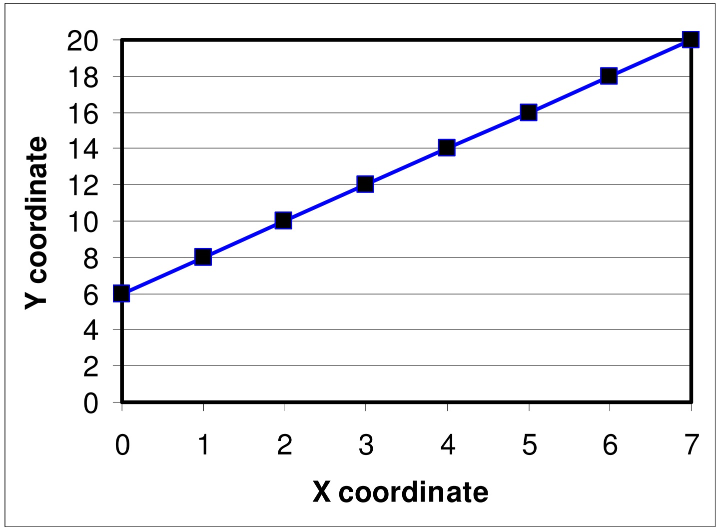
  = 2

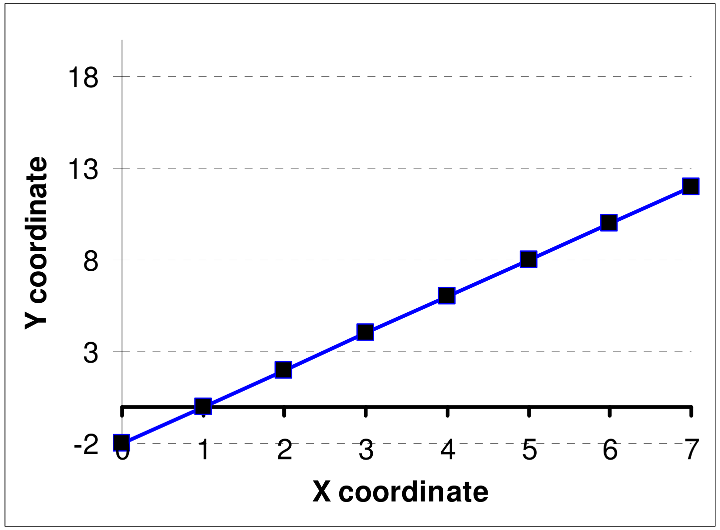


What is a line?

$y = mx + c$

- c has the effect of moving the line up or down
- When x = 0, y = c , so the line cuts the y-axis at y = c. That's why it is also known as the y-intercept.
- Examples, c = 6, c = -2.

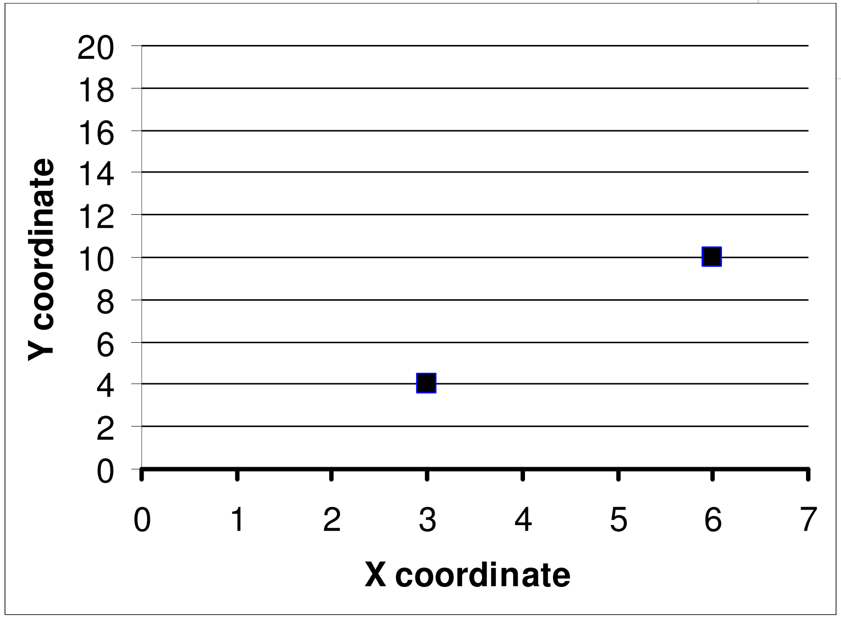




What is a line?

$y = mx + c$

- A character in a game is at position A (x1,y1) = (3,4). The player clicks on location B (x2,y2) = (6,10). What is the equation of the straight line path to take the character from A to B?



What is a line?

$y = mx + c$

(x1,y1) = (3,4)

(x2,y2) = (6,10)

m = (y2-y1)/(x2-x1)

= (10-4)/(6-3)

= 2

So the slope is 2, and all points on the line must obey the equation

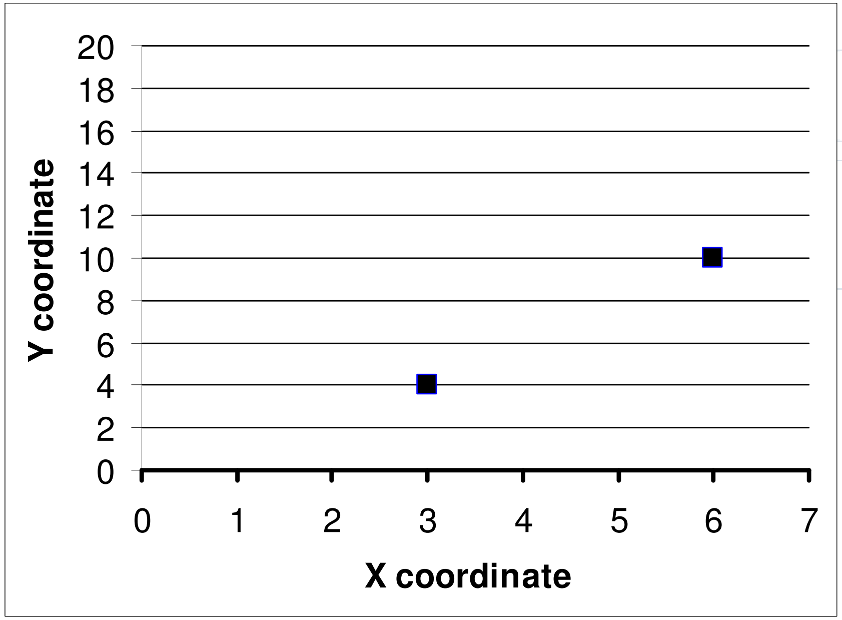
y = 2x + c

Putting point A into the equation (same result if you put point B in):

4 = 2*3 + c

c = 4 - (2*3)

c = -2



So: y = 2x + (-2)

What is a line?

$y = mx + c$

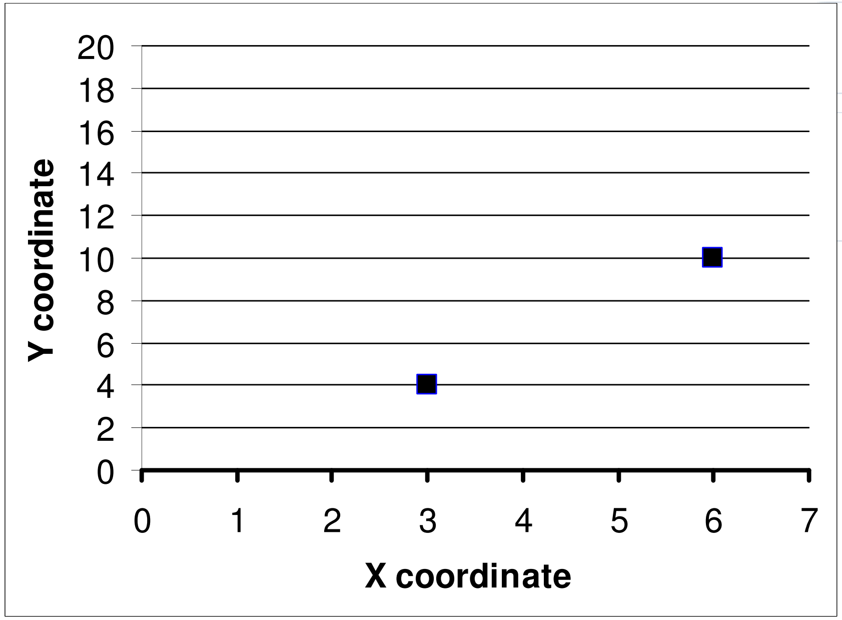
How to code this?

x1 = 3, x2 = 6, y1 = 4, y2 = 10;

m = (y2-y1)/(x2-x1);

c = y1 – m*x1;

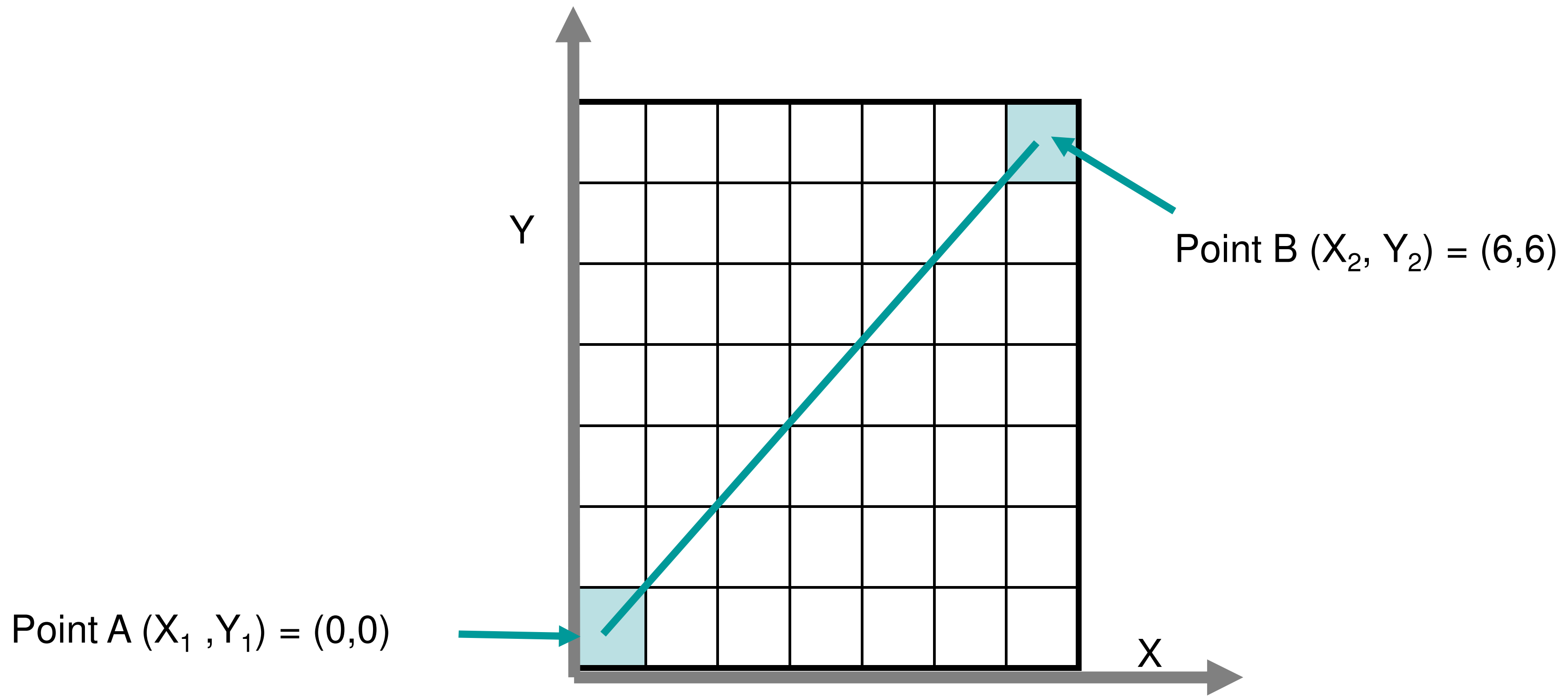
- variables m and c must be floating point type as the slope may not be a whole number.



So: $y = 2x + (-2)$

Line drawing

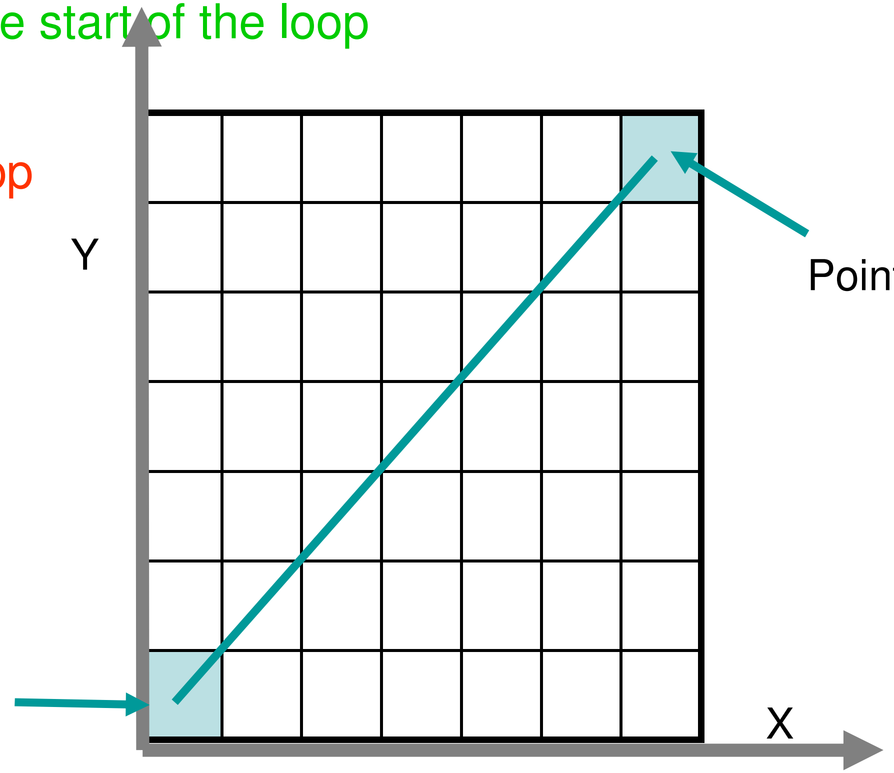
- How to draw a continuous line segment between point A (x_1,y_1), and point B (x_2,y_2) using pixels?



Line drawing

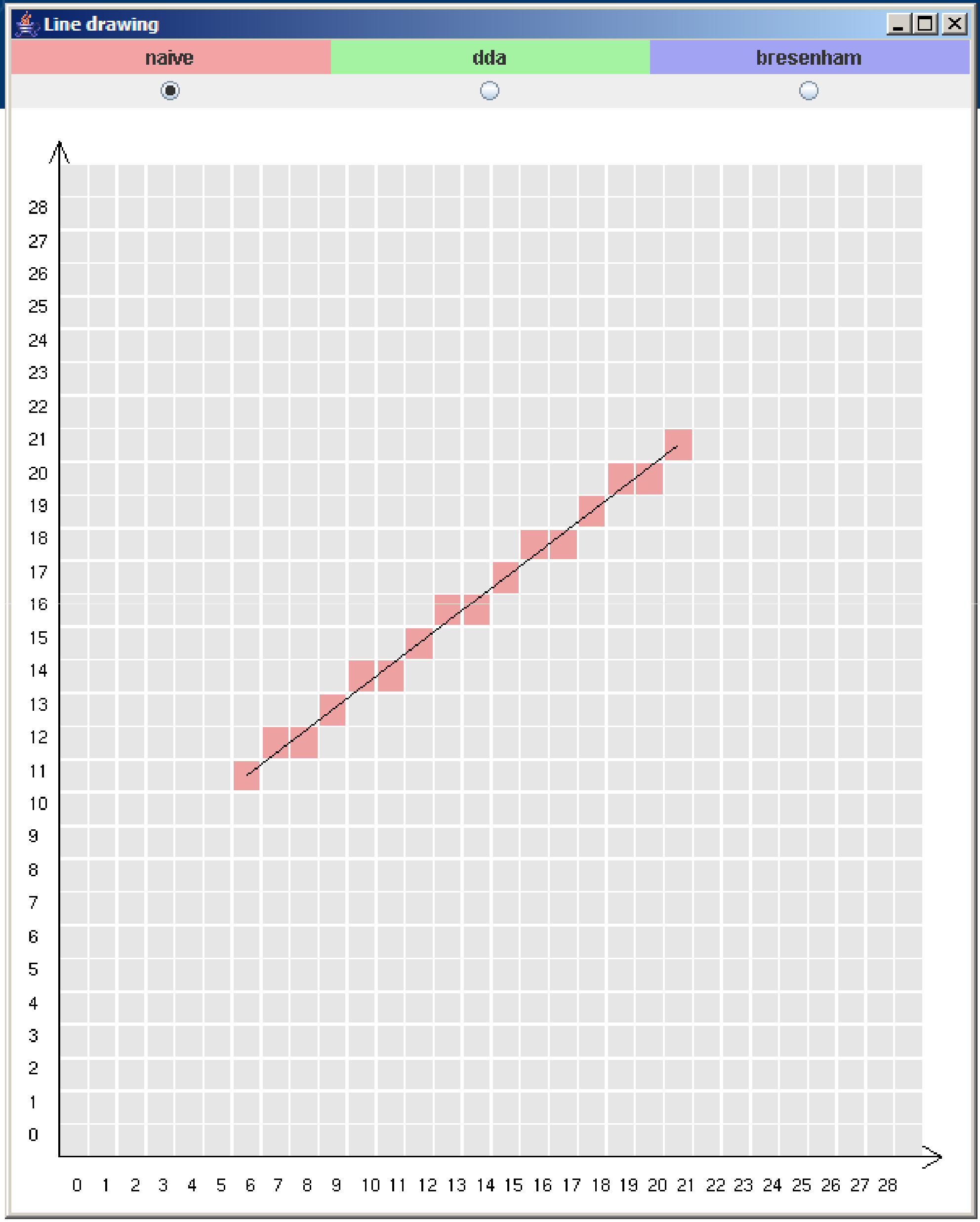
```
m = 1, c = 0; // set the line equation parameters
for (x = x1 to x2) // start a loop to go through all possible x-values
  y = m*x + c; // work out the y-value (may have to round)
  putPixel(x,y); // put a pixel at position x,y
end for // go back to the start of the loop
```

Try out line drawing app



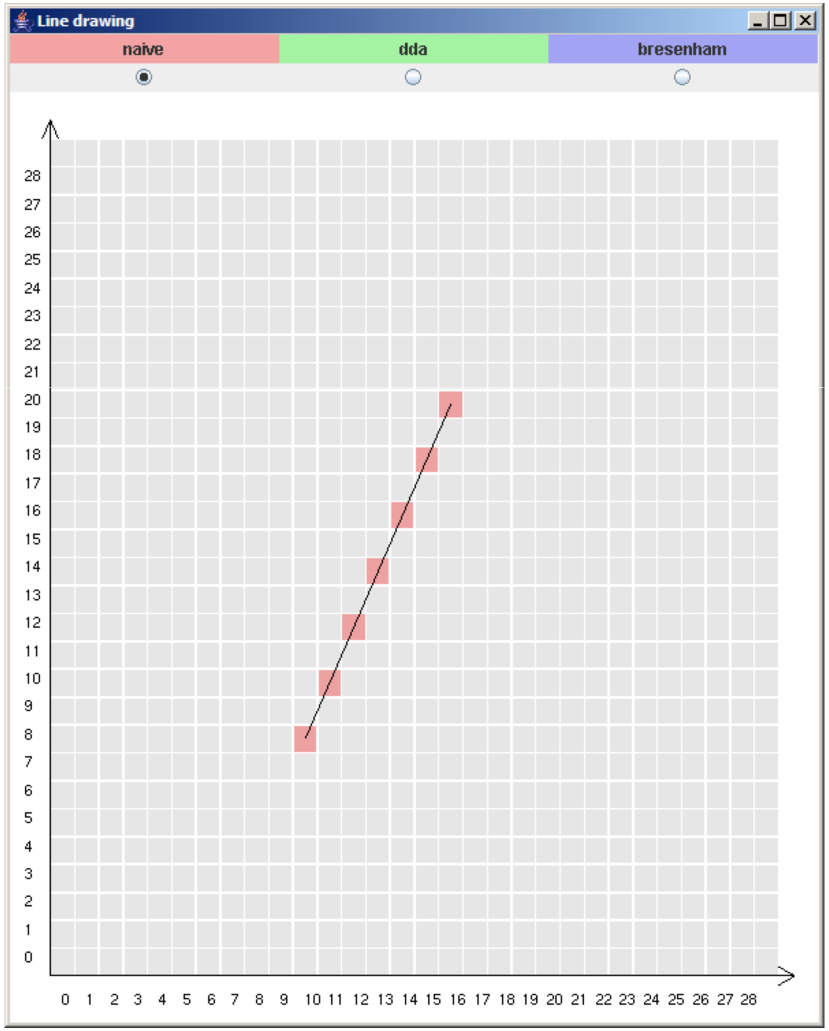
Point B $(X_2, Y_2) = (6,6)$

Point A $(X_1, Y_1) = (0,0)$



Line drawing
naive
dda
bresenham
28
27
26
25
24
23
22
21
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28

A problem!!



Line drawing
naive
dda
bresenham
28
27
26
25
24
23
22
21
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28

The problem with gaps

- m >1 means a change of one pixel in the x direction leads to a change of more than one pixel in the y direction. This results in gaps when we base our calculation on finding a y-value for every possible x. **The same problem occurs when m < -1**.
- **Solution: Switch the coordinates around! If the absolute value of m (disregarding its sign) |m| is greater than 1, loop for every possible y and work out the corresponding x.**

(y = mx + c can be re-written as x = y/m - c/m)

- Try this in the line drawing application (see code for this week's workshop)

Making it faster

Different operations involve different processes – and take different times.

As a general rule:

- Addition and subtraction are faster than multiplication and division.
- Operations on integers are faster than operations on floating point numbers.
  - Minimise FLOPS (floating point operations)
- Loops imply that whatever is in them is executed multiple times, take as much code out of the loop body as possible to make the program run faster.

Making it faster

```
m = (y2-y1)/(x2-x1); // m is a floating point number
c = y1 – m*x1; // c is a floating point number

If (abs(m)<=1) // floating point comparison
  for (x = x1 to x2)
    y = m*x + c; // floating point multiply and add
    putPixel(x,y);
end for
else
for (y=y1 to y2)
    x = y/m - c/m; // floating point division x 2 and floating point subtraction
    putPixel(x,y);
  end for
end if
```

Making it faster 1

The DDA Algorithm

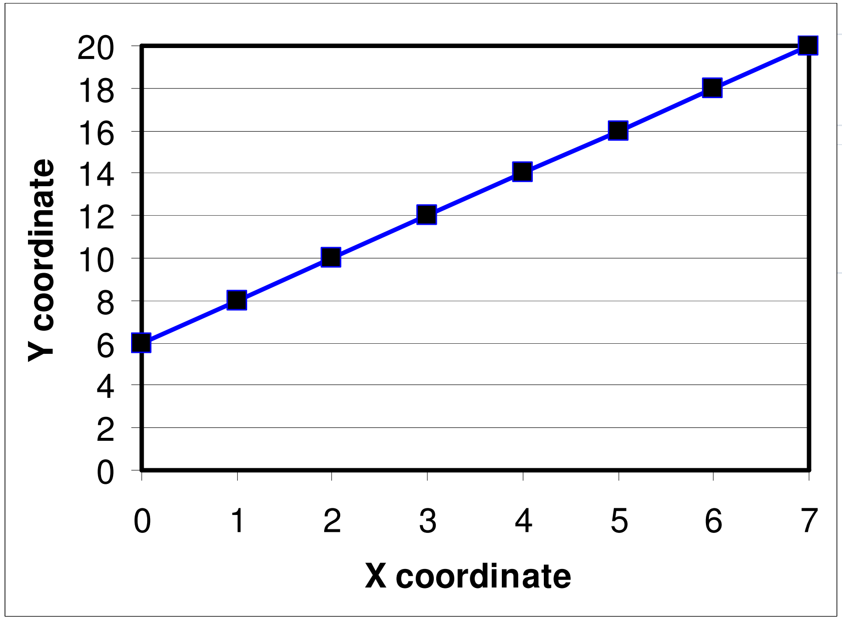
- Stands for: Digital Differential Analyser
- Faster way of drawing a line

Making it faster 1 - dda

- In the last program, we solve the line equation for each step – this can be simplified.
- Look at the line in the figure – if we know the coordinates of one point (x_1,y_1), what is the y-coordinate of the point at $x = x_1 + 1$?
- The gradient, m, tells us by how much the y-coordinate changes for a change of 1 unit in the x-coordinate.
- So the next point is (x_1+1,y_1+m)

Check this out on the graph!

y-coords are 6,8,10… always adding 2



Line: y = 2x + 6

So, instead of solving the line equation at each loop iteration - where gradient < 1 can just add the gradient (m) to the previous y-coordinate to get the next one. If gradient >= 1 add 1/m to the next x coordinate.

Making it faster 1 - dda

Old Code

```
m = (y2-y1)/(x2-x1);
c = y1 – m*x1;

If (abs(m) <=1)
  for (x = x1 to x2)
    y = m*x + c;
    putPixel(x,y);
end for
else
  for (y=y1 to y2)
    x = y/m - c/m;
    putPixel(x,y);
  end for
end if
```

New DDA Code

```
m = (y2-y1)/(x2-x1);
c = y1 – m*x1;

If (abs(m) <=1)
  y = y1; //initialise for the first point
  for (x = x1 to x2)
    putPixel(x,y);
    y = y + m; // 1 floating point addition!
end for
else
x = x1; // initialise the first point
  for (y=y1 to y2)
    putPixel(x,y);
    x = x + 1/m; // still 2 FLOPS!
  end for
end if
```

Making it faster 1 - dda

New DDA Code

```
m = (y2-y1)/(x2-x1);
c = y1 – m*x1;

If (abs(m) <=1)
  y = y1; //initialise for the first point
  for (x = x1 to x2)
    putPixel(x,y);
    y = y + m; // 1 floating point addition!
end for
else
  for (y=y1 to y2)
    putPixel(x,y);
    x = x + 1/m; // still 2 FLOPS!
  end for
end if
```

Even better DDA Code

```
m = (y2-y1)/(x2-x1);
c = y1 – m*x1;

If (abs(m) <=1)
  y = y1; //initialise for the first point
  for (x = x1 to x2)
    putPixel(x,y);
    y = y + m; //
end for
else
m = 1/m; // 1 FLOP taken out of loop
x = x1; // initialise the first point
  for (y=y1 to y2)
    putPixel(x,y);
    x = x + m; // 1 floating point addition!
  end for
end if
```

Making it faster 1 - dda

Why is it called DDA (Digital Differential Analyser)???

Because we only consider the **difference** (m, or 1/m) between the last coordinate and the next to calculate the next point.

This is an example of gaining efficiency using coherence (in this case, spatial)

A problem with DDA:

Pixels that represent the line are only close approximates to where the actual line should be – this introduces some error.

Using the last pixel coordinate (containing some error) and using it to calculate the next pixel coordinate by adding m (and rounding again to get a pixel coordinate) introduces more error.

So for the DDA – error in the line grows with line length.

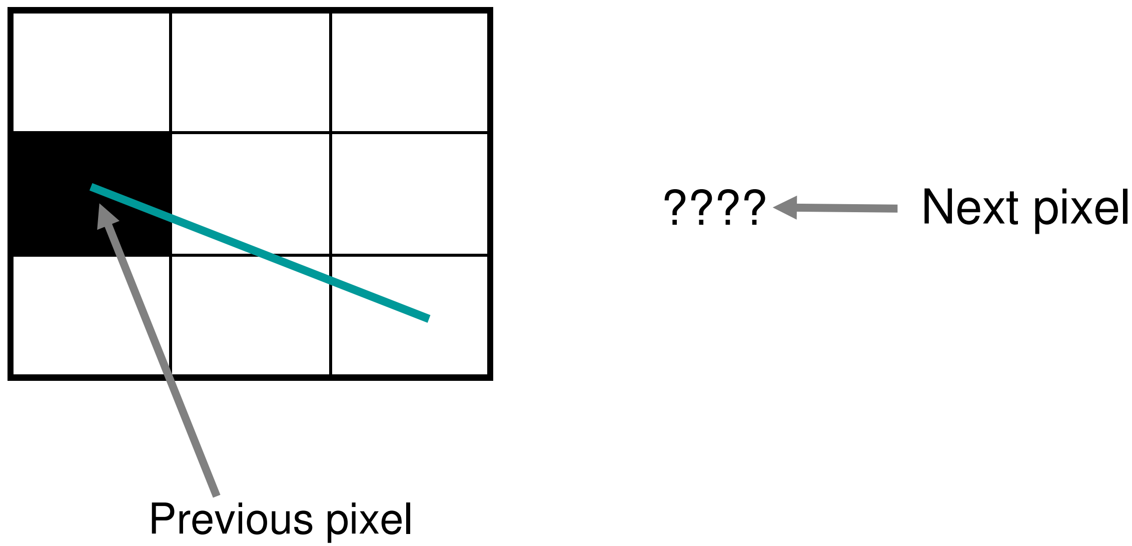
Making it faster 2

Bresenham’s line drawing algorithm

- Reduce line calculation to integer operation
- Reference line equation at each step – avoids errors seen in DDA

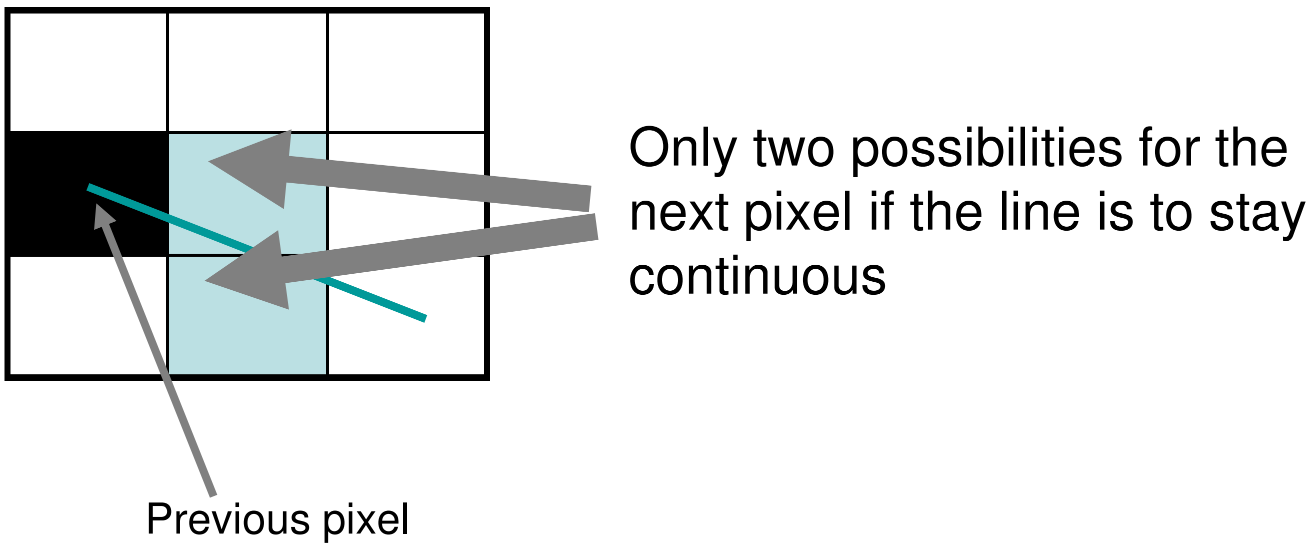
Brensenham's algorithm

- Assuming we have switched our variables around according to slope in order to draw lines without gaps
- If the last pixel was drawn in position (x_1,y_1) and we are iterating across the x-axis, there are only two possible y-coordinate locations of the pixel at x-coordinate = (x_1+1) that will result in a continuous line.
  - Keep the y-coordinate the same (horizontal line between the two pixels)
  - Move the y-coordinate one pixel in the direction of the gradient.



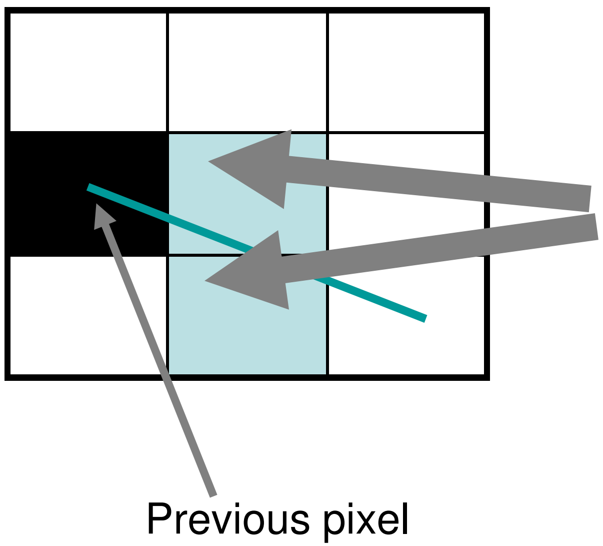
Brensenham's algorithm

- Assuming we have switched our variables around according to slope in order to draw lines without gaps
- If the last pixel was drawn in position (x_1,y_1) and we are iterating across the x-axis, there are only two possible y-coordinate locations of the pixel at x-coordinate = (x_1+1) that will result in a continuous line.
  - Keep the y-coordinate the same (horizontal line between the two pixels)
  - Move the y-coordinate one pixel in the direction of the gradient.



Brensenham's algorithm

- Assuming we have switched our variables around according to slope in order to draw lines without gaps
- If the last pixel was drawn in position (x_1,y_1) and we are iterating across the x-axis, there are only two possible y-coordinate locations of the pixel at x-coordinate = (x_1+1) that will result in a continuous line.
  - Keep the y-coordinate the same (horizontal line between the two pixels)
  - Move the y-coordinate one pixel in the direction of the gradient.

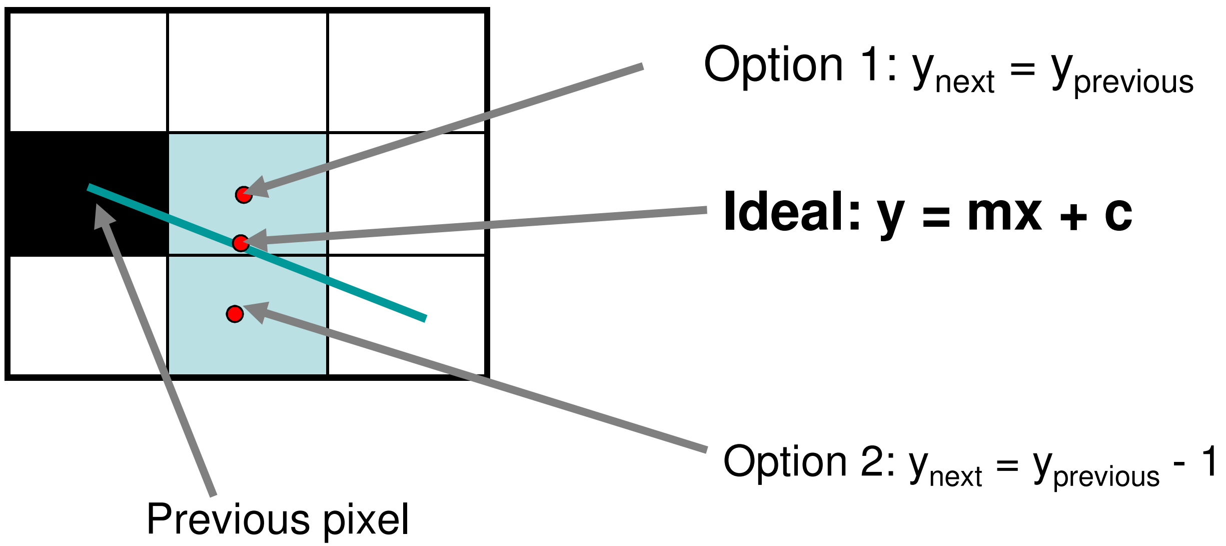


Q: So which one do we draw?

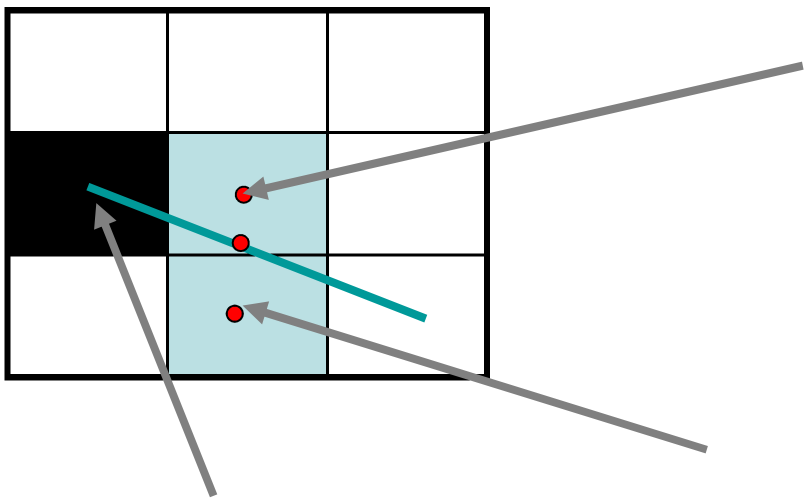
A: The one whose location is closest to that given by the 'ideal' line
$y = mx + c$

Brensenham's algorithm

- Assuming we have switched our variables around according to slope in order to draw lines without gaps
- If the last pixel was drawn in position (x_1,y_1) and we are iterating across the x-axis, there are only two possible y-coordinate locations of the pixel at x-coordinate = (x_1+1) that will result in a continuous line.
  - Keep the y-coordinate the same (horizontal line between the two pixels)
  - Move the y-coordinate one pixel in the direction of the gradient.



Brensenham's algorithm



Distance 1:

$= |y_{Ideal} - y_{previous}|$

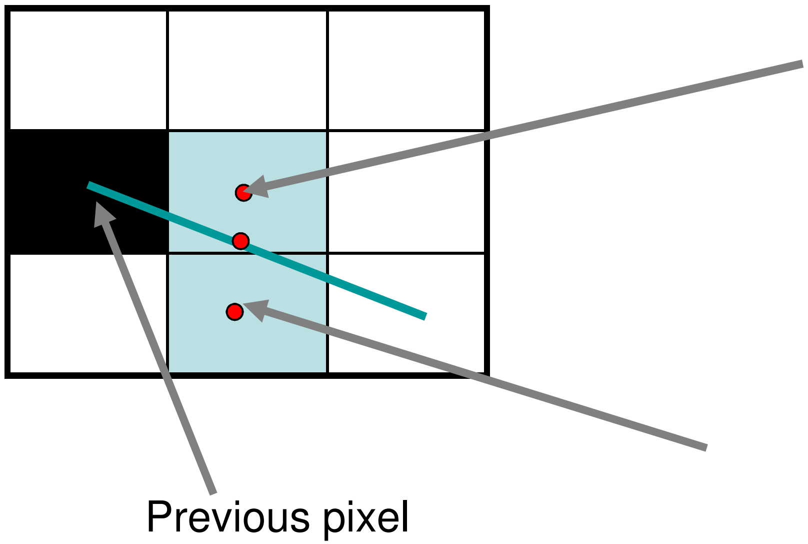
$= |(mx + c) - y_{previous}|$

Distance 2:

$= |y_{Ideal} - (y_{previous} - 1)|$

$= |(mx + c) - (y_{previous} - 1)|$

Brensenham's algorithm



Distance 1:

$= |y_{Ideal} - y_{previous}|$

$= |(mx + c) - y_{previous}|$

Distance 2:

$= |y_{Ideal} - (y_{previous} - 1)|$

$= |(mx + c) - (y_{previous} - 1)|$

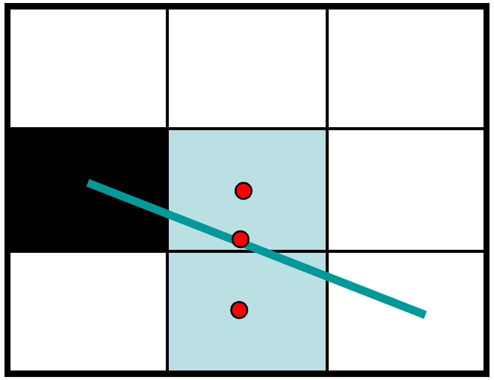
If (distance1 – distance2) is positive

- Choose option 2

If (distance1 – distance2) is negative

- Choose option 1

Brensenham's algorithm



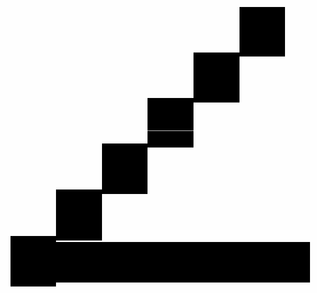
- The concept behind Bresenham's line algorithm is simple – choose based on the closest distance.
- As presented so far – it's readable (we can work out what's going on) but less efficient than the DDA algorithm (more floating point operations).
- The 'genius' behind Bresenham's algorithm is that the previous equations can be reduced to all integer operations. (see Hearn & Baker p95-99, or 4th ed. p140-144 for derivation and example)
- Because the distance calculated is with reference to the ideal pixel position at each step – the incremental error seen in the DDA algorithm is avoided.

Some line drawing problems

- Lines should have constant density i.e equal spacing of pixels and pixels/line length should ideally remain constant
- Problems with lines on screen
  - Unequal intensity
  - Aliasing (“jaggies” and “crawling ants”)

Unequal intensity

- Line at 45 degree angle is lower intensity because it is longer yet same number of pixels

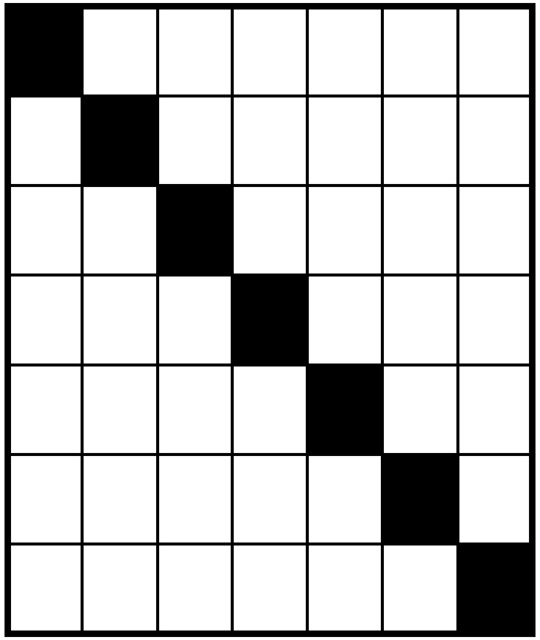


- Could compensate by adjusting intensity depending on slope of line

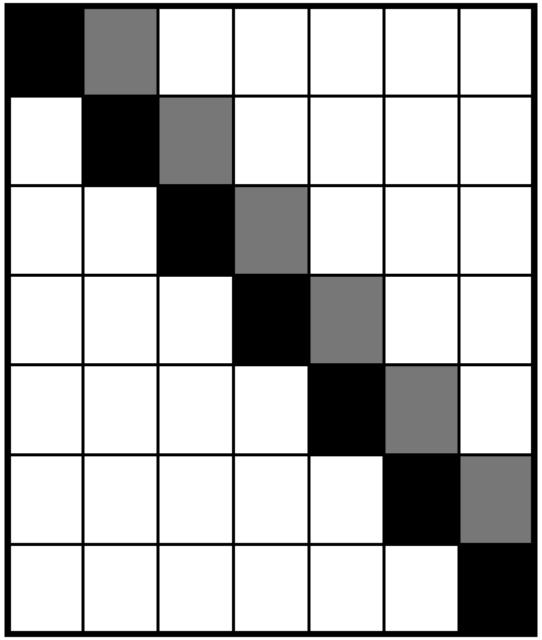
Aliasing

- Plotting a point in a position other than true position
- Pixels are plotted at integer positions resulting in jagged lines
- Improve by using higher resolution and by using anti-aliasing algorithms
- Anti-aliasing algorithms – see Hearn and Baker 4.17 (or Hearn, Baker & Carithers 6.15)
  - Numerous, for example:
    - Blurring the line edges by inserting pixels
    - Model a line as a thin rectangle and calculate area of intersection between rectangle and pixel
    - Adjusting pixel intensity depending on distance from true centre of line

Anti-aliasing (one idea)



Jagged looking line



Line made to look smoother (blurred) from a distance by inserting lighter coloured pixels in the gaps

Polylines

- Polylines are collections of joined straight lines.
- Defined as a series of points
  - Eg. P1 = (x1,y1), P2 = (x2,y2), P3 = (x3,y3)
- To draw them, simply iterate over the points and use the line drawing algorithms previously discussed.

```
Line(P1,P2);
Line(P2,P3);
```

- Important that end points join up!

Other curves *

- See Hearn and Baker 3.11 or Hearn, Baker & Carithers 6.6
  - Conic sections (parabolas, ellipse, hyperbolas)
  - Splines (we cover this later)
- Ellipses – can use similar idea to Bresenham's algorithm
  - See Hearn and Baker 3.10 or Hearn, Baker & Carithers 6.5
  - Midpoint-ellipse algorithm

* Not examined