

Output Primitives

Dr. Subhadip Basu

*Senior Lecturer, CSE Dept., JU
subhadip@cse.jdvu.ac.in*



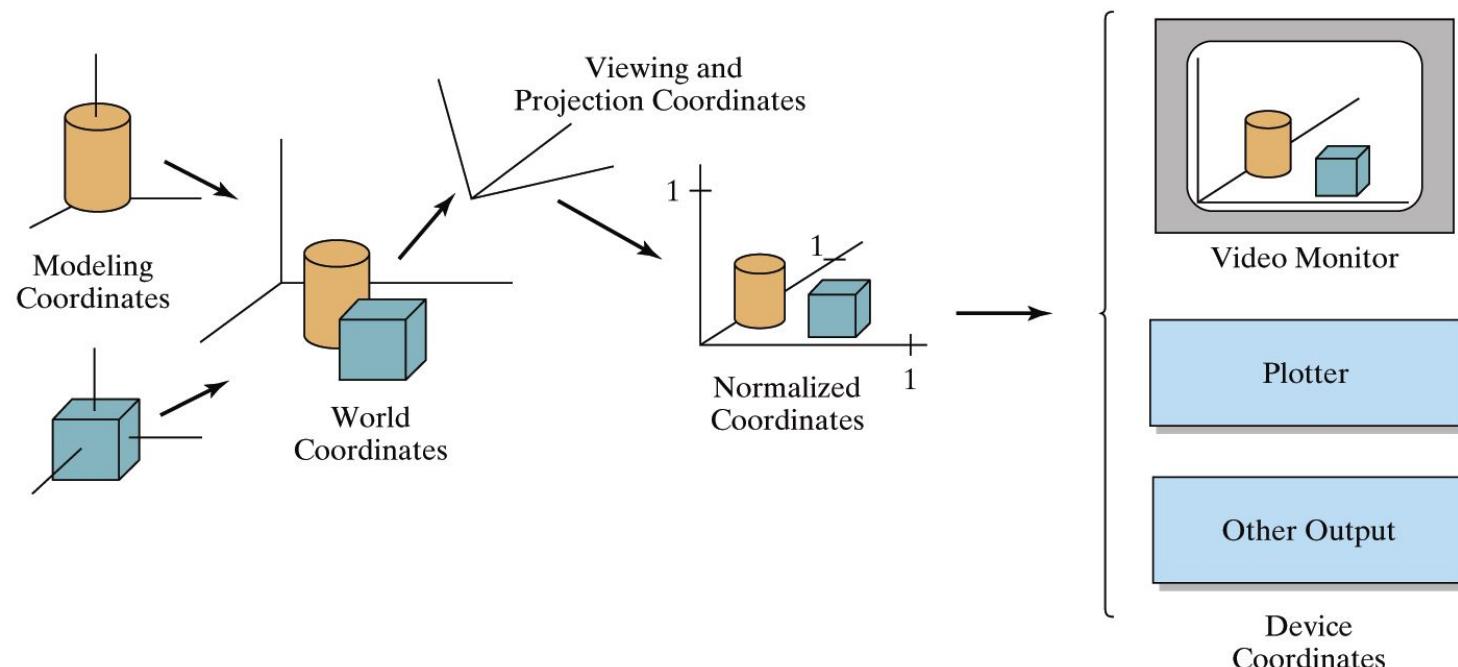
Contents (Part-I)

- Graphics hardware
- The problem of scan conversion
- Considerations
- Line equations
- Scan converting algorithms
 - A very simple solution
 - The DDA algorithm
- Conclusion



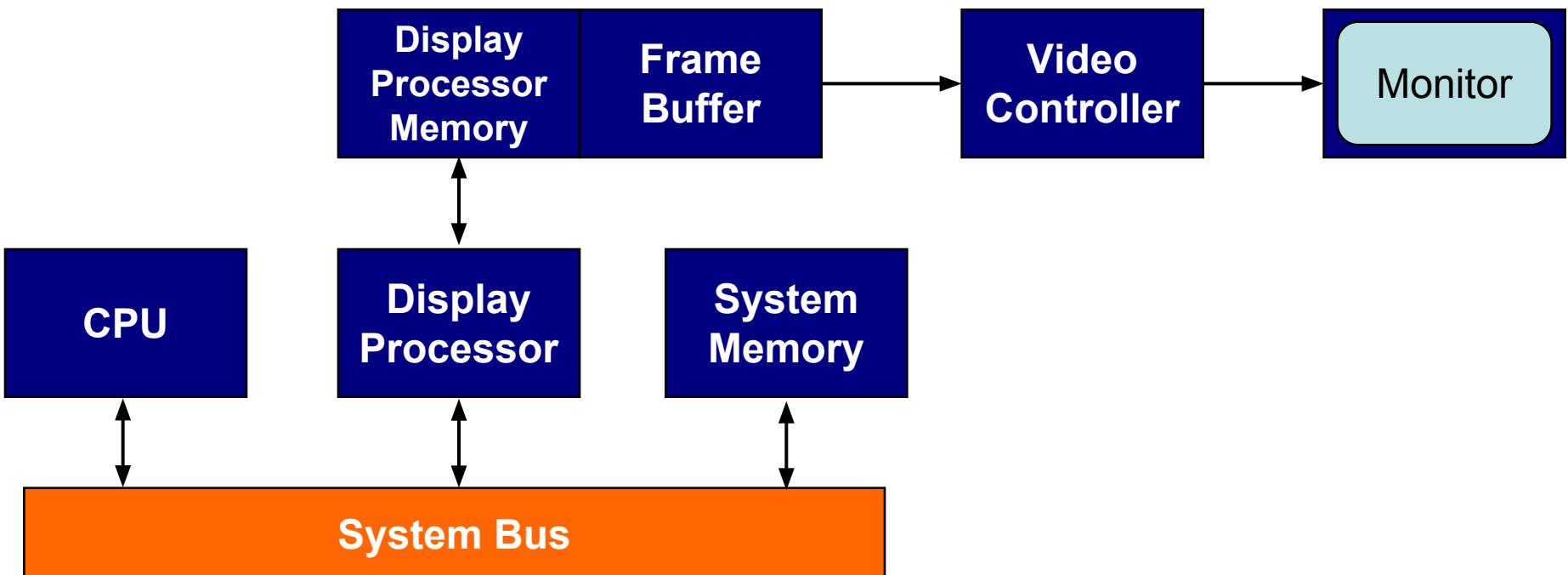
Graphics Hardware

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:

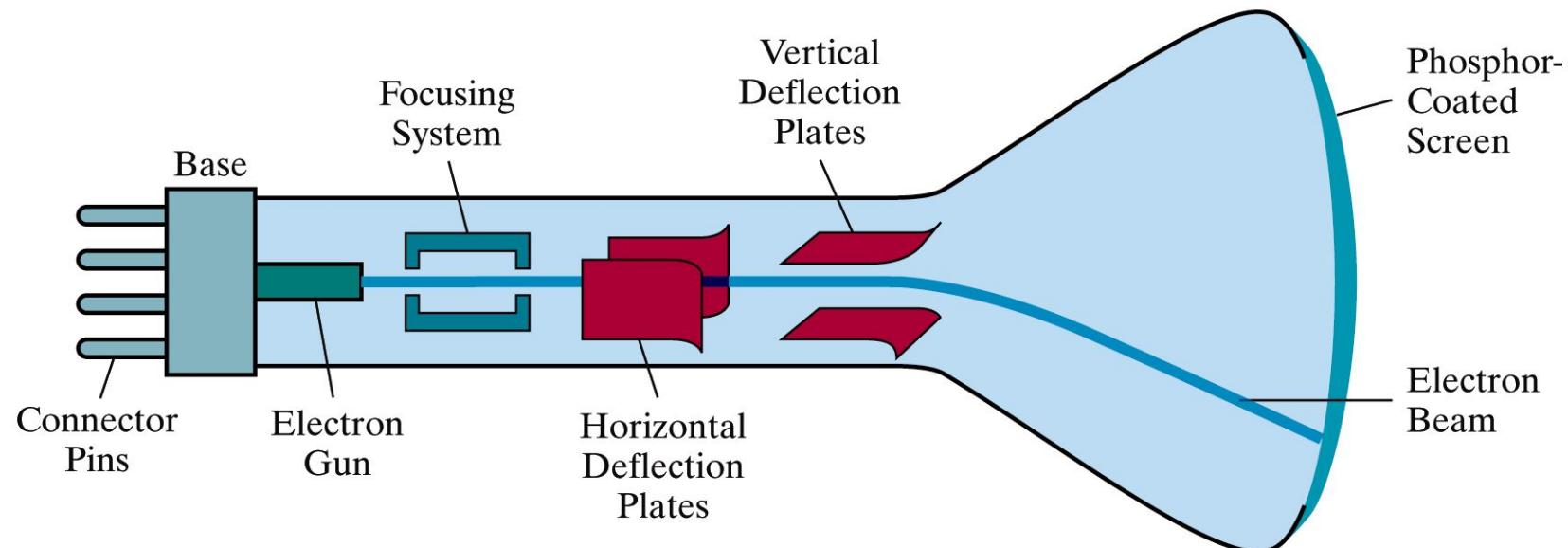
- Printers/plotters
- Cathode ray tube displays
- Plasma displays
- LCD displays
- 3 dimensional viewers
- Virtual/augmented reality headsets

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



Basic Cathode Ray Tube (CRT)

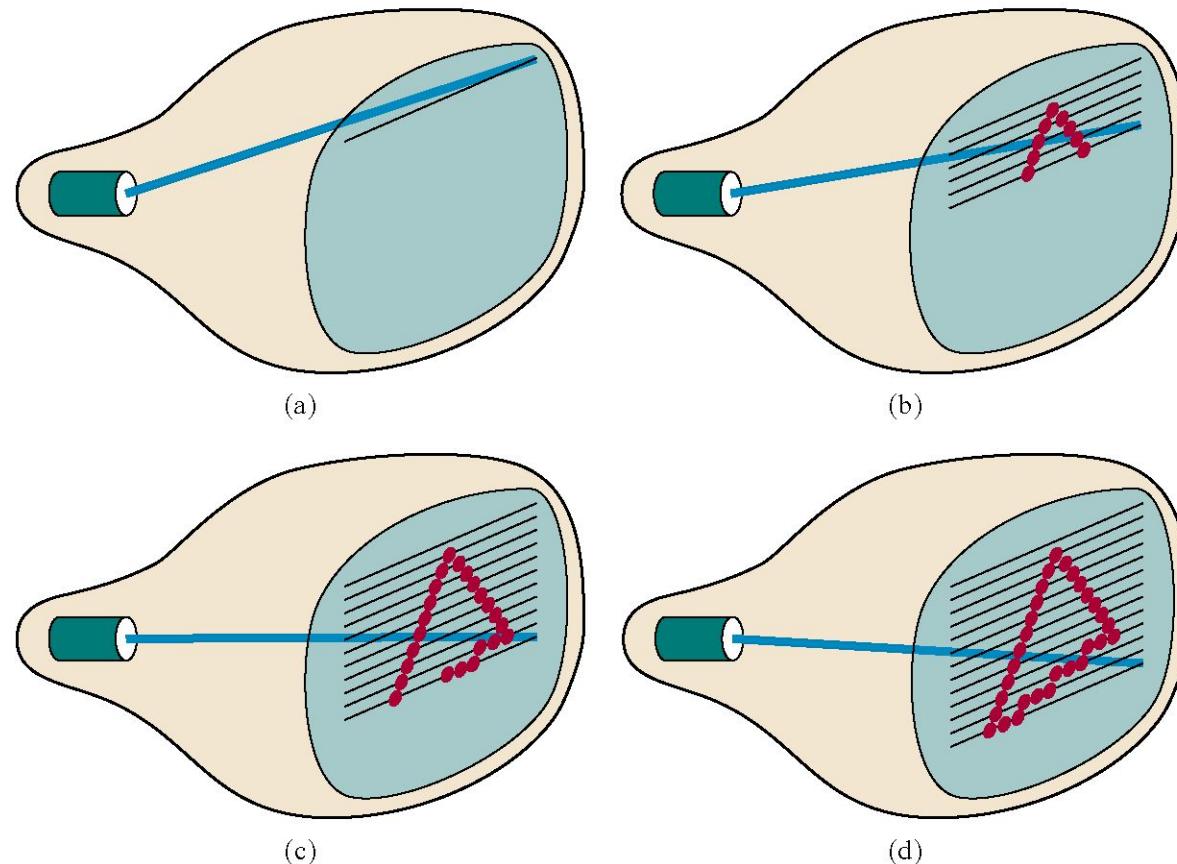
Fire an electron beam at a phosphor coated screen





Raster Scan Systems

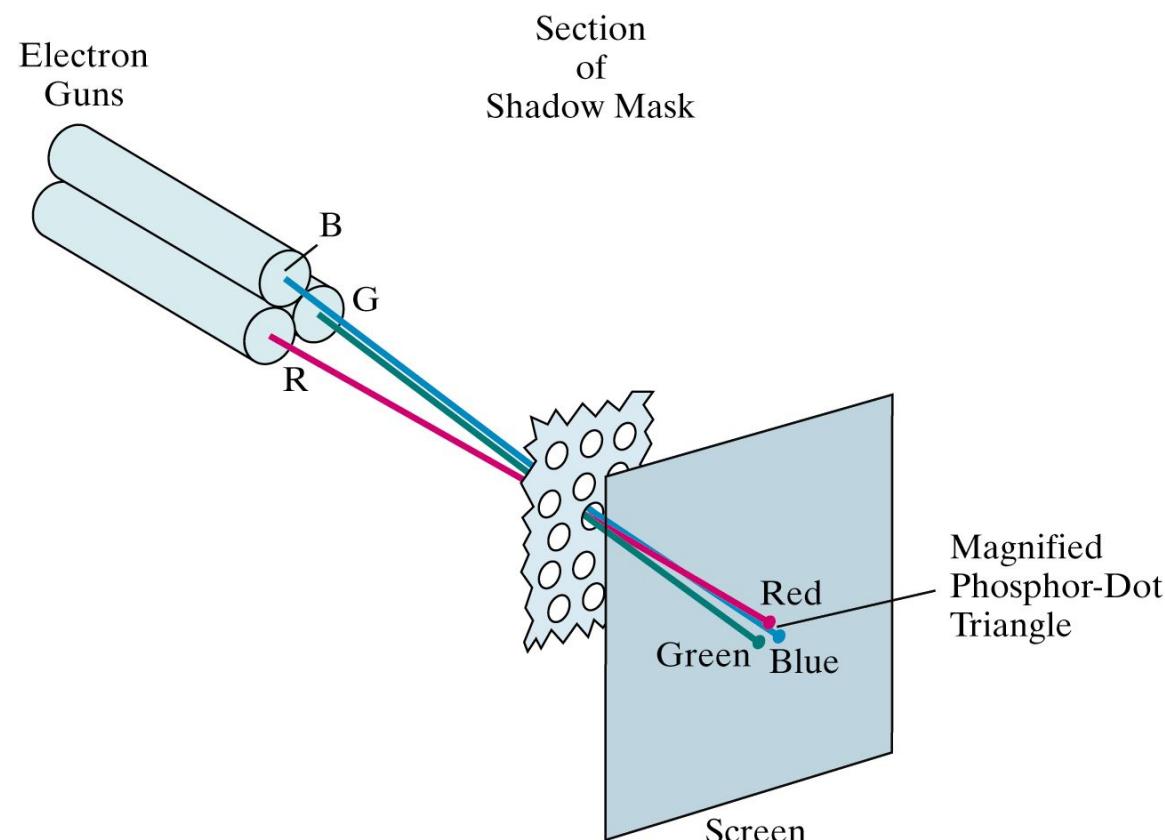
Draw one line at a time





Colour CRT

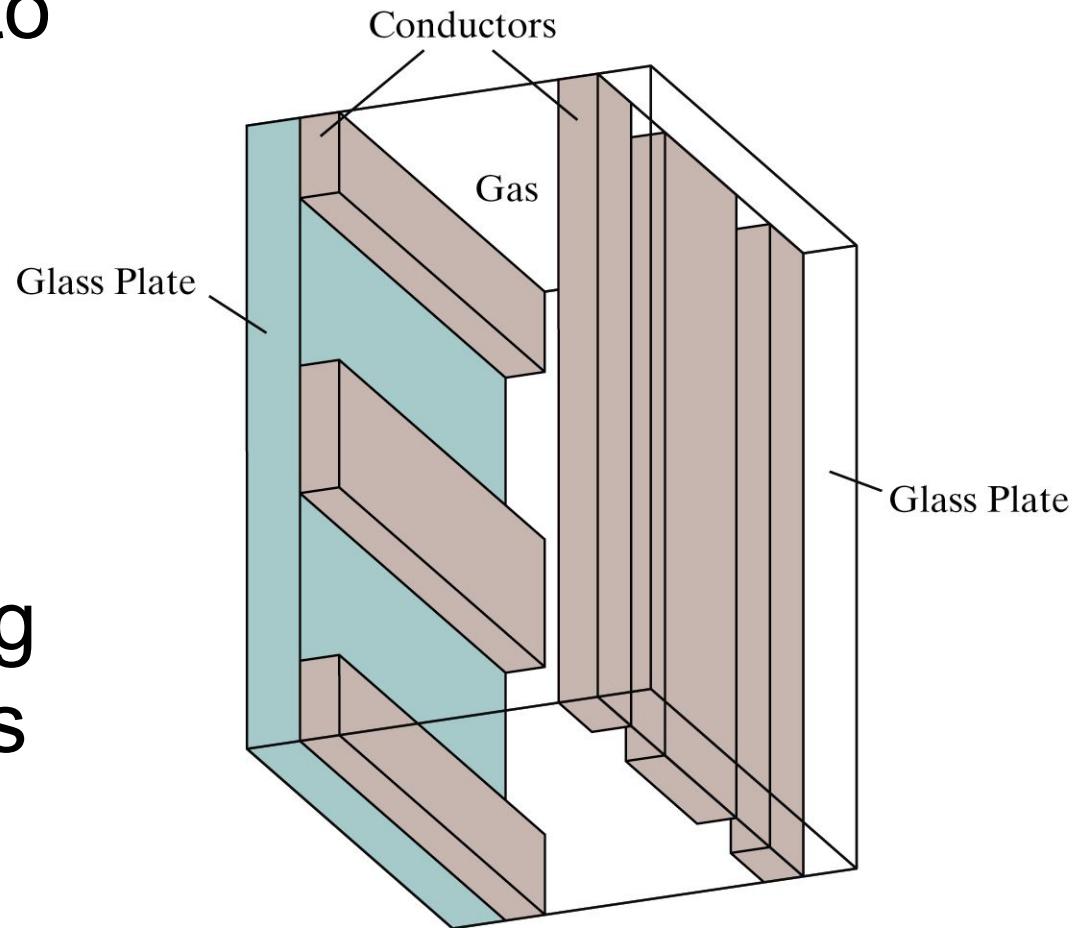
An electron gun for each colour – red, green and blue





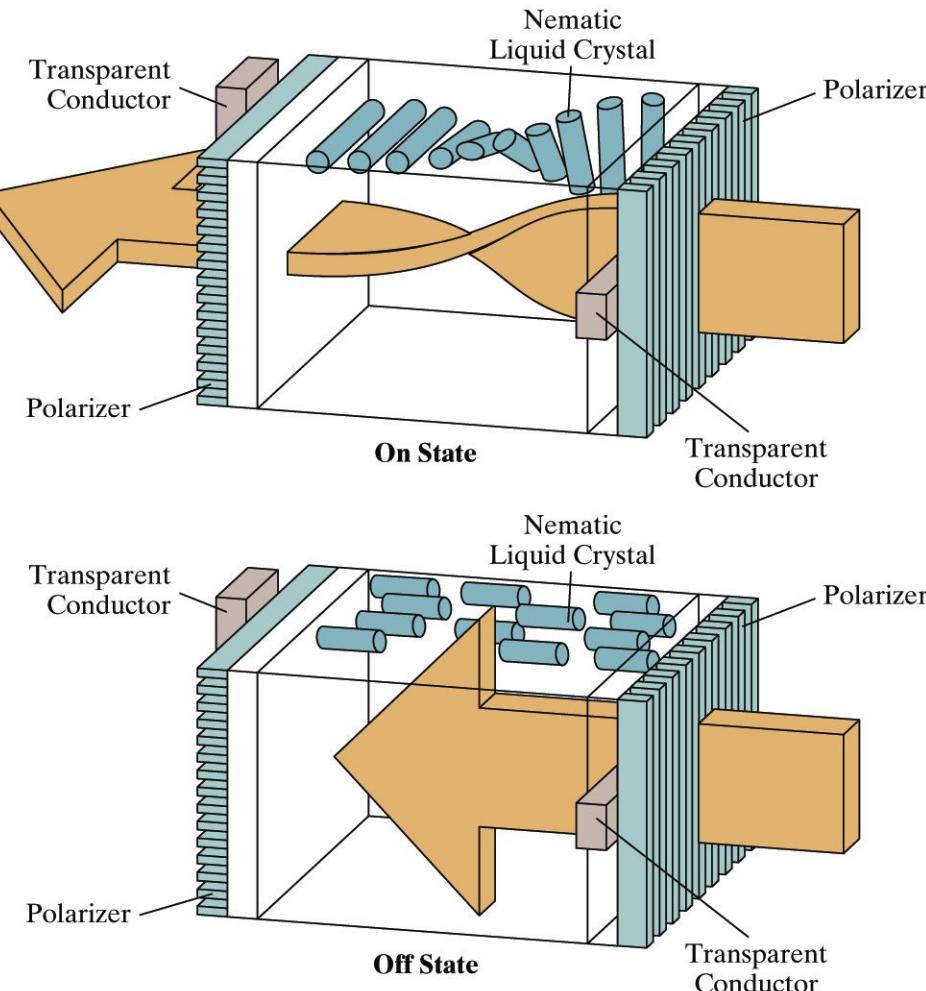
Plasma-Panel Displays

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



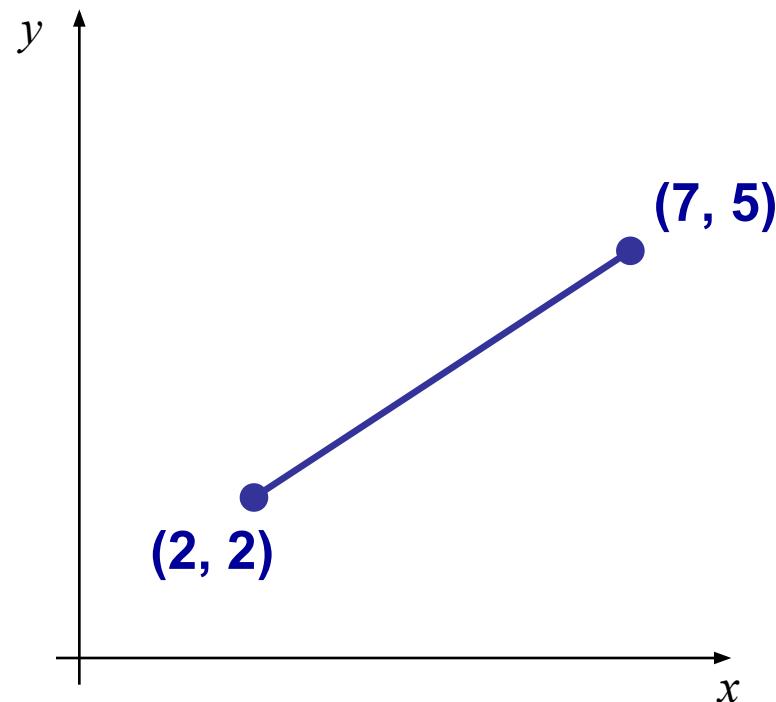
Light passing through the liquid crystal is twisted so it gets through the polarizer

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



The Problem Of Scan Conversion

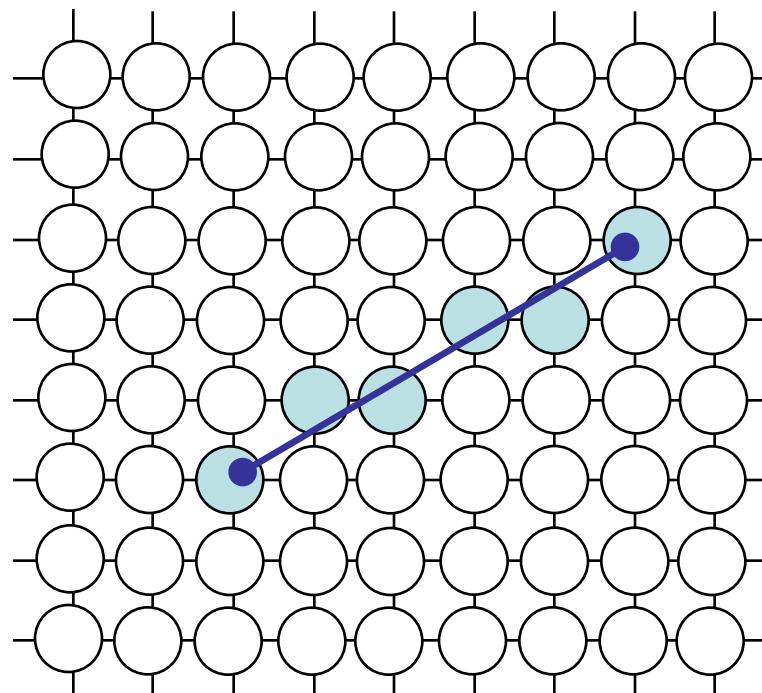
A line segment in a scene is defined by the coordinate positions of the line end-points





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?



Considerations

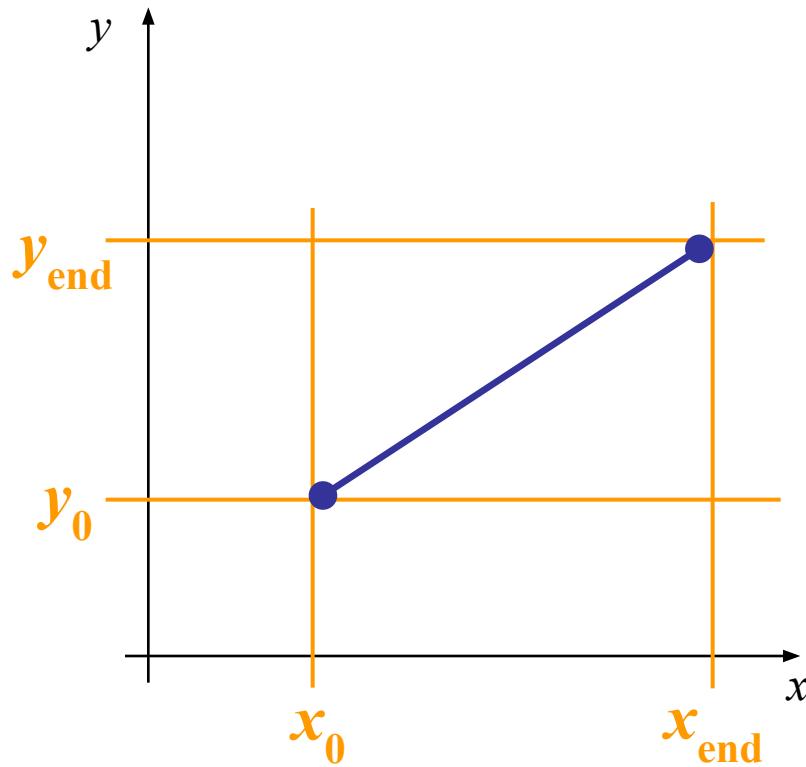
Considerations to keep in mind:

- 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



Line Equations

Let's quickly review the equations involved in drawing lines



Slope-intercept line equation:

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

where:

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

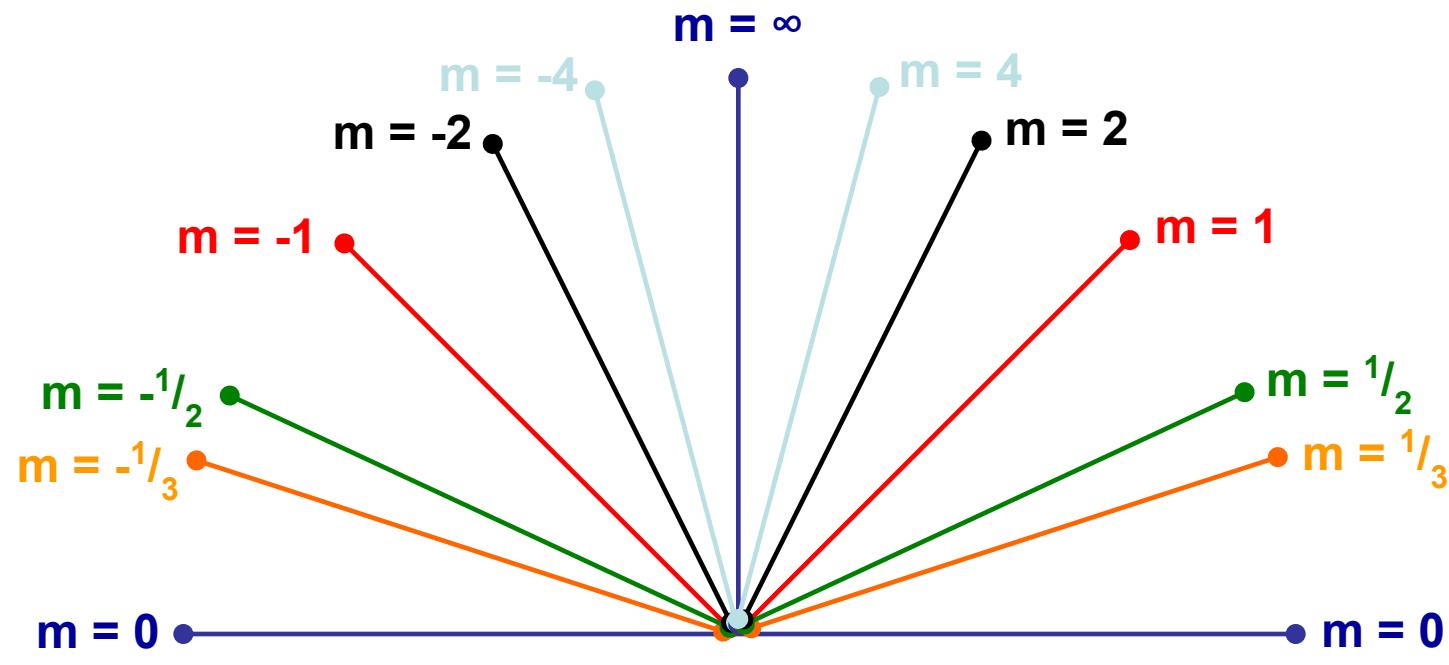
$$b = y_0 - m \cdot x_0$$



Lines & Slopes

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

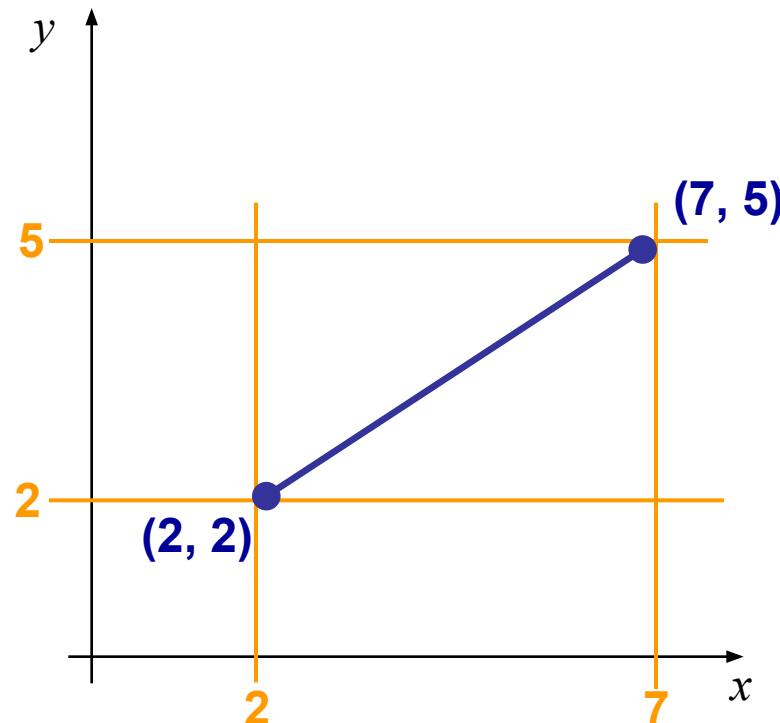
The diagram below shows some examples of lines and their slopes





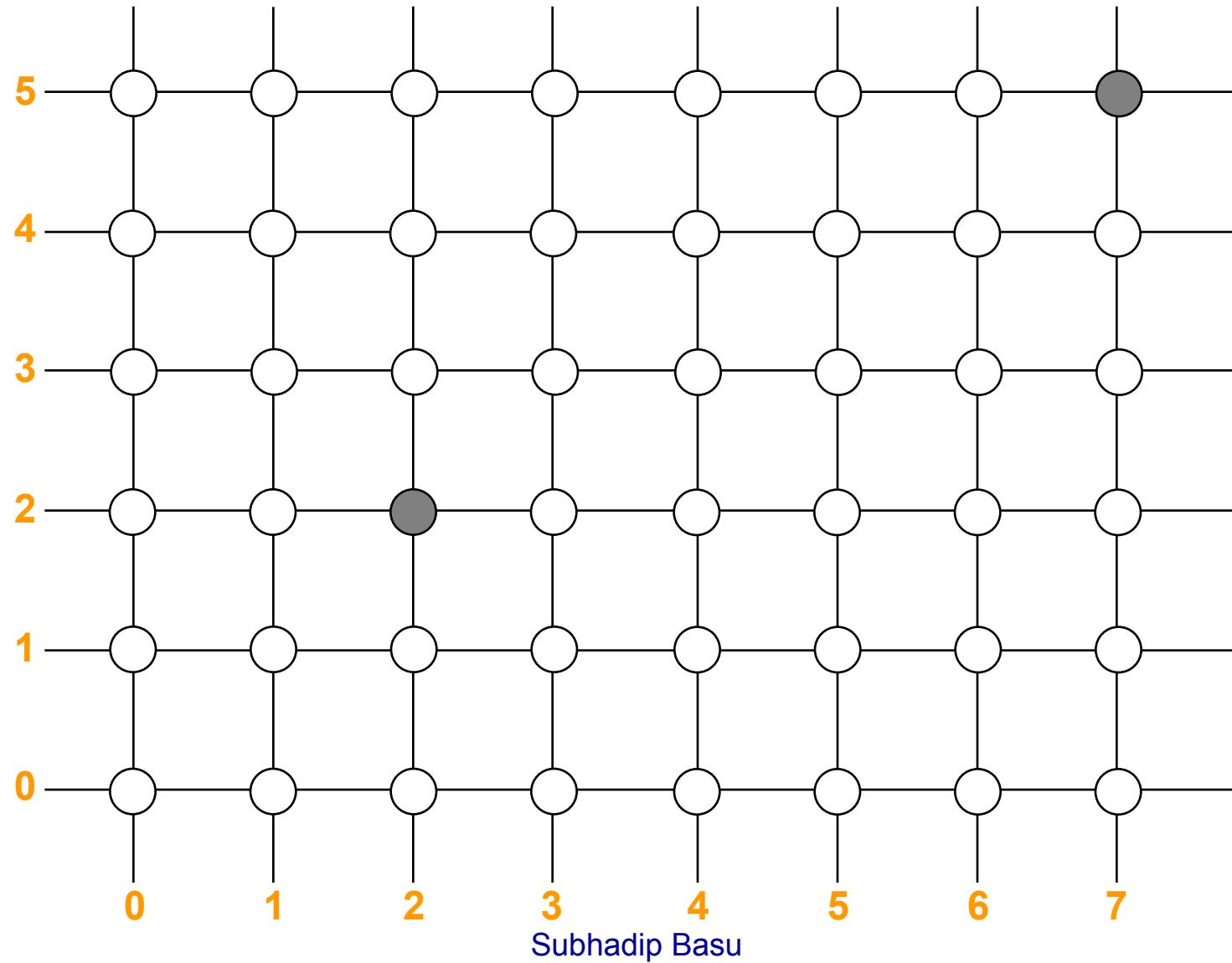
A Very Simple Solution

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



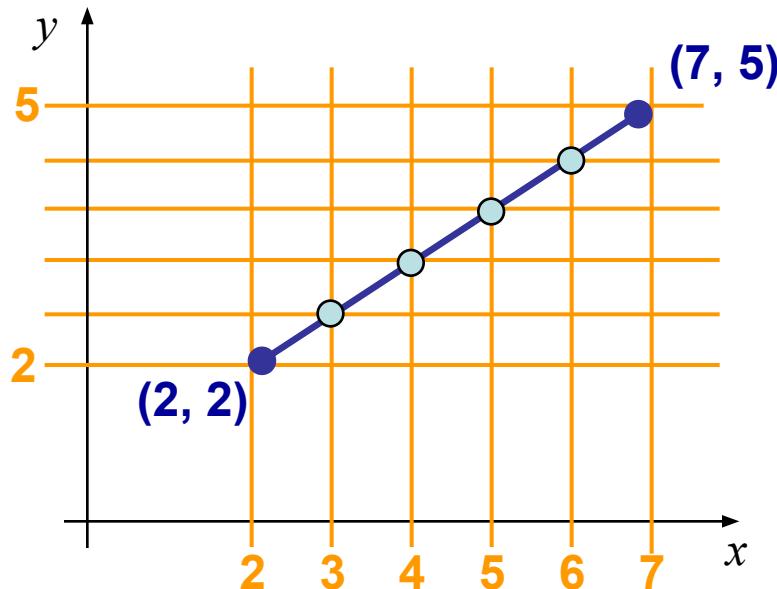


A Very Simple Solution (cont...)





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:

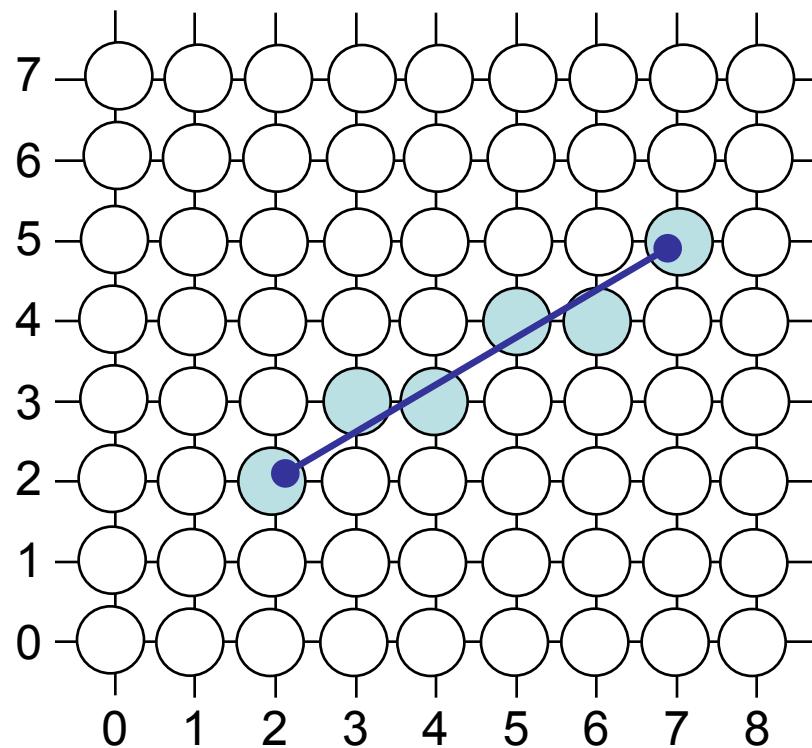
$$y(3) = \frac{3}{5} \cdot 3 + \frac{4}{5} = 2\frac{3}{5} \quad y(4) = \frac{3}{5} \cdot 4 + \frac{4}{5} = 3\frac{1}{5}$$

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



A Very Simple Solution (cont...)

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



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

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

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

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



Simple Algorithm

Start x at origin of line

$$y = m(x - x_0) + y_0$$

Evaluate y value

Set nearest pixel (x,y)

Move x one pixel toward
other endpoint

Repeat until done



Implementation

```
// SimplePixelCalc - elementary pixel calculation using
// variation of y=mx+b
void SimplePixelCalc( int x0, int xn, int y0, int yn )
{
    for( int ix=x0; ix<=xn; ix++ )
    {
        float fy = y0 + float(yn-y0)/float(xn-x0)*ix;           // round to nearest whole integer
        int iy = (int)(fy+0.5);
        SetPixel(ix, iy, 1); // write pixel value
    }
}
```

How well
does this
work?

How does this work?



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...)

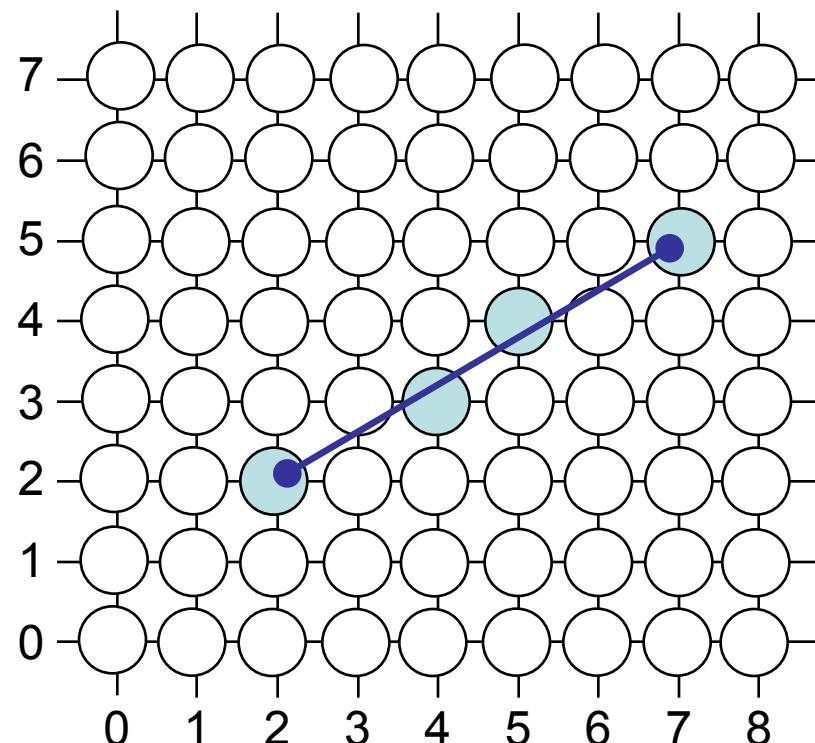
Leaving out the details this gives us:

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

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

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

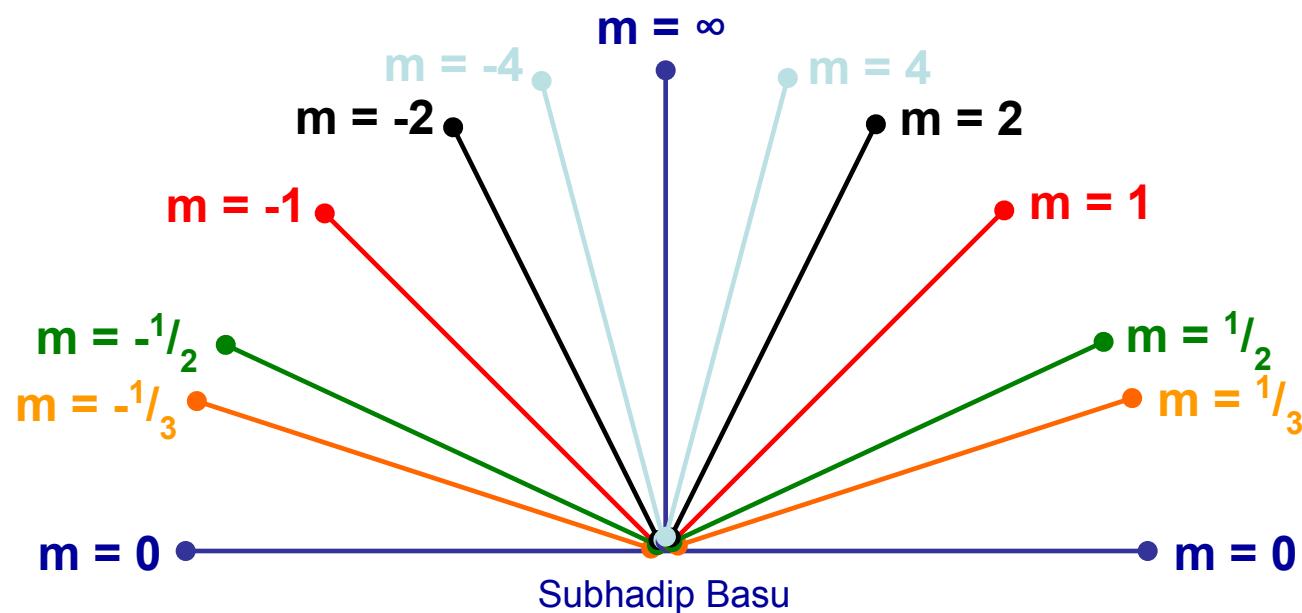




A Quick Note About Slopes (cont...)

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

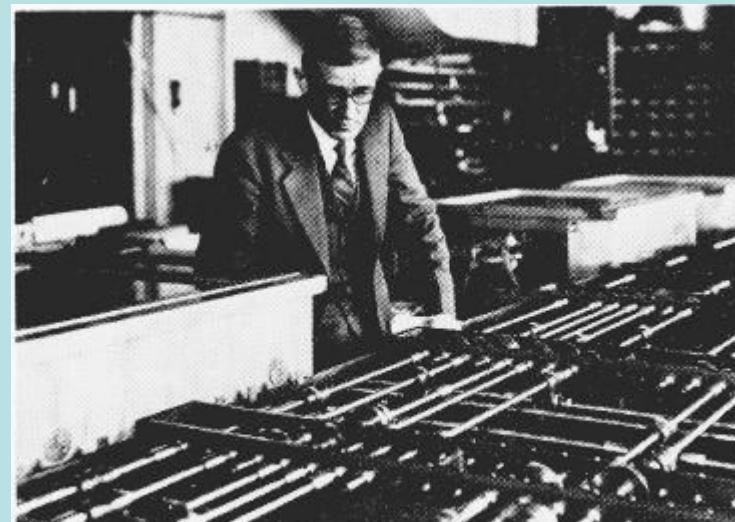




The DDA Algorithm

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](#).



Digital Differential Analyzer (DDA) Algorithm

Step through either x or y based on slope

Build the line parametrically

- If $|m| < 1$
 - $x_{k+1} = x_k + 1$
 - $y_{k+1} = y_k + m$
- If $|m| > 1$
 - $x_{k+1} = x_k + 1/m$
 - $y_{k+1} = y_k + 1$
- If $|m| = 1$
 - $x_{k+1} = x_k + 1$
 - $y_{k+1} = y_k + 1$



The DDA Algorithm (cont...)

Consider the list of points that we determined for the line in our previous example:

$$(2, 2), (3, \frac{2^3}{5}), (4, \frac{3^1}{5}), (5, \frac{3^4}{5}), (6, \frac{4^2}{5}), (7, 5)$$

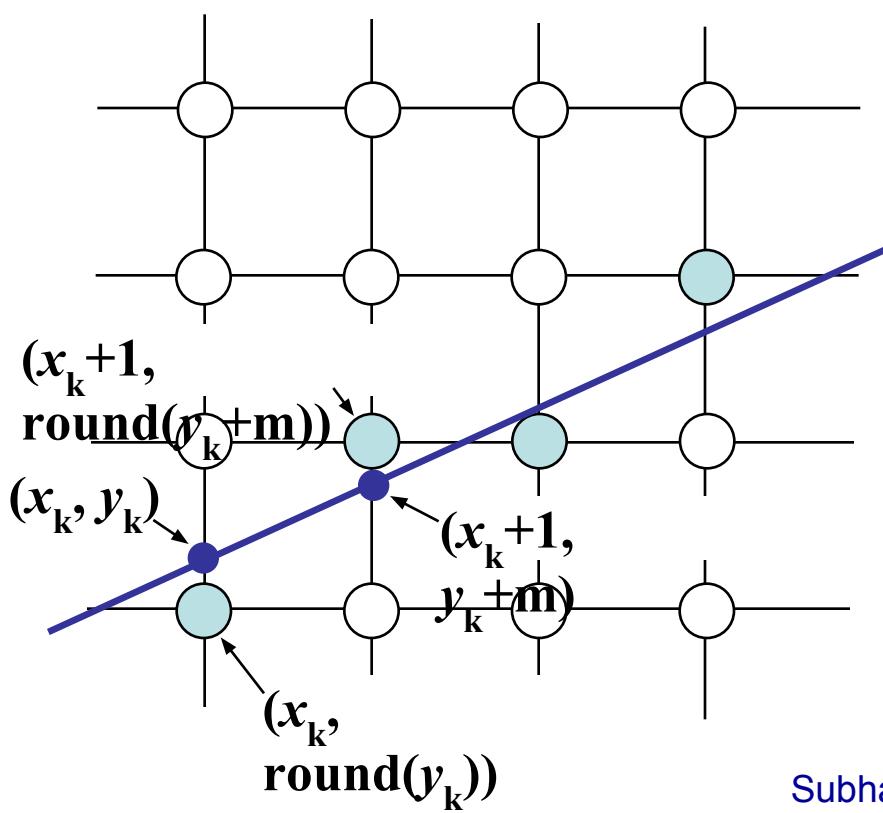
Notice that as the x coordinates go up by one, the y coordinates simply go up by the slope of the line

This is the key insight in the DDA algorithm

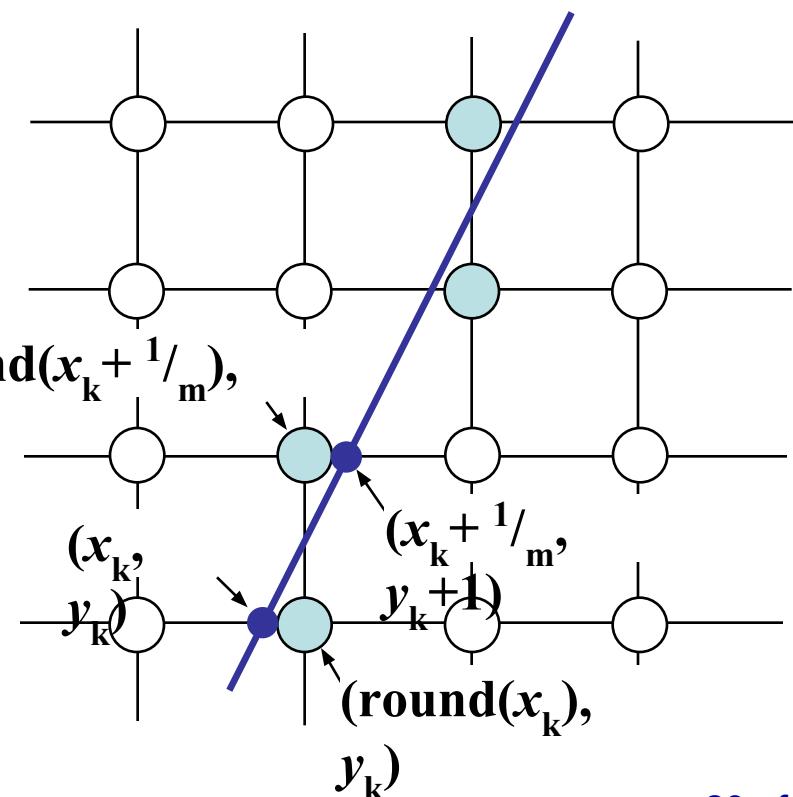


The DDA Algorithm (cont...)

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



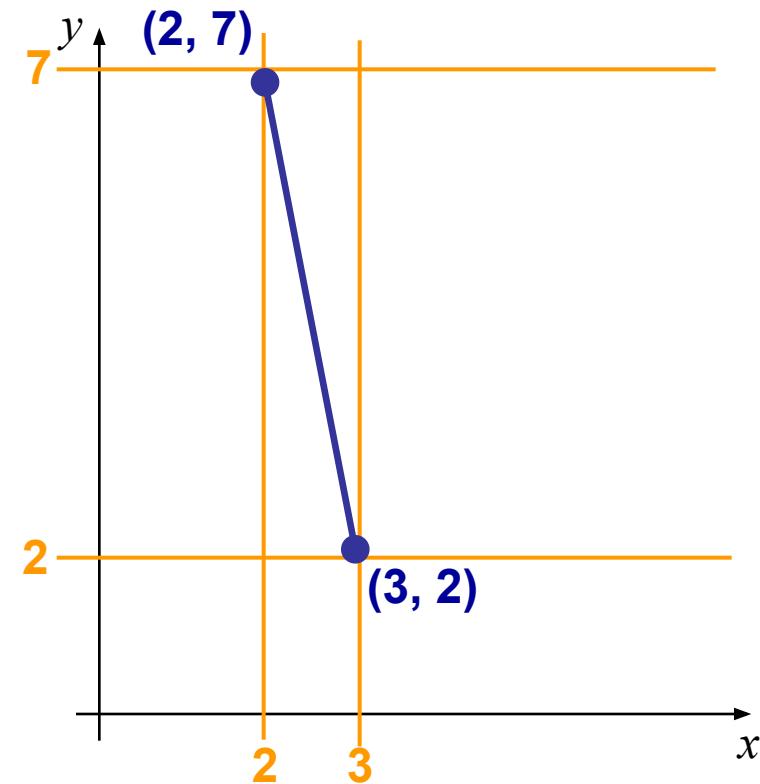
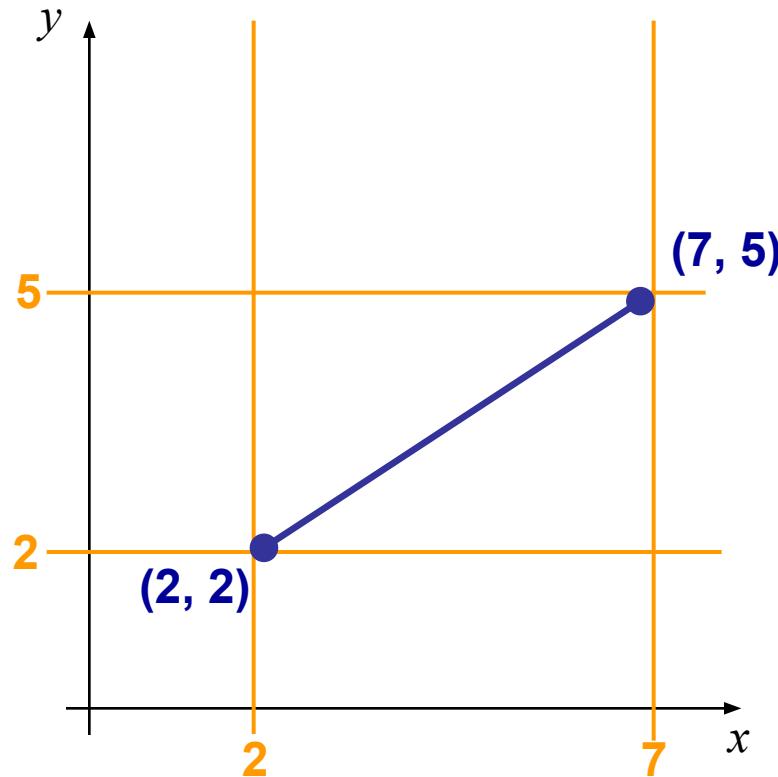
Subhadip Basu





DDA Algorithm Example

Let's try out the following examples:





DDA Implementation

```
void dda( int x0, int xn, int y0, int yn )
{
    float dx = float(xn - x0); // total span in x
    float dy = float(yn - y0); // total span in y
    float y = float(y0);
    float x = float(x0);

    float Dx, Dy; // incremental steps in x & y
    // determine if slope m GT or LT 1
    if( dx > dy )
    {
        Dx = 1;
        Dy = dy/dx;// floating division, but only done once per line
    }
    else
    {
        Dx = dx/dy;
        Dy = 1;
    }
    int ix, iy; // pixel coords
    for( int k=0; k<=((dx>dy)? dx:dy) ; k++ )
    {
        ix = int(x + 0.5); // round to nearest pixel coordinate
        iy = int(y + 0.5);
        x += Dx; // floating point calculations
        y += Dy;
    }
}
```



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

Line Drawing Algorithms

Part - 2

Dr. Subhadip Basu

*Senior Lecturer, CSE Dept., JU
subhadip@cse.jdvu.ac.in*



Contents (Part-II)

In today's lecture we'll have a look at:

- Bresenham's line drawing algorithm
- Line drawing algorithm comparisons
- Circle drawing algorithms
 - A simple technique
 - The mid-point circle algorithm
- Polygon fill algorithms
- Summary of raster drawing algorithms



The Bresenham Line Algorithm

The Bresenham algorithm is another incremental scan conversion algorithm

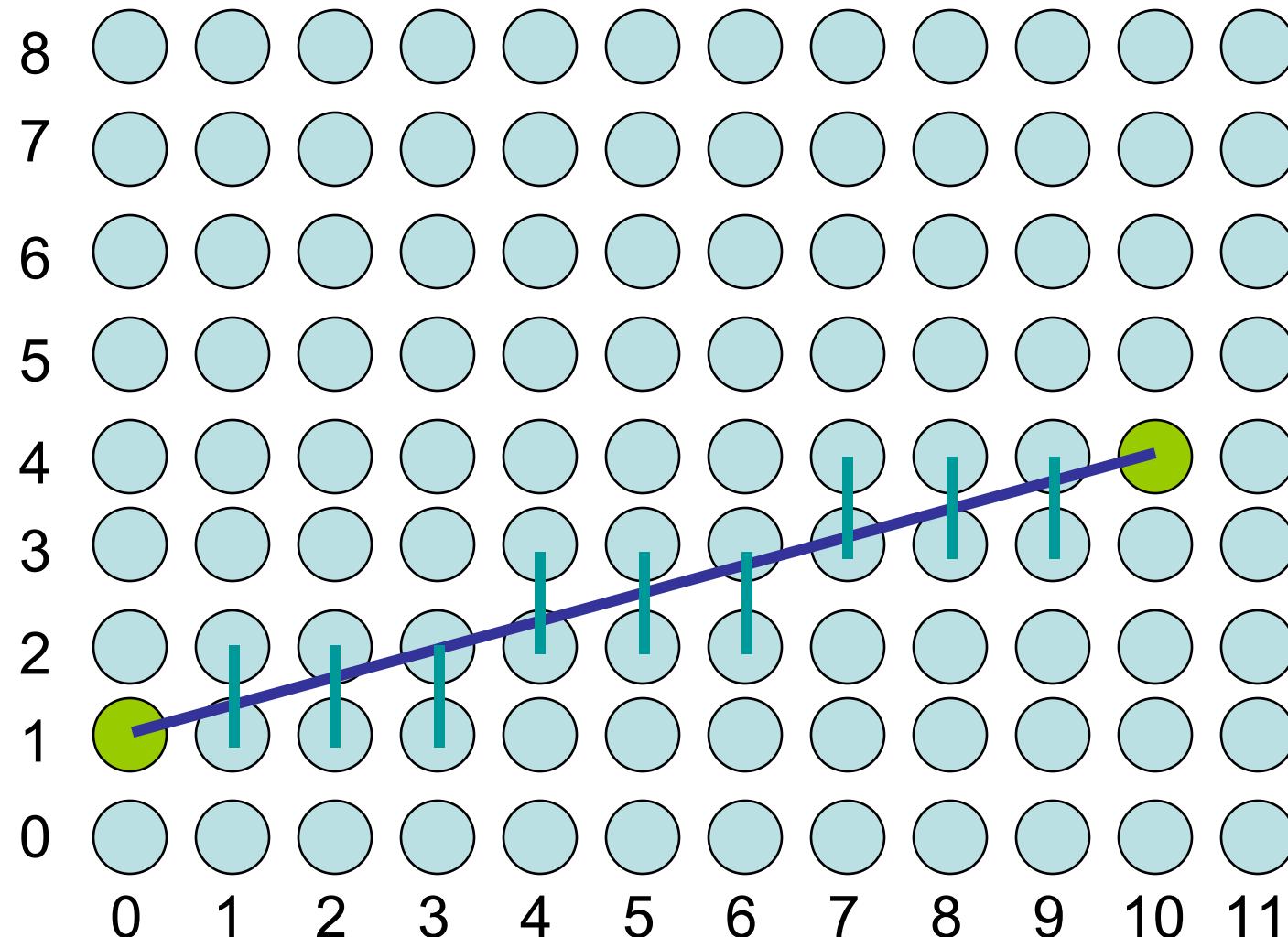
The big advantage of this algorithm is that it uses only integer calculations



Jack Bresenham worked for 27 years at IBM before entering academia. Bresenham developed his famous algorithms at IBM in the early 1960s

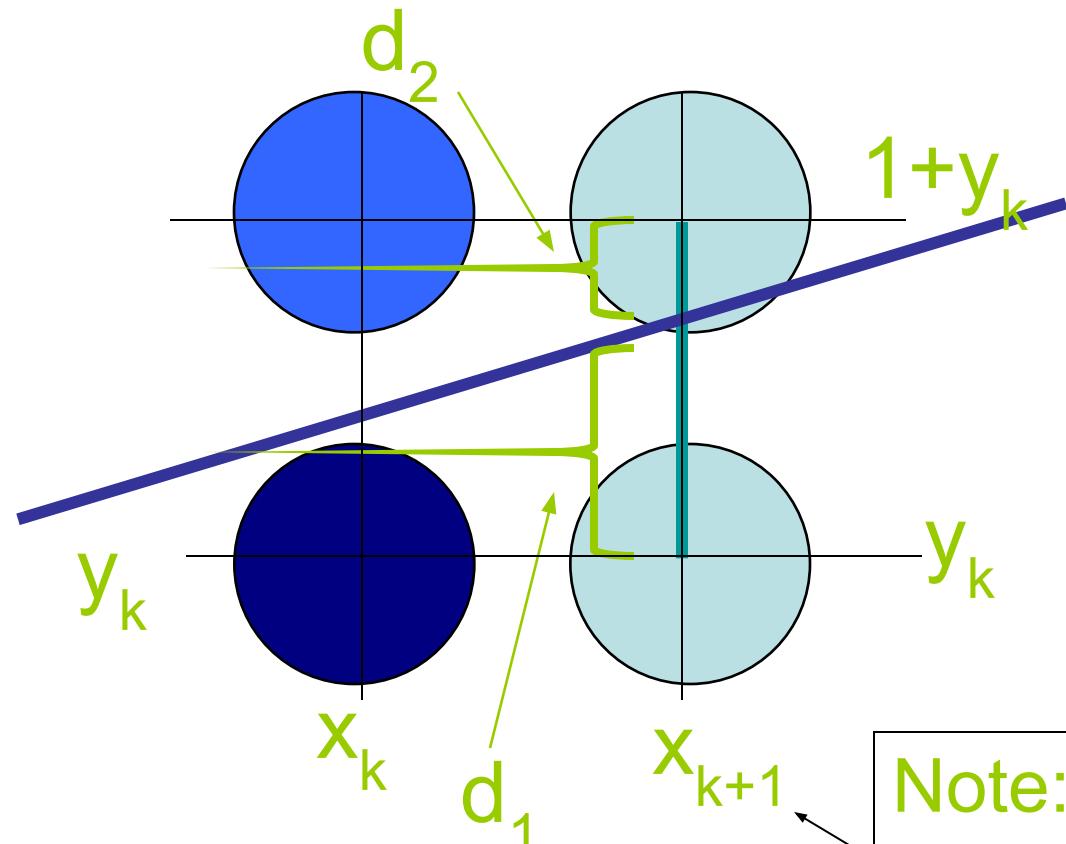


Analyzing errors at each point





Error Analysis – for $m < 1$

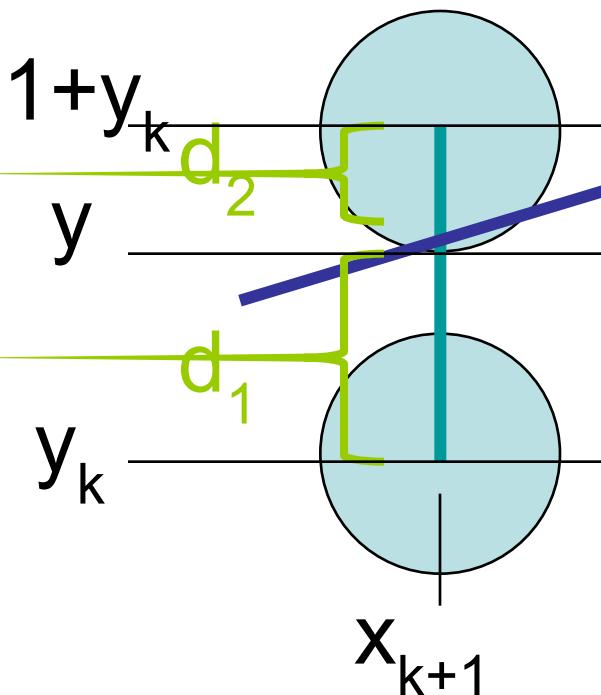


For $|m| < 1$,
there is only 1
pixel lit in each
column

Note: when $|m| < 1$,
 $x_{k+1} = 1 + x_k$



Computation of error values



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

$$d_1 = y - y_k \quad \text{Now substitute for } y$$

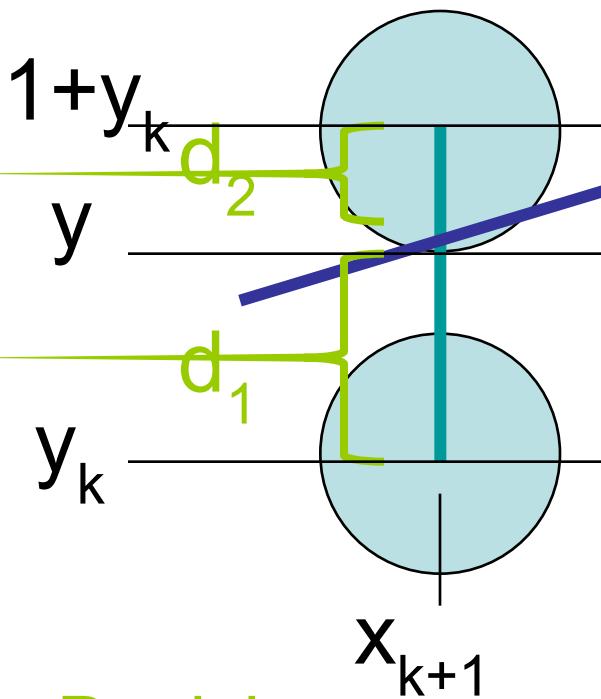
$$= m(1 + x_k) + b - y_k$$

$$\begin{aligned} d_2 &= (1 + y_k) - y \\ &= y_k + 1 - m(1 + x_k) - b \end{aligned}$$

Note: when
 $|m|<1$,
 $x_{k+1} = 1+x_k$



Deciding on the Next Point



Decision
value

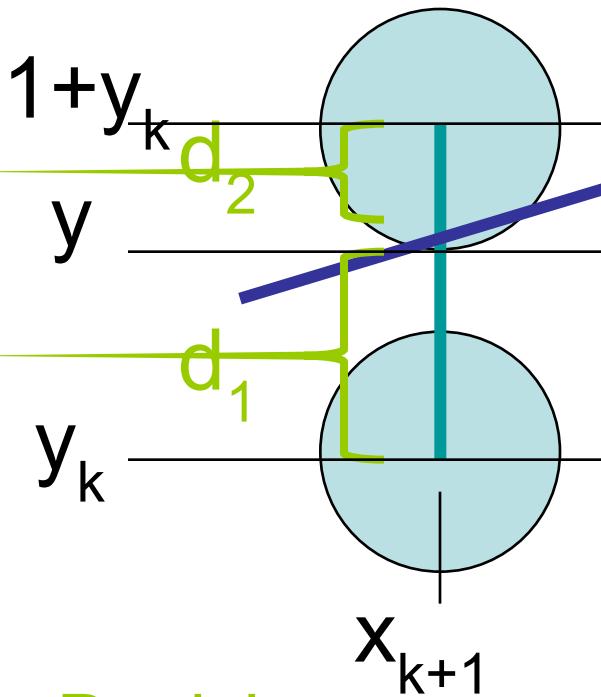
$$y = m(1 + x_k) + b$$

$$d_1 = m(1 + x_k) + b - y_k$$

$$d_2 = y_k + 1 - m(1 + x_k) - b$$

$$d_1 - d_2 = 2m(1 + x_k) - 2y_k + 2b - 1$$

Decision point



Decision
value

If $d_2 > d_1$ (ie $d_1 - d_2 < 0$),
 $y_{k+1} = y_k$

If $d_2 < d_1$ (ie $d_1 - d_2 > 0$),
 $y_{k+1} = 1 + y_k$

$$d_1 - d_2 = 2m(1 + x_k) - 2y_k + 2b - 1$$



Status Check

$$d_1 - d_2 = 2m(1 + x_k) - 2y_k + 2b - 1$$

What have we accomplished?

- One value to test (+/-)

But ...

- Still have floating-point calculation
 - “m” and “b” are non-integer



Removing Non-Integral Values

$$d_1 - d_2 = 2m(x_k + 1) - 2y_k + 2b - 1 \quad \text{What is } m?$$

$$d_1 - d_2 = 2 \frac{\Delta y}{\Delta x} (x_k + 1) - 2y_k + 2b - 1 \quad \text{Now multiply throughout by } \Delta x$$

$$\Delta x(d_1 - d_2) = 2\Delta y \cdot x_k + 2\Delta y - 2\Delta x \cdot y_k + \Delta x(2b - 1) \quad \text{Which terms are constants?}$$

Decision value

$$p_k = 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + c \quad c \text{ is constant, but non-integer}$$



Recurrence Relation for p_k

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

$$p_{k+1} = 2\Delta y \cdot x_{k+1} - 2\Delta x \cdot y_{k+1} + c$$

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

$$p_{k+1} = p_k + 2\Delta y - 2\Delta x \left(\frac{y_{k+1} - y_k}{0 \text{ or } +1} \right)$$



Initializing the Algorithm

Initial value of decision parameter

$$p_0 = 2\Delta y \cdot x_0 - 2\Delta x \cdot y_0 + c$$

$$p_0 = 2\Delta y \cdot x_0 - 2\Delta x \cdot y_0 + 2\Delta y + \Delta x(2b - 1)$$

$$b = y_0 - (\Delta y / \Delta x)x_0$$

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



Bresenham Algorithm Summary

Calculate once: $\Delta x, \Delta y, 2\Delta y, 2\Delta y - 2\Delta x, p_0$

At each point:

If $p_k < 0$:

Plot($x_k + 1, y_k$)

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

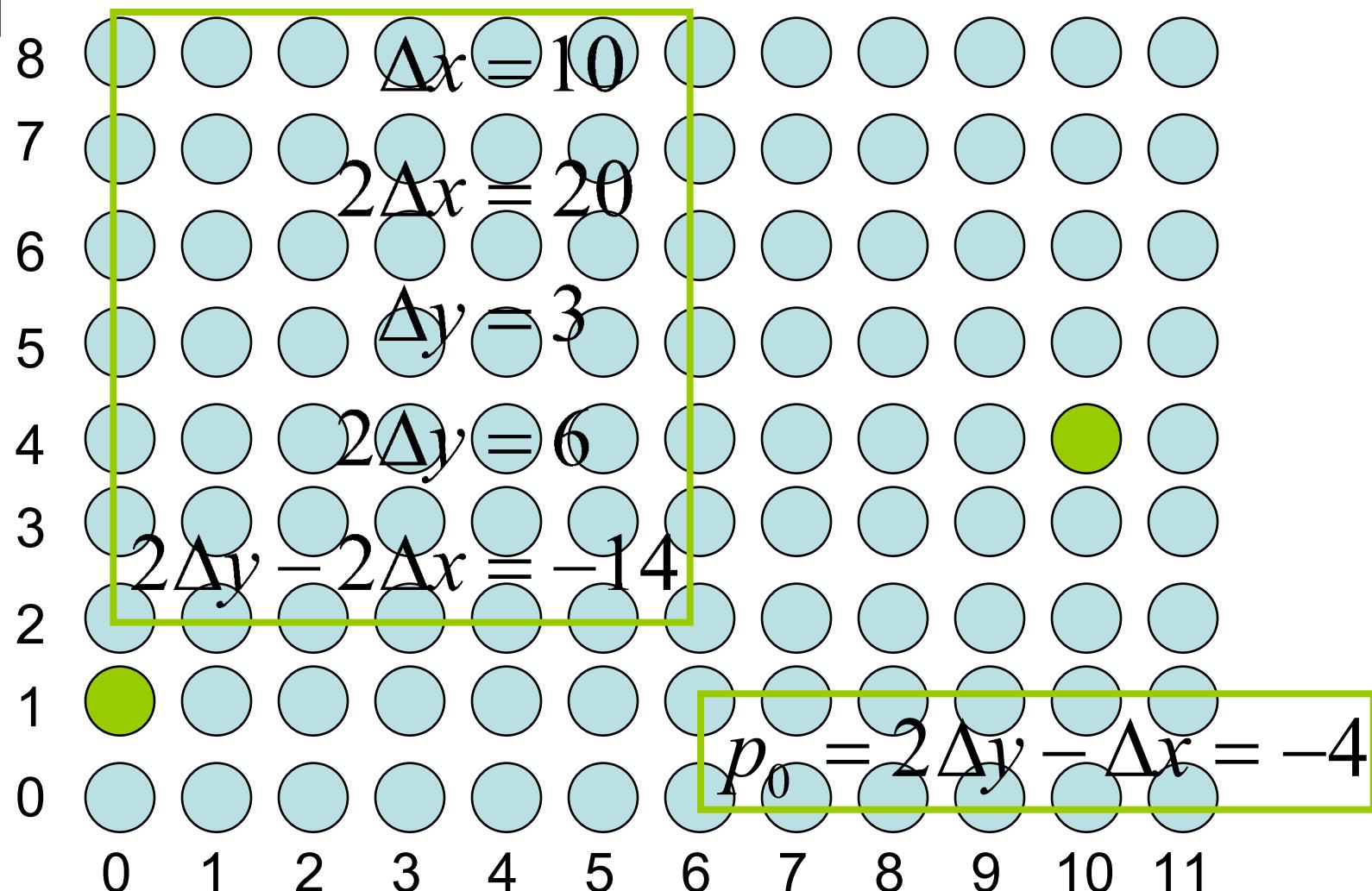
If $p_k \geq 0$:

Plot($x_k + 1, y_k + 1$)

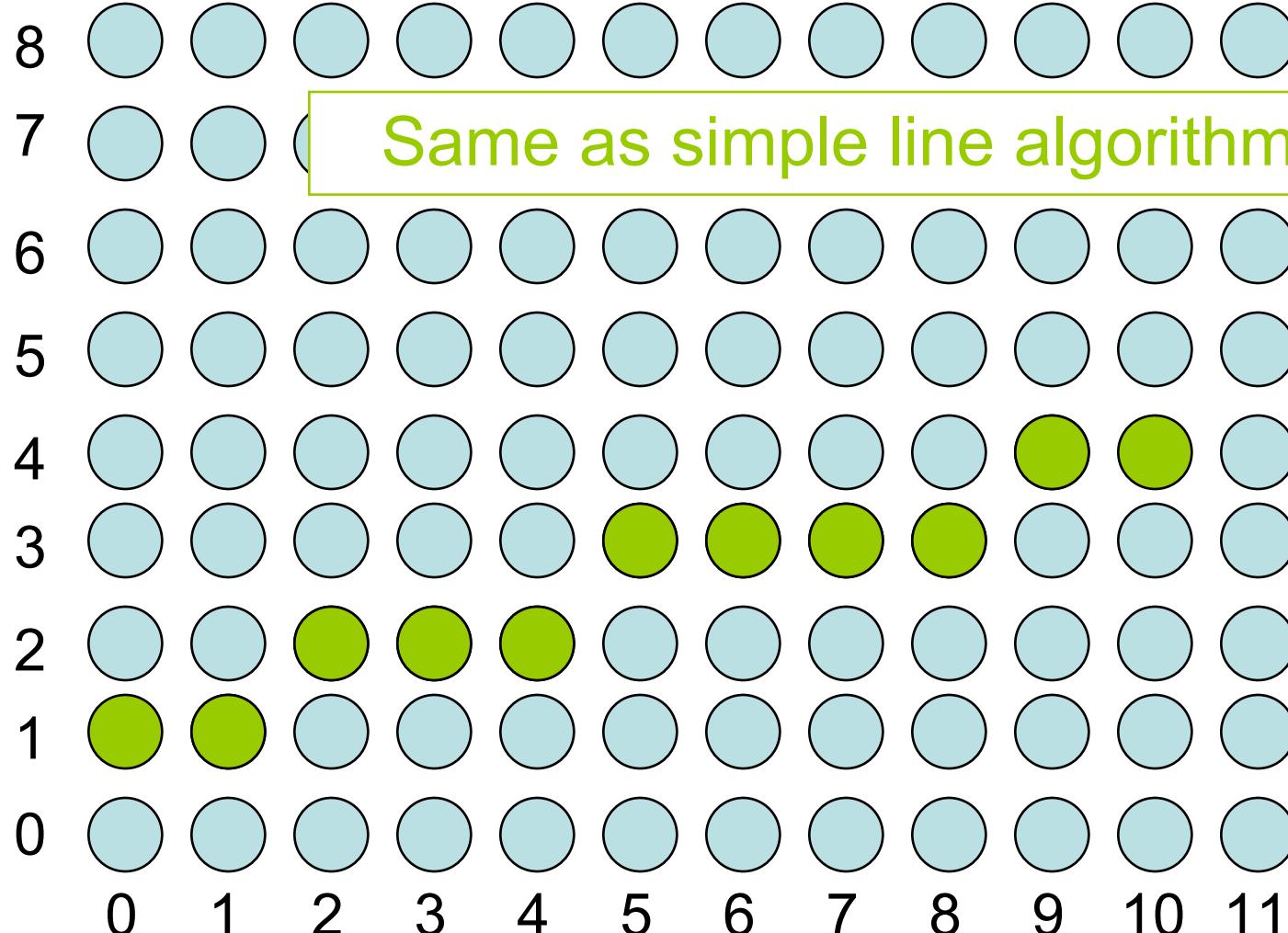
$$p_{k+1} = p_k + 2\Delta y - 2\Delta x$$



Example 1 Parameters



Example 1 Path





The Bresenham Line Algorithm

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:

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



The Bresenham Line Algorithm (cont...)

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

NOTE! The algorithm and derivation above assumes slopes are less than 1. for other slopes we need to adjust the algorithm accordingly



Bresenham Modifications

So far, $0 < m < 1$, $\Delta x > 0$

How do we handle other cases?

- Horizontal
- Vertical
- Other octants
 - $\text{abs}(\text{Slope}) \in (1, \infty]$
 - Transpose x with y, Δx with Δy , etc.
 - Plot(y,x)
 - Slope < 0
 - Revise: $y_{k+1} - y_k = 0$ or -1



Bresenham Line Algorithm Summary

The Bresenham 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



Drawing Circles and Arcs

Similar to line drawing

- In that you have to determine what pixels to activate

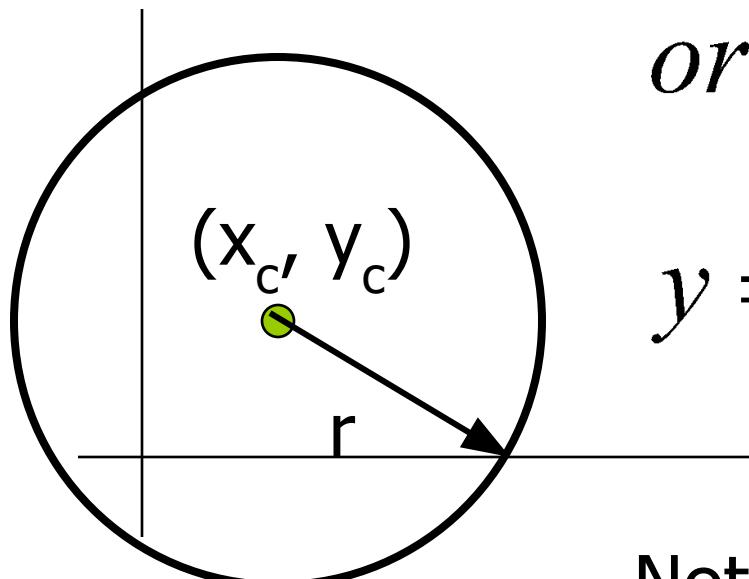
But non-linear

- Simple equation implementation
- Optimized



Cartesian form of Circle Equations

$$(x - x_c)^2 + (y - y_c)^2 = r^2$$



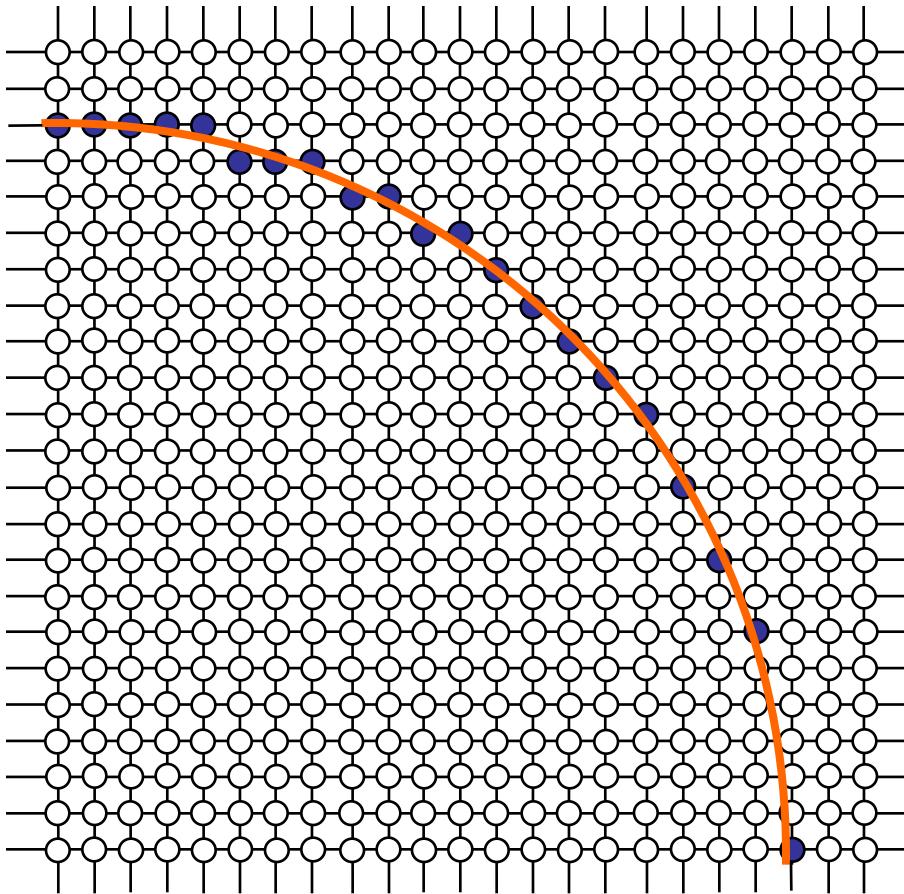
or

$$y = y_c \pm \sqrt{r^2 - (x - x_c)^2}$$

Not a very good method. Why?



A Simple Circle Drawing Algorithm (cont...)



$$y_0 = \sqrt{20^2 - 0^2} \approx 20$$

$$y_1 = \sqrt{20^2 - 1^2} \approx 20$$

$$y_2 = \sqrt{20^2 - 2^2} \approx 20$$

⋮

$$y_{19} = \sqrt{20^2 - 19^2} \approx 6$$

$$y_{20} = \sqrt{20^2 - 20^2} \approx 0$$



A Simple Circle Drawing Algorithm (cont...)

However, unsurprisingly this is not a brilliant solution!

Firstly, the resulting circle has large gaps where the slope approaches the vertical

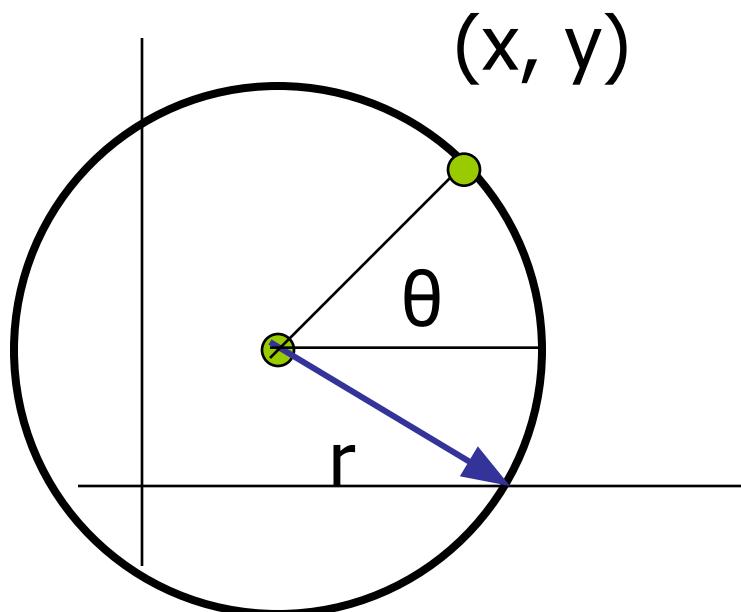
Secondly, the calculations are not very efficient

- The square (multiply) operations
- The square root operation – try really hard to avoid these!

We need a more efficient, more accurate solution



Polar Coordinate Form



Simple method: plot directly
from parametric equations

$$x = x_c + r \cos \theta$$

$$y = y_c + r \sin \theta$$

$$\Delta s = r\theta$$

$$\Delta s \approx 1$$

$$\Delta \theta \approx \frac{1}{r}$$

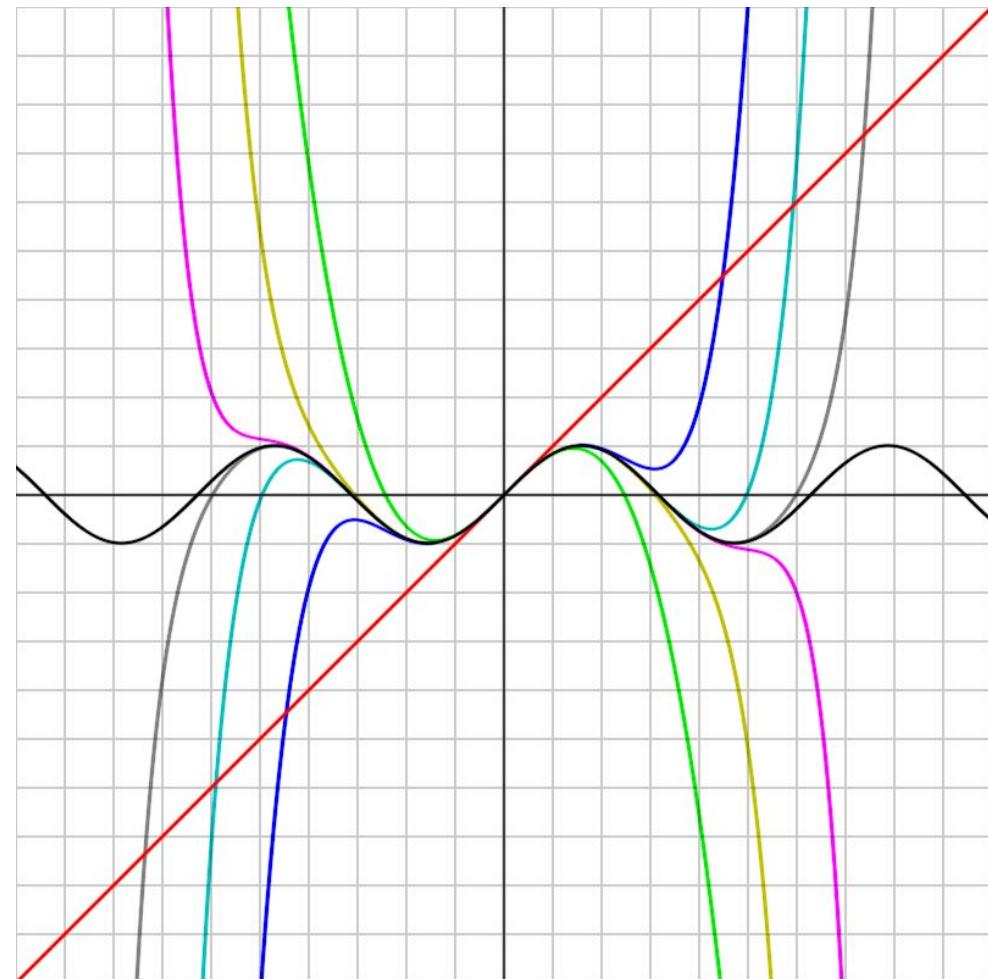


Trig functions are expensive

$$\sin \theta = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} \theta^{2n+1}$$

$$\cos \theta = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} \theta^{2n}$$

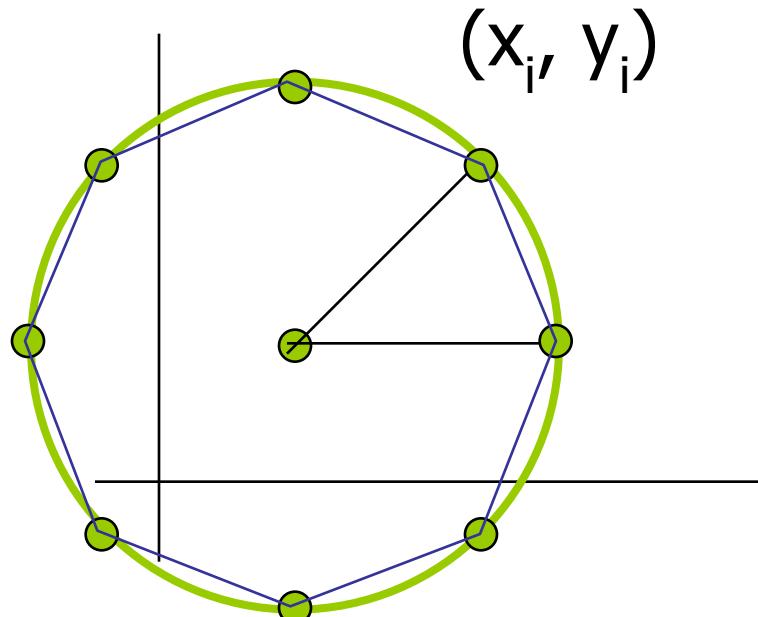
Plots of first 1, 3, 5, 7, 9, 11, and 13 terms in the Taylor's series for sin





Polygon Approximation

Calculate polygon vertices
from polar equation;
connect with line algorithm





Mid-Point Circle Algorithm

Similarly to the case with lines, there is an incremental algorithm for drawing circles – the *mid-point circle algorithm*

In the mid-point circle algorithm we use eight-way symmetry so only ever calculate the points for the top right eighth of a circle, and then use symmetry to get the rest of the points



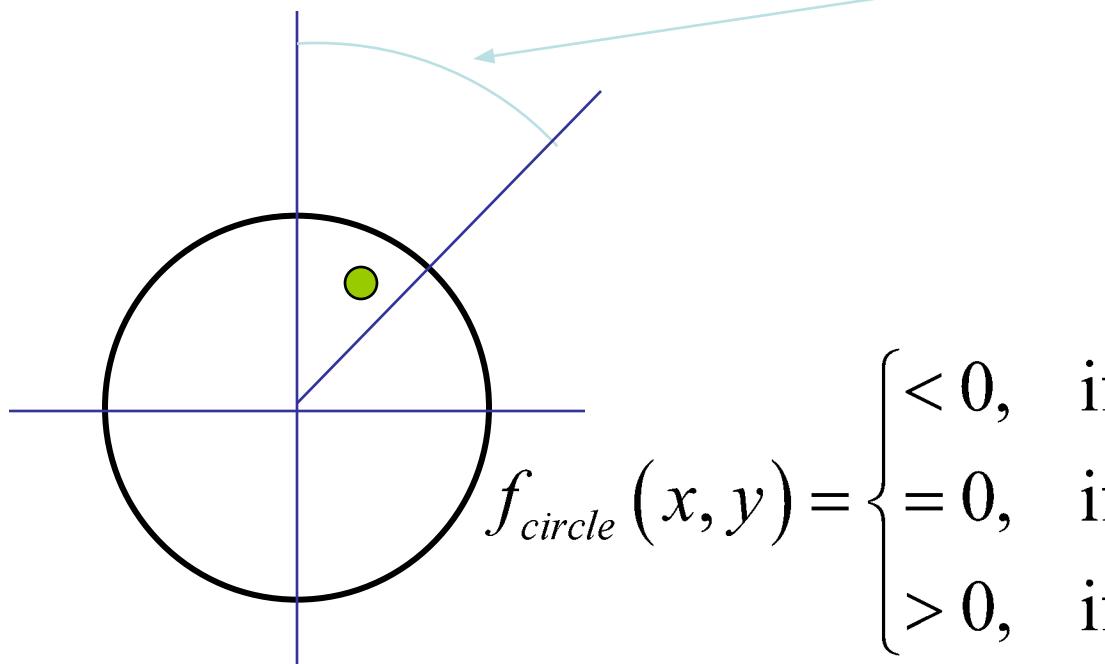
The mid-point circle algorithm was developed by Jack Bresenham, who we heard about earlier. Bresenham's patent for the algorithm can be viewed [here](#).



Bresenham's Midpoint Circle Algorithm

$$f_{circle}(x, y) = x^2 + y^2 - r^2$$

Consider this region



$$f_{circle}(x, y) = \begin{cases} < 0, & \text{if } (x, y) \text{ inside circle} \\ = 0, & \text{if } (x, y) \text{ on circle} \\ > 0, & \text{if } (x, y) \text{ outside circle} \end{cases}$$



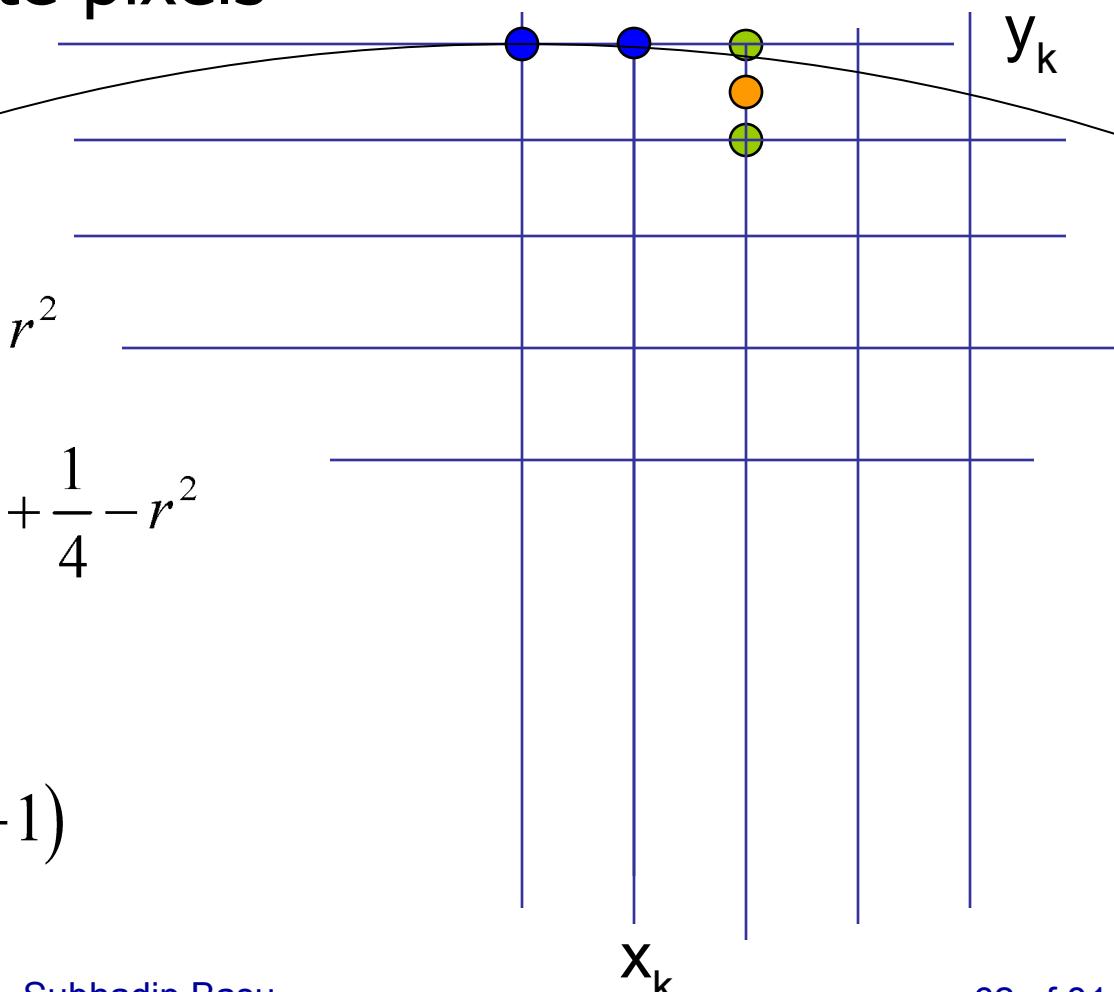
The Circle Decision Parameter

Calculate f_{circle} for point midway between candidate pixels

$$\begin{aligned} p_k &= f_{circle}\left(x_k + 1, y_k - \frac{1}{2}\right) \\ &= \left(x_k + 1\right)^2 + \left(y_k - \frac{1}{2}\right)^2 - r^2 \\ &= x_k^2 + 2x_k + 1 + y_k^2 - y_k + \frac{1}{4} - r^2 \end{aligned}$$

If $p_k < 0$: Plot $(x_k + 1, y_k)$

If $p_k \geq 0$: Plot $(x_k + 1, y_k - 1)$





Calculating p_{k+1}

$$\begin{aligned} p_{k+1} &= f_{circle} \left(x_{k+1} + 1, y_{k+1} - \frac{1}{2} \right) \\ &= (x_k + 1 + 1)^2 + \left(y_{k+1} - \frac{1}{2} \right)^2 - r^2 \\ &= x_k^2 + 4x_k + 4 + y_{k+1}^2 - y_{k+1} + \frac{1}{4} - r^2 \end{aligned}$$



Recurrence Relation for p_k

$$p_k = x_k^2 + 2x_k + 1 + y_k^2 - y_k + \frac{1}{4} - r^2$$

$$p_{k+1} = x_k^2 + 4x_k + 4 + y_{k+1}^2 - y_{k+1} + \frac{1}{4} - r^2$$

Compute $p_{k+1} - p_k$

$$\begin{aligned} p_{k+1} &= p_k + \underline{(2x_k + 2)} + 1 + \left(y_{k+1}^2 - y_k^2 \right) - \underline{\left(y_{k+1} - y_k \right)} \\ &= p_k + 2x_{k+1} + 1 + \underline{\left(y_{k+1}^2 - y_k^2 \right)} - \underline{\left(y_{k+1} - y_k \right)} \end{aligned}$$



One Last Term ...

$$\left({y_{k+1}}^2 - {y_k}^2 \right) - (y_{k+1} - y_k) = ?$$

If $y_{k+1} = y_k$, then:

$$\left({y_k}^2 - {y_k}^2 \right) - (y_k - y_k) = 0$$

If $y_{k+1} = y_k - 1$, then:

$$\left((y_k - 1)^2 - {y_k}^2 \right) - ((y_k - 1) - y_k) =$$

$$\left({y_k}^2 - 2y_k + 1 - {y_k}^2 \right) - (-1) =$$

$$-2y_k + 2 = -2(y_k - 1)$$



Initial Values

$$(x_0, y_0) = (0, r)$$

$$p_0 = f_{circle} \left(0 + 1, r - \frac{1}{2} \right)$$

$$= 1 + \left(r - \frac{1}{2} \right)^2 - r^2$$

$$= 1 + r^2 - r + \frac{1}{4} - r^2$$

$$= \frac{5}{4} - r \approx 1 - r$$



Bresenham Midpoint Circle Algorithm Summary

At each point:

If $p_k < 0$:

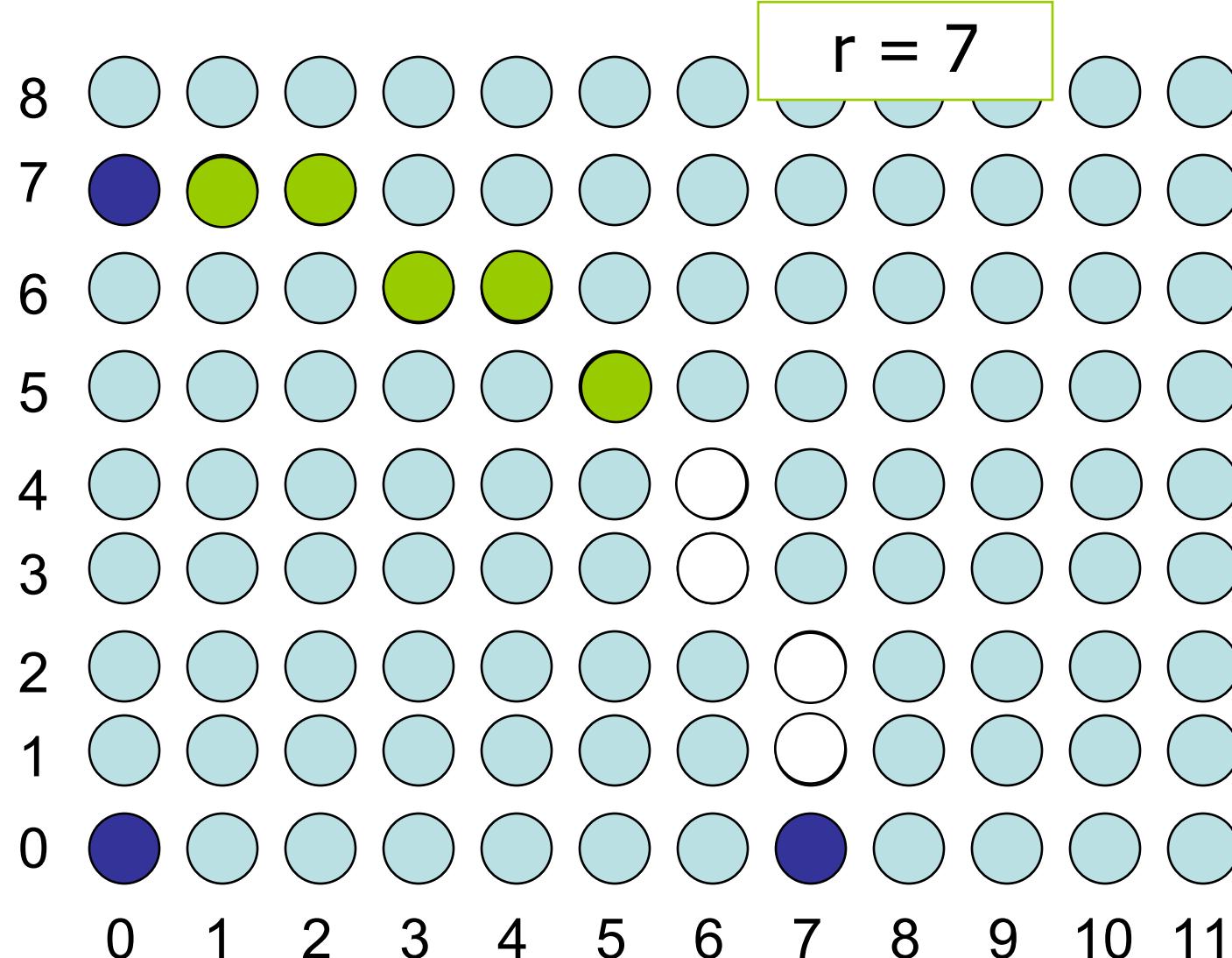
$$\text{Plot}(x_k + 1, y_k)$$
$$p_{k+1} = p_k + 2(x_k + 1) + 1$$

If $p_k \geq 0$:

$$\text{Plot}(x_k + 1, y_k - 1)$$
$$p_{k+1} = p_k + 2(x_k + 1) + 1 - 2(y_k - 1)$$

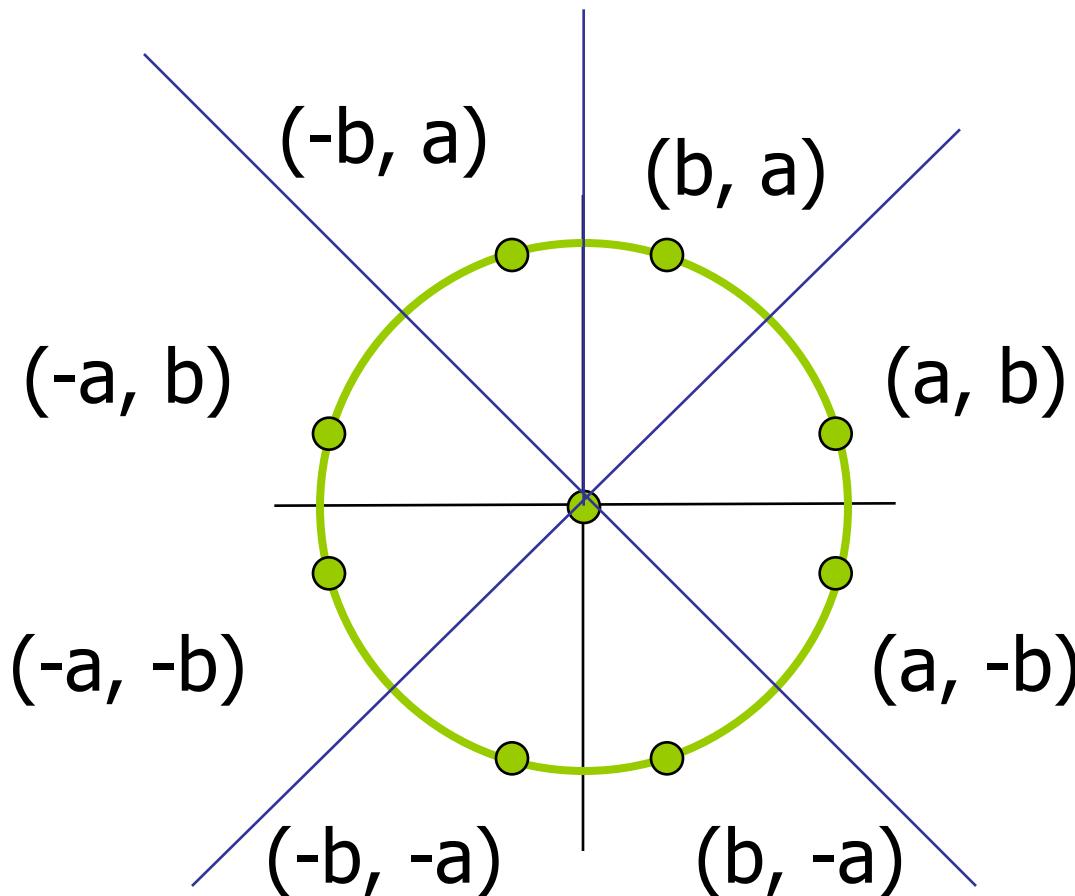


Circle Example 1





Symmetry Optimization



Calculate
points for one
octant;
replicate in
other seven



The Mid-Point Circle Algorithm

MID-POINT CIRCLE ALGORITHM

- Input radius r and circle centre (x_c, y_c) , then set the coordinates for the first point on the circumference of a circle centred on the origin as:

$$(x_0, y_0) = (0, r)$$

- Calculate the initial value of the decision parameter as:

$$p_0 = 1 - r$$

- Starting with $k = 0$ at each position x_k , perform the following test. If $p_k < 0$, the next point along the circle centred on $(0, 0)$ is $(x_k + 1, y_k)$ and:

$$p_{k+1} = p_k + 2x_{k+1} + 1$$



The Mid-Point Circle Algorithm (cont...)

Otherwise the next point along the circle is $(x_k + 1, y_k - 1)$ and:

$$p_{k+1} = p_k + 2x_{k+1} + 1 - 2y_{k+1}$$

4. Determine symmetry points in the other seven octants
5. Move each calculated pixel position (x, y) onto the circular path centred at (x_c, y_c) to plot the coordinate values:

$$x = x + x_c \quad y = y + y_c$$

6. Repeat steps 3 to 5 until $x \geq y$



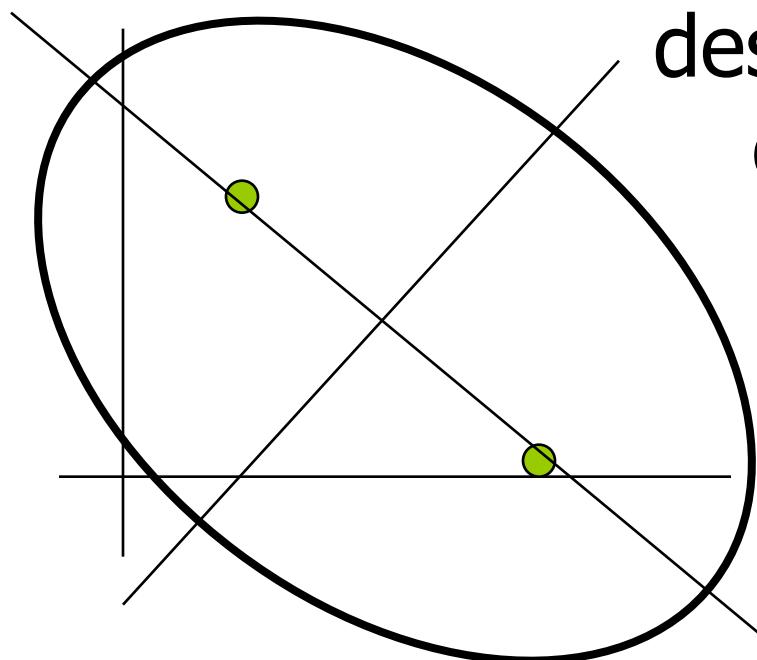
Mid-Point Circle Algorithm Summary

The key insights in the mid-point circle algorithm are:

- Eight-way symmetry can hugely reduce the work in drawing a circle
- Moving in unit steps along the x axis at each point along the circle's edge we need to choose between two possible y coordinates



General Ellipse Equation



In general, ellipse described by constant sum of distances from foci

Difficult to solve equations and plot points
(use parametric polar form?)



Properties of Ellipses

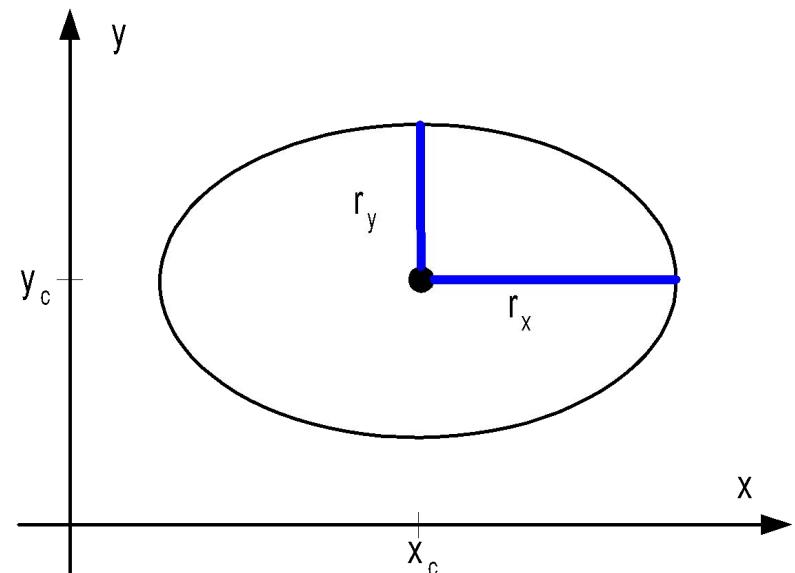
Equation simplified if ellipse axis parallel to coordinate axis

$$\left(\frac{x - x_c}{r_x}\right)^2 + \left(\frac{y - y_c}{r_y}\right)^2 = 1$$

Parametric form

$$x = x_c + r_x \cos \theta$$

$$y = y_c + r_y \sin \theta$$





Symmetry Considerations

4-way symmetry

Unit steps in x until reach region boundary

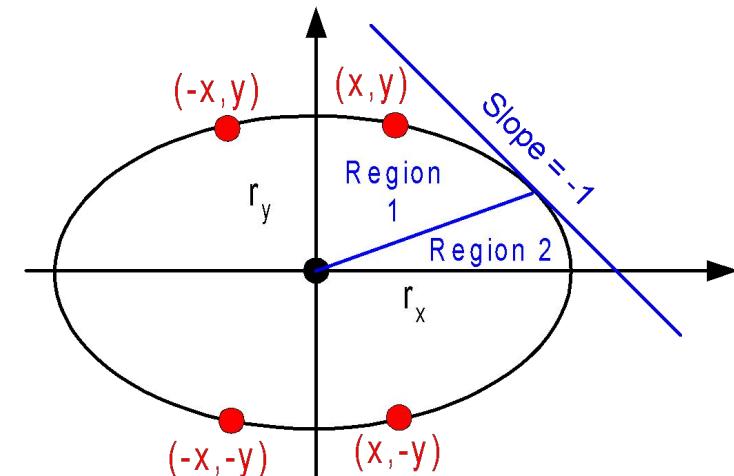
Switch to unit steps in y

$$f(x,y) = r_y^2 x^2 + r_x^2 y^2 - r_x^2 r_y^2$$

$$\frac{dy}{dx} = -\frac{r_y^2 x}{r_x^2 y}$$

$$\frac{dy}{dx} = -1$$

$$r_y^2 x = r_x^2 y$$



- Step in x while
$$r_y^2 x < r_x^2 y$$
- Switch to steps in y when
$$r_y^2 x \geq r_x^2 y$$

$$r_y^2 x \geq r_x^2 y$$



Midpoint Algorithm (initializing)

Similar to circles

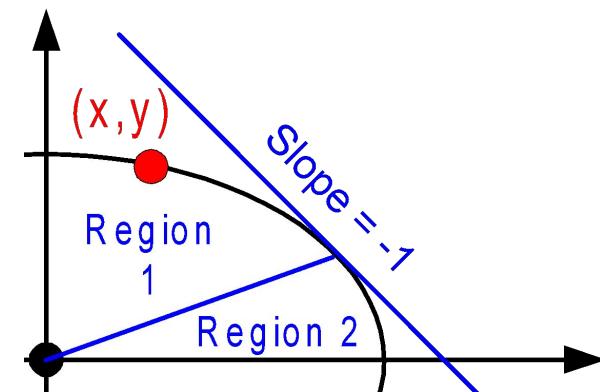
The initial value for region 1

$$\begin{aligned} D_{init1} &= f(1, r_y - \frac{1}{2}) \\ &= r_y^2 - r_x^2 r_y + \frac{1}{4} r_x^2 \end{aligned}$$

The initial value for region 2

$$\begin{aligned} D_{init2} &= f(x_p + \frac{1}{2}, y_p - 1) \\ &= r_y^2 (x_p + \frac{1}{2})^2 + r_x^2 (y_p - 1)^2 - r_x^2 r_y^2 \end{aligned}$$

We have initial values, now we need the increments





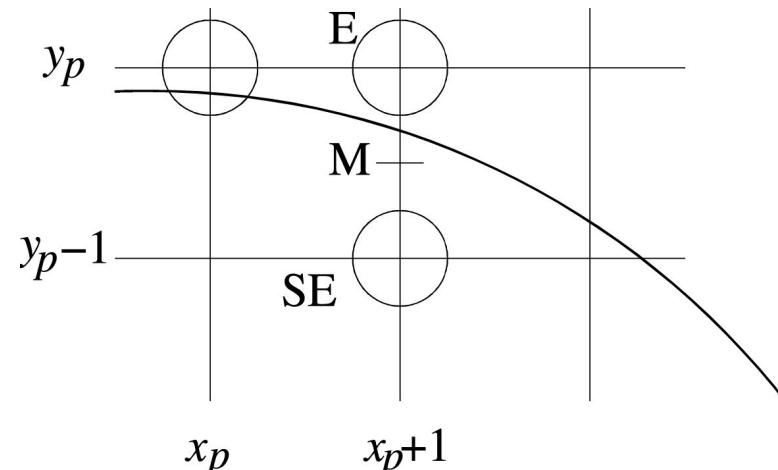
Making a Decision

Computing the decision variable

$$\begin{aligned} D &= f(x_p + 1, y_p - \frac{1}{2}) \\ &= r_y^2(x_p + 1)^2 + r_x^2(y_p - \frac{1}{2})^2 - r_x^2r_y^2 \end{aligned}$$

If $D < 0$ then M is *below* the arc,
hence the E pixel is closer to the line.

If $D \geq 0$ then M is *above* the arc,
hence the SE pixel is closer to the line.





Computing the Increment

E case

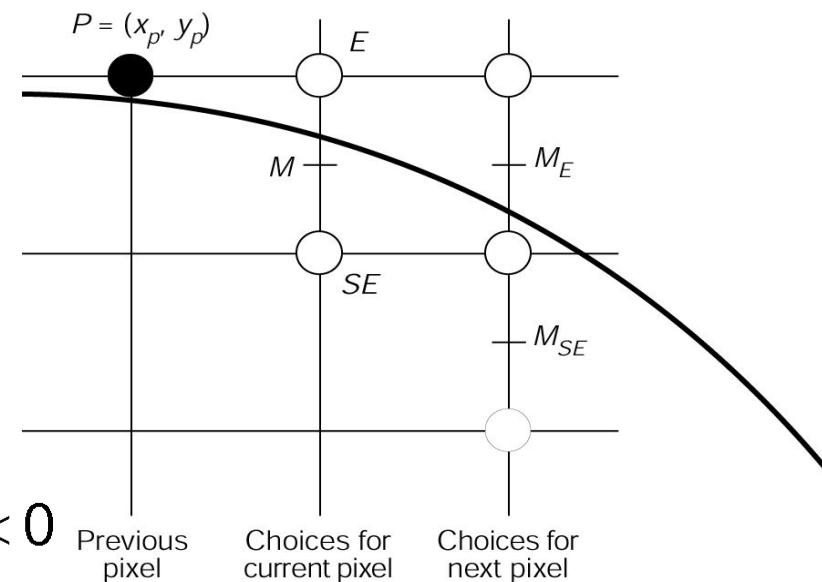
$$\begin{aligned}D_{new} &= f(x_p + 2, y_p - \frac{1}{2}) \\&= r_y^2(x_p + 2)^2 + r_x^2(y_p - \frac{1}{2})^2 - r_x^2r_y^2 \\&= D_{old} + r_y^2(2x_p + 3)\end{aligned}$$

SE case

$$\begin{aligned}D_{new} &= f(x_p + 2, y_p - \frac{3}{2}) \\&= r_y^2(x_p + 2)^2 + r_x^2(y_p - \frac{3}{2})^2 - r_x^2r_y^2 \\&= D_{old} + r_y^2(2x_p + 3) + r_x^2(-2y_p + 2)\end{aligned}$$

increment

$$\begin{cases} r_y^2(2x_p + 3) & D_{old} < 0 \\ r_y^2(2x_p + 3) + r_x^2(-2y_p + 2) & D_{old} \geq 0 \end{cases}$$





Computing the Increment

increment $\begin{cases} 2r_y^2x_{k+1} + r_y^2 & D_k < 0 \quad (1) \\ 2r_y^2x_{k+1} + r_y^2 - 2r_x^2y_{k+1} & D_k \geq 0 \quad (2) \end{cases}$

Computing the increment incrementally...

r_x^2 and r_y^2 can be pre-computed

Only $2r_y^2x$ and $2r_x^2y$ need to be recomputed for each point

At the initial position $(0, r_y)$, $2r_y^2x = 0$ and $2r_x^2y = 2r_x^2r_y$

The new values of the increment can be computed by adding $2r_y^2$ to (1 & 2) and subtracting $2r_x^2$ from (2)



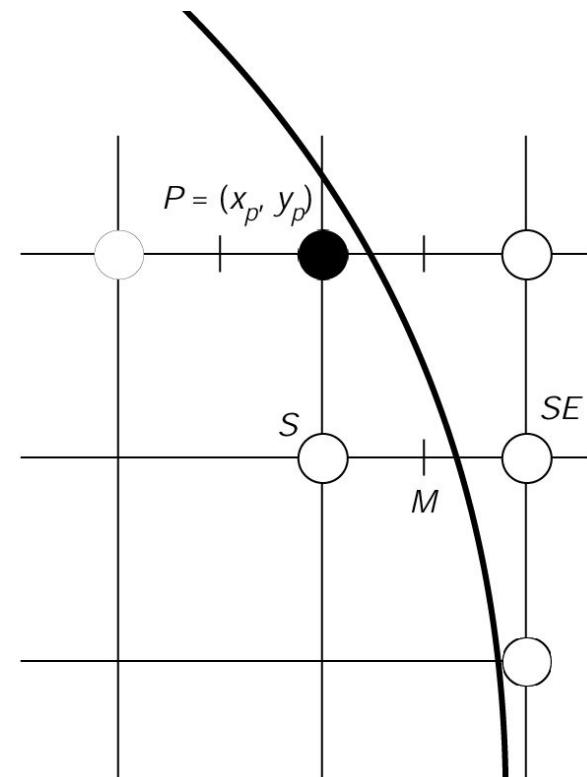
Computing the Increment in 2nd Region

Decision variable in 2nd region

$$\begin{aligned} D &= f(x_p + \frac{1}{2}, y_p - 1) \\ &= r_y^2(x_p + \frac{1}{2})^2 + r_x^2(y_p - 1)^2 - r_x^2r_y^2 \end{aligned}$$

If $D < 0$ then M is *left of* the arc,
hence the *SE* pixel is closer to the line.

If $D \geq 0$ then M is *right of* the arc,
hence the *S* pixel is closer to the line.





Computing the Increment in 2nd Region

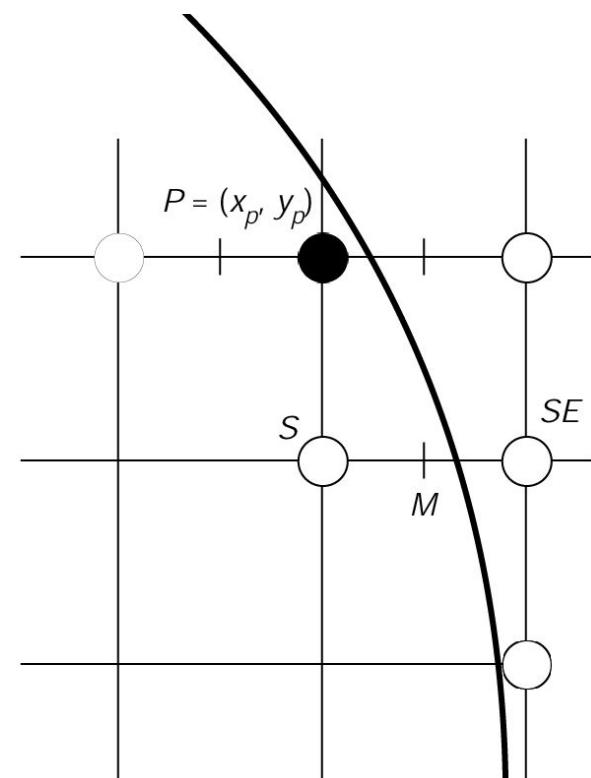
SE case

$$\begin{aligned}D_{new} &= f(x_p + \frac{3}{2}, y_p - 2) \\&= r_y^2(x_p + \frac{3}{2})^2 + r_x^2(y_p - 2)^2 - r_x^2 r_y^2 \\&= D_{old} + r_x^2(-2y_p + 3) + r_y^2(2x_p + 2)\end{aligned}$$

S case

$$\begin{aligned}D_{new} &= f(x_p + \frac{1}{2}, y_p - 2) \\&= r_y^2(x_p + \frac{1}{2})^2 + r_x^2(y_p - 2)^2 - r_x^2 r_y^2 \\&= D_{old} + r_x^2(-2y_p + 3)\end{aligned}$$

increment $\begin{cases} r_x^2(-2y_p + 3) + r_y^2(2x_p + 2) & D_{old} < 0 \\ r_x^2(-2y_p + 3) & D_{old} \geq 0 \end{cases}$





Midpoint Ellipse Algorithm

```
void MidpointEllipse (int a, int b, int value)
/* Assumes center of ellipse is at the origin. Note that overflow may occur */
/* for 16-bit integers because of the squares. */
{
    double d2;

    int x = 0;
    int y = b;
    double dI = b2 - (a2b) + (0.25 a2);
    EllipsePoints (x, y, value);           /* The 4-way symmetrical WritePixel */

    /* Test gradient if still in region 1 */
    while ( a2(y - 0.5) > b2(x + 1) ) { /* Region 1 */
        if (dI < 0)                         /* Select E */
            dI += b2(2x + 3);
        else {                                /* Select SE */
            dI += b2(2x + 3) + a2(-2y + 2);
            y--;
        }
        x++;
        EllipsePoints (x, y, value);
    } /* Region 1 */
}
```



Midpoint Ellipse Algorithm (cont.)

```
d2 = b2(x + 0.5)2 + a2(y - 1)2 - a2b2;
while (y > 0) {                                /* Region 2 */
    if (d2 < 0) {                                /* Select SE */
        d2 += b2(2x + 2) + a2(-2y + 3);
        x++;
    } else
        d2 += a2(-2y + 3);                      /* Select S */
    y--;
    EllipsePoints (x, y, value);
} /* Region 2 */
} /* MidpointEllipse */
```