Dr. George Karraz, Ph. D.

Line Drawing Algorithms

Contents

Graphics hardware

The problem of scan conversion

Considerations

Line equations

Scan converting algorithms

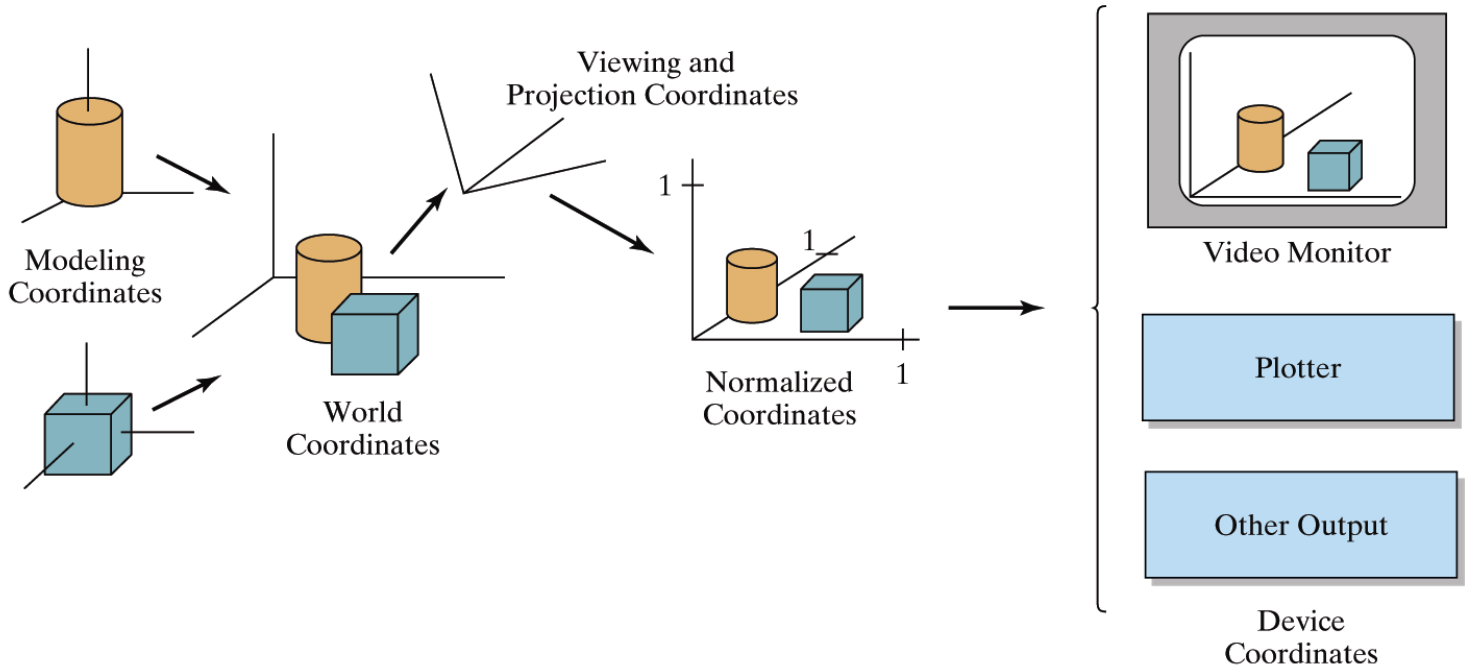
- A very simple solution
- The DDA algorithm

Conclusion

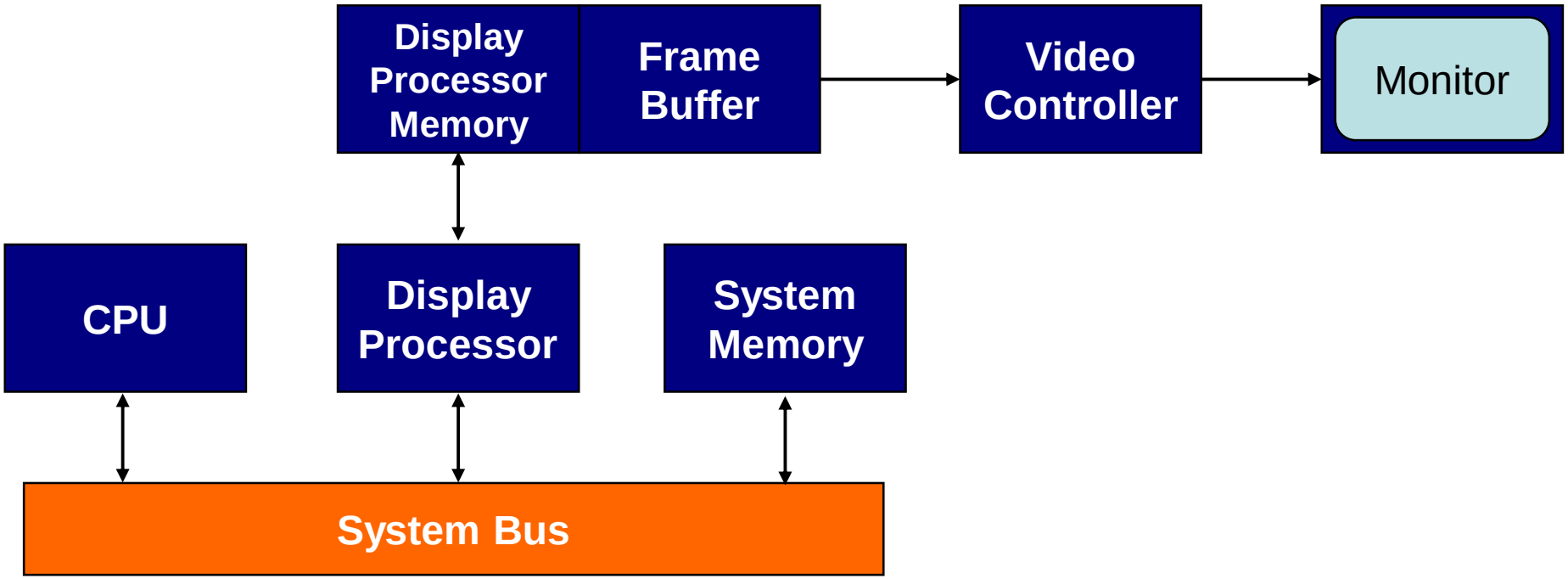
Graphics Hardware

It's worth taking a little look at how graphics hardware works before we go any further

How do things end up on the screen?



Architecture Of A Graphics System



Output Devices

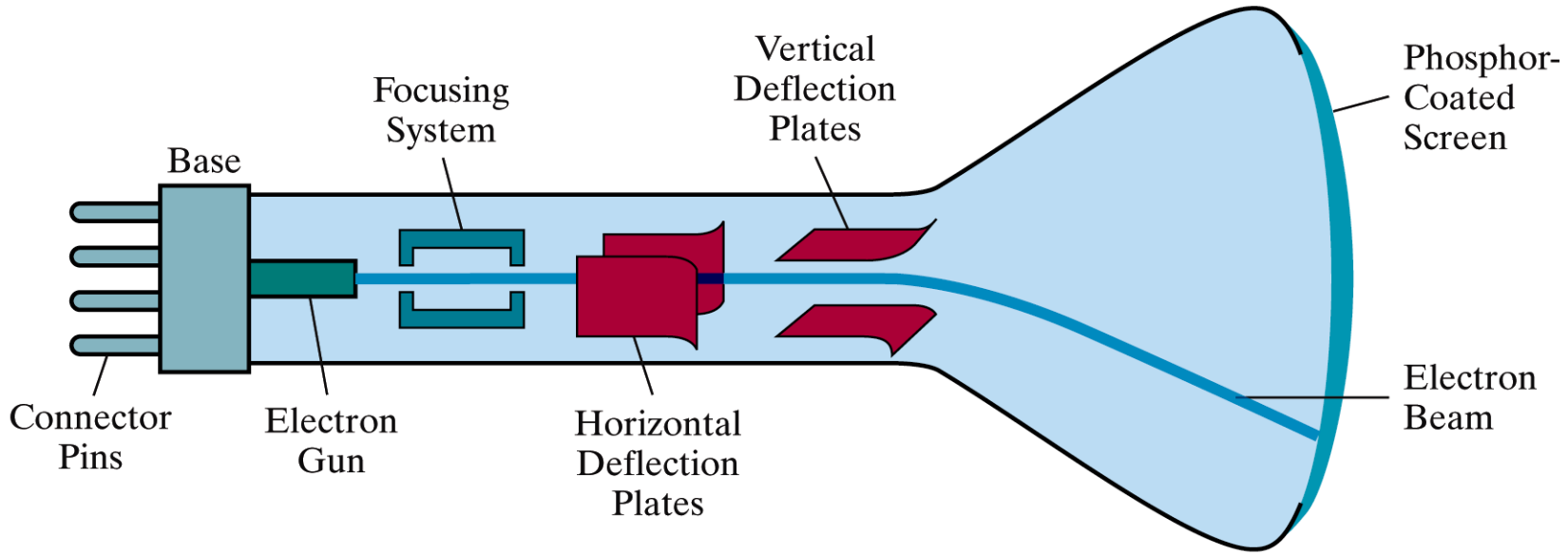
There are a range of output devices currently available:

- Printers/plotters
- Cathode ray tube displays
- Plasma displays
- LCD displays
- 3 dimensional viewers
- Virtual/augmented reality headsets

We will look briefly at some of the more common display devices

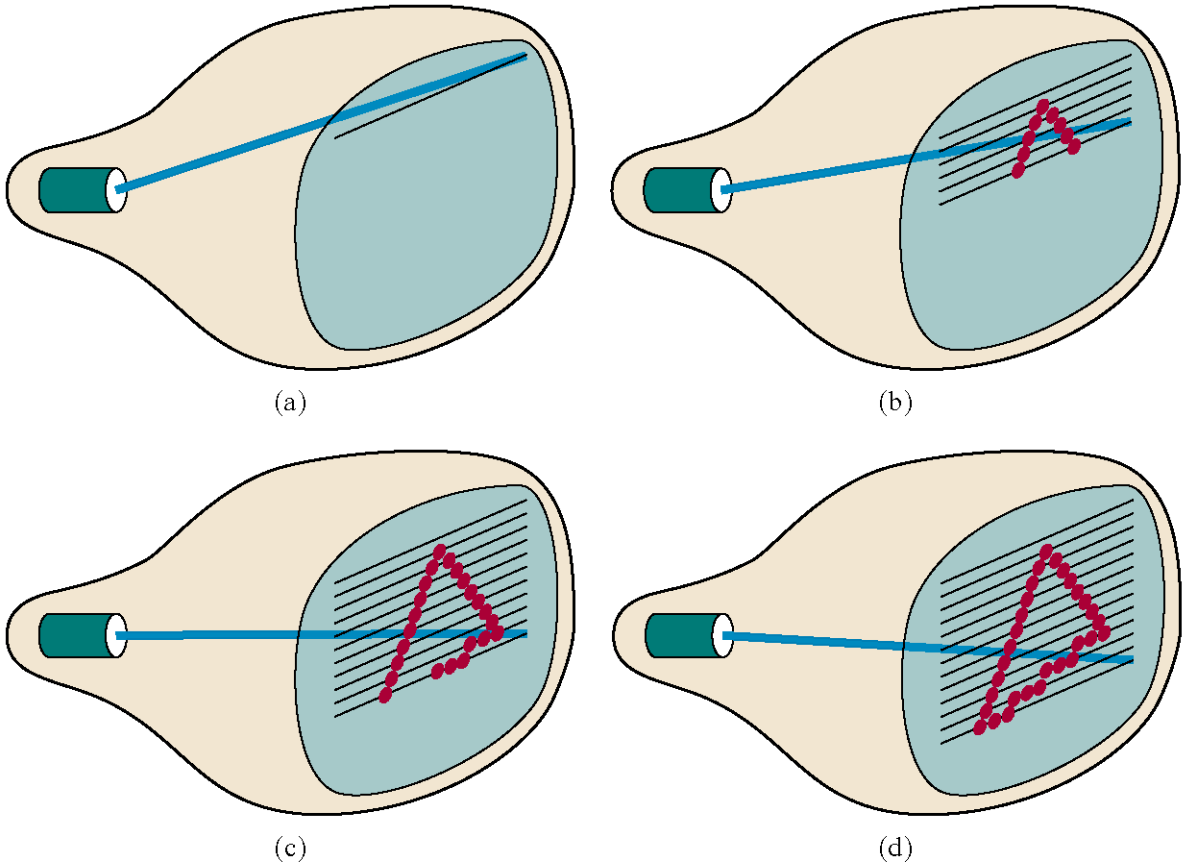
Basic Cathode Ray Tube (CRT)

Fire an electron beam at a phosphor coated screen



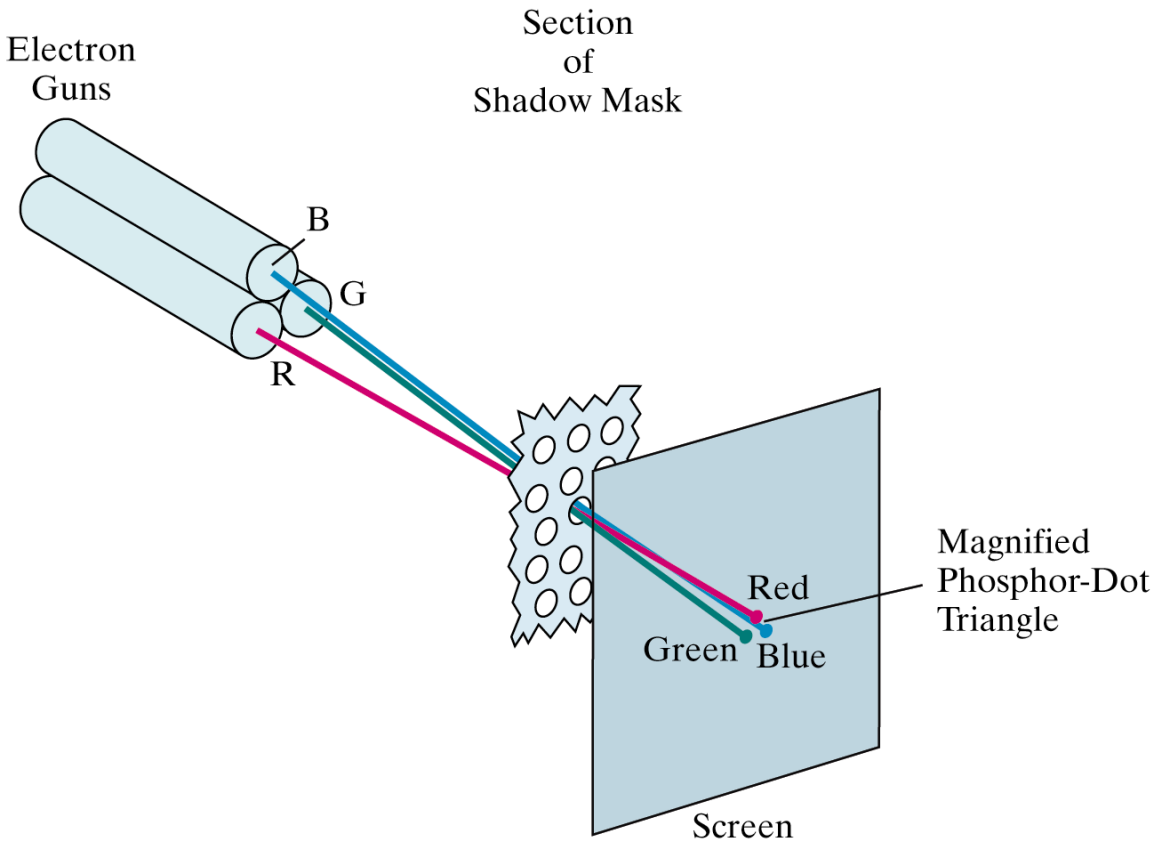
Raster Scan Systems

Draw one line at a time



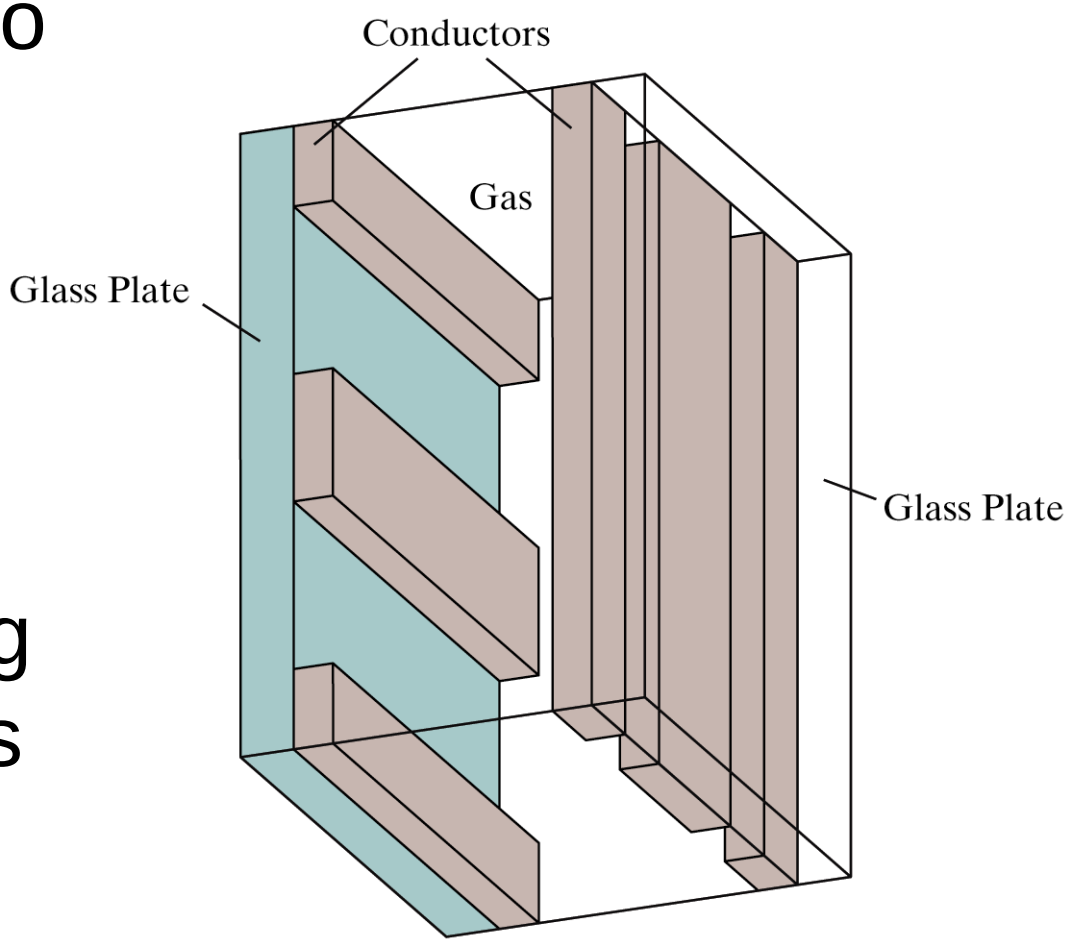
Colour CRT

An electron gun for each colour – red, green and blue

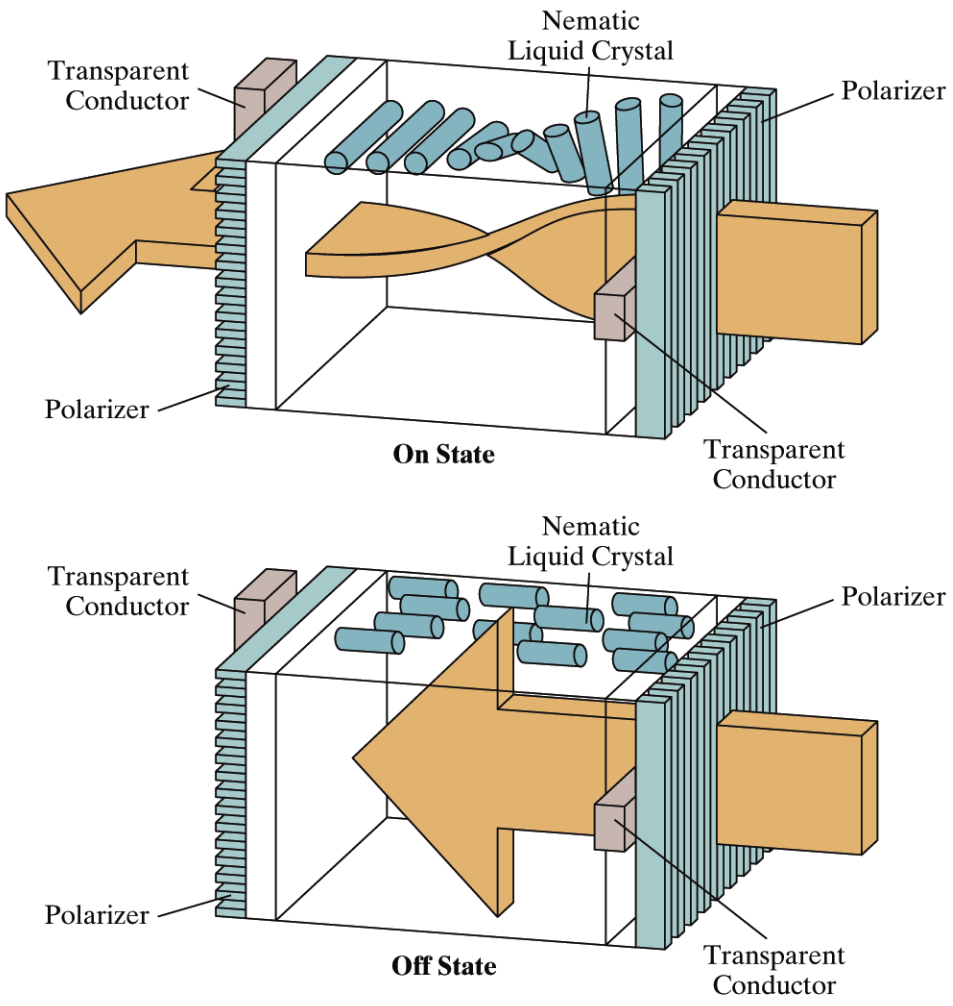


Plasma-Panel Displays

Applying voltages to crossing pairs of conductors causes the gas (usually a mixture including neon) to break down into a glowing plasma of electrons and ions



Liquid Crystal Displays

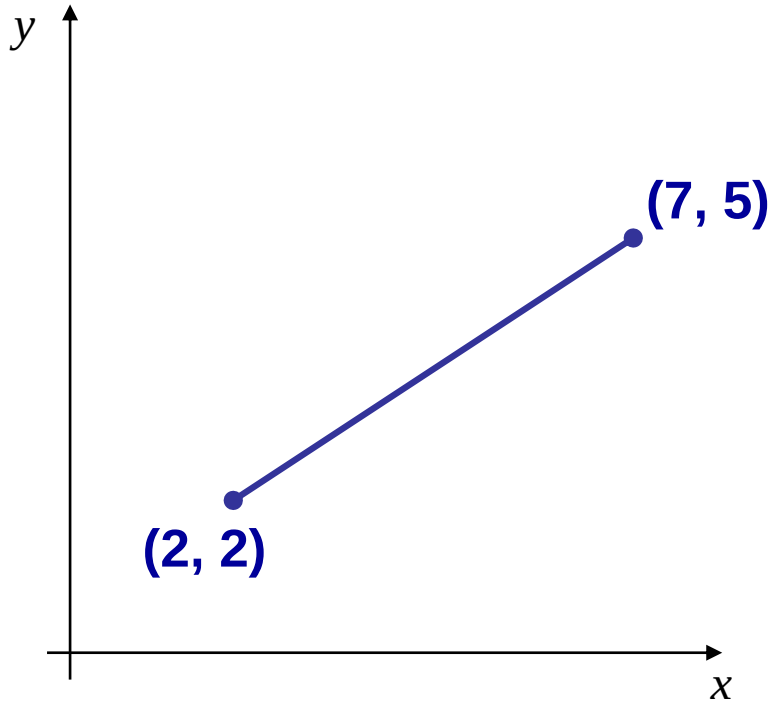


Light passing through the liquid crystal is twisted so it gets through the polarizer

A voltage is applied using the crisscrossing conductors to stop the twisting and turn pixels off

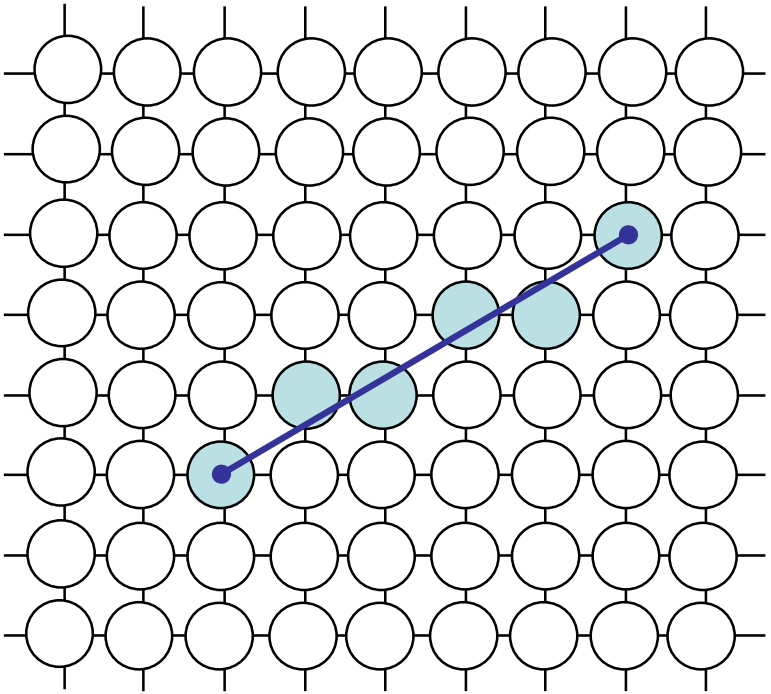
The Problem Of Scan Conversion

A line segment in a scene is defined by the coordinate positions of the line end-points



The Problem (cont…)

But what happens when we try to draw this on a pixel based display?

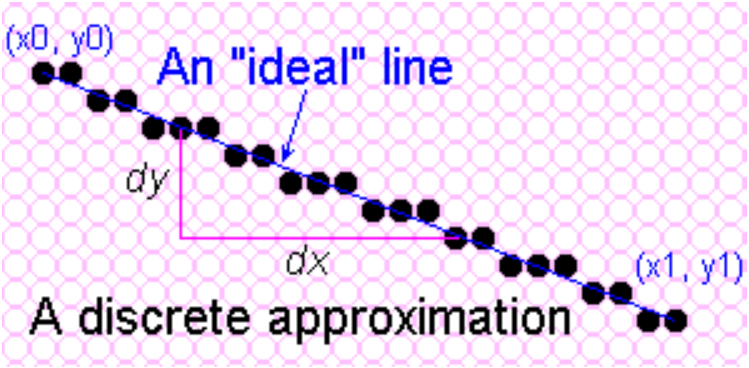


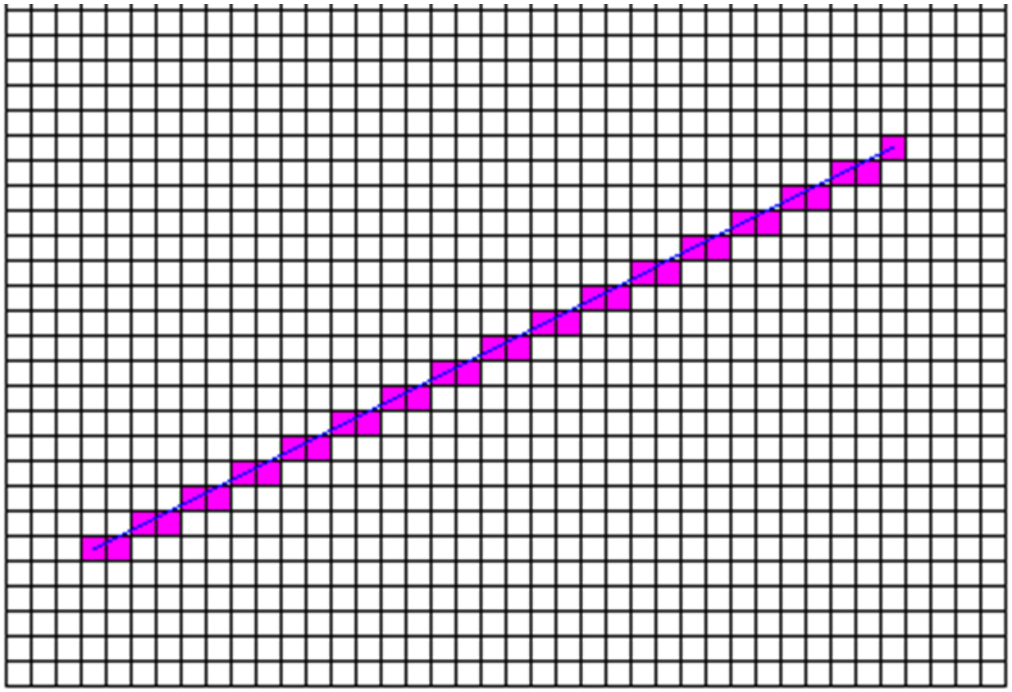
How do we choose which pixels to turn on?

The best we can do is a discrete approximation of an ideal line

Important line qualities:

- Continuous appearence
- Uniform thickness and brightness
- Turn on the pixels nearest the ideal line
- How fast is the line generated





Note that since vector-graphics displays, capable of drawing nearly perfect lines, predated raster-graphics displays. Thus, the expectations for line quality were set very high. The nature of raster-graphics display, however, only allows us to display a discrete approximation of a line, since we are restricted to only turn on discrete points, or pixels. In order to discuss, line drawing we must first consider the mathematically ideal line (or line segment).

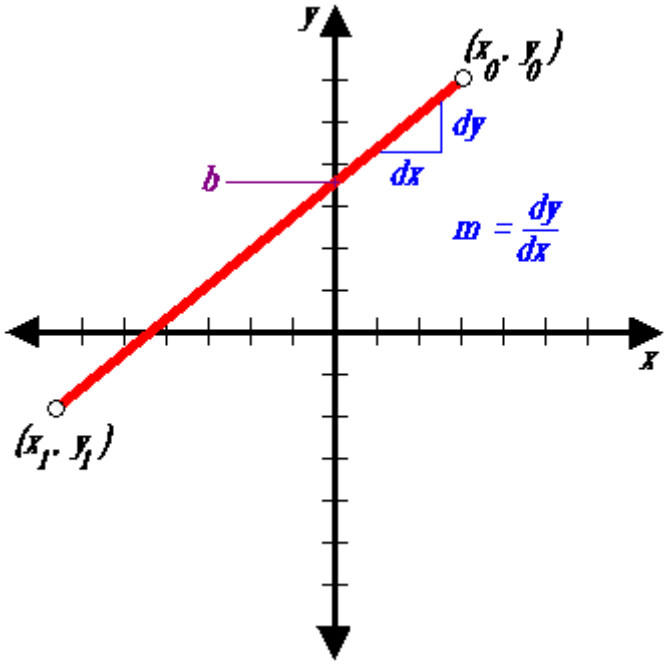
Considerations

Considerations to keep in mind:

- The line has to look good
  - Avoid *jaggies*
- It has to be lightening fast!
  - How many lines need to be drawn in a typical scene?
  - This is going to come back to bite us again and again

Simple Line

The first line-drawing algorithm presented is called the simple slope-intercept algorithm. It is a striaght forward implementation of the slope-intercept formula for a line.

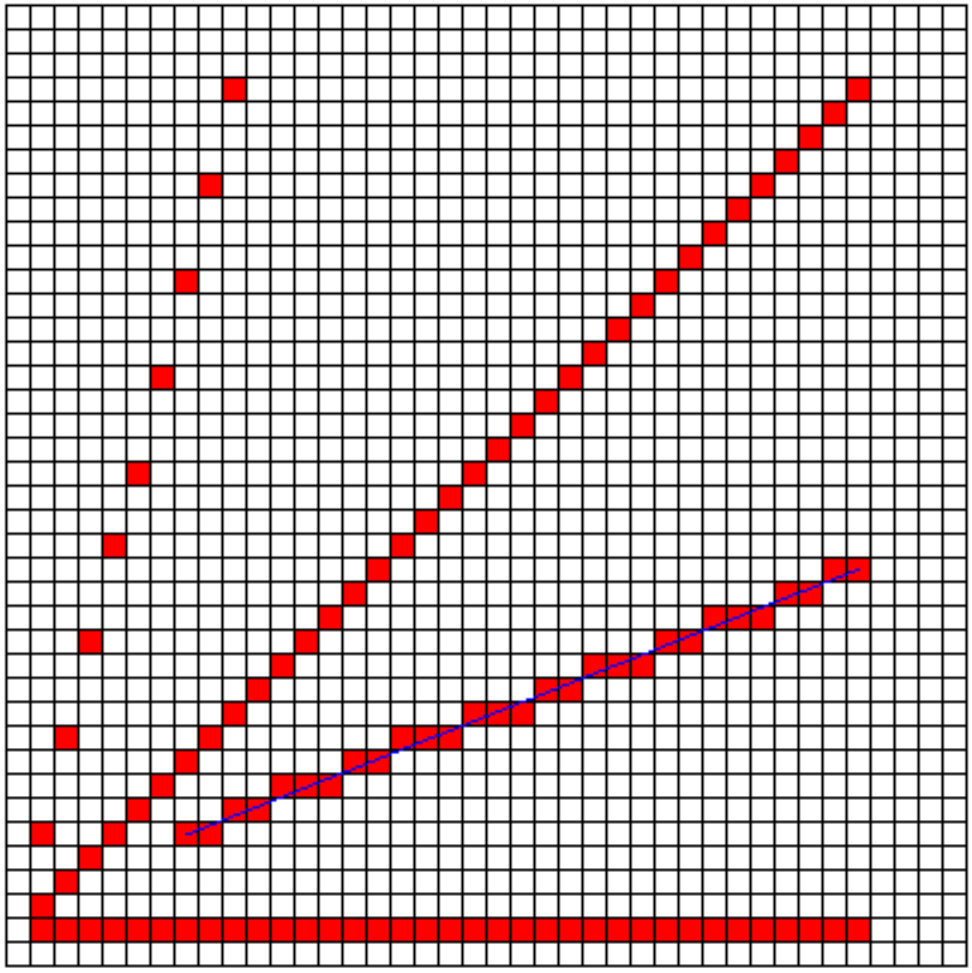


Based on the simple *slope-intercept algorithm*

```
public void lineSimple(int x0, int y0, int x1, int y1, Color color) {
        int pix = color.getRGB();
        int dx = x1 - x0;
        int dy = y1 - y0;

        raster.setPixel(pix, x0, y0);
        if (dx != 0) {
            float m = (float) dy / (float) dx;
            float b = y0 - m*x0;
            dx = (x1 > x0) ? 1 : -1;
            while (x0 != x1) {
                x0 += dx;
                y0 = Math.round(m*x0 + b);
                raster.setPixel(pix, x0, y0);
            }
        }
    }
```

LineSimple with various slopes



Optimize Inner Loops

Optimize those code fragments where the algorithm spends most of its time

- remove unnecessary method invocations

  replace **Math.round(m*x0 + b)**
  with **(int)(m*x0 + b + 0.5)**

- use incremental calculations

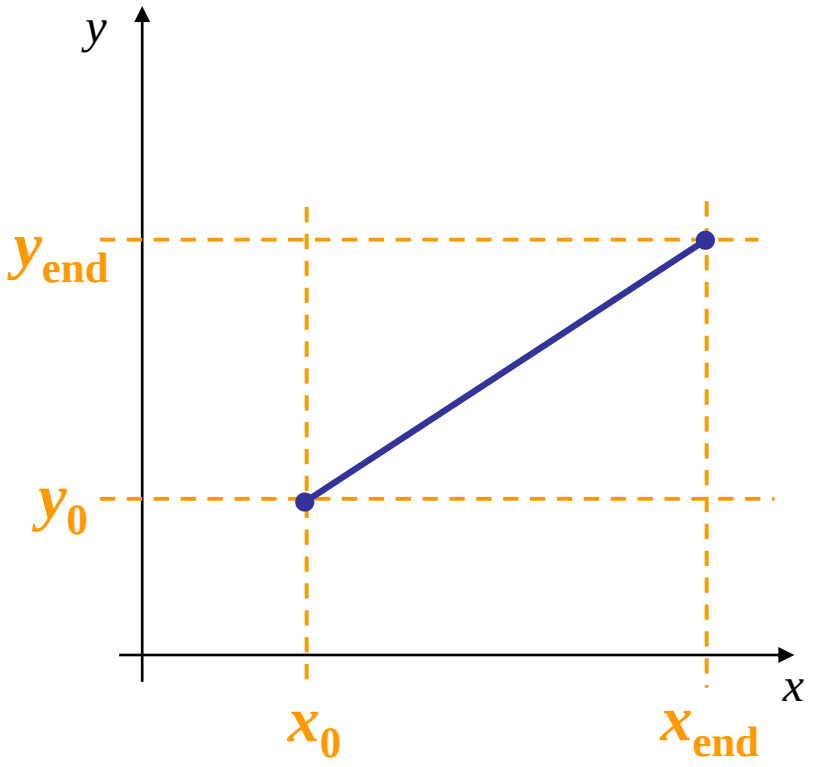
  Consider the expression
  y = (int)(m*x + b + 0.5)

The value of *y* is known at x_0 (i.e. it is $y_0 + 0.5$)
Future values of *y* can be expressed in terms of previous values with a **difference equation**:

$y_{i+1} = y_i + m$;
or
$y_{i+1} = y_i - m$;

Line Equations

Let's quickly review the equations involved in drawing lines



Slope-intercept line equation:

$$y = m \cdot x + b$$

where:

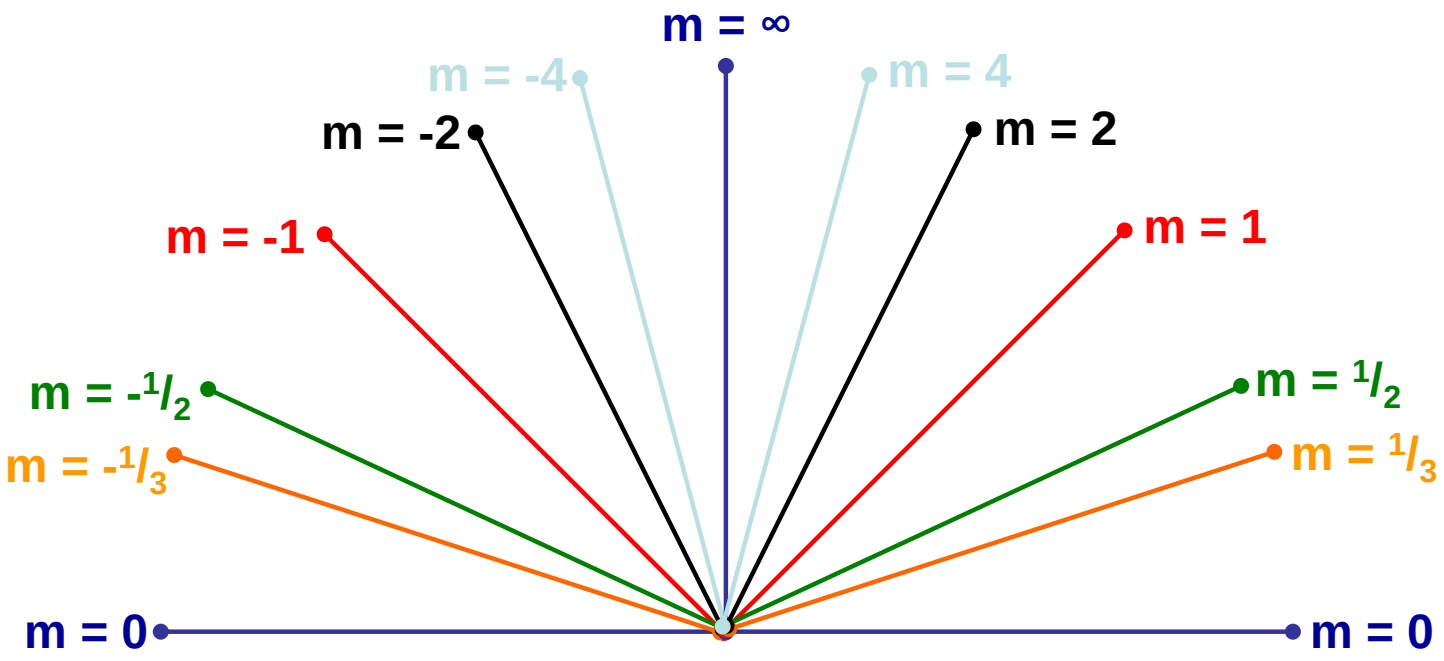
$$m = \frac{y_{end} - y_0}{x_{end} - x_0}$$

$$b = y_0 - m \cdot x_0$$

Lines & Slopes

The slope of a line (m) is defined by its start and end coordinates

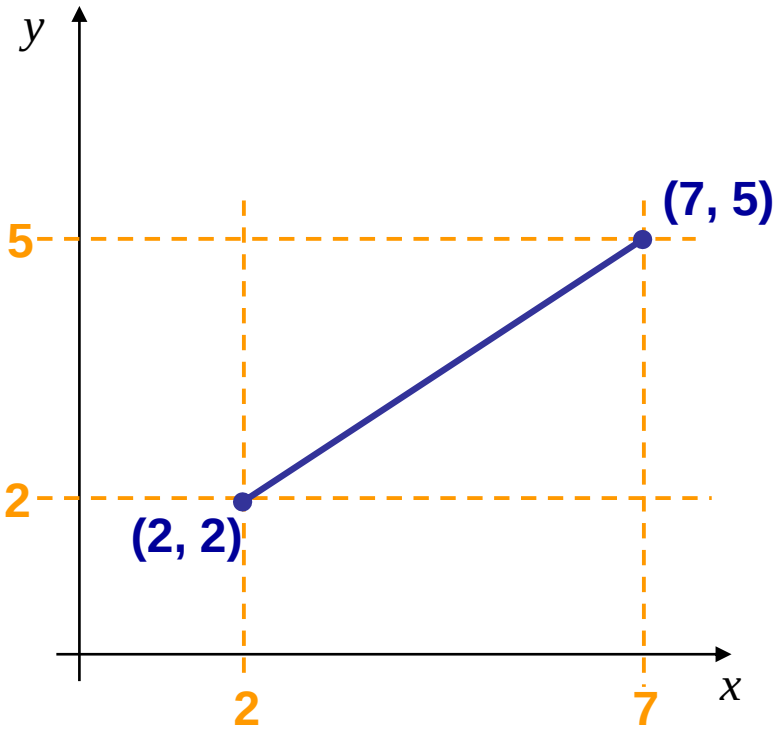
The diagram below shows some examples of lines and their slopes



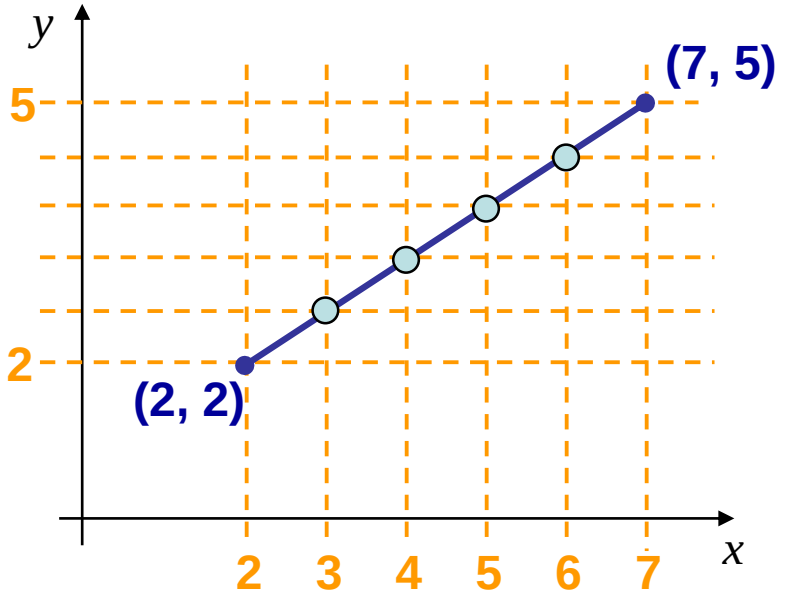
A Very Simple Solution

We could simply work out the corresponding y coordinate for each unit x coordinate

Let's consider the following example:



A Very Simple Solution (cont…)



First work out m and b:

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

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

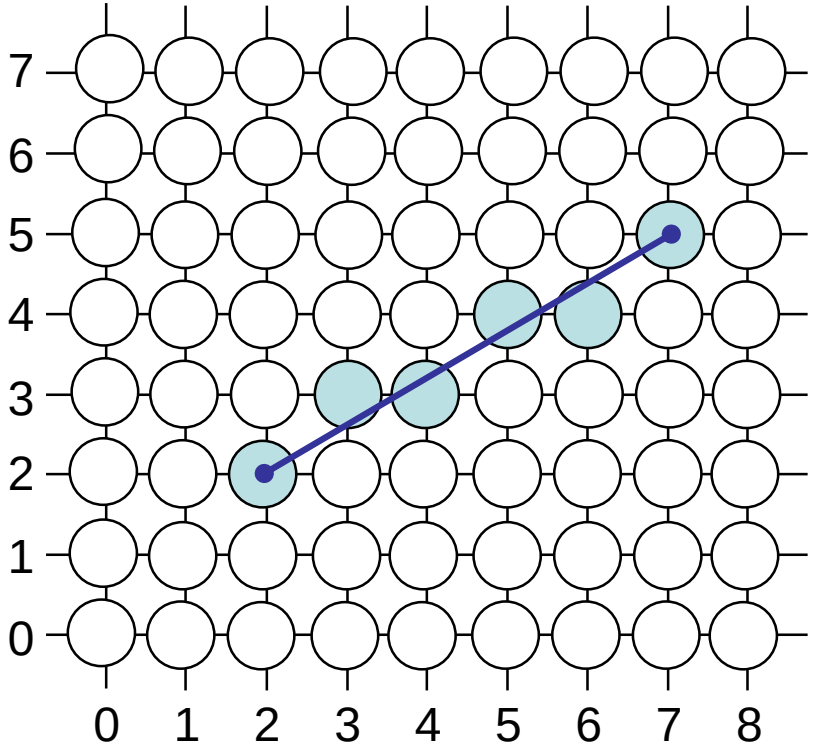
Now for each x value work out the y value:

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

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

A Very Simple Solution (cont…)

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



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

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

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

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

A Very Simple Solution (cont…)

However, this approach is just way too slow

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

A Quick Note About Slopes

In the previous example we chose to solve the parametric line equation to give us the y coordinate for each unit x coordinate

What if we had done it the other way around?

So this gives us: $x = \frac{y - b}{m}$

where: $m = \frac{y_{end} - y_0}{x_{end} - x_0}$ and $b = y_0 - m \cdot x_0$

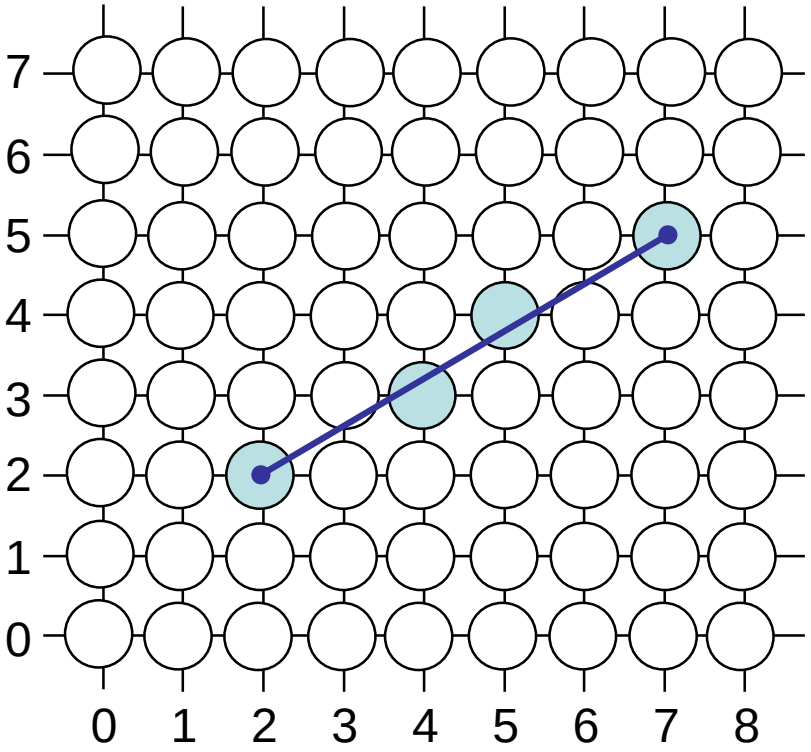
A Quick Note About Slopes (cont…)

Leaving out the details this gives us:

$$x(3) = 3\frac{2}{3} \approx 4 \qquad x(4) = 5\frac{1}{3} \approx 5$$

We can see easily that this line doesn't look very good!

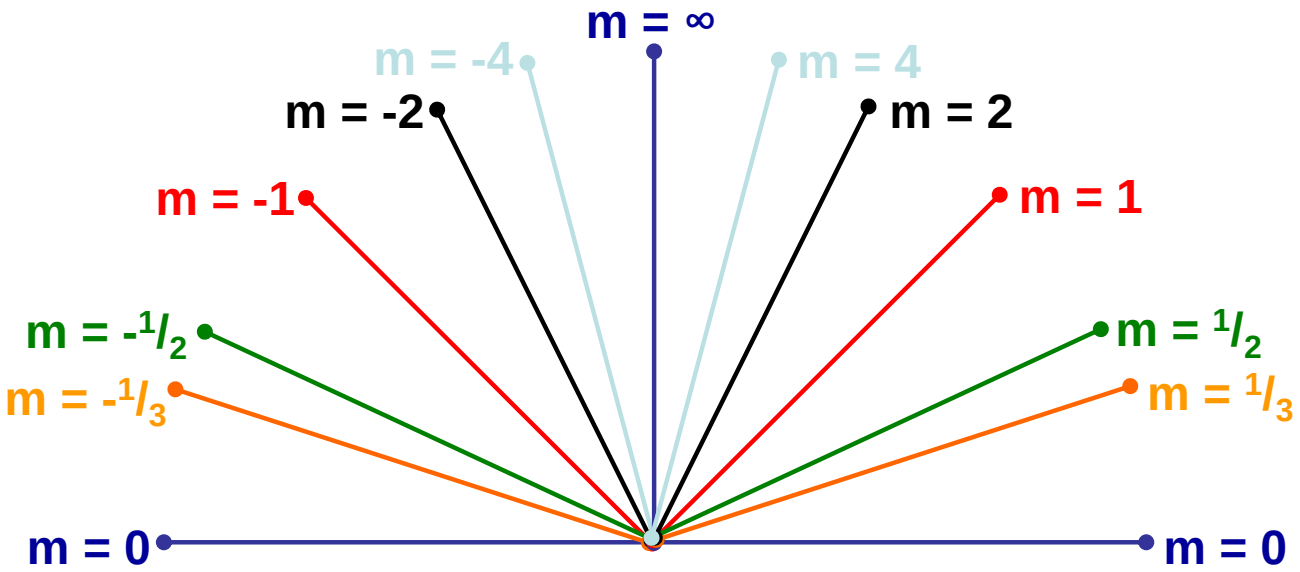
We choose which way to work out the line pixels based on the slope of the line



A Quick Note About Slopes (cont…)

If the slope of a line is between -1 and 1 then we work out the y coordinates for a line based on it's unit x coordinates

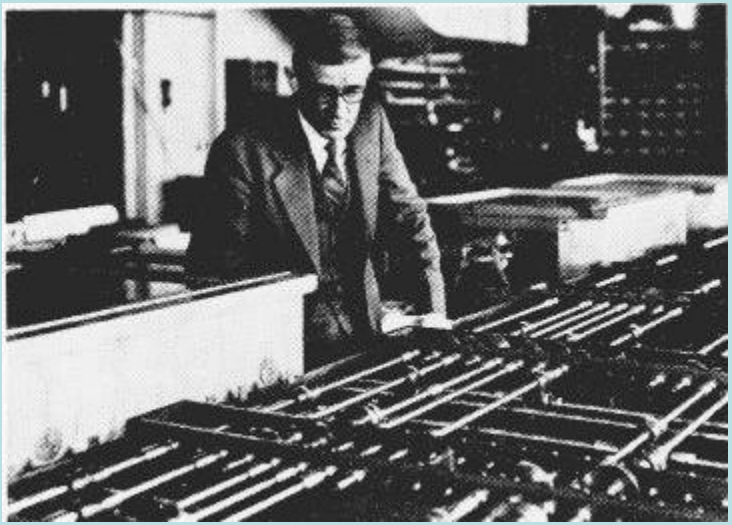
Otherwise we do the opposite – x coordinates are computed based on unit y coordinates



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



The original differential analyzer was a physical machine developed by Vannevar Bush at MIT in the 1930's in order to solve ordinary differential equations.

More information here.

The DDA Algorithm (cont…)

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 (cont…)

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:

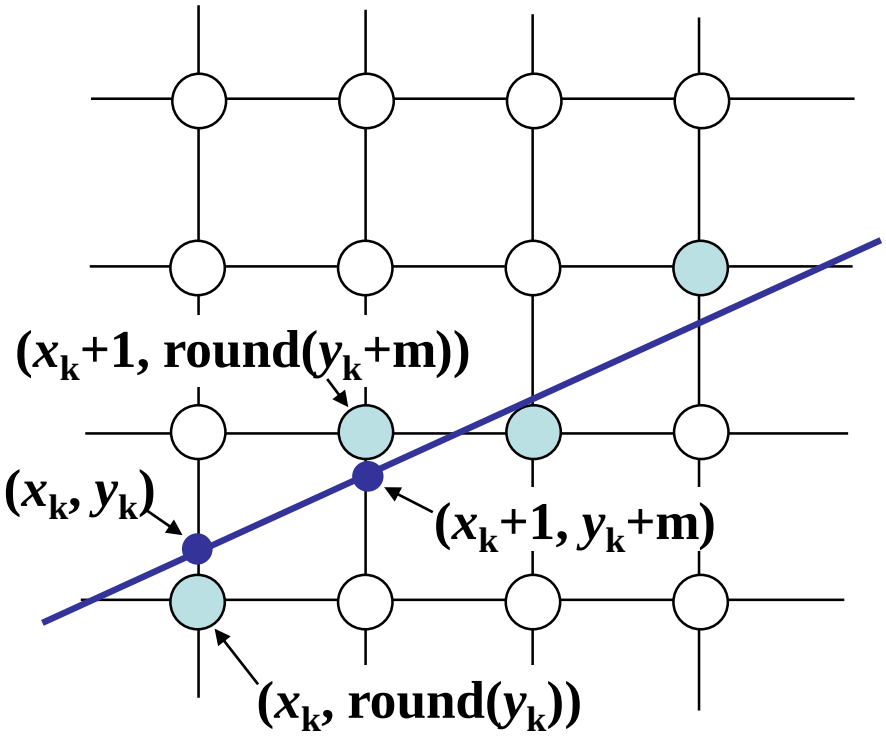
$$y_{k+1} = y_k + m$$

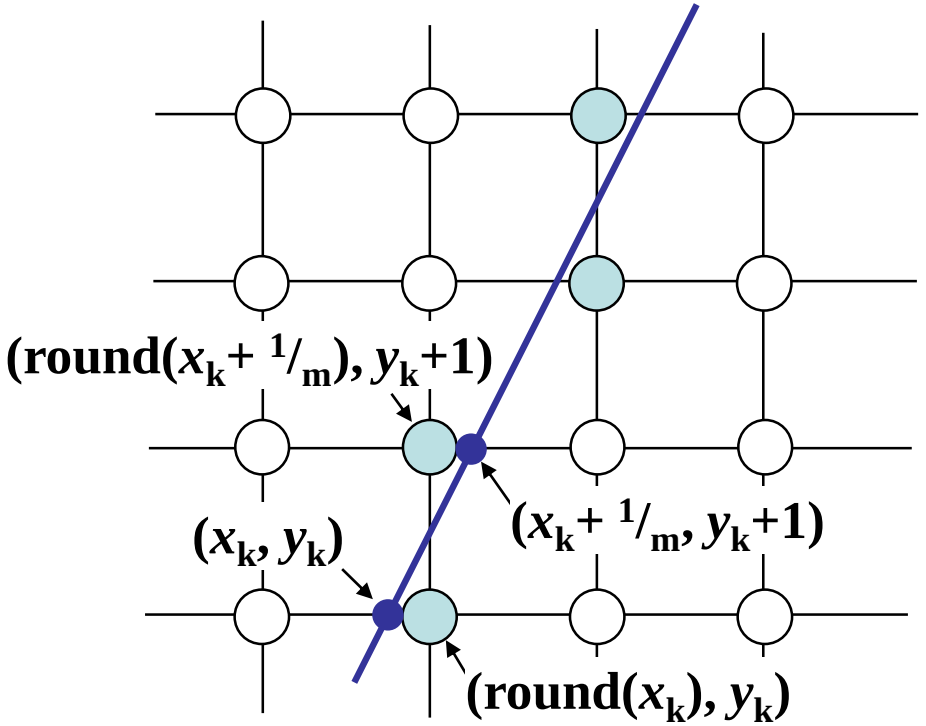
When the slope is outside these limits, increment the y coordinate by 1 and calculate the corresponding x coordinates as follows:

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

The DDA Algorithm (cont…)

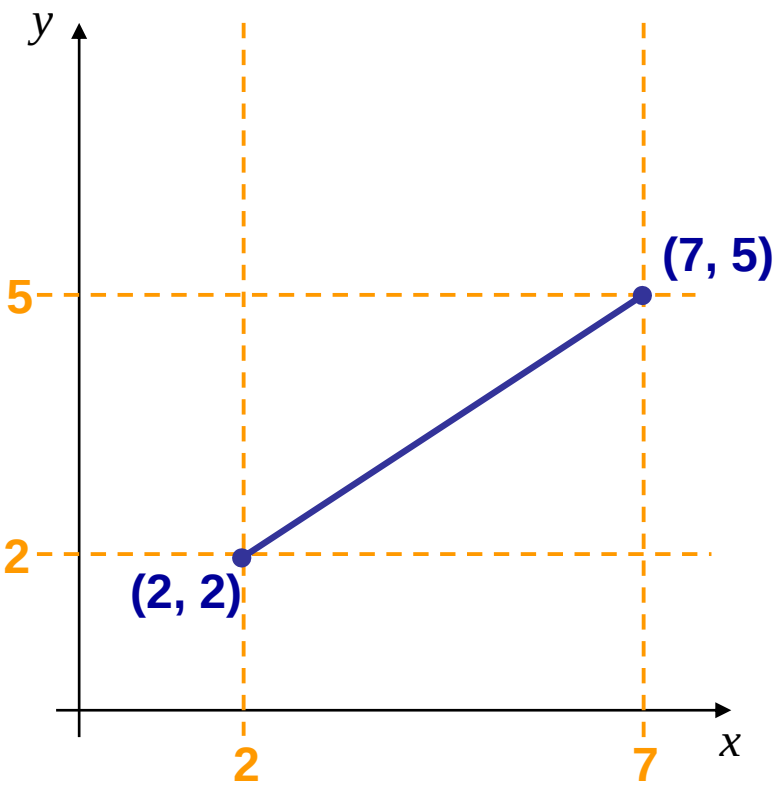
Again the values calculated by the equations used by the DDA algorithm must be rounded to match pixel values

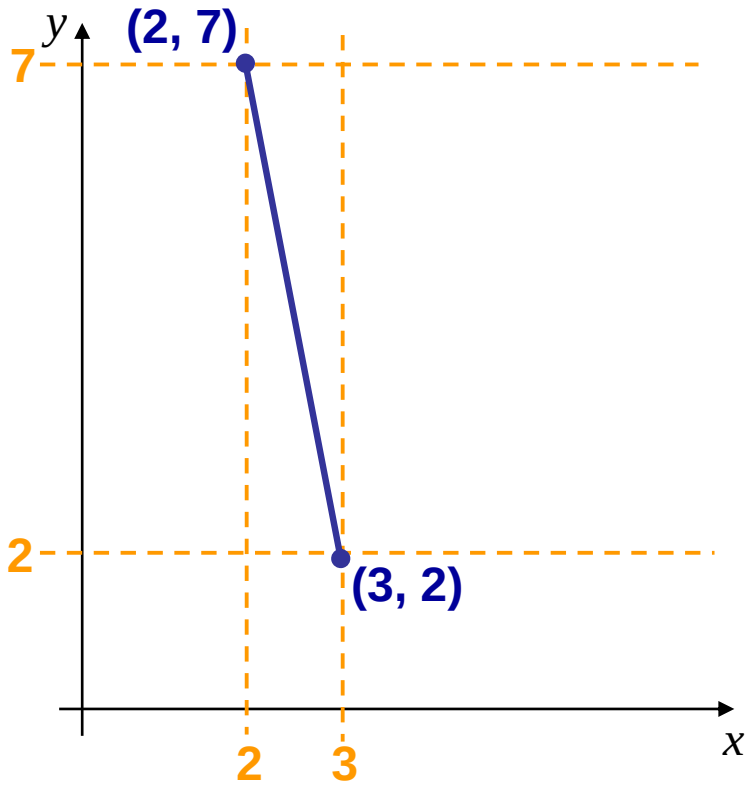




DDA Algorithm Example

Let's try out the following examples:





```
public void lineDDA(int x0, int y0, int x1, int y1, Color color) {
        int pix = color.getRGB();
        int dy = y1 - y0;
        int dx = x1 - x0;
        float t = (float) 0.5;                    // offset for rounding
        raster.setPixel(pix, x0, y0);
        if (Math.abs(dx) > Math.abs(dy)) {          // slope < 1
            float m = (float) dy / (float) dx;      // compute slope
            t += y0;
            dx = (dx < 0) ? -1 : 1;
            m *= dx;
            while (x0 != x1) {
                x0 += dx;                           // step to next x value
                t += m;                             // add slope to y value
                raster.setPixel(pix, x0, (int) t);
            }
        } else {                                    // slope >= 1
            float m = (float) dx / (float) dy;      // compute slope
            t += x0;
            dy = (dy < 0) ? -1 : 1;
            m *= dy;
            while (y0 != y1) {
                y0 += dy;                           // step to next y value
                t += m;                             // add slope to x value
                raster.setPixel(pix, (int) t, y0);
            }
        }
    }
```

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

Conclusion

In this lecture we took a very brief look at how graphics hardware works

Drawing lines to pixel based displays is time consuming so we need good ways to do it

The DDA algorithm is pretty good – but we can do better

Next time we'll like at the Bresenham line algorithm and how to draw circles, fill polygons and anti-aliasing