Computer Graphics - **Rasterization**

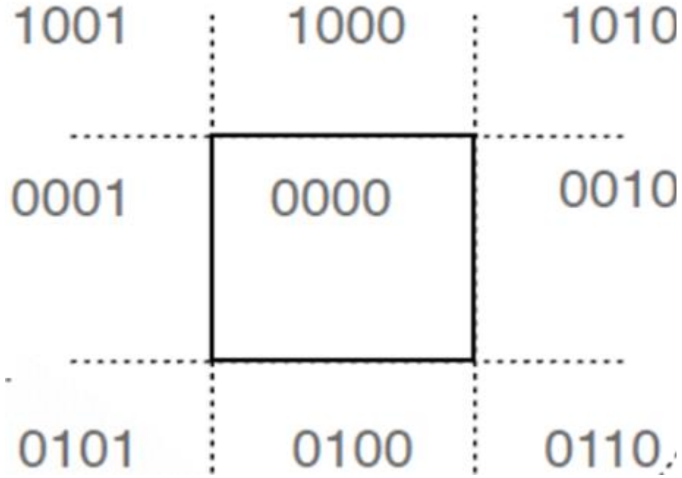
Junjie Cao @ DLUT

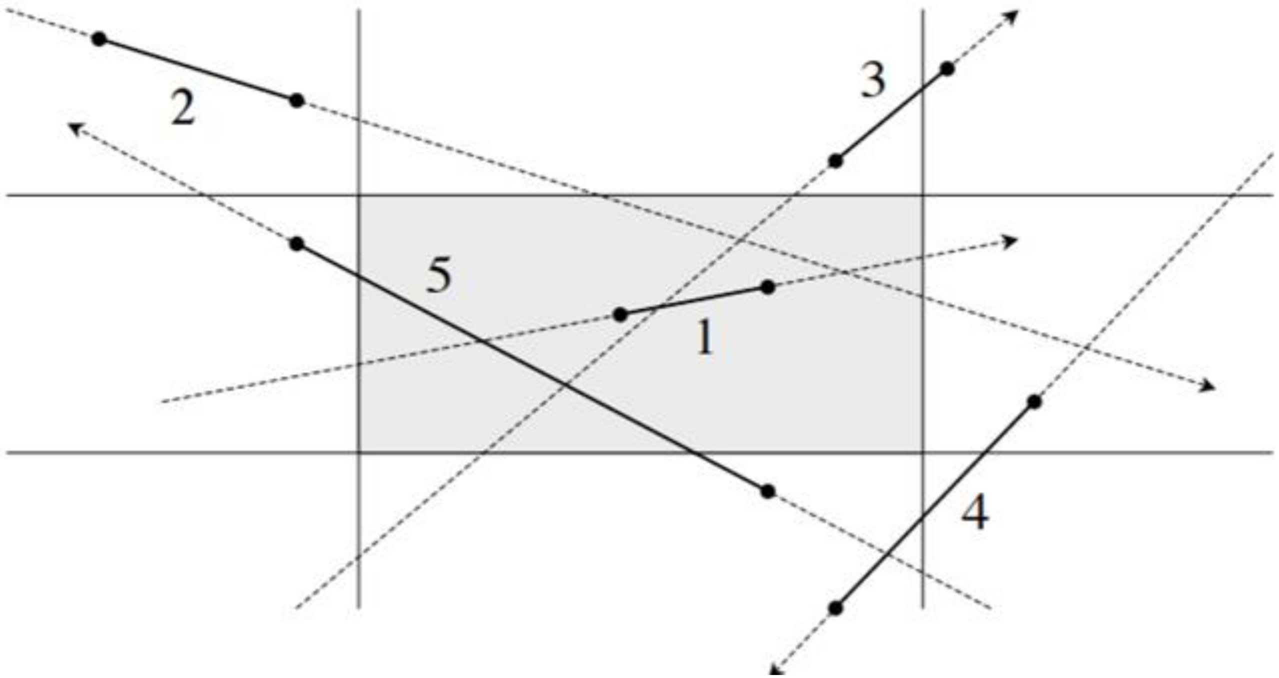
Spring 2018

http://jjcao.github.io/ComputerGraphics/

Review

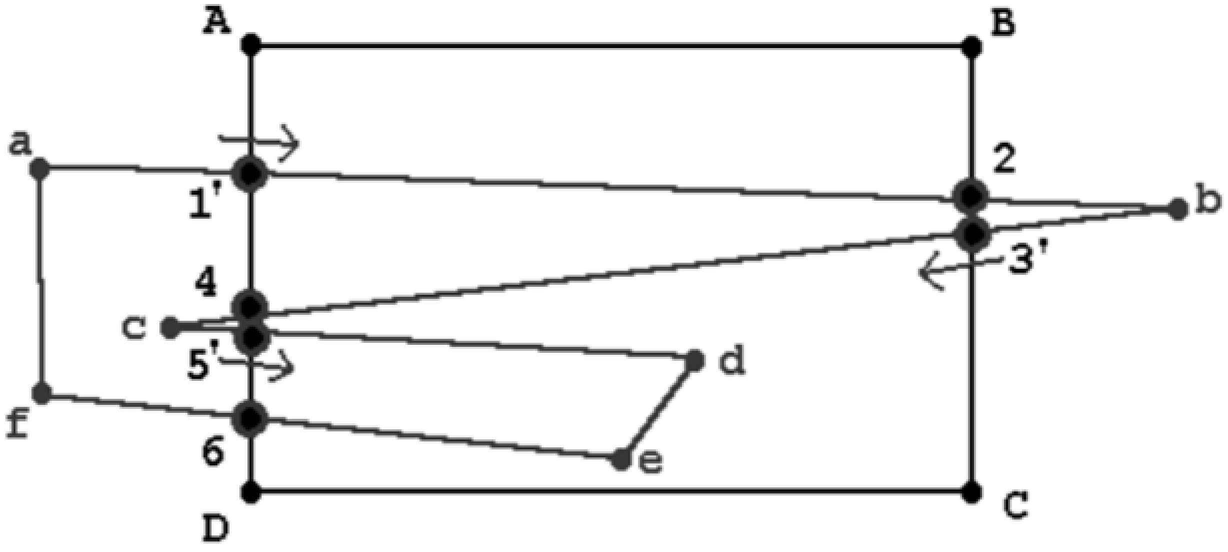
- Line Clipping
  - Cohen-Sutherland Clipping
  - **Liang-Barsky Clipping**



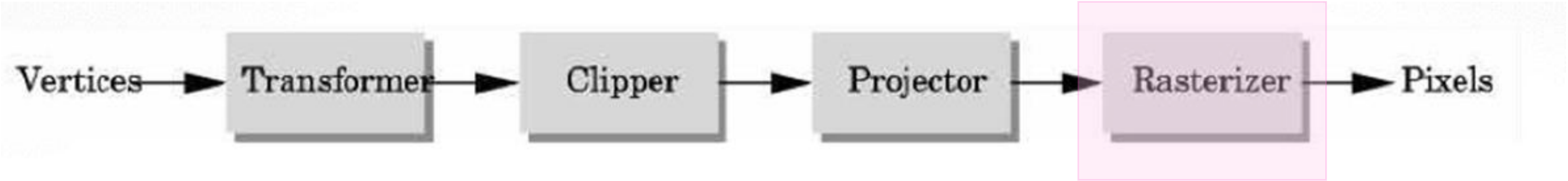


Review

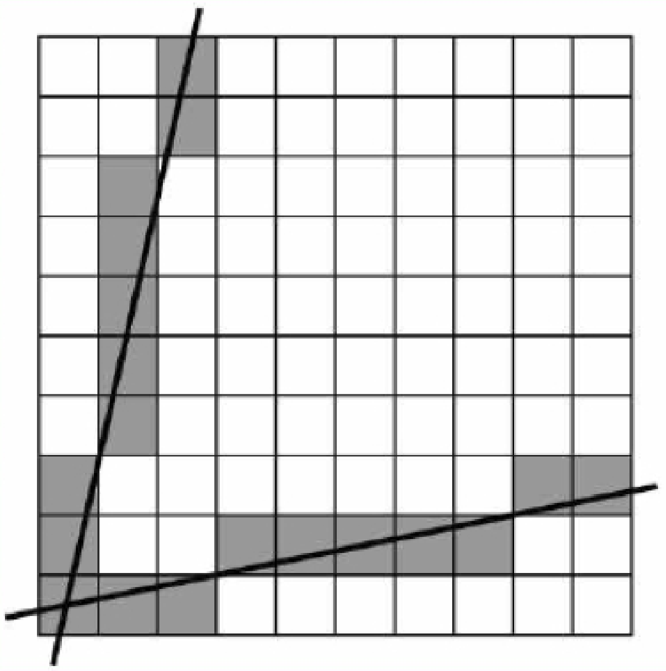
- Polygon Clipping
  - Sutherland-Hodgeman Clipping
  - Weiler-Atherton Clipping



Rasterization (scan conversion)



- Final step in pipeline: rasterization
- **NDC** => **Screen** coordinates (**float**) to **window** coordinates**:** pixels (**int**)
- PIXEL is a single element at (x,y) of Raster Array
  - No smaller drawing unit exists



- Antialiasing

Pixel based Geometry

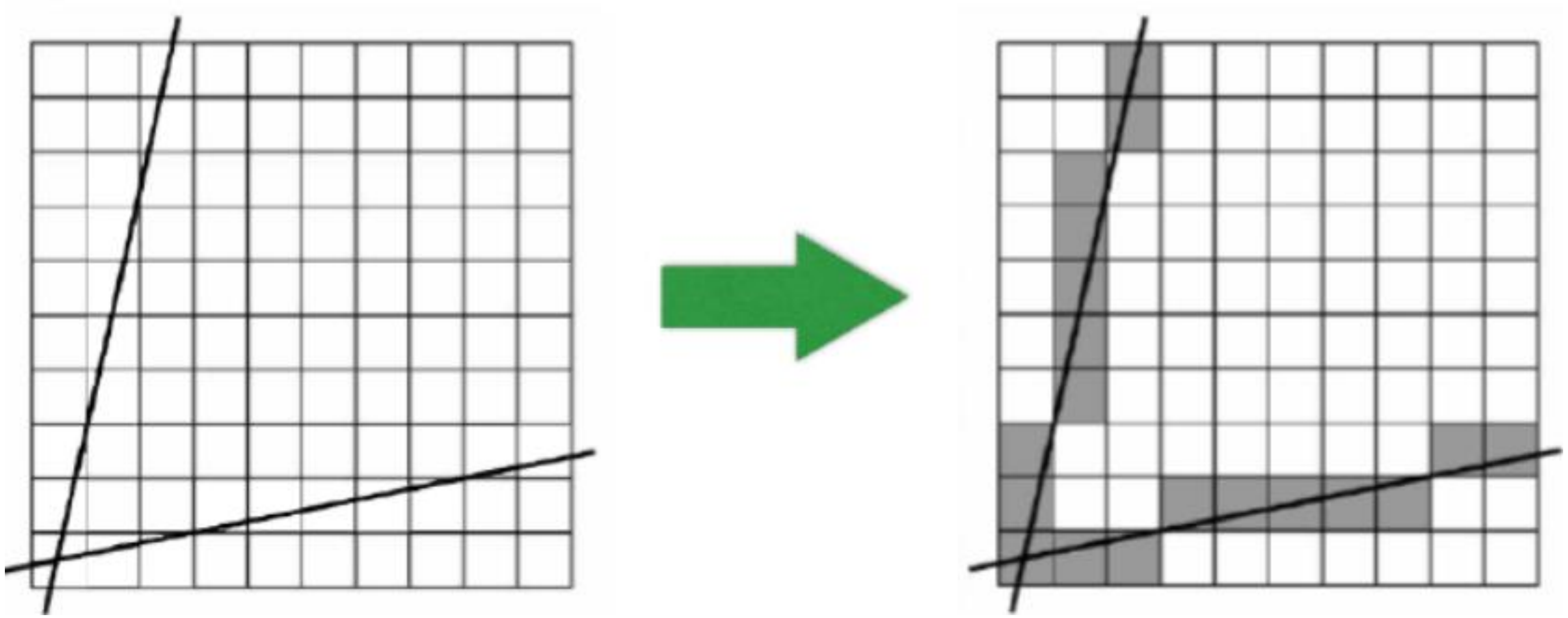
- Points, lines, circles, conics, curves, splines, polygons, shaded polygons, text fonts & icons.

- all basically reduce to scan conversion problem:

FIND Digital algorithms for continuous geometric concepts

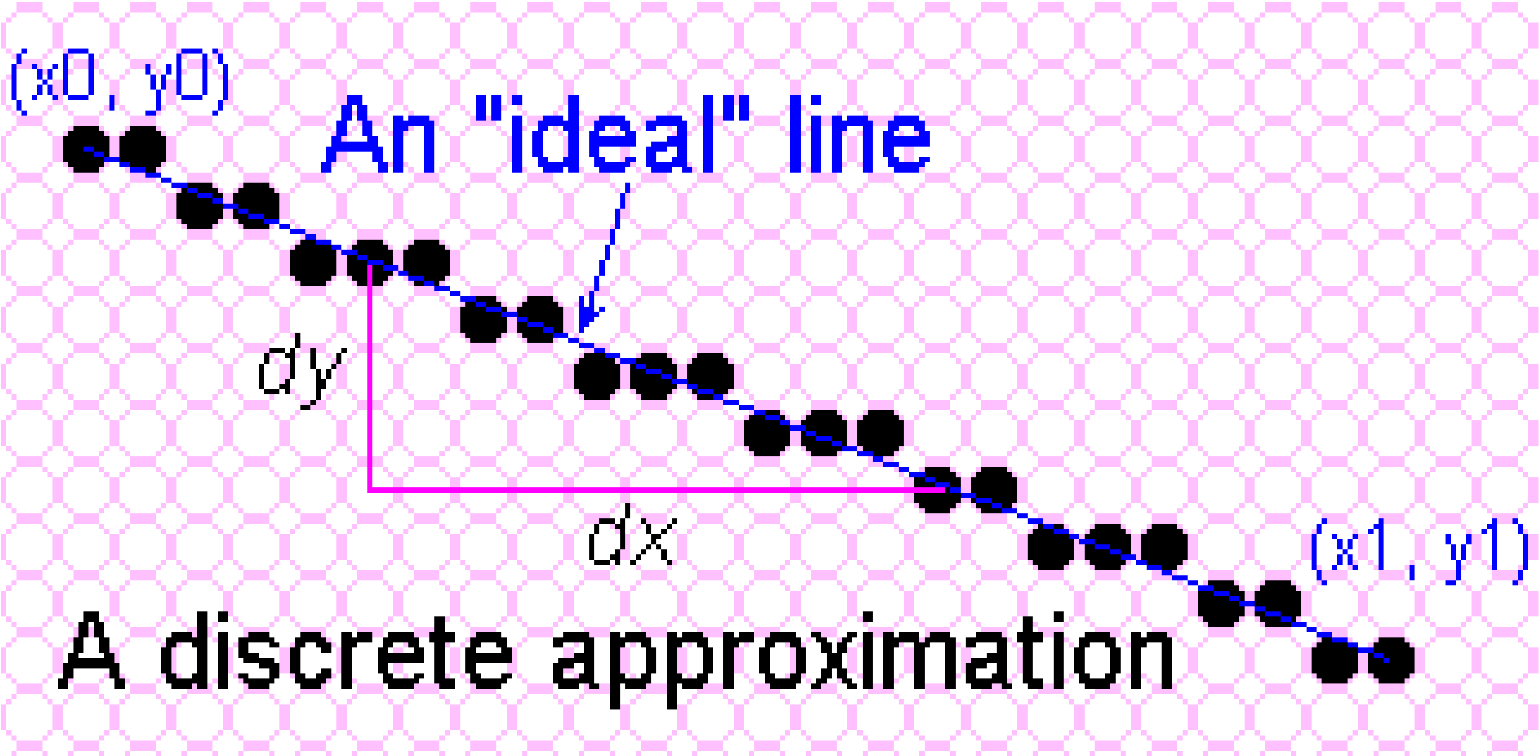
Outline

- **Rasterizing a line**
- Rasterizing a Polygon



Towards the Ideal Line

- We can only do a discrete approximation



- Illuminate pixels as close to the true path as possible, consider **bi-level** display only
  - Pixels are either lit or not lit

What is an *ideal* line

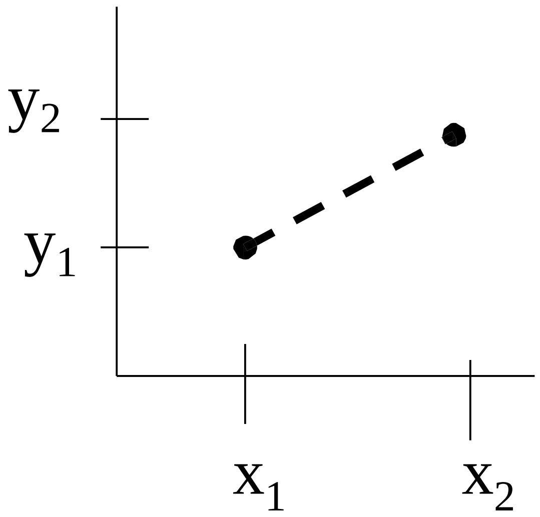
- Must appear **straight and continuous**
  - Only possible axis-aligned and 45^o lines

- Must **interpolate** both defining end points

- Must be **efficient**, drawn quickly
  - Lots of them are required!!!

Drawing Lines

From (x1,y1) to (x2,y2)



Equation:

y = mx + b

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

b = y1 – m*x1

```
compute -- m, b
for x=x1 to x2, step 1
        y = m*x + b
        pixel( x, round(y), color)
loop
```

round(): Returns the integral value that is nearest to *x*, with halfway cases rounded away from zero.

DDA Algorithm

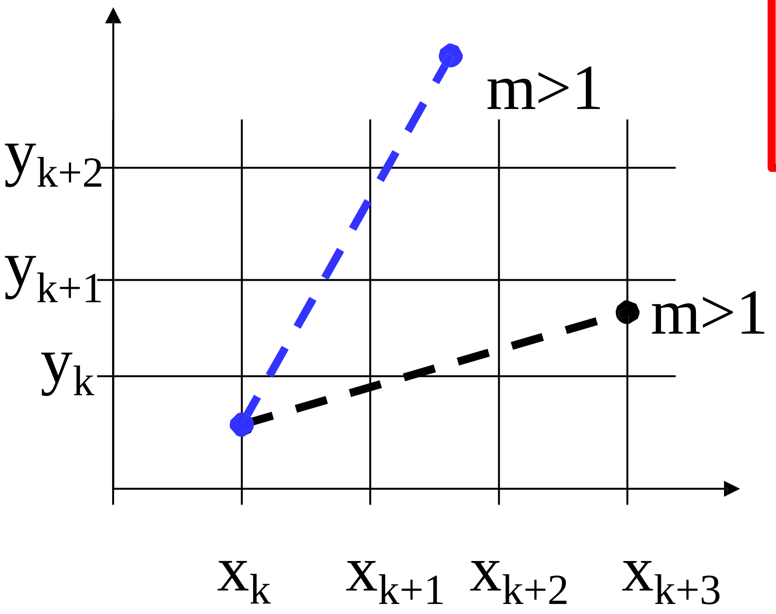
(Digital Differential Analyzer)
[DDA avoids the multiple by slope *m*]

Equation:

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

if slope |m| > 1

$$y_{k+1} = y_k + 1$$
$$x_{k+1} = x_k + 1/m$$




```
compute m (assume |m| <1)

y = y1, x = x1

pixel(round(x1), round(y1))
for x=x1 to x2, step 1
        x = x + 1
        y = y + m
        pixel(x, round(y))
end
```

DDA Algorithm

(Digital Differential Analyzer)

[avoid the float multiple by slope *m*]

slope ***m, y*** – floats Problems:

1. Necessary to perform float addition
2. Necessary to have float representation
3. Necessary to perform -- *round()*

Float representations/operations are more expensive than integer operations. We would like to avoid them if possible.

Mid-Point Line Algorithm (Bresenham)

Uses only integer math to draw a line

Lets assume slope-m is positive and less than 1

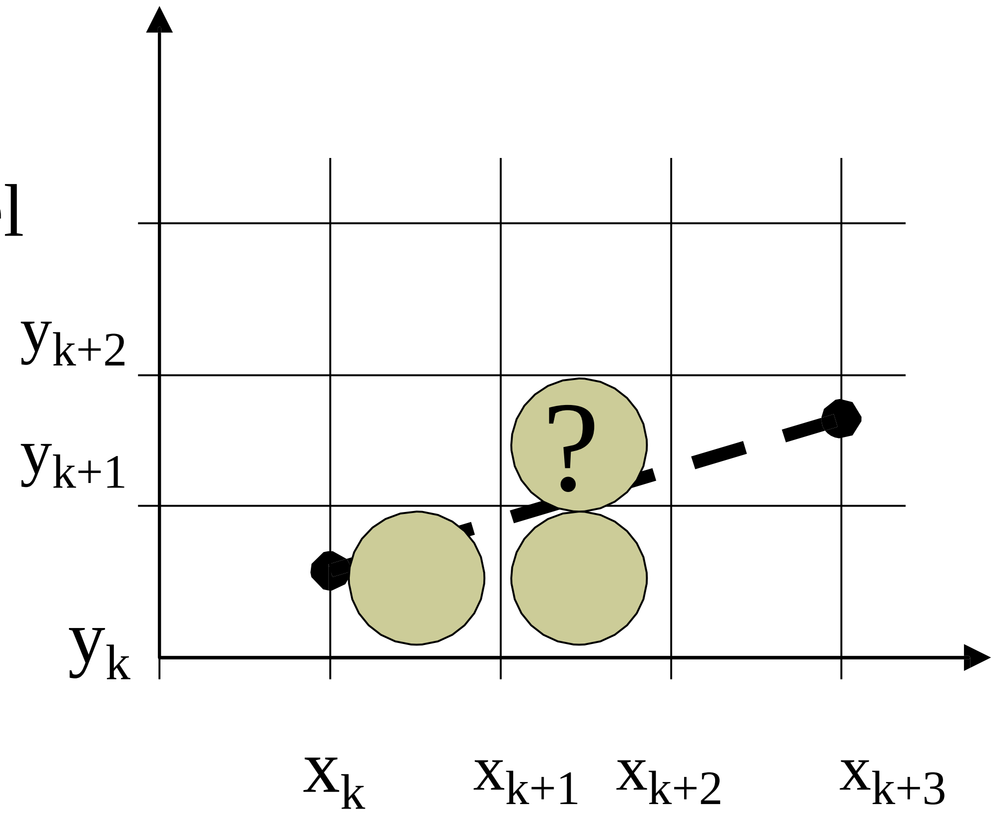
we know that: $x_{k+1} = x_k + 1$

at x_{k+1} we need to make a decision for next pixel

draw either at:

choice 1 -- $(x_k + 1, y_k)$

choice 2 -- $(x_k + 1, y_k + 1)$



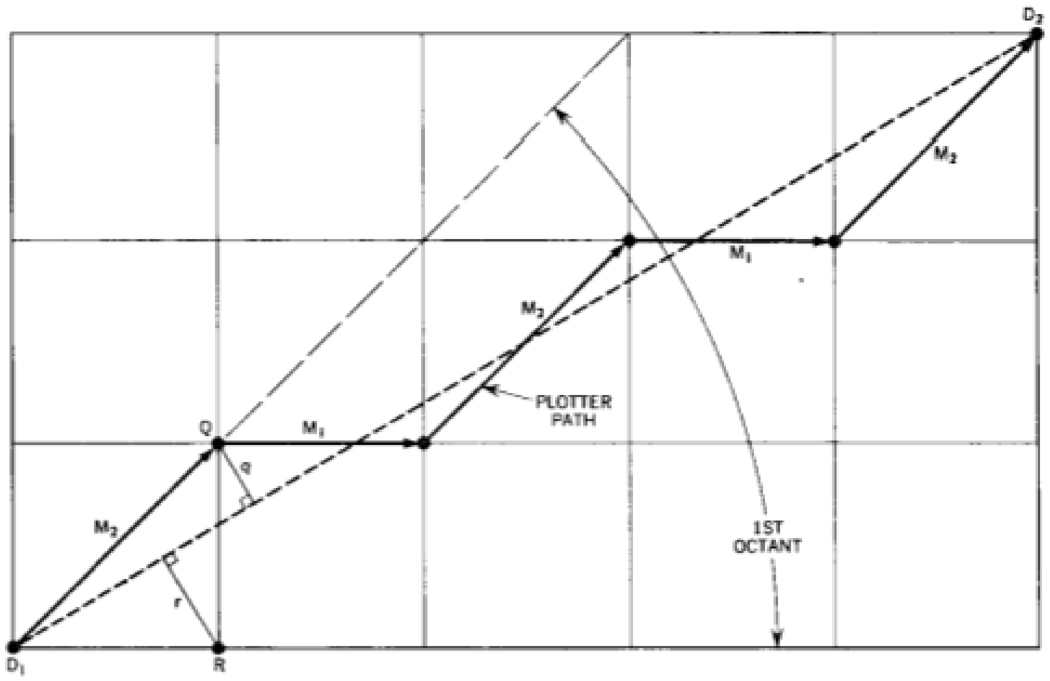
How do we decide between choice 1 or 2?

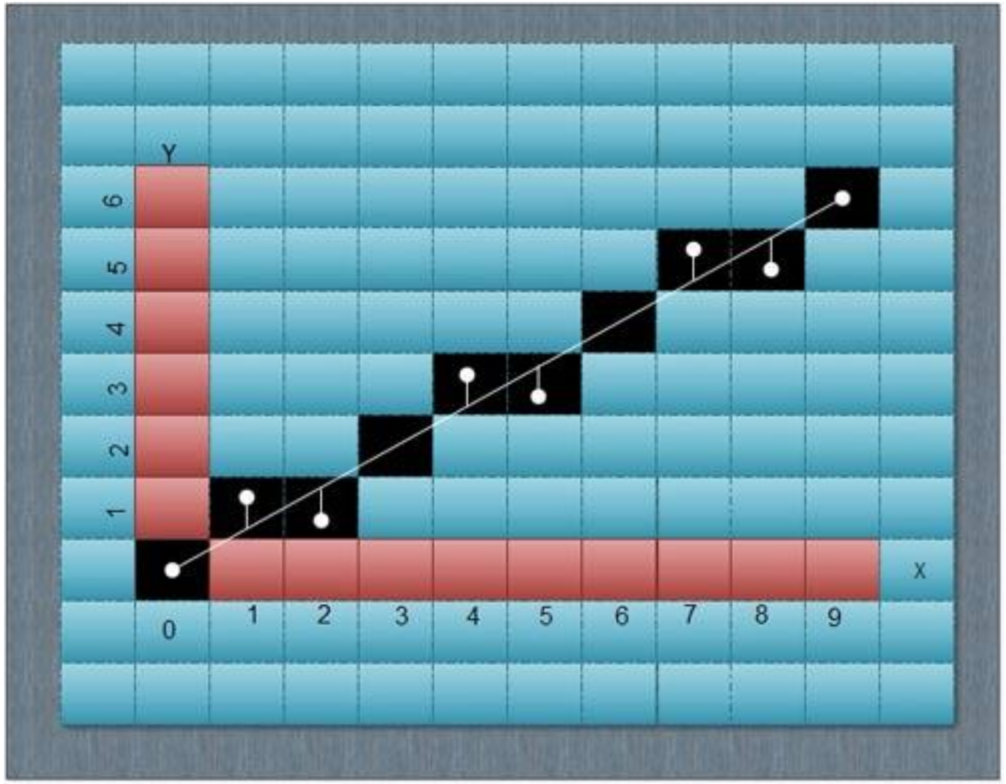
- DDA: $(x_k + 1, \text{round}(f(x_k + 1)))$
- Bresenham: ?

Algorithm for computer control of a digital plotter

- 1962 by Jack Elton Bresenham

Figure 3 Sequence of plotter movements





Comparison of r and q can be implemented by comparing hypotenuse since the two triangles are similar.
Computation of distance of the hypotenuse is simpler, see next page.

Mid-Point Line Algorithm (Bresenham)

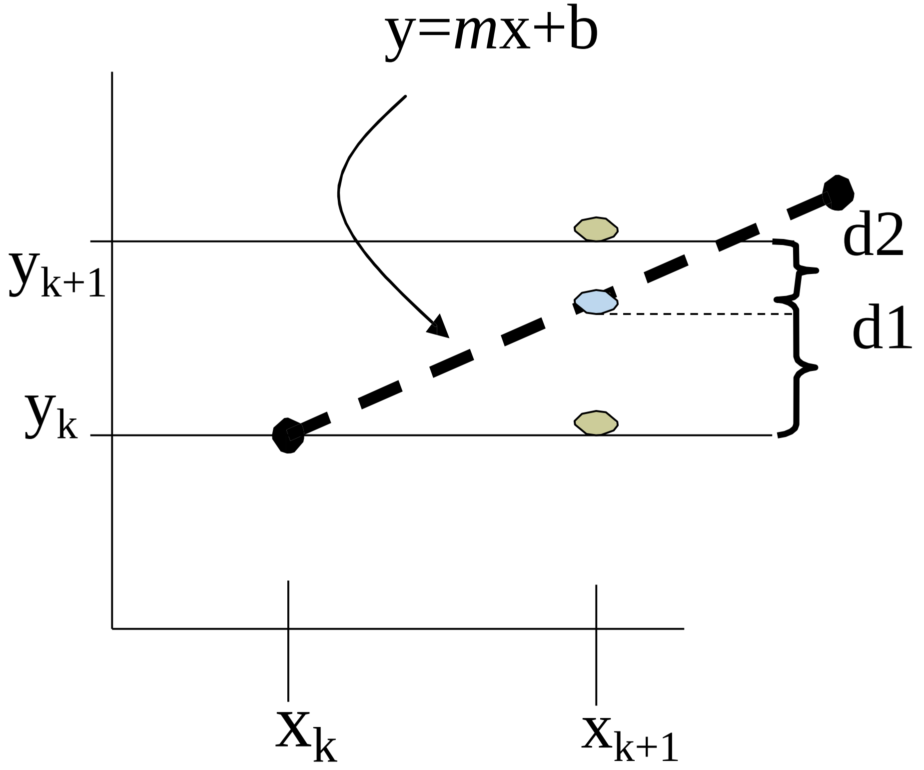
Consider the distance with the actual line ($y=mx+b$) at the two choices:

$d1 = y - y_k$

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

$d2 = (y_k+1) - y$

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



Difference between these two separations is:

```
d = d1 - d2
 if d > 0
     move to (yk +1)
else stay at yk
```

Mid-Point Line Algorithm (Bresenham)

- Still have a "float" operation in calculation of "d"
- Lets create a new decision operator by multiplying Δx (recall $m=\Delta y/\Delta x$)

$p_k = \Delta x\,(d_1 - d_2)$ ~ note p_k sign is the same as $(d_1 - d_2)$

$= 2\,\Delta y\, x_k - 2\,\Delta x\, y_k + 2\,\Delta y + \Delta x\,(2b - 1)$ [Eq 1]

$= 2\,\Delta y\, x_k - 2\,\Delta x\, y_k + C$

We can use p_k as a decision operator – instead of $(d_1 - d_2)$

Example:

(20,10) to (30,18)

$\Delta x = 10$, $\Delta y = 8$

(slope 0.8)

$d1\text{-}d2 = 2\,m\,x_k + 2\,m + (2b - 1) - 2\,y_k$

Mid-Point Line Algorithm (Bresenham)

$p_{k+1} = 2\,\Delta y\; x_{k+1} - 2\,\Delta x\;\; y_{k+1} + C$

What is the change from $\underbrace{p_k \text{ to } p_{k+1}}$?

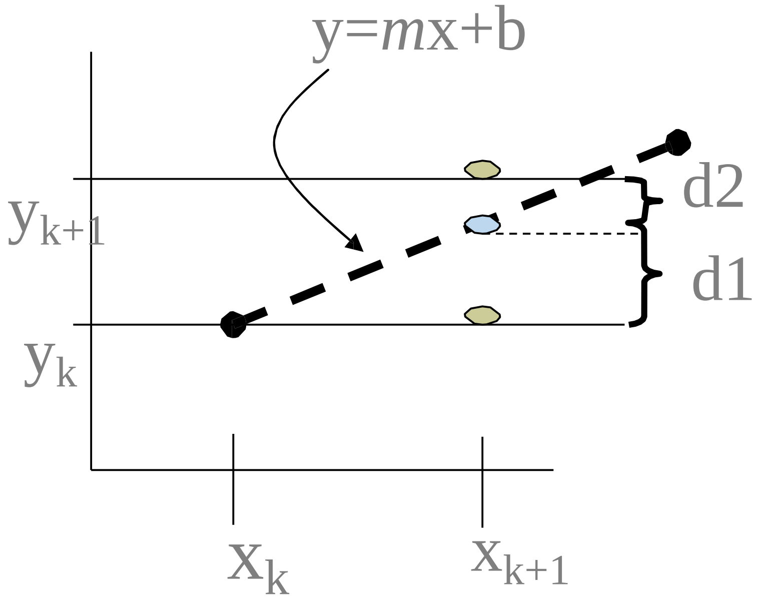
$$p_{k+1} - p_k = 2\,\Delta y\,(x_{k+1} - x_k) - 2\,\Delta x\,(y_{k+1} - y_k)$$

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

$y_{k+1} = y_k+1$ or y_k =>
either 1 or 0

SO

```
if pk < 0                          // yk+1 = yk
    pk+1 = pk + 2 Δy
else                               // yk+1 = yk + 1
    pk+1 = pk + 2 Δy - 2 Δx
```



$p_k = \Delta x\,(d_1 - d_2)$

Mid-Point Line Algorithm (Bresenham)

```
if pk < 0                        // yk+1 = yk
    pk+1 = pk + 2 Δy
else                             // yk+1 = yk + 1
    pk+1 = pk + 2 Δy - 2 Δx
```

- **How to get p_o?**

Use [Eq. 1] with x0 and y0 to find: $p_o = 2\,\Delta y - \Delta x$

$$p_0 = 2\,\Delta y\, x_0 - 2\,\Delta x\, y_0 + 2\,\Delta y + \Delta x\,(2b - 1) \qquad \text{[Eq 1]}$$

$$= 2\,\Delta y - \Delta x + 2\,\Delta y\, x_0 - 2\,\Delta x\, y_0 + 2\,\Delta x\, b$$

$$= 2\,\Delta y - \Delta x + 2\Delta x\,(m\, x_o + b - y_o)$$

$0 == m\, x_o + b - y_o$

Mid-Point Line Algorithm (Bresenham)

Compute constants: **Δx, Δy, 2Δy, 2Δy-2Δx**

Calculate: $p_o = 2\Delta y - \Delta x$

```
plot(xo,yo)
for x=x1 to x2
        if (p < 0)
                // let y=y1
                p = p + 2Δy
        else
                y++
                p = p + 2Δy - 2Δx
        end
        plot(x,y)
loop
```

Note -- main loop:

- Only integer math.
- No float representation, or operations needed.
- Constants: 2dy, 2dx are integers.

Mid-Point Line Algorithm (Bresenham)

Example:

(20,10) to (30,18)

$\Delta x = 10, \quad \Delta y = 8$

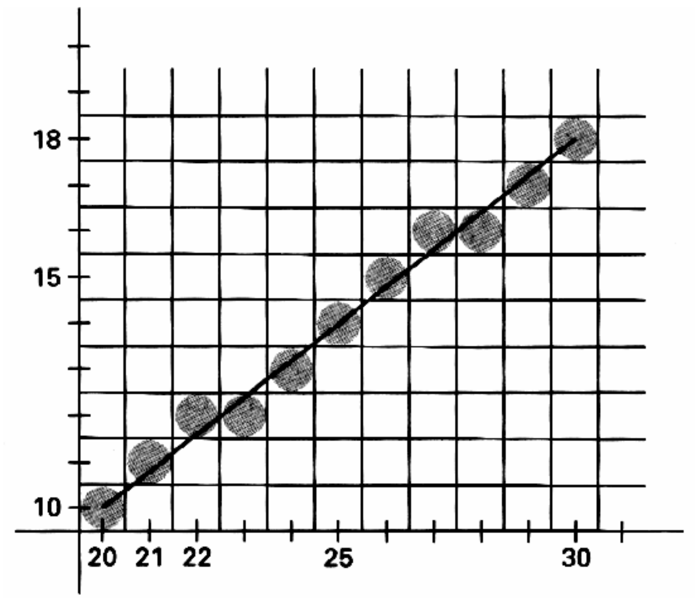
(slope 0.8)

k	P_k	(x_{k+1}, y_{k+1})	k	p_k	(x_{k+1}, y_{k+1})
0	*6*	(21,11)	5	*6*	(26,15)
1	*2*	(22,12)	6	*2*	(27,16)
2	*-2*	(23,12)	7	*-2*	(28,16)
3	*14*	(24,13)	8	*14*	(29,17)
4	*10*	(25,14)	9	*10*	(30,18)

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

$2\Delta y = 16$ [E]

$2\Delta y - 2\Delta x = -4$ [NE]



full algorithm -- page 90-91 Hearn

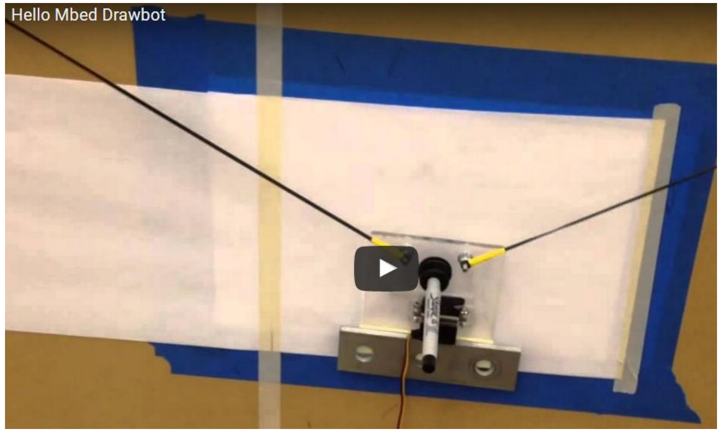
- **adjusts for slope m>1**
- **re-orders x1,x2,y1,y2 as necessary**

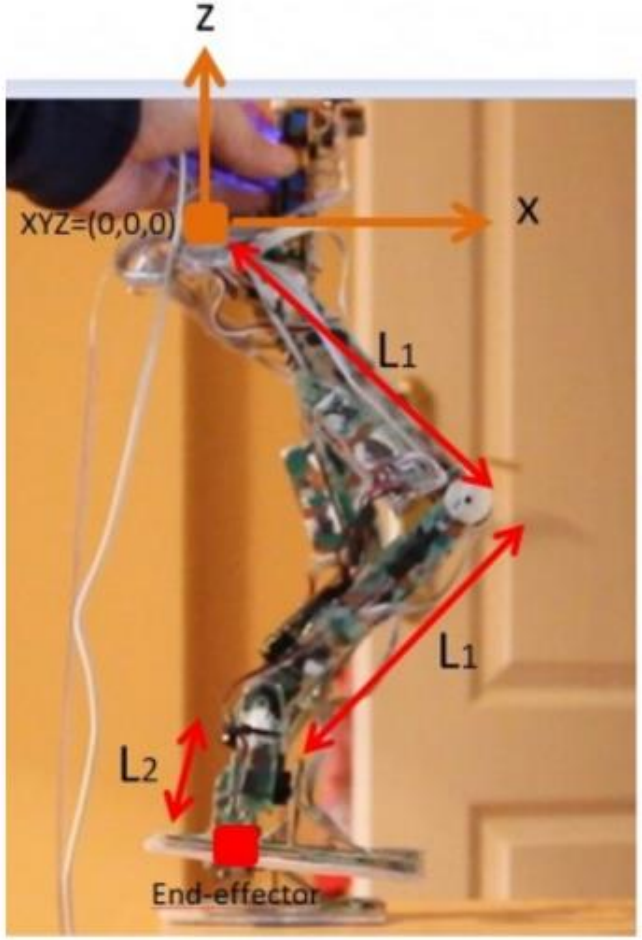
http://www.cosc.canterbury.ac.nz/people/mukundan/cogr/LineMP.html

Summary of Concepts

- **Incremental** algorithm for more efficient computation
- **Integer** algorithm by introducing harmless **scale** factor
- Logic must be added to handle all line orientations (m>1)

- Highly efficient
- Widely used
  - Robot
    - Path planning
    - Trajectory Generation





History

- Bresenham's line algorithm is named after Jack Elton Bresenham who developed it in 1962 at IBM.
- The Calcomp 565 drum plotter, introduced in 1959, was one of the first computer graphics output devices sold.
- Later extended to *Bresenham's circle algorithm* or midpoint circle algorithm.

A Calcomp 565 drum plotter.

Closeup of Calcomp plotter right side, showing controls for manually moving the drum. Similar controls on the left move the pen carriage.

A Simple Circle Drawing Algorithm

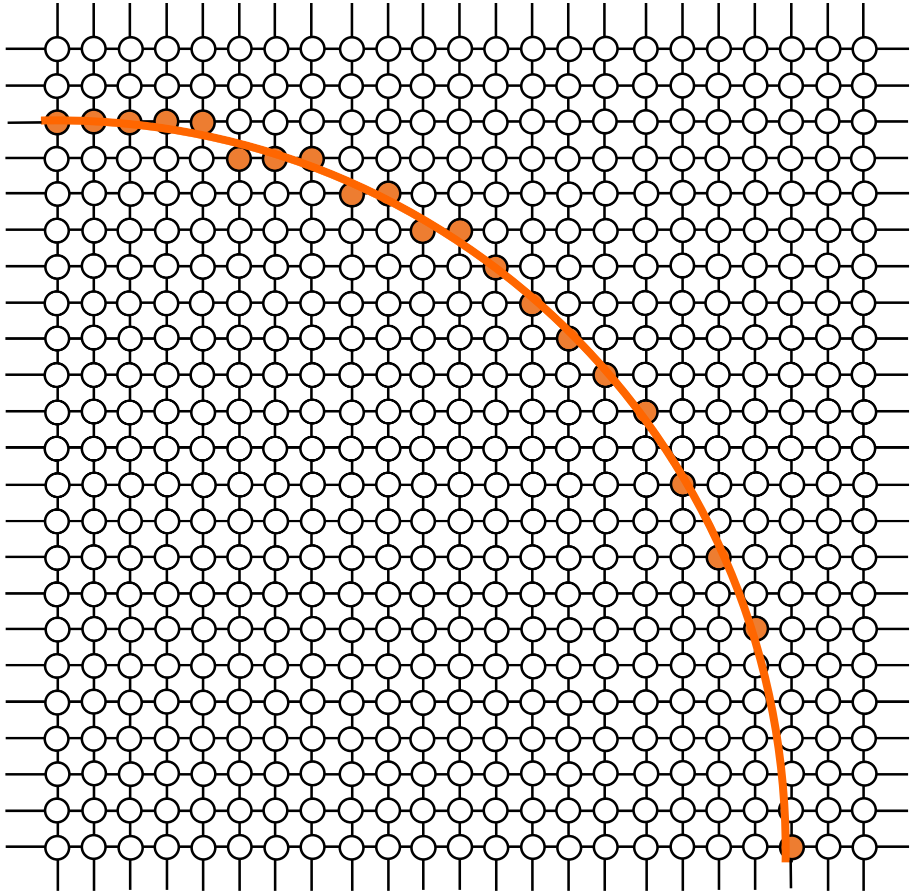
- The equation for a circle is:

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

- 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…)



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

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

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

$$\vdots$$

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

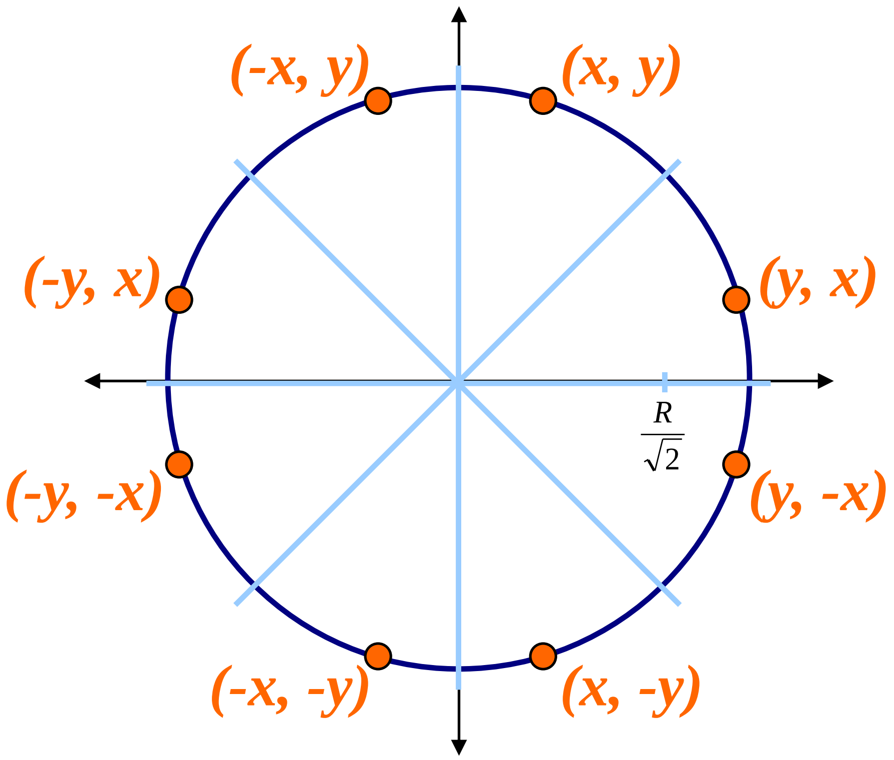
$$y_{20} = \sqrt{20^2 - 20^2} \approx 0$$

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*



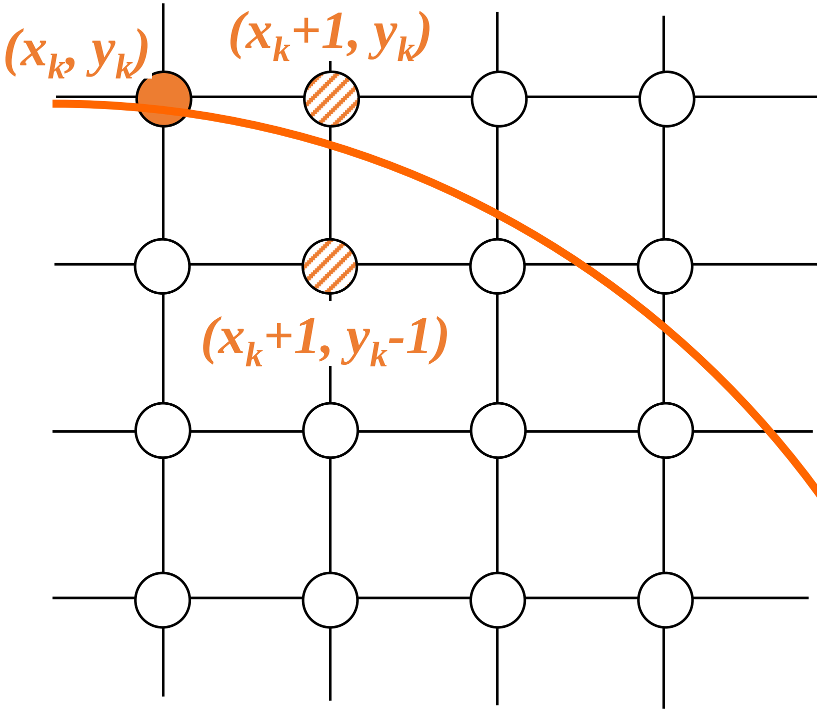
Mid-Point Circle Algorithm

- 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.

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?



Mid-Point Circle Algorithm (cont…)

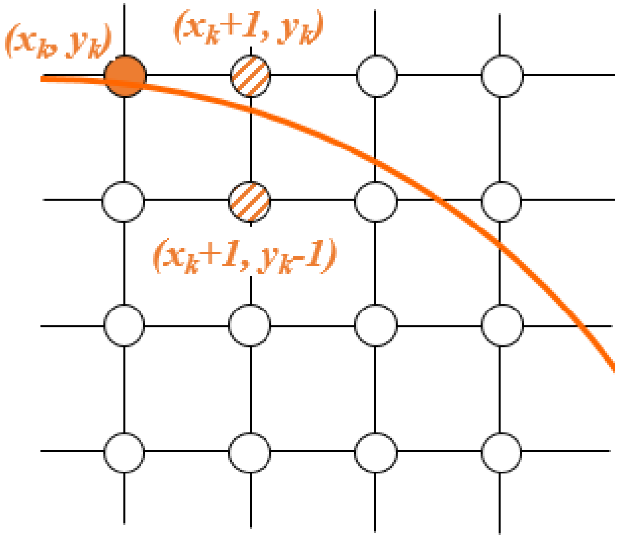
- Let's re-jig the equation of the circle slightly to give us:

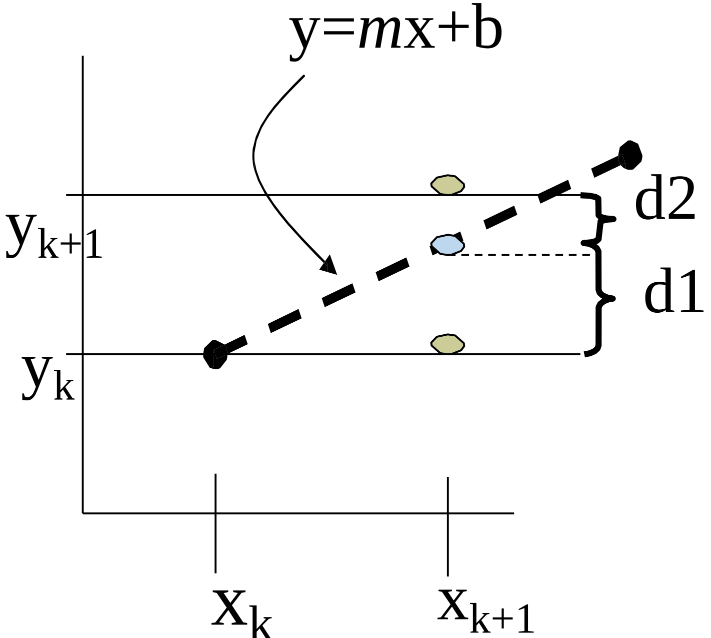
$$f_{circ}(x,y) = x^2 + y^2 - r^2$$

- 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

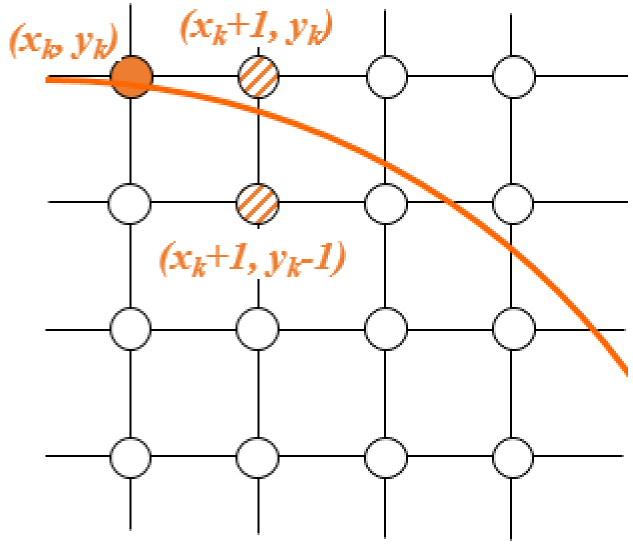




Mid-Point Circle Algorithm (cont…)

- 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 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:

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

- 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:

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

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

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:

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

- 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 $x >= y$

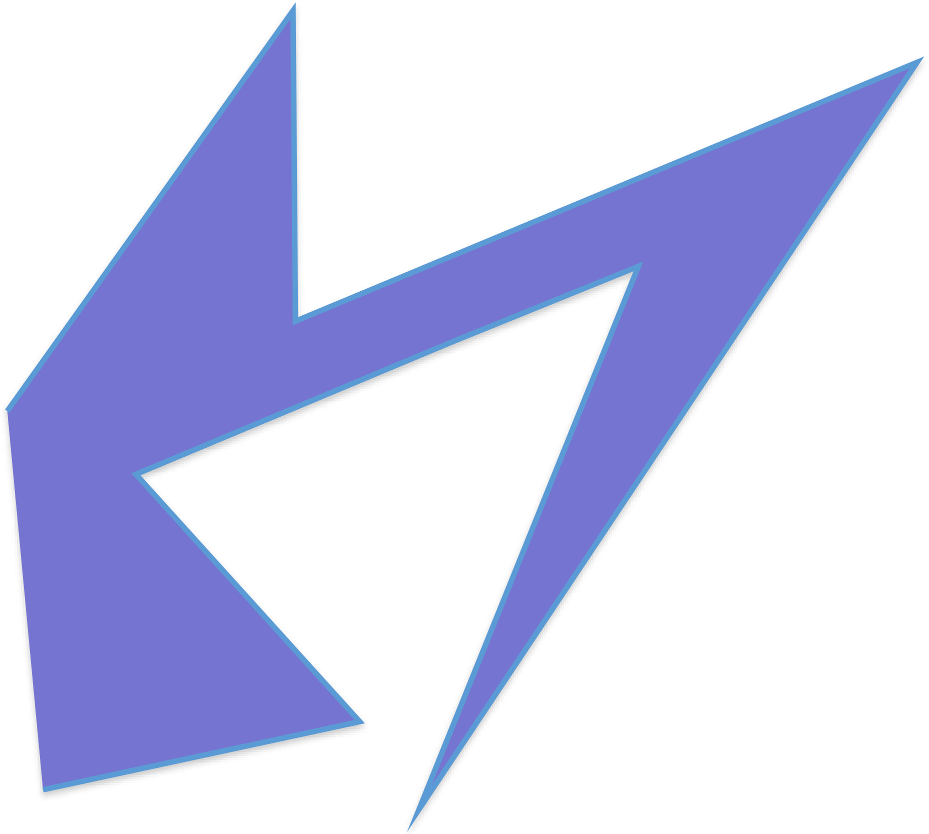
Mid-Point Circle Algorithm Summary

- 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

Outline

- Rasterizing a line
- **Rasterizing a Polygon**



Scan Conversion of Polygons

- Multiple tasks:
  - Filling polygon (inside/outside)
  - Pixel shading (color interpolation)
  - Blending (accumulation, not just writing)
  - Depth values (z-buffer hidden-surface removal)
  - Texture coordinate interpolation (texture mapping)
- Hardware efficiency is critical
- Many algorithms for filling (inside/outside)
- Much fewer that handle all tasks well

Simple 2D Polygon

- an ordered sequence of line segments

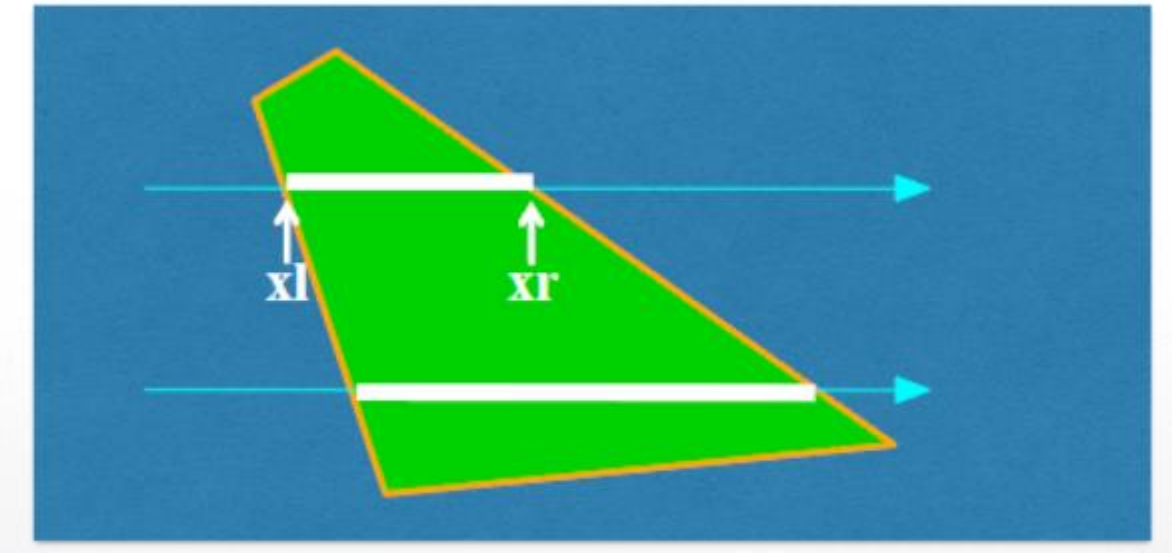
$$g_0, g_1, \ldots, g_{n-1} \quad n \geq 2$$

- such that
  - each edge $g_i = (s_i, e_i)$ is the segment from a start vertex s_i to an end vertex e_i
  - $e_{i-1} = s_i$ for $0 < i \leq n-1$ and $e_{n-1} = s_0$
  - non-adjacent edges do not intersect
  - the only intersection of adjacent edges is at their shared vertex

Flood Fill v.s. Scan Line Polygon Fill

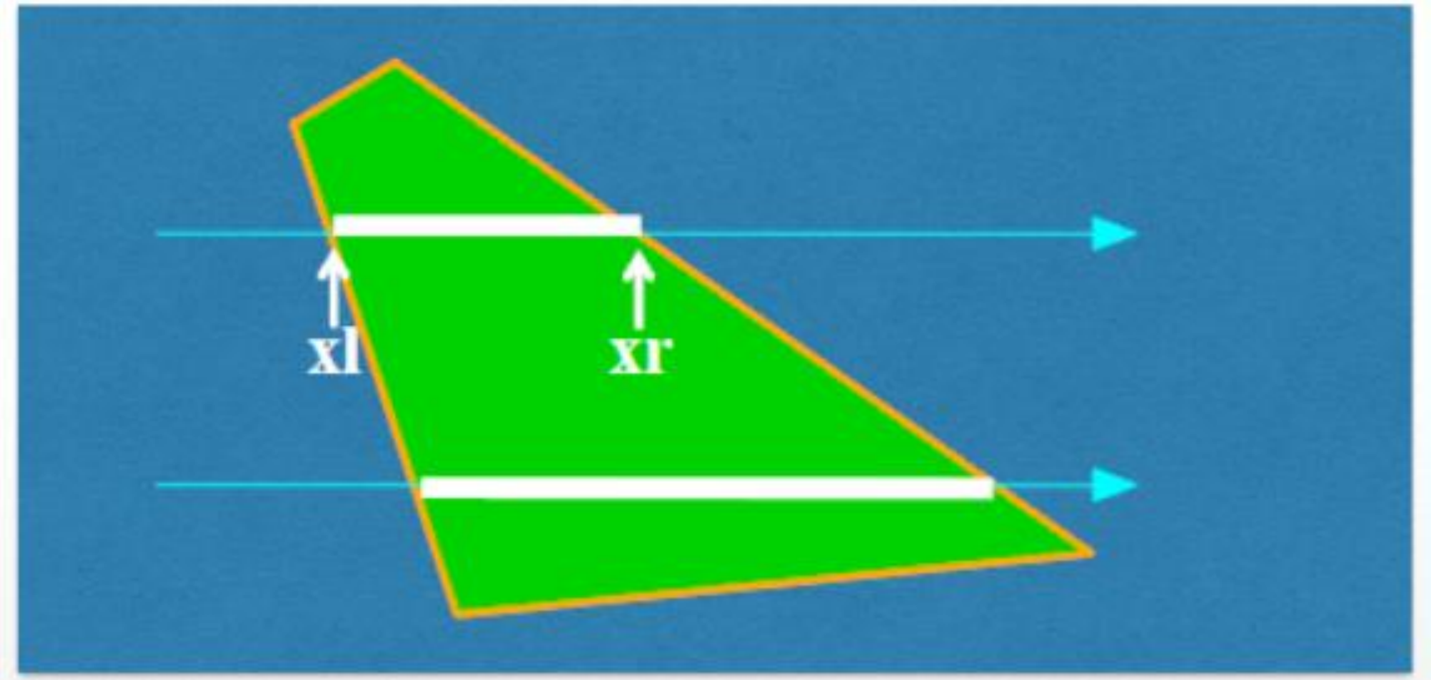
Draw outline of polygon

- Pick color seed
- Color surrounding pixels and recurse
- Must be able to test boundary and duplication
- More appropriate for drawing than rendering （See Photoshop for Flood Fill）



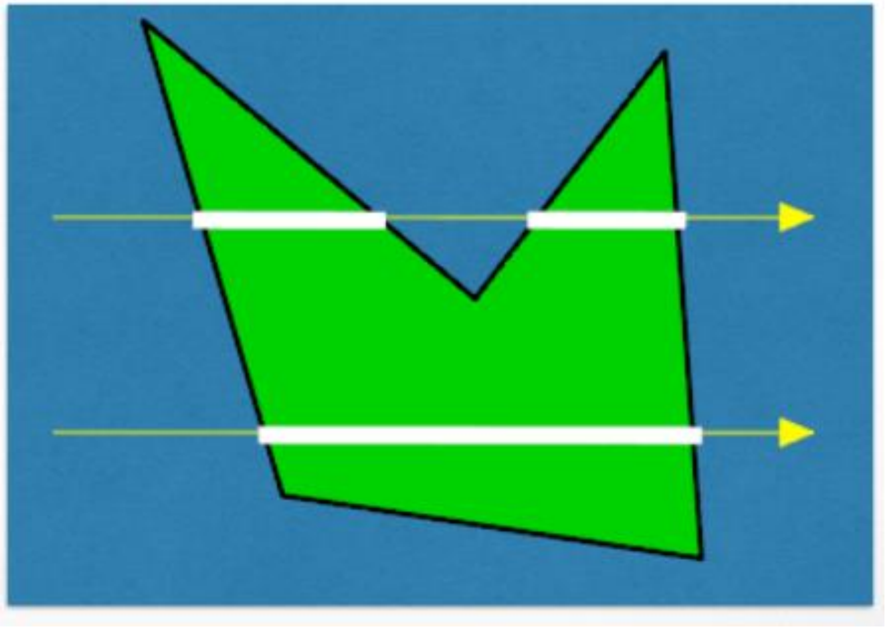
Filling *Convex* Polygons

- Find top and bottom vertices
- List edges along left and right sides
- For each scan line from bottom to top
  - Find left and right endpoints of span, xl and xr
  - Fill pixels between xl and xr
  - Can use Bresenham's algorithm to update xl and xr



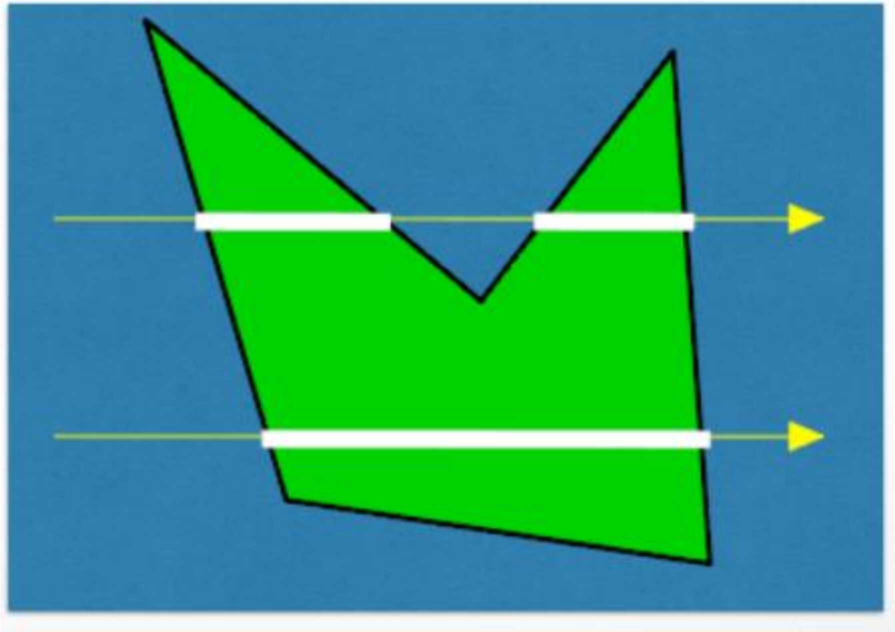
Concave Polygons: Tessellation

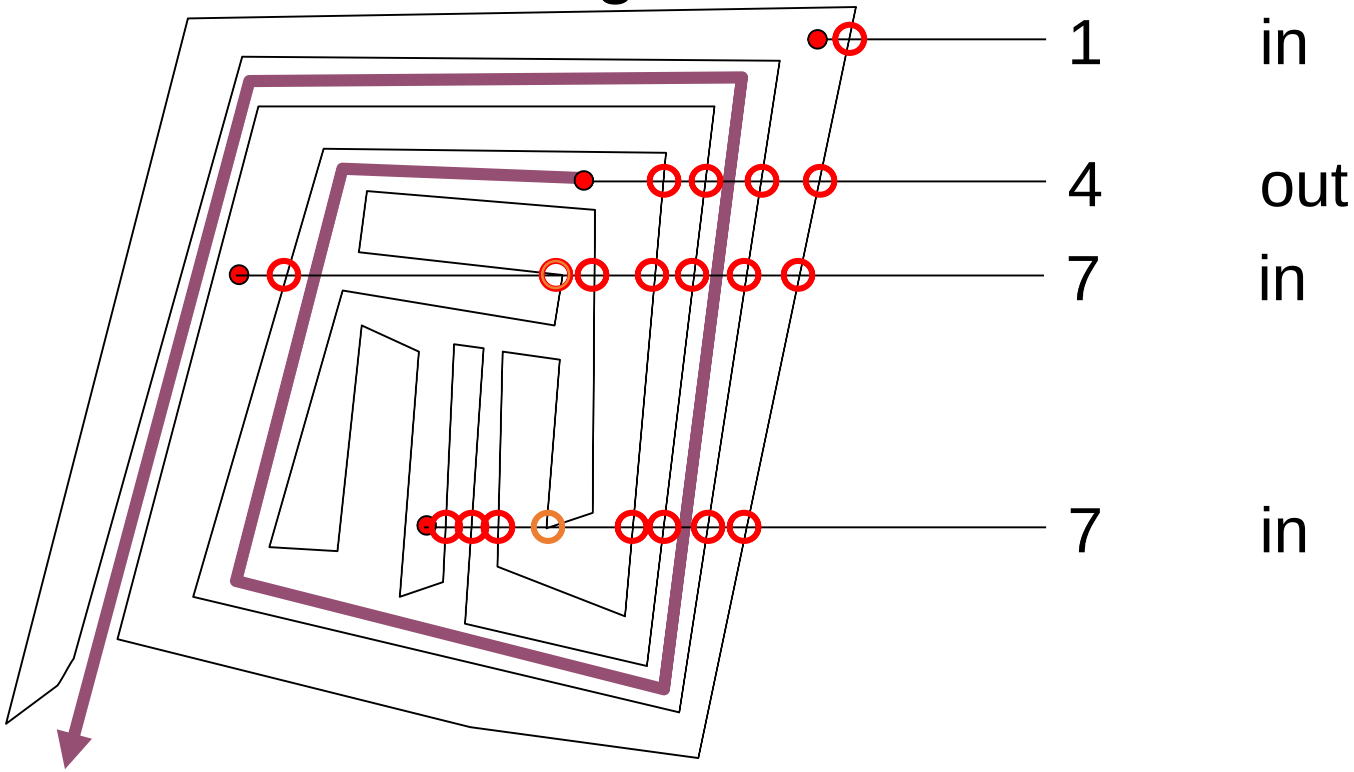
- Approach 1: divide non-convex, non-flat, or non-simple polygons into triangles
- OpenGL specification
  - Need accept only simple, flat, convex polygons
  - Tessellate explicitly with tessellator objects
- Most modern GPUs scan-convert only triangles



Concave Polygons: Odd-Even Test

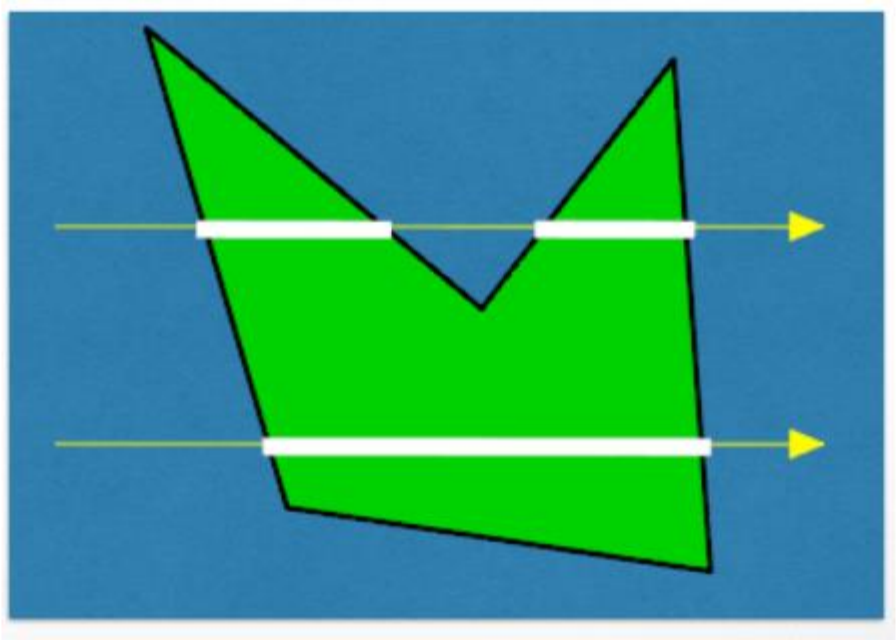
- Approach 2: odd-even test
- For each scan line
  - Find all scan line/polygon intersections
  - Sort them left to right
  - Fill the interior spans between intersections
- Parity rule: inside after an odd number of crossings





Edge vs Scan Line Intersections

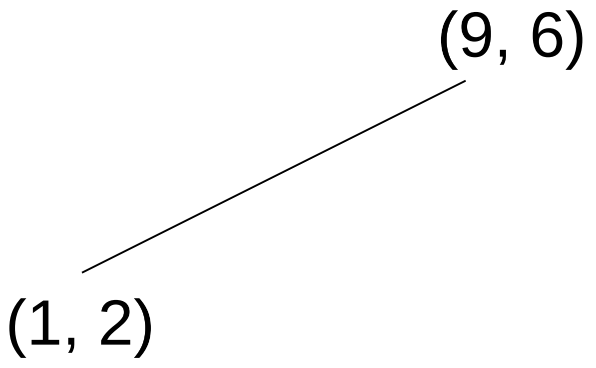
- Brute force: calculate intersections explicitly
- Incremental method (Bresenham's algorithm)
- Caching intersection information
  - Edge table with edges sorted by ymin
  - Active edges, sorted by x-intersection, left to right
- Process image from smallest ymin up



Polygon Data Structure

edges

xmin	ymax	1/m	→
1	6	8/4	→



xmin = x value at lowest y

ymax = highest y

Why 1/m?

If $y = mx + b$, $x = (y-b)/m$.

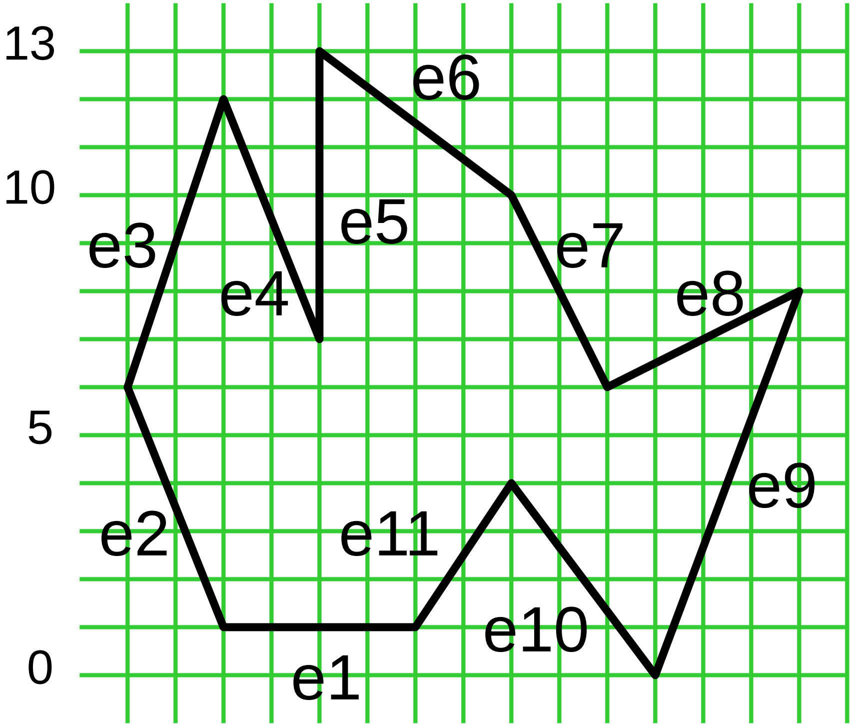
x at $y+1 = (y+1-b)/m = (y-b)/m + 1/m$

$= x+1/m$

Polygon Data Structure

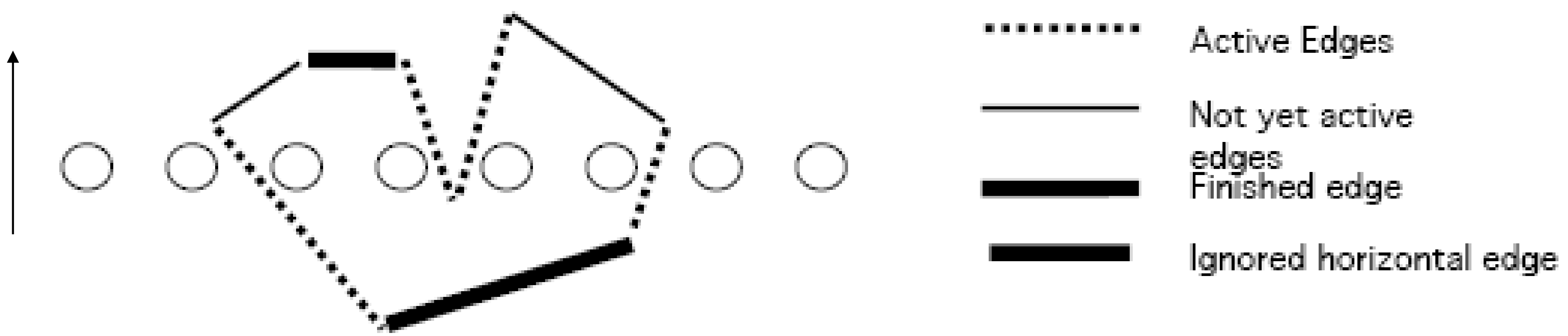
13			
12			
11			
10	e6		
9			
8			
7	e4	e5	
6	e3	e7	e8
5			
4			
3			
2			
1	e2	e1	e11
0	e10	e9	

Edge Table (ET) has a list of edges for each scan line.



Active Edge Table

- A table of edges that are currently used to fill the polygon
- Scan lines are processed in increasing Y order.
- Polygon edges are sorted according to their minimum Y.
- When the current scan line reaches the lower endpoint of an edge it becomes active.
- When the current scan line moves above the upper endpoint, the edge becomes inactive.

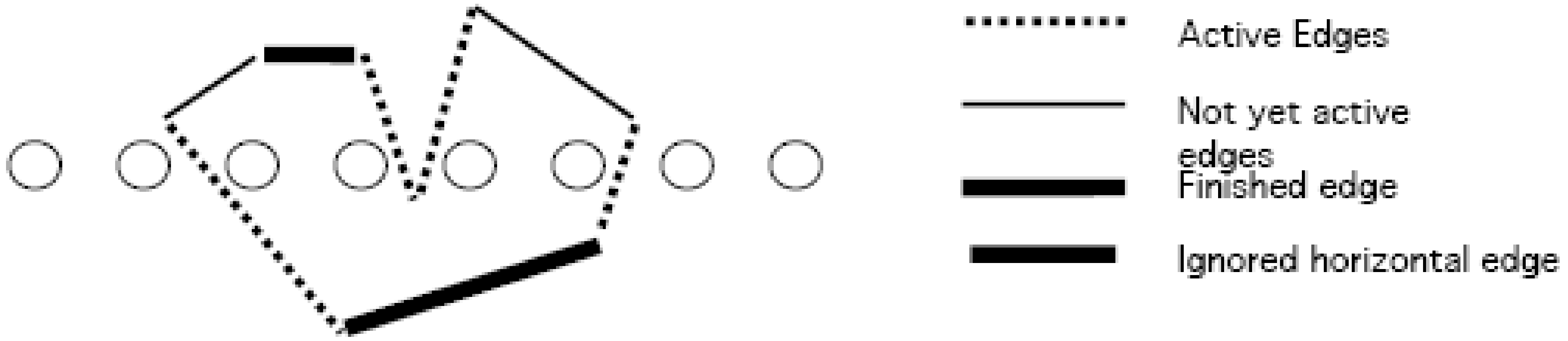


Active Edge Table

•Active edges are sorted according to increasing X.

•Filling in pixels between left most edge intersection and stops at the second.

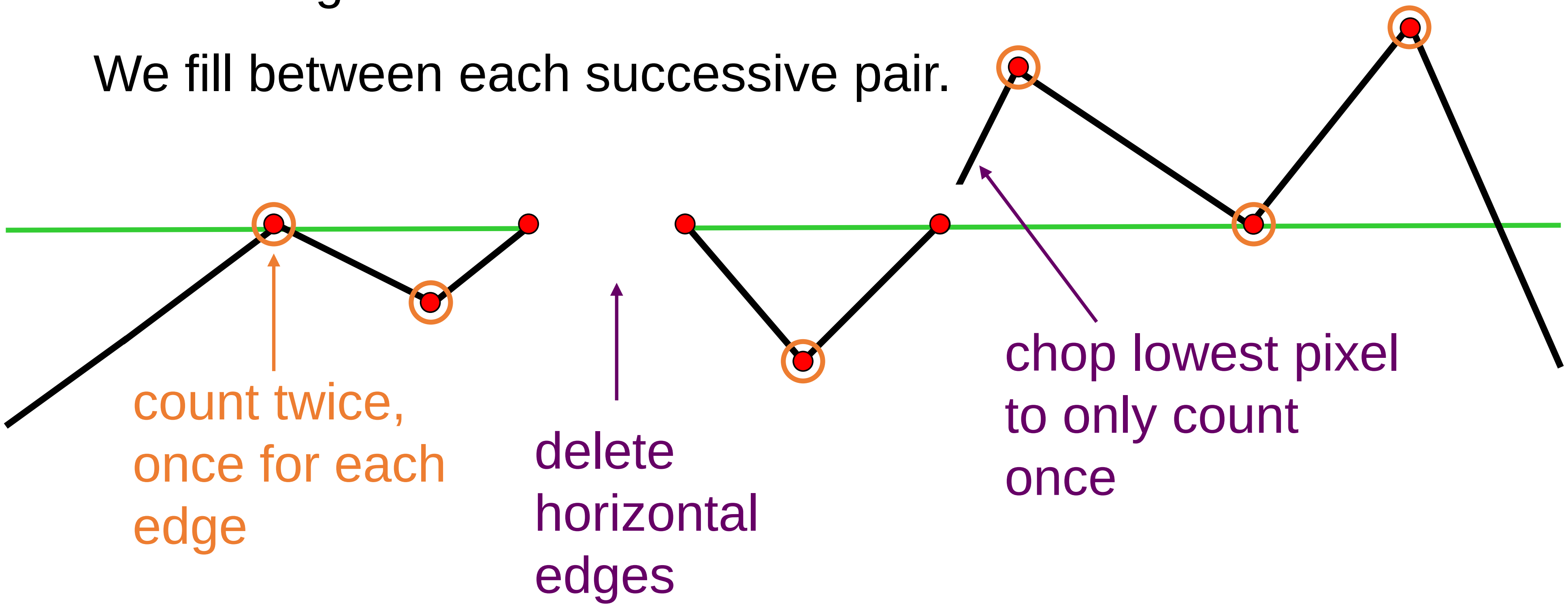
•Restarts at the third intersection and stops at the fourth.



Preprocessing the edges

For a closed polygon, there should be an even number of crossings at each scan line.

We fill between each successive pair.



Polygon Data Structure
after preprocessing

Edge Table (ET) has a list of edges for each scan line.

13			
12			
11			
10	e6		
9			
8			
7	e4	e5	
6	e3	e7	e8
5			
4			
3			
2			
1	e2	e1	e11
0	e10	e9	

13

12

11 → e6

10

9

8

7 → e3 → e4 → e5

6 → e7 → e8

5

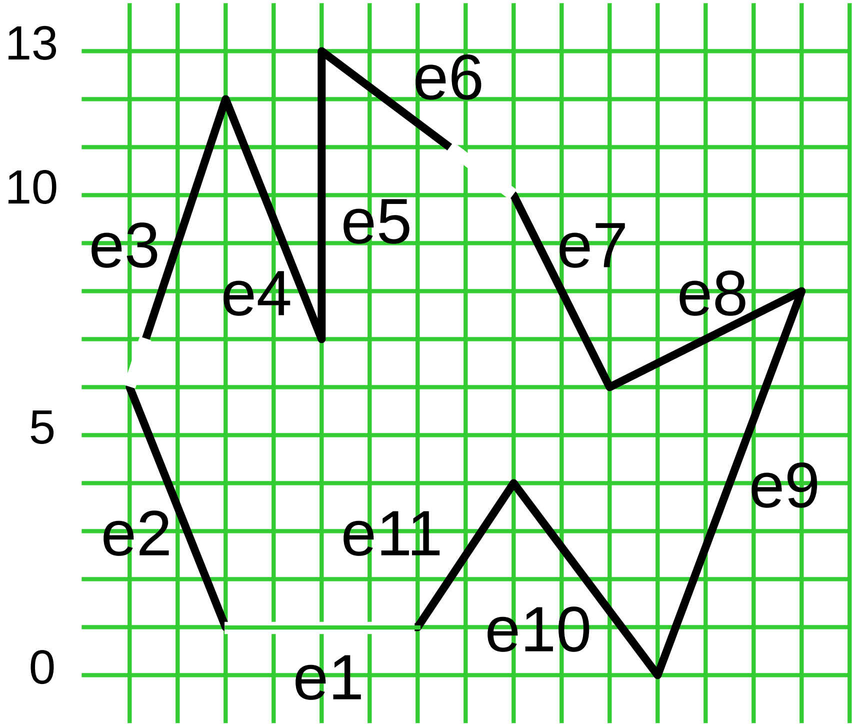
4

3

2

1 → e2 → e11

0 → e10 → e9



The Algorithm

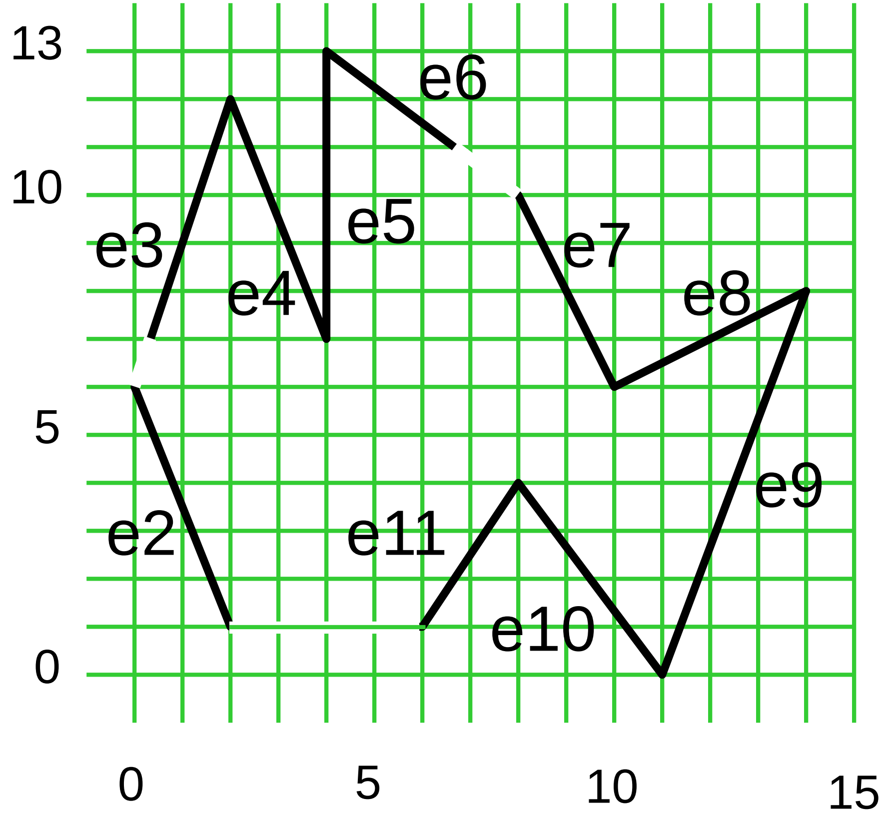
1. Start with smallest nonempty y value in ET.
2. Initialize SLB (Scan Line Bucket) to *nil*.
3. While current y ≤ top y value:
   a. Merge y bucket from ET into SLB; sort on xmin.
   b. Fill pixels between rounded pairs of x values in SLB.
   c. Remove edges from SLB whose ytop = current y.
   d. Increment xmin by 1/m for edges in SLB.
   e. Increment y by 1.

Running the Algorithm

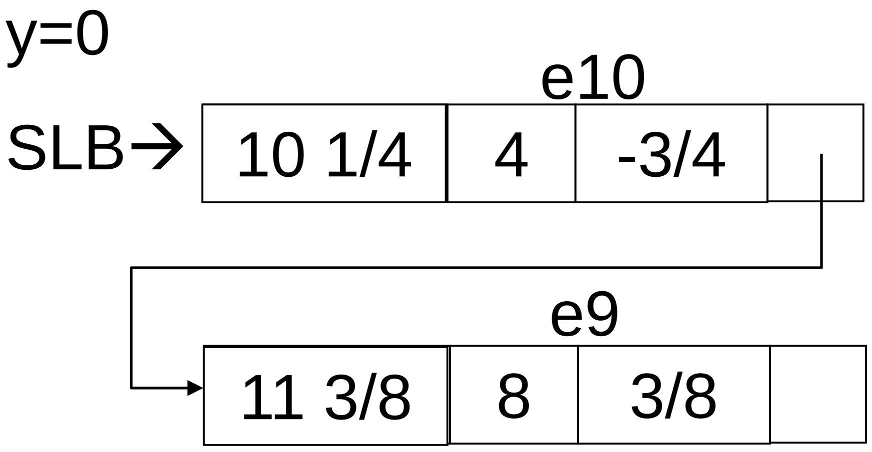
ET

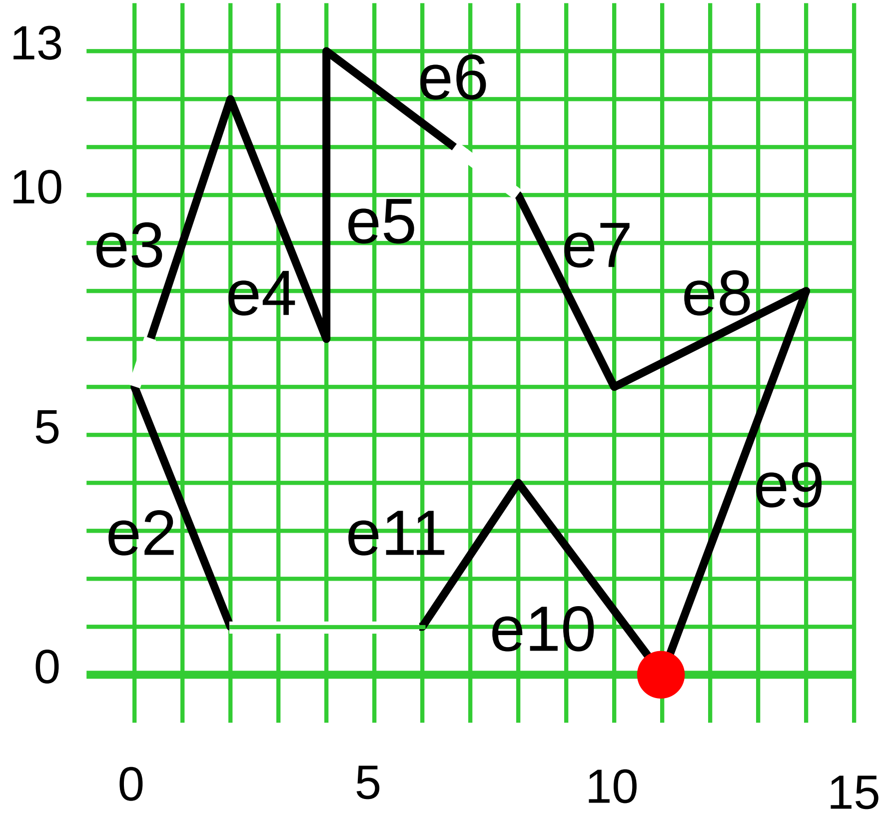
13
12
11 → e6
10
9
8
7 → e3 → e4 → e5
6 → e7 ve8
5
4
3
2
1 → e2 → e11
0 → e10 → e9

	xmin	ymax	1/m
e2	2	6	-2/5
e3	1/3	12	1/3
e4	4	12	-2/5
e5	4	13	0
e6	6 2/3	13	-4/3
e7	10	10	-1/2
e8	10	8	2
e9	11	8	3/8
e10	11	4	-3/4
e11	6	4	2/3

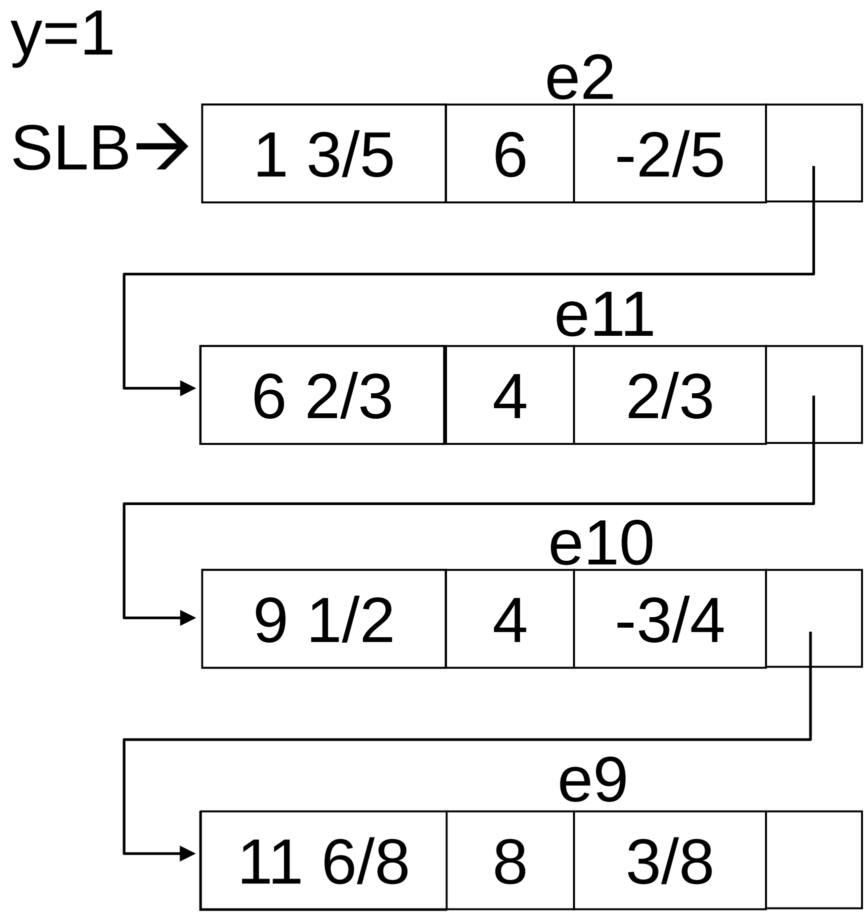


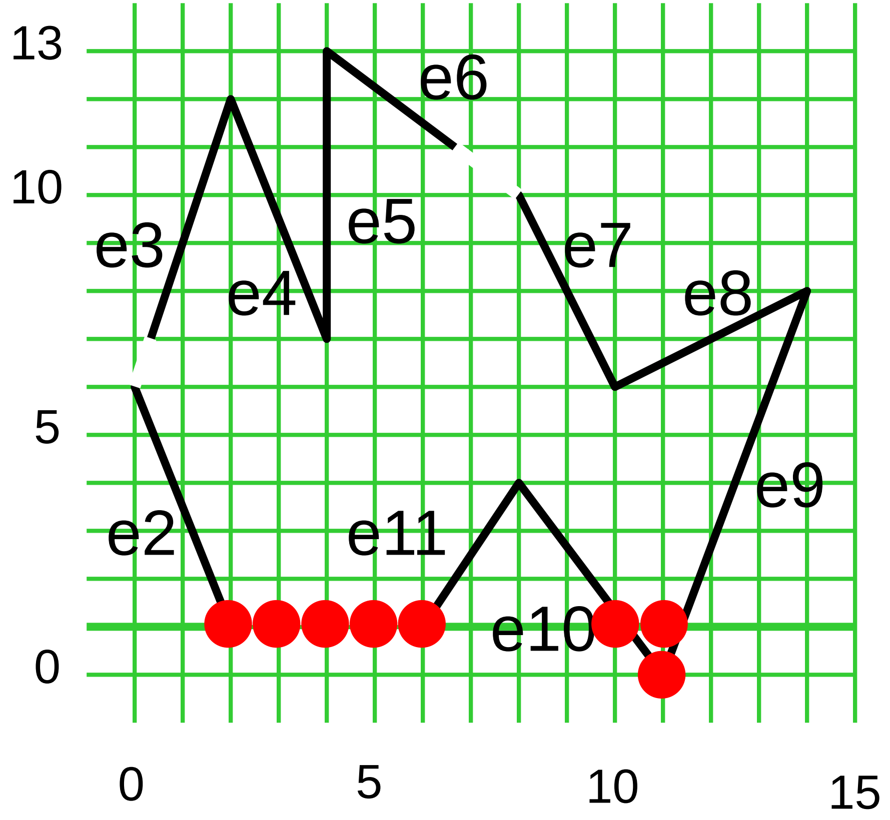
Running the Algorithm





Running the Algorithm



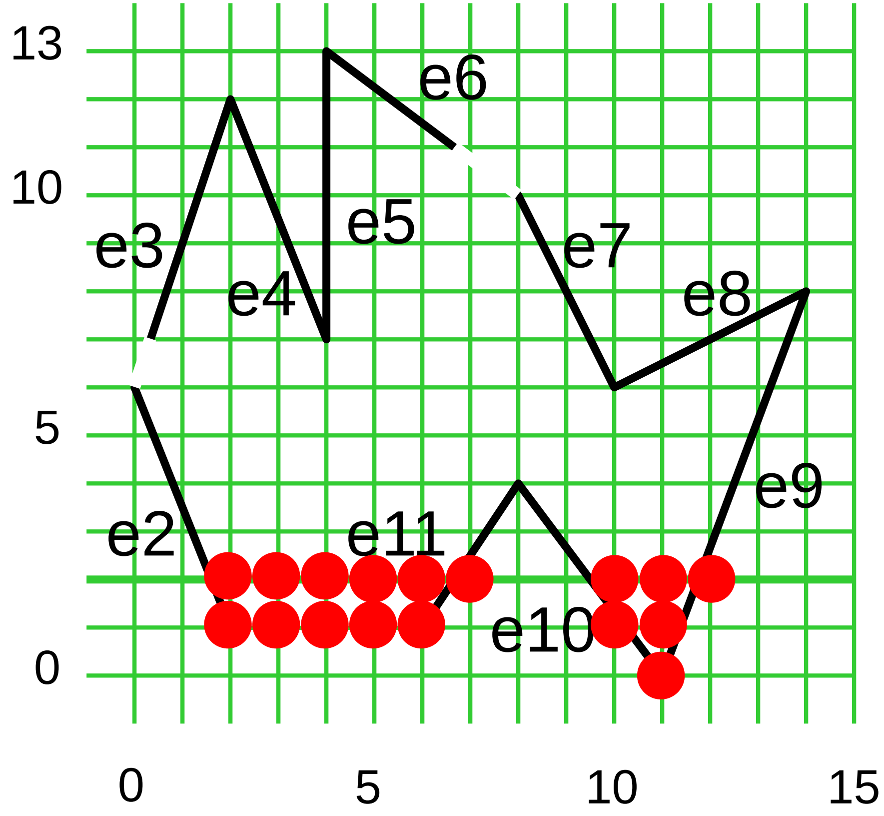


Running the Algorithm

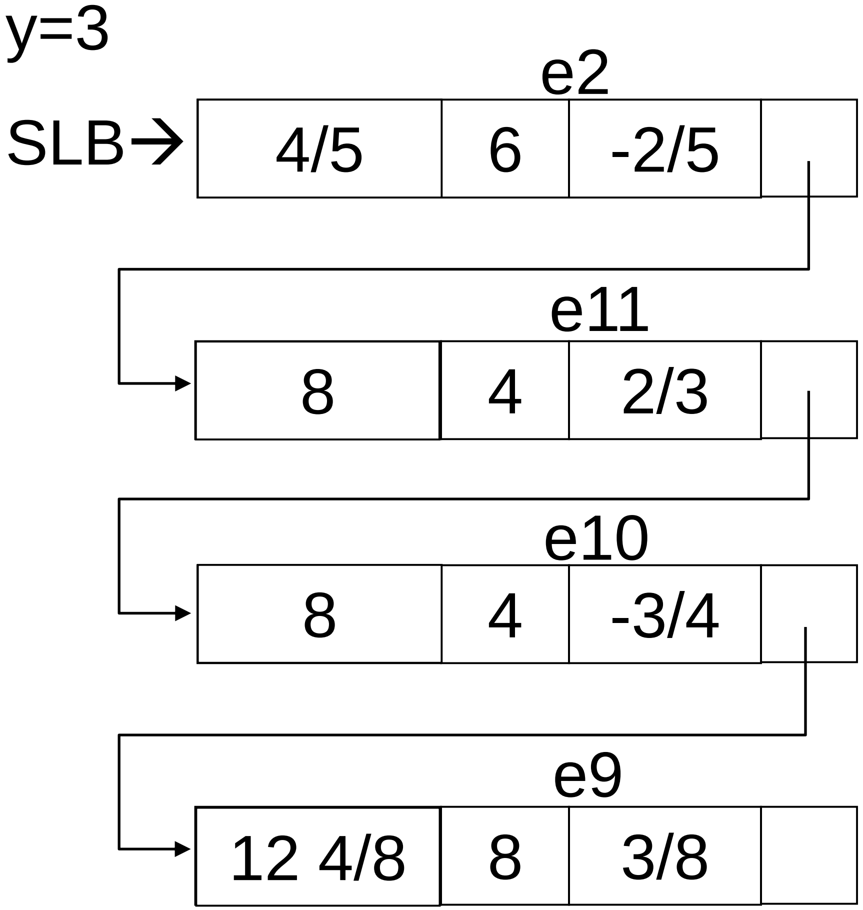
y=2

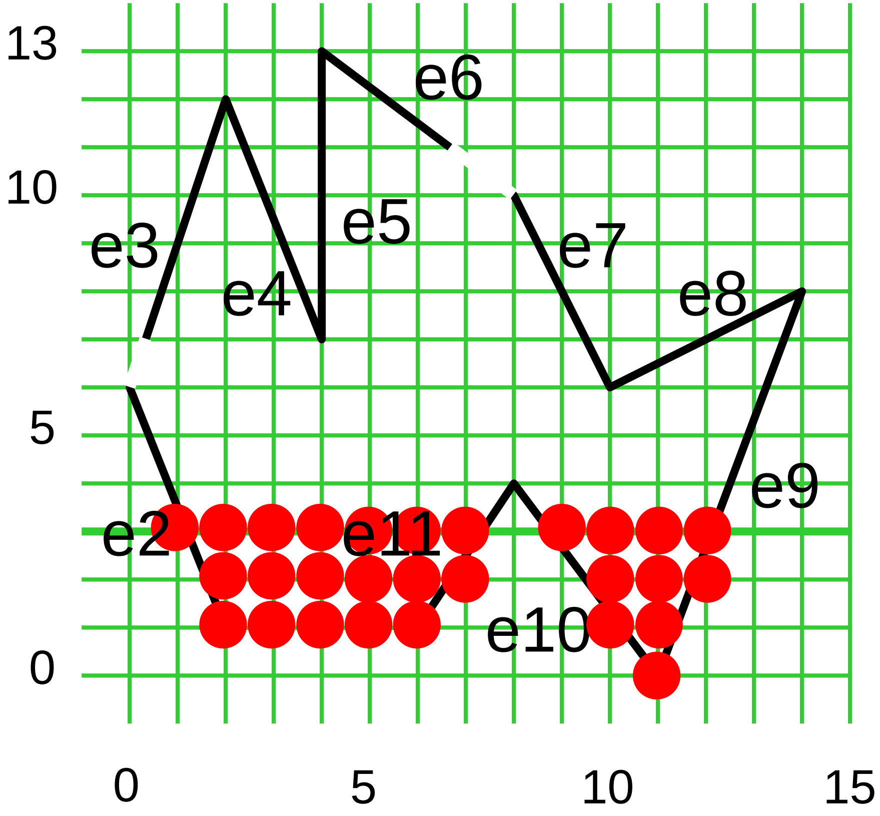
SLB→

Edge			
e2	1 1/5	6	-2/5
e11	7 1/3	4	2/3
e10	8 3/4	4	-3/4
e9	12 1/8	8	3/8

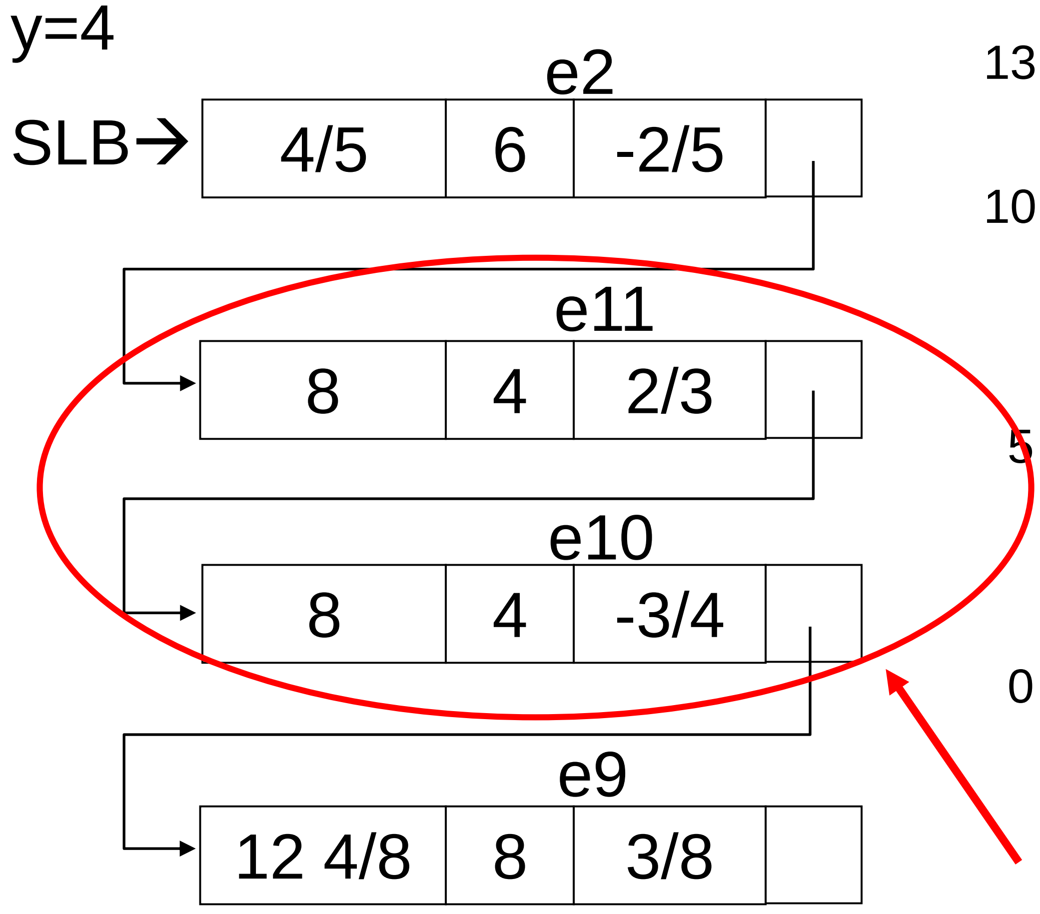


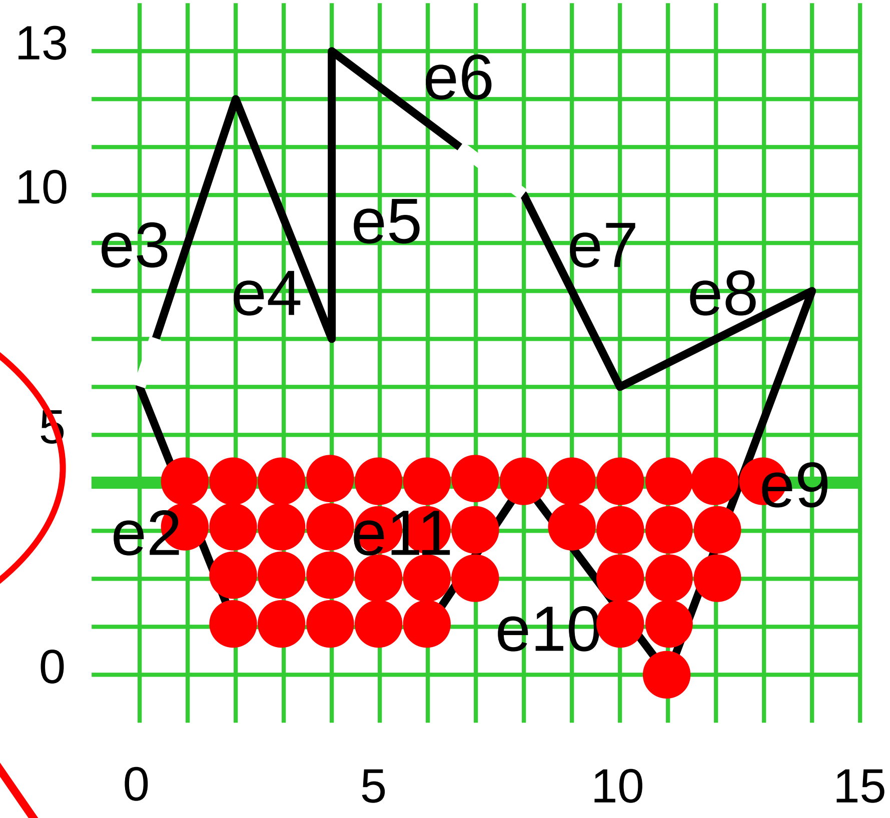
Running the Algorithm





Running the Algorithm

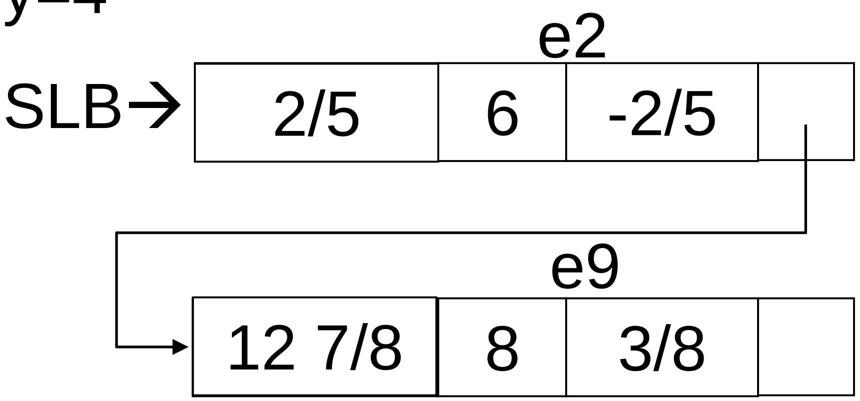




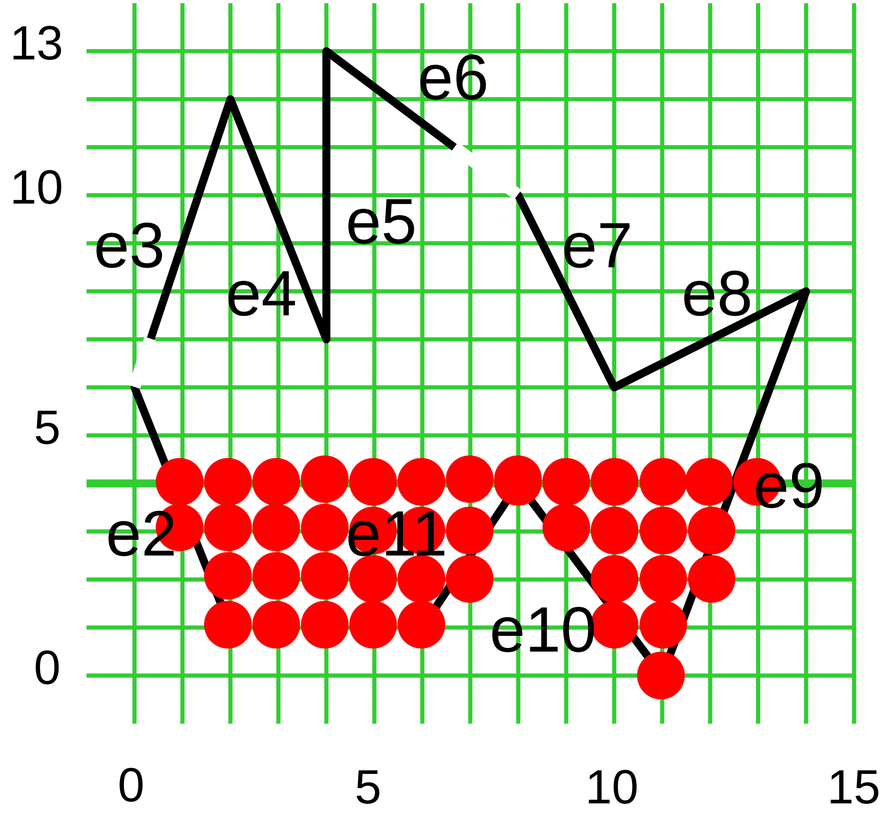
Remove these edges.

Running the Algorithm

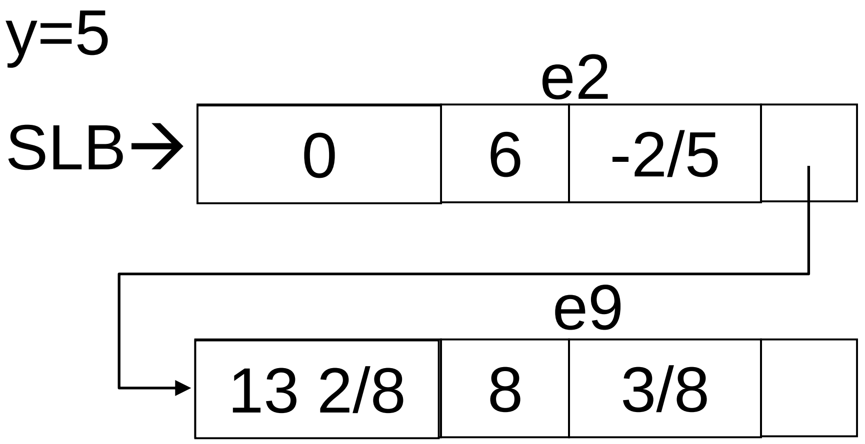
y=4

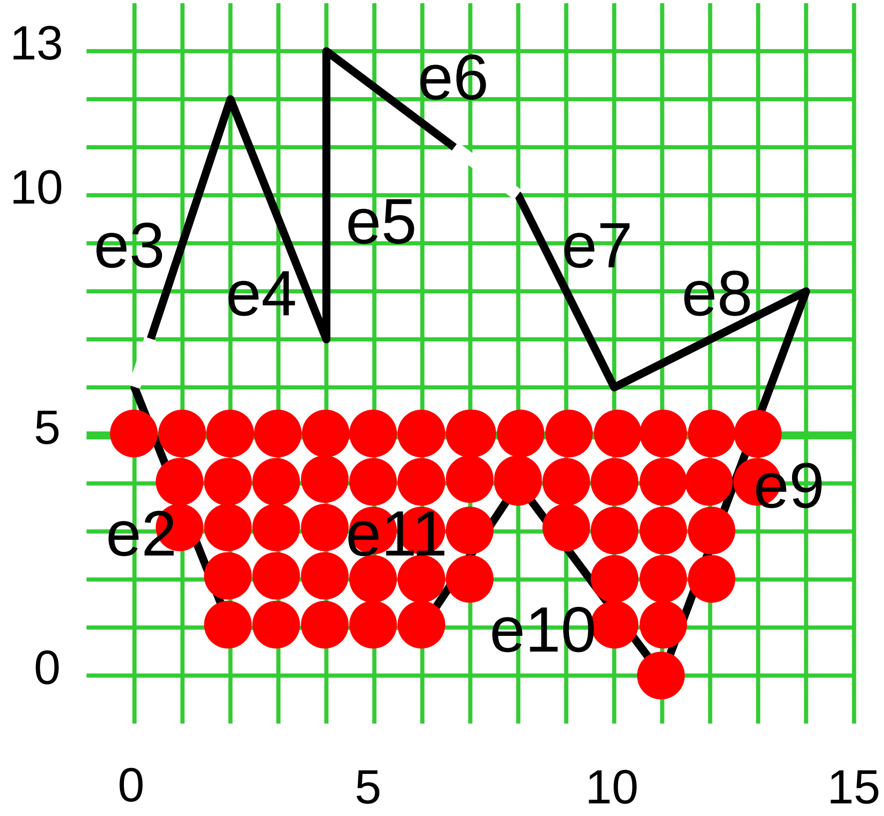


e11 and e10 are removed.



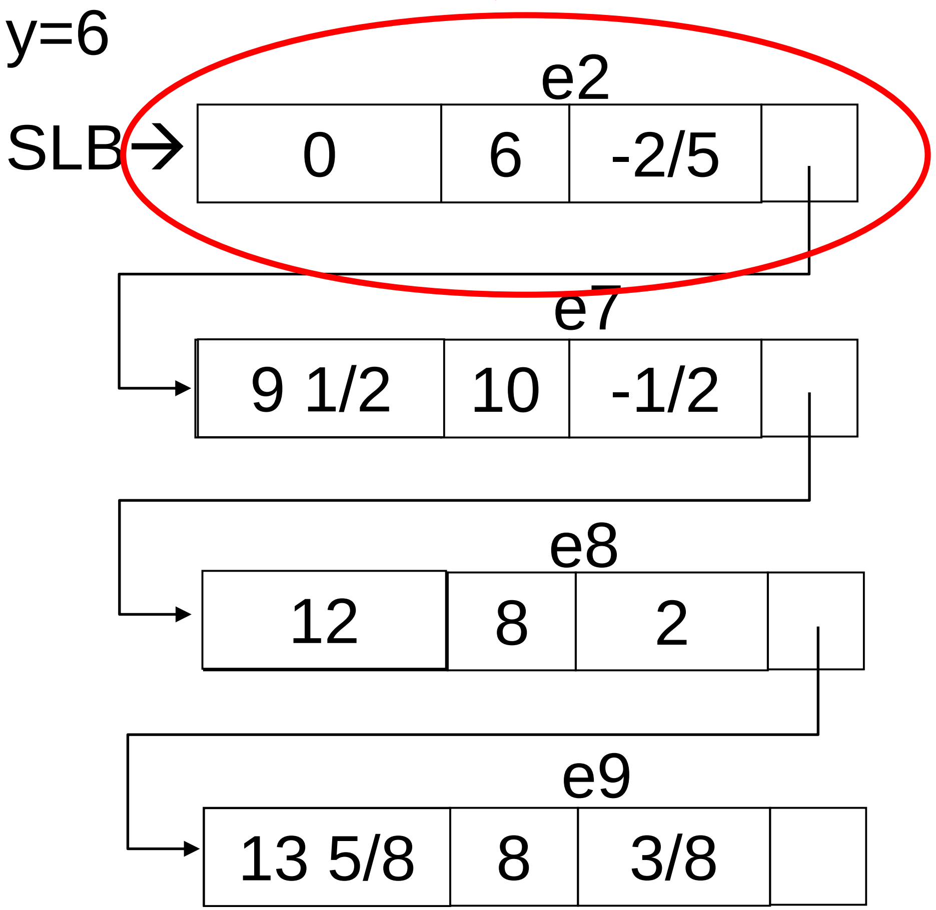
Running the Algorithm

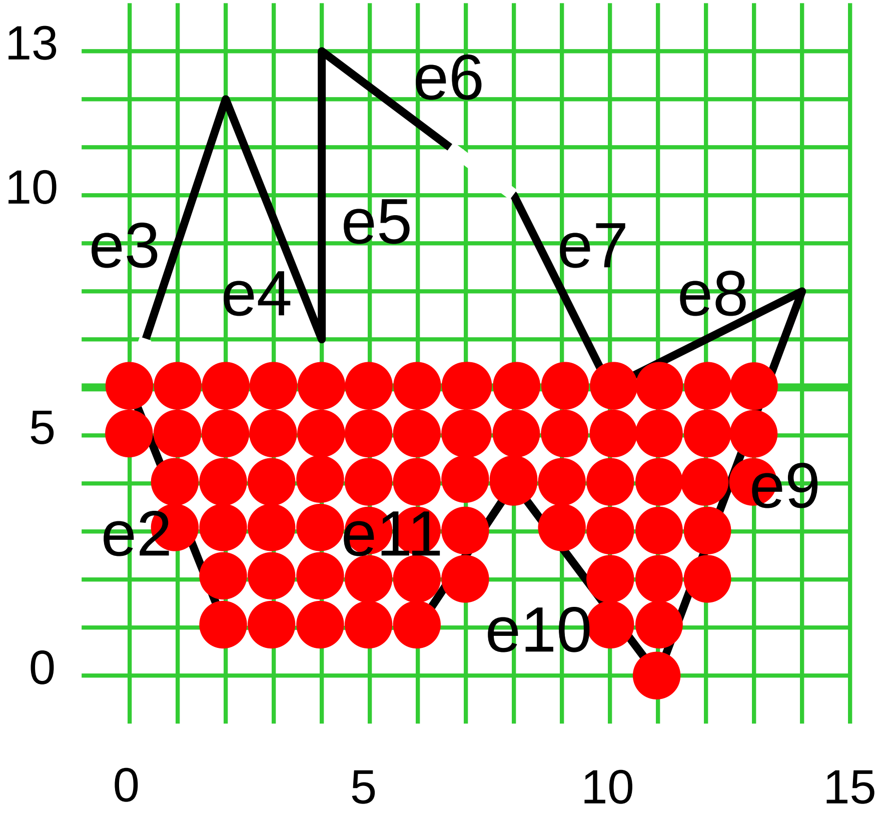




Running the Algorithm

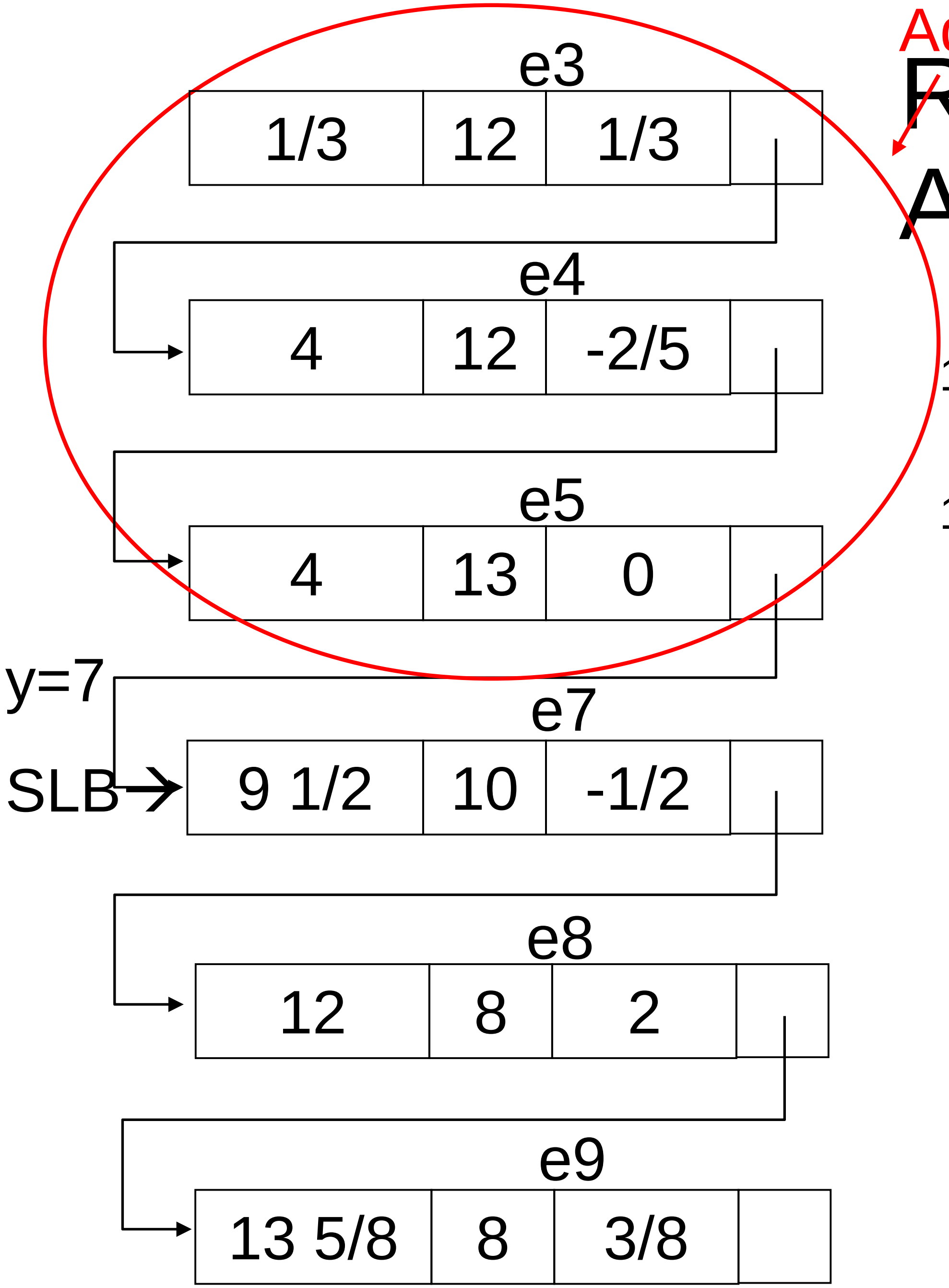
Remove this edge.

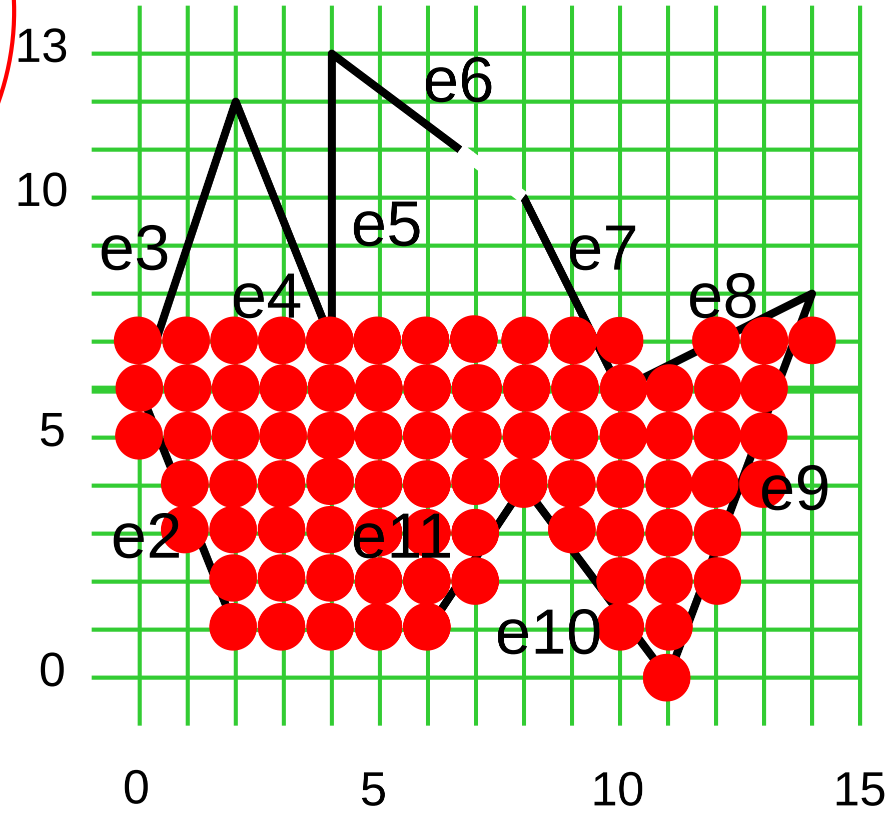




Add these edges.

Running the Algorithm





Thanks