



# DMET 502/701

# Computer Graphics

---

## **2D Graphics**

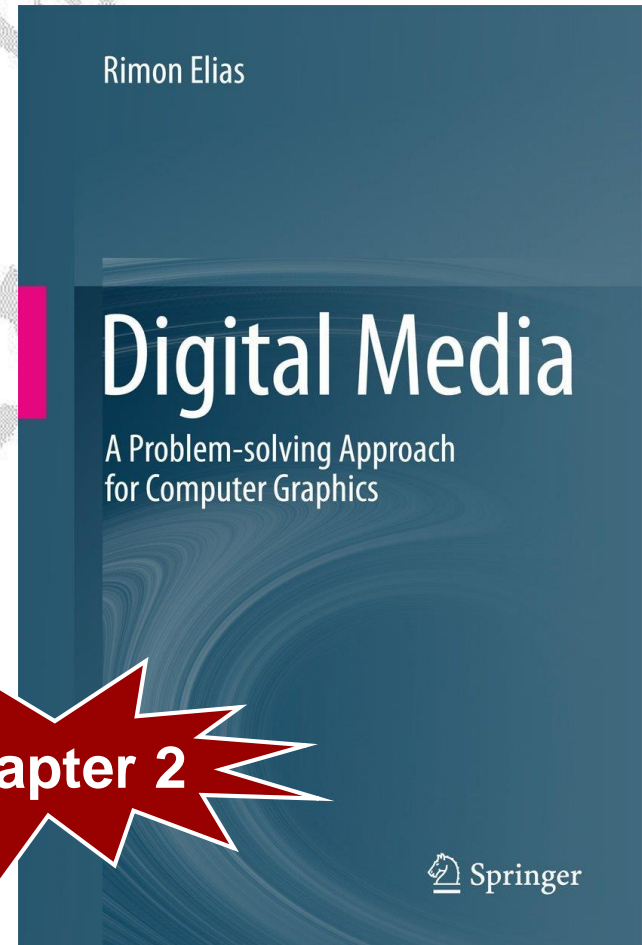
**Assoc. Prof. Dr. Rimon Elias**





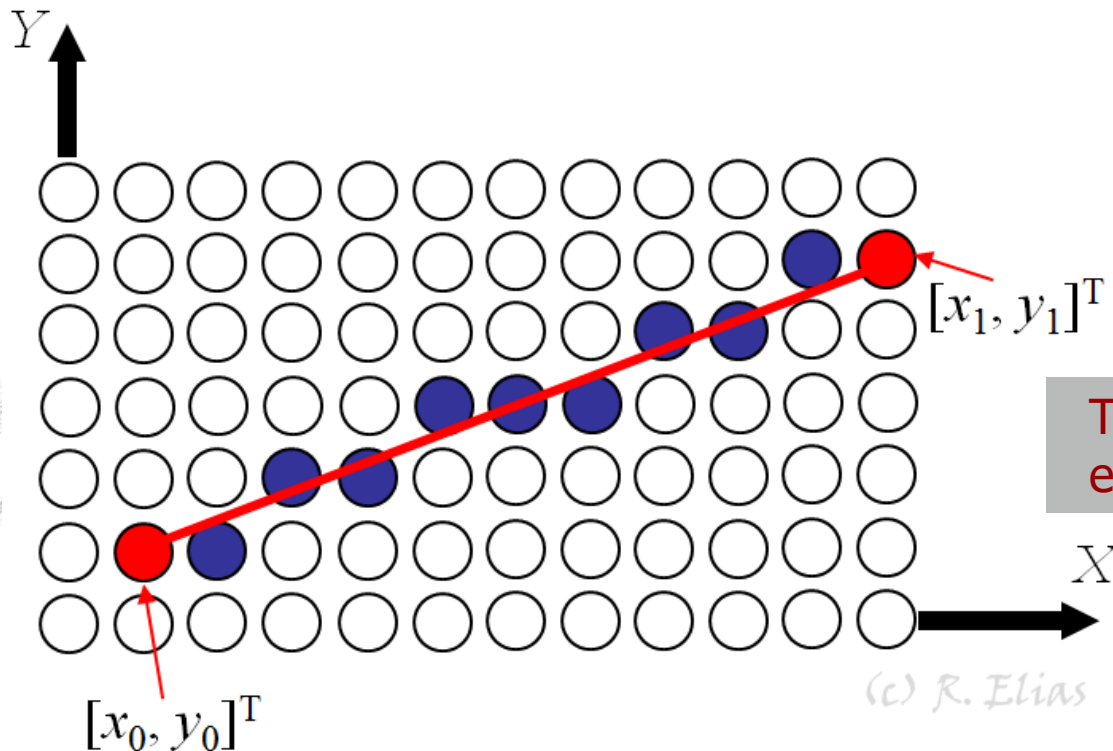
# Contents

- 2D graphics
  - Drawing lines
    - Bresenham's algorithm
    - Midpoint algorithm
  - Drawing circles
    - 2-way symmetry
    - 4-way symmetry
    - 8-way symmetry
    - Midpoint algorithm
  - Polygons
  - Line clipping
  - Polygon clipping



# Drawing Lines

- Drawing a line between two pixels  $[x_0, y_0]^T$  and  $[x_1, y_1]^T$  is done by intensifying (or turning on) pixels along the path of this line.



The blue pixels are not exactly on the line path!

Note that this line has a slope  $< 1$ . Why?



# Drawing Lines: Line Equation

- The general formula for the line between the two points is given by:

$$y - y_0 = \frac{y_1 - y_0}{x_1 - x_0} (x - x_0)$$

- Hence,

Explicit form

$$y = \frac{y_1 - y_0}{x_1 - x_0} (x - x_0) + y_0$$

Slope  $m$



# Drawing Lines: DDA Algorithm

- A simple and direct way to draw the line is performed by looping or iterating through the previous equation, incrementing the value of  $x$  from  $x_0$  to  $x_1$  (i.e., moving from one column to the next) and obtaining the corresponding  $y$  to plot the pixel  $[x, y]^T$ .
- Two observations:
  1. The  $x$ -value is an integer value as it is a column number; however, the resulting  $y$ -value could be a floating-point number. In this case,  $y$  must be rounded to the nearest integer.
  2. At each iteration, the slope  $m$  is re-calculated although it is a constant number for the entire line. Thus, the slope  $m$  can be pre-calculated before the loop.



# Drawing Lines: DDA Algorithm

- Using the previous equation at iteration  $i$

$$y_i = m(x_i - x_0) + y_0 = mx_i + \underbrace{y_0 + mx_0}_B$$

$y$ -intercept  
(constant for  
the entire line)

- At iteration  $i + 1$

$$y_{i+1} = mx_{i+1} + B = m(x_i + 1) + B = \underbrace{mx_i + B}_{y_i} + m = y_i + m$$

- This means that subsequent  $y$ -values can be calculated by adding the value of the slope  $m$  to the previous  $y$  at each iteration.



# Drawing Lines: DDA Algorithm

- We can write the **digital differential analyzer** (DDA) algorithm.

## Algorithm 2.1

Input:  $x_0, y_0, x_1, y_1$

1:  $m = \frac{\Delta y}{\Delta x} = \frac{y_1 - y_0}{x_1 - x_0}$

2:  $y = y_0$

3: for  $(x = x_0 \text{ to } x_1)$  do

4: Plot  $[x, [y + 0.5]]^T$

5:  $y = y + m$

6: end for

end

This is good when  $|m| \leq 1$ ,  
what modifications should be  
done when  $|m| > 1$ ?

The constant slope  
 $m$  is pre-calculated  
before the loop

$y$  is rounded to the  
nearest integer





# DDA Algorithm: An Example

- Example:** Using the DDA line drawing algorithm, determine the pixels along a line segment that goes from  $[3,4]^T$  to  $[8,6]^T$ .

- Solution:**

$$m = \frac{\Delta y}{\Delta x} = \frac{y_1 - y_0}{x_1 - x_0} = \frac{6 - 4}{8 - 3} = 0.4.$$

## Algorithm 2.1

**Input:**  $x_0, y_0, x_1, y_1$

- 1:  $m = \frac{\Delta y}{\Delta x} = \frac{y_1 - y_0}{x_1 - x_0}$
- 2:  $y = y_0$
- 3: **for** ( $x = x_0$  to  $x_1$ ) **do**
- 4:     Plot  $[x, \lfloor y + 0.5 \rfloor]^T$
- 5:      $y = y + m$
- 6: **end for**

**end** (c) 2022, Dr. R. Elias

Line			
3	4	4	5
$x$	$y$ (rounded)	Plot	$y$ (exact value)
3	4	$[3, 4]^T$	$4.0 + 0.4 = 4.4$
4	4	$[4, 4]^T$	$4.4 + 0.4 = 4.8$
5	5	$[5, 5]^T$	$4.8 + 0.4 = 5.2$
6	5	$[6, 5]^T$	$5.2 + 0.4 = 5.6$
7	6	$[7, 6]^T$	$5.6 + 0.4 = 6.0$
8	6	$[8, 6]^T$	$6.0 + 0.4 = 6.4$

Line pixels





# Drawing Lines: Modified DDA Algorithm

## ■ Notice that

1. a difference could arise between the accurate value of  $y$  and its rounded value to be used for plotting; and
2. the  $y$ -value is incremented by 1 only if the fraction included in the accurate value of  $y$  is greater than or equal to 0.5; otherwise,  $y$  remains unchanged.

This suggests  
a modification!



# Drawing Lines: Modified DDA Algorithm

## Modification

1. Each time  $x$  is incremented, the slope  $m$  can be added to an *error* value. Line 6
2. If the new *error* is greater than or equal to 0.5 (i.e., the line gets closer to the next  $y$ -value),  $y$  is incremented by 1 while *error* is decremented by 1. Lines 7-10

### Algorithm 2.3 Modified DDA algorithm

**Input:**  $x_0, y_0, x_1, y_1$

```
1:  $m = \frac{\Delta y}{\Delta x} = \frac{y_1 - y_0}{x_1 - x_0}$ 
2:  $y = y_0$ 
3:  $error = 0$ 
4: for ( $x = x_0$  to  $x_1$ ) do
5:   Plot  $[x, y]^T$ 
6:   {  $error = error + m$ 
7:     if ( $error \geq 0.5$ ) then
8:        $y = y + 1$ 
9:        $error = error - 1$ 
10:    end if
11: end for
```

**end**

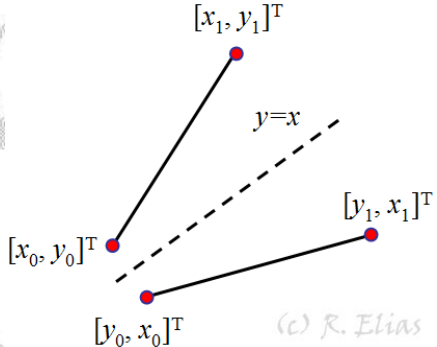
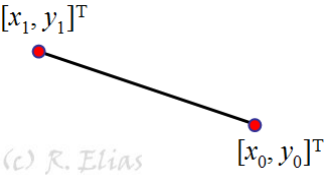
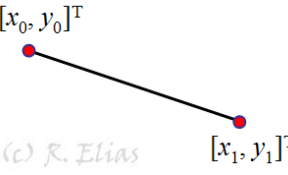
*error* indicates the vertical distance between the rounded  $y$ -value and the exact  $y$ -value.



# Drawing Lines: Bresenham's Algorithm

**This previous algorithm is a special case!**

- Consider the following cases and their solutions to generalize.

Case	Solution
<p><b>1</b> If the slope is <math>&gt; 1</math> (i.e., a steep line)</p>  <p>(c) R. Elias</p>	<p>A steep line can be reflected across the line <math>y=x</math> to obtain a line with a small slope. So switch the <math>x</math> and <math>y</math> variables. Switch the parameters to plot.</p>
<p><b>2</b> If the line slopes upwards but heads in the opposite direction</p>  <p>(c) R. Elias</p>	<p>Swap the initial points if <math>x_0 &gt; x_1</math></p>
<p><b>3</b> If the line goes down</p>  <p>(c) R. Elias</p>	<p>Check if <math>y_0 &gt; y_1</math>; if so, step <math>y</math> by <math>-1</math> instead of <math>1</math></p>



# Bresenham's Algorithm (floating-point version)

## Algorithm 2.4

Input:  $x_0, y_0, x_1, y_1$

```
1:  $\Delta x = x_1 - x_0$ 
2:  $\Delta y = y_1 - y_0$ 
3:  $steep = |\frac{\Delta y}{\Delta x}| > 1$ 
4: if ( $steep = TRUE$ ) then
5:   swap ( $x_0, y_0$ )
6:   swap ( $x_1, y_1$ )
7: end if
8:
9: if ( $x_0 > x_1$ ) then
10:  swap ( $x_0, x_1$ )
11:  swap ( $y_0, y_1$ )
12: end if
13:
14: if ( $y_0 > y_1$ ) then
15:    $\delta y = -1$ 
16: else
17:    $\delta y = 1$ 
18: end if
19:
20:  $m = \frac{|\Delta y|}{\Delta x} = \frac{|y_1 - y_0|}{x_1 - x_0}$ 
21:  $y = y_0$ 
22:  $error = 0$ 
23:
24: for ( $x = x_0$  to  $x_1$ ) do
25:   if ( $steep = TRUE$ ) then
26:     Plot [ $y, x$ ]T
27:   else
28:     Plot [ $x, y$ ]T
29:   end if
30:    $error = error + m$ 
31:   if ( $error \geq 0.5$ ) then
32:      $y = y + \delta y$ 
33:      $error = error - 1$ 
34:   end if
35: end for
end
```



# Bresenham's Algorithm (integer version)

- The main source of problem with the previous algorithm is that it works with floating-point numbers (e.g.,  $m$  and  $error$ ), which slows the process down and may result in error accumulation.
- Working with integer numbers will be much faster and more accurate.
- Switching to integers can be achieved easily by
  - multiplying  $m$  and  $error$  by the denominator of the slope; i.e.,  $\Delta x$ ; and
  - doubling both sides of the condition ( $error \geq 0.5$ ) to get rid of the fraction.



# Bresenham's Algorithm (integer version)

## Algorithm 2.5 *Bresenham's algorithm*

**Input:**  $x_0, y_0, x_1, y_1$

```
1:  $steep = |y_1 - y_0| > |x_1 - x_0|$ 
2: if ( $steep = TRUE$ ) then
3:   swap ( $x_0, y_0$ )
4:   swap ( $x_1, y_1$ )
5: end if
6:
7: if ( $x_0 > x_1$ ) then
8:   swap ( $x_0, x_1$ )
9:   swap ( $y_0, y_1$ )
10: end if
11:
```

```
12: if ( $y_0 > y_1$ ) then
13:    $\delta y = -1$ 
14: else
15:    $\delta y = 1$ 
16: end if
17:
18:  $\Delta x = x_1 - x_0$ 
19:  $\Delta y = |y_1 - y_0|$ 
20:  $y = y_0$ 
21:  $error = 0$ 
22:
```

```
23: for ( $x = x_0$  to  $x_1$ ) do
24:   if ( $steep = TRUE$ ) then
25:     Plot  $[y, x]^T$ 
26:   else
27:     Plot  $[x, y]^T$ 
28:   end if
29:    $error = error + \Delta y$ 
30:   if ( $2 \times error \geq \Delta x$ ) then
31:      $y = y + \delta y$ 
32:      $error = error - \Delta x$ 
33:   end if
34: end for
end
```

*error* indicates old  
error (Alg. 2.4)  
multiplied by  $\Delta x$ .



# Bresenham's Algorithm: An Example

- **Example:** You are asked to draw a line segment between the points  $[1,1]^T$  and  $[4,3]^T$ . Use **Bresenham's line drawing algorithm** to specify the locations of pixels that should approximate the line.

- **Solution:**

- The line is not steep as  $|y_1 - y_0| < |x_1 - x_0|$ .
- The  $y$ -step is 1 as  $y_0 < y_1$ .
- We have

$$\begin{aligned}\delta y &= 1, \\ \Delta x = x_1 - x_0 &= 4 - 1 = 3, \\ \Delta y = y_1 - y_0 &= 3 - 1 = 2, \\ y &= 1, \\ error &= 0.\end{aligned}$$





# Bresenham's Algorithm: An Example

- The following table shows the loop along the x-direction from 1 to 4 (i.e., the loop that starts at Line 23 in Alg. 2.5):

```

23: for ( $x = x_0$  to  $x_1$ ) do
24:   if ( $steep = TRUE$ ) then
25:     Plot  $[y, x]^T$ 
26:   else
27:     Plot  $[x, y]^T$ 
28:   end if
29:    $error = error + \Delta y$ 
30:   if ( $2 \times error \geq \Delta x$ ) then
31:      $y = y + \delta y$ 
32:      $error = error - \Delta x$ 
33:   end if
34: end for
  
```

Line				
23	27	29	31	32
$x$	Plot	$error$	$y$	$error$
1	$[1, 1]^T$	$0 + 2 = 2$	$1 + 1 = 2$	$2 - 3 = -1$
2	$[2, 2]^T$	$-1 + 2 = 1$		
3	$[3, 2]^T$	$1 + 2 = 3$	$2 + 1 = 3$	$3 - 3 = 0$
4	$[4, 3]^T$	$0 + 2 = 2$	$3 + 1 = 4$	$2 - 3 = -1$

..... Line pixels

- The line can be approximated as:  $[1, 1]^T$ ,  $[2, 2]^T$ ,  $[3, 2]^T$  and  $[4, 3]^T$ .



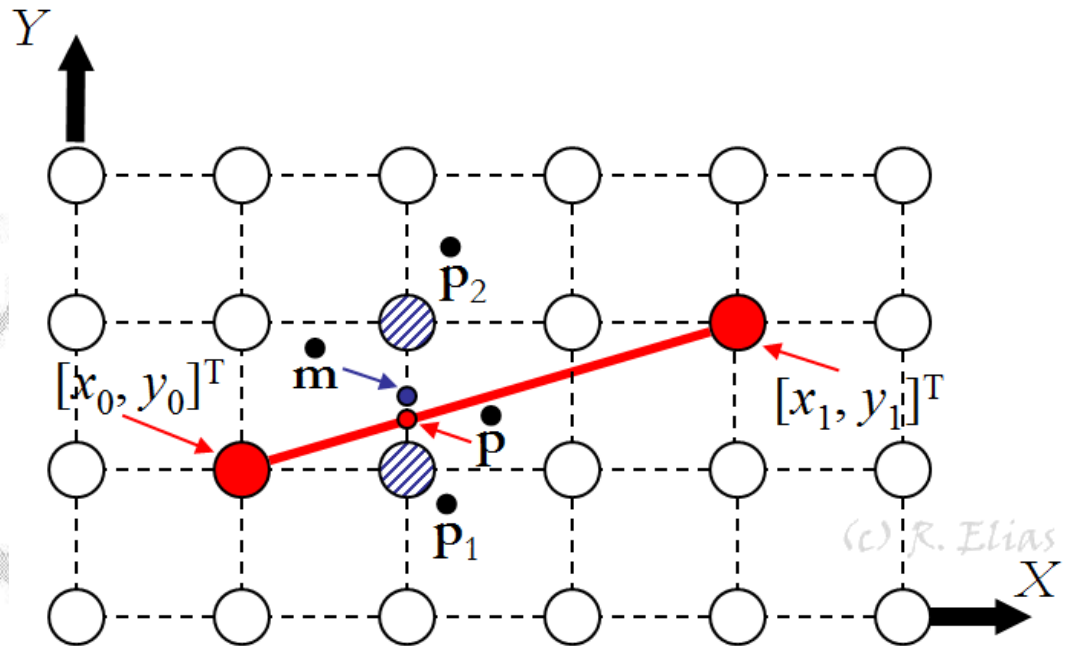
# Drawing Lines: Midpoint Algorithm

- The **midpoint algorithm** can be used instead of Bresenham's algorithm to approximate straight lines.
- A difference does exist.
  - In Bresenham's algorithm, the smallest  $y$ -difference between the actual accurate  $y$ -value and the rounded integer  $y$ -value is used to pick up the nearest pixel to approximate the line.
  - On the other hand, in the midpoint technique, we determine which side of the linear equation the midpoint between pixels lies.



# Drawing Lines: Midpoint Algorithm

- We use a loop that increments  $x$  as done before.
- At  $x_0+1$ , the exact intersection happens at point  $\dot{p}$ .



- Thus, there will be two choices for the pixels to be picked up;  $\dot{p}_1$  and  $\dot{p}_2$ .



# Drawing Circles

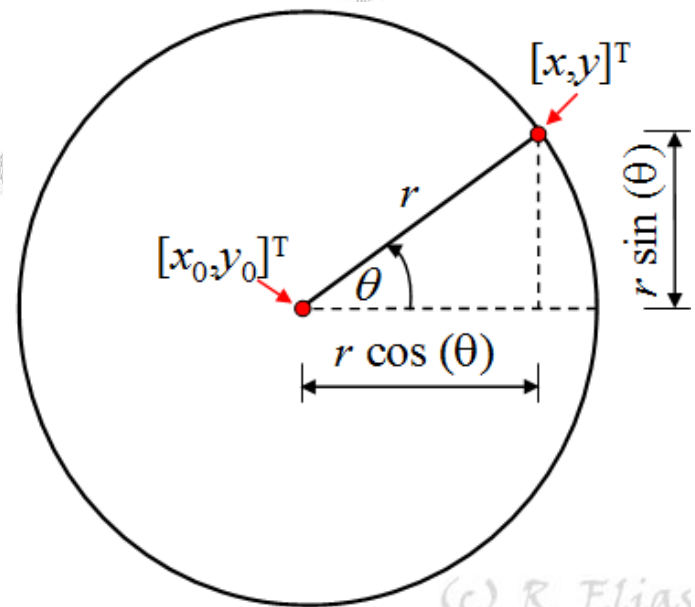
- A pixel  $[x, y]^T$  on the circumference of a circle can be estimated if the center  $[x_0, y_0]^T$ , the radius  $r$  and the angle  $\theta$  are known.

Parametric  
form

$$x = x_0 + r \cos(\theta)$$

$$y = y_0 + r \sin(\theta)$$

- You need to iterate through different  $\theta$  angles ( $0 \leq \theta < 2\pi$ ) to draw the circle.



(c) R. Elias





# Drawing Circles

---

- **Problems with the previous approach:**
  - Working with trigonometric functions could be time-consuming.
  - If  $\theta$  increment is too small, calculations will be very slow.
  - If  $\theta$  increment is too large, pixels may be skipped.
  - If the radius  $r$  is too large, pixels may be skipped.



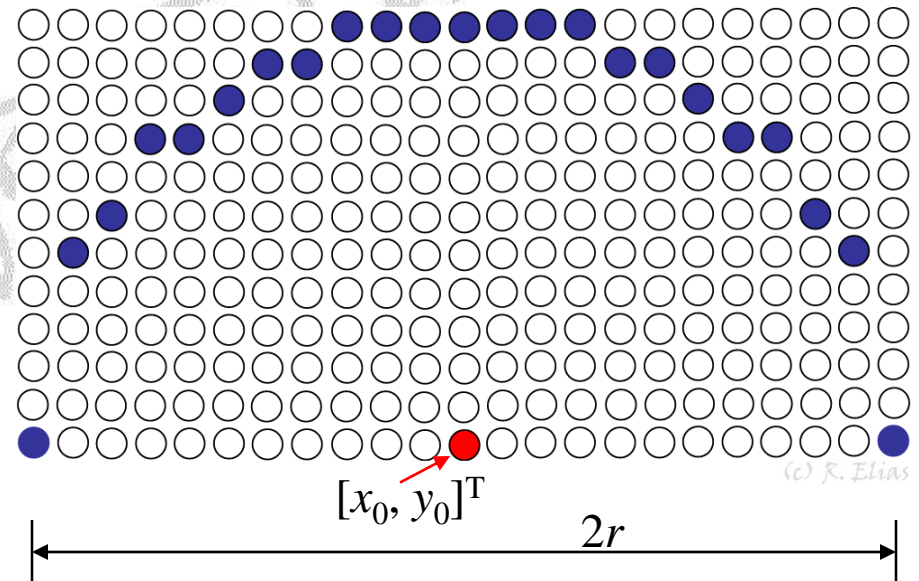
# Drawing Circles: 2-Way Symmetry

- A circle equation can be expressed as:  $(x - x_0)^2 + (y - y_0)^2 = r^2$
- Solve  $y$  in terms of  $x$  to get:

Explicit form

$$y = y_0 \pm \sqrt{r^2 - (x - x_0)^2}$$

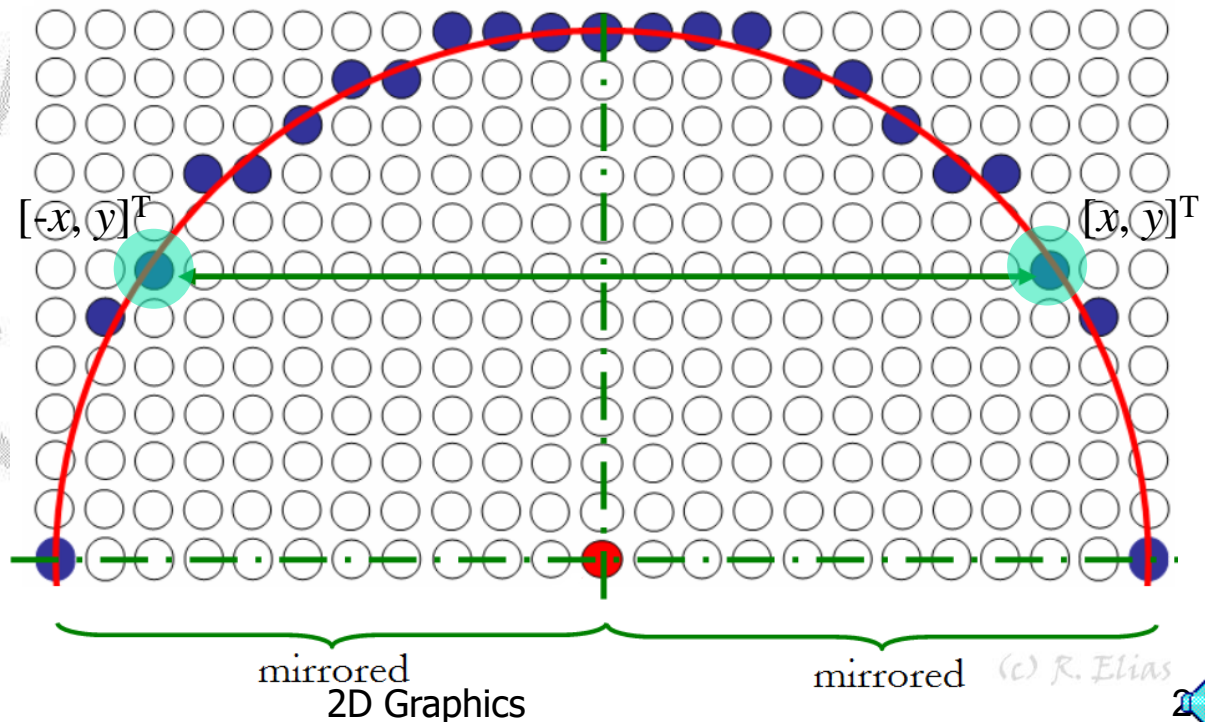
- Here we will use a loop that goes  $2r$  steps along the  $x$ -direction.
- We can use the positive and negative values of the square root to come up with a faster answer.
- Notice the discontinuities!





# Drawing Circles: 4-Way Symmetry

- The symmetry of the circle implies that a circle may be split into 4 quadrants.
- By drawing only one quadrant (going from the center to  $r$ ), the remaining 3 quadrants can be mirrored/reflected.
- When center is at the origin,  $[x, y]^T$  is reflected to
  - $[-x, y]^T$
  - $[x, -y]^T$
  - $[-x, -y]^T$





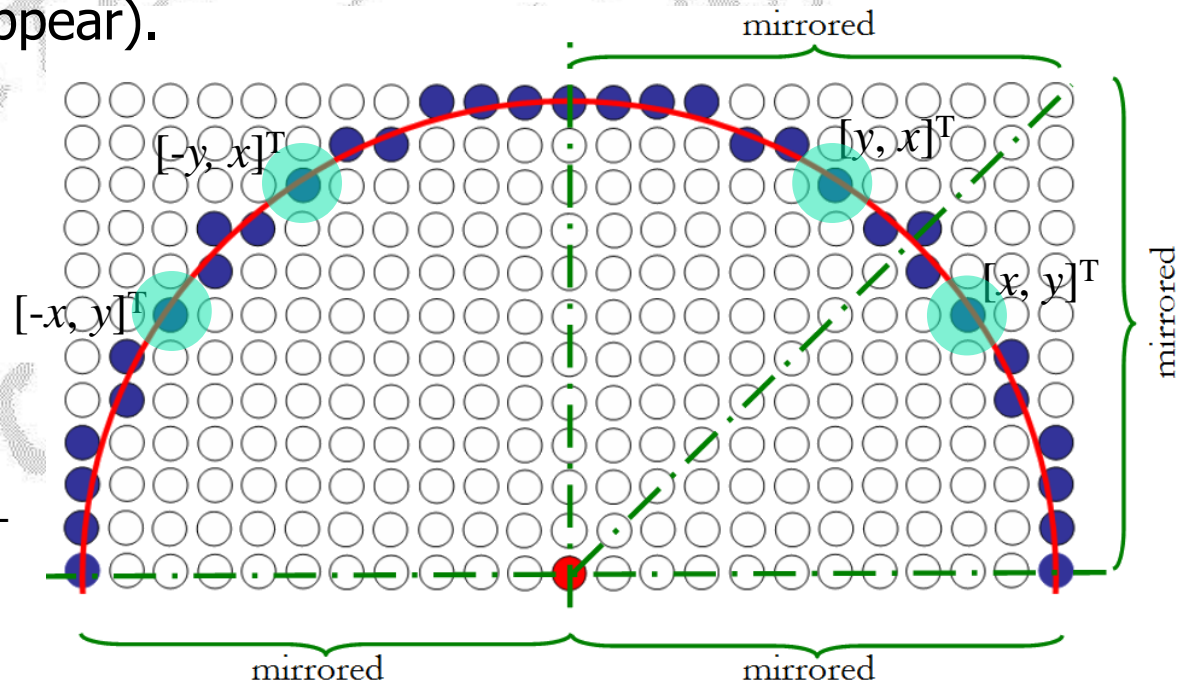
# Drawing Circles: 8-Way Symmetry

Furthermore, a circle may be split into 8 octants.

- By drawing only one octant, the remaining 7 octants can be mirrored/reflected.
- 8-way symmetry results in a better circle (notice that the discontinuities disappear).

- When center is at the origin,  $[x, y]^T$  is reflected to

- $[y, x]^T$
- $[-x, y]^T, [-y, x]^T$
- $[x, -y]^T, [y, -x]^T$
- $[-x, -y]^T, [-y, -x]^T$



# Drawing Circles: 8-Way Symmetry

Assume that the center is at  $[0,0]^T$ . When plotting, the actual center coordinates; i.e.,  $[x_0, y_0]^T$  are added to get the actual locations of the circumferential pixels.

## Algorithm 2.10

**Input:**  $x_0, y_0, r$

```
1: Plot  $[x_0, y_0 + r]^T$ 
2: Plot  $[x_0, y_0 - r]^T$ 
3: Plot  $[x_0 + r, y_0]^T$ 
4: Plot  $[x_0 - r, y_0]^T$ 
5:  $x = 1$ 
6:  $y = \lfloor \sqrt{r^2 - x^2} + 0.5 \rfloor$ 
8: while  $(x < y)$  do
9:   Plot  $[x_0 + x, y_0 + y]^T$ 
10:  Plot  $[x_0 + x, y_0 - y]^T$ 
11:  Plot  $[x_0 - x, y_0 + y]^T$ 
12:  Plot  $[x_0 - x, y_0 - y]^T$ 
13:  Plot  $[x_0 + y, y_0 + x]^T$ 
14:  Plot  $[x_0 + y, y_0 - x]^T$ 
15:  Plot  $[x_0 - y, y_0 + x]^T$ 
16:  Plot  $[x_0 - y, y_0 - x]^T$ 
17:   $x = x + 1$ 
18:   $y = \lfloor \sqrt{r^2 - x^2} + 0.5 \rfloor$ 
19: end while
20:
21: if  $x = y$  then
22:   Plot  $[x_0 + x, y_0 + y]^T$ 
23:   Plot  $[x_0 + x, y_0 - y]^T$ 
24:   Plot  $[x_0 - x, y_0 + y]^T$ 
25:   Plot  $[x_0 - x, y_0 - y]^T$ 
26: end if
end
```



# Drawing Circles: An Example

- **Example:** A circle having a radius of 5 pixels and centered at  $[3,4]^T$  is to be drawn on a computer screen. Use the **8-way symmetry algorithm** to determine what pixels should constitute the circle.
- **Solution:** The start and endpoints of each quadrant are
  - $[3,9]^T$ ,  $[3,-1]^T$ ,  $[8,4]^T$  and  $[-2,4]^T$ .



# Drawing Circles: An Example

- The rest of the points are listed in the following table:

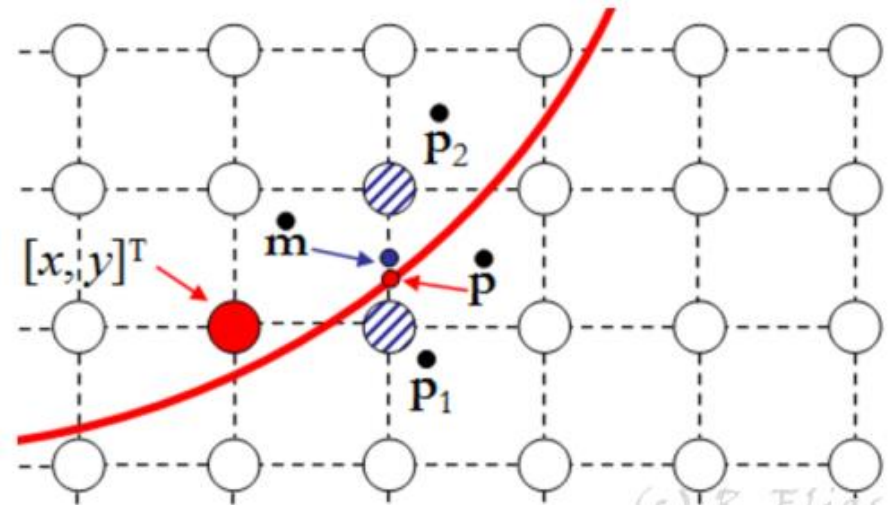
Line									
5/17	6/18	9	10	11	12	13	14	15	16
$x$	$y$	Plot	Plot	Plot	Plot	Plot	Plot	Plot	Plot
1	5	$[4, 9]^T$	$[4, -1]^T$	$[2, 9]^T$	$[2, -1]^T$	$[8, 5]^T$	$[8, 3]^T$	$[-2, 5]^T$	$[-2, 3]^T$
2	5	$[5, 9]^T$	$[5, -1]^T$	$[1, 9]^T$	$[1, -1]^T$	$[8, 6]^T$	$[8, 2]^T$	$[-2, 6]^T$	$[-2, 2]^T$
3	4	$[6, 8]^T$	$[6, 0]^T$	$[0, 8]^T$	$[0, 0]^T$	$[7, 7]^T$	$[7, 1]^T$	$[-1, 7]^T$	$[-1, 1]^T$
4	3	no iteration as $x > y$							

- After neglecting all points with negative coordinates, the points considered are  $[3, 9]^T$ ,  $[8, 4]^T$ ,  $[4, 9]^T$ ,  $[2, 9]^T$ ,  $[8, 5]^T$ ,  $[8, 3]^T$ ,  $[5, 9]^T$ ,  $[1, 9]^T$ ,  $[8, 6]^T$ ,  $[8, 2]^T$ ,  $[6, 8]^T$ ,  $[6, 0]^T$ ,  $[0, 8]^T$ ,  $[0, 0]^T$ ,  $[7, 7]^T$  and  $[7, 1]^T$ .



# Drawing Circles: Midpoint Algorithm

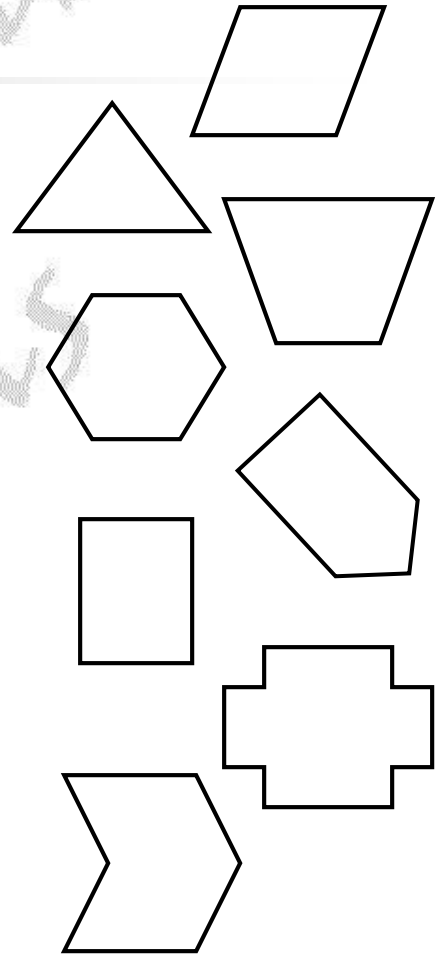
- The midpoint algorithm used to draw lines can be modified to draw circles.
- Similar to the *8-way symmetry* algorithm, only one octant is considered. The rest of the circle is obtained by symmetry as done before.
- Similar to the *midpoint* technique used to draw lines, we determine on which side of the circle equation the midpoint between pixels lies.





# Polygons

- A **2D polygon** is a closed planar path composed of a number of sequential straight line segments.
- Each line segment is called a *side* or an *edge*.
- The intersections between line segments are called *vertices*.
- A 2D polygon is often stored as a sequence of vertex coordinates.
- The end vertex of the last edge is the start vertex of the first edge.

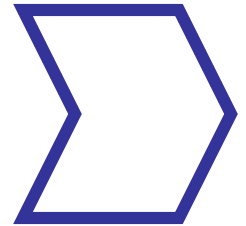




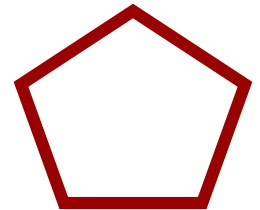
# Polygons:

## Convexity versus Concavity

- A polygon may be *convex* or *concave*.
- A polygon is convex if the line connecting any two interior points is included completely in the interior of the polygon.
- Each interior angle in a **convex polygon** is  $\leq 180^\circ$ .



Concave polygon



Convex polygon





# How to Decide if a Polygon is Convex/Concave?

1

## Cross Product

- For each edge  $\mathbf{e}_i$ :
  - Define a vector connecting the vertices:  $\mathbf{e}_i = \dot{\mathbf{v}}_{i+1} - \dot{\mathbf{v}}_i$ .
  - Compute the 2D cross products along adjacent edges.

$$\begin{aligned}\mathbf{e}_{i-1} \times \mathbf{e}_i &= [\dot{\mathbf{v}}_i - \dot{\mathbf{v}}_{i-1}] \times [\dot{\mathbf{v}}_{i+1} - \dot{\mathbf{v}}_i] \\ &= \begin{bmatrix} x_i - x_{i-1} \\ y_i - y_{i-1} \end{bmatrix} \times \begin{bmatrix} x_{i+1} - x_i \\ y_{i+1} - y_i \end{bmatrix} \\ &= (x_i - x_{i-1})(y_{i+1} - y_i) - (y_i - y_{i-1})(x_{i+1} - x_i).\end{aligned}$$

- If the signs of the cross products are not all the same, then the polygon is concave.



# How to Decide if a Polygon is Convex/Concave?

2

## Linear Equations

- For every edge of the polygon in turn:
  - Extend the edge into a line.
  - Test to see if any two of the vertices of the polygon lie on opposite sides of the line.
  - If so, then the polygon is concave.

**How to implement  
this mathematically?**



# Convex/Concave Polygons: An Example

- **Example:** At least two different methods may be used to decide whether or not a 2D polygon is concave or convex. Apply each of them to the 2D polygon specified by the vertices  $[0,0]^T$ ,  $[5,0]^T$ ,  $[-1,5]^T$  and  $[-1,-5]^T$ . Based on your calculations, determine whether this polygon is concave or convex.



# Convex/Concave Polygons: An Example

## Solution:

→  $\dot{\mathbf{v}}_{i-1} = [-1, -5]^T$ ,  $\dot{\mathbf{v}}_i = [0, 0]^T$  and  $\dot{\mathbf{v}}_{i+1} = [5, 0]^T$

$$\begin{aligned} \mathbf{e}_{i-1} \times \mathbf{e}_i &= [\dot{\mathbf{v}}_i - \dot{\mathbf{v}}_{i-1}] \times [\dot{\mathbf{v}}_{i+1} - \dot{\mathbf{v}}_i] \\ &= (x_i - x_{i-1})(y_{i+1} - y_i) - (y_i - y_{i-1})(x_{i+1} - x_i) \\ &= (0 - (-1))(0 - 0) - (0 - (-5))(5 - 0) = -25 \Rightarrow -ve. \end{aligned}$$

→  $\dot{\mathbf{v}}_{i-1} = [0, 0]^T$ ,  $\dot{\mathbf{v}}_i = [5, 0]^T$  and  $\dot{\mathbf{v}}_{i+1} = [-1, 5]^T$

$$\begin{aligned} \mathbf{e}_{i-1} \times \mathbf{e}_i &= [\dot{\mathbf{v}}_i - \dot{\mathbf{v}}_{i-1}] \times [\dot{\mathbf{v}}_{i+1} - \dot{\mathbf{v}}_i] \\ &= (x_i - x_{i-1})(y_{i+1} - y_i) - (y_i - y_{i-1})(x_{i+1} - x_i) \\ &= (5 - 0)(5 - 0) - (0 - 0)(-1 - 5) = +25 \Rightarrow +ve. \end{aligned}$$

Different signs → concave polygon



# Convex/Concave Polygons: An Example

## Solution:

- Consider the linear equation given by the line segment  $\langle [-1, -5]^T, [0, 0]^T \rangle$

$$y = \frac{y_{i+1} - y_i}{x_{i+1} - x_i} (x - x_i) + y_i$$

$$y = 5x + 5 - 5$$

$$y - 5x = 0.$$

- Apply the other two points ( $[5, 0]^T$  and  $[-1, 5]^T$ ) to the previous equation

$$y - 5x = 0$$

$$0 - 5(5) = -25 \Rightarrow -ve,$$

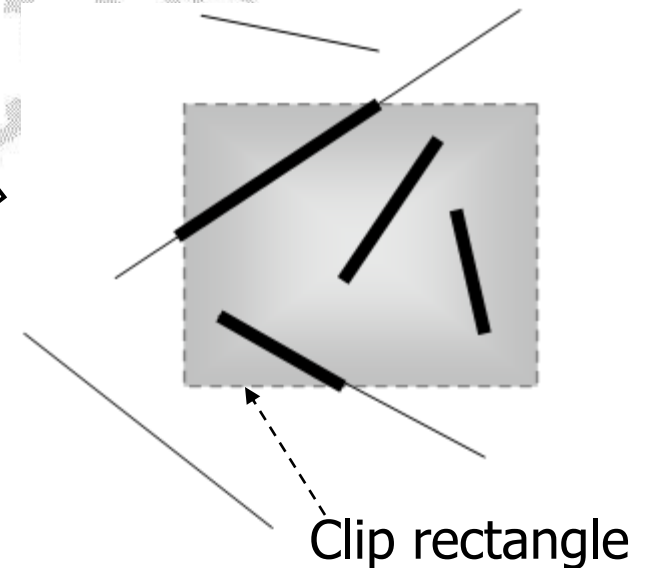
$$5 - 5(-1) = +10 \Rightarrow +ve.$$

Different signs  $\rightarrow$  concave polygon



# 2D Line Clipping

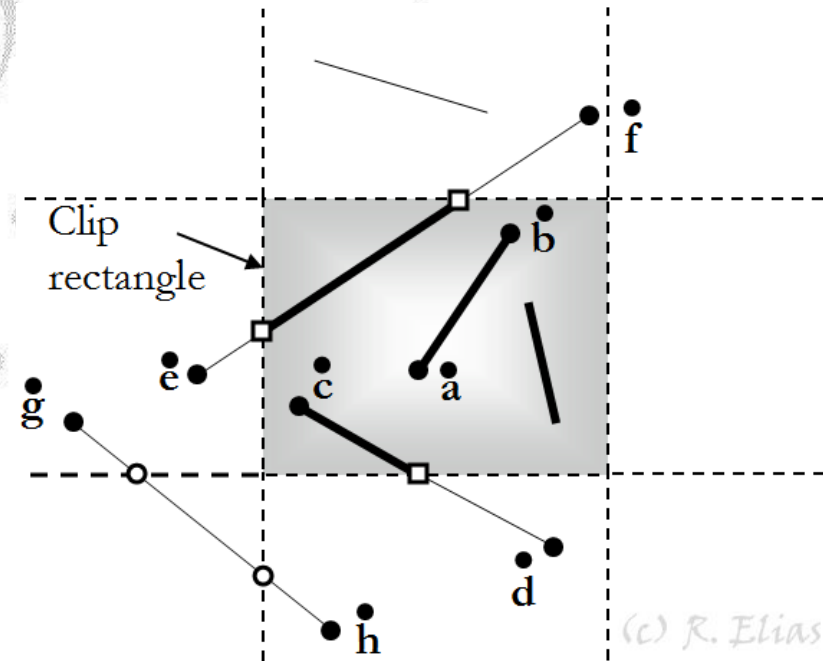
- Given a 2D line or a group of 2D lines, a clip rectangle or window can be used to clip those lines so that only lines or portions of lines inside the clip window are preserved while the other lines or portions of lines are removed.
- In the example shown, lines preserved after clipping appear thicker.
- Such an approach is referred to as a **2D line clipping** algorithm.



# 2D Line Clipping

There are three distinctive cases that may be observed:

- Both endpoints of the line are *inside* the clip rectangle as line  $\overline{ab}$  → **trivially accept the line.**
- One endpoint is *inside* the clip rectangle while the other endpoint is *outside* the rectangle as line  $\overline{cd}$  → **compute intersection point.**
- Both endpoints of the line are *outside* the clip rectangle as lines  $\overline{ef}$  and  $\overline{gh}$  → **perform further calculations.**



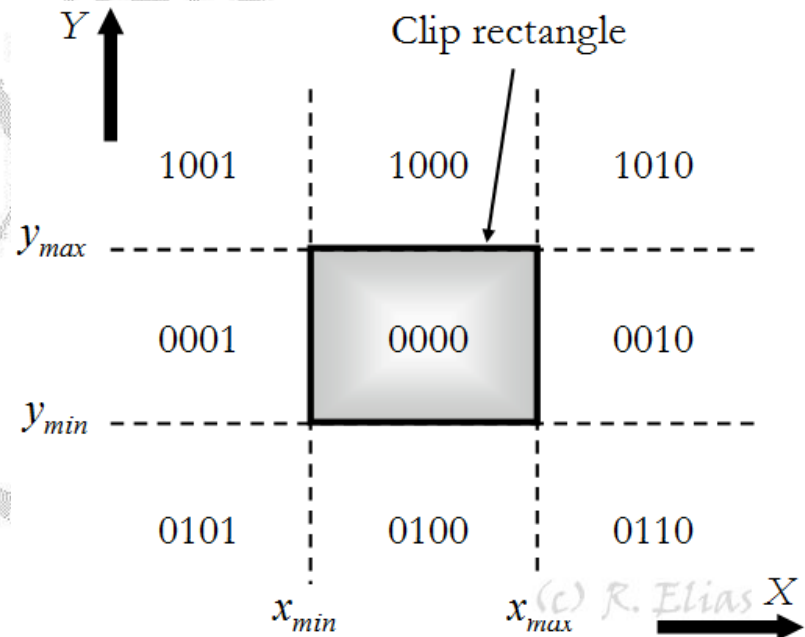
Refer to Example 2.10





# 2D Line Clipping: Cohen-Sutherland Algorithm

- This algorithm divides a 2D space into 9 regions, of which only the middle part (i.e., the clip rectangle) is visible.
- Each region is assigned a 4-bit outcode.
- The bits in the outcode represent: Top, Bottom, Right, Left.
  - For example the outcode 1010 represents a point that is top-right of the viewport.



<b>1</b>	0	<b>1</b>	0
<b>Top</b>	Bottom	<b>Right</b>	Left



# 2D Line Clipping: Cohen-Sutherland Algorithm

Steps:

1. Determine the outcode for each endpoint.
2. Dealing with the two outcodes:
  - a) Bitwise-OR the bits. If this results in 0000, **trivially accept** the line as both endpoints are in the clip rectangle.
  - b) Otherwise, bitwise-AND the bits. If this results in a value other than 0000, **trivially reject** the line as both endpoints are in a region other than the clip rectangle.
  - c) Otherwise, **segment** the line using the edges of the clip rectangle. The portion outside the clip rectangle is rejected. The outpoint is replaced by the intersection point. Go to Step 2.
3. If trivially accepted, draw the line.

How?

Refer to Alg. 2.14



# 2D Line Clipping: Cohen-Sutherland Algorithm

## How to determine the outcode for an endpoint?

- Initialize *outcode* to 0000
- Perform the following operations

$$\text{outcode} = \begin{cases} \text{outcode OR } 1000, & \text{if } y > y_{\max}; \\ \text{outcode OR } 0100, & \text{if } y < y_{\min}, \end{cases}$$

then

$$\text{outcode} = \begin{cases} \text{outcode OR } 0010, & \text{if } x > x_{\max}; \\ \text{outcode OR } 0001, & \text{if } x < x_{\min}. \end{cases}$$

### Algorithm 2.13

Input:  $x, y, x_{\min}, x_{\max}, y_{\min}, y_{\max}$

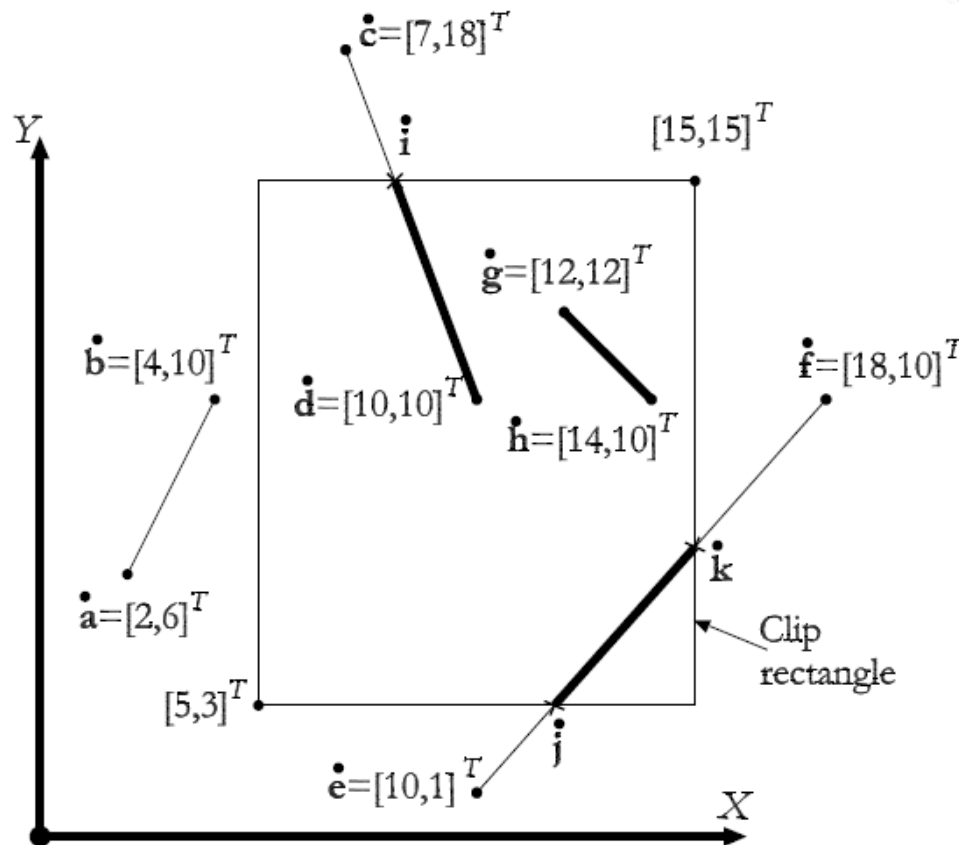
Output: *outcode*

```
1: outcode = 0000
2: if ( $y > y_{\max}$ ) then
3:   outcode = outcode OR 1000
4: else if ( $y < y_{\min}$ ) then
5:   outcode = outcode OR 0100
6: end if
7: if ( $x > x_{\max}$ ) then
8:   outcode = outcode OR 0010
9: else if ( $x < x_{\min}$ ) then
10:  outcode = outcode OR 0001
11: end if
12: return outcode
end
```



# Cohen-Sutherland Algorithm: An Example

- Example:** Get the outcodes for each of the endpoints shown below.



- Solution:**

Point	$[x, y]^T$	Condition	Outcode
$\mathbf{a}$	$[2, 6]^T$	$x < x_{min}$	0001
$\mathbf{b}$	$[4, 10]^T$	$x < x_{min}$	0001
$\mathbf{c}$	$[7, 18]^T$	$y > y_{max}$	1000
$\mathbf{d}$	$[10, 10]^T$	within	0000
$\mathbf{e}$	$[10, 1]^T$	$y < y_{min}$	0100
$\mathbf{f}$	$[18, 10]^T$	$x > x_{max}$	0010
$\mathbf{g}$	$[12, 12]^T$	within	0000
$\mathbf{h}$	$[14, 10]^T$	within	0000



# Cohen-Sutherland Algorithm: An Example

- **Example:** Given the outcodes obtained previously, determine which lines are trivially accepted/rejected and which lines need intersection determination.

Point	Outcode
$\dot{a}$	0001
$\dot{b}$	0001
$\dot{c}$	1000
$\dot{d}$	0000
$\dot{e}$	0100
$\dot{f}$	0010
$\dot{g}$	0000
$\dot{h}$	0000

- **Solution:**

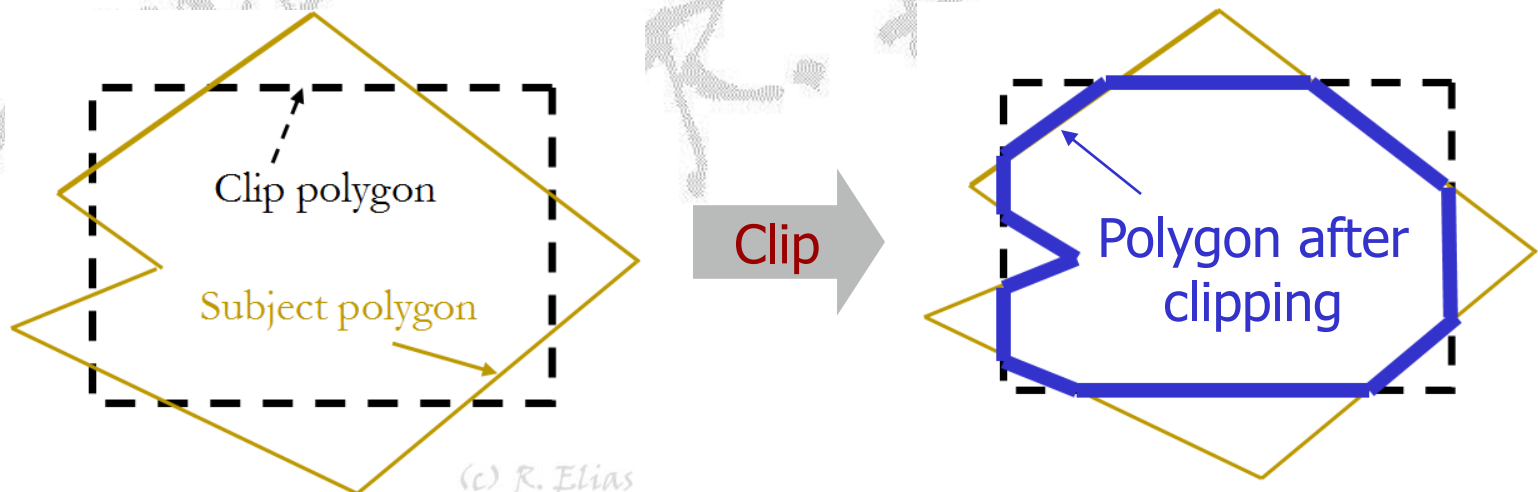
Line	ORing	ANDing	Decision
$\overline{\dot{a}}\dot{b}$	0001	0001	Trivially reject
$\overline{\dot{c}}\dot{d}$	1000	0000	intersections
$\overline{\dot{e}}\dot{f}$	0110	0000	intersections
$\overline{\dot{g}}\dot{h}$	0000		Trivially accept

Refer to Examples 2.13 and 2.14 to get the intersections and portions of lines kept.



# 2D Polygon Clipping

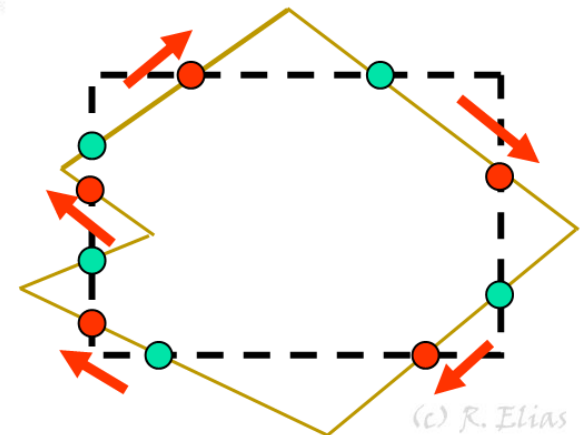
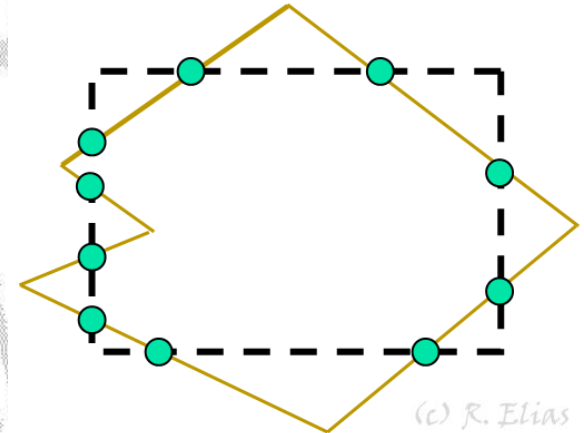
- A 2D polygon can be clipped by another polygon.
- The output of this clipping process is one or more polygons.
- The polygon after clipping may include vertices that are not part of the original vertices (i.e., new vertices may be created).



# 2D Polygon Clipping: Weiler-Atherton Algorithm

- It allows clipping of a *subject polygon* by a *clip polygon*.
- Polygons are clockwise-oriented.
- A polygon is represented as a circular linked list.

1. Compute intersection points.
2. Walking along the boundaries of the subject polygon in a clockwise direction, mark points where the subject polygon enters/leaves the clip polygon.



- Entering points
- Leaving points

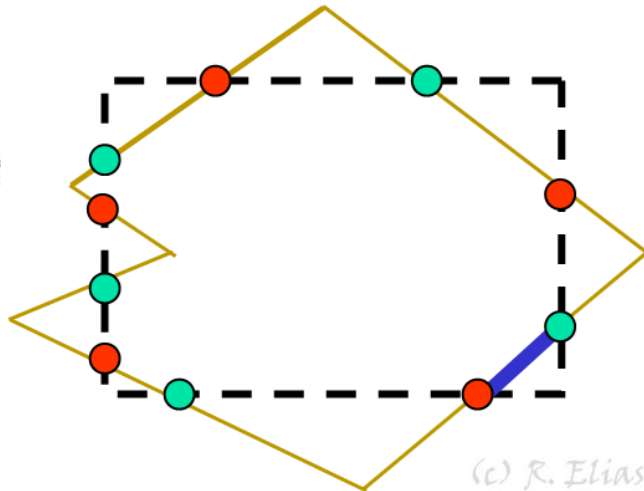




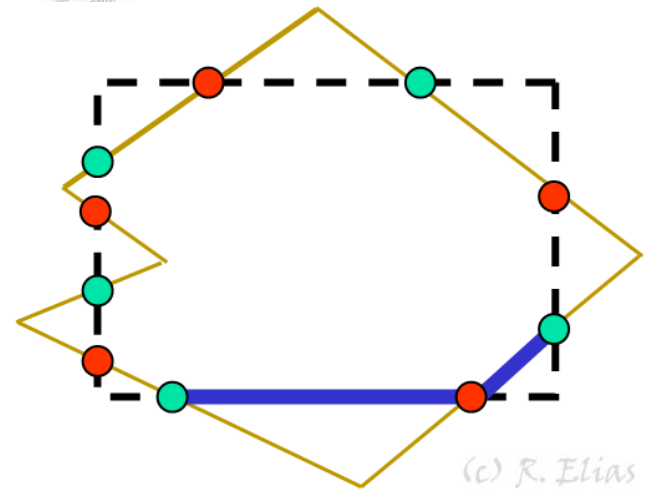
# 2D Polygon Clipping: Weiler-Atherton Algorithm

Steps (cont.)

3. There are two types of point pairs:
  - a) **Out-to-in pair:** At an entering point, follow the subject polygon vertices in its circular list until the next leaving intersection.
  - b) **In-to-out pair:** At a leaving point, follow the clip polygon vertices in its circular list until the next entering intersection.
4. Repeat Step 3 until there are no more pairs to process.



(c) 2022, Dr. R. Elias

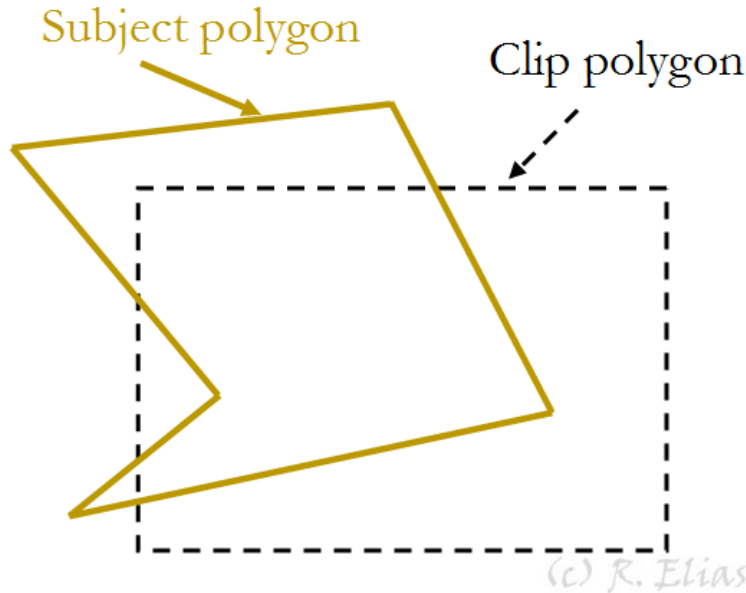


2D Graphics



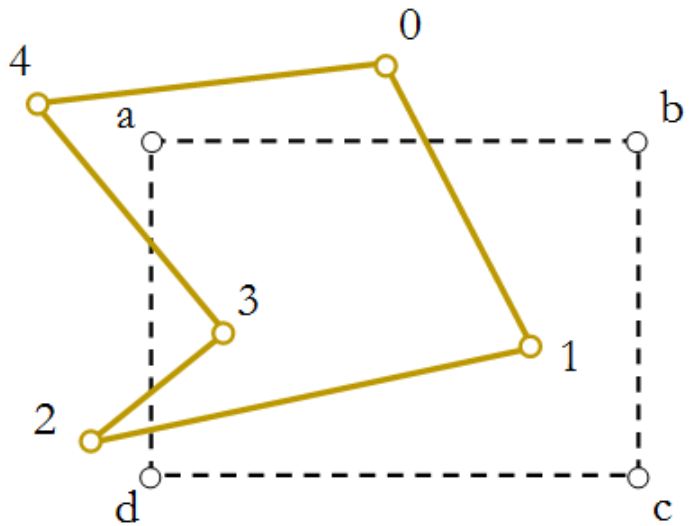
# Weiler-Atherton Algorithm: An Example

- **Example:** What are the steps that should be followed to populate circular lists for both the subject and clip polygons shown below?



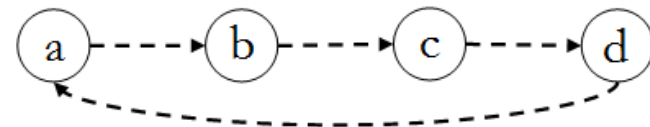
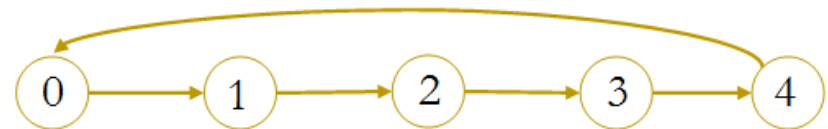
# Weiler-Atherton Algorithm: An Example

- Solution:** The sequence of vertices of the subject polygon in a clockwise order is "0," "1," "2," "3" and "4." The sequence of vertices of the clip polygon in a clockwise order is "a," "b," "c" and "d."



(c) R. Elias

Subject polygon vertices



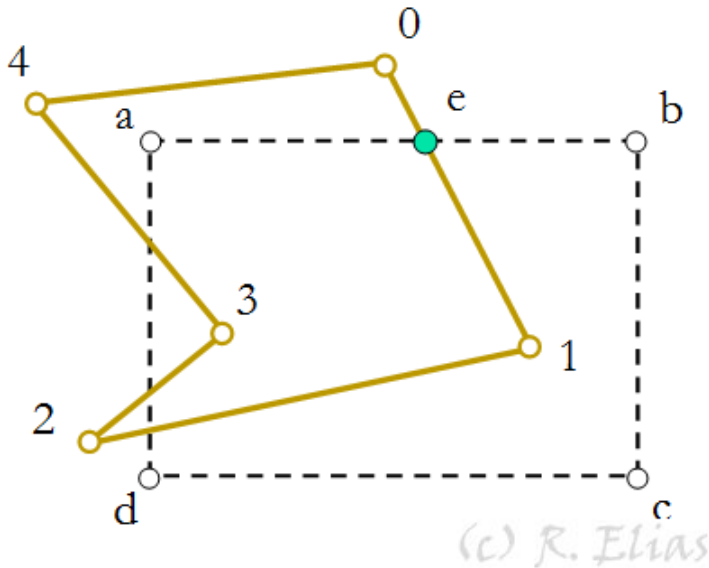
(c) R. Elias

Clip polygon vertices

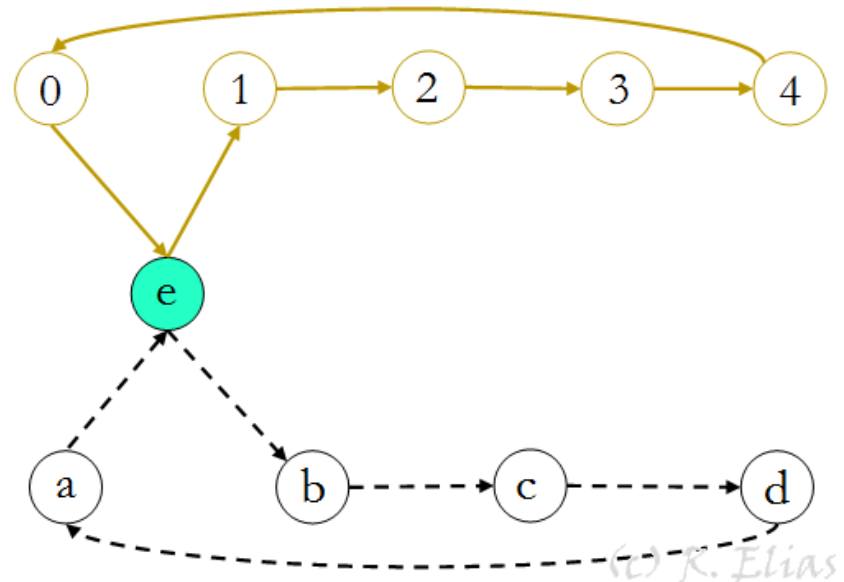


# Weiler-Atherton Algorithm: An Example

- The first intersection marked "e" is detected and a new false vertex is linked to both lists



(c) 2022, Dr. R. Elias

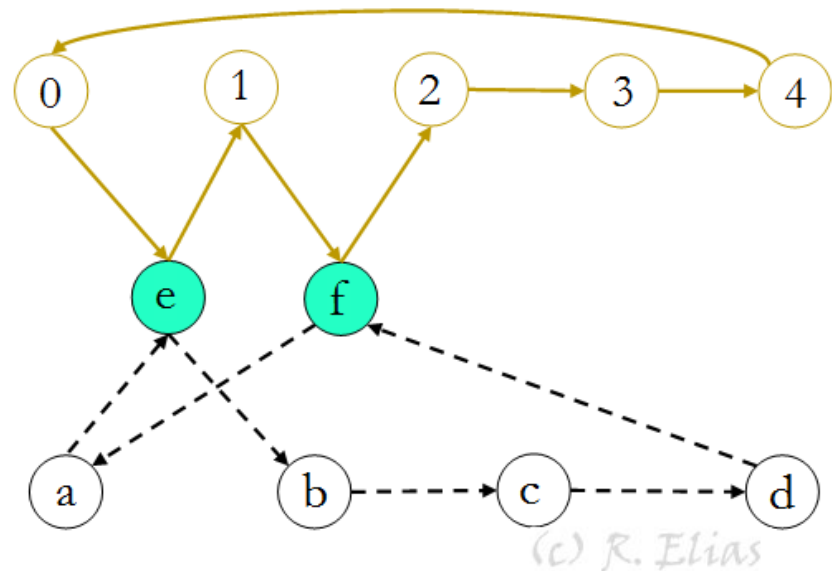
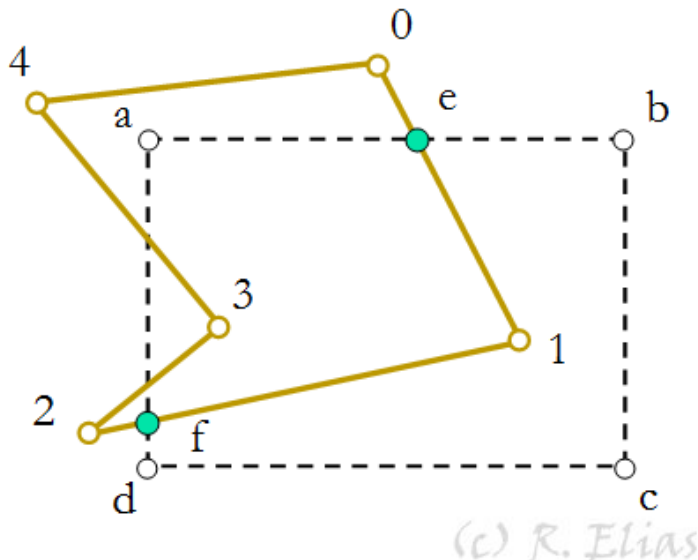


2D Graphics



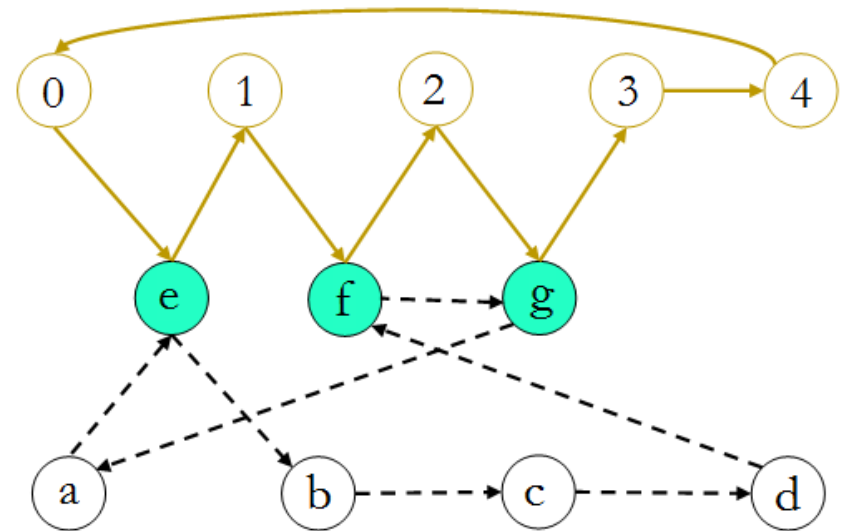
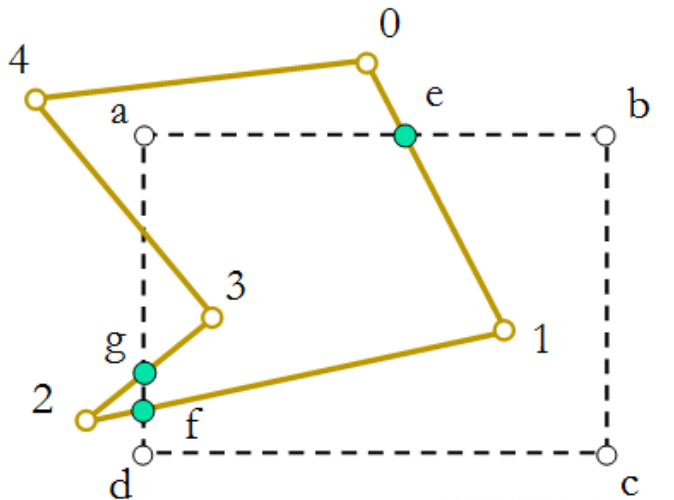
# Weiler-Atherton Algorithm: An Example

- Along the clockwise direction, the next intersection marked "f" is detected and a new false vertex is linked to both lists.
- Notice that "f" is between "1" and "2" in the subject polygon and between "d" and "a" in the clip polygon.



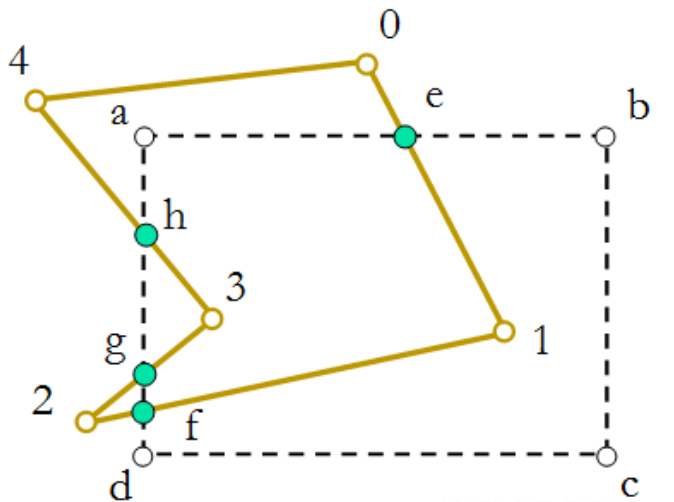
# Weiler-Atherton Algorithm: An Example

- The following intersection marked "g" is detected and a new false vertex is linked to both lists.

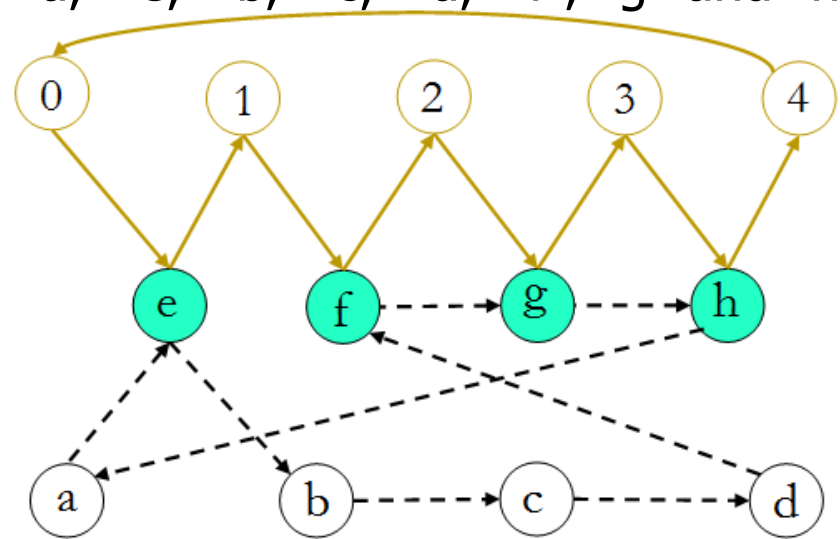


# Weiler-Atherton Algorithm: An Example

- The next intersection marked "h" is detected and a new false vertex is linked to both lists.
- The subject sequence now becomes "0," "e," "1," "f," "2," "g," "3," "h" and "4."
- The clip sequence becomes "a," "e," "b," "c," "d," "f," "g" and "h."



(c) R. Elias

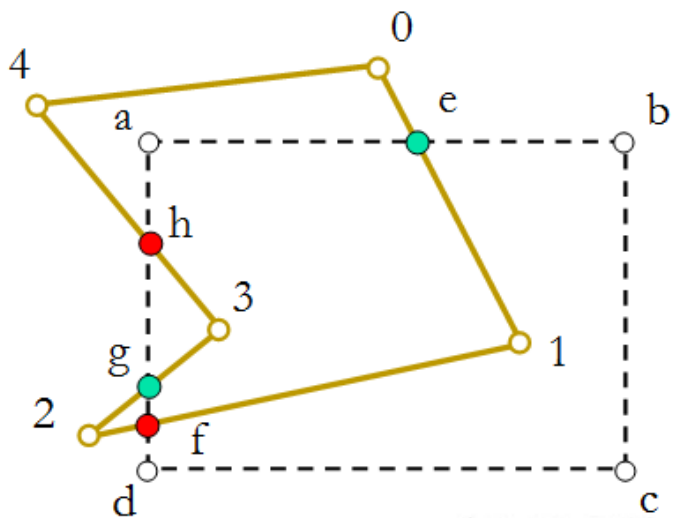


(c) R. Elias



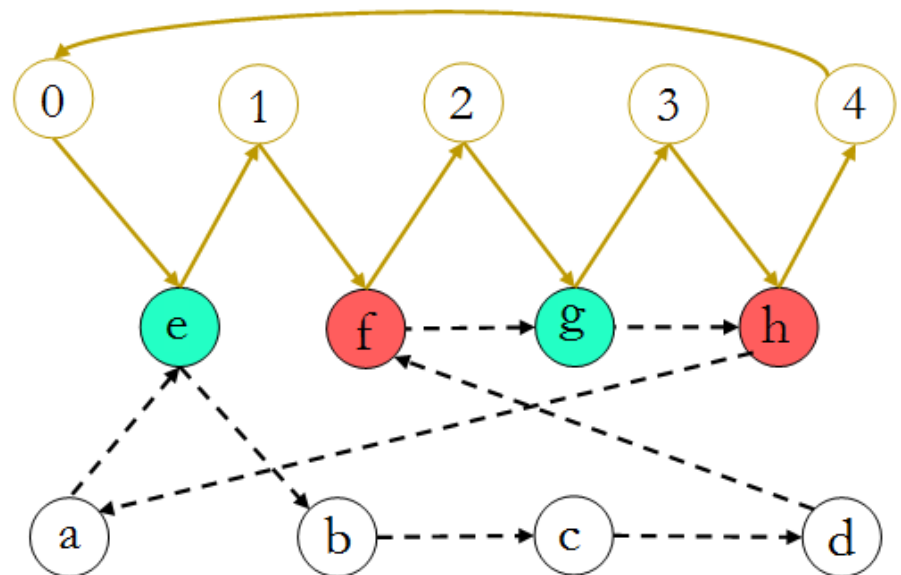
# Weiler-Atherton Algorithm: An Example

- **Example:** Building on the previous example determine the clipped polygon parts (i.e., loops of vertices).
- **Solution:** The entering and leaving vertices are determined.



(c) R. Elias

(c) 2022, Dr. R. Elias



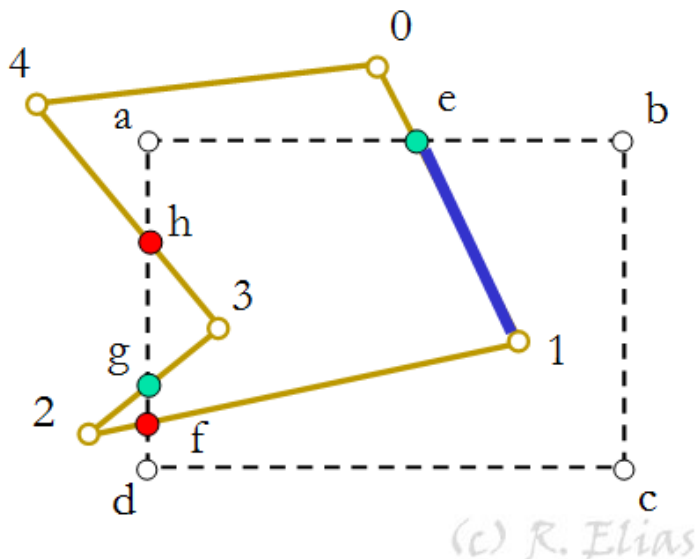
(c) R. Elias

2D Graphics

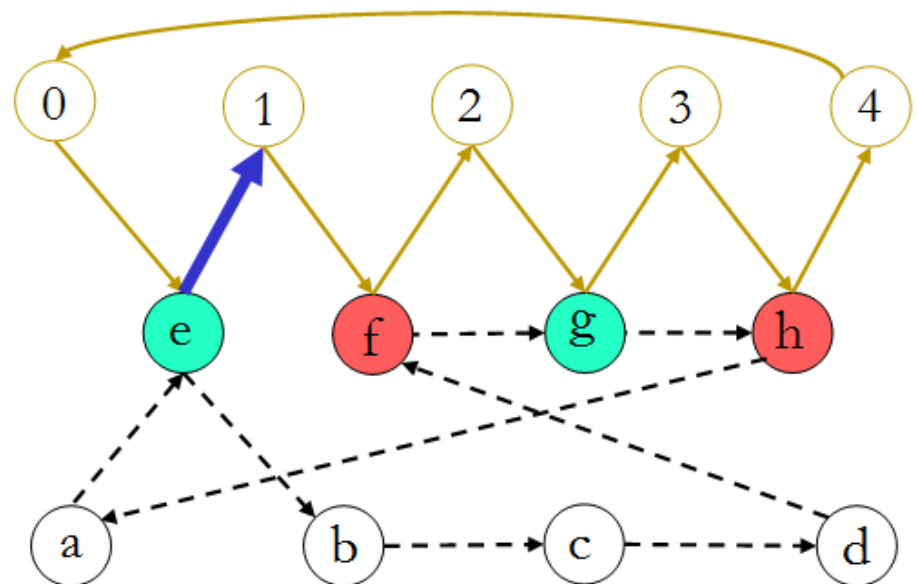


# Weiler-Atherton Algorithm: An Example

- Starting at the *entering* vertex "e," the *subject* polygon border is followed.
- This process is performed on the *subject* polygon circular list by following the link out of that vertex.



(c) 2022, Dr. R. Elias

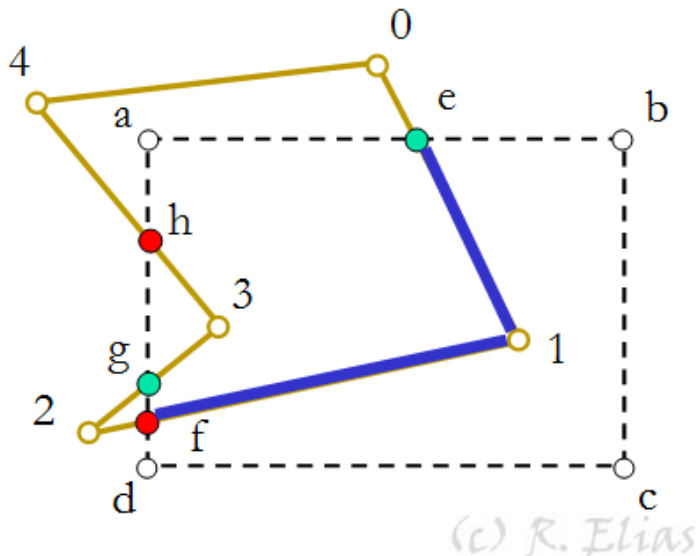


2D Graphics

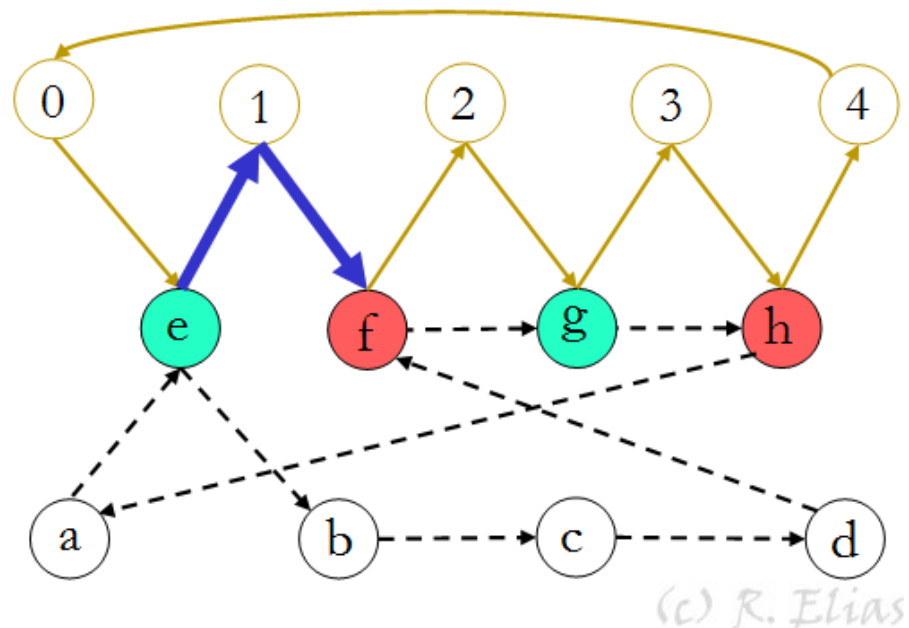


# Weiler-Atherton Algorithm: An Example

- Continuing from the subject vertex "1," the *subject* polygon is followed as in the previous step.



(c) 2022, Dr. R. Elias



2D Graphics



- link out of that vertex.



(c) 2022, Dr. R. Elias

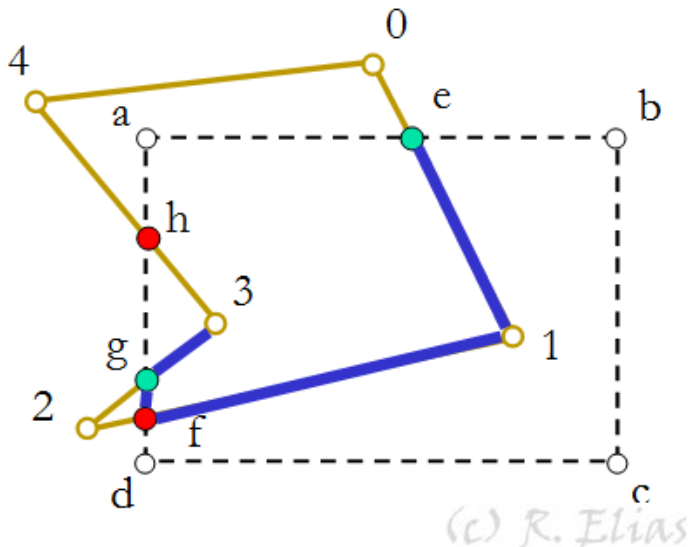


## 2D Graphics

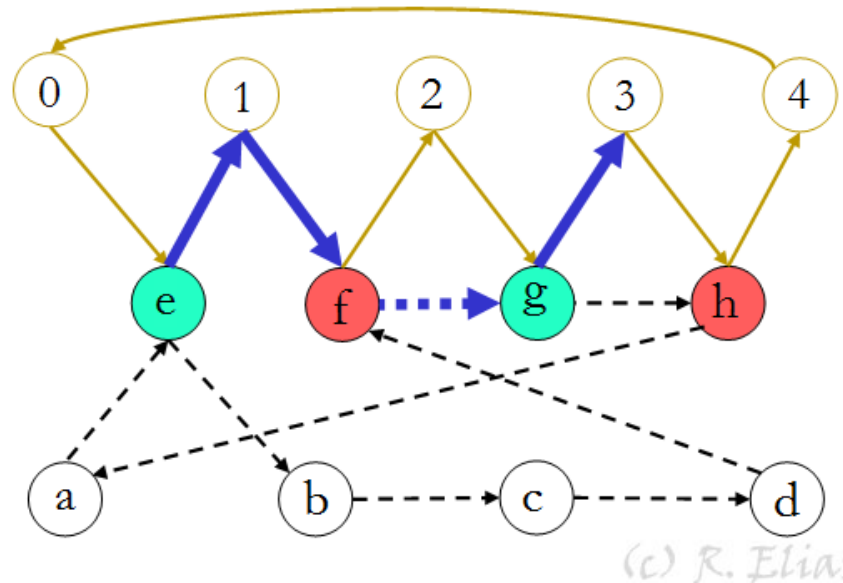


# Weiler-Atherton Algorithm: An Example

- From the *entering* vertex "g," the *subject* polygon border is followed.
- This process is performed on the *subject* polygon circular list by following the link out of that vertex.



(c) 2022, Dr. R. Elias

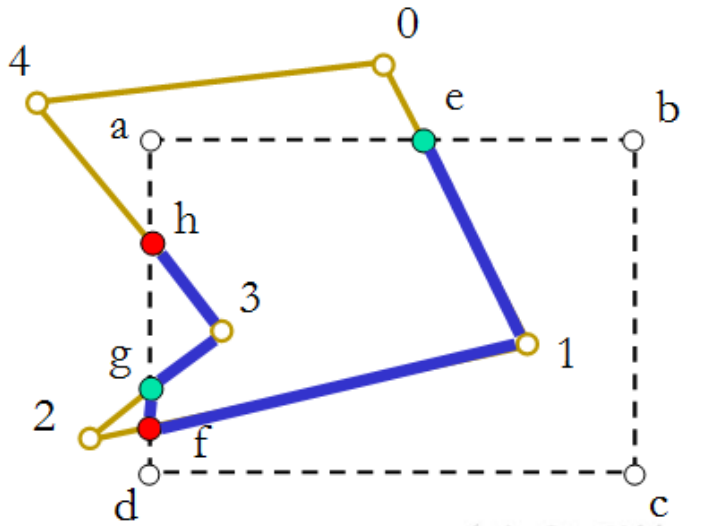


2D Graphics



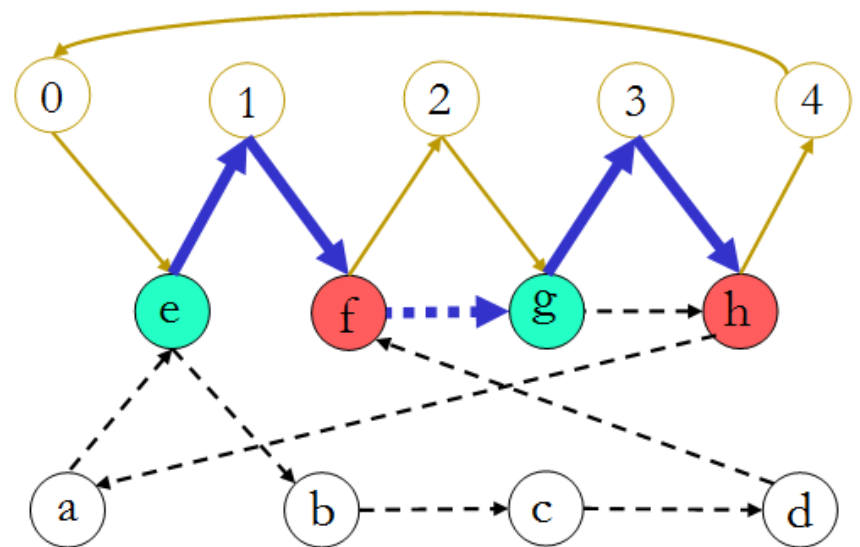
# Weiler-Atherton Algorithm: An Example

- Continuing from the subject vertex "3," the *subject* polygon is followed.



(c) R. Elias

(c) 2022, Dr. R. Elias



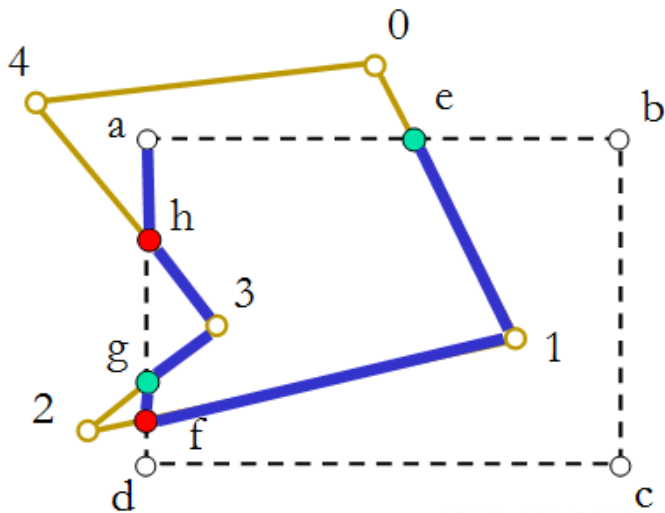
(c) R. Elias

2D Graphics



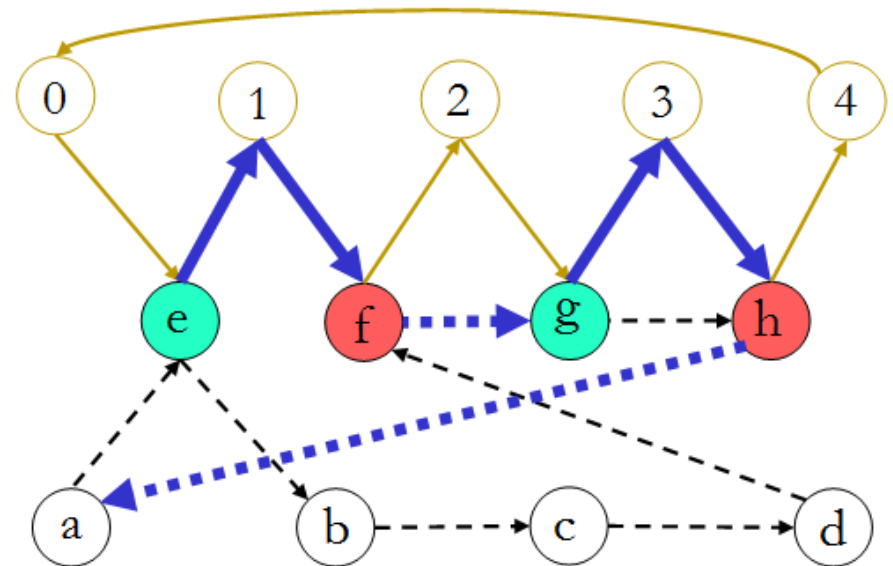
# Weiler-Atherton Algorithm: An Example

- At the *leaving* vertex "h," the *clip* polygon is followed.



(c) R. Elias

(c) 2022, Dr. R. Elias



(c) R. Elias

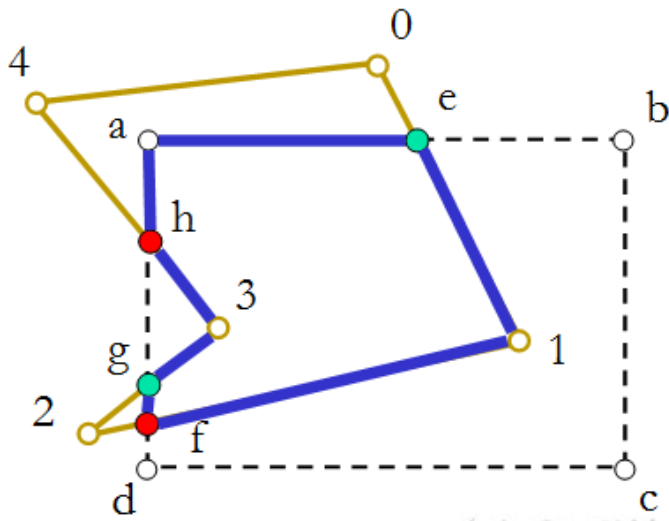
2D Graphics





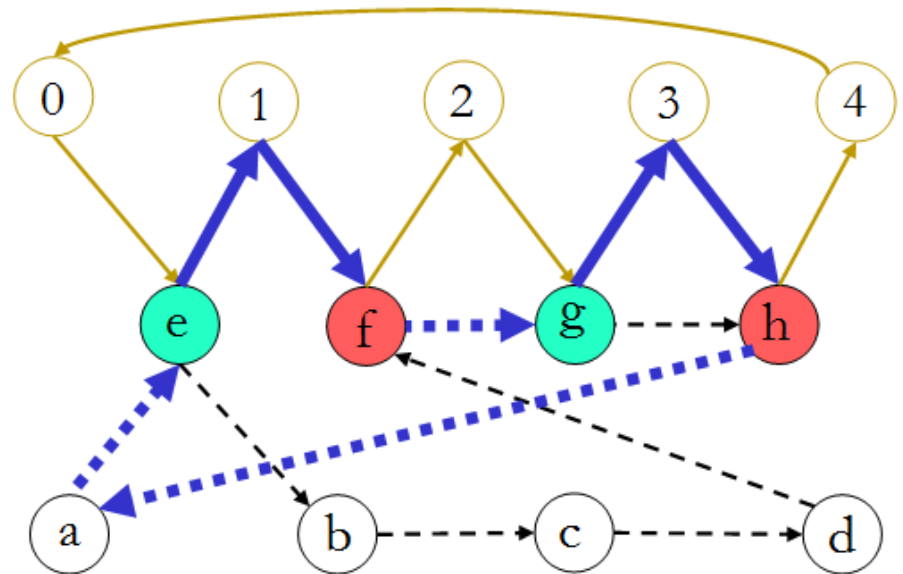
# Weiler-Atherton Algorithm: An Example

- The loop is complete by following the *clip* path from the *clip* vertex "a" to the *entering* vertex "e," which is the start vertex.
- The final loop (i.e., the clipped part of the polygon) now is "e," "1," "f," "g," "3," "h," "a" and "e."



(c) R. Elias

(c) 2022, Dr. R. Elias



(c) R. Elias

2D Graphics





# Clipping: Other Algorithms

---

- Other 2D line clipping algorithms:
  - Liang-Barsky
  - Nicholl-Lee-Nicholl
- Other 2D polygon clipping algorithms:
  - Sutherland-Hodgman
  - Patrick-Gilles Maillot





# Summary

---

- We have covered some basic algorithms to handle graphics 2D graphics
  - Drawing lines
    - Bresenham's algorithm
    - Midpoint algorithm
  - Drawing circles
    - 2-way symmetry
    - 4-way symmetry
    - 8-way symmetry
    - Midpoint algorithm
  - Polygons
  - Line clipping
  - Polygon clipping

