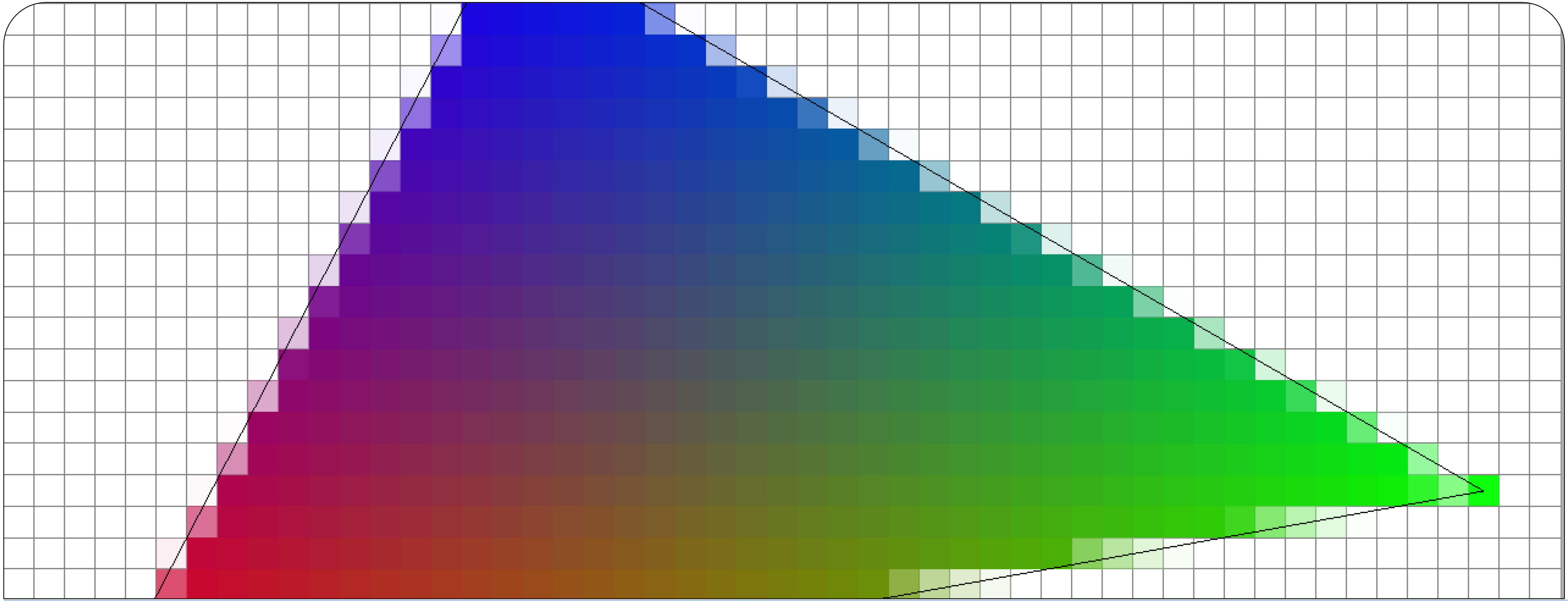


Computer Graphics

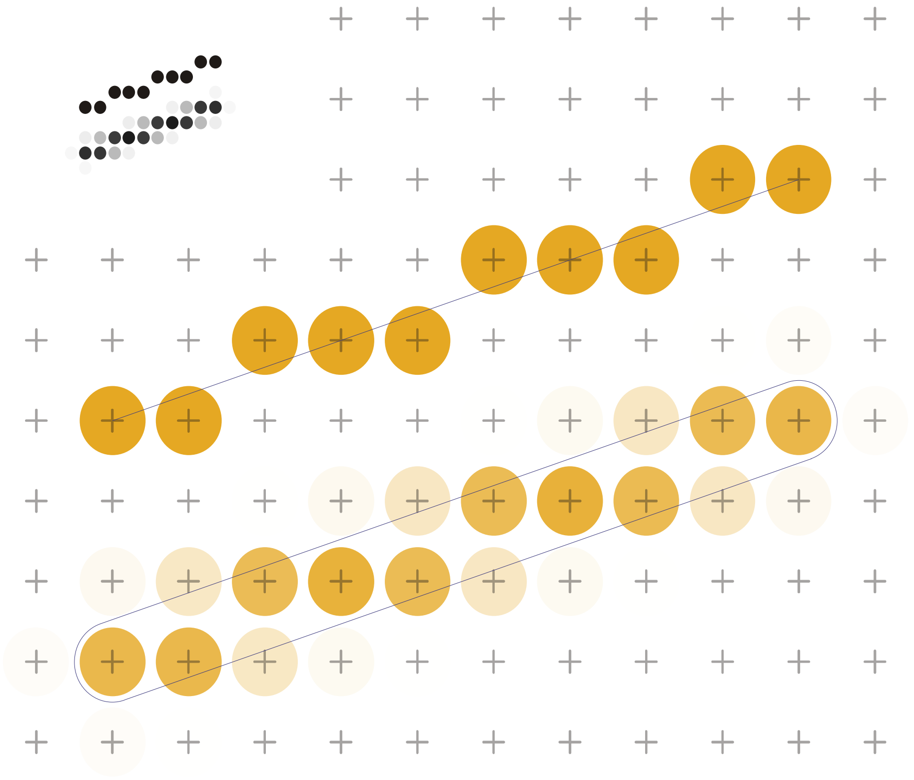
Scan Conversion - rasterization

Andreas Aristidou
andarist@ucy.ac.cy
http://www.andreasaristidou.com

What is line scan conversion

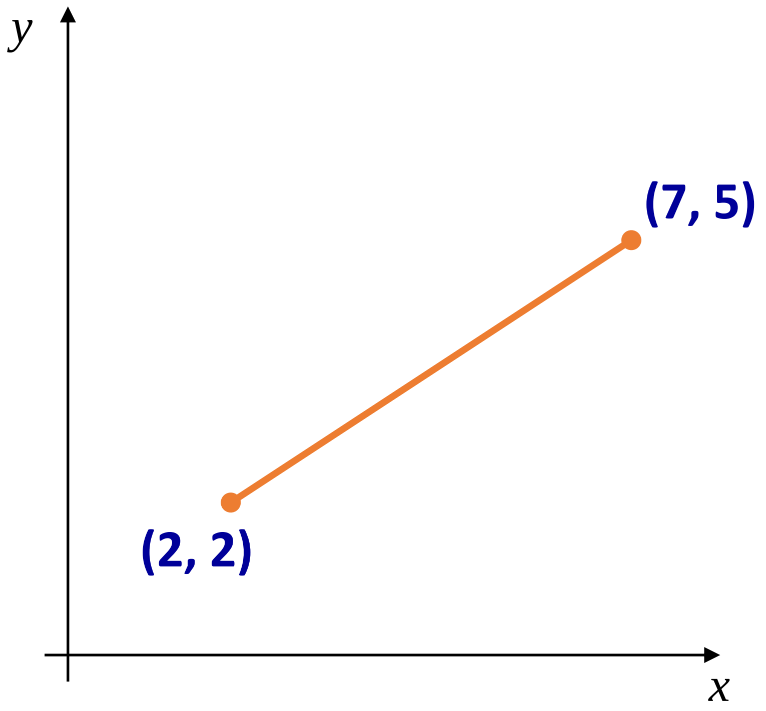
- This is the last stage of rasterization (the process in which geometric elements are converted to tables by pixels and stored in the framebuffer to be viewed)
- It follows clipping
- All graphics packages scan at the end of the rendering pipeline
- Triangles (or higher complexity polygons) are converted to pixels
- For 3D rendering, we take into account other processes, such as lighting and shading, but we will focus first on algorithms for **line scan conversion**

Line drawing algorithms



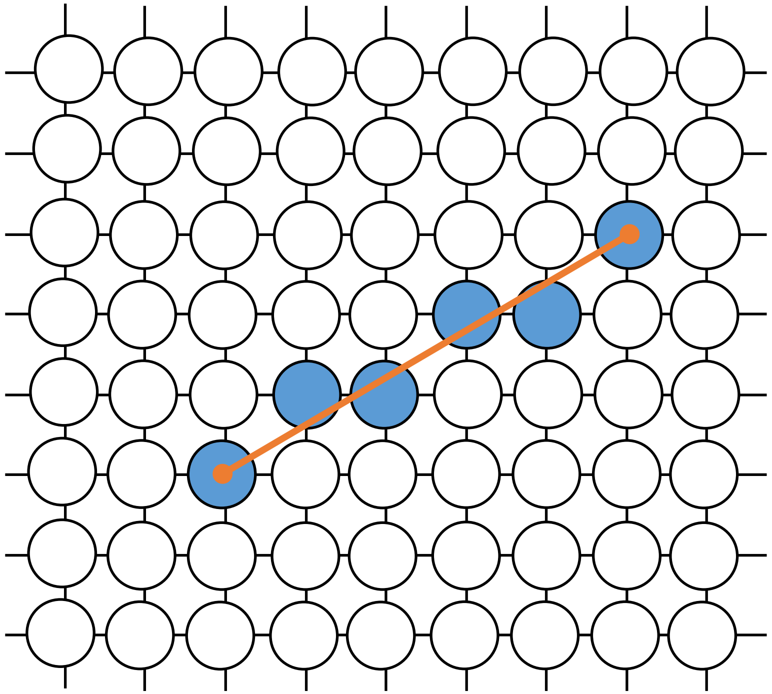
The Problem Of Scan Conversion

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



The Problem Of Scan Conversion

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



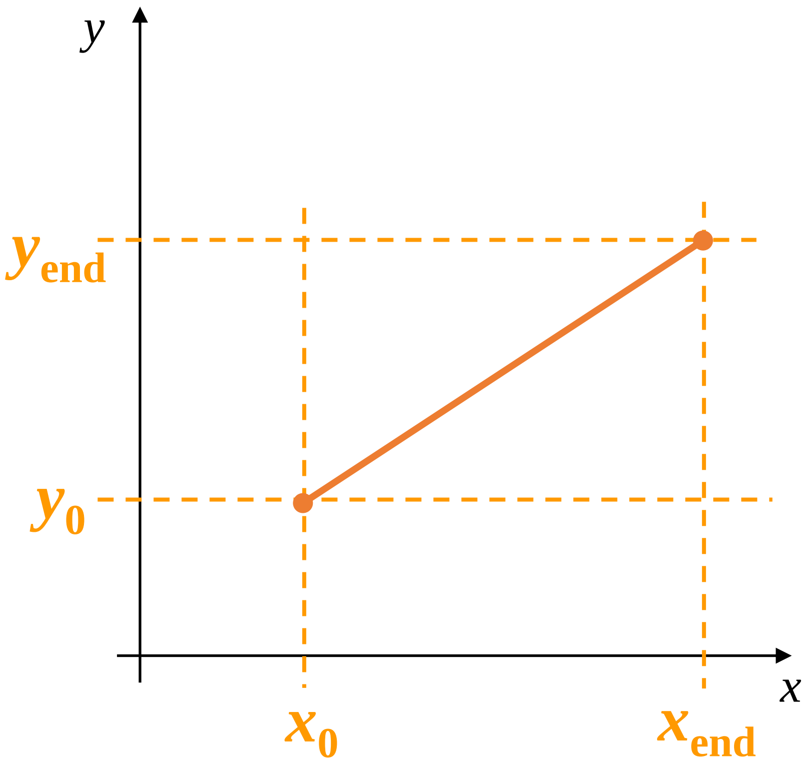
How do we choose which pixels to turn on?

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:

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

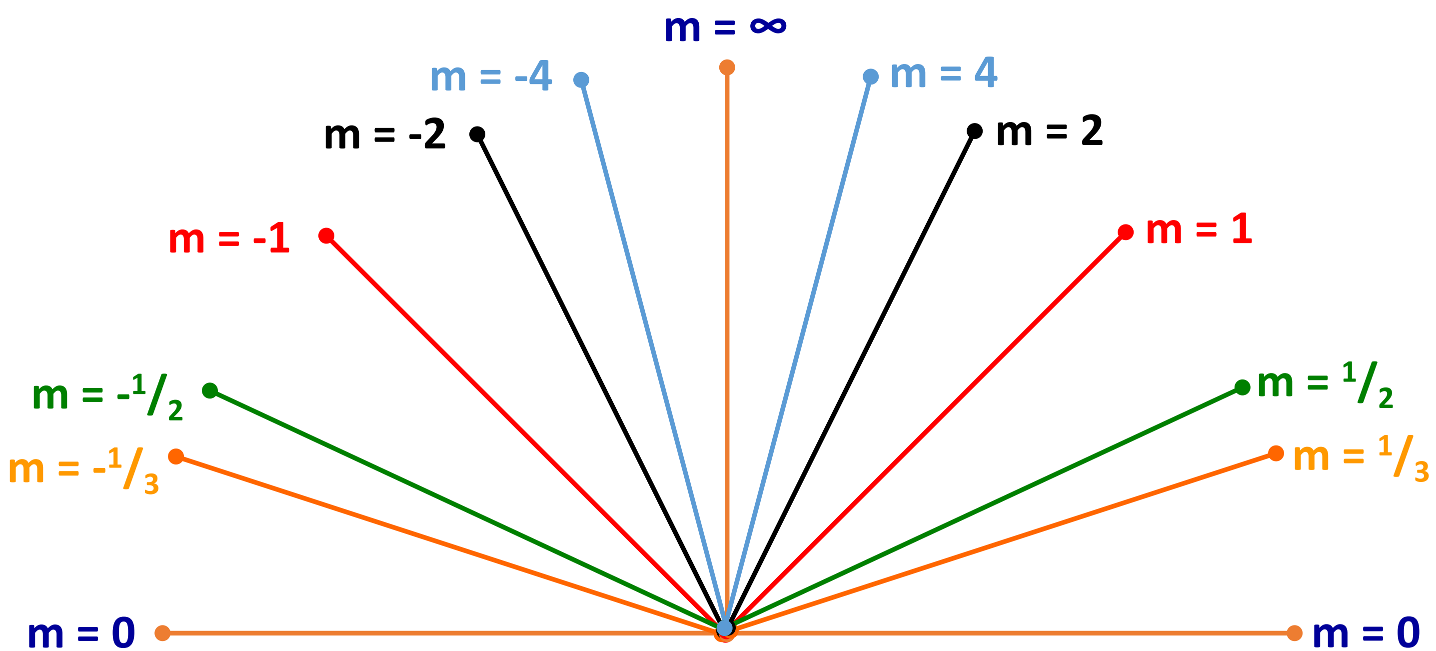
where:

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

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

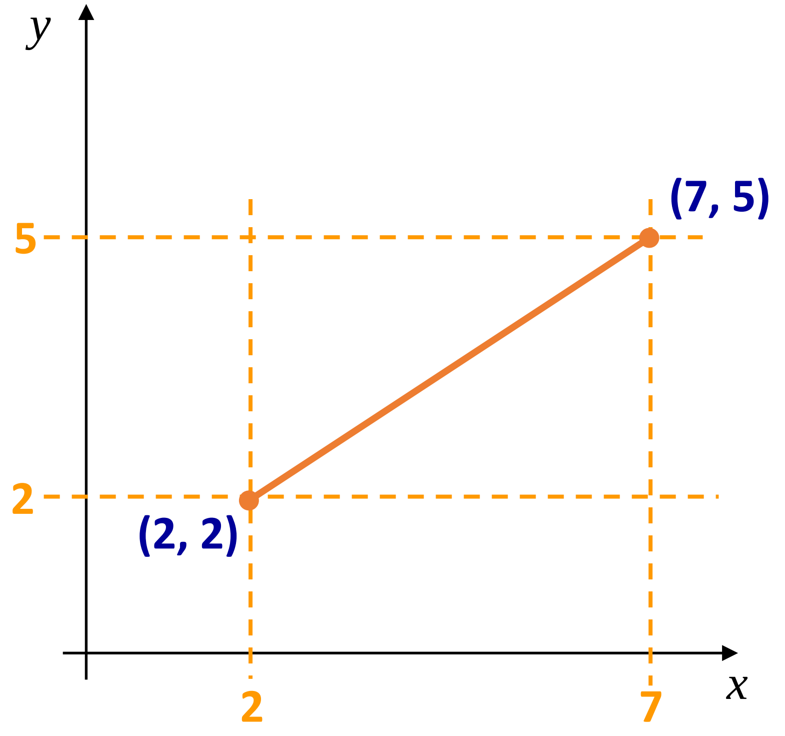
Lines & Slopes

- 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



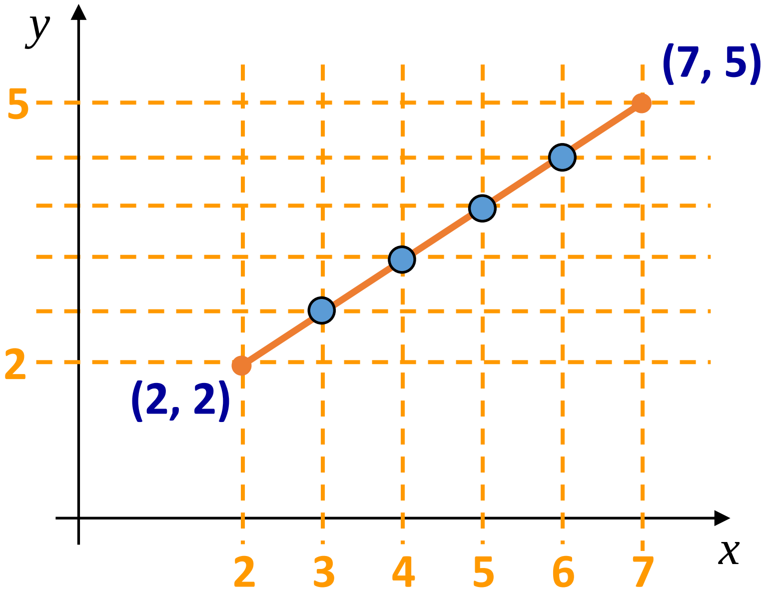
A Very Simple Solution

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



A Very Simple Solution

- First work out m and b :



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

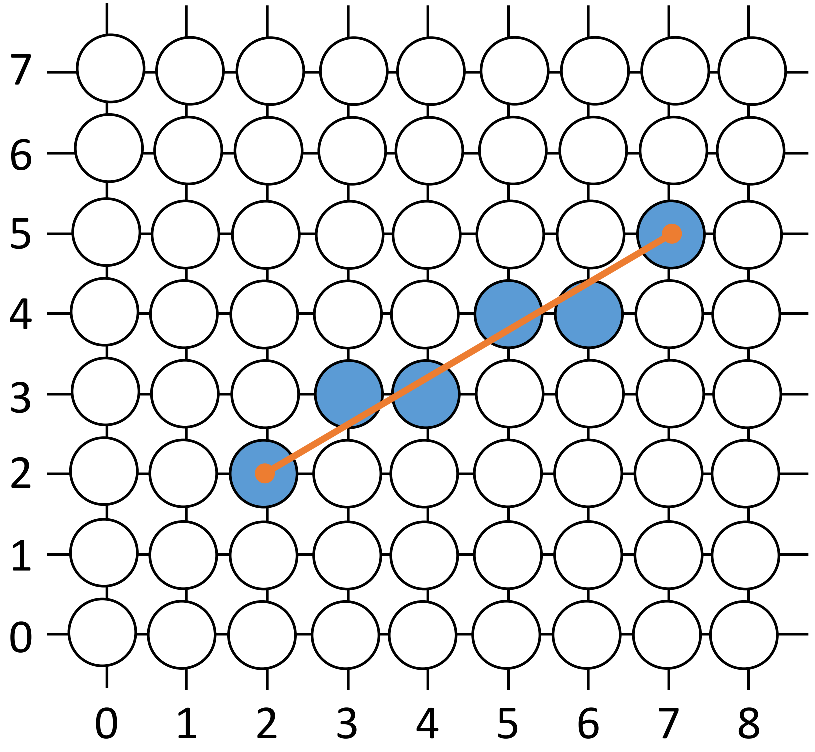
$$b = 2 - \frac{3}{5} * 2 = \frac{4}{5}$$

- Now for each x value work out the y value:

$$y(3) = \frac{3}{5} \cdot 3 + \frac{4}{5} = 2\frac{3}{5} \quad y(4) = \frac{3}{5} \cdot 4 + \frac{4}{5} = 3\frac{1}{5} \quad y(5) = \frac{3}{5} \cdot 5 + \frac{4}{5} = 3\frac{4}{5} \quad y(6) = \frac{3}{5} \cdot 6 + \frac{4}{5} = 4\frac{2}{5}$$

A Very Simple Solution

- 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

- 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

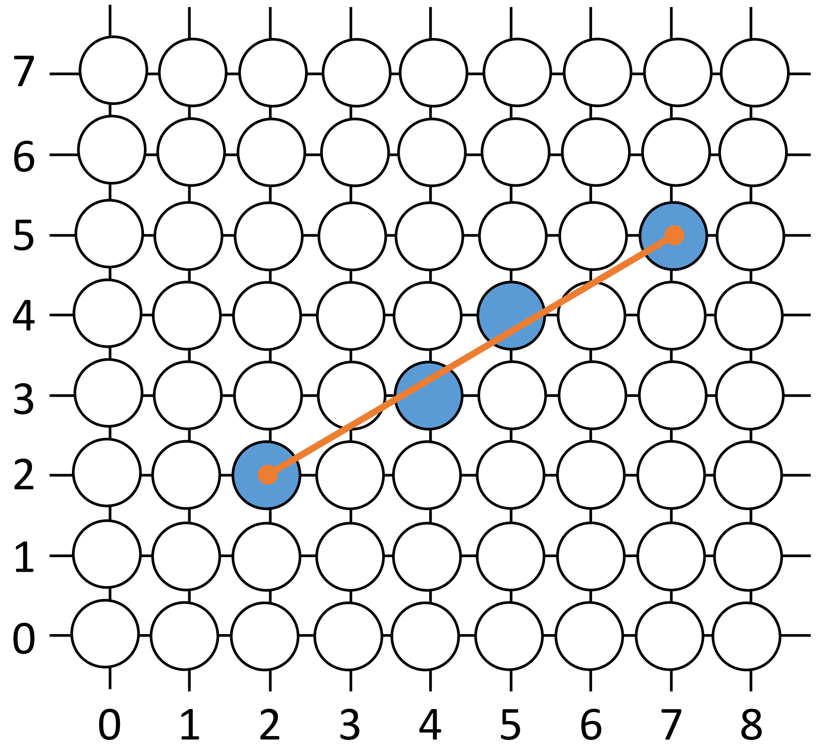
A Quick Note About Slopes

- 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

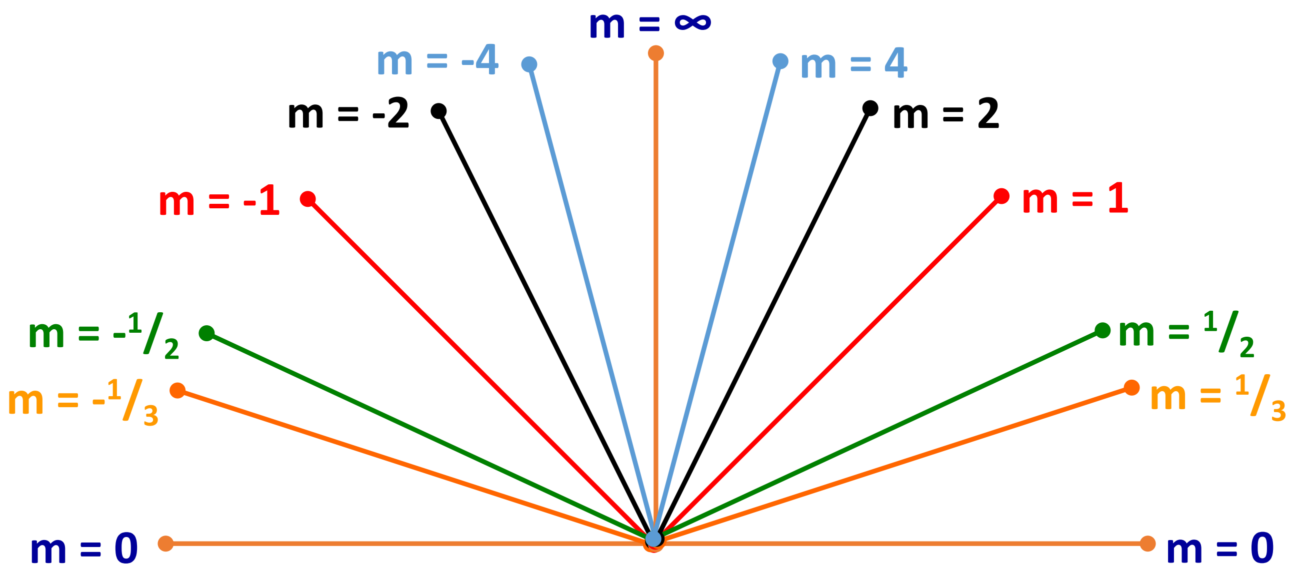
- Leaving out the details this gives us: $x(3) = 3\frac{2}{3} \approx 4$ $x(4) = 5\frac{1}{3} \approx 5$

- 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

- 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 DDA Algorithm

- 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:

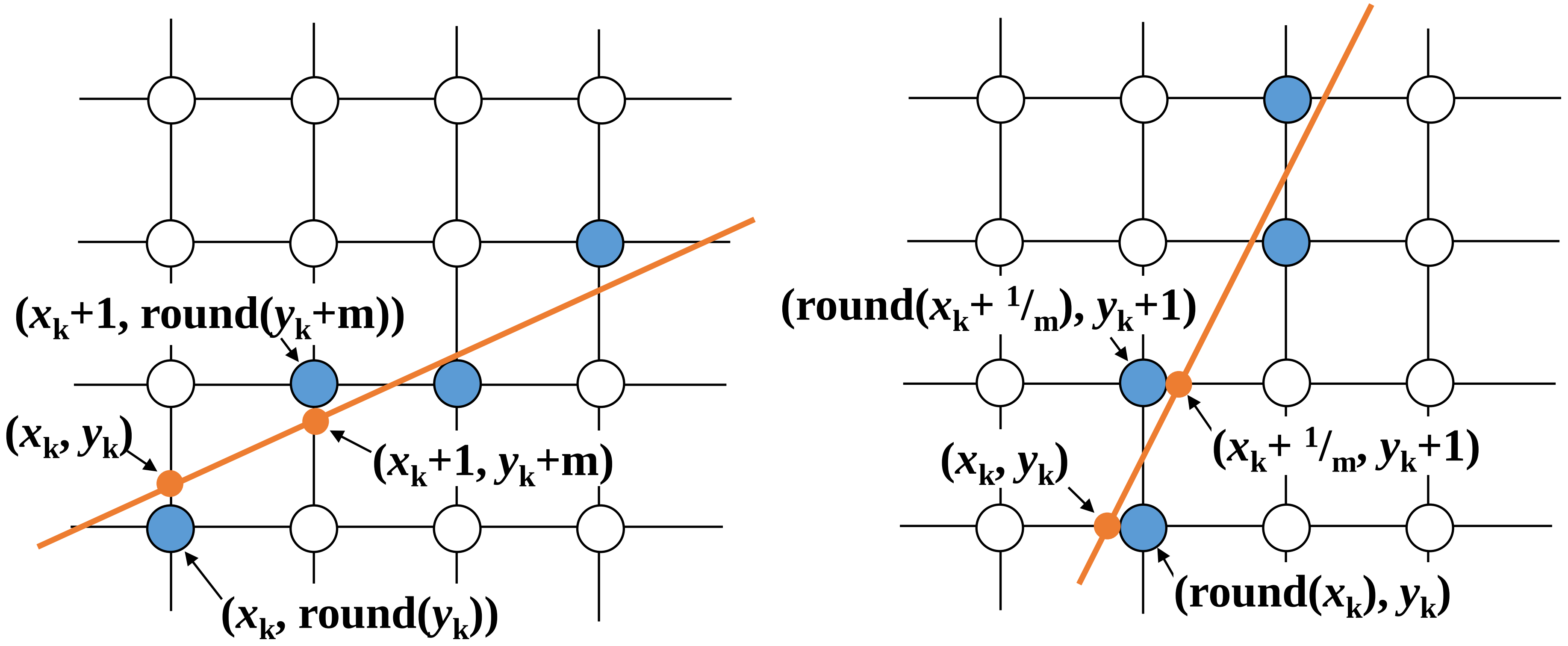
$$y_{k+1} = y_k + m$$

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

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

The DDA Algorithm

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



The DDA Algorithm

- 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

```
void Line(int x0, int y0, int x1, int y1) {
    int   x, y;
    float dy = y1 – y0;
    float dx = x1 – x0;
    float m  = dy / dx;

    y = y0;
    for (x = x0; x < x1; ++x) {
         WritePixel( x, Round(y) );
         y = y + m;
    }
}
```

Since slope is fractional, need special case for vertical lines (dx = 0)

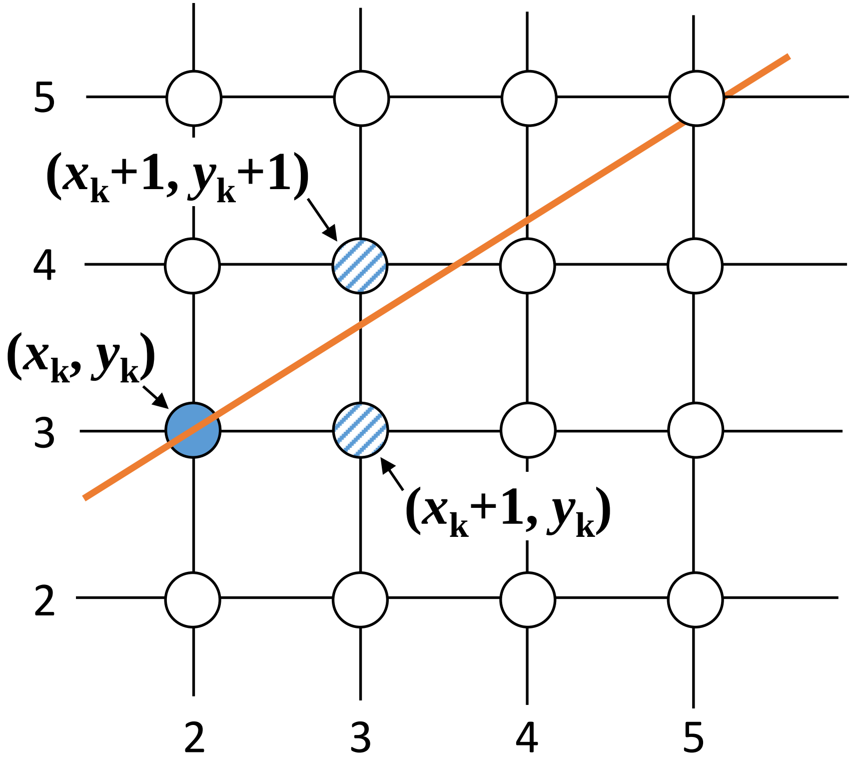
Rounding takes time

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

The Big Idea

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

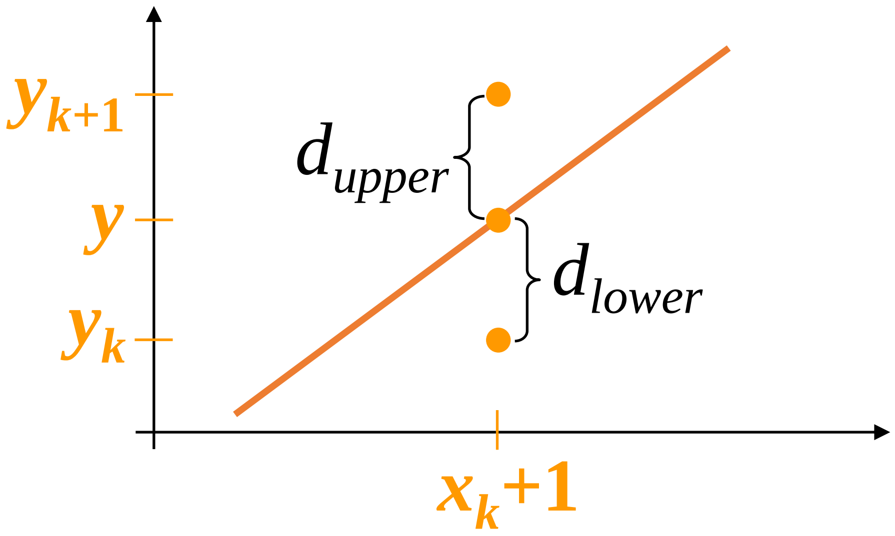


For example, from position (2, 3) we have to choose between (3, 3) and (3, 4)

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

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:

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

The Bresenham Line Algorithm

- 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

The Bresenham 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 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 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 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 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

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

Otherwise, the next point to plot is (x_k+1, y_k+1) and:

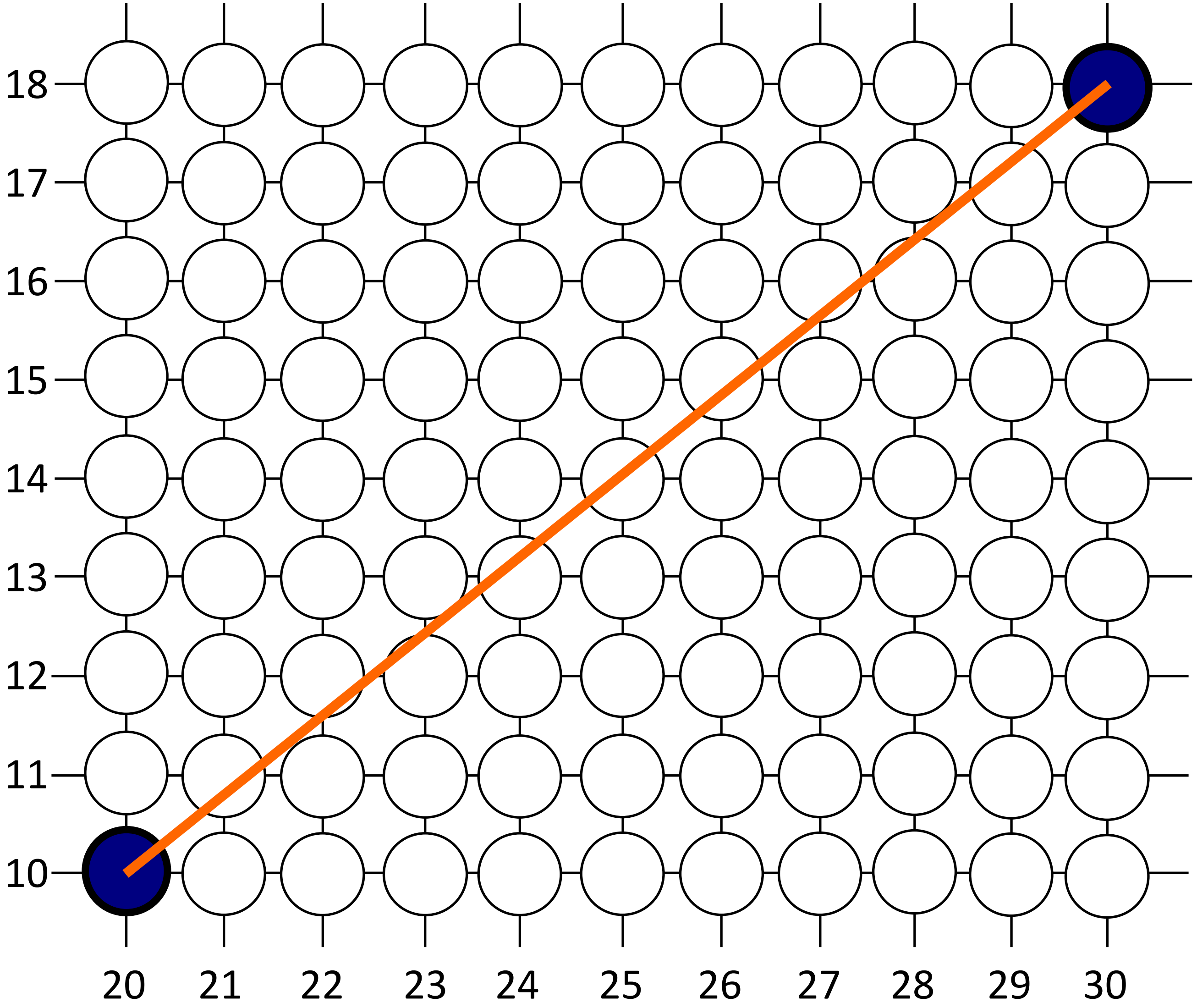
$$p_{k+1} = p_k + 2\Delta y - 2\Delta x$$

5. Repeat step 4 $(\Delta x - 1)$ times

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

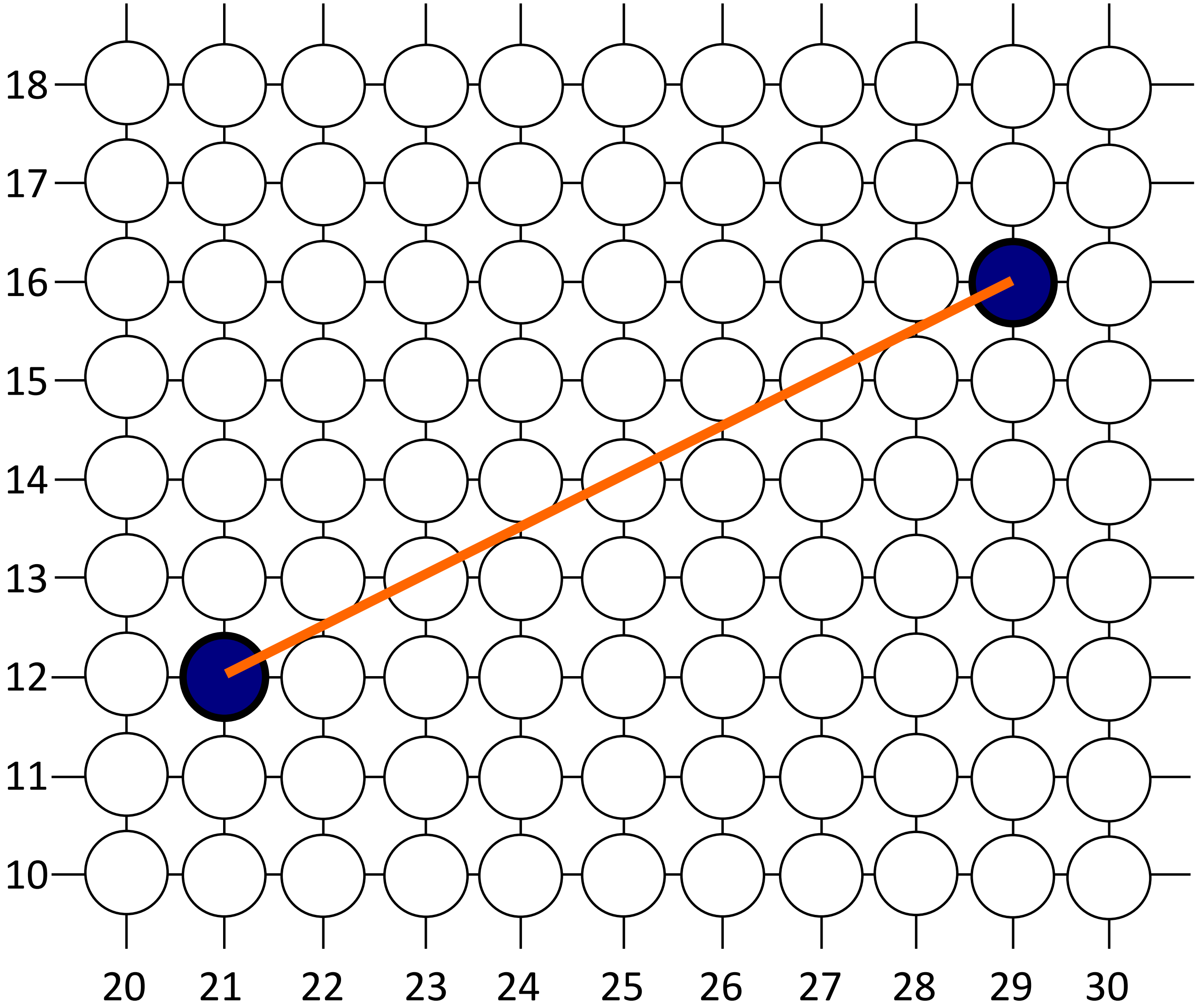


k	p_k	(x_{k+1},y_{k+1})
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		

Bresenham Example

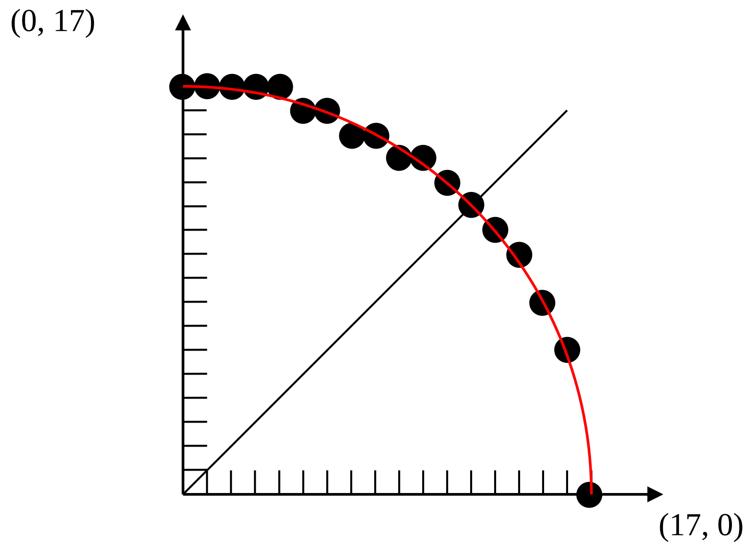
- Use the Bresenham algorithm for the line that starts and ends at points (21.12) and (29.16) respectively
-

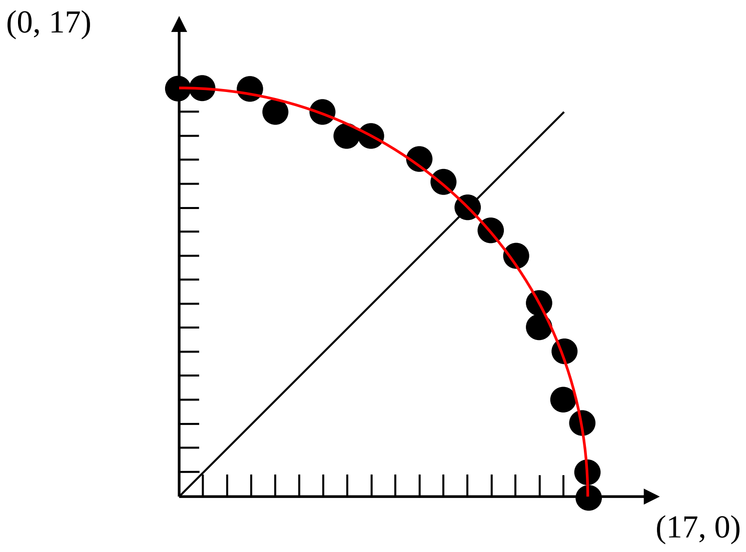
Παράδειγμα υλοποίησης του αλγόριθμου Bresenham



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

Circle design algorithms





A Simple Circle Drawing Algorithm

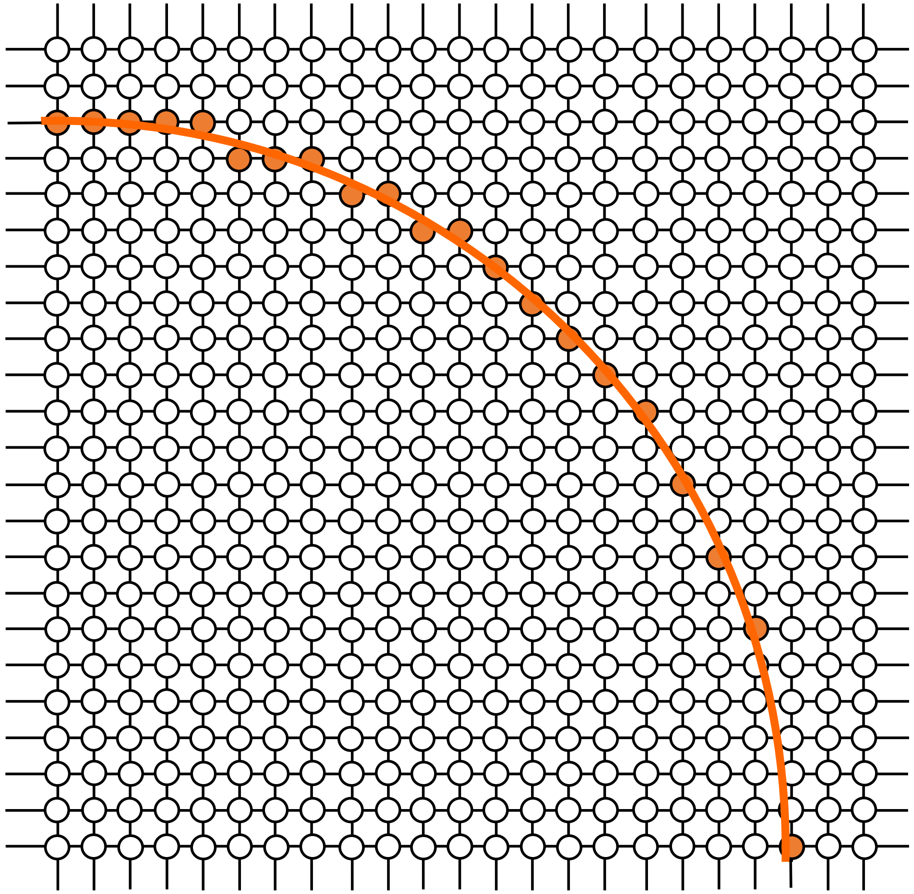
- The equation for a circle is:

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

- 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



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

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

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

$$\vdots$$

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

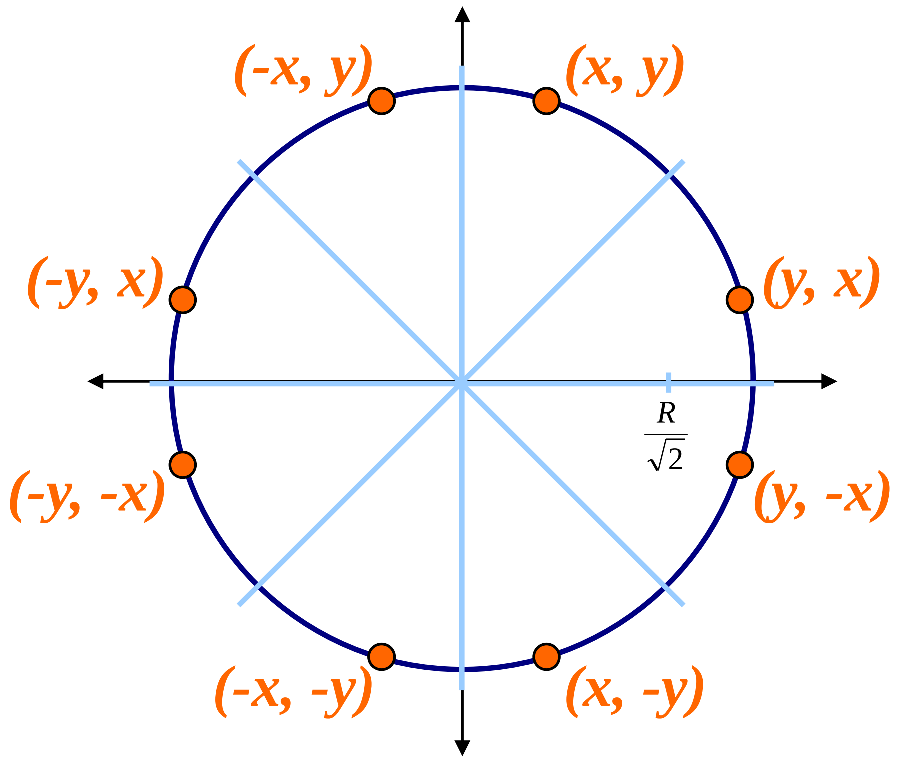
$$y_{20} = \sqrt{20^2 - 20^2} \approx 0$$

A Simple Circle Drawing Algorithm

- 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*

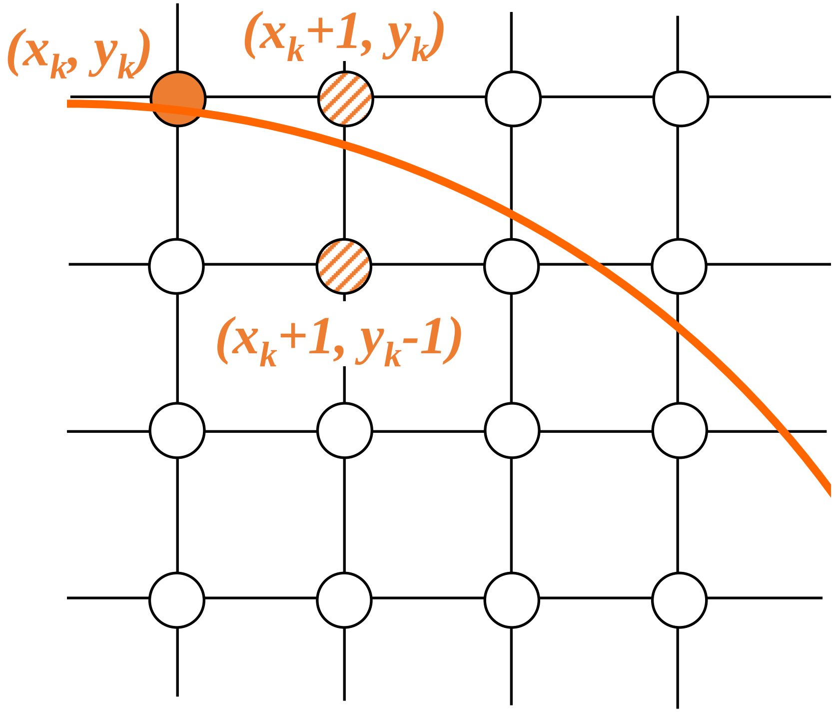


Mid-Point Circle Algorithm

- 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

Mid-Point Circle Algorithm

- 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?



Mid-Point Circle Algorithm

- 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

- 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 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

- 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

- 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:

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

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

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:

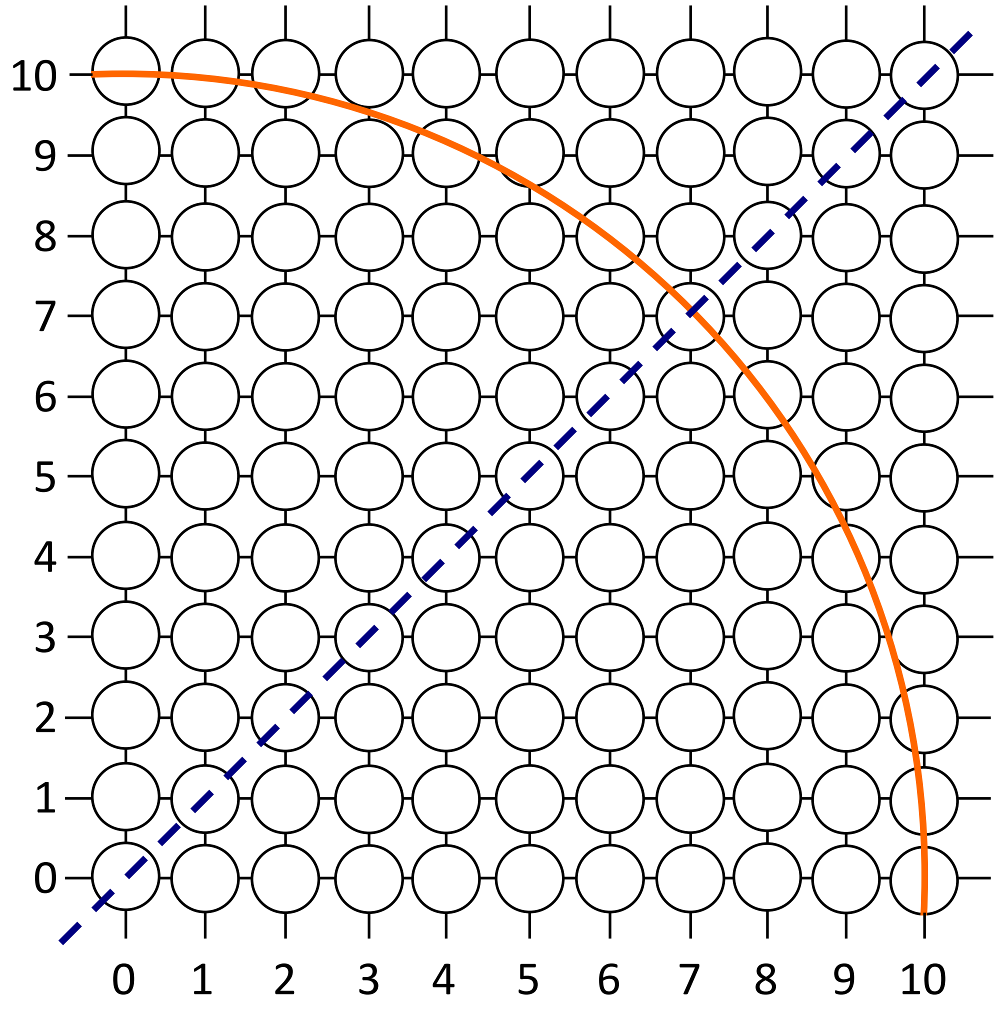
$$x = x + x_c \qquad y = y + y_c$$

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

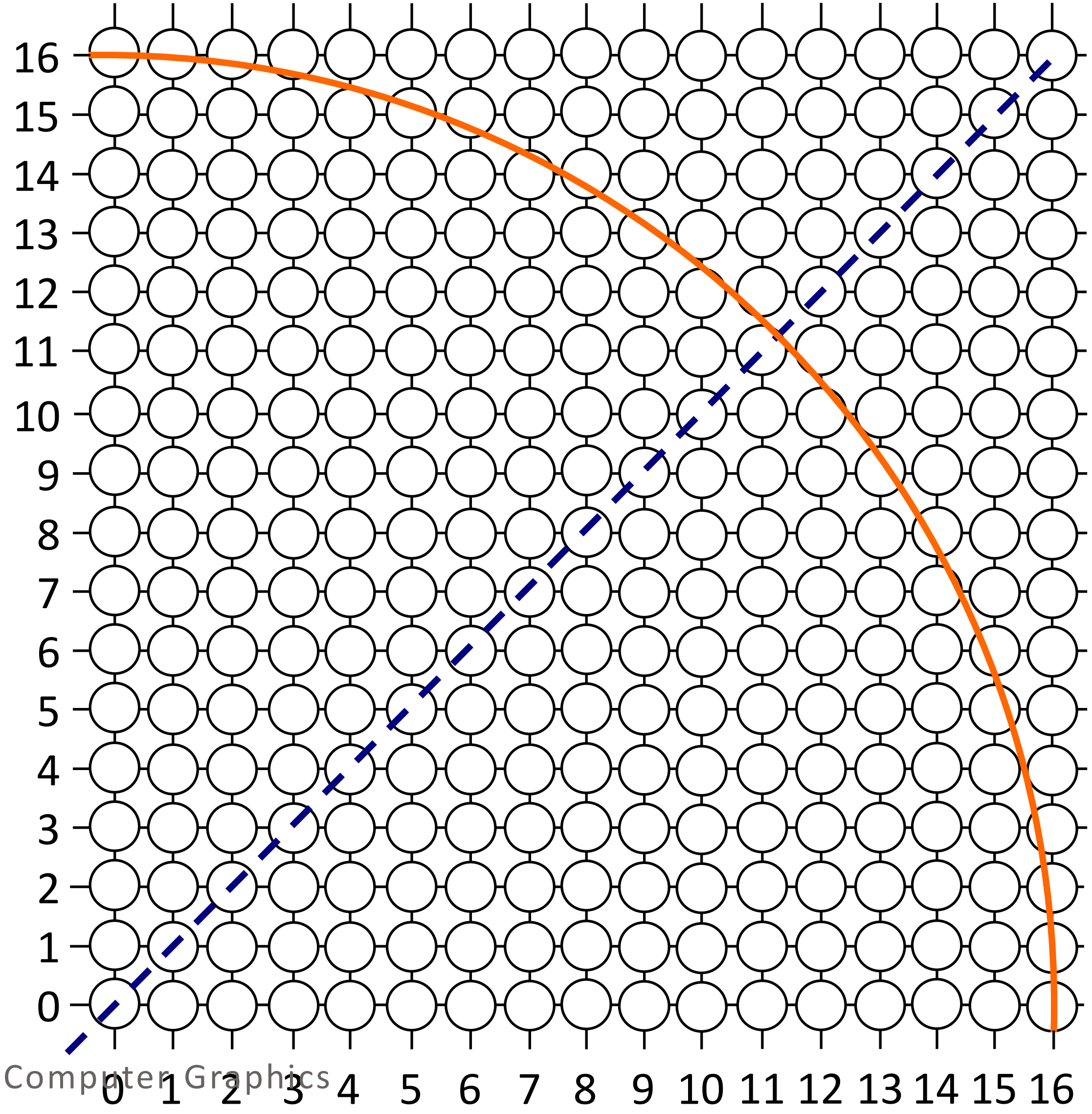


k	p_k	(x_{k+1},y_{k+1})	$2x_{k+1}$	$2y_{k+1}$
0				
1				
2				
3				
4				
5				
6				

Mid-Point Circle Algorithm Exercise

- Use the mid-point circle algorithm to draw the circle centred at (0,0) with radius 15

Mid-Point Circle Algorithm Exercise



k	p_k	(x_{k+1},y_{k+1})	$2x_{k+1}$	$2y_{k+1}$
0				
1				
2				
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				

Mid-Point Circle Algorithm Summary

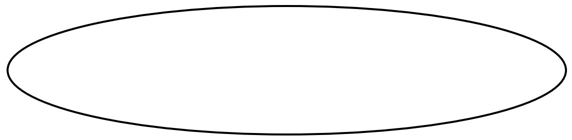
- The key insights in the mid-point circle algorithm are:
  - Eight-way symmetry can hugely reduce the work in drawing a circle
  - Moving in unit steps along the x axis at each point along the circle's edge we need to choose between two possible y coordinates

Midpoint Eighth Circle Algorithm

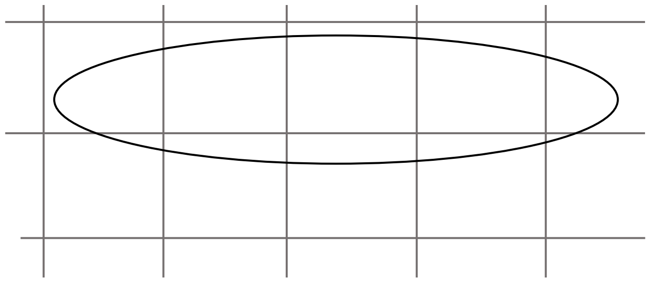
```
MidpointEighthCircle(R) { /* 1/8th of a circle w/ radius R */
    int x = 0, y = R;
    int deltaE   = 2 * x + 3;
    int deltaSE  = 2 * (x - y) + 5;
    float decision = (x + 1) * (x + 1) + (y - 0.5) * (y - 0.5) – R*R;
    WritePixel(x, y);

    while ( y > x ) {
        if (decision > 0) { // Move East
            x++; WritePixel(x, y);
            decision += deltaE;
            deltaE += 2; deltaSE += 2; // Update deltas
        } else { // Move SouthEast
            y--; x++; WritePixel(x, y);
            decision += deltaSE;
            deltaE += 2; deltaSE += 4; // Update deltas
        }
    }
}
```

Other Scan-conversion Problems

- Aligned Ellipses

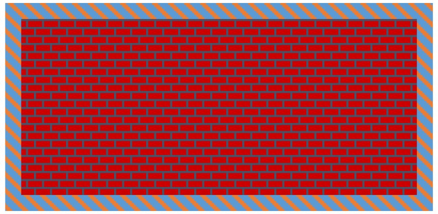


- Non-integer primitives

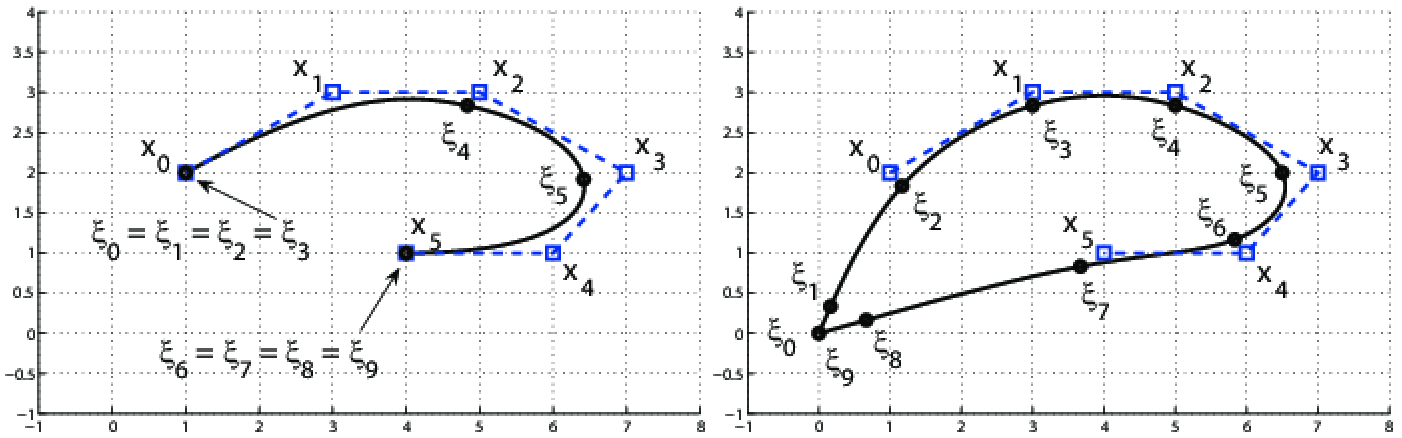


- General conics

- Patterned primitives

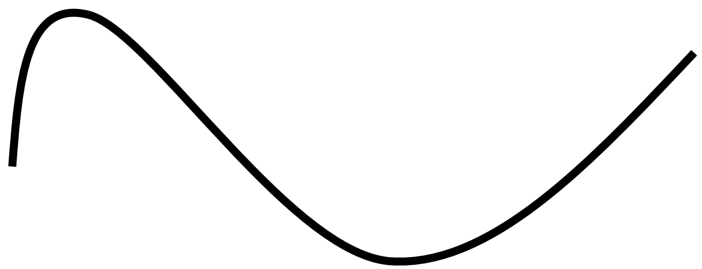


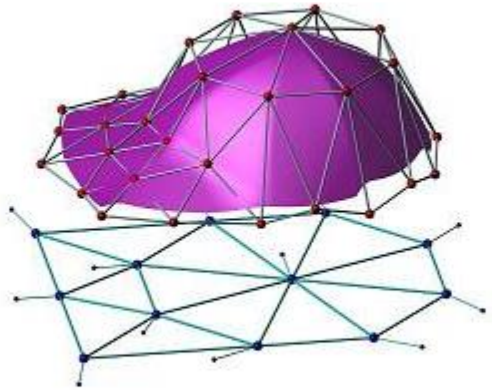
Spline Representations



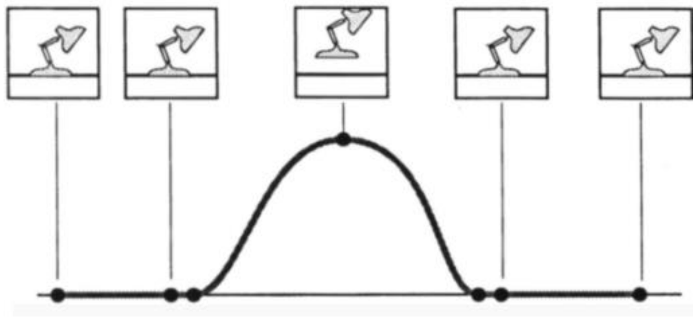
Spline Representations

- A spline is mathematically defined by using a set of constraints
- Curves have many uses:
  - 2D illustration
  - Fonts
  - 3D Modelling
  - Animation





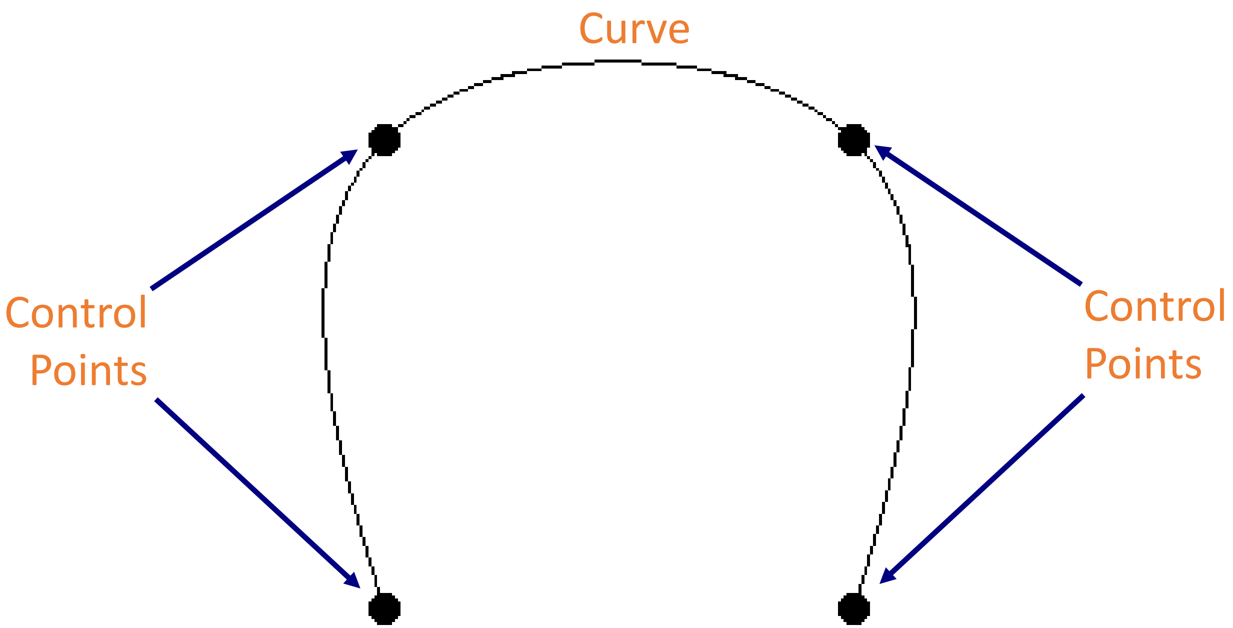
"Manifold Splines", X. Gu, Y. He & H. Qin, Solid and Physics Modeling 2005.



ACM © 1987 "Principles of traditional animation applied to 3D computer animation"

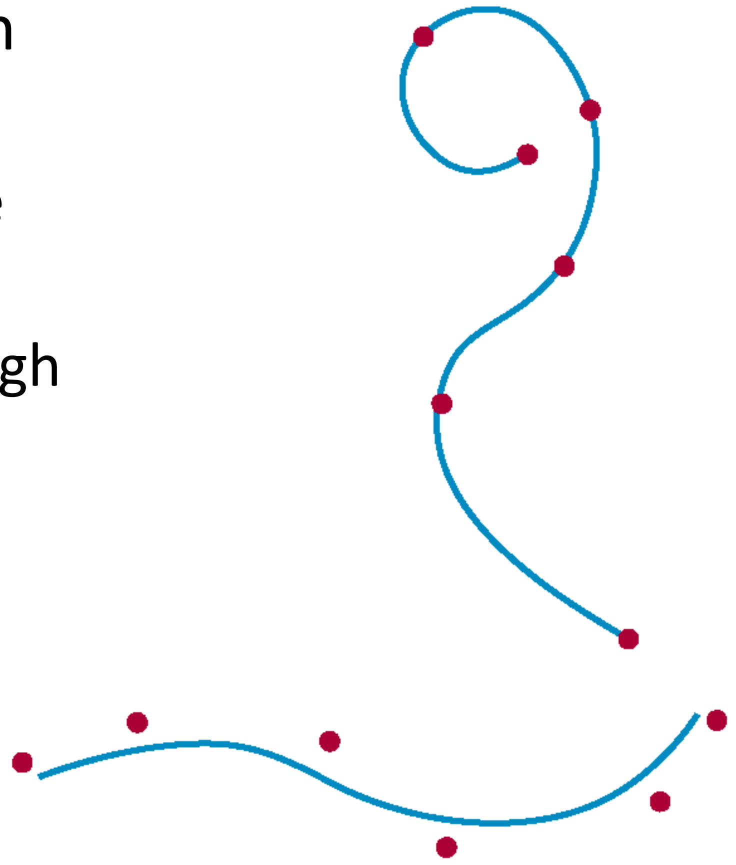
The basic idea

- The user specifies the control points
- A smooth curve is defined
-



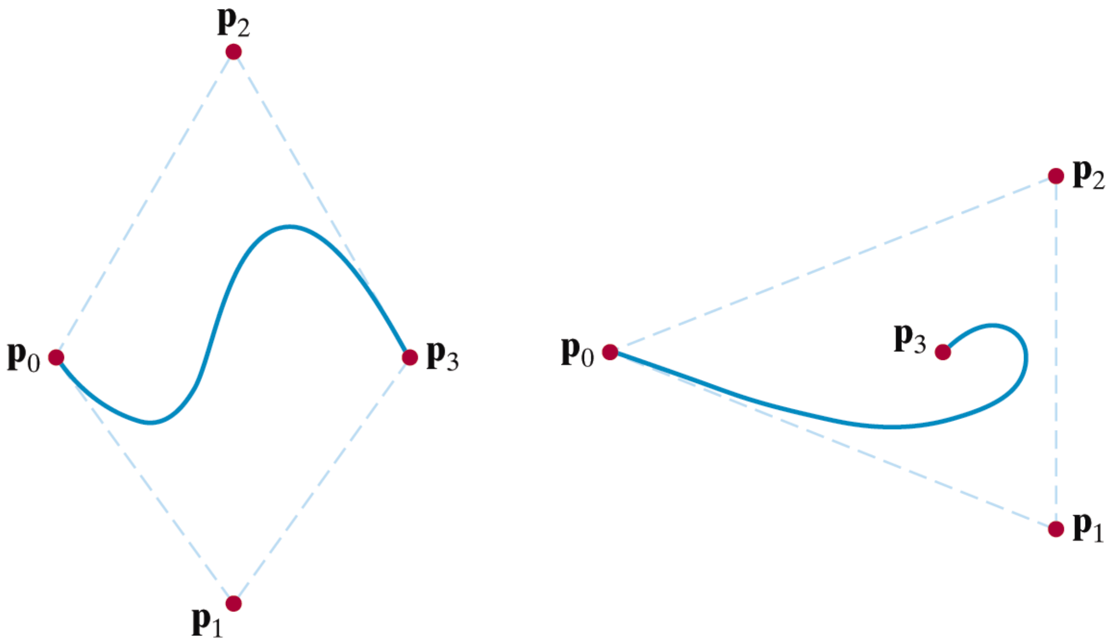
Interpolation Vs Approximation

- The curve is defined by a set of **control points**
- There are 2 ways to define the curve based on these points
  - **Interpolation -** the curve passes through all the control points
  - **Approximation -** the curve does not pass through all control points



Convex Hulls

- The boundary formed by the set of control points for a curve are known as **convex hull**



Bézier Spline Curves

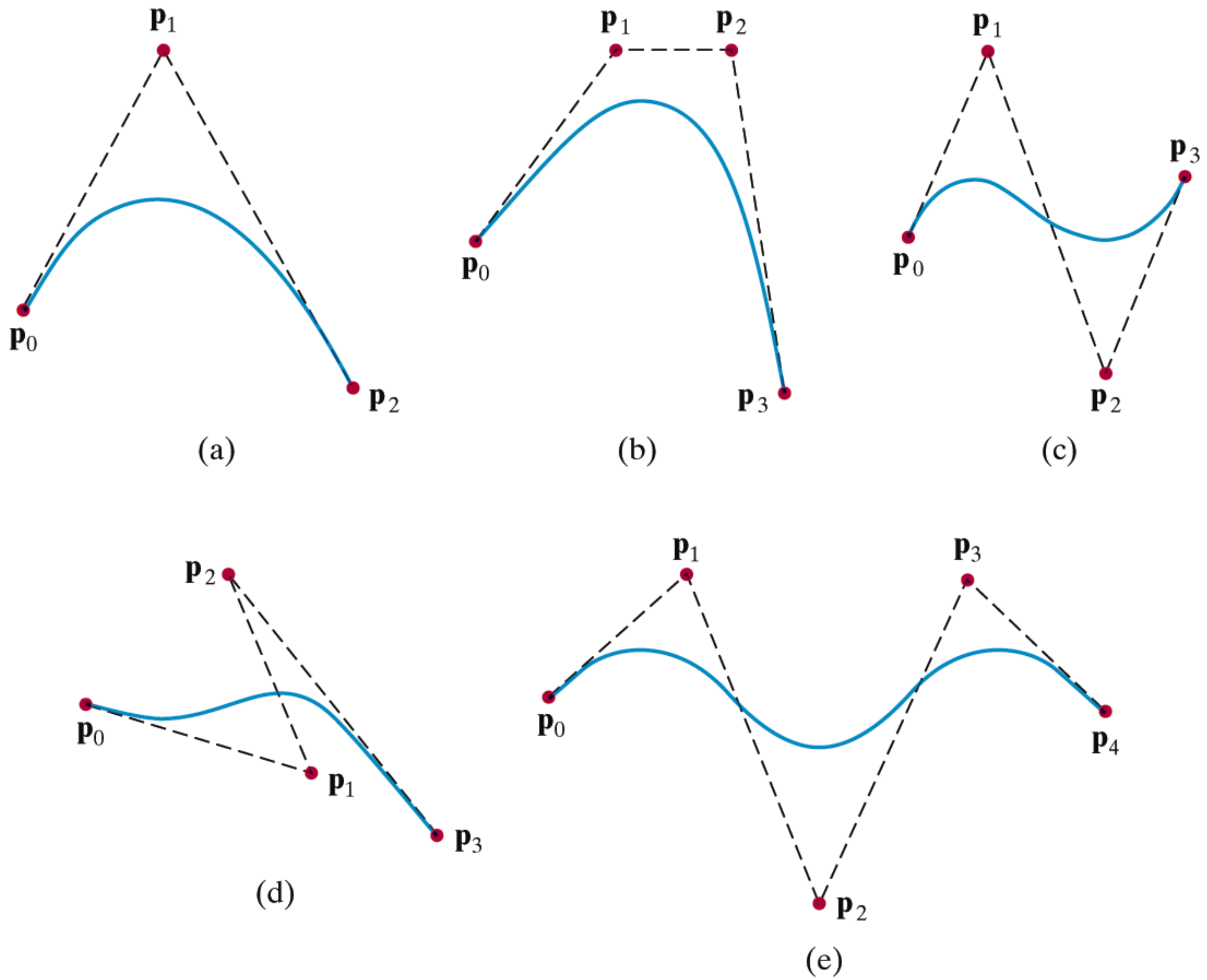
- The most famous method is the one implemented by the engineer Pierre Bézier for the design of Renault cars
- A Bézier curve can be applied to any number of points, although 4 are usually used
- Let's $n+1$ points $p_k=(x_k, y_k, z_k)$ where k is between 0 and n
- The coordinates of the path of the curve from the vector p_0 to p_n is given by the equation
-

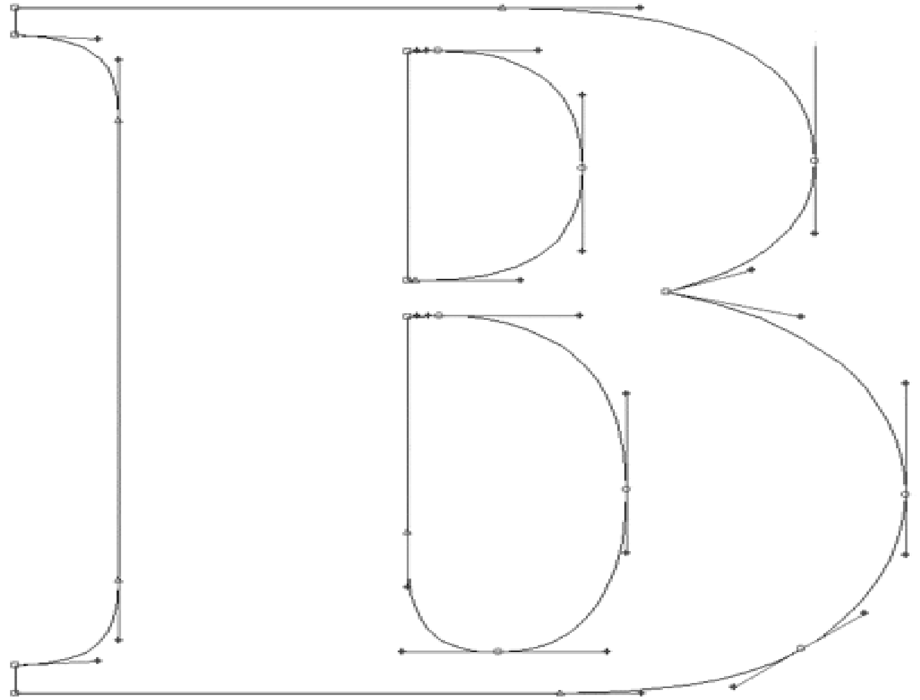
$$P(u) = \sum_{k=0}^{n} p_k BEZ_{k,n}(u), \qquad 0 \le u \le 1$$

$$BEZ_{k,n}(u) = C(n,k)u^k(1-u)^{n-k}$$

$$C(n,k) = \frac{n!}{k!(n-k)!}$$ binomial coefficients

Bézier Spline Curves



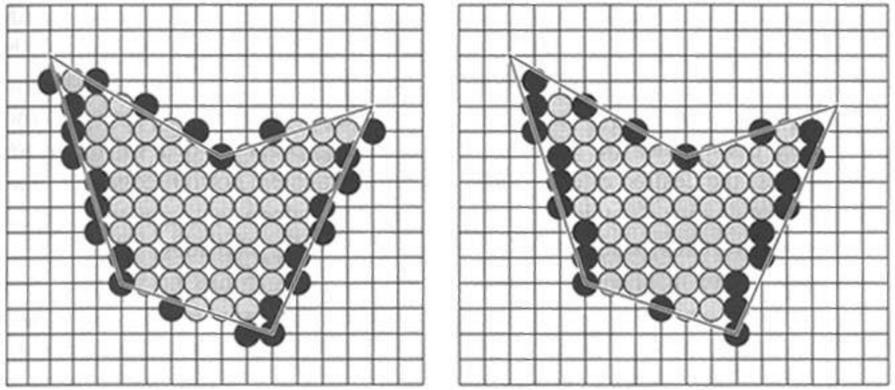


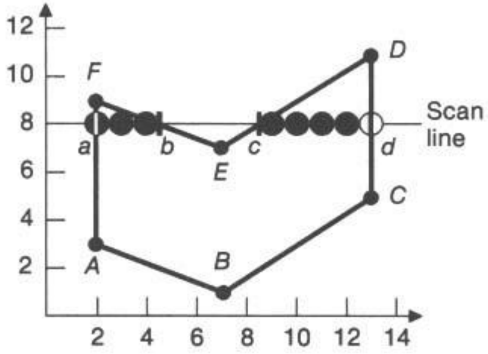
Bezier splines are widely used (Adobe, Microsoft) for font definition

Bézier Spline Curves

- Why in graphics we do not prefer the use of curves, either from simple shapes (circle), or from complex shapes (Bezier curves)?
-

Polygons





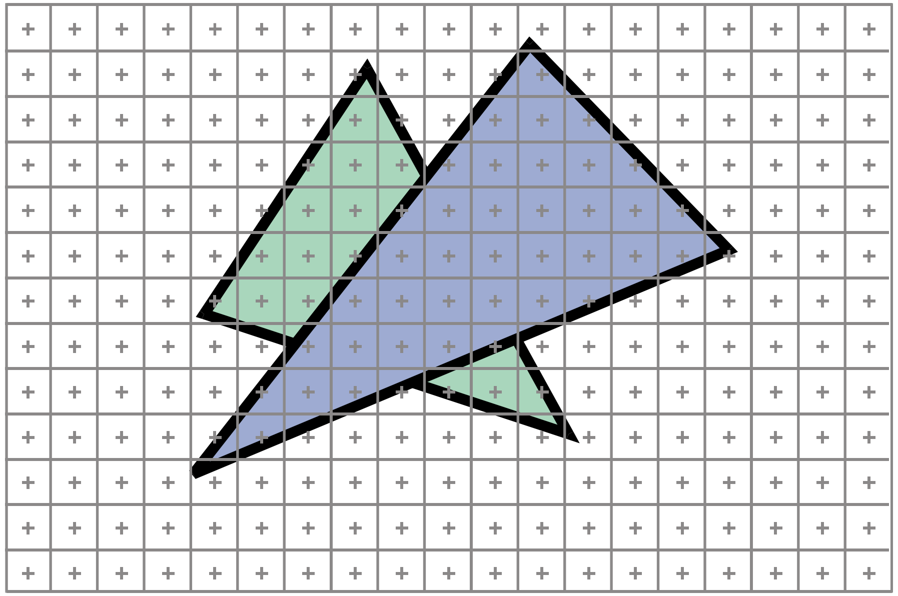
Scan-Line Polygon Fill Algorithm

- 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**

> **Rasterisation** (or **rasterization**) is the task of taking an image described in a vector graphics format (shapes) and converting it into a raster image (pixels or dots)

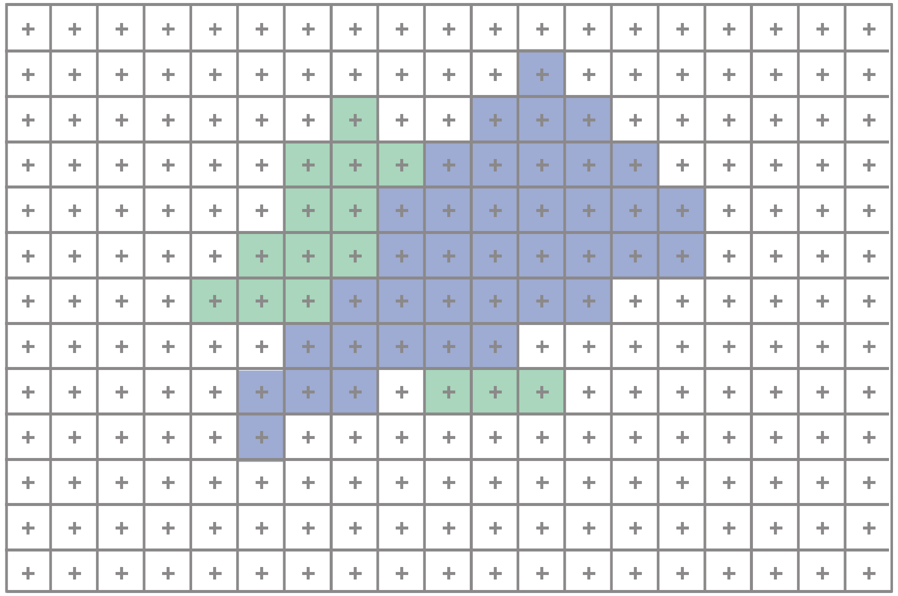
2D Scan Conversion

- Primitives are continuous – the screen is discrete
-



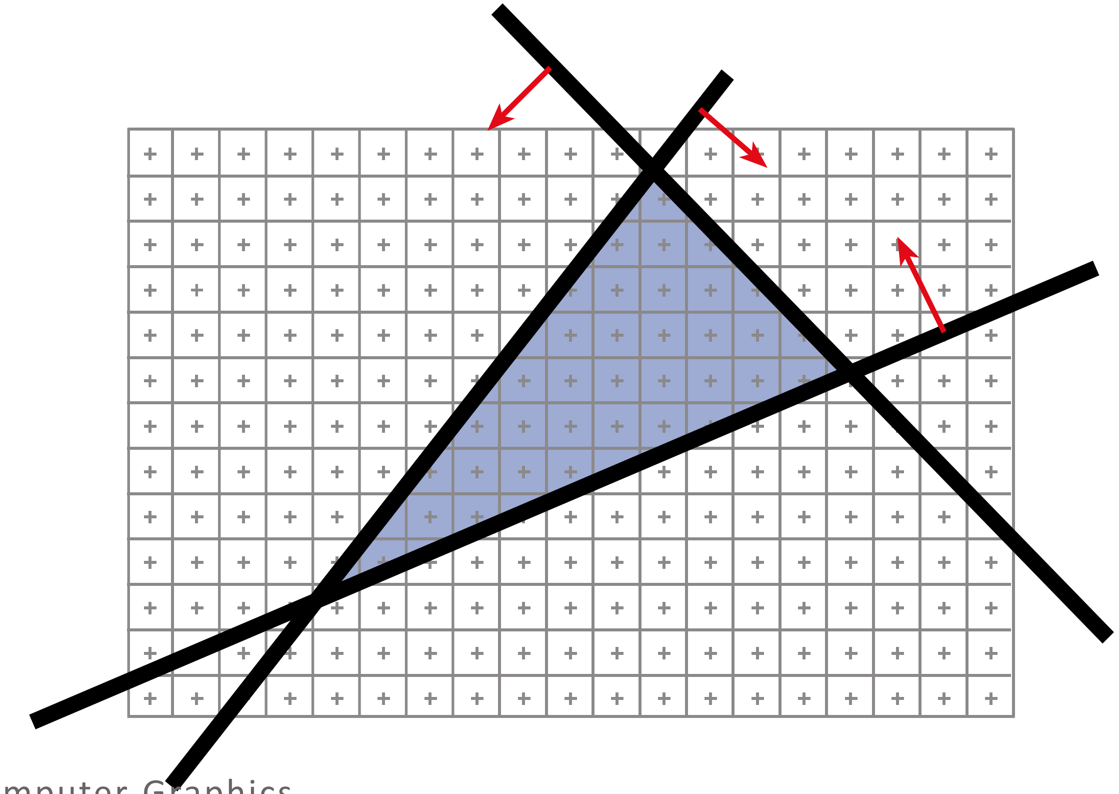
2D Scan Conversion

- Solution: calculate discretely with approximation
- Scanning: the algorithms for efficient sample creation include this approach
-



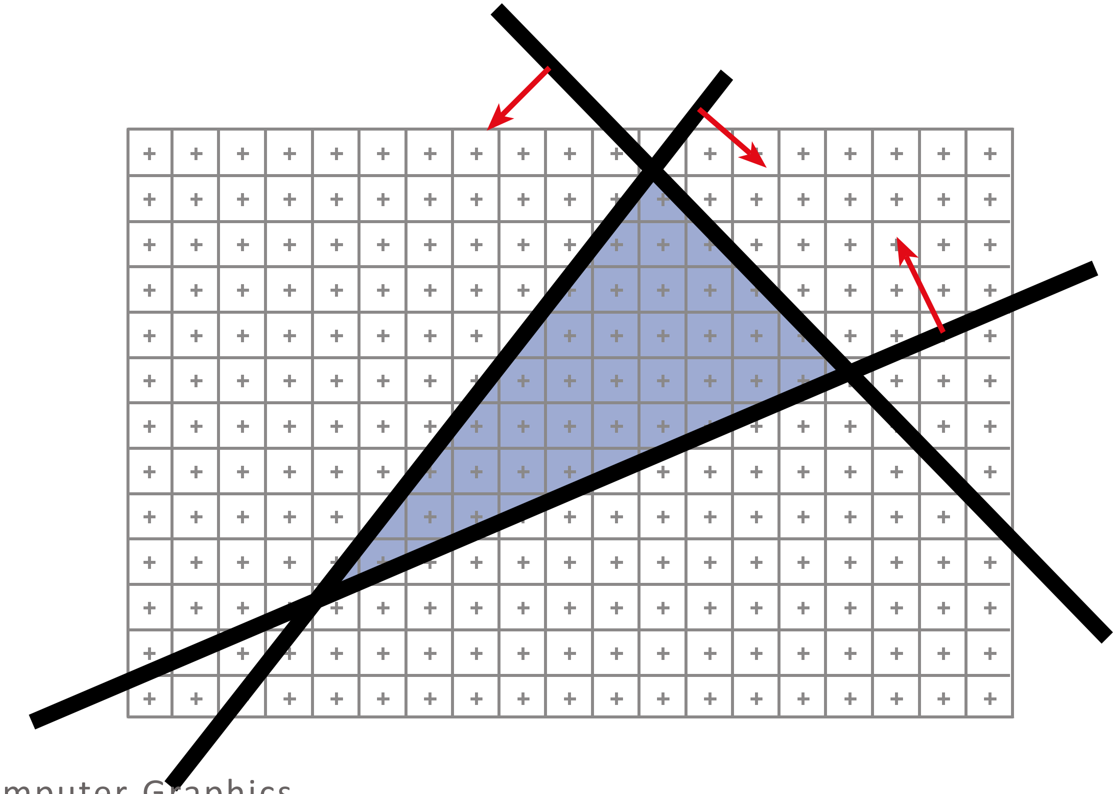
Brute force solution for triangles

- ?



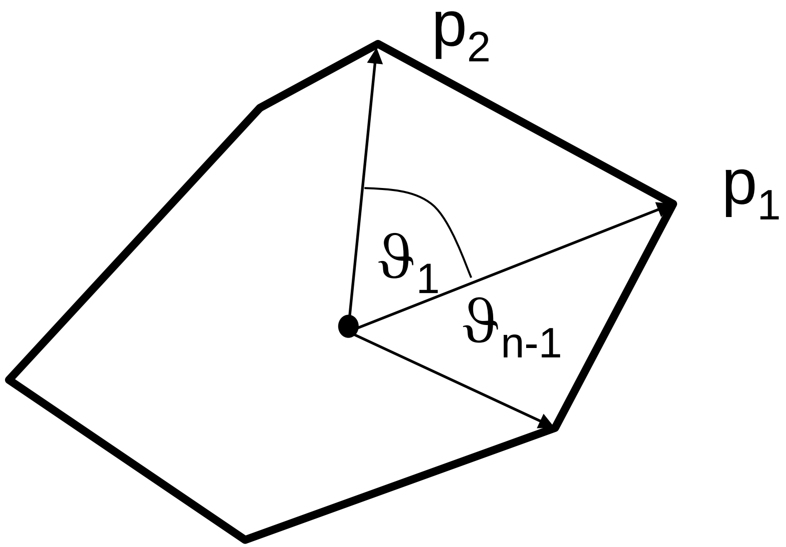
Brute force solution για τρίγωνα

- For each pixel
  - We look if it's inside the triangle



Why triangles?

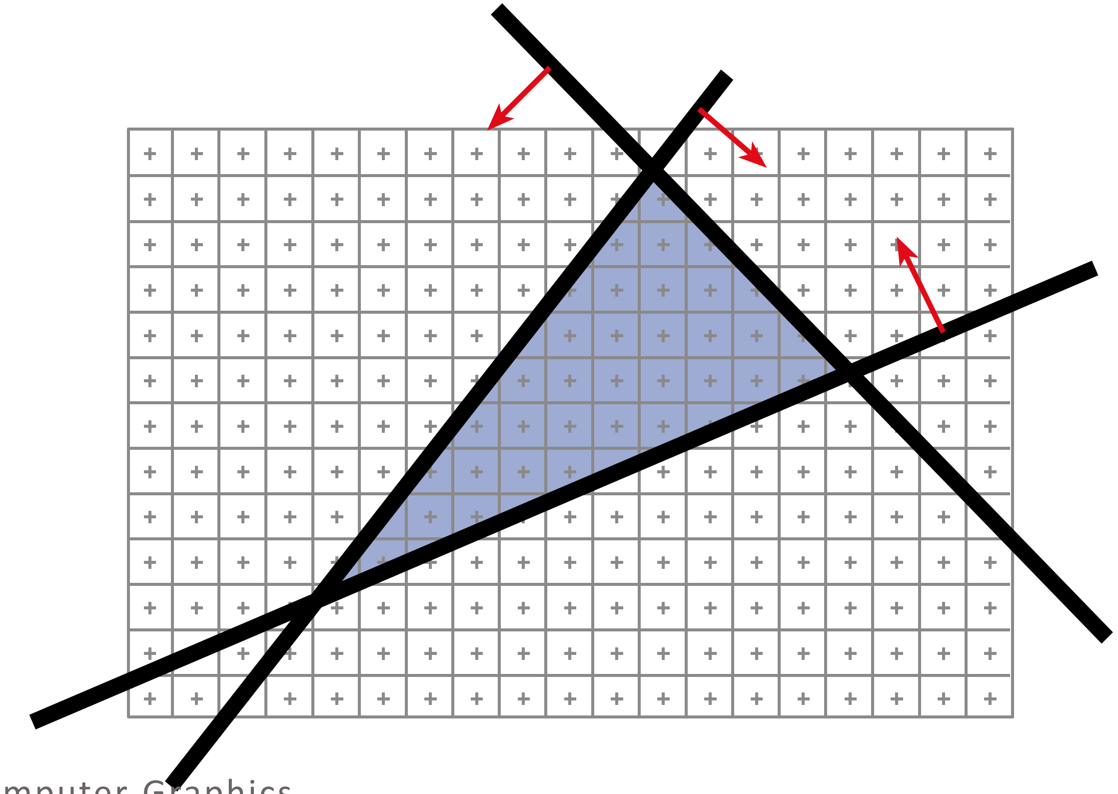
- Point on a polygon will give us triangles
-



$$\sum_{i=1}^{n} \vartheta_i = 2\Pi$$

Brute force solution for triangles

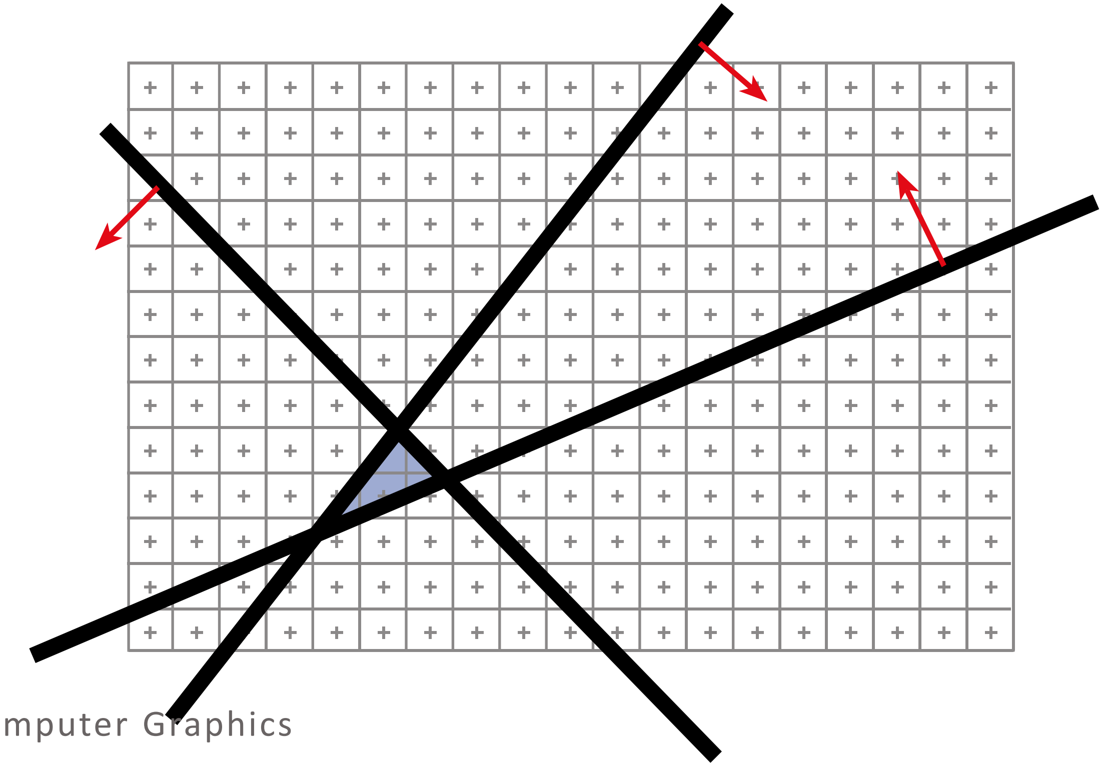
- For each pixel
  - We look if it's inside the triangle



Problem?

Brute force solution for triangles

- For each pixel
  - We look if it's inside the triangle

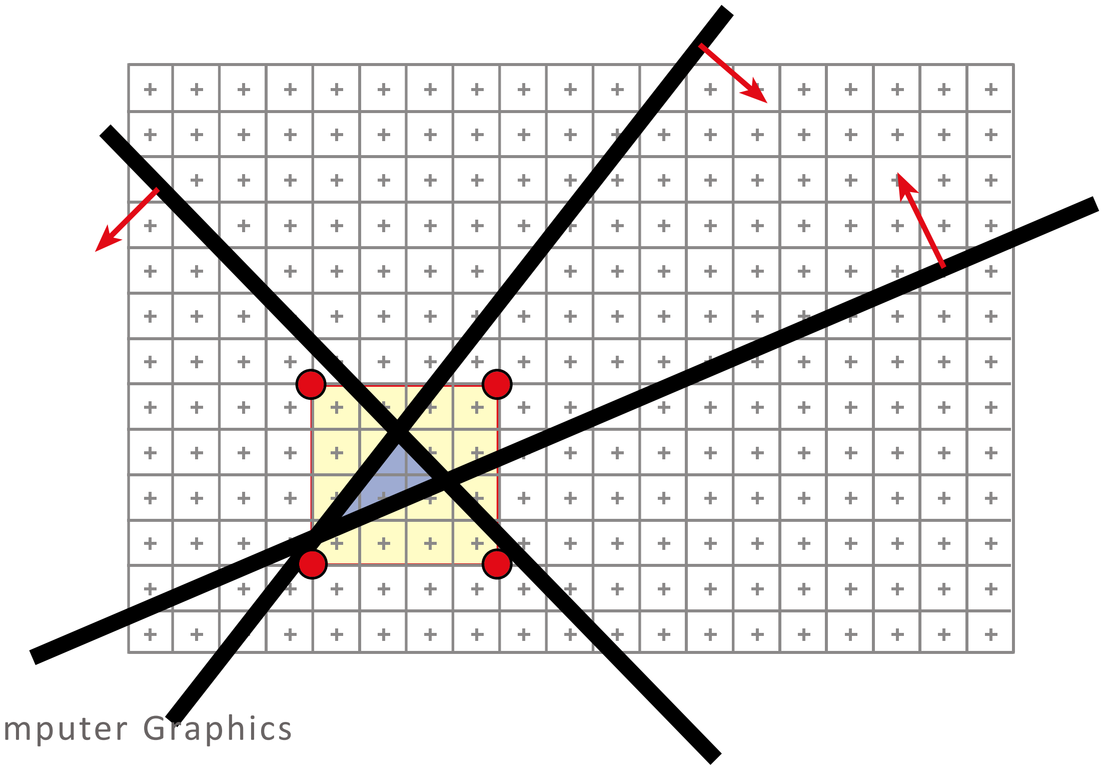


Problem?

If the triangle is small, we do many unneeded calculations

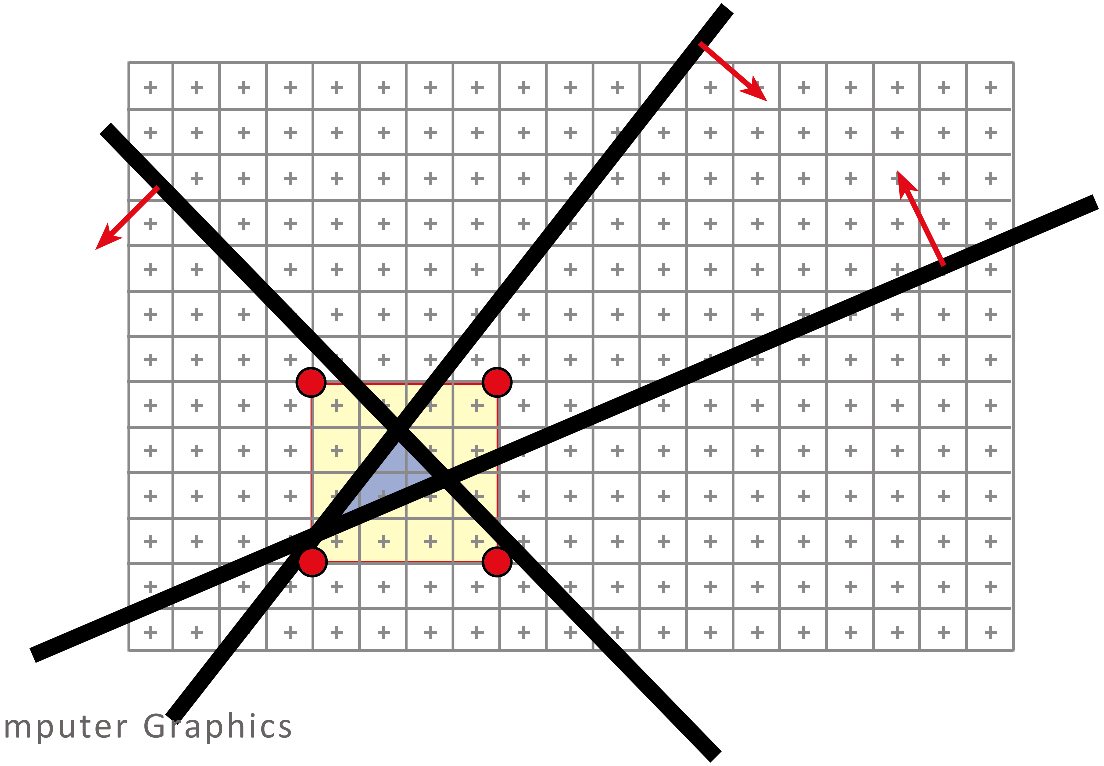
Brute force solution for triangles

- Optimization:
  - We only look at the pixels that are inside the bounding box of the triangle
  - How do we find the bounding box?



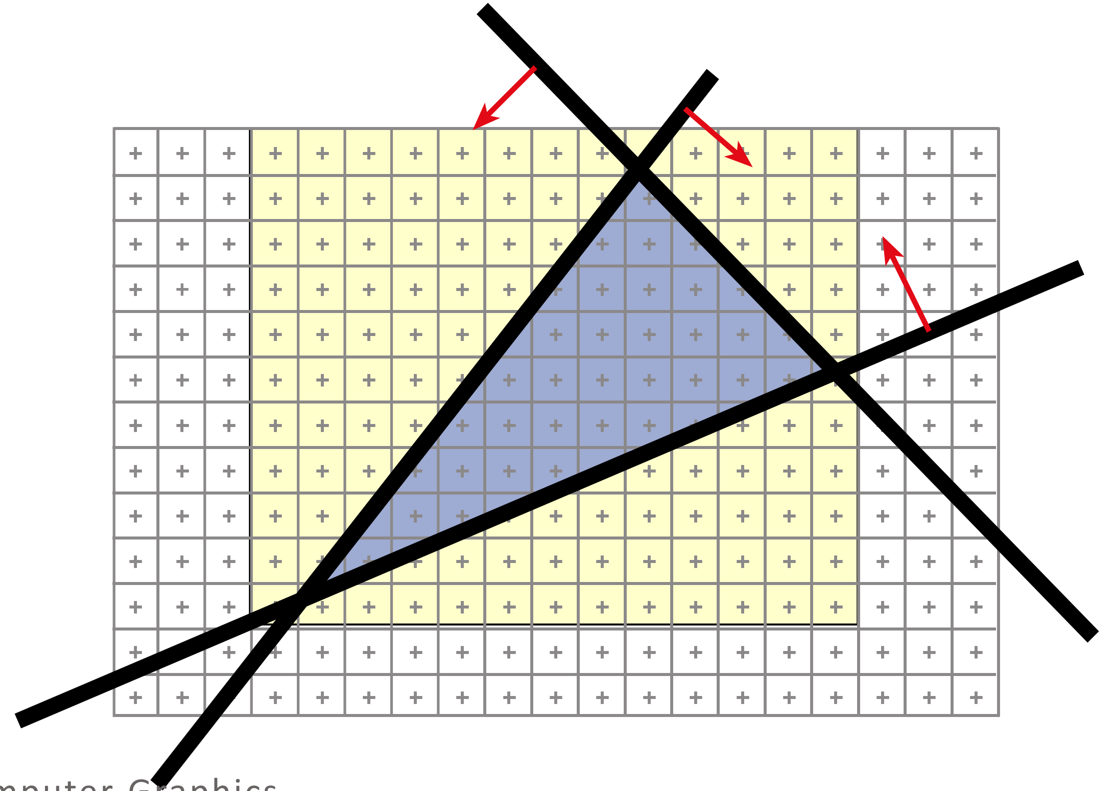
Brute force solution for triangles

- Optimization:
  - We only look at the pixels that are inside the bounding box of the triangle
  - with the Xmin, Xmax, Ymin, Ymax of its edges



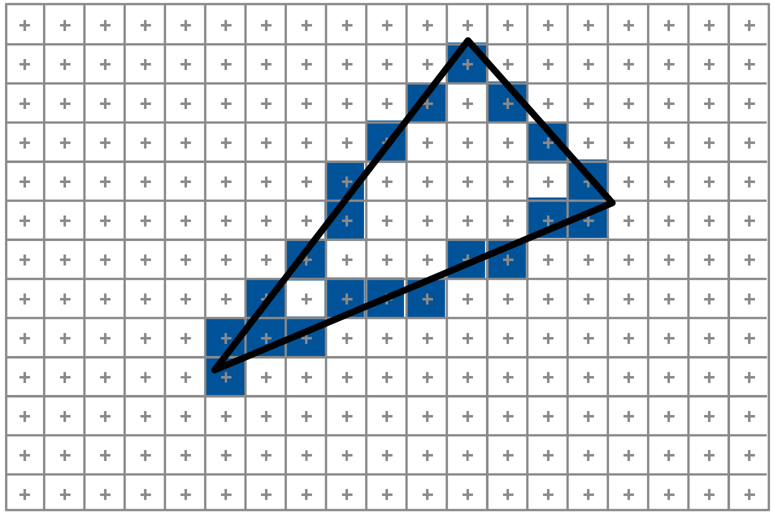
Can we do better?

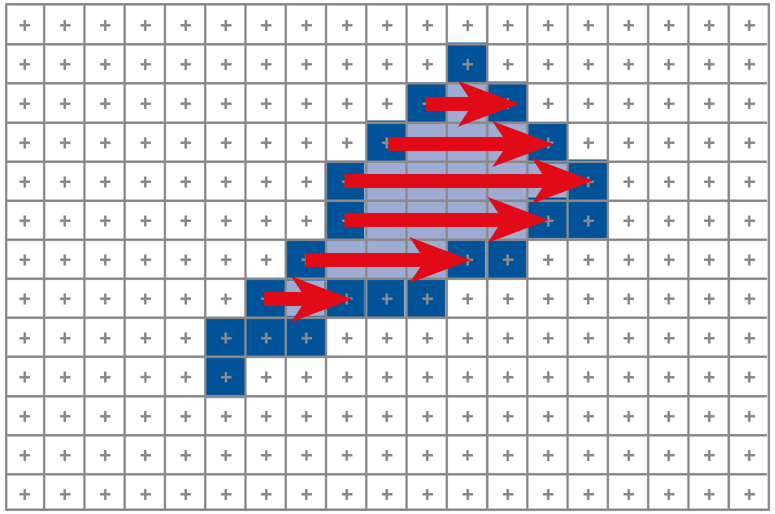
- If the triangles are large, again we have many unnecessary calculations
- What can we do?



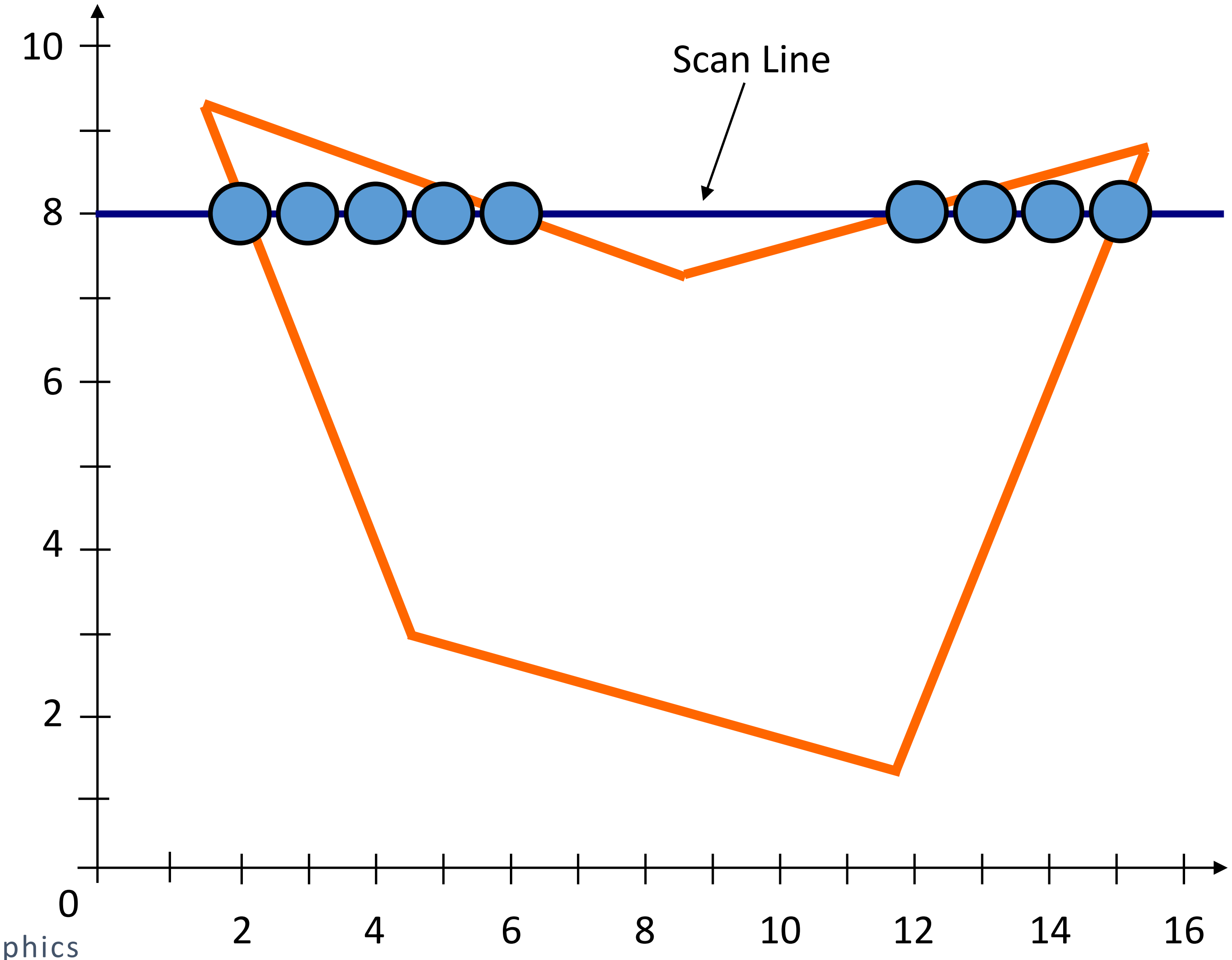
Scan-Line Polygon Fill Algorithm

- We use line rasterization
  - 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

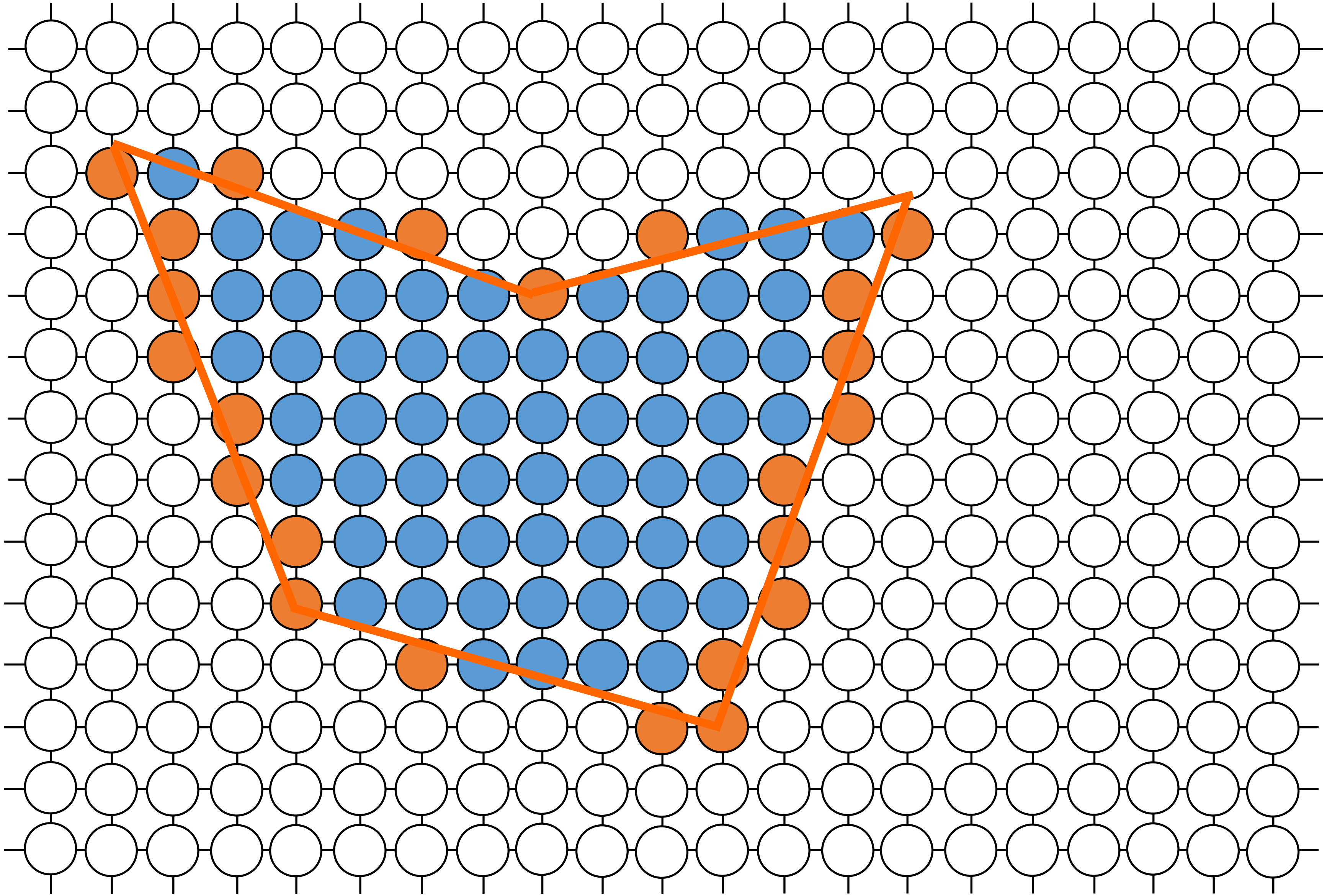




Scan-Line Polygon Fill Algorithm



Scan-Line Polygon Fill Algorithm

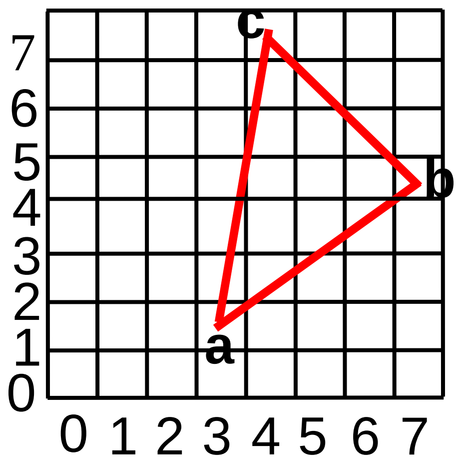


Line Drawing Summary

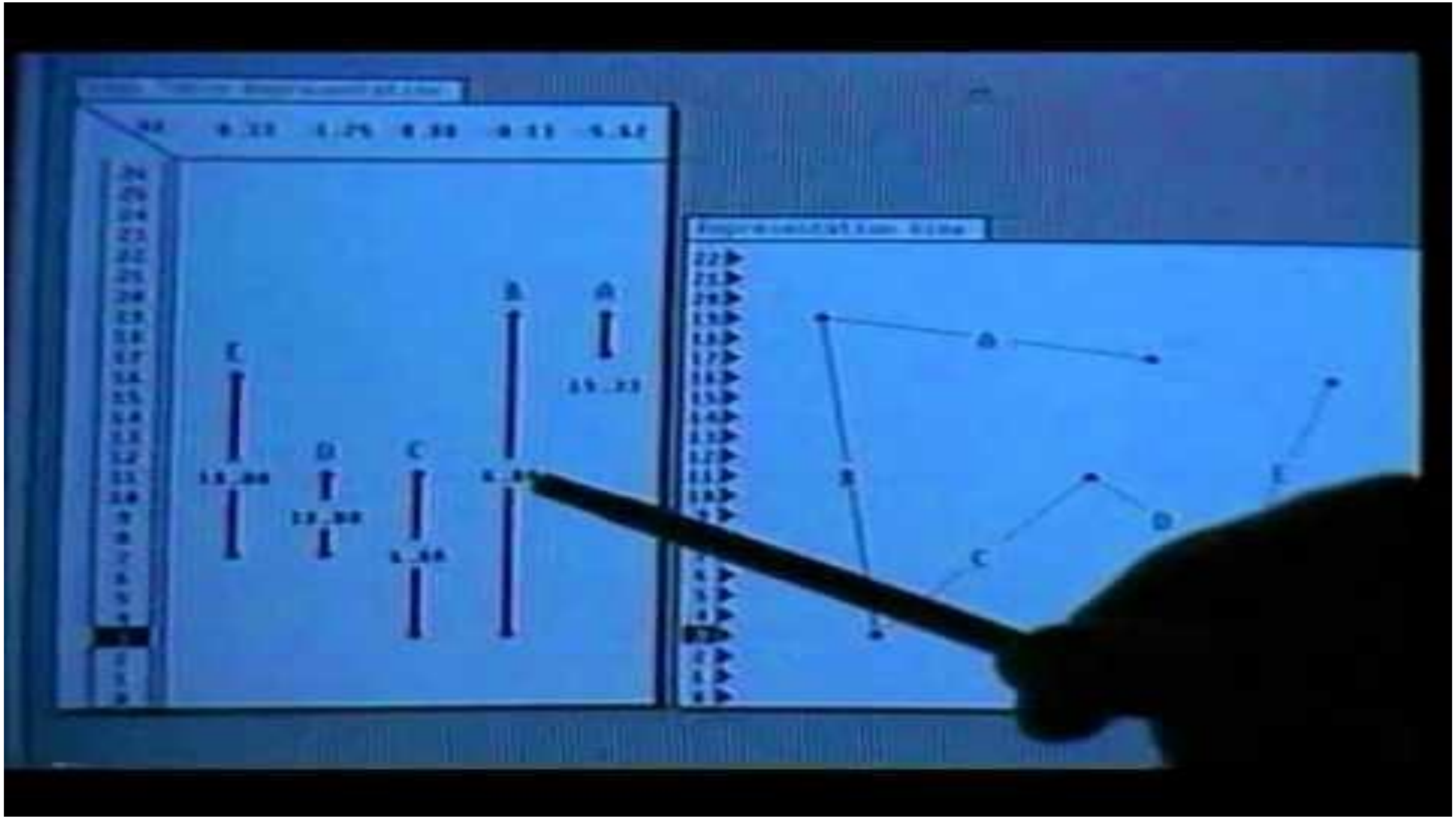
- Over the last couple of lectures we have looked at the idea of scan converting lines
- The key thing to remember is this has to be **FAST**
- For lines we have either DDA or Bresenham
- For circles the mid-point algorithm

Triangle Scan

```
void scanTriangle(Triangle T, Color rgba) {
        for each edge
                compute (y2, x1, dx/dy)
        for each scanline at y
                for the current edge pair (L, R) {
                        for (int x = xL; x <= xR; x++)
                                SetPixel(x, y, rgba);
                xL += dxL/dyL;
                xR += dxR/dyR;
        }
}
```



Demo



Demo: https://youtu.be/GXi32vnA-2A