Dr. George Karraz, Ph. D.

1

Bresenham Line Drawing Algorithm, Circle Drawing & Polygon Filling

Contents

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

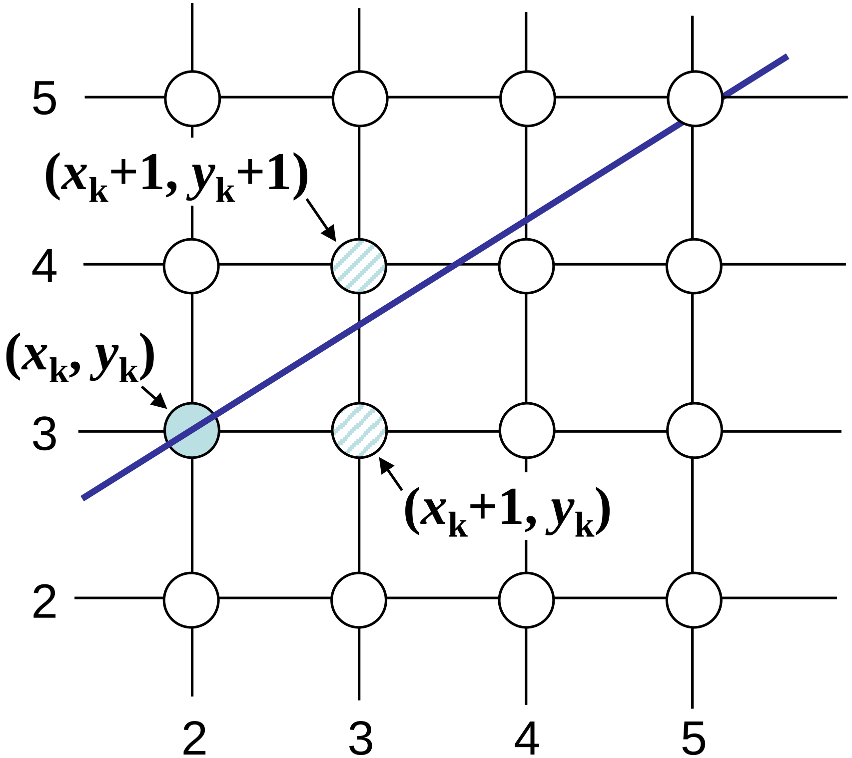
The Bresenham algorithm is another incremental scan conversion algorithm

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

The Big Idea

Move across the x axis in unit intervals and at each step choose between two different y coordinates

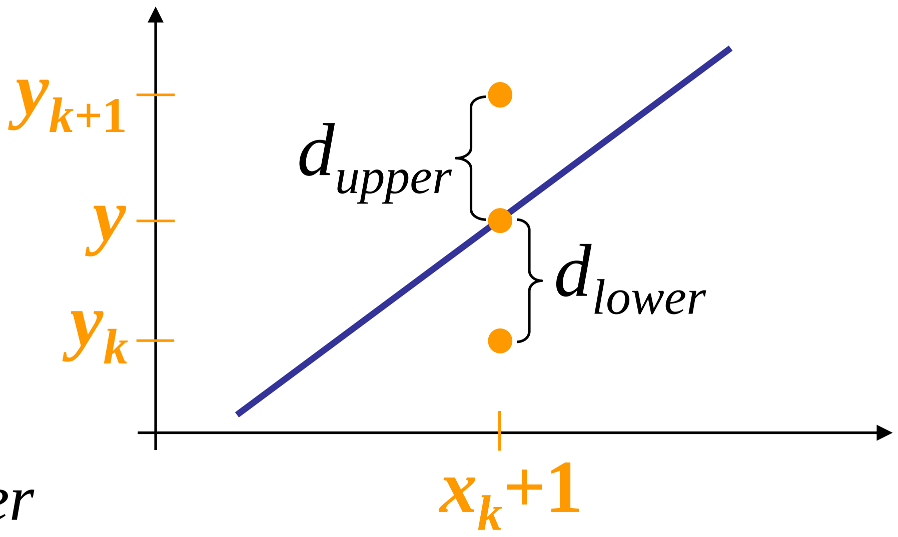


For example, from position (2, 3) we have to choose between (3, 3) and (3, 4)

We would like the point that is closer to the original line

Deriving The Bresenham 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:

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

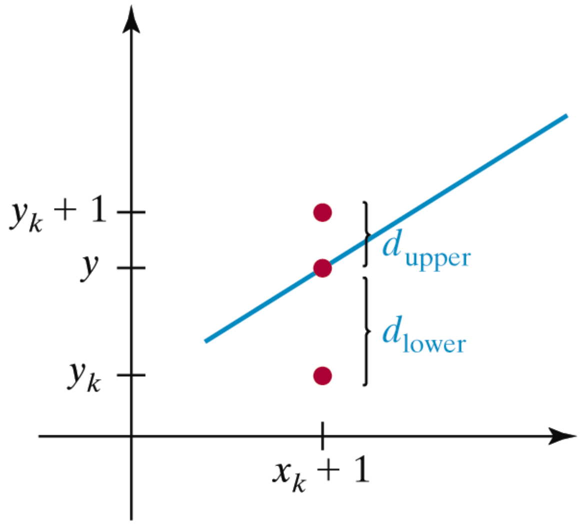


Figure 3-11

Vertical distances between pixel positions and the line y coordinate at sampling position $x_k + 1$.

Deriving The Bresenham Line Algorithm (cont…)

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

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

and:

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

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

Deriving The Bresenham Line Algorithm (cont…)

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:

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

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

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

Deriving The Bresenham Line Algorithm (cont…)

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

Deriving The Bresenham Line Algorithm (cont…)

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:

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

Deriving The Bresenham Line Algorithm (cont…)

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 p0 is evaluated at (x0, y0) is given as:

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

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

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

Bresenham Example

Let's have a go at this

Let's plot the line from (20, 10) to (30, 18)

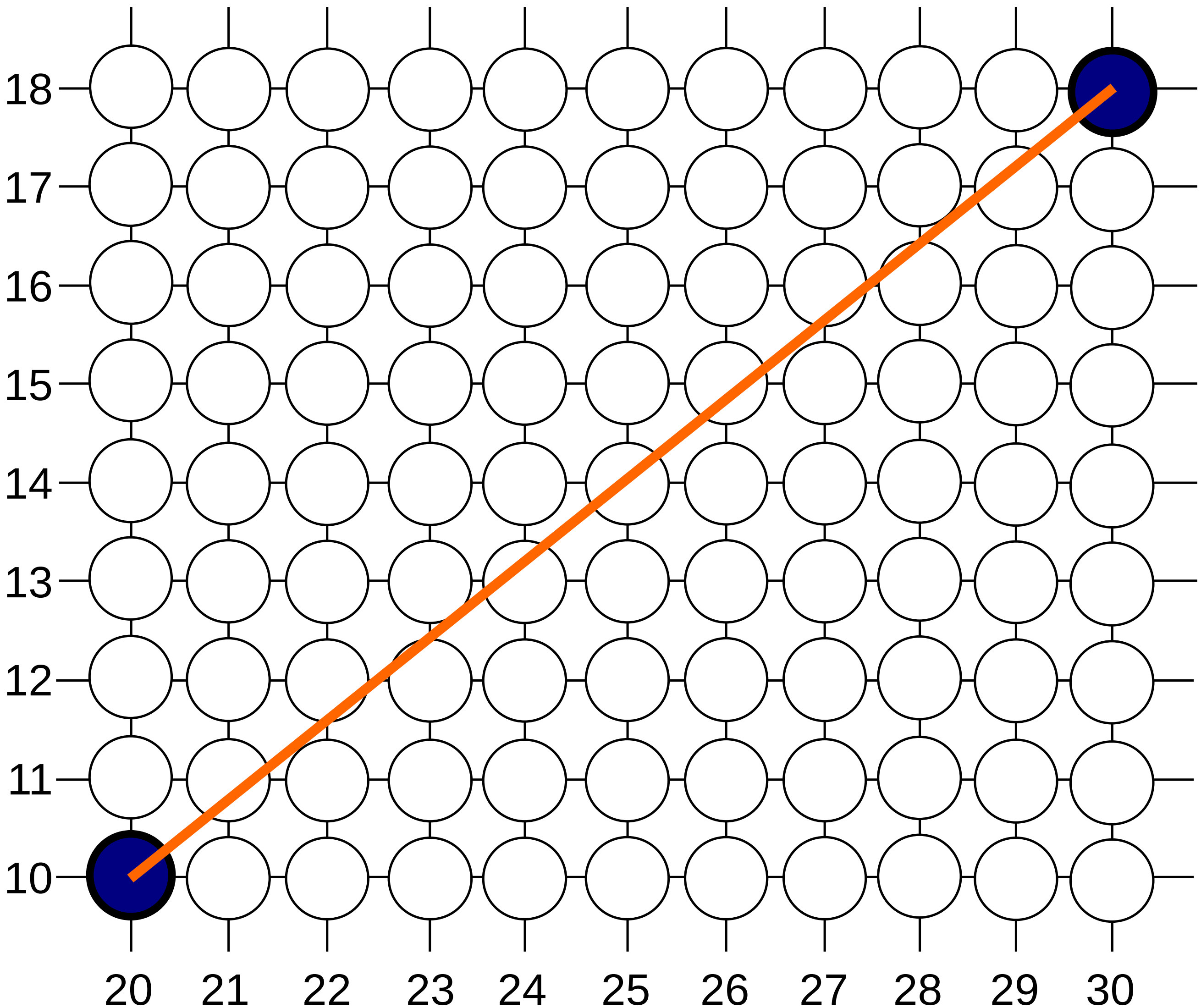
First off calculate all of the constants:

- Δx: 10
- Δy: 8
- $2\Delta y$: 16
- $2\Delta y$ - $2\Delta x$: -4

Calculate the initial decision parameter p_0:

- $p0 = 2\Delta y - \Delta x$ = 6

Bresenham Example (cont…)



k	p_k	(x_{k+1},y_{k+1})
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		

Bresenham Line Algorithm Summary

The Bresenham line algorithm has the following advantages:

- An 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

A Simple Circle Drawing Algorithm

The equation for a circle is:

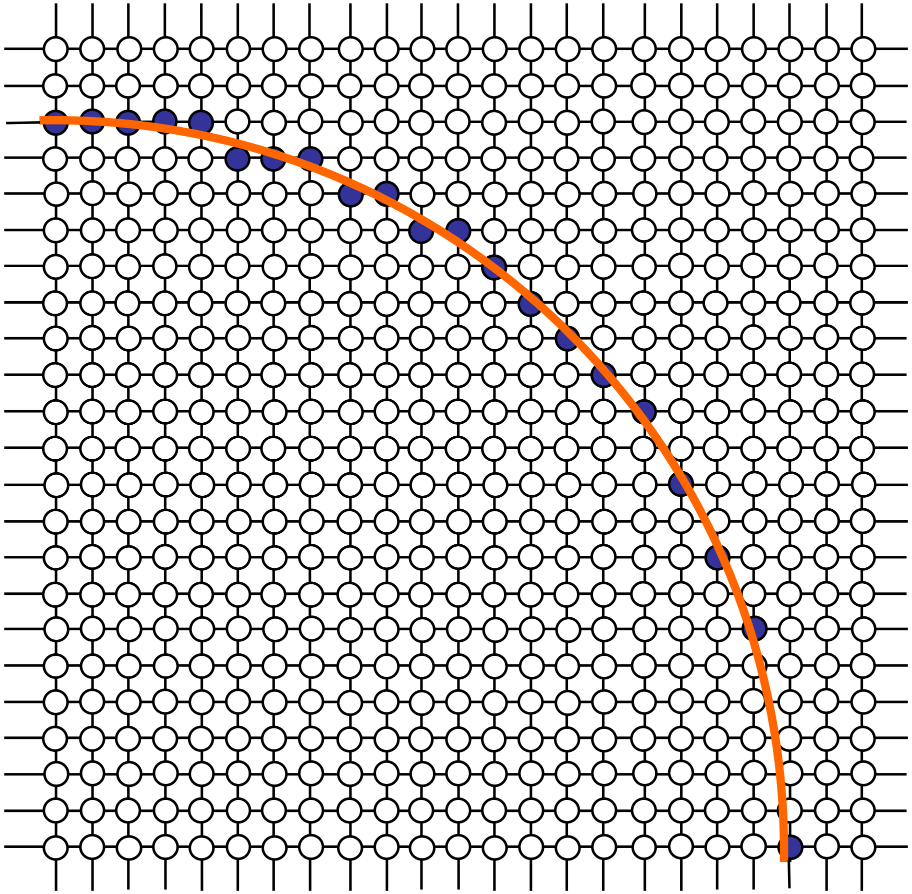
$$x^2 + y^2 = r^2$$

where r is the radius of the circle

So, we can write a simple circle drawing algorithm by solving the equation for y at unit x intervals using:

$$y = \pm\sqrt{r^2 - x^2}$$

A Simple Circle Drawing Algorithm (cont…)



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

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

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

$$\vdots$$

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

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

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

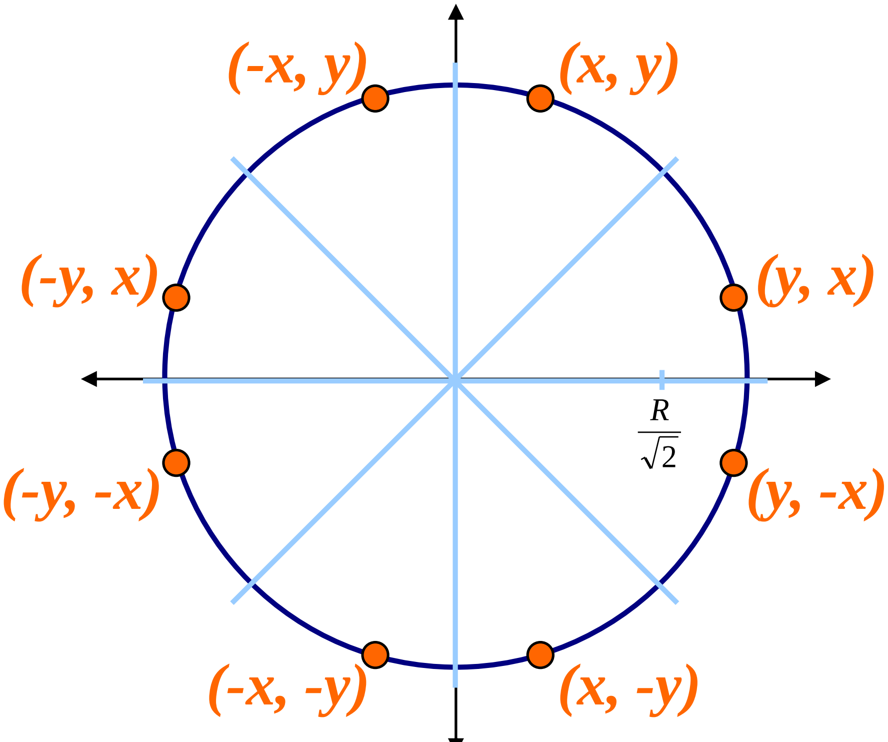
Secondly, the calculations are not very efficient

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

We need a more efficient, more accurate solution

Eight-Way Symmetry

The first thing we can notice to make our circle drawing algorithm more efficient is that circles centred at $(0, 0)$ have *eight-way symmetry*



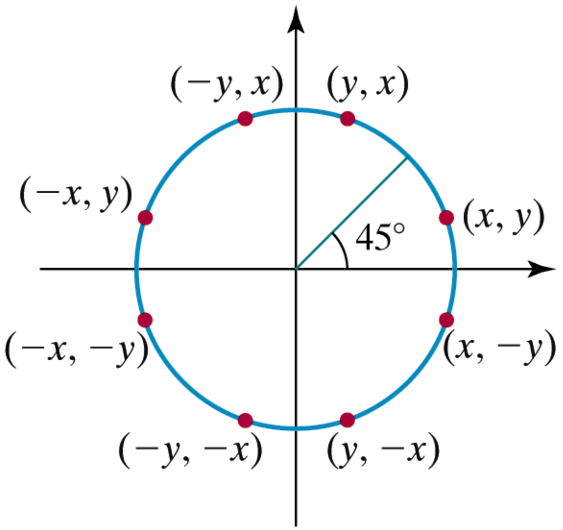


Figure 3-18

Symmetry of a circle. Calculation of a circle point (x, y)in one octant yields the circle points shown for the other seven octants.

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

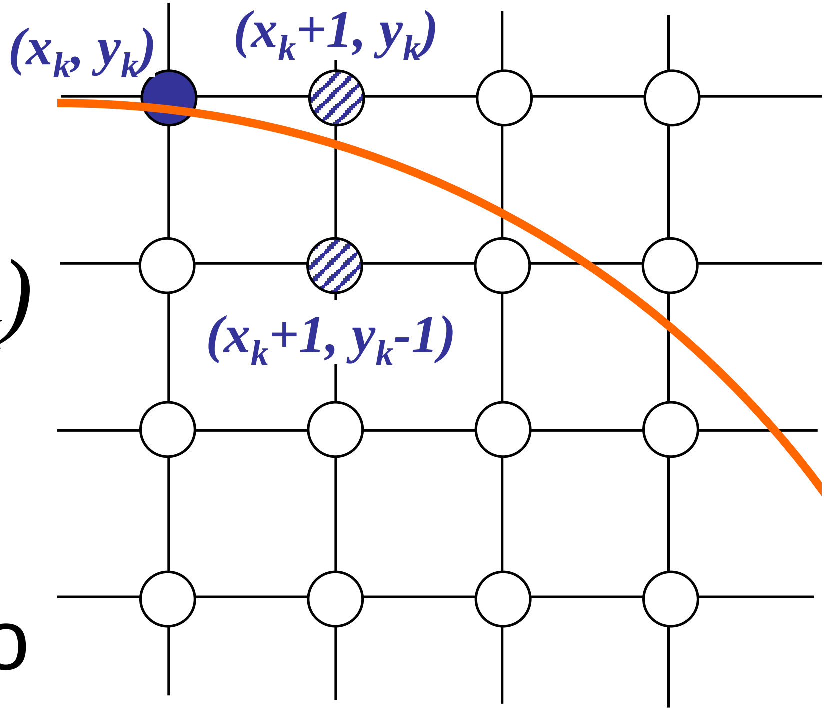
The mid-point circle algorithm was developed by Jack Bresenham, who we heard about earlier.

Mid-Point Circle Algorithm (cont…)

Assume that we have just plotted point (x_k, y_k)

The next point is a choice between (x_k+1, y_k) and (x_k+1, y_k-1)

We would like to choose the point that is nearest to the actual circle



So how do we make this choice?

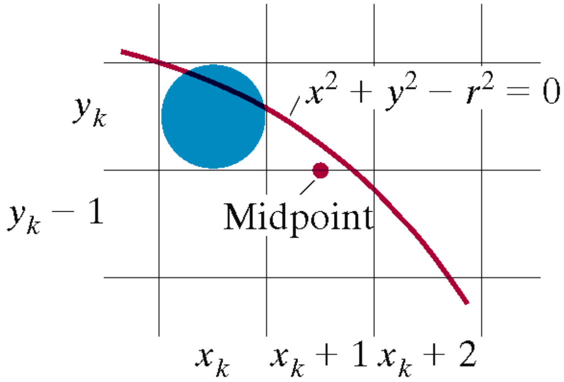


Figure 3-19

Midpoint between candidate pixels at sampling position $x_k + 1$ along a circular path.

Mid-Point Circle Algorithm (cont…)

Let’s re-jig the equation of the circle slightly to give us:

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

The equation evaluates as follows:

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

By evaluating this function at the midpoint between the candidate pixels we can make our decision

Mid-Point Circle Algorithm (cont…)

Assuming we have just plotted the pixel at (x_k, y_k) so we need to choose between (x_k+1, y_k) and (x_k+1, y_k-1)

Our decision variable can be defined as:

$$p_k = f_{circ}(x_k + 1, y_k - \tfrac{1}{2})$$
$$= (x_k + 1)^2 + (y_k - \tfrac{1}{2})^2 - r^2$$

If $p_k < 0$ the midpoint is inside the circle and the pixel at y_k is closer to the circle

Otherwise the midpoint is outside and y_k-1 is closer

Mid-Point Circle Algorithm (cont…)

To ensure things are as efficient as possible we can do all of our calculations incrementally

First consider:

$$p_{k+1} = f_{circ}\left(x_{k+1}+1, y_{k+1}-\tfrac{1}{2}\right)$$

$$= [(x_k+1)+1]^2 + \left(y_{k+1}-\tfrac{1}{2}\right)^2 - r^2$$

or:

$$p_{k+1} = p_k + 2(x_k+1) + (y_{k+1}^2 - y_k^2) - (y_{k+1} - y_k) + 1$$

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

Mid-Point Circle Algorithm (cont…)

The first decision variable is given as:

$$p_0 = f_{circ}(1, r - \tfrac{1}{2})$$
$$= 1 + (r - \tfrac{1}{2})^2 - r^2$$
$$= \tfrac{5}{4} - r$$

Then if $p_k < 0$ then the next decision variable is given as:

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

If $p_k > 0$ then the decision variable is:

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

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:

$$p_0 = \frac{5}{4} - r$$

- 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:

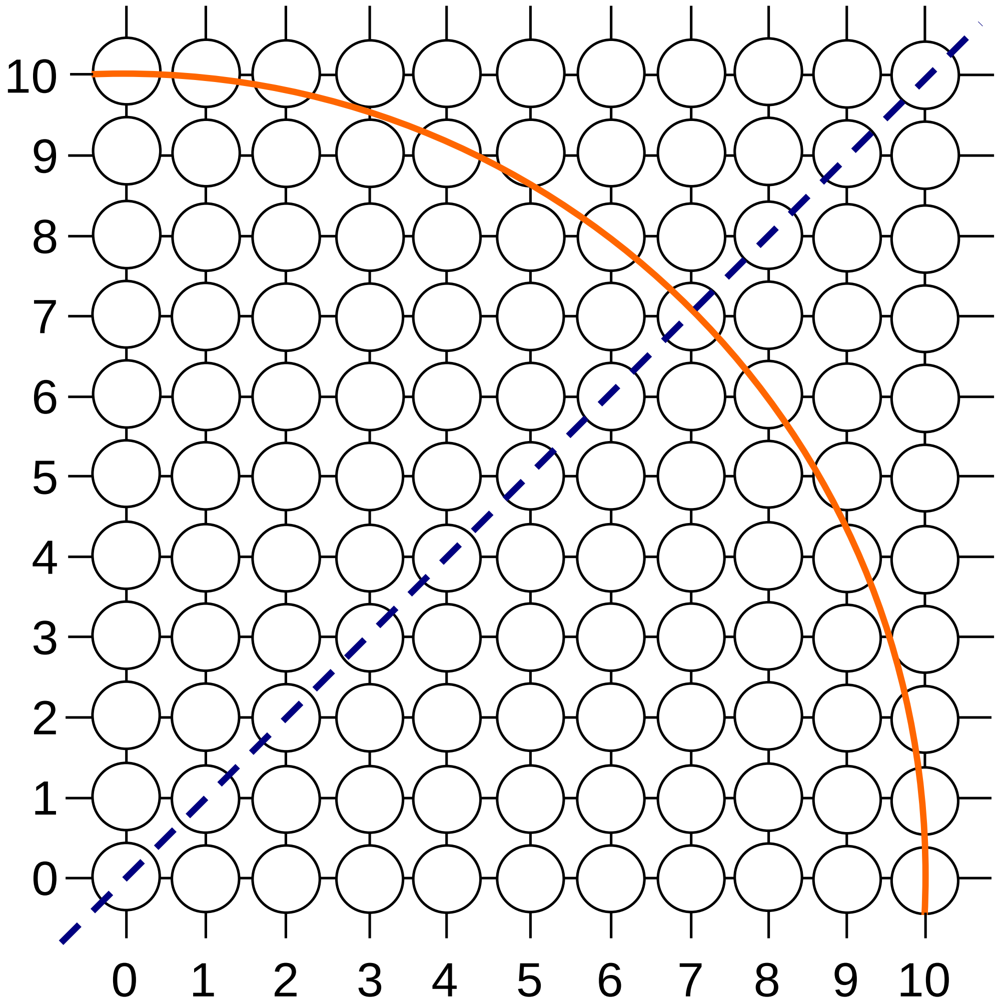
$$x = x + x_c \qquad y = y + y_c$$

6. Repeat steps 3 to 5 until $x >= y$

Mid-Point Circle Algorithm Example

To see the mid-point circle algorithm in action lets use it to draw a circle centred at (0,0) with radius 10

Mid-Point Circle Algorithm Example (cont…)



k	p_k	(x_{k+1},y_{k+1})	$2x_{k+1}$	$2y_{k+1}$
0				
1				
2				
3				
4				
5				
6				

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

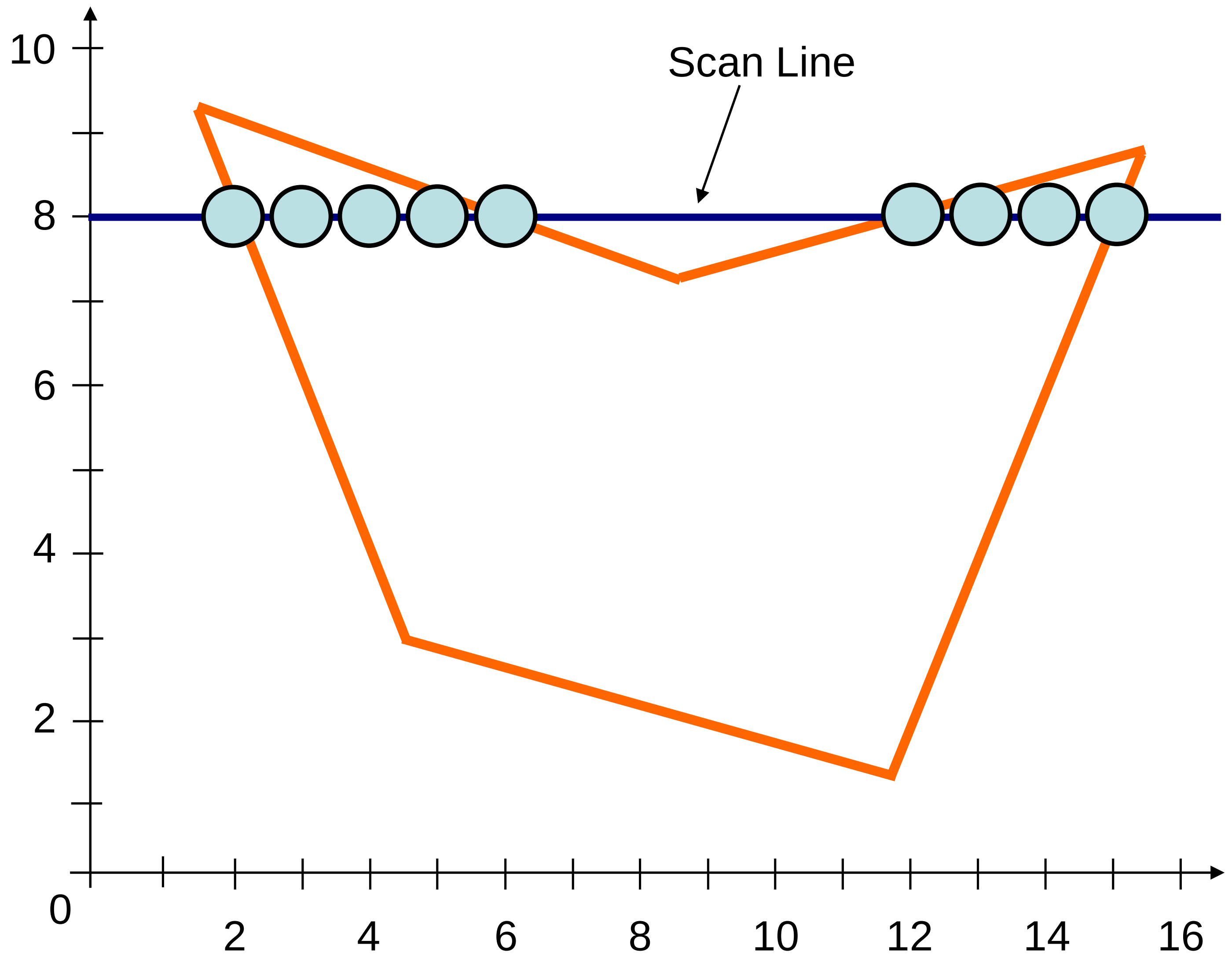
Filling Polygons

So we can figure out how to draw lines and circles

How do we go about drawing polygons?

We use an incremental algorithm known as the scan-line algorithm

Scan-Line Polygon Fill Algorithm

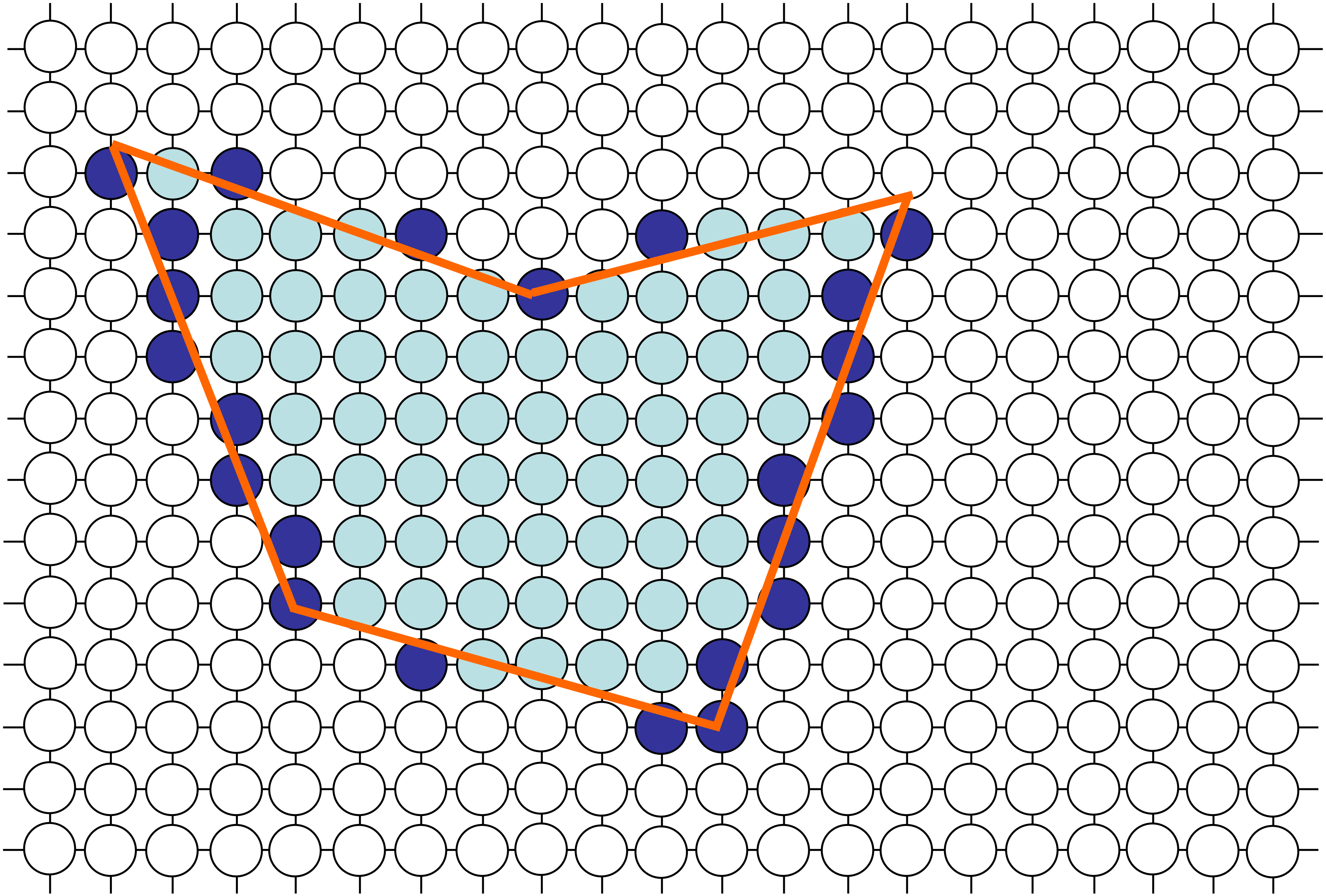


Scan-Line Polygon Fill Algorithm

The basic scan-line algorithm is as follows:

- Find the intersections of the scan line with all edges of the polygon
- Sort the intersections by increasing x coordinate
- Fill in all pixels between pairs of intersections that lie interior to the polygon

Scan-Line Polygon Fill Algorithm (cont…)



Line Drawing Summary

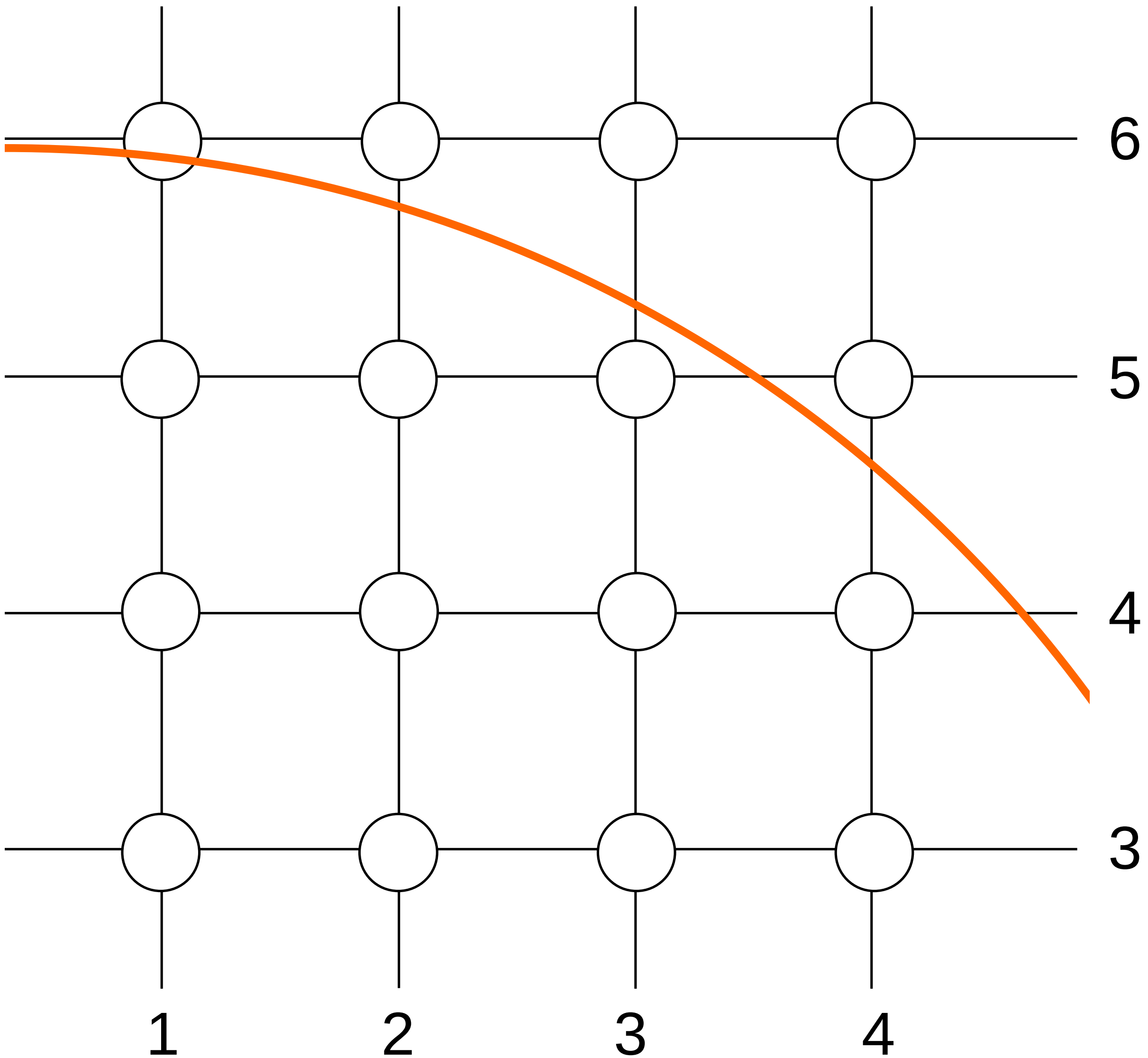
Over the last couple of lectures we have looked at the idea of scan converting lines

The key thing to remember is this has to be **FAST**

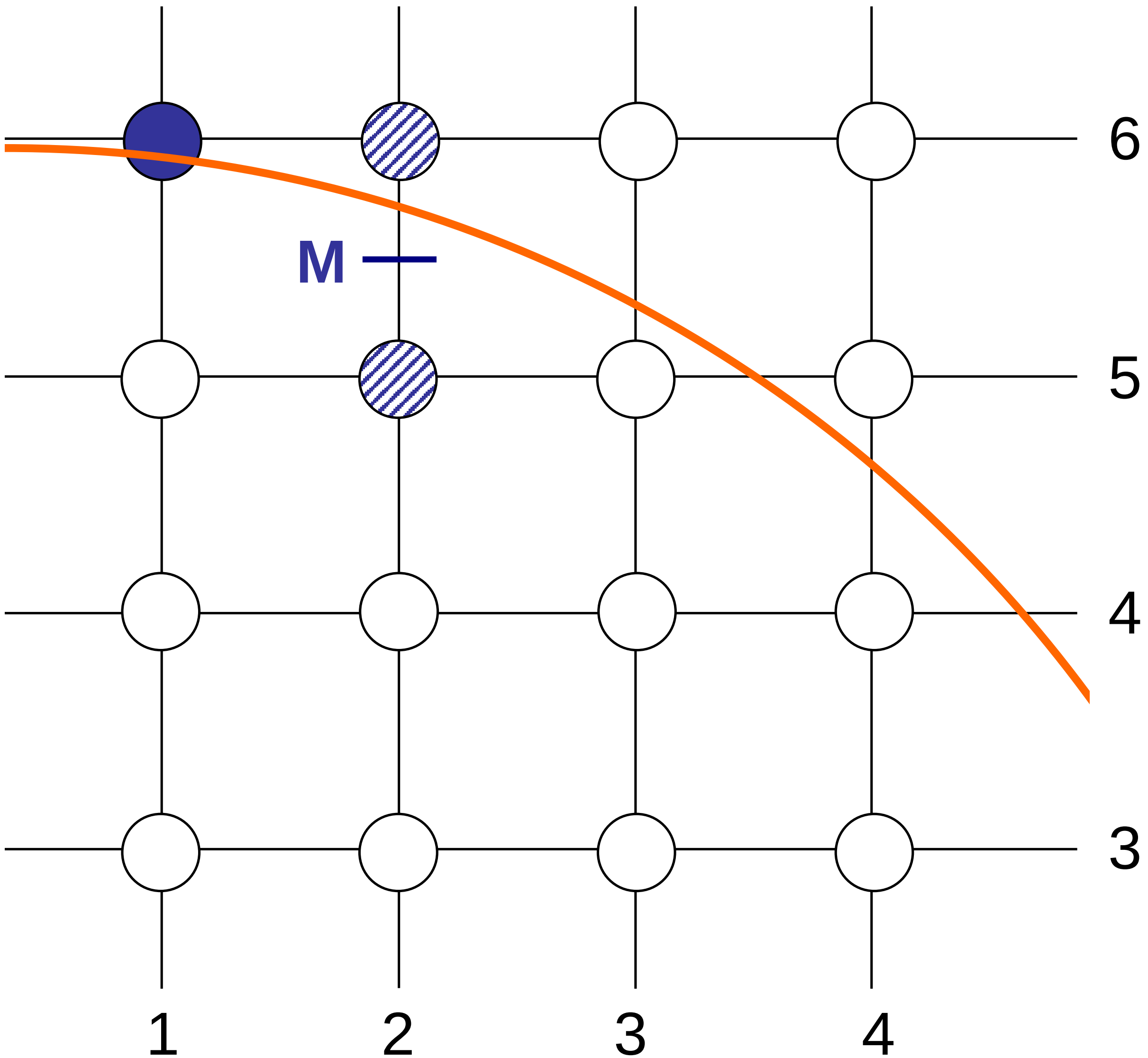
For lines we have either DDA or Bresenham

For circles the mid-point algorithm

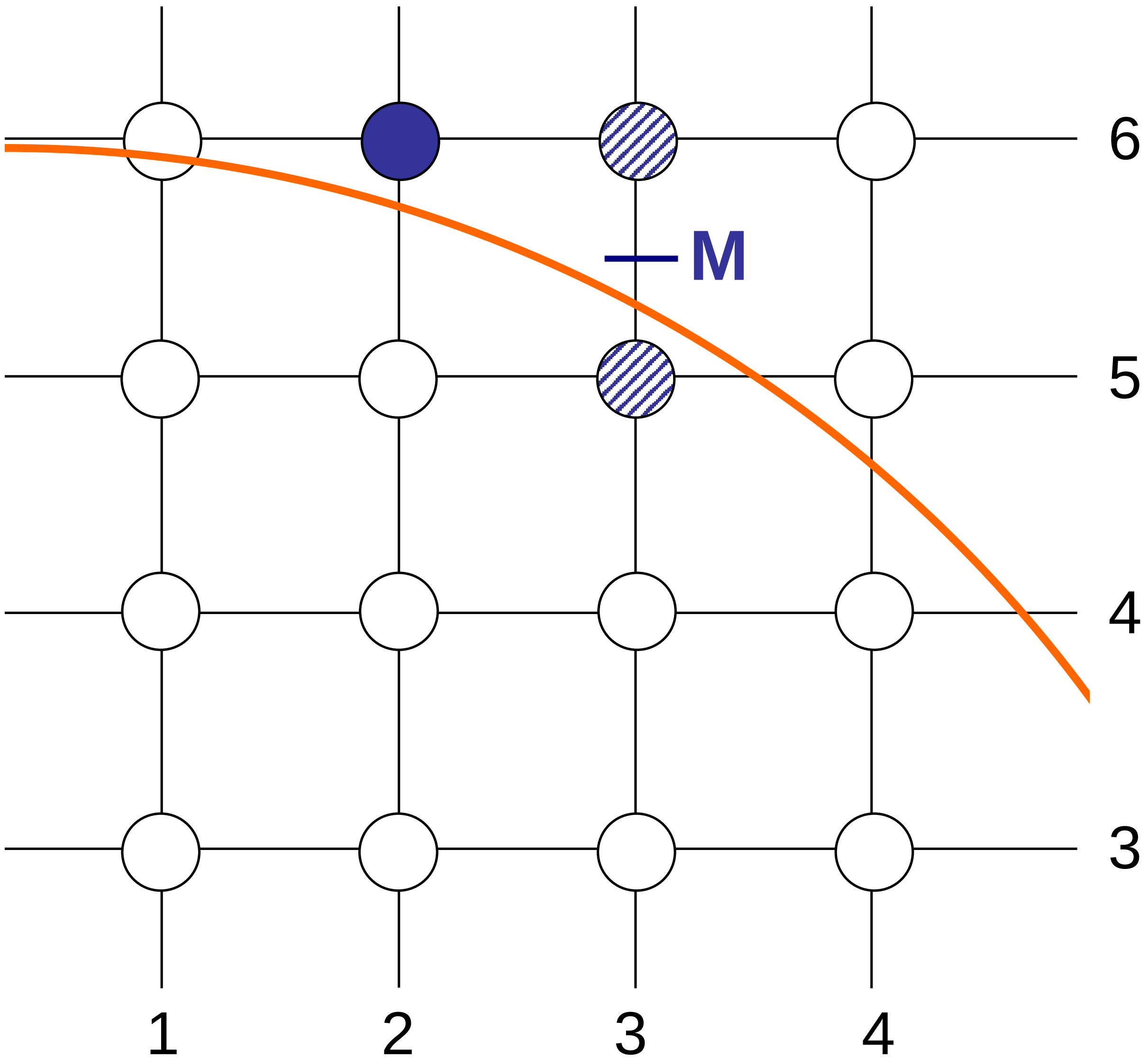
Mid-Point Circle Algorithm (cont…)



Mid-Point Circle Algorithm (cont…)



Mid-Point Circle Algorithm (cont…)



Blank Grid