RV College of Engineering®

Go, change the world

COMPUTER GRAPHICS (16CS73)

UNIT-III

Raster Graphics Algorithms

Raster graphics algorithms and Geometric Transformations

- ✓ ***Raster graphics algorithms and Geometric Transformations:***

  Points and lines, line drawing algorithms, mid-point circle algorithm;

- ✓ ***Filled area primitives****: Scan line polygon fill algorithm, boundary-fill and flood-fill algorithms.*

- ✓ ***2-D viewing:*** *The viewing pipeline, viewing coordinate reference frame, window to view-port coordinate transformation, viewing functions, Cohen-Sutherland and Liang Barsky line clipping algorithms, Sutherland –Hodgeman polygon clipping algorithm.*

- ✓ ***Geometric Transformations:*** *2-D geometrical transformations: Translation, Scaling, Rotation, reflection and shear transformations, matrix representations and homogeneous coordinates, composite transforms, transformations between coordinate systems.*

Raster graphics algorithms and Geometric Transformations

- ✓ ***Raster graphics algorithms and Geometric Transformations:***

  Points and lines, line drawing algorithms, mid-point circle algorithm;

- ✓ ***Filled area primitives****: Scan line polygon fill algorithm, boundary-fill and flood-fill algorithms.*

- ✓ ***2-D viewing:*** *The viewing pipeline, viewing coordinate reference frame, window to view-port coordinate transformation, viewing functions, Cohen-Sutherland and Liang Barsky line clipping algorithms, Sutherland –Hodgeman polygon clipping algorithm.*

- ✓ ***Geometric Transformations:*** *2-D geometrical transformations: Translation, Scaling, Rotation, reflection and shear transformations, matrix representations and homogeneous coordinates, composite transforms, transformations between coordinate systems.*

Filled- Area Primitives

- A standard output primitive in general graphics packages is a solid-color or patterned polygon area.

- There are two basic approaches to area filling on raster systems:

1. **The scan-line approach**
   - Determine the overlap intervals for scan lines that cross the area.
   - is typically used in general graphics packages to fill polygons, circles, ellipses

     works at the polygon level

     better performance
2. **Filling approaches** –(Seed fill approach)
   - start from a given interior position and paint outward from this point until we encounter the specified boundary conditions.
   - useful with more complex boundaries and in interactive painting systems.

     2 algorithms: Boundary Fill and Flood Fill

     works at the pixel level

Fill area : an area that is filled with solid colour or pattern

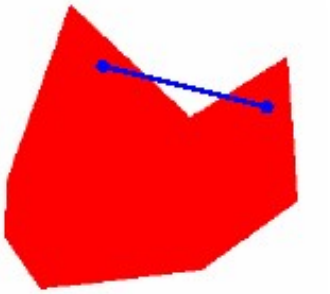
Polygon Fill Areas

- Most library routines require that a fill area be specified as a polygon
  - OpenGL only allows ***convex*** polygons
- Non-polygon (curved) objects can be approximated by polygons
  - Surface tessellation, polygon mesh, triangular mesh

Polygon types

- Simple polygons are either ***convex*** or ***concave***:
  - Convex polygon: All interior angles < 180°, or any line segment combining two points in the interior is also in the interior

convex polygon



concave polygon

can be split into a number of convex polygons

Identifying a concave polygon

- has at least one interior angle >180 degrees
- extensions of some edges will intersect other edges

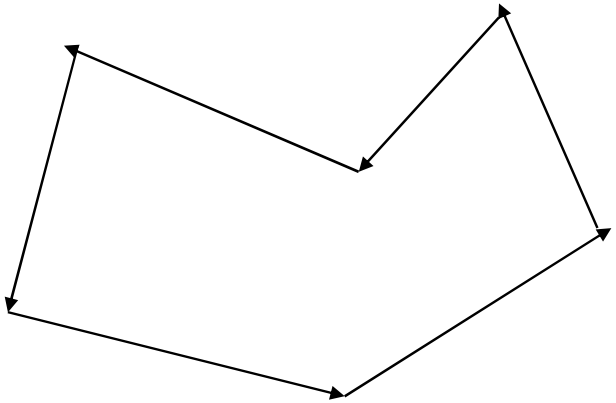
One test:

Express each polygon edge as a vector, with a consistent orientation.

Can then calculate cross-products of adjacent edges

Identifying a concave polygon

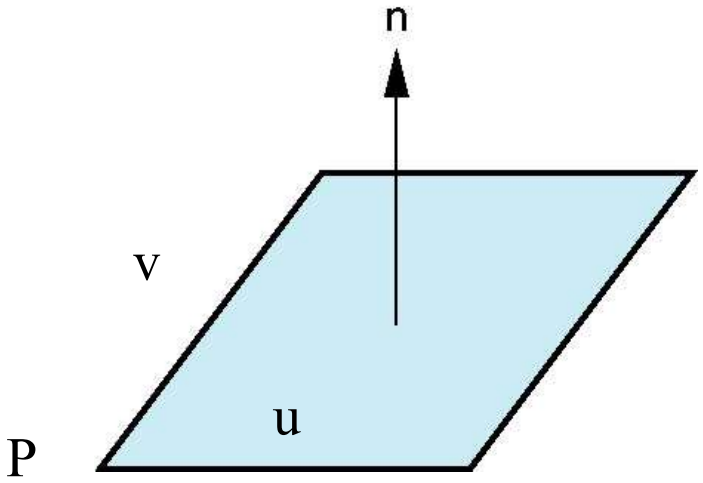
- When polygon edges are oriented with an anti-clockwise sense
  - cross product at convex vertex has positive sign
  - concave vertex gives negative sign



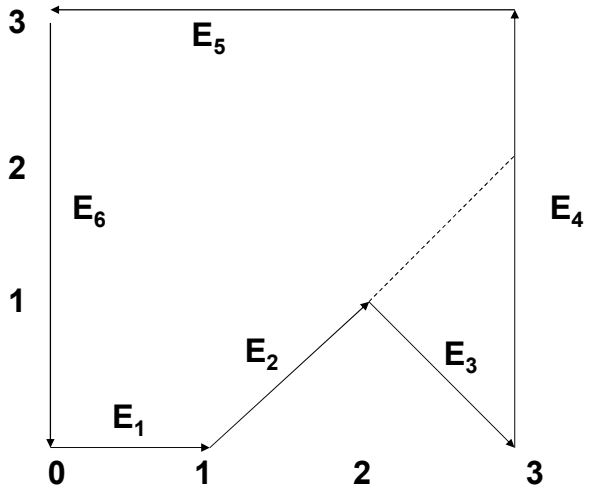
Normals

- Every plane has a vector **n** normal (perpendicular, orthogonal) to it
- **n** = **u** x **v** (vector cross product)

$$\bar{u} \times \bar{v} = \begin{pmatrix} u_y v_z - u_z v_y \\ u_z v_x - u_x v_z \\ u_x v_y - u_y v_x \end{pmatrix}$$



Vector method for splitting concave polygons

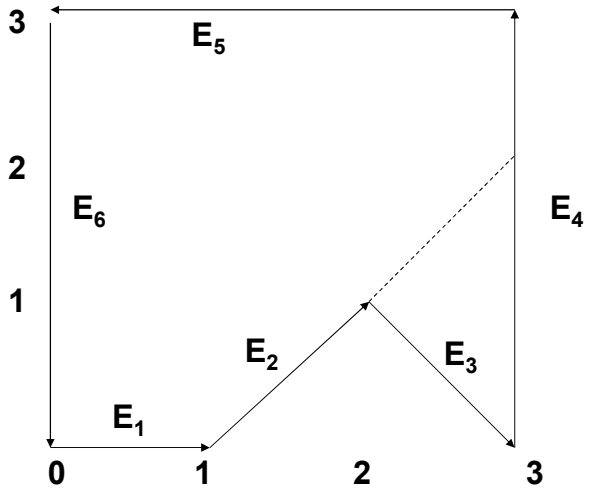


- $E_1=(1,0,0)$ $E_2=(1,1,0)$
- $E_3=(1,-1,0)$ $E_4=(0,3,0)$
- $E_5=(-3,0,0)$ $E_6=(0,-3,0)$

- All z components have 0 value.
- Cross product of two vectors $E_j x E_k$ is perpendicular to them with z component

$$E_{jx}E_{ky}-E_{kx}E_{jy}$$

Example continued

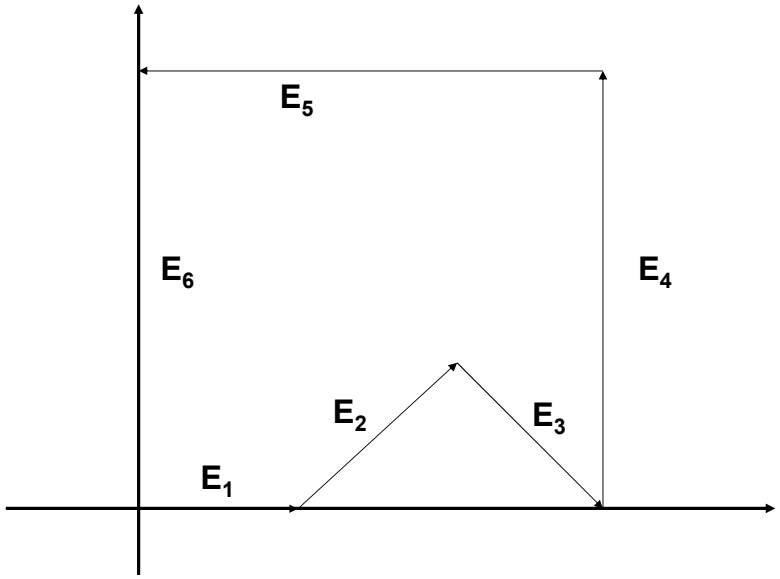


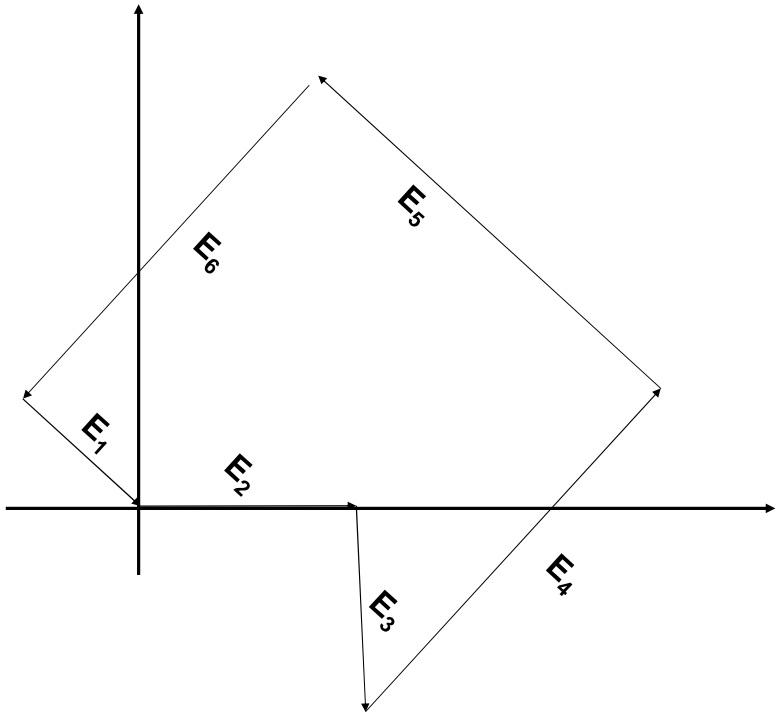
- $E_1 x E_2 = (0,0,1)$
  $E_2 x E_3 = (0,0,-2)$
  $E_3 x E_4 = \ldots$
  $E_4 x E_5 = \ldots$
- $E_5 x E_6 = \ldots$
  $E_6 x E_1 = \ldots$

- Since $E_2 x E_3$ has negative sign, split the polygon along the line of vector E_2

Rotational method

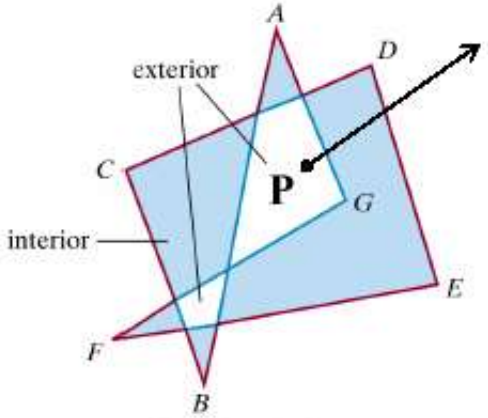
- Rotate the polygon so that each vertex in turn is at coordinate origin.
- If following vertex is below the x axis, polygon is concave.
- Split the polygon by x axis.


E5
E6
E4
E2
E3
E1


E5
E6
E1
E2
E3
E4

Inside-Outside Tests

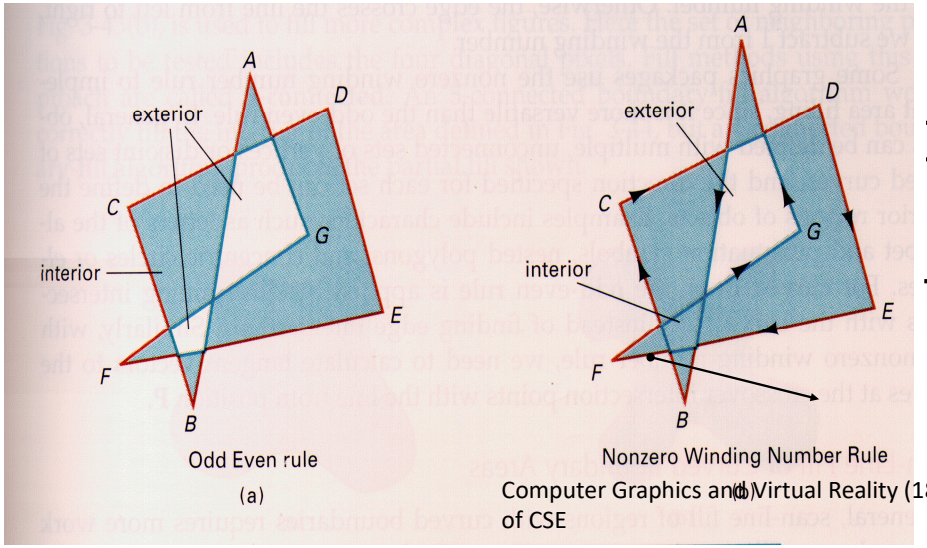
- Identifying the interior of a polygon (simple or complex) is important to identify the region to be filled
- Odd-even rule: To determine whether point **P** is inside or not. Draw a line starting from P to a distant position. Count the number of edges that cross this line. If the count is **odd** then the point is **inside**, otherwise it is outside.



Odd-Even Rule

Inside-Outside? nonzero winding-number rule

A winding number is an attribute of a point with respect to a polygon that tells us how many times the polygon encloses (or wraps around) the point. It is an integer, greater than or equal to 0. Regions of winding number 0 (unenclosed) are obviously outside the polygon, and regions of winding number 1 (simply enclosed) are obviously inside the polygon.



Initially 0

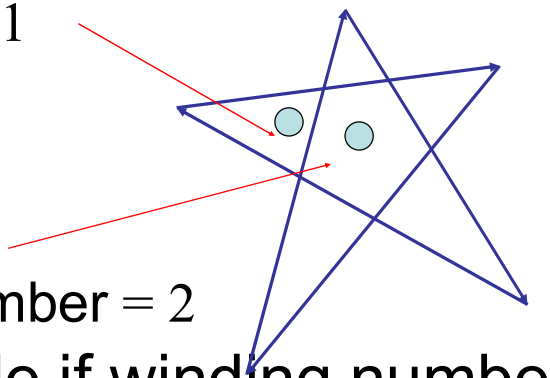
+1: edge crossing the line from right to left

-1: left to right

Winding Number

- Count clockwise encirclements of point

winding number = 1



winding number = 2

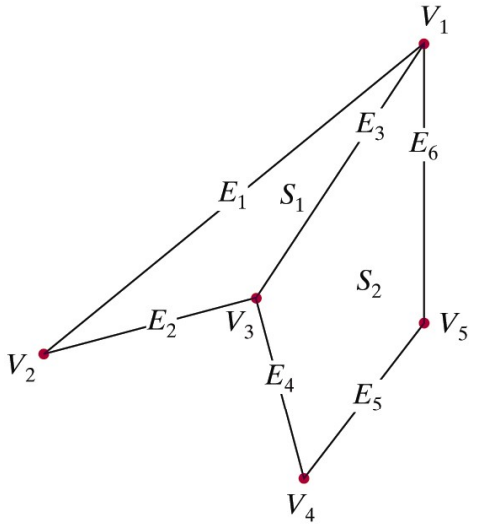
- Alternate definition of inside: inside if winding number ≠ 0

Polygon tables

- store descriptions of polygon geometry and topology, and surface parameters: colour, transparency, light-reflection
- organise in 2 groups
  - geometric data
  - attribute data

Polygon Tables:

Geometric data



VERTEX TABLE	
V_1:	x_1, y_1, z_1
V_2:	x_2, y_2, z_2
V_3:	x_3, y_3, z_3
V_4:	x_4, y_4, z_4
V_5:	x_5, y_5, z_5

EDGE TABLE	
E_1:	V_1, V_2
E_2:	V_2, V_3
E_3:	V_3, V_1
E_4:	V_3, V_4
E_5:	V_4, V_5
E_6:	V_5, V_1

SURFACE-FACET TABLE	
S_1:	E_1, E_2, E_3
S_2:	E_3, E_4, E_5, E_6

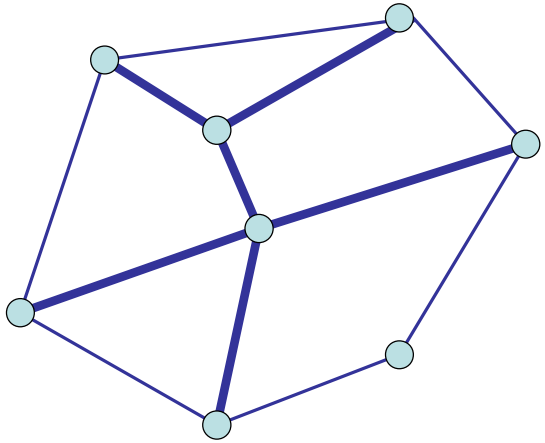
Figure 3-50

Geometric data-table representation for two adjacent polygon surface facets, formed with six edges and five vertices.

- Data can be used for consistency checking
- Additional geometric data stored: slopes, bounding boxes

Shared Edges

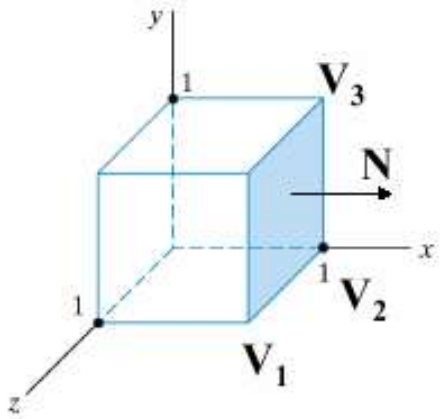
- Vertex lists will draw filled polygons correctly but if we draw the polygon by its edges, shared edges are drawn twice



- Can store mesh by *edge list*

Front and Back Face of a Polygon

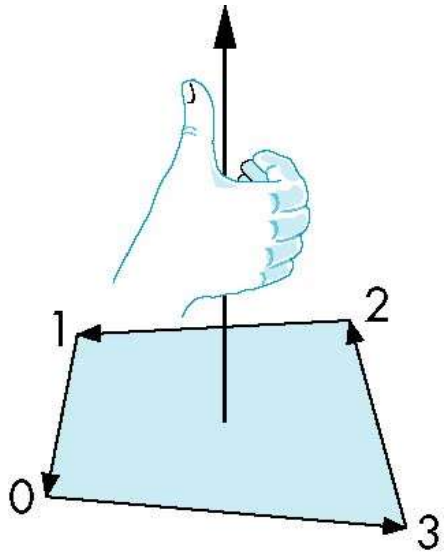
- The normal vector points in a direction from the back face of the polygon to the front face
- Normal vector is the cross product of the two edges of the polygon in counter-clockwise direction



$$\mathbf{N} = (\mathbf{V}_2 - \mathbf{V}_1) \times (\mathbf{V}_3 - \mathbf{V}_2)$$

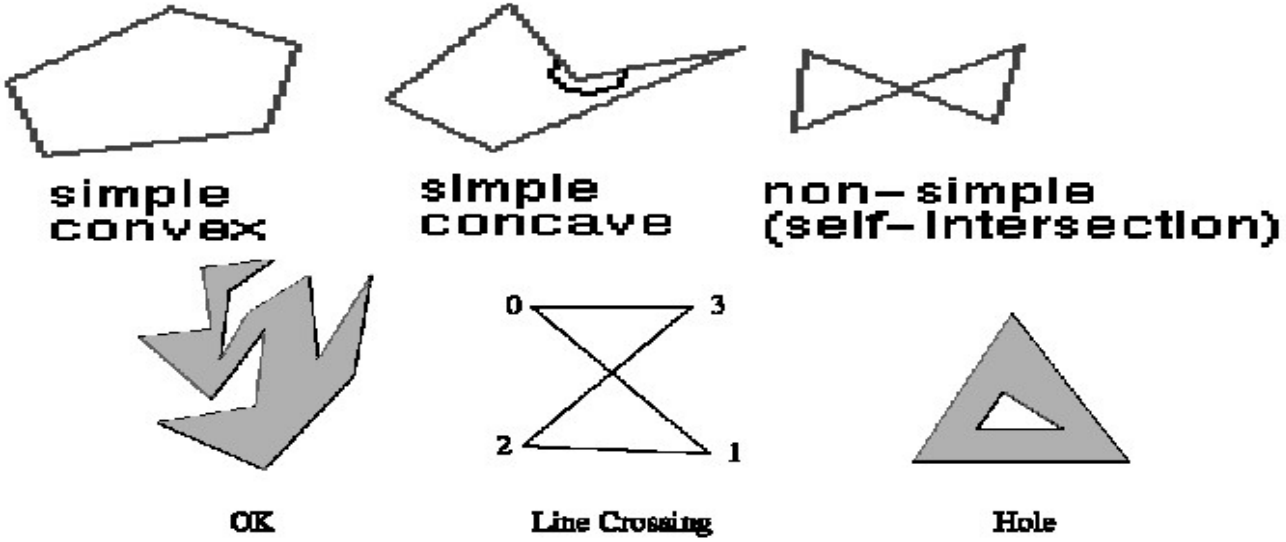
Inward and Outward Facing Polygons

- The order $\{v_1, v_6, v_7\}$ and $\{v_6, v_7, v_1\}$ are equivalent in that the same polygon will be rendered by OpenGL but the order $\{v_1, v_7, v_6\}$ is different
- The first two describe *outwardly facing* polygons
- Use the *right-hand rule* = counter-clockwise encirclement of outward-pointing normal
- OpenGL can treat inward and outward facing polygons differently



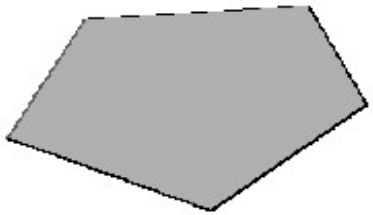
Polygon Types

- simple convex, simple concave, non-simple (self-intersecting)
- want no holes, no intersections (line crossings)
- rectangles and triangles always simple convex

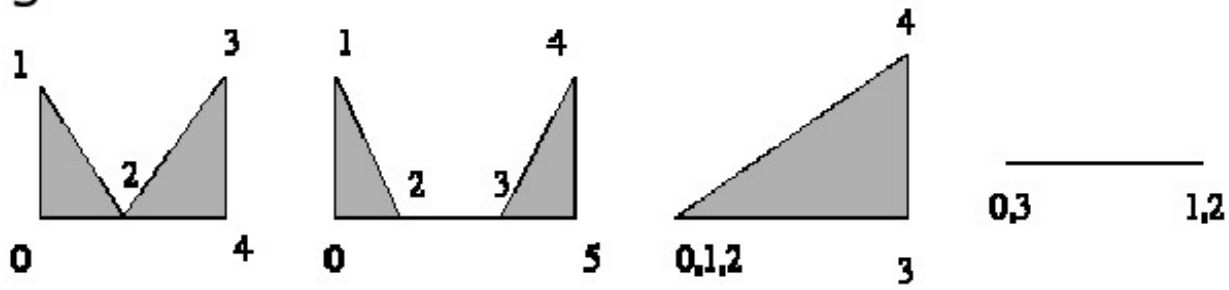


Convex, Concave, Degenerate

- Convex polygons are preferable to concave
  - Polygon is convex if for any two points inside polygon, the line segment joining these two points is also inside.

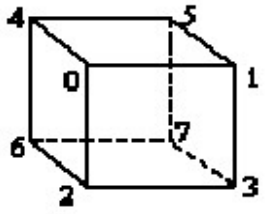


  - Degeneracies



Polygon Representation

- Polygon Representation
  - ordered list of vertices
  - avoids redundant storage and computations
  - associate other information with vertices
    - colors, normals, textures



faces		vertex list	
#	vertex list	#	x,y,z
0	0,2,3,1	0	0,1,1
1	1,3,7,5	1	1,1,1
2	5,7,6,4	2	0,0,1
3	4,6,2,0	3	1,0,1
4	4,0,1,5	4	0,1,0
5	2,6,7,3	5	1,1,0
		6	0,0,0
		7	1,0,0

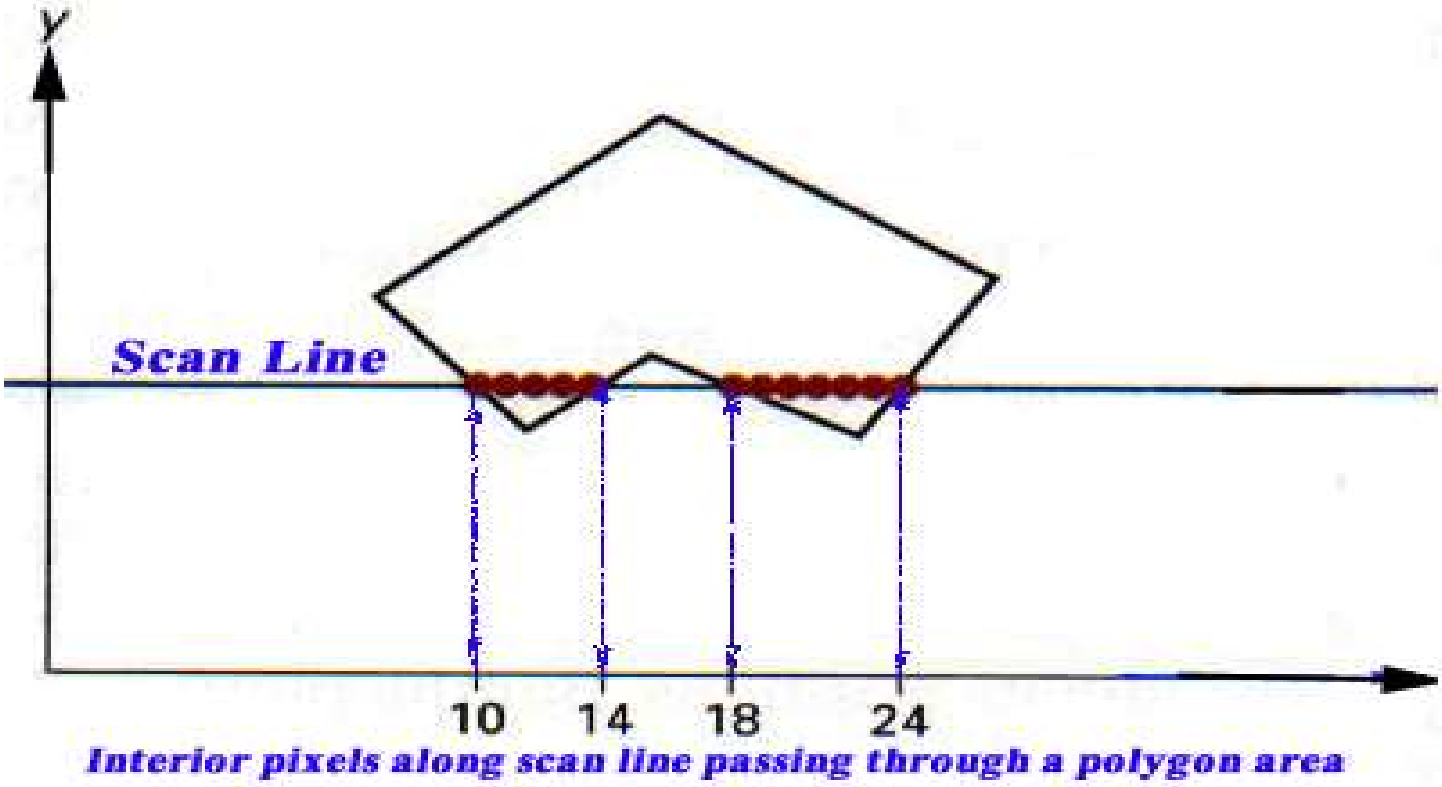
Filled- Area Primitives (cont.)

Scan-Line Fill Algorithm:

- For each scan line crossing a polygon, the area-fill algorithm locates the intersection points of the scan line with the polygon edges.

- These intersection points are then sorted from left to right, and the corresponding frame-buffer positions between each intersection pair are set to the specified fill color.

Filled- Area Primitives (cont.)



Filled- Area Primitives (cont.)

- Calculations performed in scan-conversion and other graphics algorithms typically take advantage of various coherence properties of a scene that is to be displayed.

- Coherence is simply that the properties of one part of a scene are related in some way to other parts of the scene so that the relationship can be used to reduce processing.

- Coherence methods often involve incremental calculations applied along a single scan line or between successive scan lines.

Inside-Outside Tests

- Area-filling algorithms and other graphics processes often need to identify interior regions of objects.

- To identify interior regions of an object graphics packages normally use either:

1. Odd-Even rule
2. Nonzero winding number rule

Inside-Outside Tests

Odd-Even rule (Odd Parity Rule, Even-Odd Rule):

1. draw a line from any position P to a distant point outside the coordinate extents of the object and counting the number of edge crossings along the line.

2. If the number of polygon edges crossed by this line is **odd** then

   P is an **interior** point.

   Else

P is an **exterior** point

Inside-Outside Tests

Nonzero Winding Number Rule :

- Counts the number of times the polygon edges wind around a particular point in the counterclockwise direction. This count is called the winding number, and the interior points of a two-dimensional object are defined to be those that have a nonzero value for the winding number.

1. Initializing the winding number to 0.
2. Imagine a line drawn from any position P to a distant point beyond the coordinate extents of the object.

Inside-Outside Tests

Nonzero Winding Number Rule :

3. Count the number of edges that cross the line in each direction. We add 1 to the winding number every time we intersect a polygon edge that crosses the line from right to left, and we subtract 1 every time we intersect an edge that crosses from left to right.

4. If the winding number is **nonzero**, then

   P is defined to be an **interior** point

   Else

   P is taken to be an **exterior** point**.**

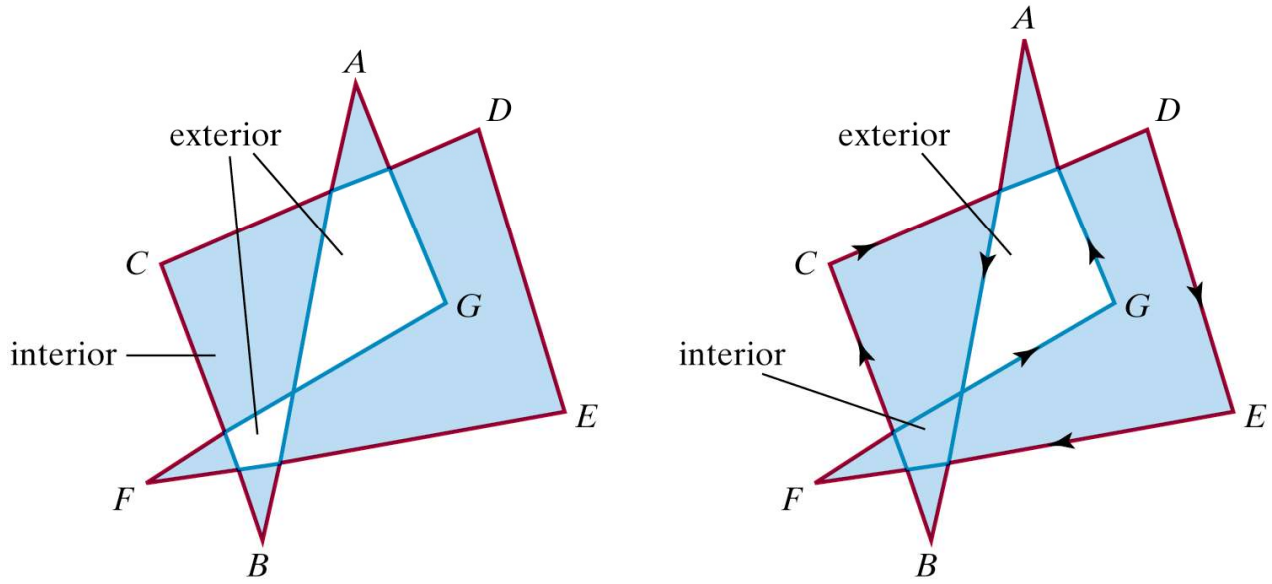


Figure 3-46

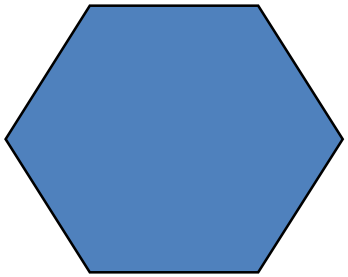
Identifying interior and exterior regions of a closed polyline that contains self-intersecting segments.

Polygon Filling

Types of filling

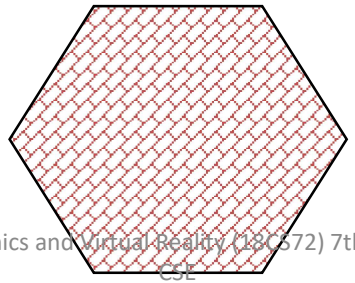
- **Solid-fill**

All the pixels inside the polygon's boundary are illuminated.



- **Pattern-fill**

the polygon is filled with an arbitrary predefined pattern.



Polygon Representation

•The polygon can be represented by listing its n vertices in an ordered list.

$$\mathbf{P = \{(x_1, y_1), (x_2, y_2), \ldots\ldots, (x_n, y_n)\}.}$$

•The polygon can be displayed by drawing a line between $\mathbf{(x_1, y_1)}$, and $\mathbf{(x_2, y_2)}$, then a line between $\mathbf{(x_2, y_2)}$, and $\mathbf{(x_3, y_3)}$, and so on until the end vertex. In order to close up the polygon, a line between $\mathbf{(x_n, y_n)}$, and $\mathbf{(x_1, y_1)}$ must be drawn.

•One problem with this representation is that if we wish to translate the polygon, it is necessary to apply the translation transformation to each vertex in order to obtain the translated polygon.

Polygon Representation

•For objects described by many polygons with many vertices, this can be a time consuming process.

• One method for reducing the computational time is to represent the polygon by the (**absolute**) **location** of its first vertex, and represent subsequent vertices as **relative positions** from the previous vertex. This enables us to translate the polygon simply by changing the coordinates of the first vertex.

Inside-Outside Tests

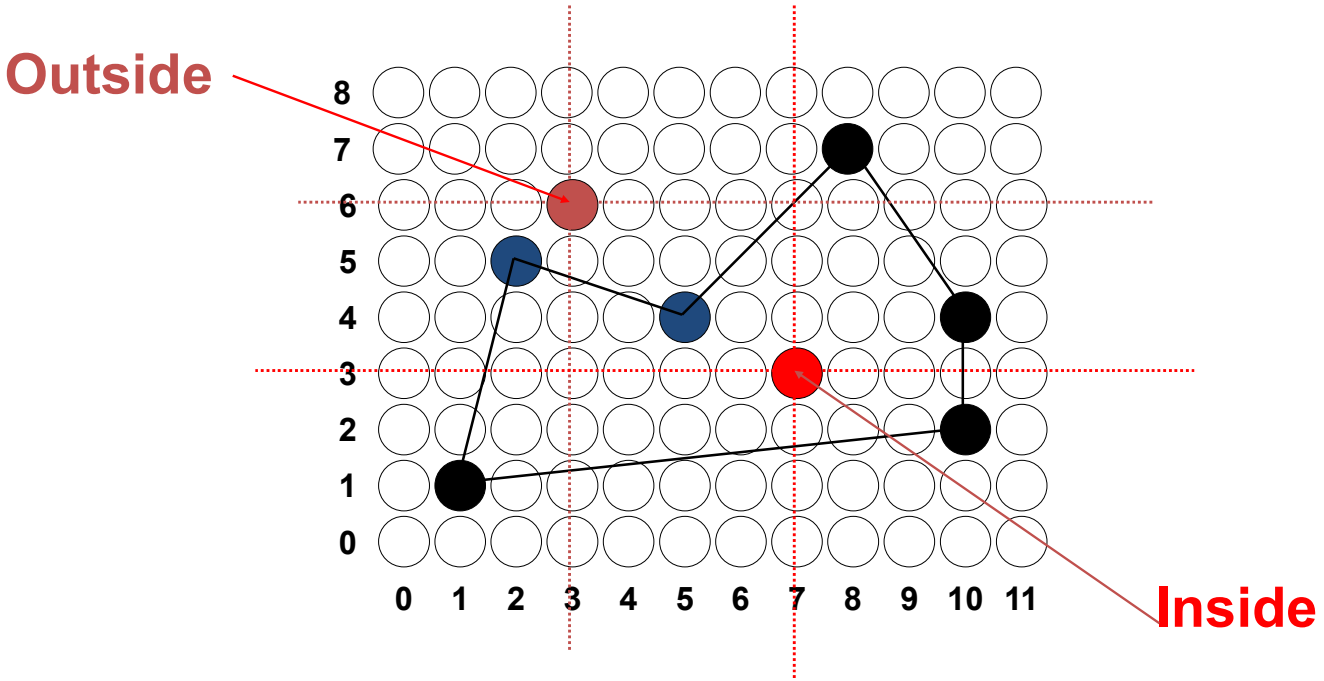
•when filling polygons we should decide whether a particular point is interior or exterior to a polygon.

•A rule called the **odd-parity** (or the **odd-even rule**) is applied to test whether a point is interior or not.

•To apply this rule, we conceptually draw a line starting from the particular point and extending to a distance point outside the coordinate extends of the object in any direction such that **no polygon vertex intersects** with **the line**.

Inside-Outside Tests

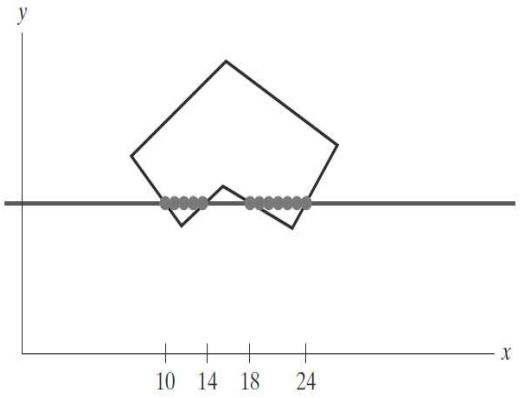
The point is considered to be **interior** if the number of intersections between the line and the polygon edges is **odd**. Otherwise, The point is exterior point.

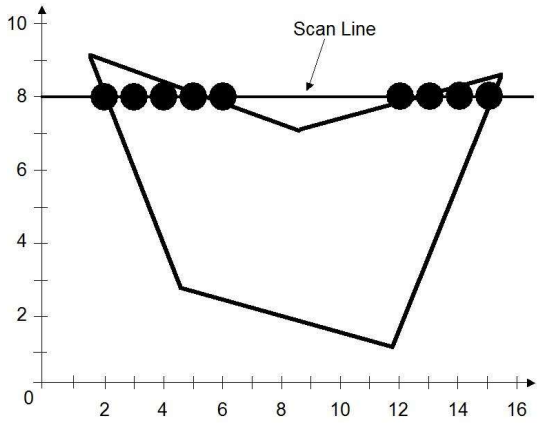


The Scan-Line Polygon Fill Algorithm

The scan-line polygon-filling algorithm involves

- the **horizontal scanning** of the polygon from its **lowermost** to its **topmost** vertex,
- identifying which edges intersect the scan-line,
- and finally drawing the interior horizontal lines with the specified fill color process.



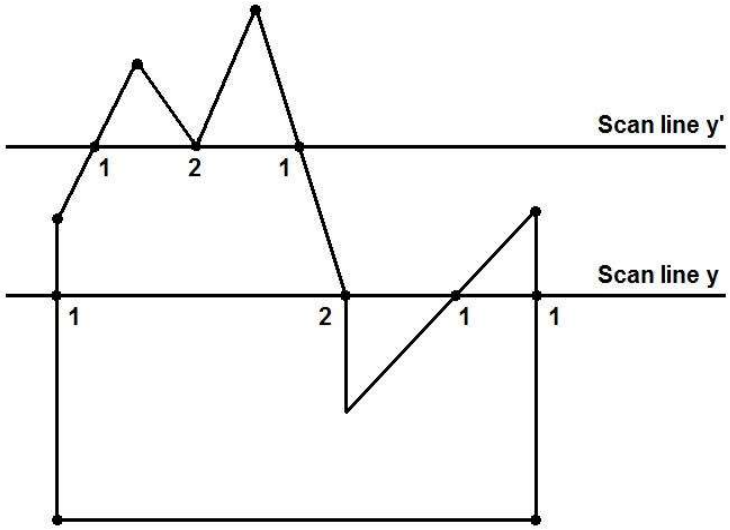


Figures--Interior pixels along a scan line passing through a polygon fill area

--- illustrates the basic scan-line procedure for a solid-color fill of a polygon.

The Scan-Line Polygon Fill Algorithm

Dealing with vertices



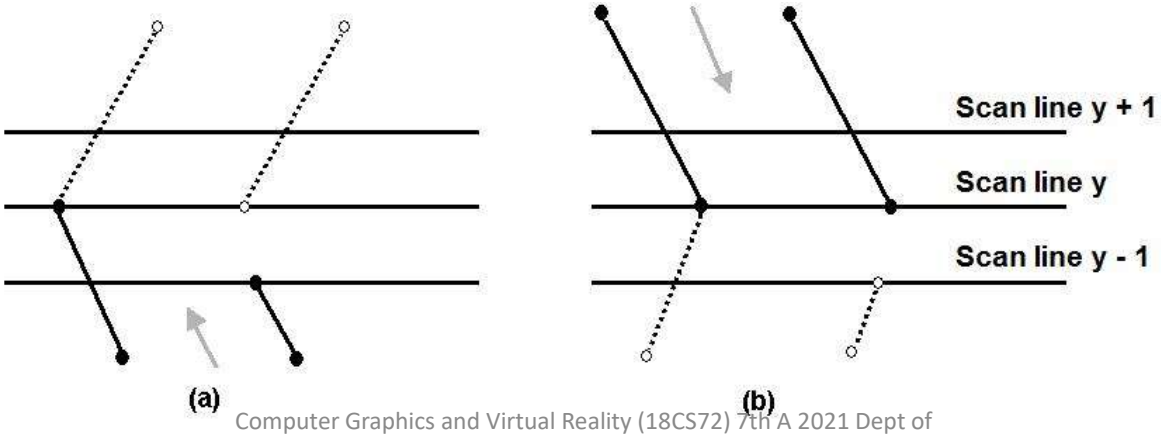
Intersection points along scan lines that intersect polygon vertices.

Scan line y generates an odd number of intersections, but **scan line y'** generates an even number of intersections that can be paired to identify correctly the interior pixel spans.

The Scan-Line Polygon Fill Algorithm

Dealing with vertices

- When the endpoint **y** coordinates of the two edges are **increasing**, the **y** value of the upper endpoint for the **current edge** is decreased by one (a)
- When the endpoint **y** values are **decreasing**, the **y** value of the **next edge** is decreased by one (b)



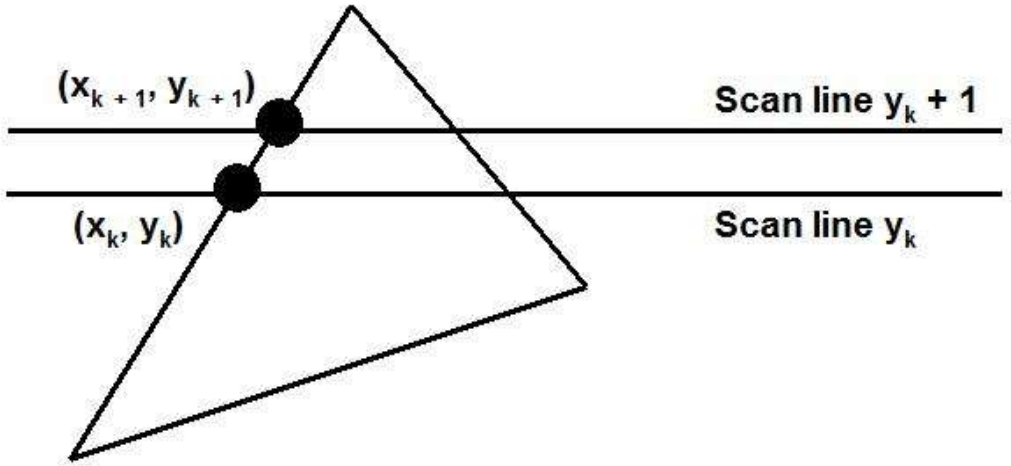
The Scan-Line Polygon Fill Algorithm

Determining Edge Intersections

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

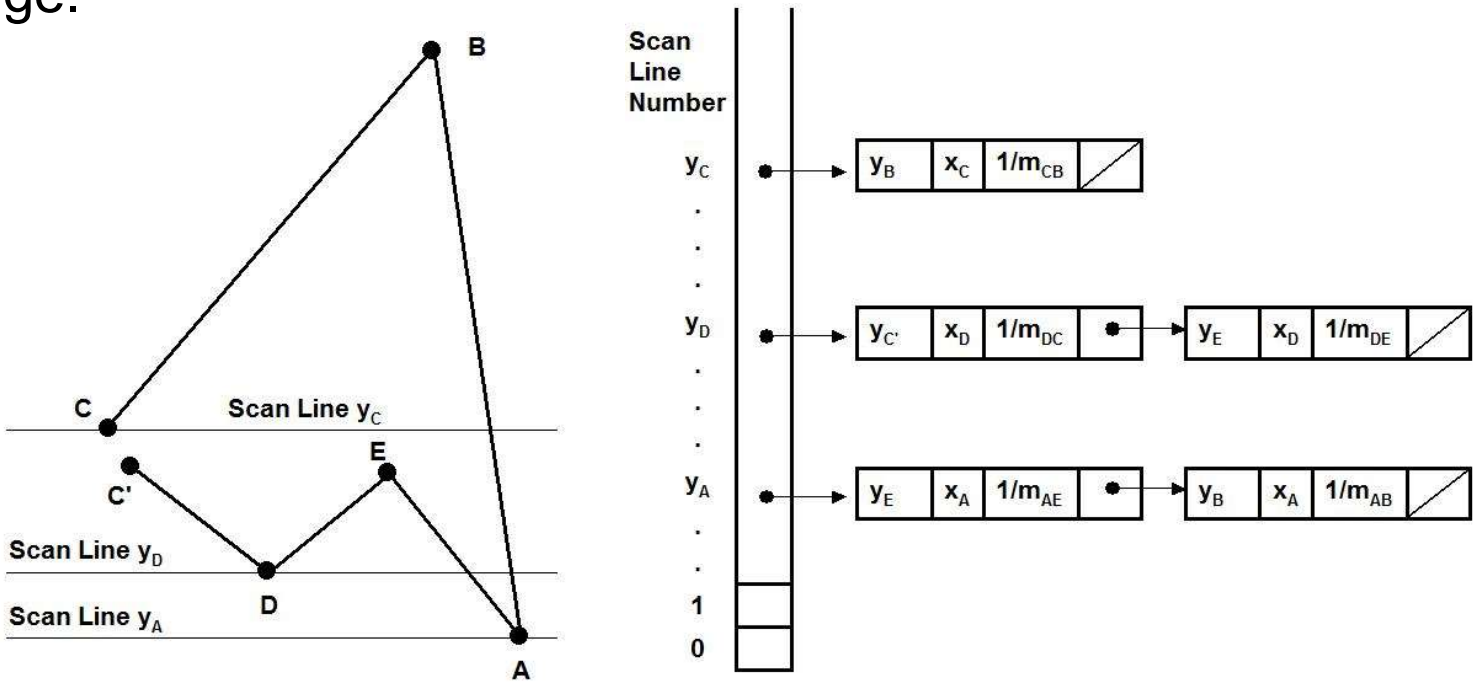
$$y_{k+1} - y_k = 1$$

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



The Scan-Line Polygon Fill Algorithm

• Each **entry** in the table for a particular scan line contains the **maximum y** value for that edge, the **x-intercept** value (**at the lower vertex**) for the edge, and the **inverse slope** of the edge.

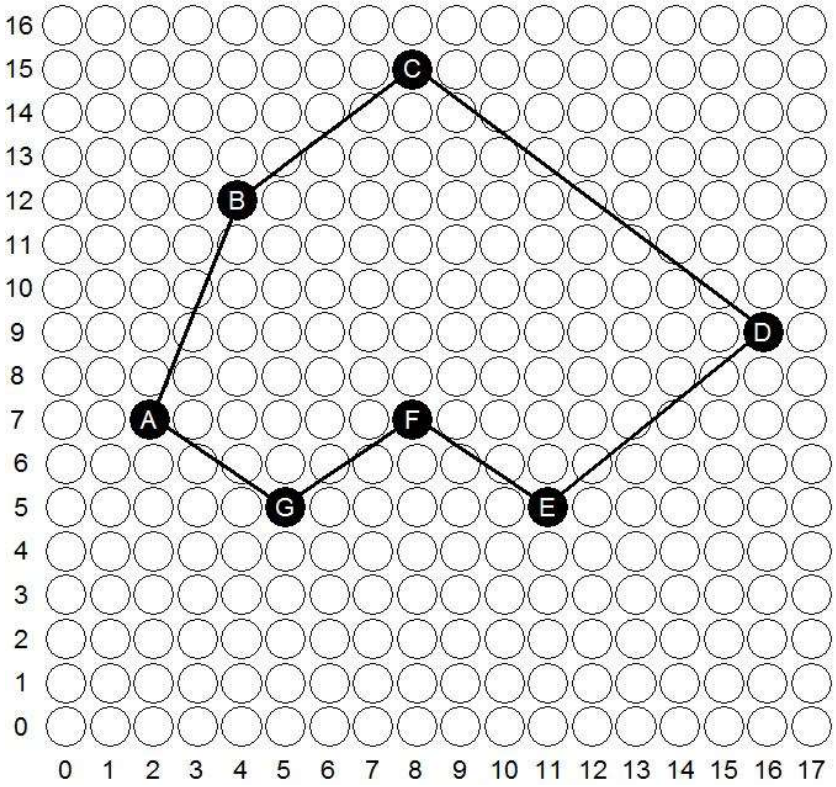


A polygon and its sorted edge table, with edge $\overline{DC}$ shortened by one unit in the y direction.

The Scan-Line Polygon Fill Algorithm

(Example) Polygon = {A, B, C, D, E, F, G}

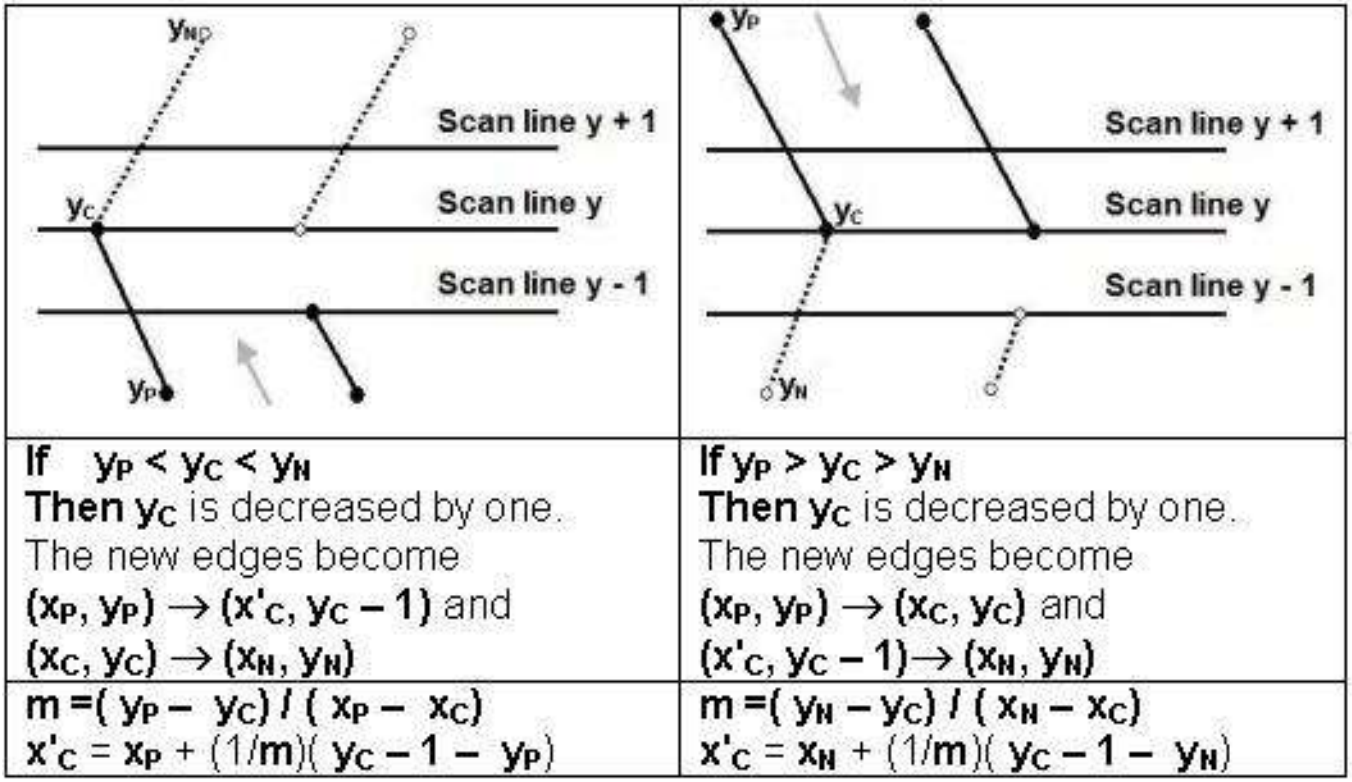
Polygon = {(2, 7), (4, 12), (8,15), (16, 9), (11, 5), (8, 7), (5, 5)}



Edge Table						
#	Edge		1/m	y_{min}	x	y_{max}
0	A (2, 7)	B (4, 12)	2/5	7	2	12
1	B (4, 12)	C (8,15)	4/3	12	4	15
2	C (8,15)	D (16, 9)	– 8/6	9	16	15
3	D (16, 9)	E (11, 5)	5/4	5	11	9
4	E (11, 5)	F (8, 7)	– 3/2	5	11	7
5	F (8, 7)	G (5, 5)	3/2	5	5	7
6	G (5, 5)	A (2, 7)	– 3/2	5	5	7

The Scan-Line Polygon Fill Algorithm

(Example)



If $y_P < y_C < y_N$ Then y_C is decreased by one. The new edges become $(x_P, y_P) \rightarrow (x'_C, y_C - 1)$ and $(x_C, y_C) \rightarrow (x_N, y_N)$	If $y_P > y_C > y_N$ Then y_C is decreased by one. The new edges become $(x_P, y_P) \rightarrow (x_C, y_C)$ and $(x'_C, y_C - 1) \rightarrow (x_N, y_N)$
$m = (y_P - y_C) / (x_P - x_C)$ $x'_C = x_P + (1/m)(y_C - 1 - y_P)$	$m = (y_N - y_C) / (x_N - x_C)$ $x'_C = x_N + (1/m)(y_C - 1 - y_N)$

The Scan-Line Polygon Fill Algorithm (Example)

Previous Vertex	Current Vertex	Next Vertex	y_P ? y_C ? y_N	Current Vertex Type	Action
G (5, **5**)	**A** (2, **7**)	B (4, **12**)	$y_P < y_C < y_N$	**Not local extremum**	Split **A**
A (2, **7**)	**B** (4, **12**)	C (8, **15**)	$y_P < y_C < y_N$	**Not local extremum**	Split **B**
B (4, **12**)	**C** (8, **15**)	D (16, **9**)	$y_P < y_C > y_N$	**Local Maximum**	None
C (8, **15**)	**D** (16, **9**)	E (11, **5**)	$y_P > y_C > y_N$	**Not local extremum**	Split **D**
D (16, **9**)	**E** (11, **5**)	F (8, **7**)	$y_P > y_C < y_N$	**Local Minimum**	None
E (11, **5**)	**F** (8, **7**)	G (5, **5**)	$y_P < y_C > y_N$	**Local Maximum**	None
F (8, **7**)	**G** (5, **5**)	A (2, **7**)	$y_P > y_C < y_N$	**Local Minimum**	None

- **Vertex A** should be split into two vertices **A'** ($x_{A'}$, **6**) and **A(2, 7)**

$$m = (5 - 7)/(5 - 2) = -2/3$$

$$x'_A = 5 + (-3/2)(7 - 1 - 5) = 7/2 = 3.5 \cong 4$$

The vertex **A** is split to **A'** (**4, 6**) and **A(2, 7)**

The Scan-Line Polygon Fill Algorithm

(Example)

Previous Vertex	Current Vertex	Next Vertex	y_P ? y_C ? y_N	Current Vertex Type	Action
G (5, **5**)	**A** (2, **7**)	B (4, **12**)	$y_P < y_C < y_N$	**Not local extremum**	Split **A**
A (2, **7**)	**B** (4, **12**)	C (8, **15**)	$y_P < y_C < y_N$	**Not local extremum**	Split **B**
B (4, **12**)	**C** (8, **15**)	D (16, **9**)	$y_P < y_C > y_N$	**Local Maximum**	None
C (8, **15**)	**D** (16, **9**)	E (11, **5**)	$y_P > y_C > y_N$	**Not local extremum**	Split **D**
D (16, **9**)	**E** (11, **5**)	F (8, **7**)	$y_P > y_C < y_N$	**Local Minimum**	None
E (11, **5**)	**F** (8, **7**)	G (5, **5**)	$y_P < y_C > y_N$	**Local Maximum**	None
F (8, **7**)	**G** (5, **5**)	A (2, **7**)	$y_P > y_C < y_N$	**Local Minimum**	None

- **Vertex B** should be split into two vertices **B'** ($x_{B'}$, **11**) and **B(4, 12)**

$$m = (7 - 12)/(2 - 4) = 5/2$$

$$x'_A = 2 + (2/5)(12 - 1 - 7) = 18/5 = 3.6 \cong 4$$

The vertex **B** is split to **B'** (**4, 11**) and **B(4, 12)**

The Scan-Line Polygon Fill Algorithm

(Example)

Previous Vertex	Current Vertex	Next Vertex	y_P ? y_C ? y_N	Current Vertex Type	Action
G (5, **5**)	**A** (2, **7**)	B (4, **12**)	$y_P < y_C < y_N$	**Not local extremum**	Split **A**
A (2, **7**)	**B** (4, **12**)	C (8, **15**)	$y_P < y_C < y_N$	**Not local extremum**	Split **B**
B (4, **12**)	**C** (8, **15**)	D (16, **9**)	$y_P < y_C > y_N$	**Local Maximum**	None
C (8, **15**)	**D** (16, **9**)	E (11, **5**)	$y_P > y_C > y_N$	**Not local extremum**	Split **D**
D (16, **9**)	**E** (11, **5**)	F (8, **7**)	$y_P > y_C < y_N$	**Local Minimum**	None
E (11, **5**)	**F** (8, **7**)	G (5, **5**)	$y_P < y_C > y_N$	**Local Maximum**	None
F (8, **7**)	**G** (5, **5**)	A (2, **7**)	$y_P > y_C < y_N$	**Local Minimum**	None

- **Vertex D** should be split into two vertices **D(16, 9)** and **D'** ($x_{D'}$, **8**)

$$m = (5 - 9)/(11 - 16) = 4/5$$

$$x'_D = 11 + (5/4)(9 - 1 - 5) = 59/4 = 14.75 \cong 15$$

The vertex **D** is split to **D(16, 9)** and **D' (15, 8)**

The Scan-Line Polygon Fill Algorithm

(Example)

Modified Edge Table						
#	Edge		1/m	y_{min}	x	y_{max}
0	A (2, 7)	B' (4, 11)	2/5	7	2	11
1	B (4, 12)	C (8,15)	4/3	12	4	15
2	C (8,15)	D (16, 9)	– 8/6	9	16	15
3	D' (15, 8)	E (11, 5)	5/4	5	11	8
4	E (11, 5)	F (8, 7)	– 3/2	5	11	7
5	F (8, 7)	G (5, 5)	3/2	5	5	7
6	G (5, 5)	A' (4, 6)	– 3/2	5	5	6

Activation Table											
y	5	6	7	8	9	10	11	12	13	14	15
Activated Edge #s	3, 4, 5, 6		0		2			1			

The Scan-Line Polygon Fill Algorithm

(Example)

Edge number 0

#	Edge		1/m	y_{min}	x	y_{max}
0	A (2, 7)	B' (4, 11)	2/5 = 0.4	7	2	11

Scan line	x-intersection
y = 7	2
y = 8	2 + 0.4 = 2.4 ~ 2
y = 9	2.4 + 0.4 = 2.8 ~ 3
y = 10	2.8 + 0.4 = 3.2 ~ 3
y = 11	4

Edge number 1

#	Edge		1/m	y_{min}	x	y_{max}
1	B (4, 12)	C (8,15)	4/3 = 1.3	12	4	15

Scan line	x-intersection
y = 12	4
y = 13	4 + 1.3 = 4.3 ~ 4
y = 14	4.3 + 1.3 = 5.6 ~ 6
y = 15	8

The Scan-Line Polygon Fill Algorithm

(Example)

Edge number 2

#	Edge		1/m	y_{min}	x	y_{max}
2	C (8,15)	D (16, 9)	– 8/6 = –1.3	9	16	15

Scan line	x-intersection
y = 9	16
y = 10	16 – 1.3 = 14.7 ~ 15
y = 11	14.7 – 1.3 = 13.4 ~ 13
y = 12	13.4 – 1.3 = 12.1 ~ 12
y = 13	12.1 – 1.3 = 10.8 ~ 11
y = 14	10.8 – 1.3 = 9.5 ~ 10
y = 15	8

Edge number 3

#	Edge		1/m	y_{min}	x	y_{max}
3	D' (15, 8)	E (11, 5)	5/4 = 1.25	5	11	8

Scan line	x-intersection
y = 5	11
y = 6	11 + 1.25 = 12.25 ~ 12
y = 7	12.25 +1.25 = 13.5 ~ 14
y = 8	15

The Scan-Line Polygon Fill Algorithm

(Example)

Edge number 4

#	Edge		1/m	y_{min}	x	y_{max}
4	E (11, 5)	F (8, 7)	– 3/2 = –1.5	5	11	7

Scan line	x-intersection
y = 5	11
y = 6	11 – 1.5 = 9.5 ~ 10
y = 7	8

Edge number 5

#	Edge		1/m	y_{min}	x	y_{max}
5	F (8, 7)	G (5, 5)	3/2 = 1.5	5	5	7

Scan line	x-intersection
y = 5	5
y = 6	5+ 1.5 = 6.5 ~ 7
y = 7	8

Edge number 6

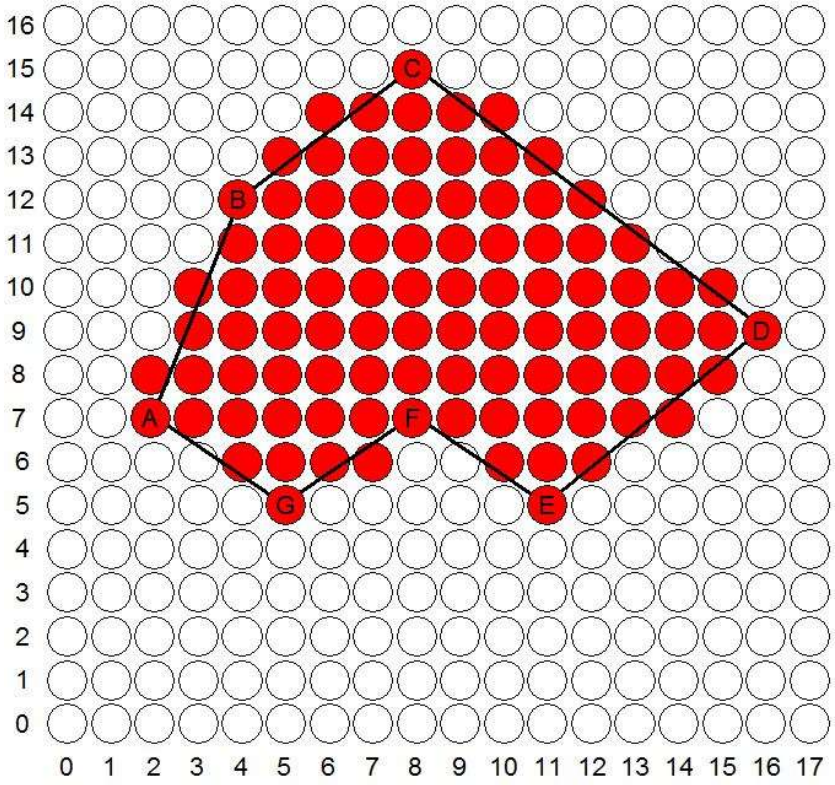
#	Edge		1/m	y_{min}	x	y_{max}
6	G (5, 5)	A' (4, 6)	– 3/2 = –1.5	5	5	6

Scan line	x-intersection
y = 5	5
y = 6	4

The Scan-Line Polygon Fill Algorithm

(Example)

Scan line	x-intersections Edge#							x-intersections pair Ascending order
	0	1	2	3	4	5	6	
5				11	11	5	5	(5, 5), (11, 11)
6				12	10	7	4	(4, 7), (10, 12)
7	2			14	8	8		(2, 8), (8,14)
8	2			15				(2,15)
9	3		16					(3, 16)
10	3		15					(3, 15)
11	4		13					(4, 13)
12		4	12					(4,12)
13		4	11					(4, 11)
14		6	10					(6, 10)
15		8	8					(8, 8)



Boundary Fill Algorithm

- Another approach to area filling is to start at a point inside a region and paint the interior outward toward the boundary.

- If the boundary is specified in a single color, the fill algorithm processed outward pixel by pixel until the boundary color is encountered.

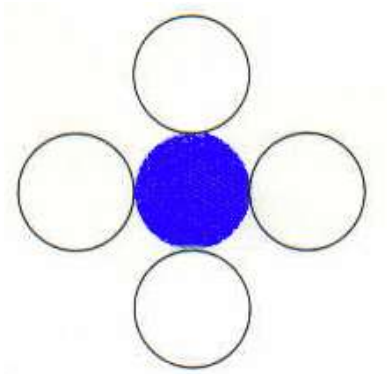
- It is useful in interactive painting packages, where interior points are easily selected.
- The inputs of the this algorithm are:
  - Coordinates of the interior point (x, y)
  - Fill Color
  - Boundary Color

Boundary-Fill Algorithm (cont.)

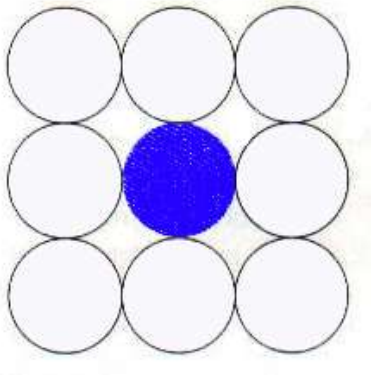
- Starting from (x, y), the algorithm tests neighboring pixels to determine whether they are of the boundary color.
- If not, they are painted with the fill color, and their neighbors are tested. This process continues until all pixels up to the boundary have been tested.
- There are two methods for proceeding to neighboring pixels from the current test position:

Boundary-Fill Algorithm (cont.)

1. The 4-connected method.



2. The 8-connected method.



Boundary Fill Algorithm

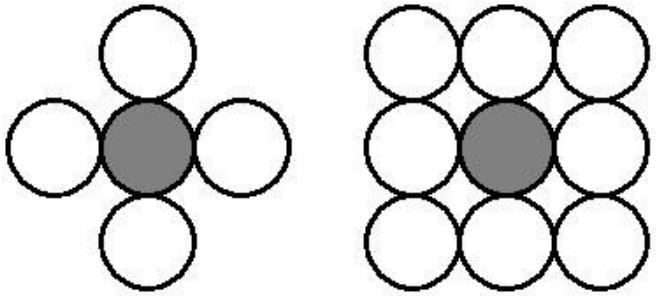
The following steps illustrate the idea of the **recursive** boundary-fill algorithm:

1. Start from an interior point.
2. If the current pixel is **not already** filled and if it is not an edge point, then set the pixel with the fill color, and store its neighboring pixels (**4** or **8-connected**) in the stack for processing. Store only neighboring pixel that is **not already** filled and is not an edge point.
3. Select the next pixel from the stack, and continue with step **2**.

```
void boundaryFill4 (int x, int y, int fillColor, int borderColor)
{   int interiorColor;
    /* set current color to fillColor, then perform following operations. */
    getPixel (x, y, interiorColor);
    if ( (interiorColor != borderColor)  &&  (interiorColor != fillColor) )
    {
       setPixel (x, y);   // set color of pixel to fillColor
       boundaryFill4  (x + 1, y, fillColor, borderColor);
       boundaryFill4 (x - 1,   y, fillColor, borderColor);
       boundaryFill4 (x, y + 1,  fillColor, borderColor);
       boundaryFill4 (x, y - 1,   fillColor, borderColor);
    }
}
```

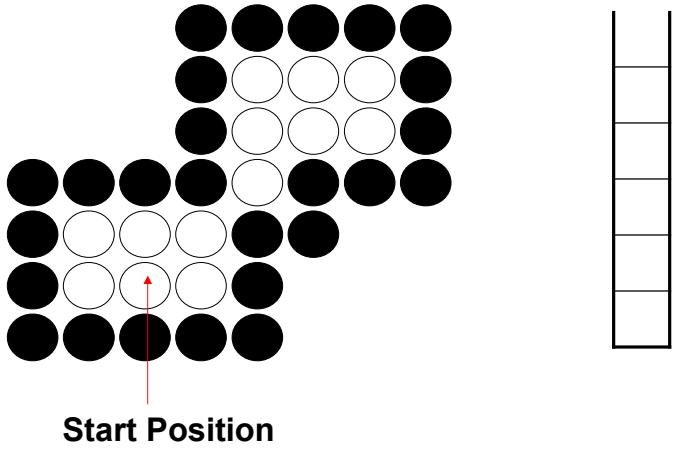
Boundary Fill Algorithm

•The order of pixels that should be added to stack using **4-connected** is above, below, left, and right.
•For **8-connected** is above, below, left, right, above-left, above-right, below-left, and below-right.



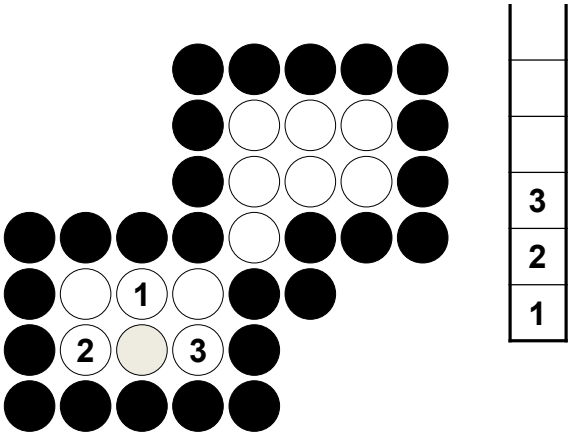
Boundary Fill Algorithm

4-connected (Example)



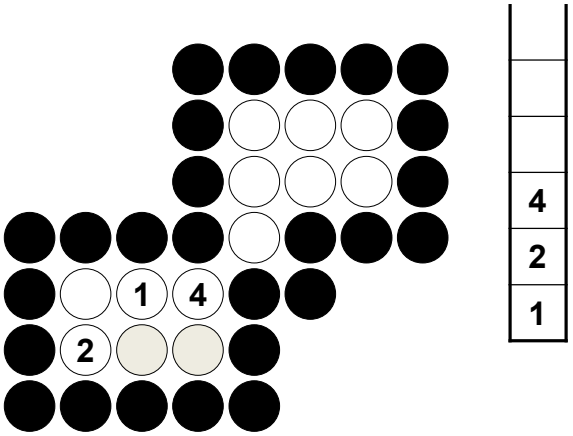
Boundary Fill Algorithm

4-connected (Example)



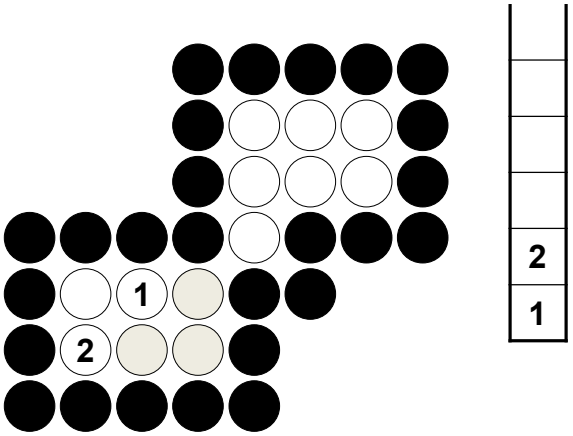
Boundary Fill Algorithm

4-connected (Example)



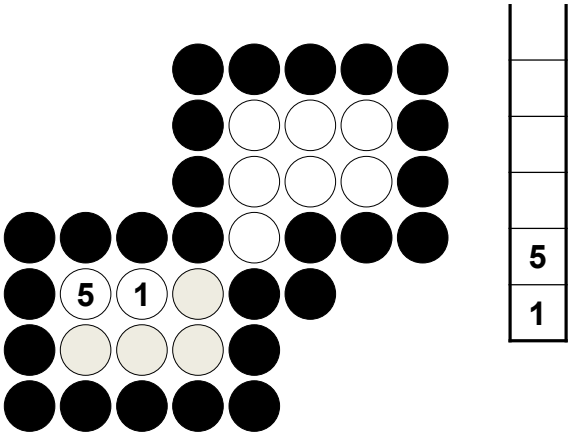
Boundary Fill Algorithm

4-connected (Example)



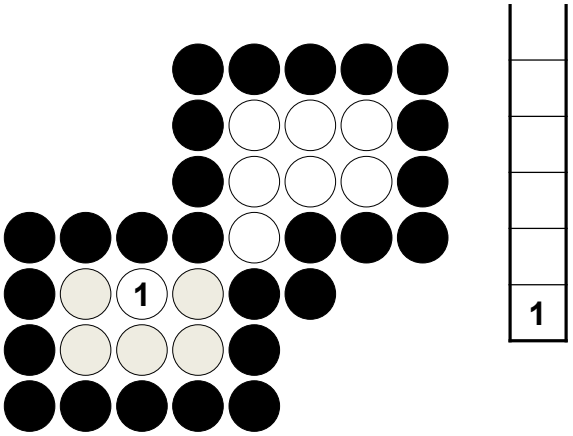
Boundary Fill Algorithm

4-connected (Example)



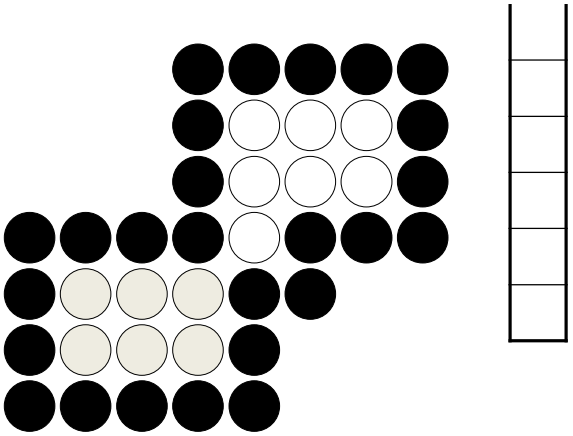
Boundary Fill Algorithm

4-connected (Example)



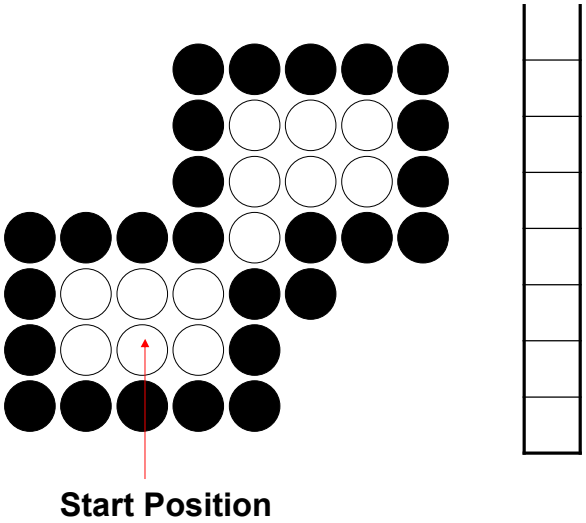
Boundary Fill Algorithm

4-connected (Example)



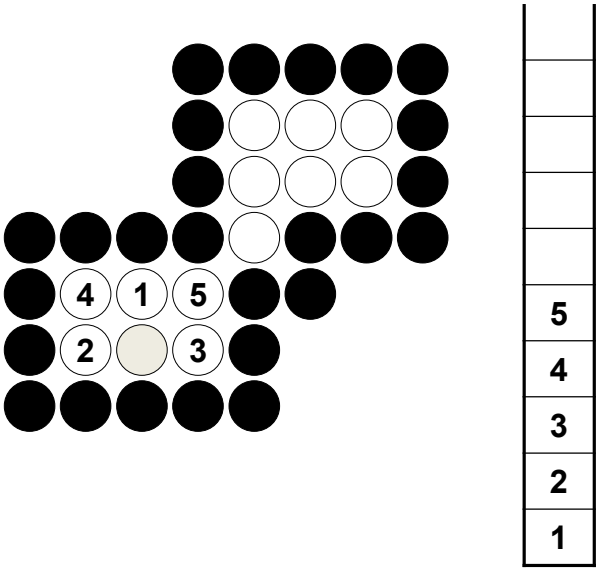
Boundary Fill Algorithm

8-connected (Example)



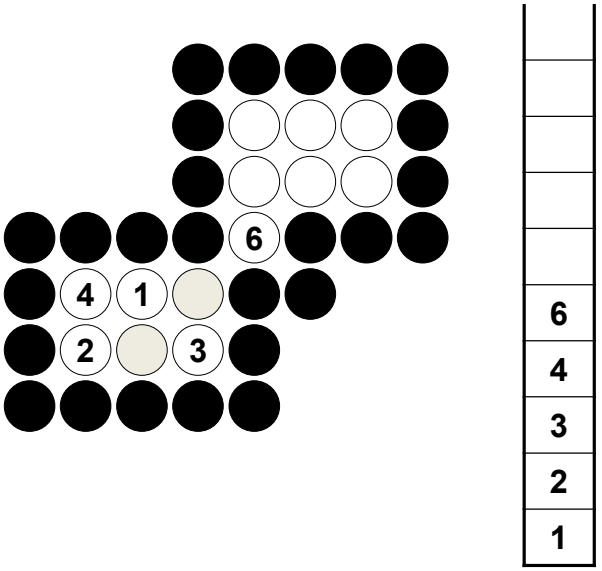
Boundary Fill Algorithm

8-connected (Example)



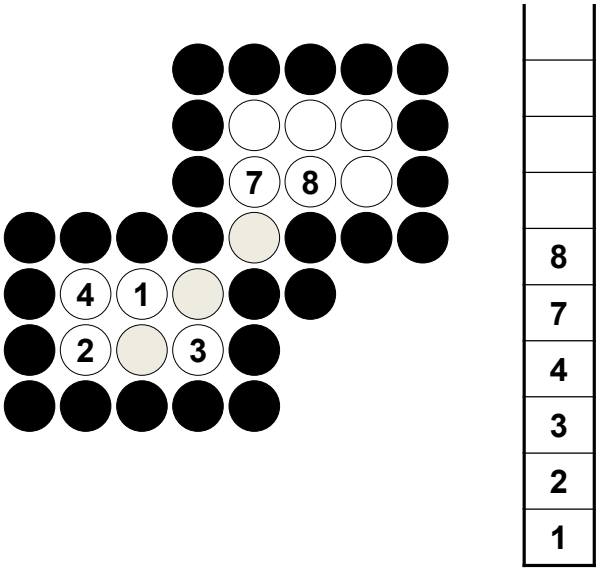
Boundary Fill Algorithm

8-connected (Example)



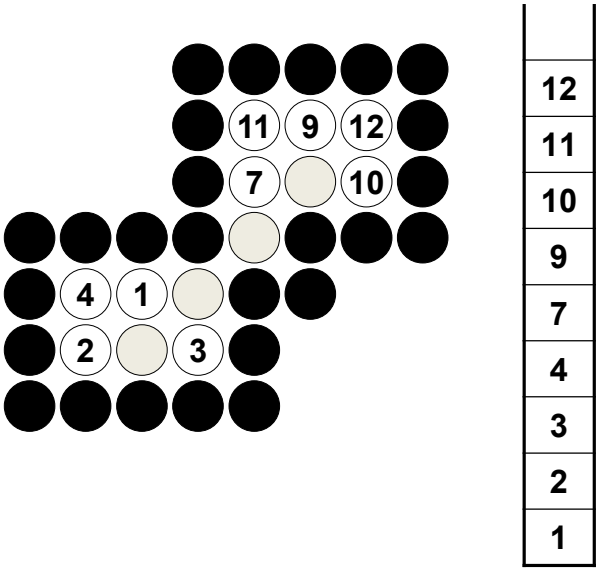
Boundary Fill Algorithm

8-connected (Example)



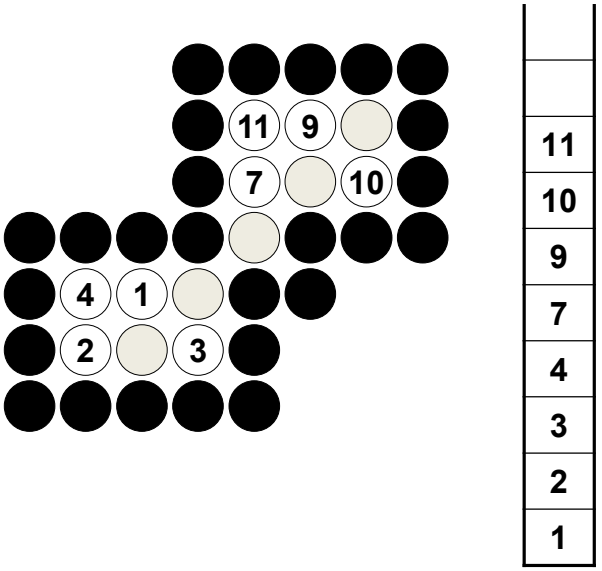
Boundary Fill Algorithm

8-connected (Example)



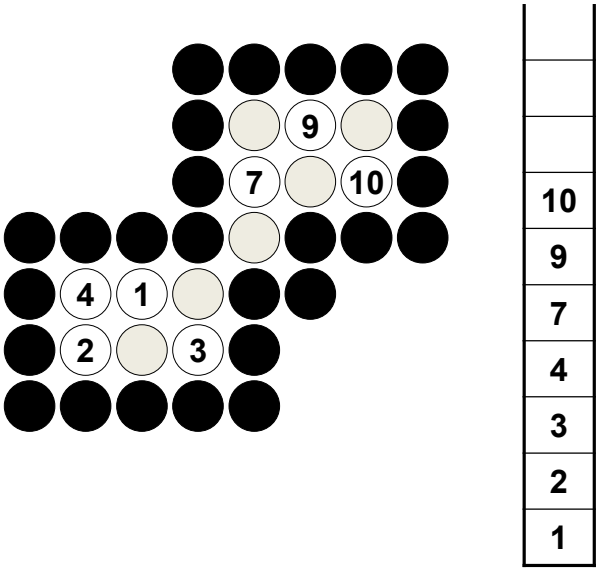
Boundary Fill Algorithm

8-connected (Example)



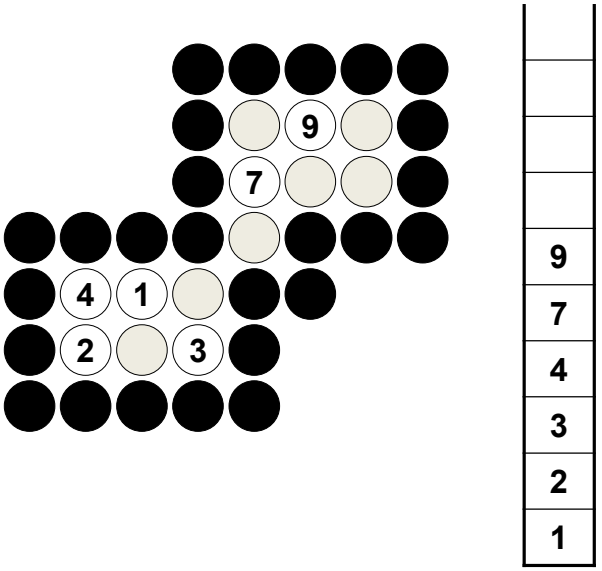
Boundary Fill Algorithm

8-connected (Example)



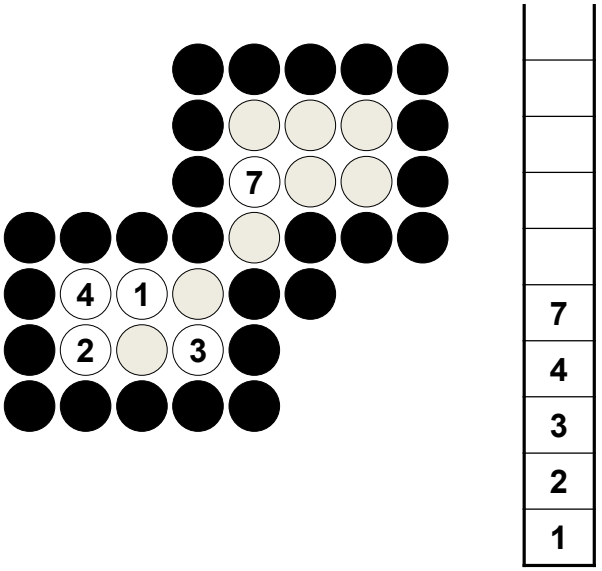
Boundary Fill Algorithm

8-connected (Example)



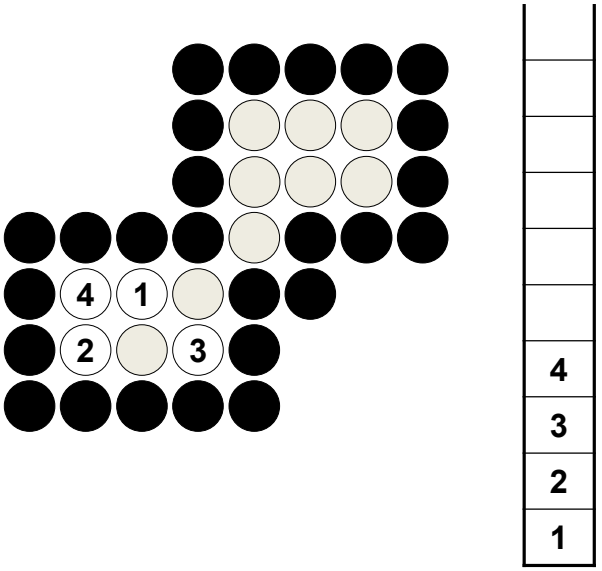
Boundary Fill Algorithm

8-connected (Example)



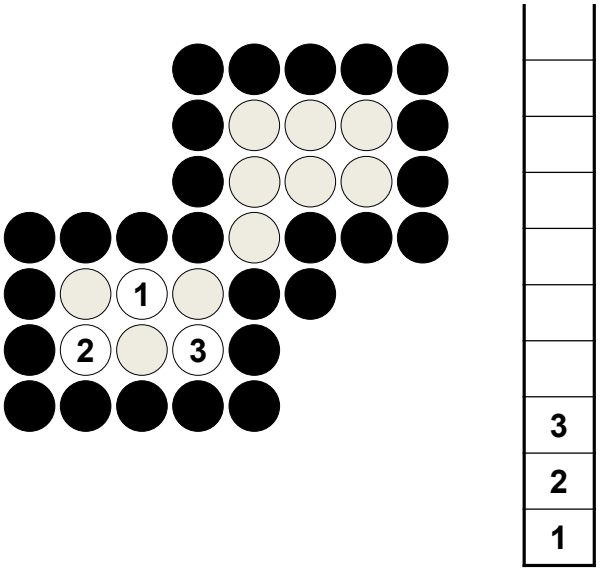
Boundary Fill Algorithm

8-connected (Example)



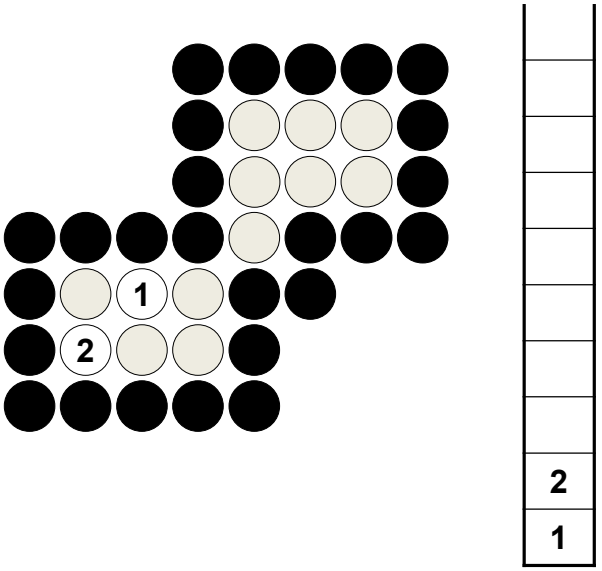
Boundary Fill Algorithm

8-connected (Example)



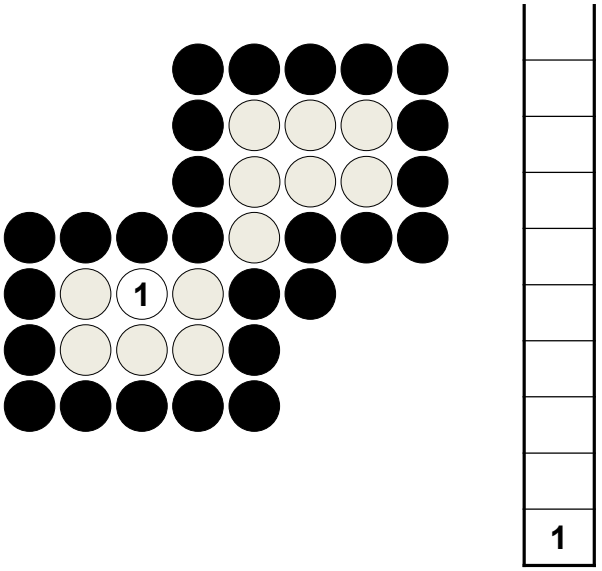
Boundary Fill Algorithm

8-connected (Example)



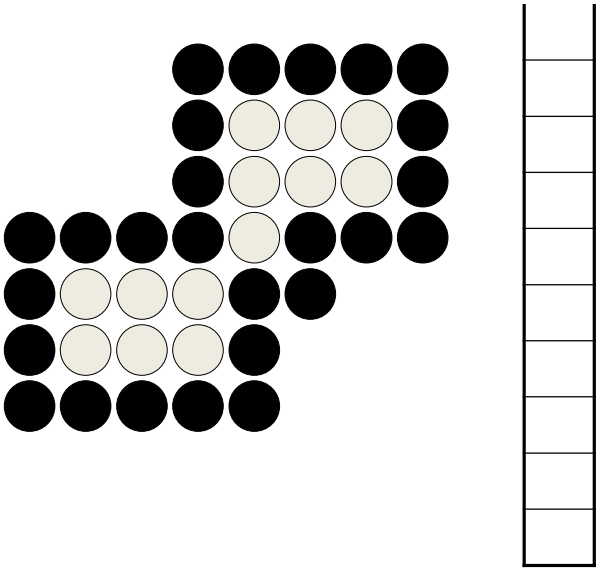
Boundary Fill Algorithm

8-connected (Example)



Boundary Fill Algorithm

8-connected (Example)



Boundary-Fill Algorithm (cont.)

- 4-connected and 8-connected methods involve heavy recursion which may consume memory and time.
  - Since the previous procedure requires considerable stacking of neighboring pixels, more **efficient methods** are generally employed.
- More efficient methods are used.
  - These methods fill horizontal pixel spans across scan line.
  - This called a **Pixel Span method.**
- We need only stack a beginning position for each horizontal pixel span, instead of stacking all unprocessed neighboring positions around the current position (where spans are defined as the contiguous horizontal string of positions).
- These methods (**Span Flood-Fill**) **fill horizontal pixel spans across scan lines**, instead of proceeding to 4-connected or 8-connected neighboring pixels.

Span Flood-Fill Algorithm

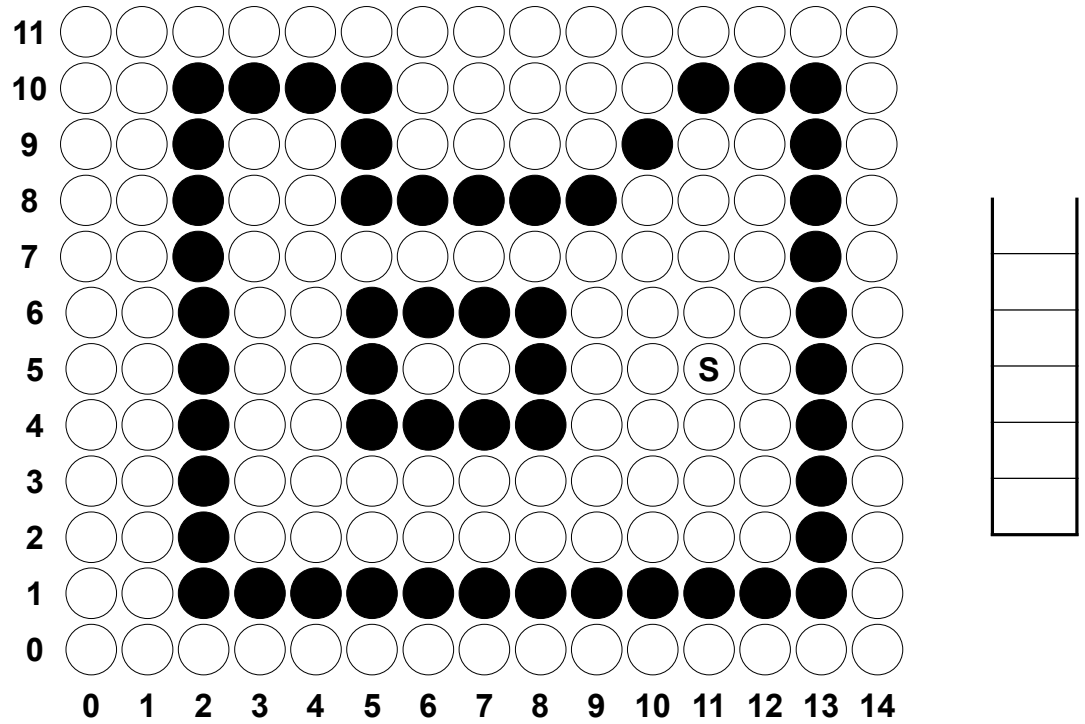
The algorithm is summarized as follows:

- **Starting** from the initial interior pixel, then fill in the contiguous span of pixels on this starting scan line.

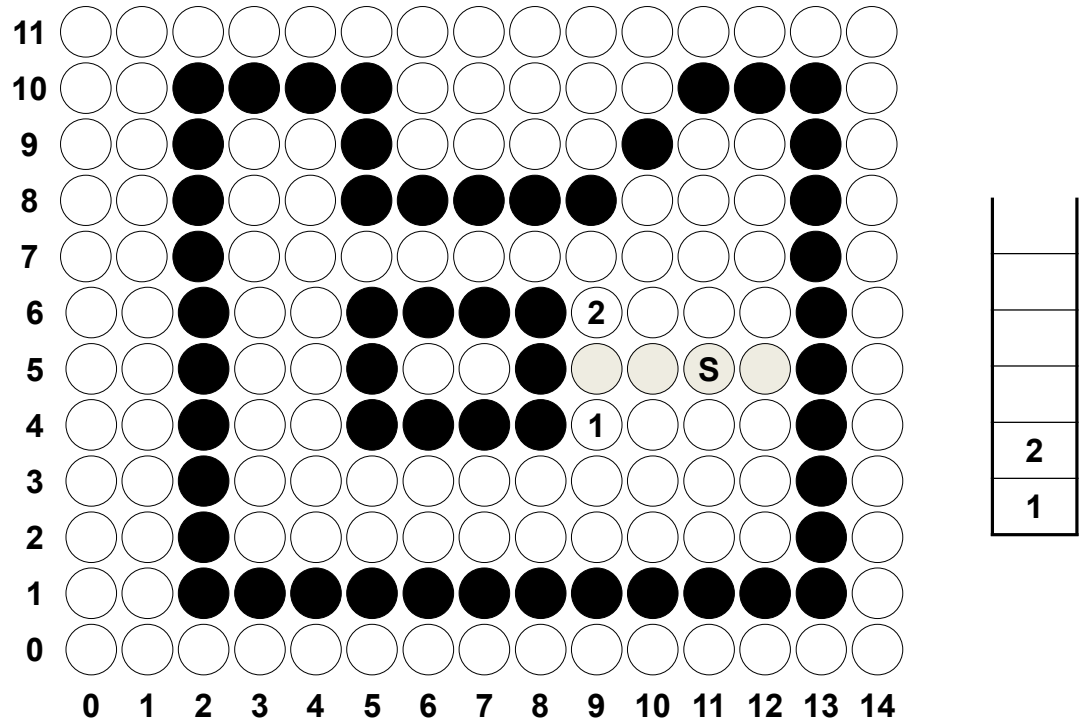
- **Then locate** and **stack** starting positions for spans on the adjacent scan lines, where spans are defined as the contiguous horizontal string of positions bounded by pixels displayed in the area border color.

- **At each subsequent step**, **unstack** the next start position and repeat the process.

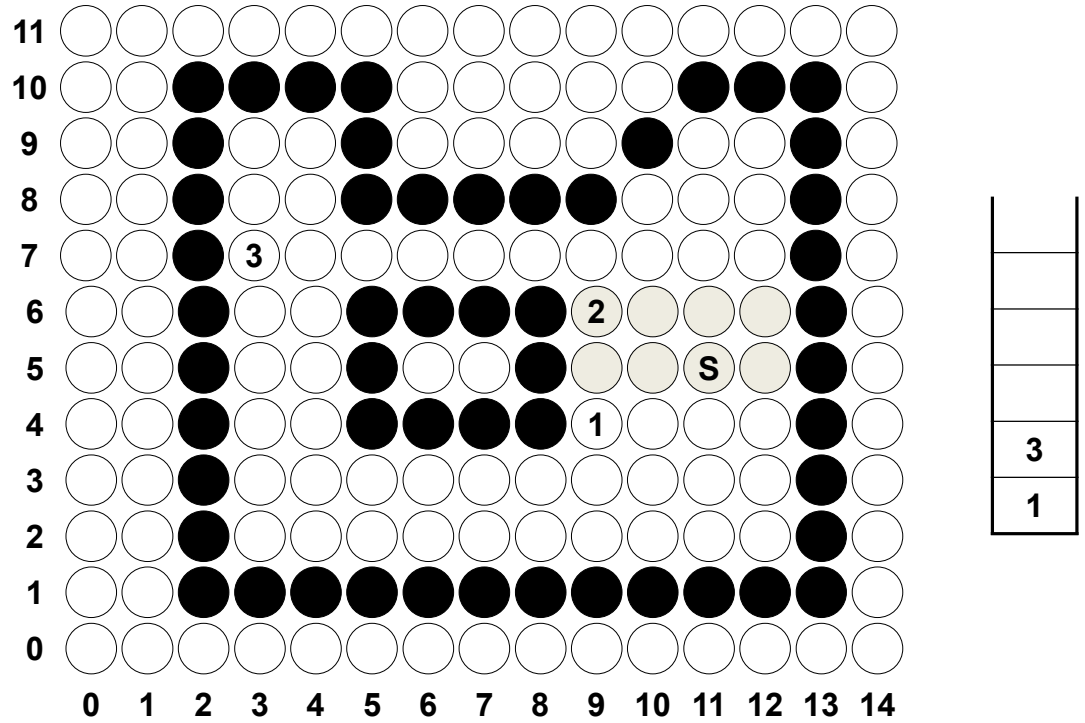
Span Flood-Fill Algorithm (example)



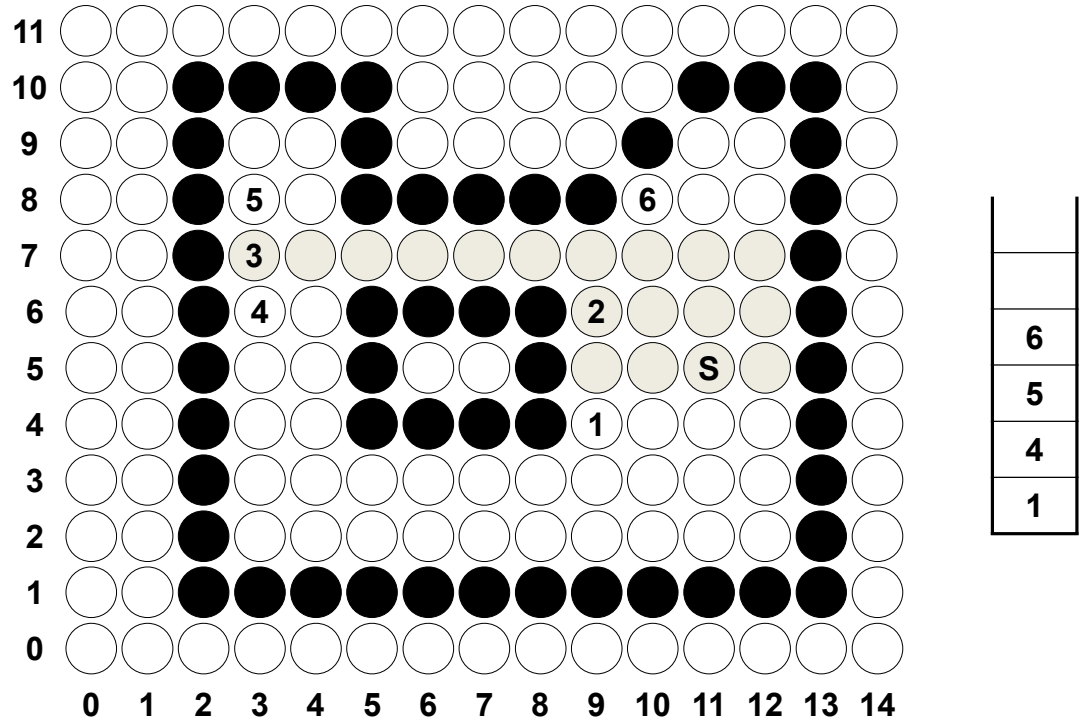
Span Flood-Fill Algorithm (example)



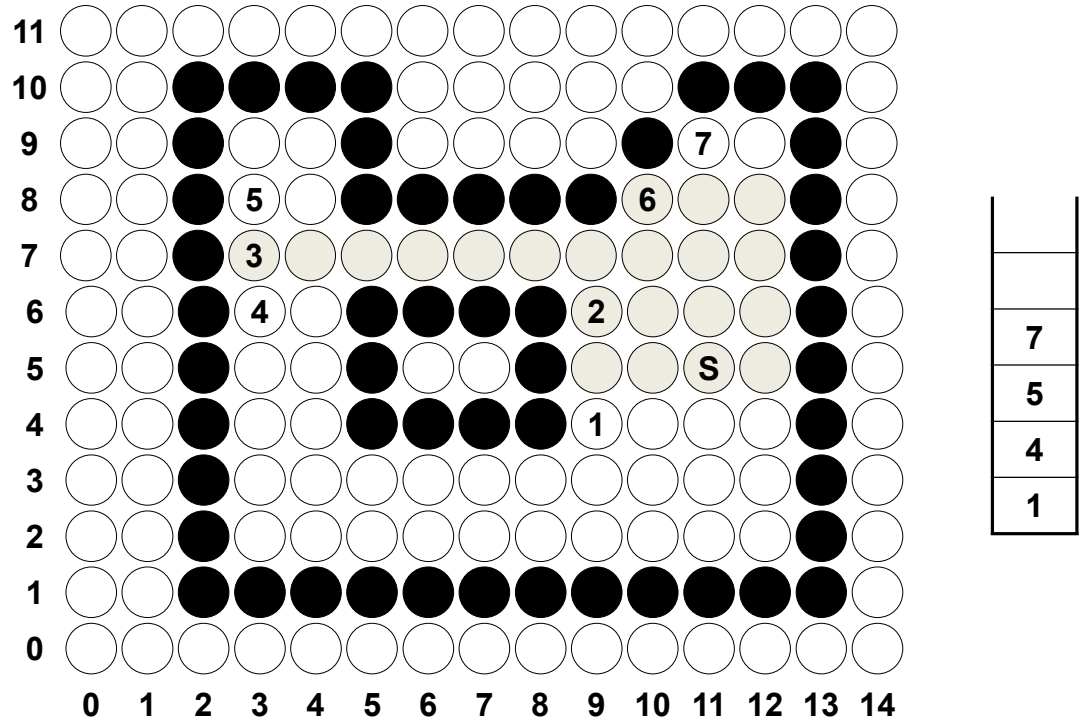
Span Flood-Fill Algorithm (example)



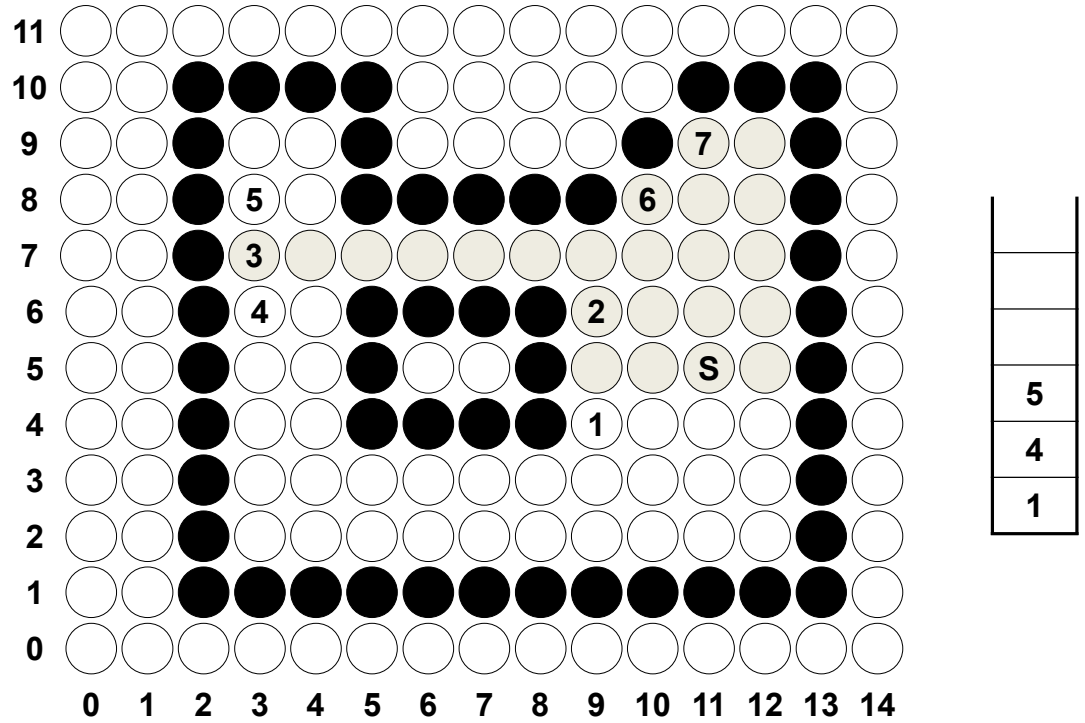
Span Flood-Fill Algorithm (example)



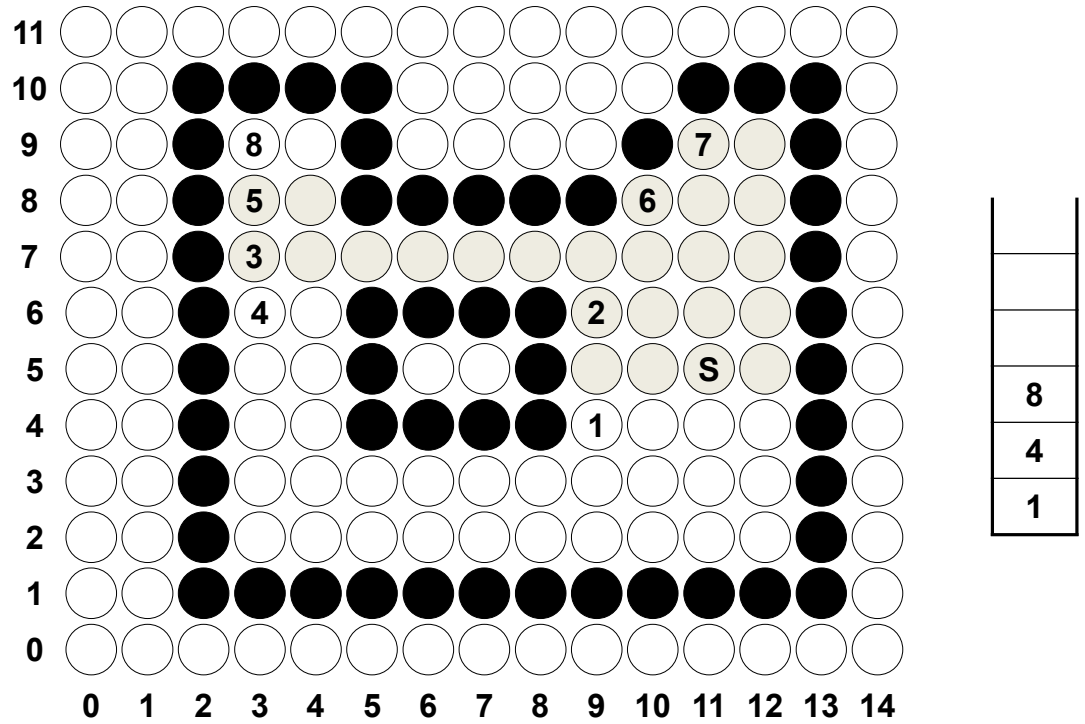
Span Flood-Fill Algorithm (example)



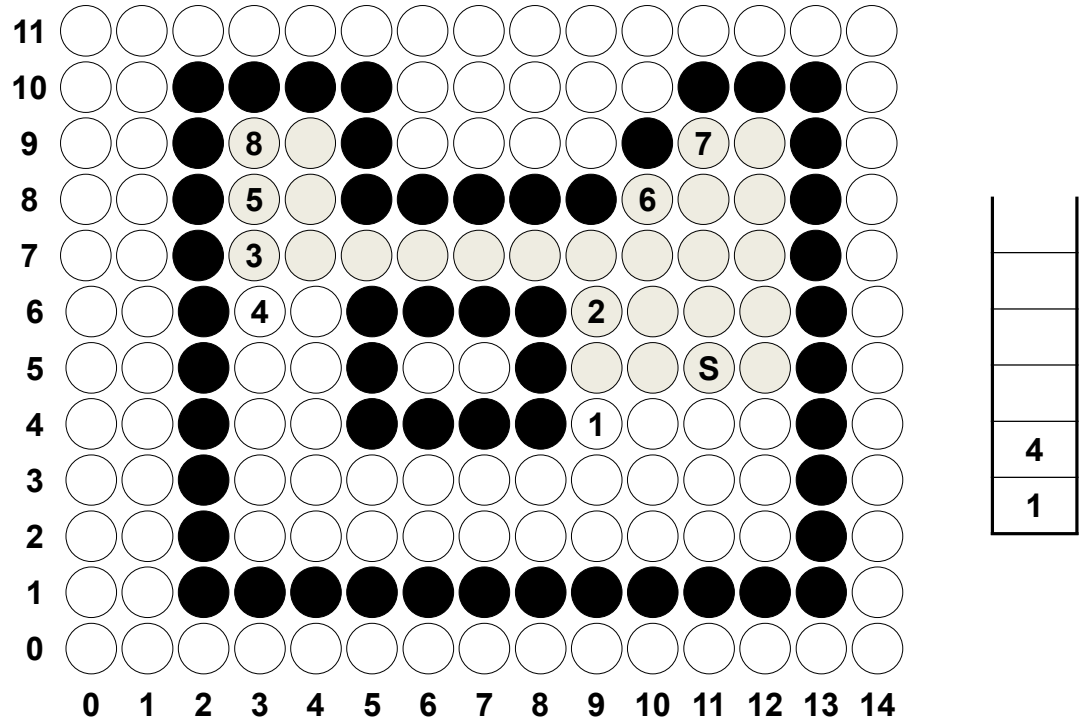
Span Flood-Fill Algorithm (example)



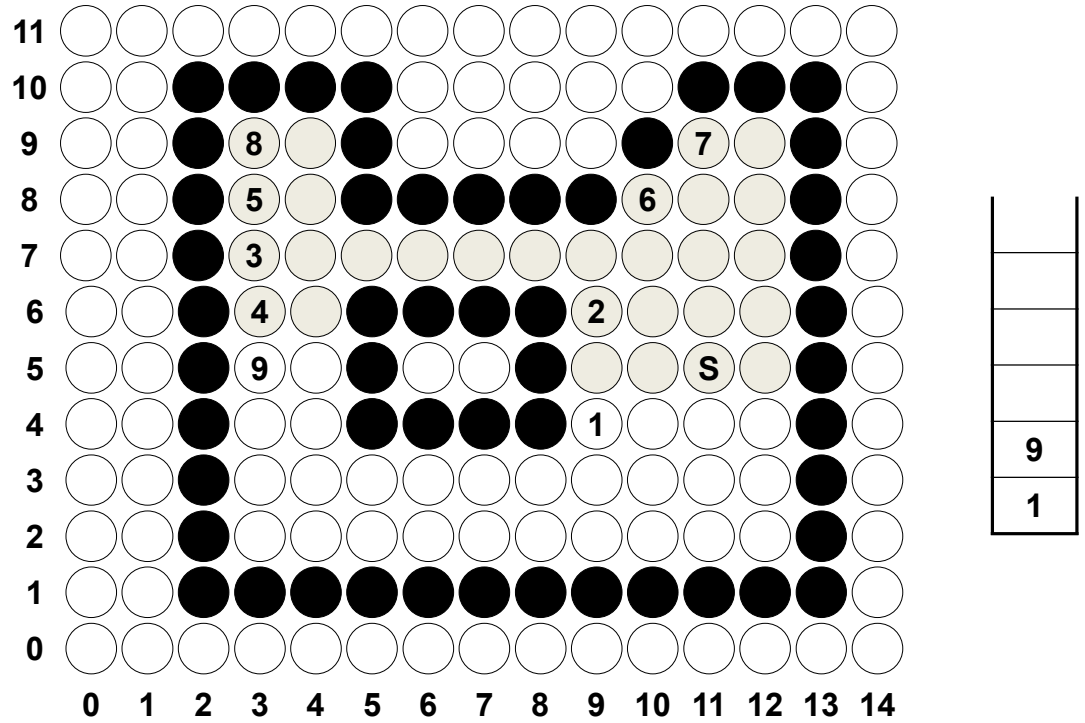
Span Flood-Fill Algorithm (example)



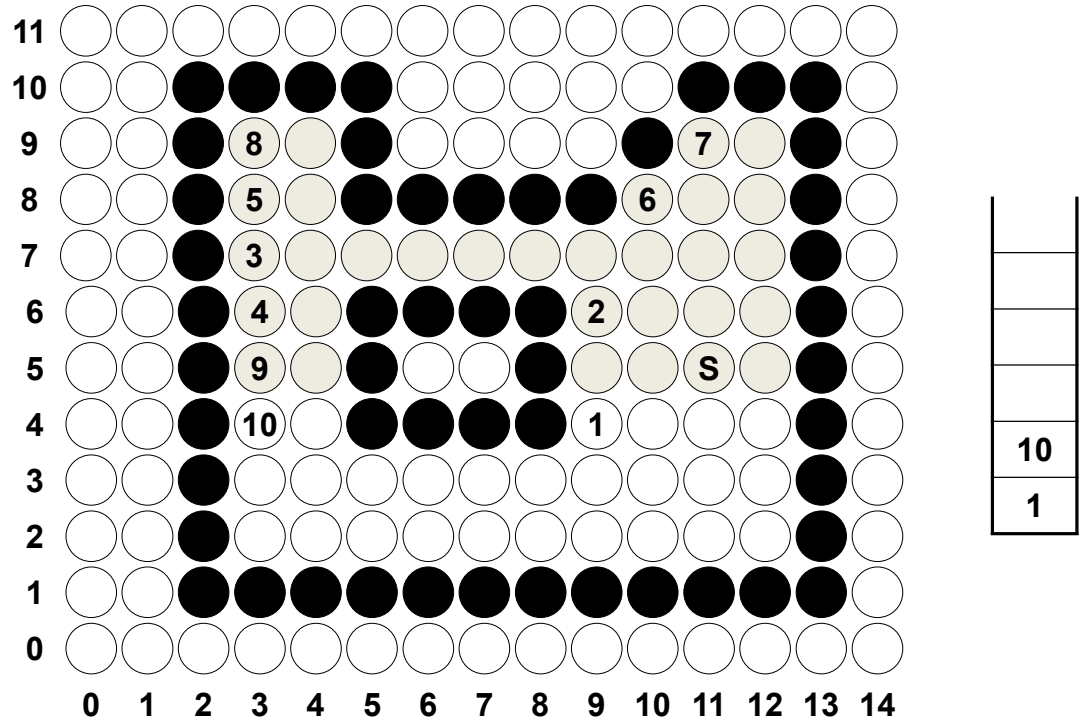
Span Flood-Fill Algorithm (example)



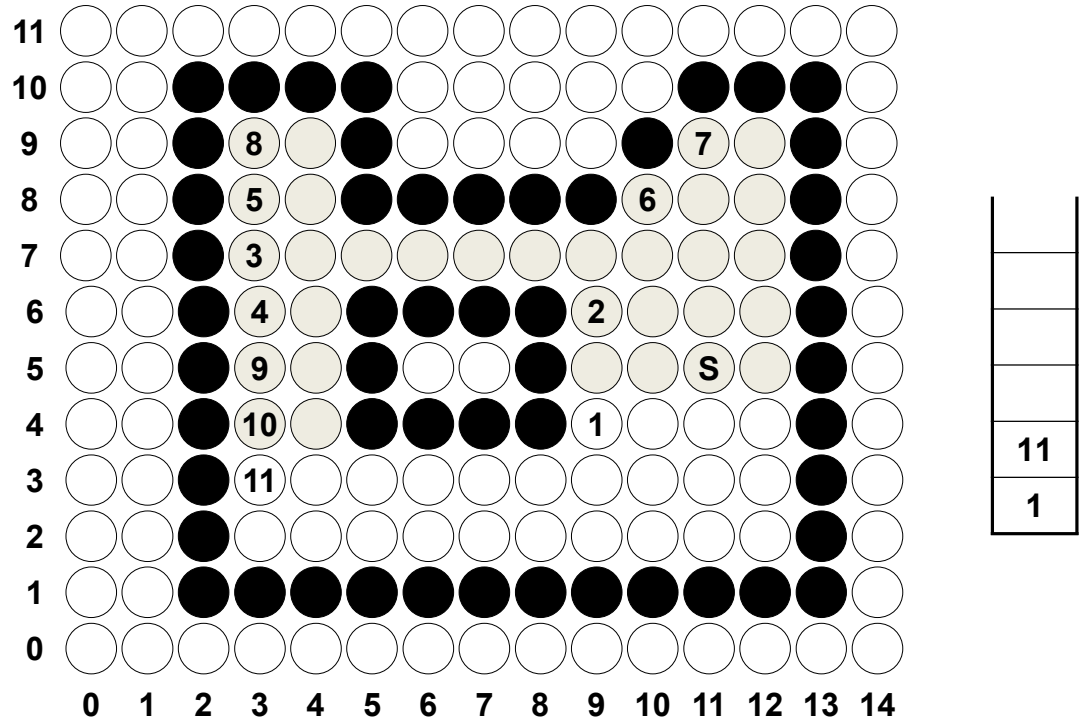
Span Flood-Fill Algorithm (example)



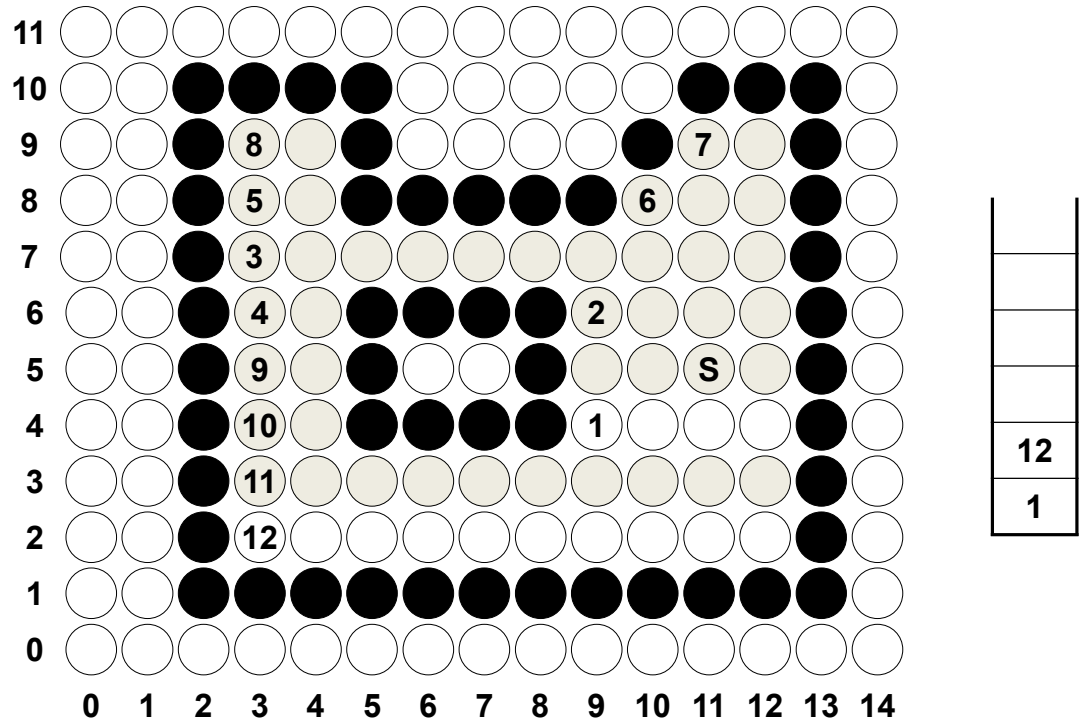
Span Flood-Fill Algorithm (example)



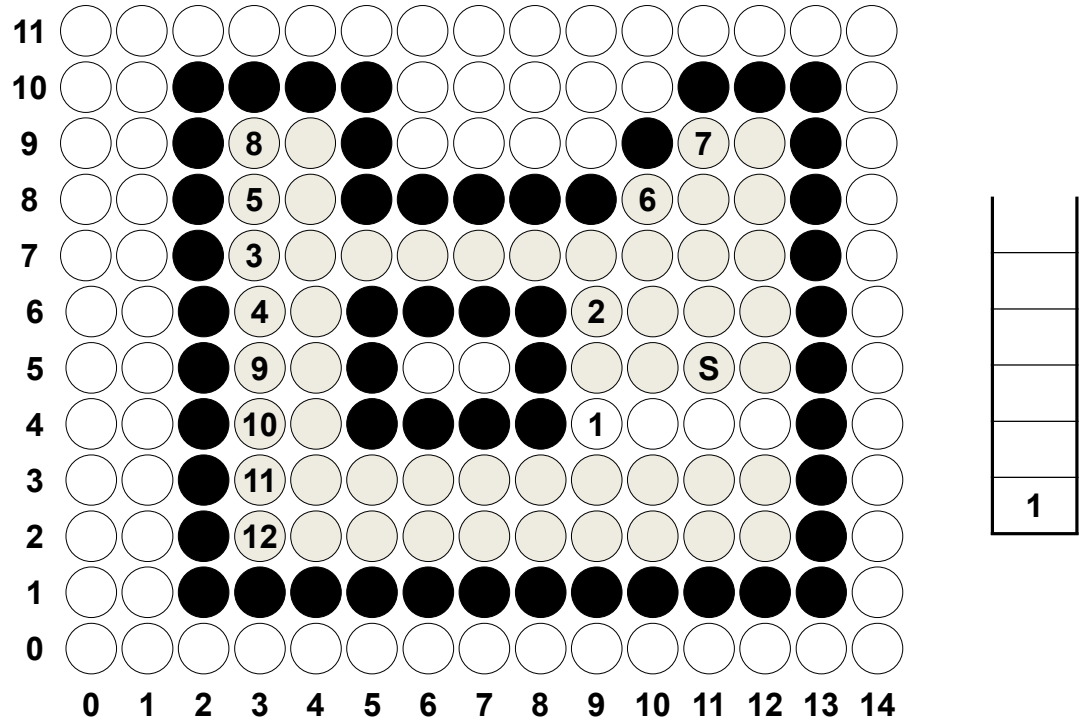
Span Flood-Fill Algorithm (example)



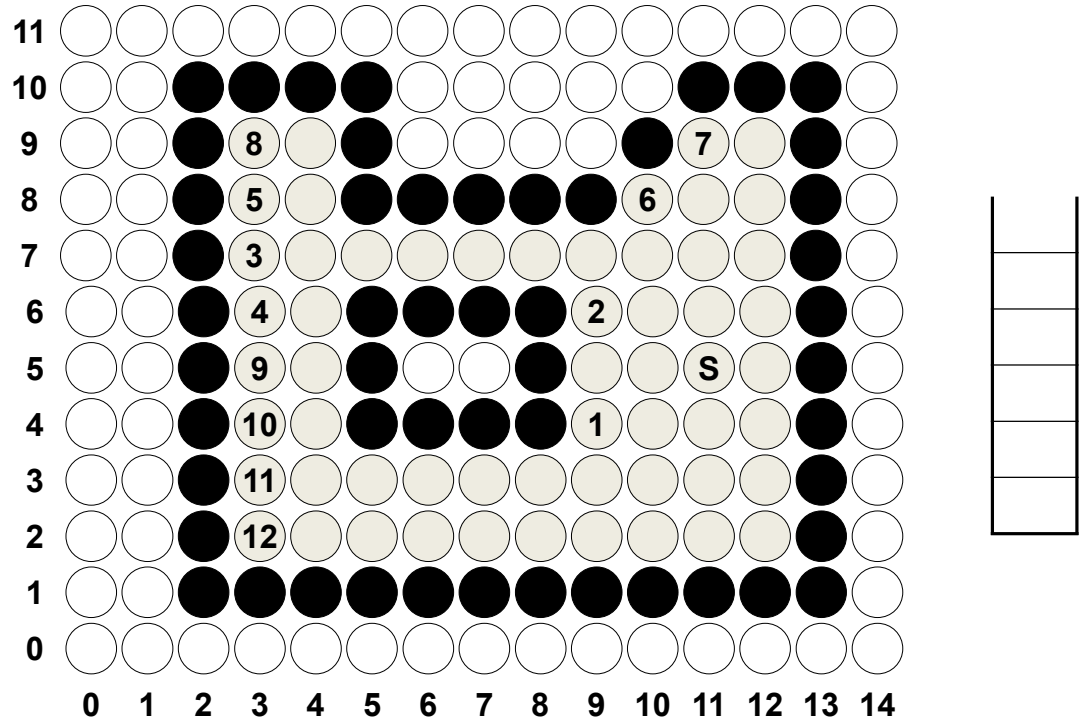
Span Flood-Fill Algorithm (example)



Span Flood-Fill Algorithm (example)

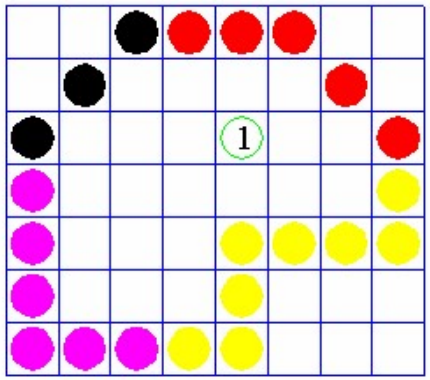


Span Flood-Fill Algorithm (example)



Flood Fill Algorithm

- Sometimes we want to fill in (recolor) an area that is not defined within a single color boundary.
- We paint such areas by replacing a specified interior color instead of searching for a boundary color value.
- This approach is called a **flood-fill algorithm**.



Flood-Fill Algorithm

- We can paint such areas by replacing a specified interior color instead of searching for *a* boundary color value.
- This approach is called a flood-fill algorithm.

Fig An area defined within multiple color boundaries

Flood Fill Algorithm

- We start from a specified interior pixel (x, y) and reassign all pixel values that are currently set to a given interior color with the desired fill color.

- If the area has **more than one** interior color, we can first **reassign pixel values** so that all interior pixels have the same color.

- Using either **4-connected** or **8-connected** approach, we then step through pixel positions until all interior pixels have been repainted.

Flood-Fill Algorithm (cont.)

```
void floodFill4 (int x, int y, int fillColor, int interiorColor)
{  int color;
   /* set current color to fillColor, then perform following operations. */
   getPixel (x, y, color);
   if  (color = interiorColor)
   {
        setPixel (x, y);   // set color of pixel to fillColor
        floodFill4  (x + 1, y, fillColor, interiorColor);
        floodFill4 (x - 1,   y, fillColor, interiorColor);
        floodFill4 (x, y + 1,  fillColor, interiorColor);
        floodFill4 (x, y - 1,   fillColor, interiorColor);
   }
}
```

Flood Fill Algorithm

- Used when an area defined with multiple color boundaries
- Start at a point inside a region
- Replace a specified interior color (old color) with fill color
- Fill the 4-connected or 8-connected region until all interior points being replaced

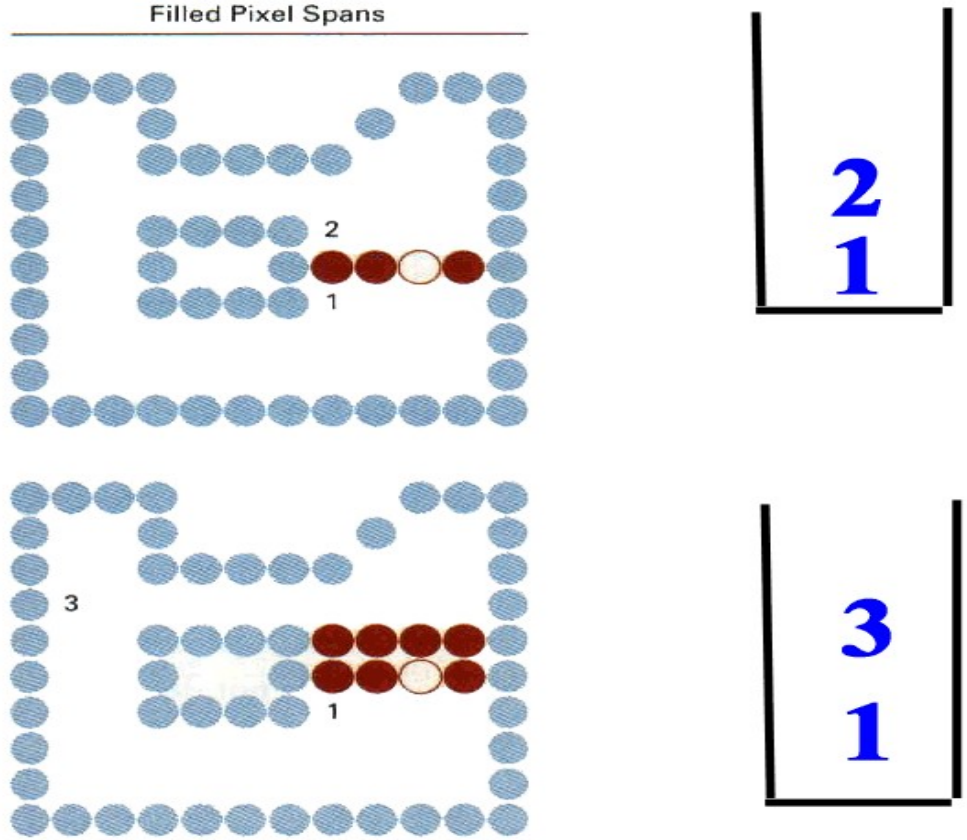
Flood Fill Algorithm (cont.)

```
void FloodFill4(int x, int y, color newcolor, color oldColor)
{
   if(ReadPixel(x, y) == oldColor)
  {
        FloodFill4(x+1, y, newcolor, oldColor);
        FloodFill4(x-1, y, newcolor, oldColor);
        FloodFill4(x, y+1, newcolor, oldColor);
        FloodFill4(x, y-1, newcolor, oldColor);
  }
}
```

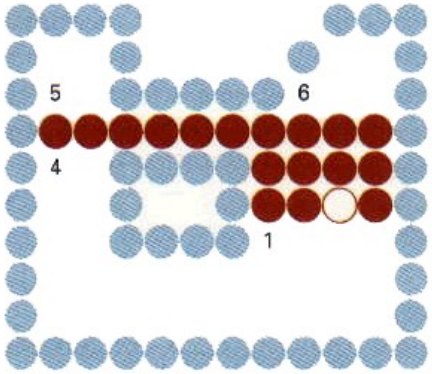
Flood-fill Pixel Span Method

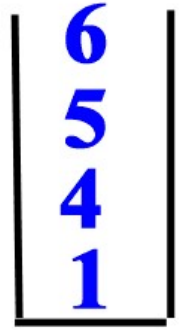
- Start from the initial interior point, then fill in the contiguous span of pixels on this starting scan line.
- Then we locate and stack starting positions for spans on the adjacent scan lines, where spans are defined as the contiguous horizontal string of positions bounded by pixels displayed in the area border color.
- At each subsequent step, unstack the next start position and repeat the process.
- An example of how pixel spans could be filled using this approach is illustrated for the 4-connected fill region in the following figure.

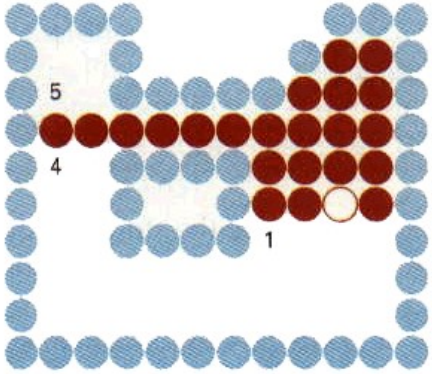
Pixel Span Method (cont.)

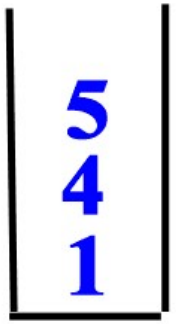


Pixel Span Method (cont.)









Raster Graphics Algorithms

- Coordinate reference frames
- Line drawing algorithms
- Curve drawing algorithms

Basic Elements

Three basic elements

- Scalars
- Vectors
- Points

- Develop mathematical operations among them
- Define basic primitives:
  - Points
  - Line segments

Basic Elements

- Points are associated with locations in space
- Vectors have magnitude and direction
  - represent displacements between points, or directions
- Points, vectors, and operators that combine them are the common tools for solving many geometric problems that arise in
  - Geometric Modeling,
  - Computer Graphics,
  - Animation,
  - Visualization, and
  - Computational Geometry.

3.1 COORDINATE REFERENCE FRAMES

To describe a picture, we first decide upon

- A convenient Cartesian coordinate system, called the world-coordinate reference frame, which could be either 2D or 3D.

- We then describe the objects in our picture by giving their geometric specifications in terms of positions in world coordinates.
  - e.g., we define a straight-line segment with two endpoint positions, and a polygon is specified with a set of positions for its vertices.
  - These coordinate positions are stored in the scene description along with other info about the objects, such as their color and their **coordinate extents**
    - coordinate extents are the minimum and maximum *x, y,* and z values for each object.
    - A set of coordinate extents is also described as a **bounding box** for an object.
      - For a 2D figure, the coordinate extents are sometimes called its **bounding rectangle**.

3.1 COORDINATE REFERENCE FRAMES

- Objects are then displayed by passing the scene description to the viewing routines
  - which identify visible surfaces and map the objects to the frame buffer positions and then on the video monitor.
  - The scan-conversion algorithm stores info about the scene, such as color values, at the appropriate locations in the frame buffer, and then the scene is displayed on the output device.
- locations on a video monitor
  - are referenced in integer screen coordinates, which correspond to the integer pixel positions in the frame buffer.

3.1 COORDINATE REFERENCE FRAMES

- Scan-line algorithms for the graphics primitives use the coordinate descriptions to determine the locations of pixels
  - E.g., given the endpoint coordinates for a line segment, a display algorithm must calculate the positions for those pixels that lie along the line path between the endpoints.
- Since a pixel position occupies a finite area of the screen
  - the finite size of a pixel must be taken into account by the implementation algorithms.
  - for the present, we assume that each integer screen position references the centre of a pixel area.

3.1 COORDINATE REFERENCE FRAMES

- Once pixel positions have been identified the color values must be stored in the frame buffer
- Assume we have available a low-level procedure of the form

setPixel (x, y);

stores the current color setting into the frame buffer at integer position (x, y), relative to the position of the screen-coordinate origin

getPixel (x, y, color);

retrieves the current frame-buffer setting for a pixel location;

- parameter color receives an integer value corresponding to the combined RGB bit codes stored for the specified pixel at position *(x,y).*

- additional screen-coordinate information is needed for 3D scenes. For a two-dimensional scene, all depth values are 0.

3.1 Absolute and Relative Coordinate Specifications

- So far, the coordinate references discussed are given as **absolute coordinate** values
  - values are actual positions wrt origin of current coordinate system
- some graphics packages also allow positions to be specified using **relative coordinates**
  - as an **offset from** the **last position** that was referenced (called the current position)

LINE-DRAWING ALGORITHMS

A straight-line segment in a scene is defined by the coordinate positions for the endpoints of the segment.

1. To display the line on a raster monitor, the graphics system must
   - first project the endpoints to integer screen coordinates and
   - determine the nearest pixel positions along the line path between the two endpoints.
2. Then the line color is loaded into the frame buffer at the corresponding pixel coordinates.
3. Reading from the frame buffer, the video controller plots the screen pixels.
   - This process digitizes the line into a set of discrete integer positions that, in general, only approximates the actual line path.
4. A computed line position of (10.48, 20.51)

   - for example, is converted to pixel position (10, 21).

   - This rounding of coordinate values to integers causes all but horizontal and vertical lines to be displayed with a **stair-step appearance (known as "the jaggies**")

LINE-DRAWING ALGORITHMS

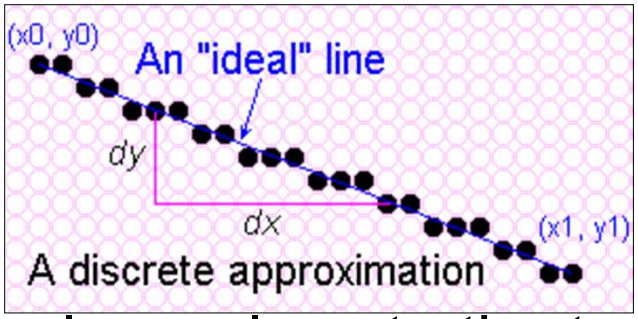
- On raster systems, lines are plotted with pixels, and step sizes in the horizontal and vertical directions are constrained by pixel separations.

- That is, we must "sample" a line at discrete positions and determine the nearest pixel to the line at sampled position.
  - Sampling is measuring the values of the function at equal intervals

- Idea: A line is sampled at unit intervals in one coordinate and the corresponding integer values nearest the line path are determined for the other coordinate.

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
- In the raster line alg.,
  - we sample at unit intervals and
  - determine the closest pixel position to the specified line path at each step

What is an *ideal* line

- Must appear straight and continuous
  - Only possible with axis-aligned and 45^{o} lines
- Must interpolate both defining end points
- Must have uniform density and intensity
  - Consistent within a line and over all lines
  - What about anti-aliasing ?
    - Aliasing is the jagged edges on curves and diagonal lines in a bitmap image.
    - Anti-aliasing is the process of smoothing out those jaggies.
      - Graphics software programs have options for anti-aliasing text and graphics.
      - Enlarging a bitmap image accentuates the effect of aliasing.
- Must be efficient, drawn quickly

Simple Line

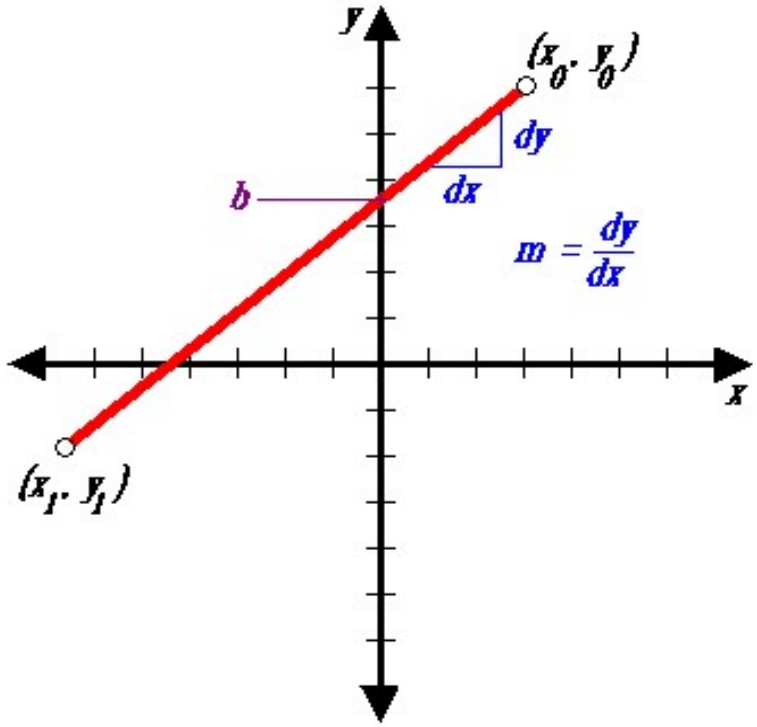
*The **Cartesian slope-intercept equation** for a straight line is :*

$$y = mx + b$$

with *m* as the slope of the line and ***b*** as the ***y*** intercept.

Simple approach:

- increment x, solve for y



Line-Drawing Algorithms:

DDA

Bresenham's Midpoint Algorithm

Algorithms for displaying lines are based on the Cartesian slope-intercept equation

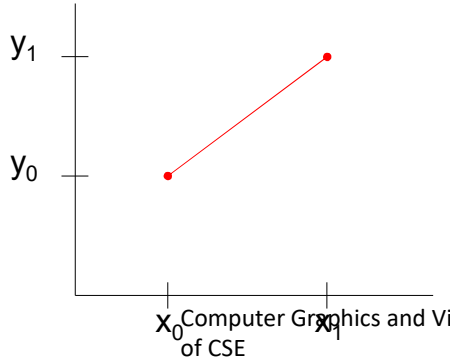
$y = m.x + b$

where m and b can be calculated from the line endpoints:

$m = (y_1-y_0) / (x_1-x_0)$

$b = y_0 - m. x_0$

For any x interval δx along a line the corresponding y interval $\delta y = m.\delta x$



Simple Line

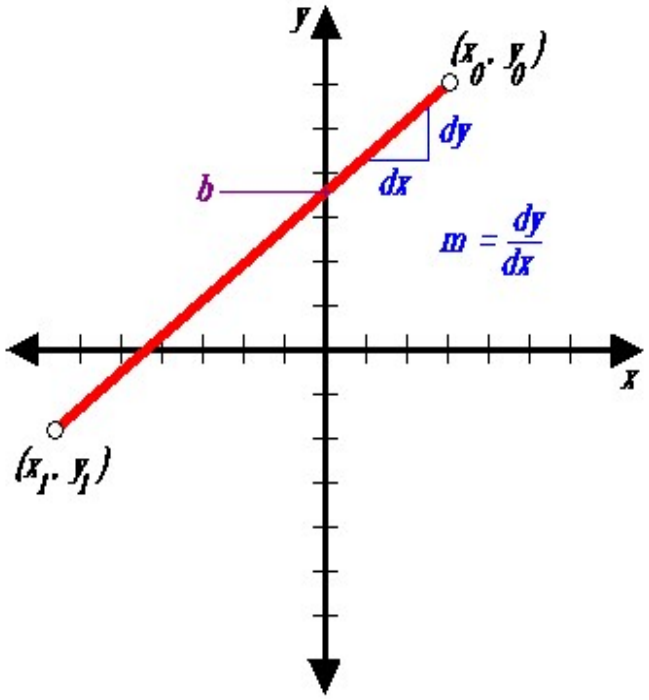
Based on *slope-intercept algorithm* from algebra:

$$y = mx + b$$

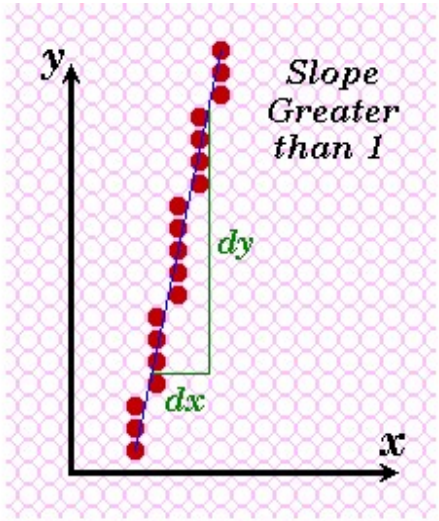
Simple approach:

increment x, solve for y

Floating point arithmetic required



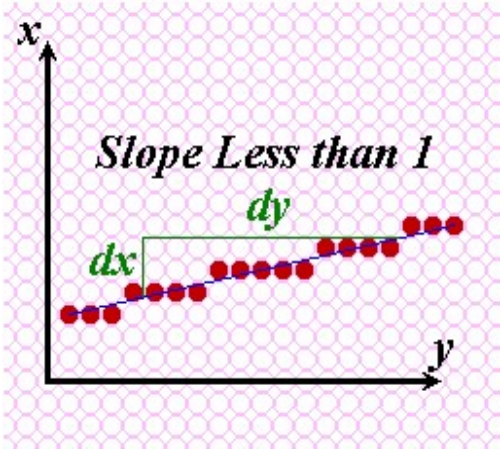
Does it Work?



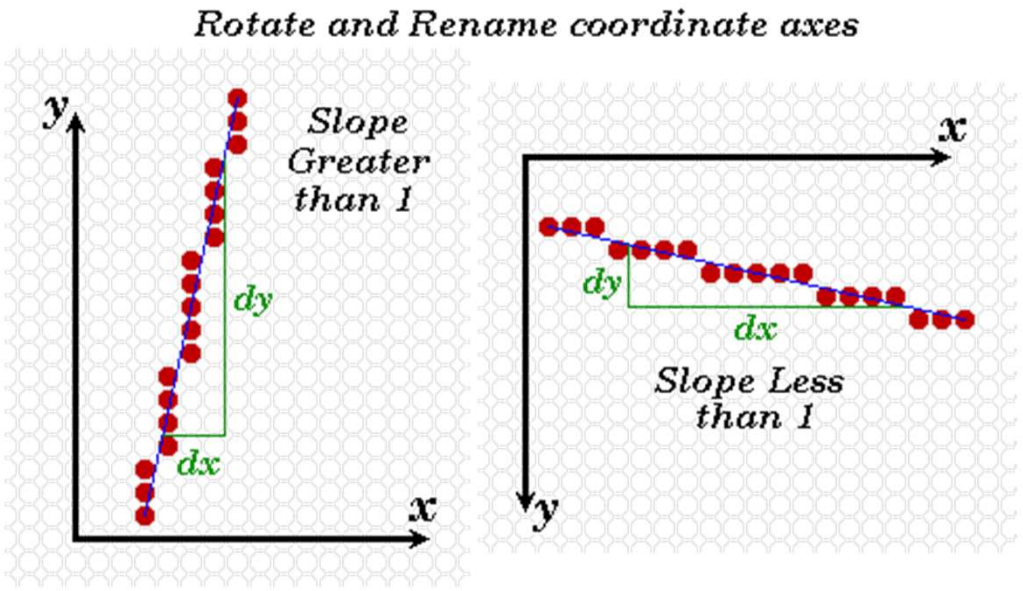
It works for lines with a slope of 1 or less,

but doesn't work well for lines with slope greater than 1 – lines become more discontinuous in appearance and we must add more than 1 pixel per column to make it work.

Solution? - use *symmetry.*



Modify algorithm per octant



OR, increment along x-axis if dy<dx else increment along y-axis

DDA Algorithm

- The digital differential analyser (DDA) is a scan-conversion line algorithm based on using δx or δy.

- A line is sampled at unit intervals in one coordinate and the corresponding integer values nearest the line path are determined for the other coordinate.

Line with positive slope

- If m<=1,
  - Sample at unit x intervals (dx=1)
  - Compute successive y values as
    - $y_{k+1}=y_k+m$ $0<=k<=x_{end}-x_0$
    - Increment k by 1 for each step
    - Round y to nearest integer value.

- If m>1,
  - Sample at unit y intervals (dy=1)
  - Compute successive x values as
    - $x_{k+1}=x_k+1/m$ $0<=k<=y_{end}-y_0$
    - Increment k by 1 for each step
    - Round x to nearest integer value.

```
inline int round (const float a)  { return int (a + 0.5); }

 void lineDDA (int x0, int y0, int xEnd, int yEnd)
 {
     int dx = xEnd - x0,  dy = yEnd - y0,  steps,  k;
     float xIncrement, yIncrement, x = x0, y = y0;

     if (fabs (dx) > fabs (dy))
       steps = fabs (dx);
     else
       steps = fabs (dy);
     xIncrement = float (dx) / float (steps);
     yIncrement = float (dy) / float (steps);

     setPixel (round (x), round (y));
     for (k = 0; k < steps; k++) {
       x += xIncrement;
       y += yIncrement;
       setPixel (round (x), round (y));
     }
 }
```

DDA algorithm

- Need a lot of floating point arithmetic.
  - 2 ‘round’s and 2 adds per pixel.

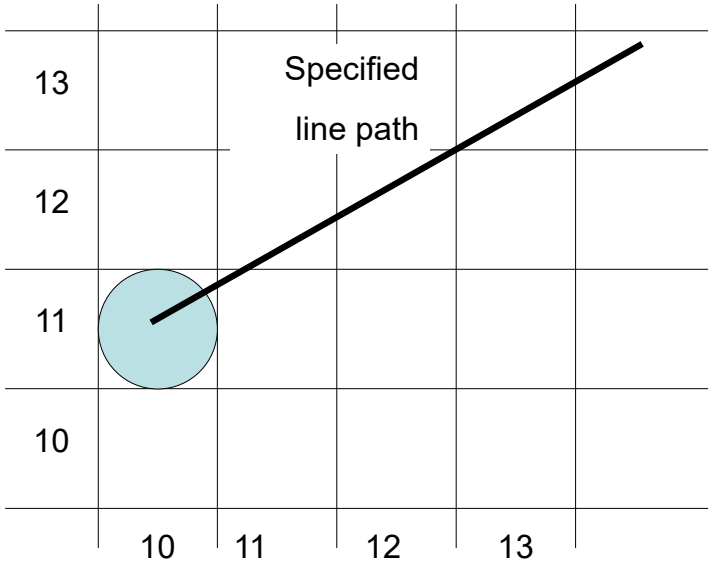
- Is there a simpler way ?
- Can we use only integer arithmetic ?
  - Easier to implement in hardware.

Bresenham's line algorithm

- Accurate and efficient
- Uses only incremental integer calculations

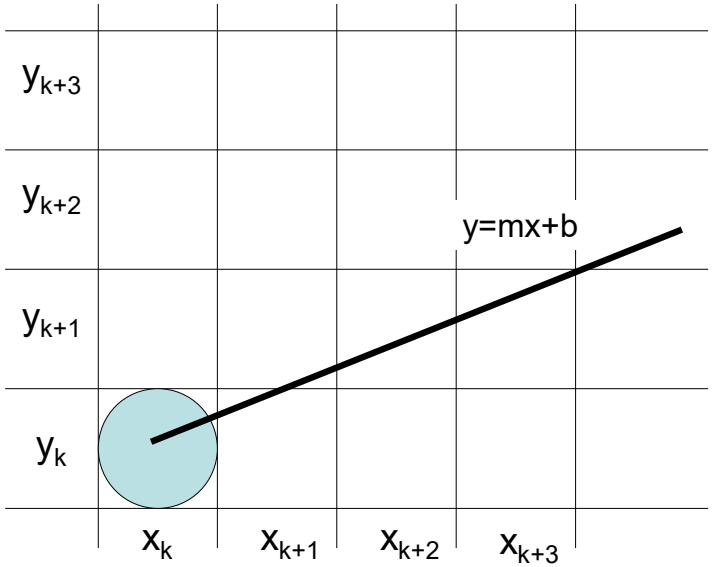
- The method is described for a line segment with a positive slope less than one
- The method generalizes to line segments of other slopes by considering the symmetry between the various octants and quadrants of the xy plane

Bresenham's line algorithm



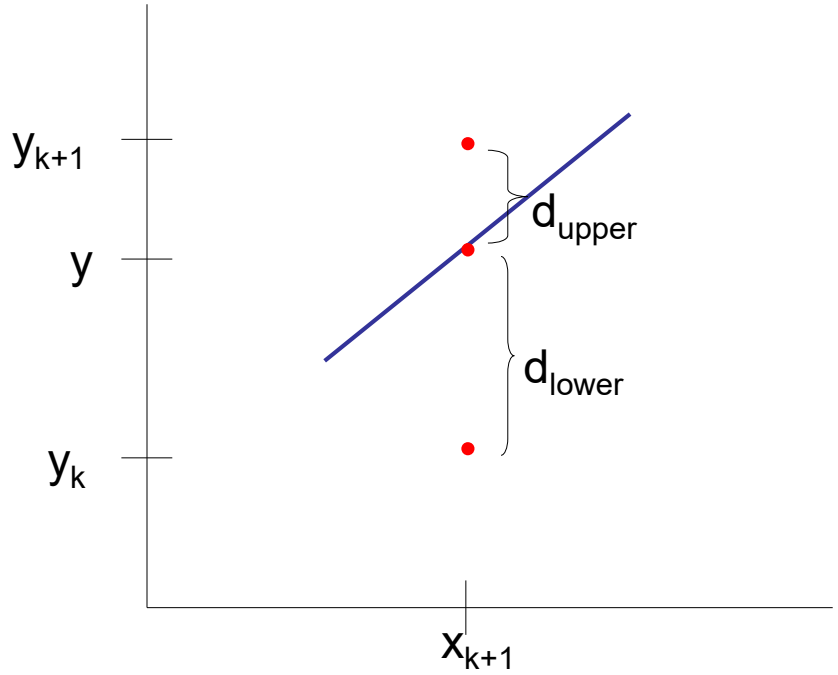
- Decide what is the next pixel position
  - (11,11) or (11,12)

Illustrating Bresenham's Approach



- For the pixel position $x_{k+1}=x_k+1$, which one we should choose:
- (x_{k+1},y_k) or (x_{k+1}, y_{k+1})

Bresenham's Approach



- $y=m(x_k + 1)+b$
- $d_{lower}=y-y_k$
  $=m(x_k + 1)+b-y_k$
- $d_{upper}=(y_k+1)-y$
  $= y_k+1 -m(x_k + 1)-b$

- $d_{lower}- d_{upper}= 2m(x_k + 1)+2y_k+2b-1$
- Rearrange it to have integer calculations:

  $m=\Delta y/\Delta x$

  Decision parameter: $p_k= \Delta x(d_{lower}- d_{upper})=2\Delta y.x_k - 2\Delta x. y_k + c$

The Decision Parameter

Decision parameter: $p_k = \Delta x(d_{lower} - d_{upper}) = 2\Delta y.x_k - 2\Delta x.\ y_k + c$

- p_k has the same sign with $d_{lower} - d_{upper}$ since $\Delta x > 0$.
- c is constant and has the value $c = 2\Delta y + \Delta x(2b-1)$
  - c is independent of the pixel positions and is eliminated from decision parameter p_k.

- If $d_{lower} < d_{upper}$ then p_k is negative.
  - Plot the lower pixel (East)
- Otherwise
  - Plot the upper pixel (North East)

Succesive decision parameter

- At step k+1

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

- Subtracting two subsequent decision parameters yields:

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

- $x_{k+1}=x_k+1$ so

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

  – $y_{k+1}-y_k$ is either 0 or 1 depending on the sign of p_k
- First parameter p_0
  – $p_0=2 \Delta y - \Delta x$

Bresenham's Line-Drawing Algorithm for I m I < 1

1. Input the twoline endpoints and store the left endpoint in (x_0, y_0).

2. Load (x_0, y_0) into the frame buffer; that is, plot the first point.

3. Calculate constants Δx, Δy, $2\Delta y$, and $2\Delta y - 2\Delta x$, and obtain the starting 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

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

   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 Δx -1 times.

Trivial Situations: Do not need Bresenham

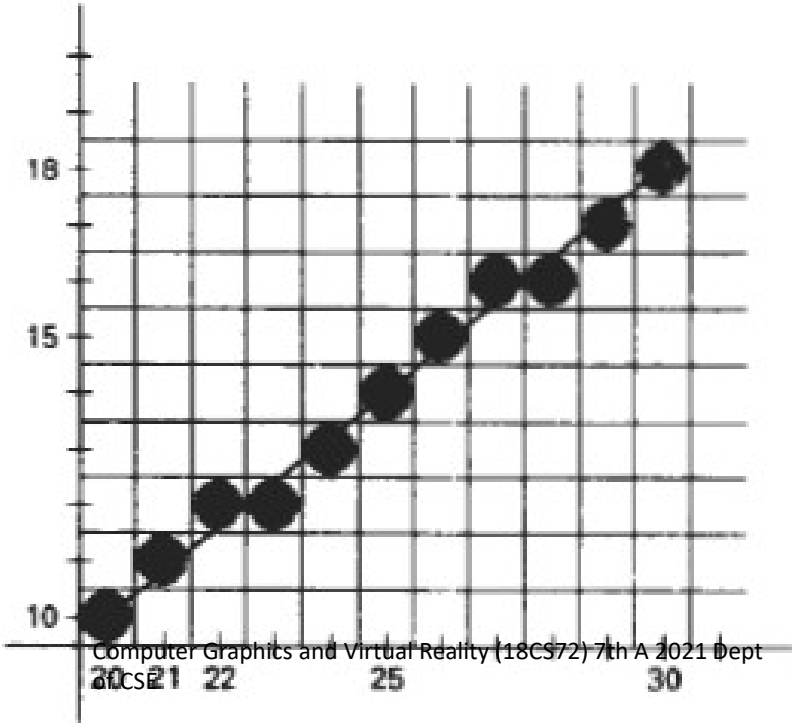
- $m = 0 \quad \Rightarrow \quad \text{horizontalline}$
- $m = \pm 1 \quad \Rightarrow \quad \text{line } y = \pm x$
- $m = \infty \quad \Rightarrow \quad \text{verticalline}$

Example

- Draw the line with endpoints (20,10) and (30, 18).
  - **Δx=30-20=10, Δy=18-10=8,**
  - **$p_0 = 2\Delta y - \Delta x$=16-10=6**
  - **$2\Delta y$=16,** and **$2\Delta y - 2\Delta x$=-4**
- Plot the initial position at (20,10), then

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)

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)



Example

- Line end points: $(x_0, y_0) = (5,8); \quad (x_1, y_1) = (9,11)$
- Deltas:

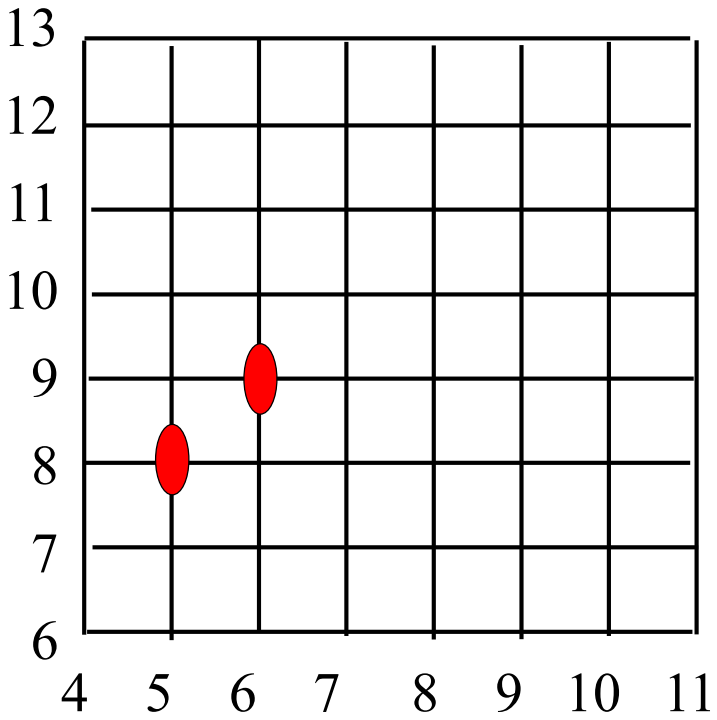
$$dx = 4; dy = 3$$

initially $p(5,8) = 2(dy)-(dx)$

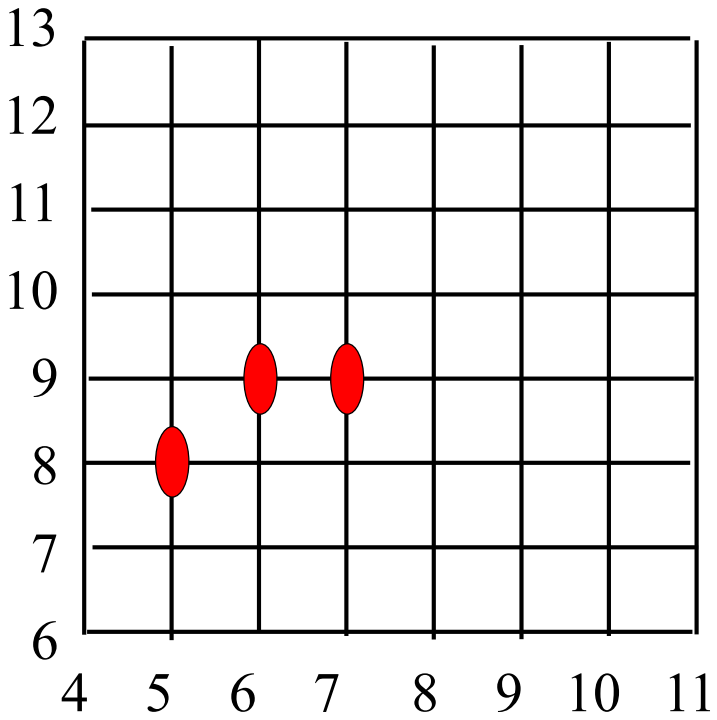
$$= 6 - 4 = 2 > 0$$

$$p = 2 \quad \Rightarrow \quad NE$$

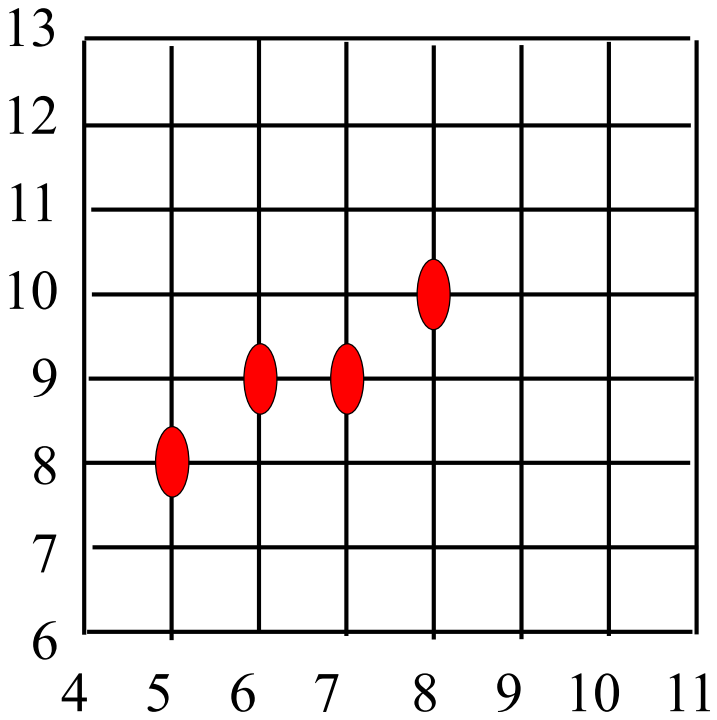
Graph



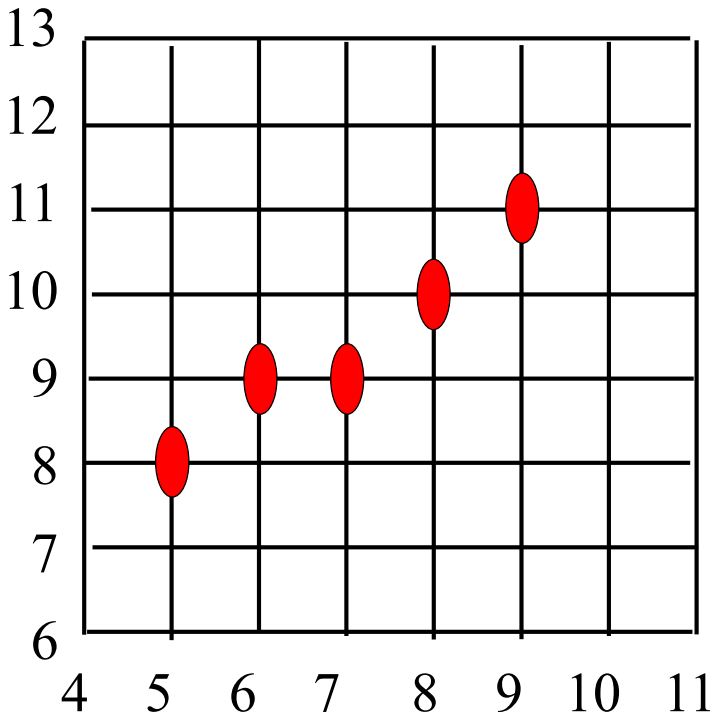
Continue the process...



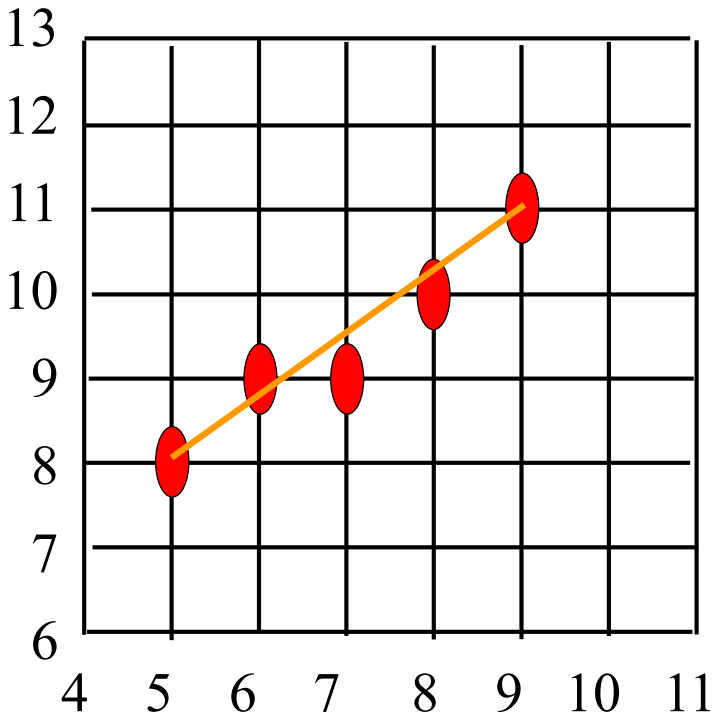
Graph



Graph



Graph

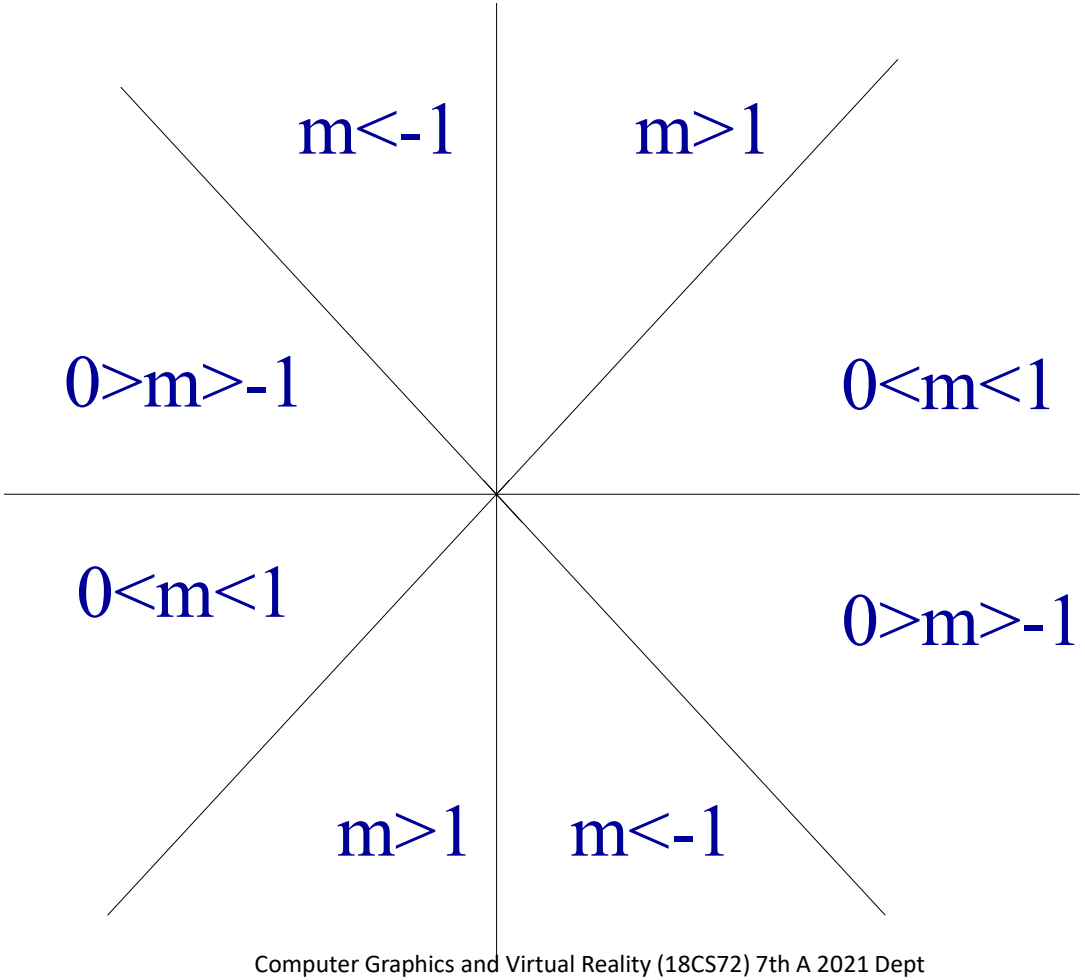


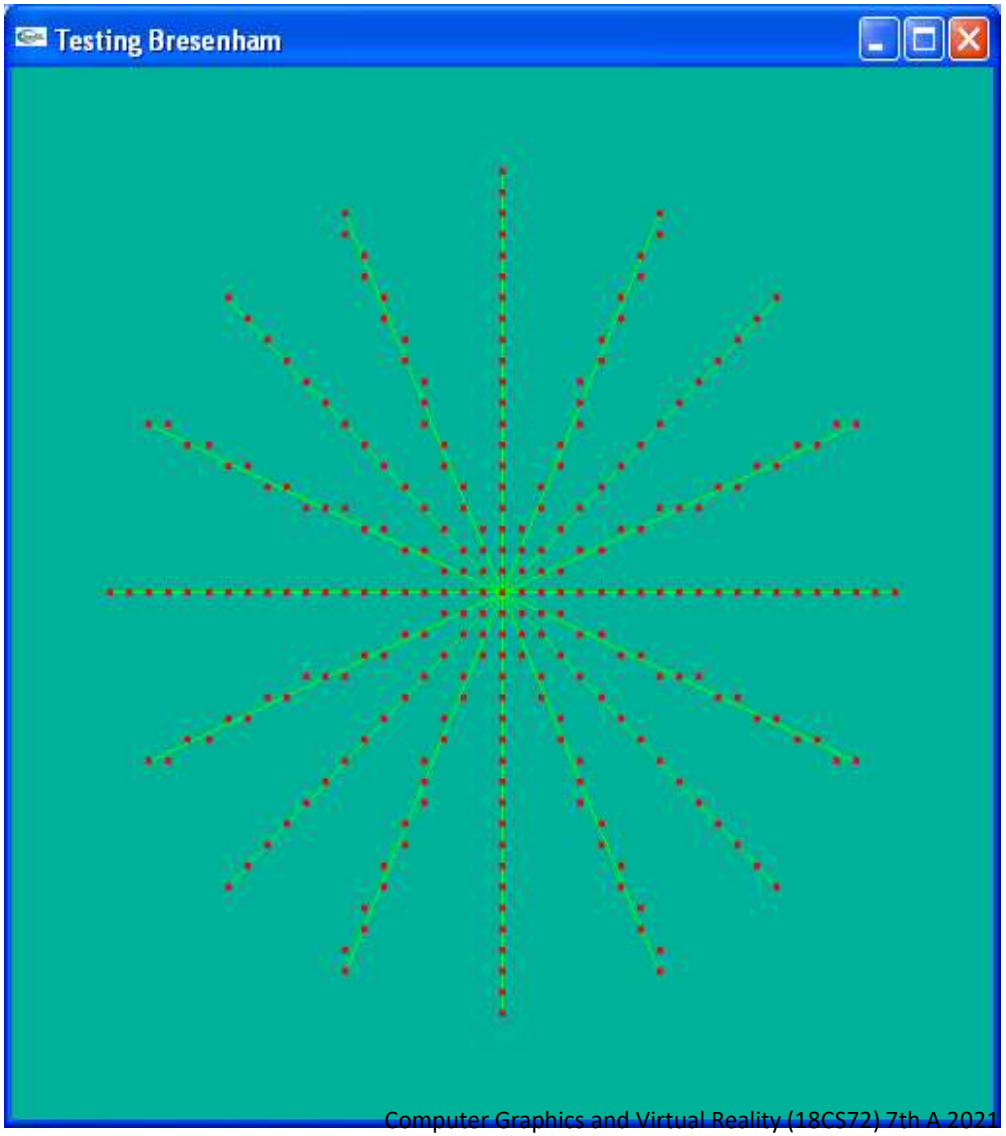
```
/*  Bresenham line-drawing procedure for |m| < 1.0.  */
  void lineBres (int x0, int y0, int xEnd, int yEnd)
  {
    int dx = fabs (xEnd - x0),  dy = fabs(yEnd - y0);
    int x, y, p = 2 * dy - dx;
    int twoDy = 2 * dy,  twoDyMinusDx = 2 * (dy - dx);

    /* Determine which endpoint to use as start position.  */
    if (x0 > xEnd) {
      x = xEnd;    y = yEnd;   xEnd = x0;
    }
    else {
      x = x0;    y = y0;
    }
    setPixel (x, y);

    while (x < xEnd) {
      x++;
      if (p < 0)
        p += twoDy;
      else {
        y++;
        p += twoDyMinusDx;
      }
      setPixel (x, y);
    }
  }
```

Line-drawing algorithm should work in every octant, and special cases





Simulating the Bresenham algorithm in drawing 8 radii on a circle of radius 20

Horizontal, vertical and ±45° radii handled as special cases

Circle and Ellipse

Another method:

Approximate it using a polyline.

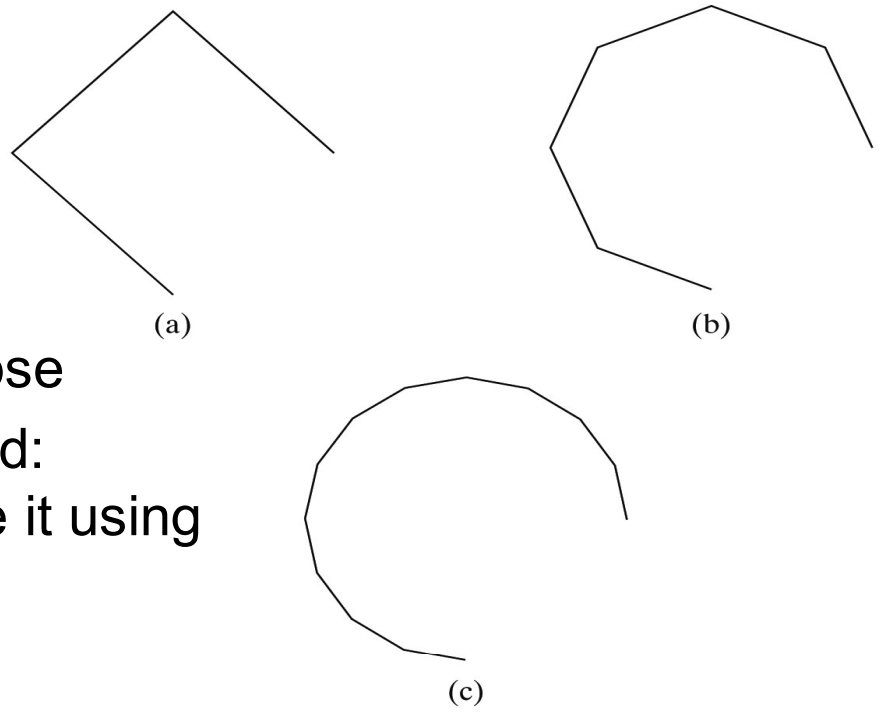


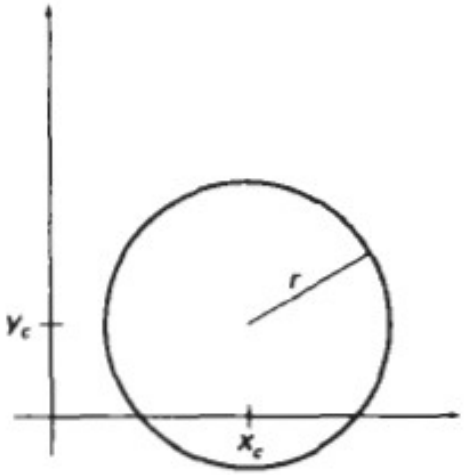
Figure 3-15

A circular arc approximated with (a) three straight-line segments, (b) six line segments, and (c) twelve line segments.

Scan Converting Circles

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

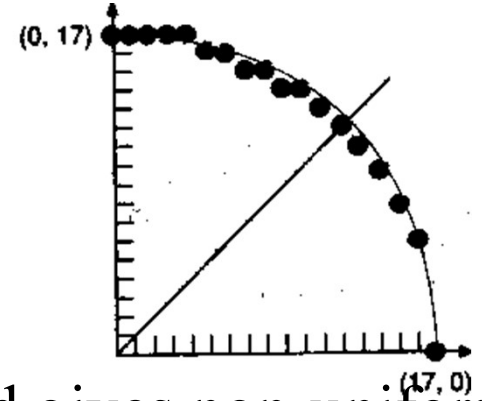
$$y = y_c \pm \sqrt{R^2 - (x - x_c)^2}$$



- Explicit: y = f(x)

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

We could draw a quarter circle by incrementing x from 0 to R in unit steps and solving for +y for each step.



Method needs lots of computation, and gives non-uniform pixel spacing

Scan Converting Circles

- Parametric:

$$x = R\cos\theta$$

$$y = R\sin\theta$$

Draw quarter circle by stepping through the angle from 0 to 90
-avoids large gaps but still unsatisfactory
-How to set angular increment
Computationally expensive trigonometric calculations

Scan Converting Circles

- Implicit: $f(x,y) = x^2+y^2-R^2$

If $f(x,y) = 0$ then it is on the circle.
$f(x,y) > 0$ then it is outside the circle.
$f(x,y) < 0$ then it is inside the circle.

Try to adapt the Bresenham midpoint approach

Again, exploit symmetries

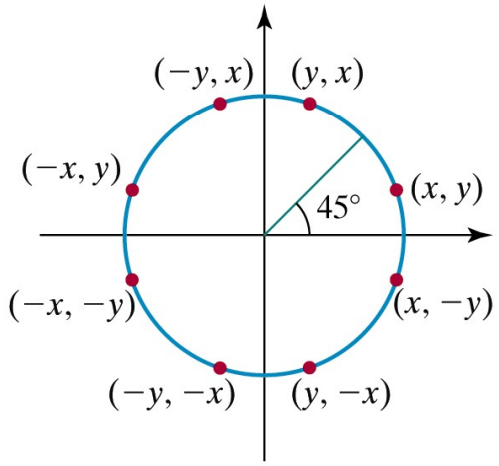


Figure 3-18

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

Generalising the Bresenham midpoint approach

- Set up decision parameters for finding the closest pixel to the circumference at each sampling step
  - Avoid square root calculations by considering the squares of the pixel separation distances
- Use direct comparison without squaring.

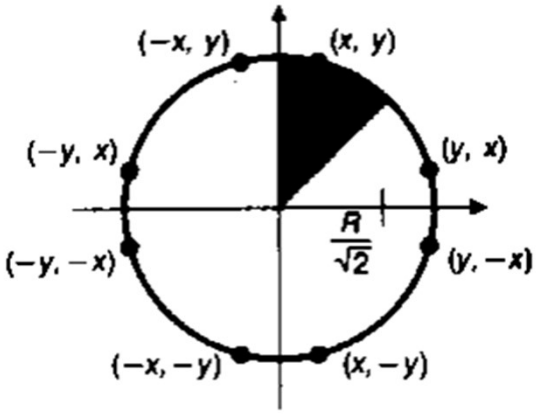
  Adapt the midpoint test idea: test the halfway position between pixels to determine if this midpoint is inside or outside the curve

  This gives the **midpoint algorithm for circles**

  Can be adapted to other curves: conic sections

Eight-way Symmetry

- only one octant's calculation needed




```
void CirclePoints (int x, int y, int value)
{
    WritePixel (x, y, value);
    WritePixel (y, x, value);
    WritePixel (y, −x, value);
    WritePixel (x, −y, value);
    WritePixel (−x, −y, value);
    WritePixel (−y, −x, value);
    WritePixel (−y, x, value);
    WritePixel (−x, y, value);
} /* CirclePoints */
```

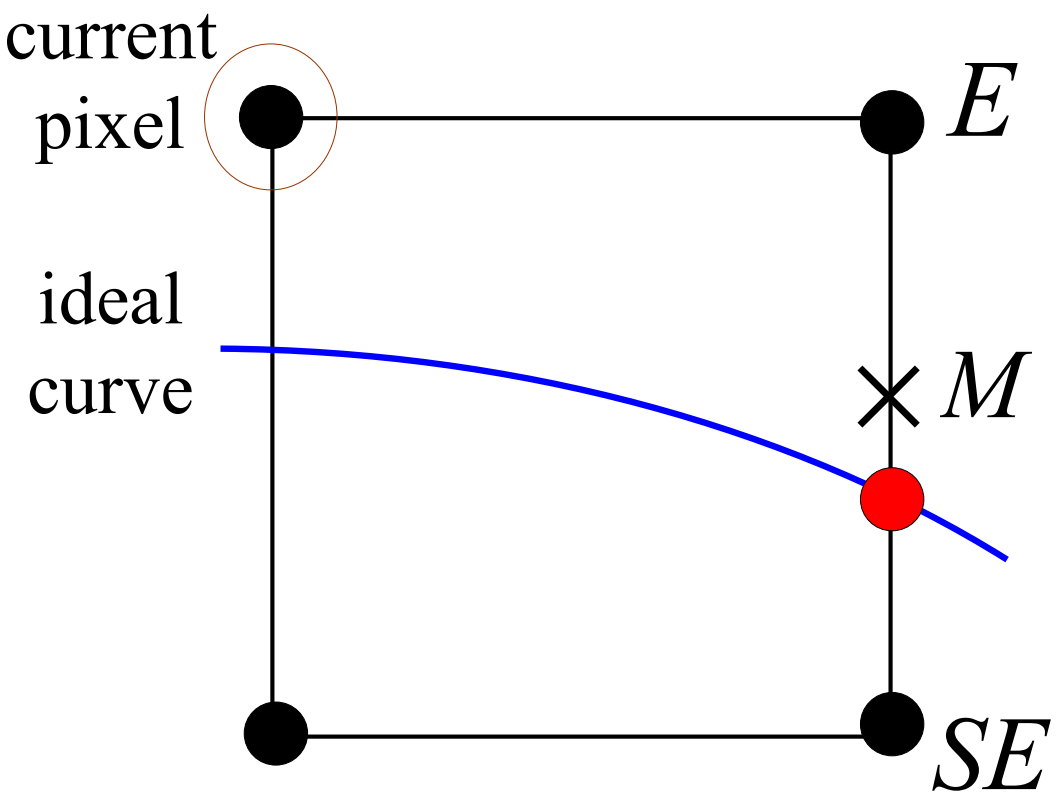
The 2nd Octant is a good arc to draw

- It is a well-defined function in this domain
  - single-valued
  - no vertical tangents: $|slope| \quad 1$
- Lends itself to the midpoint approach
  - only need consider E or SE
- Implicit formulation $F(x,y) = x^2 + y^2 - r^{\ 2}$
  - For (x,y) on the circle, $F(x,y) = 0$
  - $F(x,y) > 0 \Rightarrow (x,y)\ Outside$
  - $F(x,y) < 0 \Rightarrow (x,y)\ Inside$

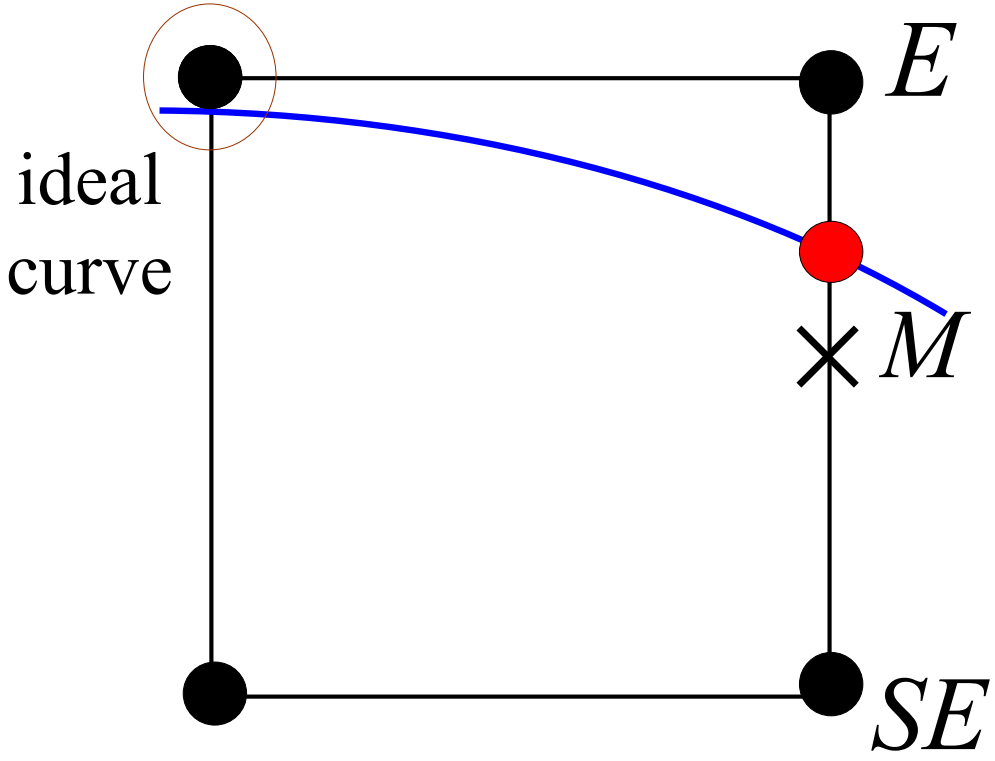
Choose *E* or *SE*

- Decision variable d is $x^2 + y^2 - r^2$
- Then d = $F(M) \quad 0 \Rightarrow$ SE
- Or d = $F(M) < 0 \Rightarrow E$

$F(M) \begin{cases} < 0 \\ = 0 \\ > 0 \end{cases} \Rightarrow \begin{matrix} E \\ M \\ SE \end{matrix}$



$F(M) < 0 \Rightarrow E$



Decision Variable *p*

As in the Bresenham line algorithm we use a decision variable to direct the selection of pixels.

Use the implicit form of the circle equation

$$p = F(M) = x^2 + y^2 - r^2$$

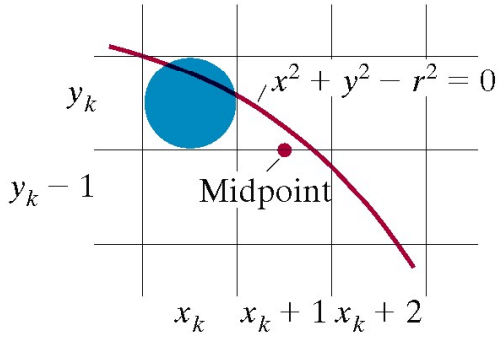


Figure 3-19

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

Midpoint coordinates are $(x_k + 1, y_k - \frac{1}{2})$

Assuming we have just plotted point at (x_k,y_k) we determine whether move E or SE by evaluating the circle function at the midpoint between the two candidate pixel positions

$$p_k = F_{circ}(x_k + 1, y_k - 1/2)$$

$$= (x_k + 1)^2 + (y_k - 1/2)^2 - r^2$$

p_k is the decision variable

if $p_k < 0$ the midpoint is inside the circle

Thus the pixel above the midpoint is closer to the ideal circle, and we select pixel on scan line y_k i.e. Go E

If $pk > 0$ the midpoint is outside the circle. Thus the pixel below the midpoint is closer to the ideal circle, and we select pixel on scan line yk-1. i.e. Go SE

Calculate successive decision parameter values p by incremental calculations.

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

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

recursive definition for successive decision parameter values *p*

$$p_{k+1} = F_{circ}(x_{k+1}+1, y_{k+1}-1/2)$$

$$= [(x_k+1)+1]^2 + (y_{k+1}-1/2)^2 - r^2$$

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

Where $y_k+1 = y_k$ if $p<0$ (move E)

$y_k+1 = y_k-1$ if $p>0$ (move SE)

y_{k+1} and x_{k+1} can also be defined recursively

Initialisation

$x_0 = 0, \quad y_0 = r$

Initial decision variable found by evaluating circle function at first midpoint test position

$$p_0 = F_{circ}(1, r - 1/2)$$

$$= 1 + (r - 1/2)^2 - r^2$$

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

For integer radius r p_0 can be rounded to $p_0 = 1-r$ since all increments are integer.

Midpoint Circle Algorithm

1. Input radius r and circle center (x_c, y_c), and obtain the first point on the circumference of a circle centered on the origin as

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

2. Calculate the initial value of the decision parameter as

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

3. At each x_k position, starting at $k = 0$, perform the following test: If $p_k < 0$, the next point along the circle centered on (0, 0) is (x_{k+1}, y_k) and

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

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

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

where $2x_{k+1} = 2x_k + 2$ and $2y_{k+1} = 2y_k - 2$.

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

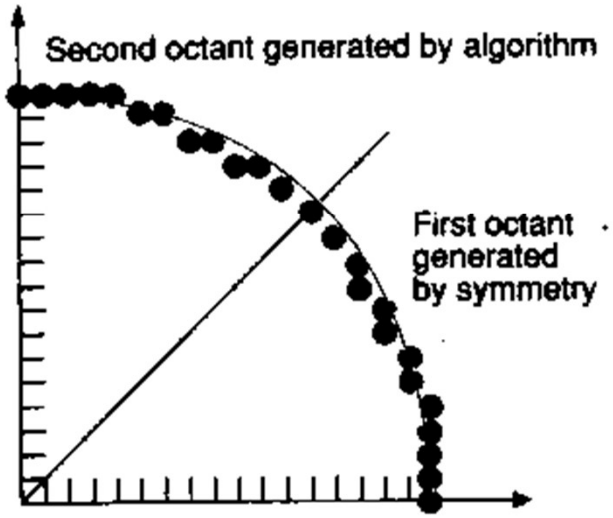
$$x = x + x_c \qquad y = y + y_c$$

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

Midpoint Circle Algorithm (cont.)

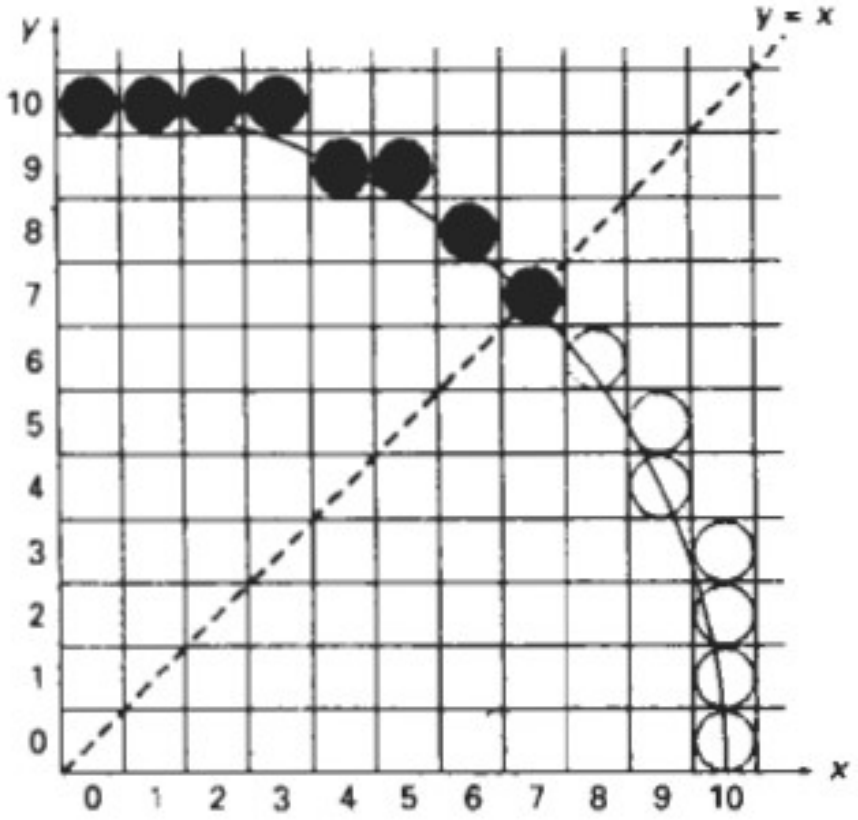
```
void MidpointCircle (int radius, int value)
/* Assumes center of circle is at origin. Integer arithmetic only */
{
    int x = 0;
    int y = radius;
    int d = 1 - radius;
    CirclePoints (x, y, value);

    while (y > x) {
        if (d < 0)          /* Select E */
            d += 2 * x + 3;
        else {              /* Select SE */
            d += 2 * (x - y) + 5;
            y--;
        }
        x++;
        CirclePoints (x, y, value);
    } /* while */
} /* MidpointCircle */
```



Example

- r=10



2D Viewing

- 2D Viewing Pipeline
- Clipping & Clipping Window
- Normalization & Viewport Transformations
- OpenGL 2D Viewing Functions
- Clipping Algorithms
  - 2D Point Clipping
  - 2D Line Clipping
  - 2D Polygon Fill-Area Clipping

2D Viewing

- Any Cartesian coordinate system can be used to define a picture.
- A view is selected by specifying a sub area of the total picture area.
- The picture parts within the selected areas are mapped onto specific areas of the device coordinates.

2D Viewing (cont.)

- Which part of a picture is to be displayed
  - clipping window or world window or viewing window
- Where that part will be displayed on the display device
  - viewport

2D Viewing (cont.)

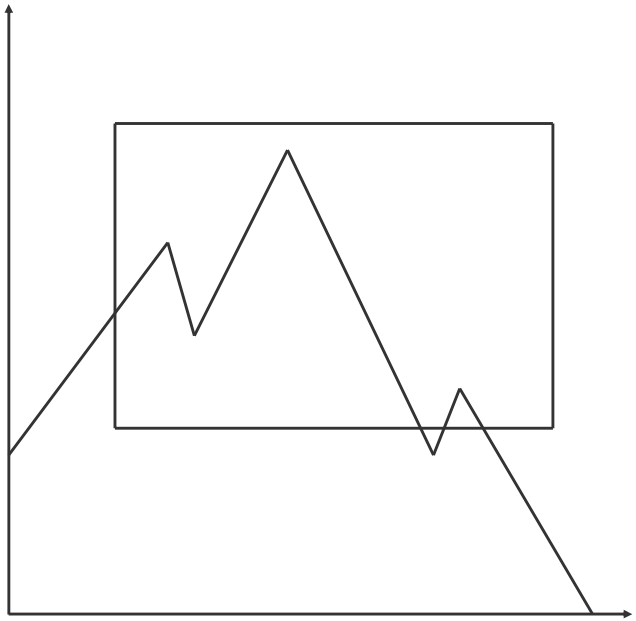
- A world coordinate area selected for display is called a window
  - The window defines what is to be viewed

- An area on a display device to which a window is mapped is called a viewport
  - The viewport defines where it is to be displayed

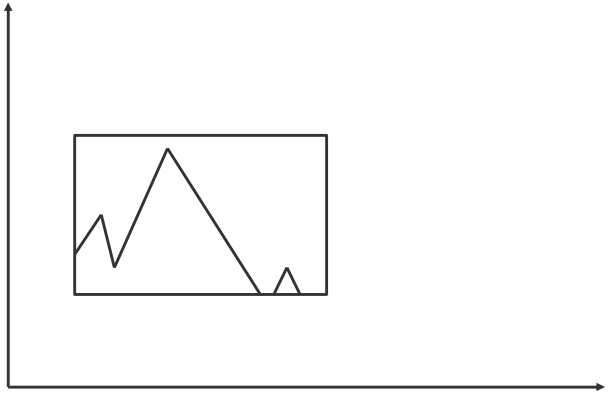
2D Viewing (cont.)

- Clipping window selects **what** we want to see
- Viewport indicates **where** it is to be viewed on the output device
- Usually, clipping windows and viewport are rectangles

2D Viewing (cont.)

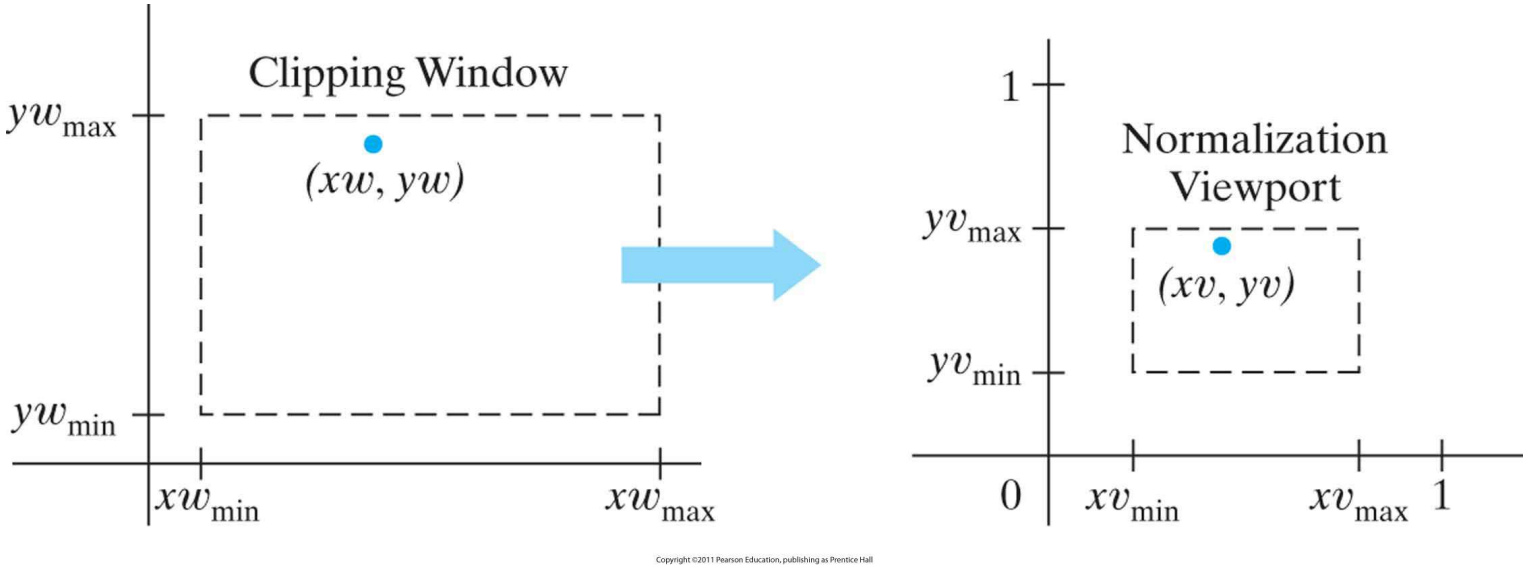


World Coordinates



Viewing Coordinates

2D Viewing (cont.)

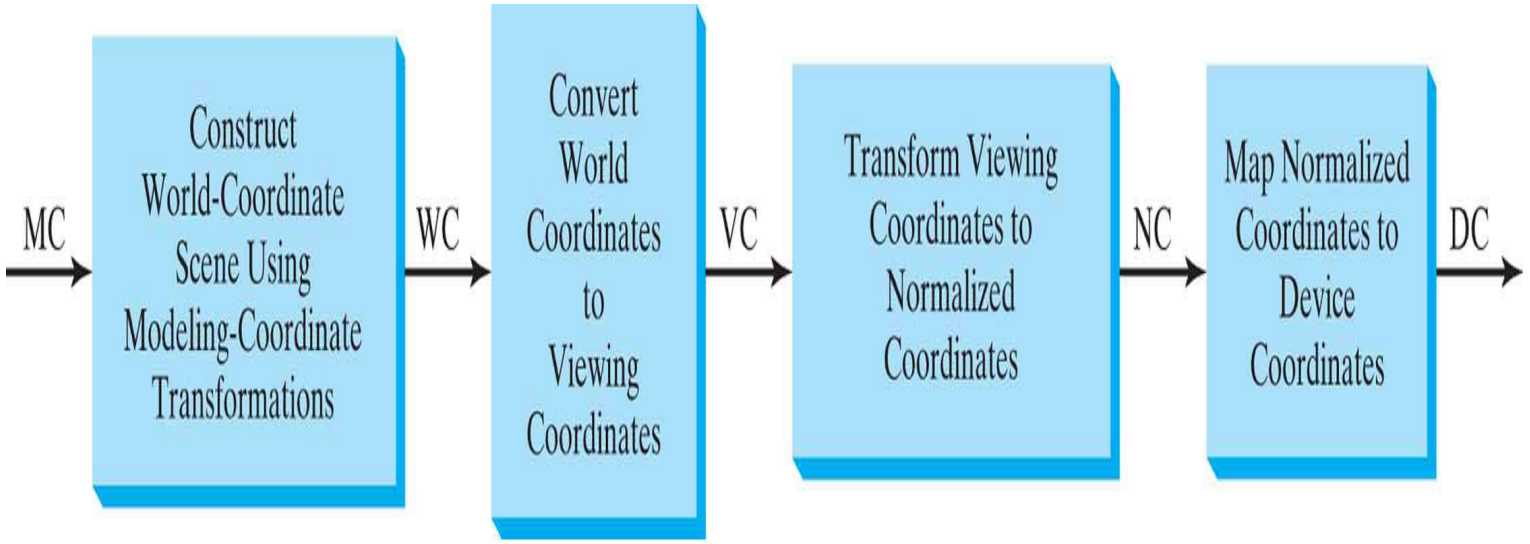


A point (*xw, yw*) in a world-coordinate clipping window is mapped to viewport coordinates (*xv, yv*), within a unit square, so that the relative positions of the two points in their respective rectangles are the same.

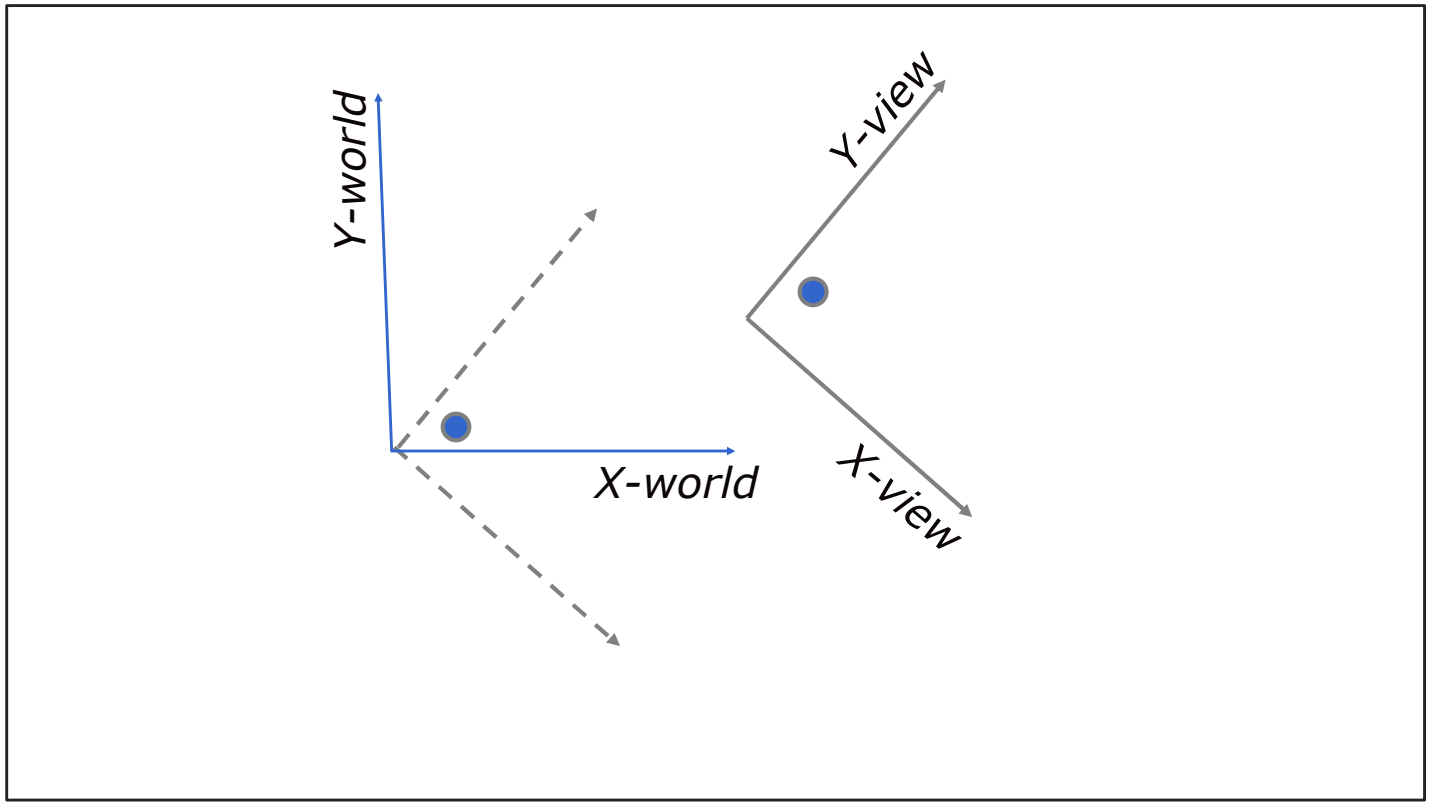
2D Viewing Pipeline

- 2D viewing pipeline
  - Construct world-coordinate scene using modeling-coordinate transformations
  - Convert world-coordinates to viewing coordinates
  - Transform viewing-coordinates to normalized-coordinates (ex: between 0 and 1, or between -1 and 1)
  - Map normalized-coordinates to device-coordinates.

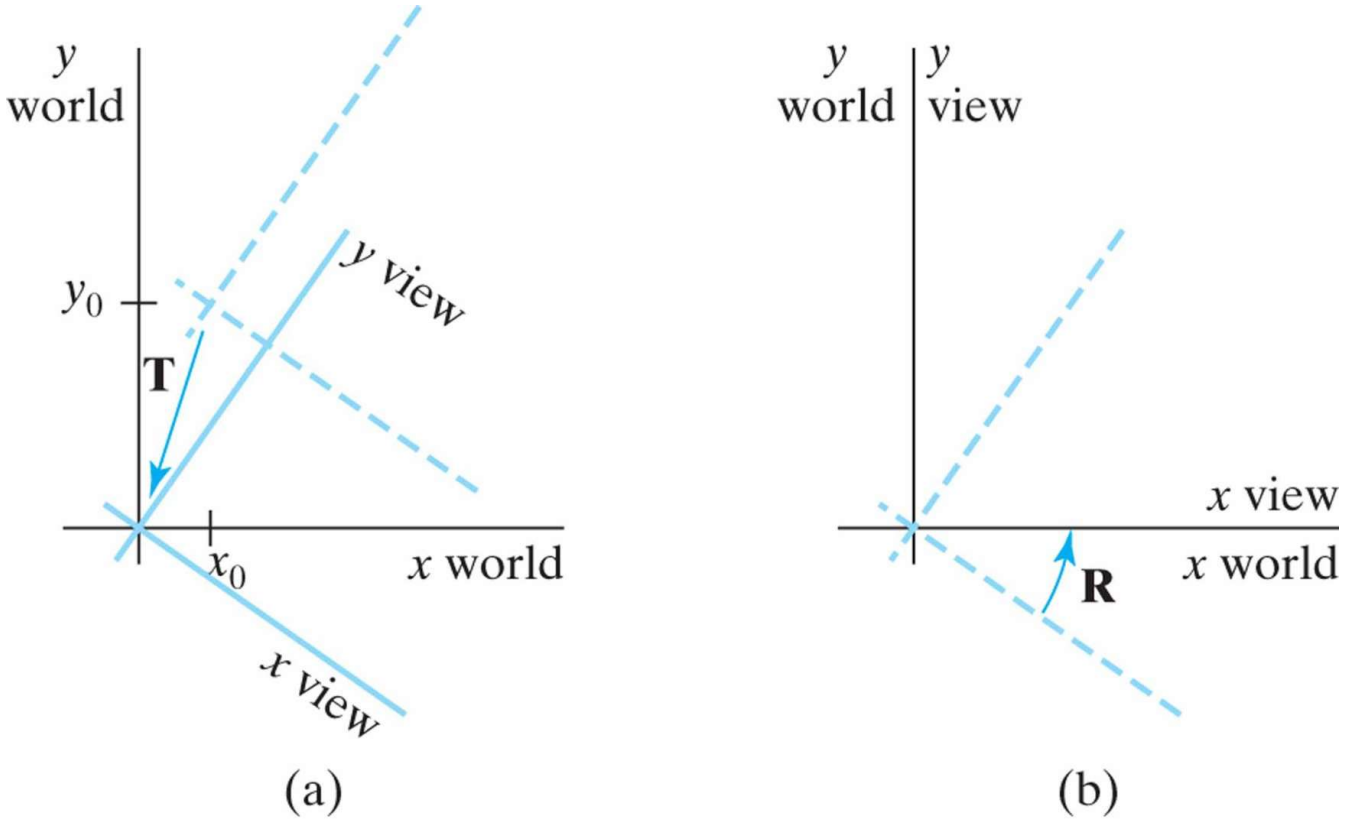
2D Viewing Pipeline (cont.)



Viewing coordinate reference system



Viewing coordinate reference system (cont.)



A viewing-coordinate frame is moved into coincidence with the world frame by (a) applying a translation matrix **T** to move the viewing origin to the world origin, then (b) applying a rotation matrix **R** to align the axes of the two systems.

Viewing coordinate reference system (cont.)

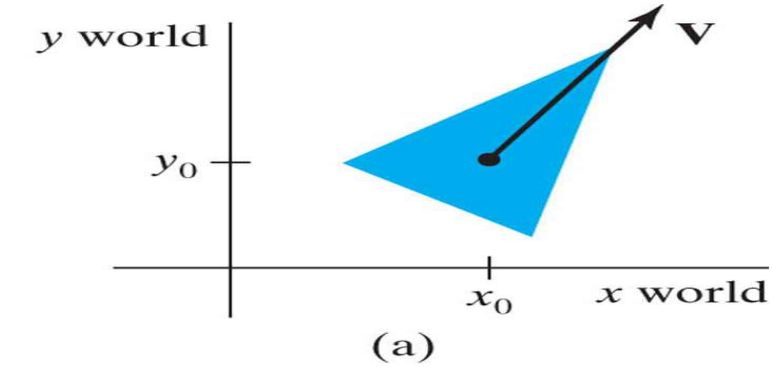
- That means:

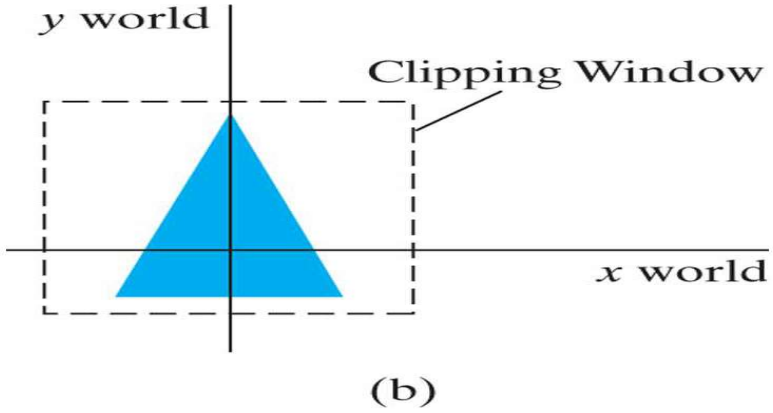
$$M_{WC,VC} = R \cdot T$$

- What about the matrix?

- Think about it ☺

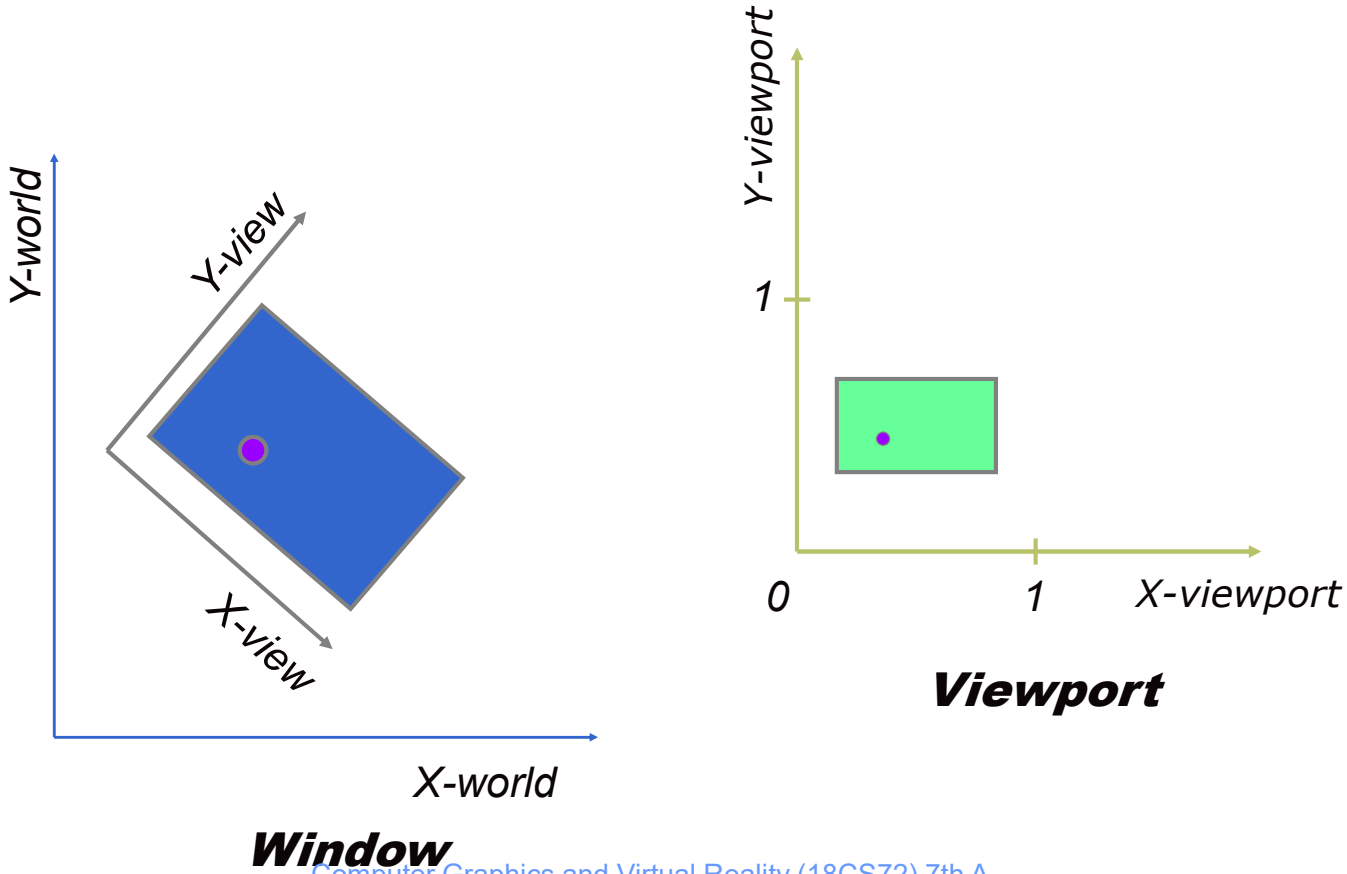
World-Coordinate Clipping Window



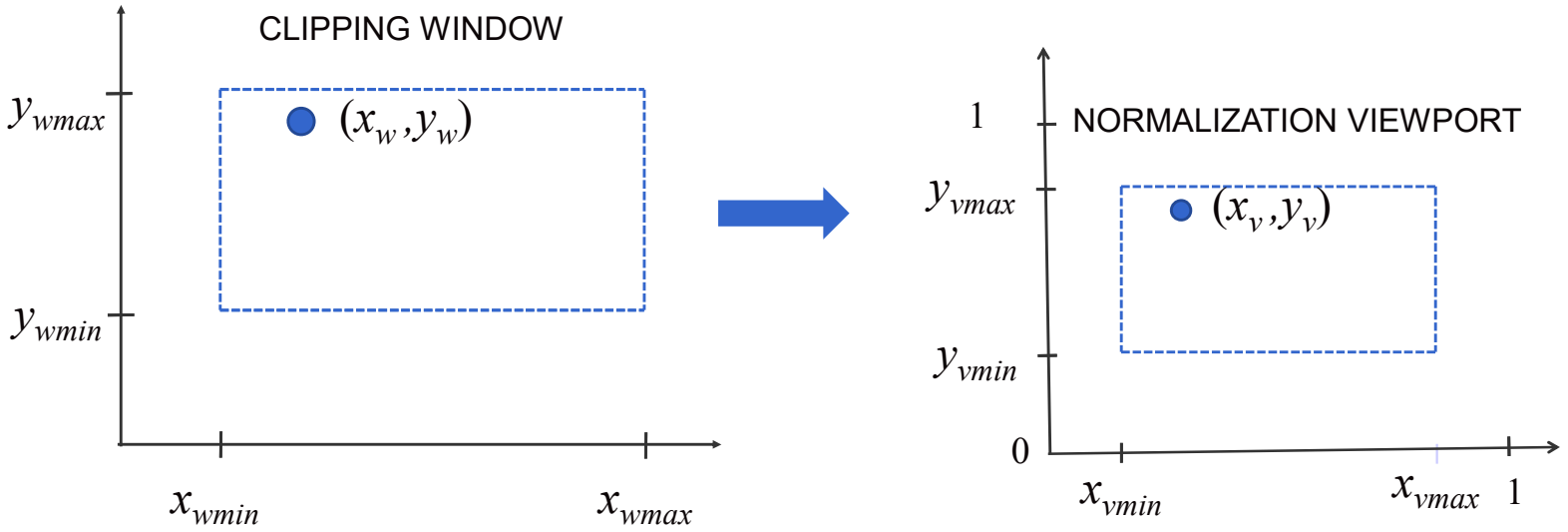


A triangle
(a), with a selected reference point and orientation vector, is translated and rotated to position
(b) within a clipping window.

Window-to-viewport coordinate transformation



Normalization and Viewport Transformations



A point (*xw, yw*) in a world-coordinate clipping window is mapped to viewport coordinates (*xv, yv*), within a unit square, so that the relative positions of the two points in their respective rectangles are the same.

Mapping the Clipping Window into a Normalized Window

- To transform the world-coordinate point into the same relative position within the viewport, we require that

$$\frac{x_v - x_{vmin}}{x_{vmax} - x_{vmin}} = \frac{x_w - x_{wmin}}{x_{wmax} - x_{wmin}}$$

$$\frac{y_v - y_{vmin}}{y_{vmax} - y_{vmin}} = \frac{y_w - y_{wmin}}{y_{wmax} - x_{wmin}}$$

Mapping the Clipping Window into a Normalized Window

- Solving these equations for the viewport position (x_v, y_v)

$$x_v = s_x x_w + t_x \qquad s_x = \frac{x_{vmax} - x_{vmin}}{x_{wmax} - x_{wmin}}$$

$$y_v = s_y y_w + t_y \qquad s_y = \frac{y_{vmax} - y_{vmin}}{y_{wmax} - y_{wmin}}$$

$$t_x = \frac{x_{wmax}\, x_{vmin} - x_{wmin}\, x_{vmax}}{x_{wmax} - x_{wmin}}$$

$$t_y = \frac{y_{wmax}\, y_{vmin} - y_{wmin}\, y_{vmax}}{y_{wmax} - y_{wmin}}$$

Mapping the Clipping Window into a Normalized Window

- Using the matrix notation we have

$$\mathbf{S}=\begin{bmatrix} s_x & 0 & x_{wmin}(1-s_x) \\ 0 & s_y & x_{ymin}(1-s_y) \\ 0 & 0 & 1 \end{bmatrix}$$

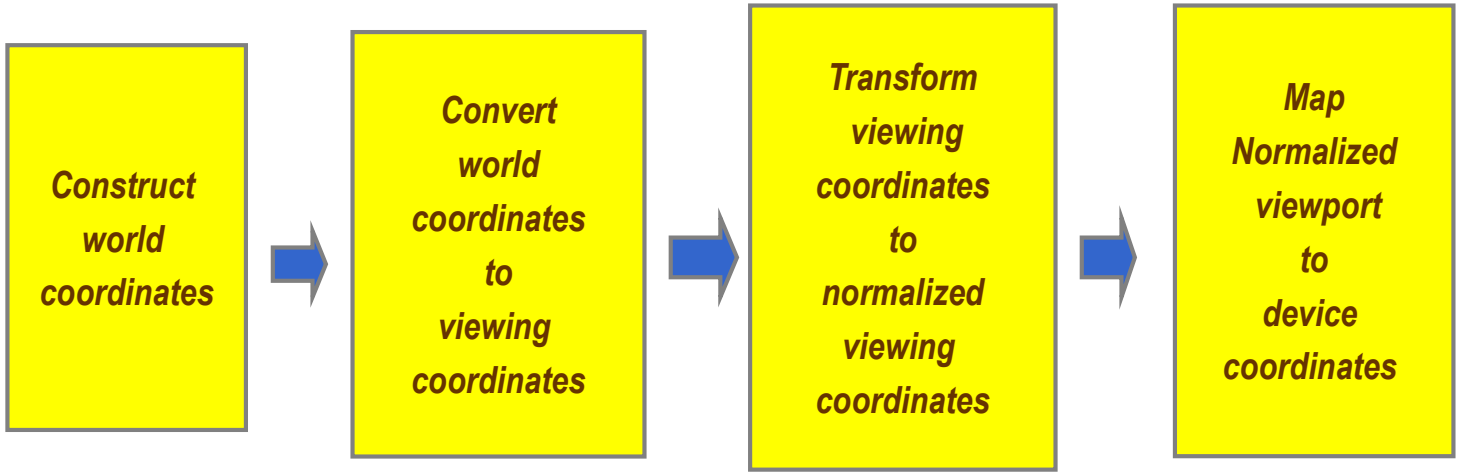
$$\mathbf{T}=\begin{bmatrix} 1 & 0 & x_{vmin}-x_{wmin} \\ 0 & 1 & y_{vmin}-y_{wmin} \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{M}_{\text{window, normviewpo rt}} = \mathbf{T}\cdot\mathbf{S} = \begin{bmatrix} s_x & 0 & t_x \\ 0 & s_y & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

World Coordinates to Viewport Coordinates

1. Scale the clipping window to the size of the viewport using a fixed point position of (xw_{min}, yw_{min})
2. Translate (xw_{min}, yw_{min}) to (xv_{min}, yv_{min})

2D Viewing Transformation Pipeline



OpenGL 2D viewing functions

- Remember: OpenGL doesn't directly support 2D
- Projection mode
  - glMatrixMode (GL_PROJECTION)
  - glLoadIdentity()
- GLU Clipping window function
  - Orthogonal projection
    gluOrtho2D (xwmin, xwmax, ywmin, ywmax)
  - If we do not specify a clipping window, the default coordinates are

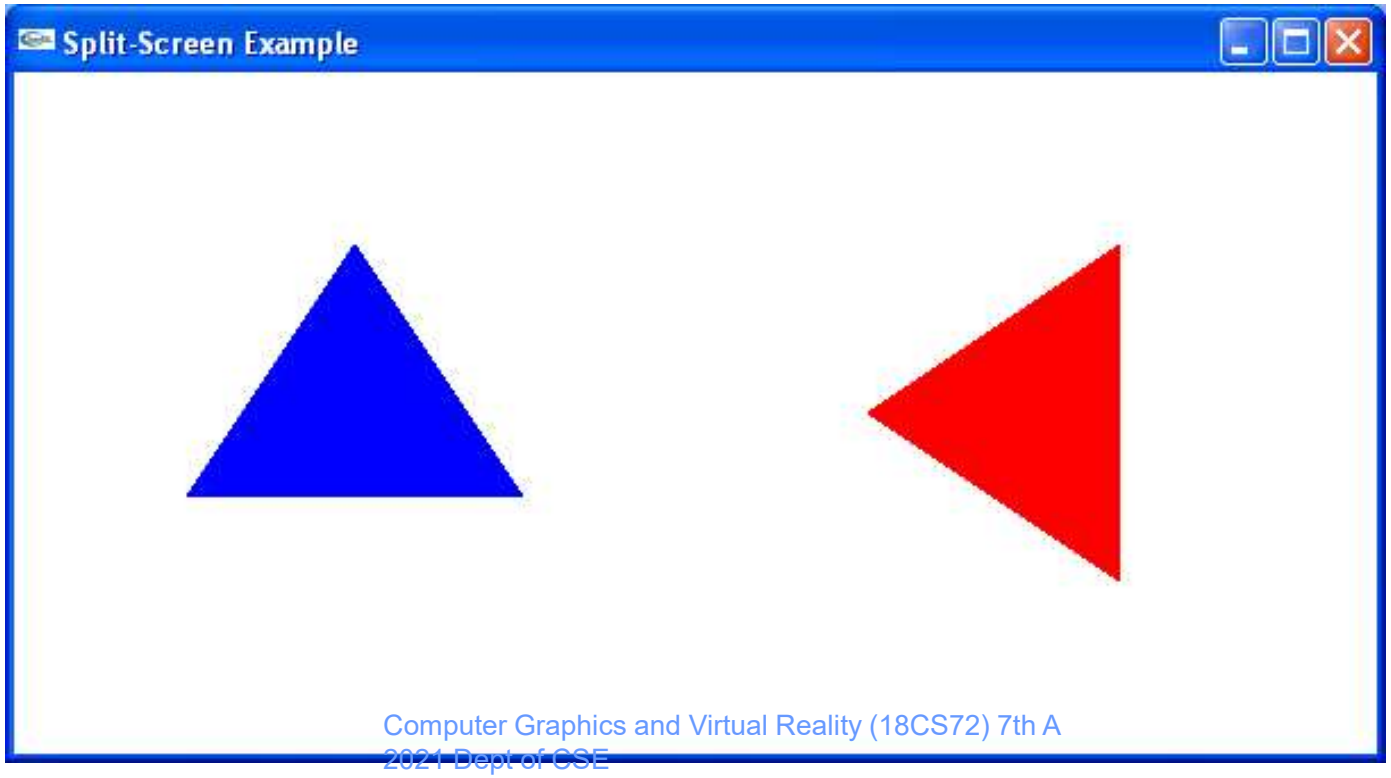
    $(xw_{min}, yw_{min})=(-1.0,-1.0)$ and
    $(xw_{max}, yw_{max}) = (1.0,1.0)$

OpenGL 2D viewing functions (cont.)

- Viewport
  - glViewport (xvmin, yvmin, vpwidth, vpHeight)
    default is size of the display window
    - xvmin and yvmin are the positions of the lower-left cornet of the viewport
    - vpwidth, vpHeight are the pixel width and height of the viewport
    - If we do not use glViewport, the default viewport size and position are the same as the display window

OpenGL 2D viewing program example

- ch8ViewingProgram2D.cpp

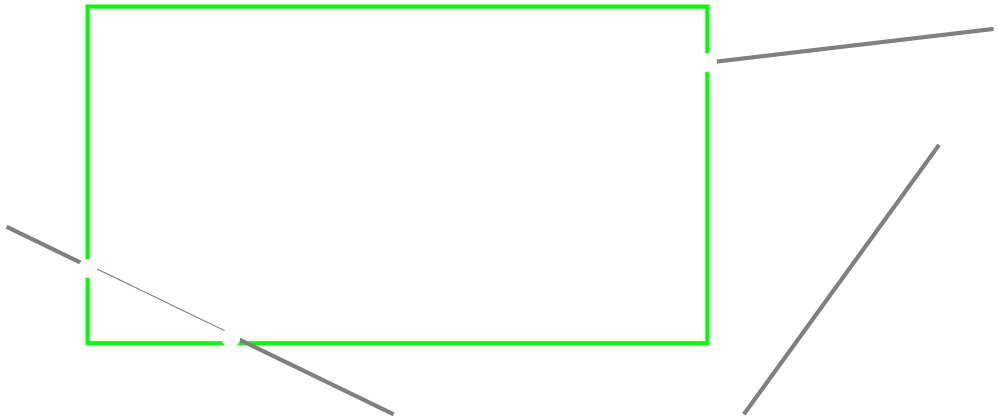


Clipping

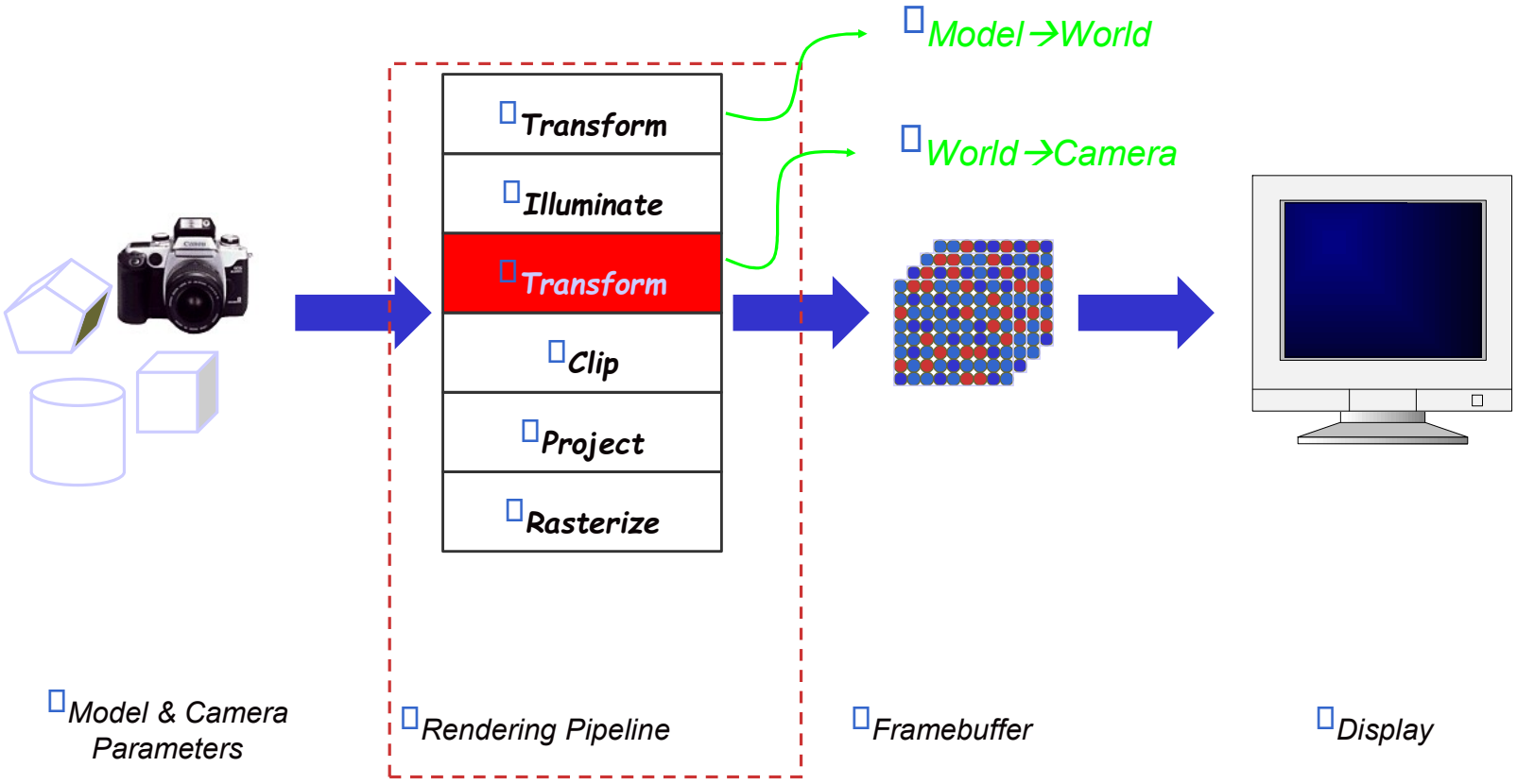
- We've been assuming that all primitives (lines, triangles, polygons) lie entirely within the viewport
- In general, this assumption will not hold:

Clipping (cont.)

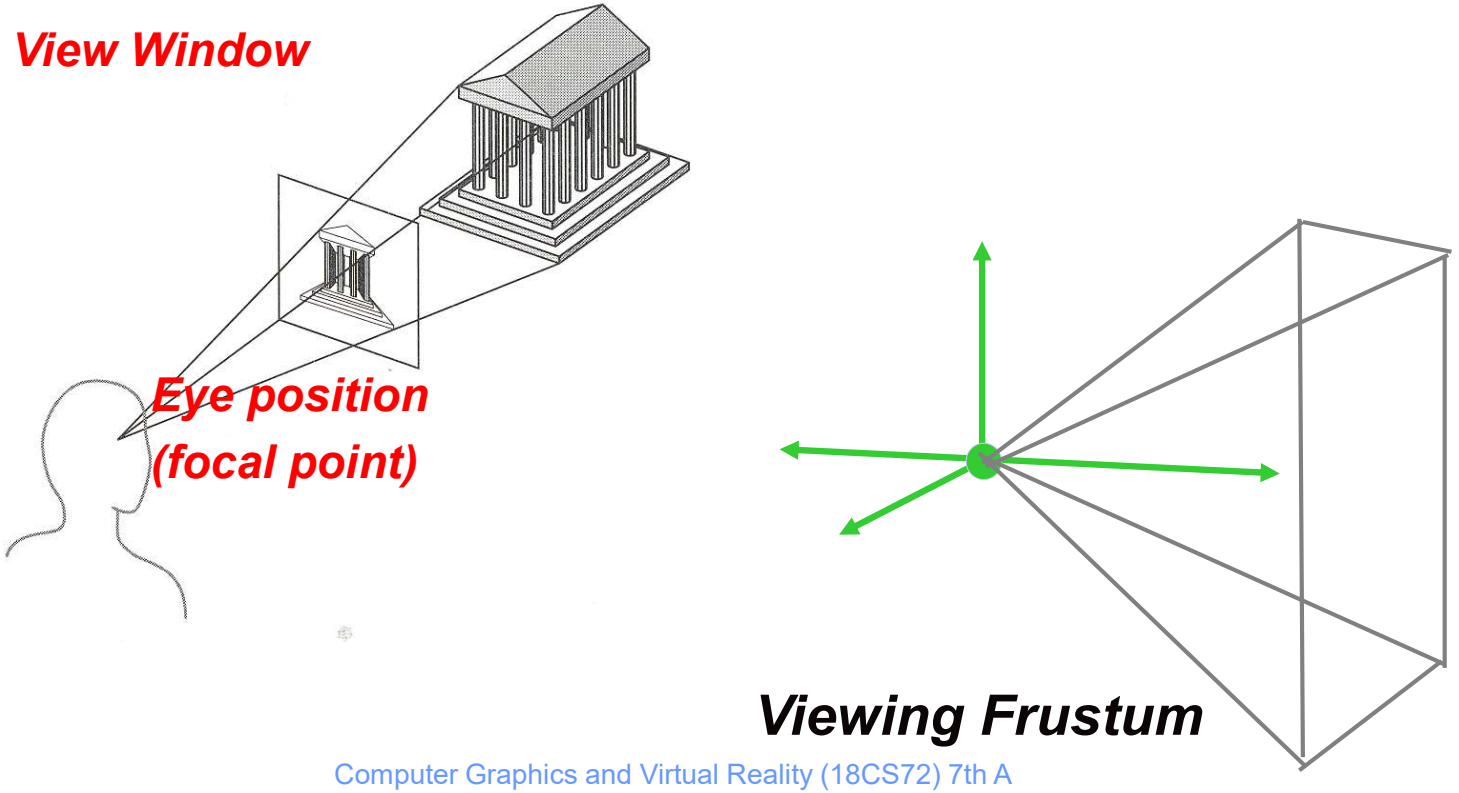
- Analytically calculating the portions of primitives within the viewport



Why World→Camera before clipping?



Clip to what?



Why Clip?

- Bad idea to rasterize outside of framebuffer bounds
- Also, don’t waste time scan converting pixels outside window

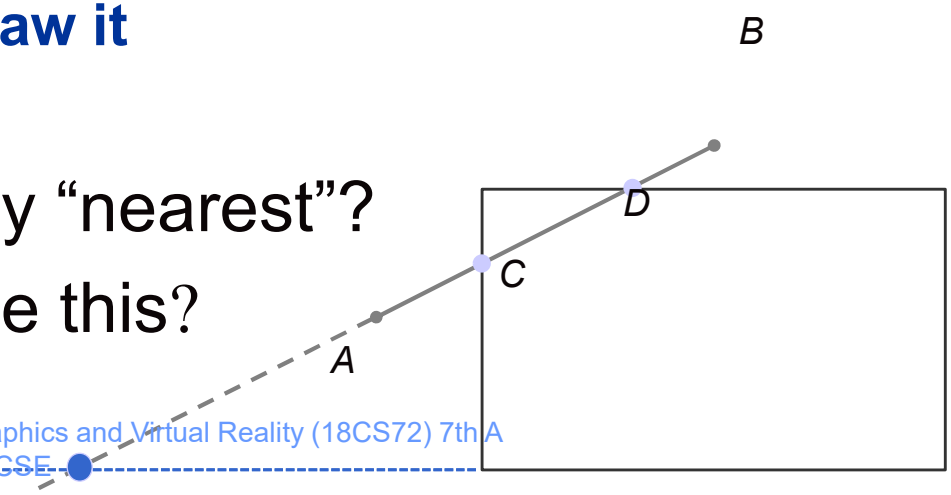
Clipping

The naive approach to clipping lines:

```
for each line segment
    for each edge of view_window
      find intersection point
      pick "nearest" point
  if anything is left, draw it
```

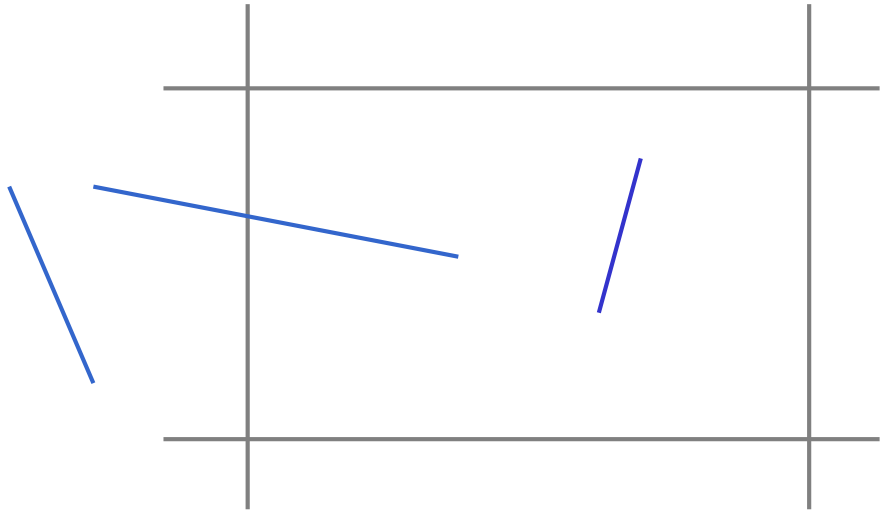
What do we mean by "nearest"?

How can we optimize this?



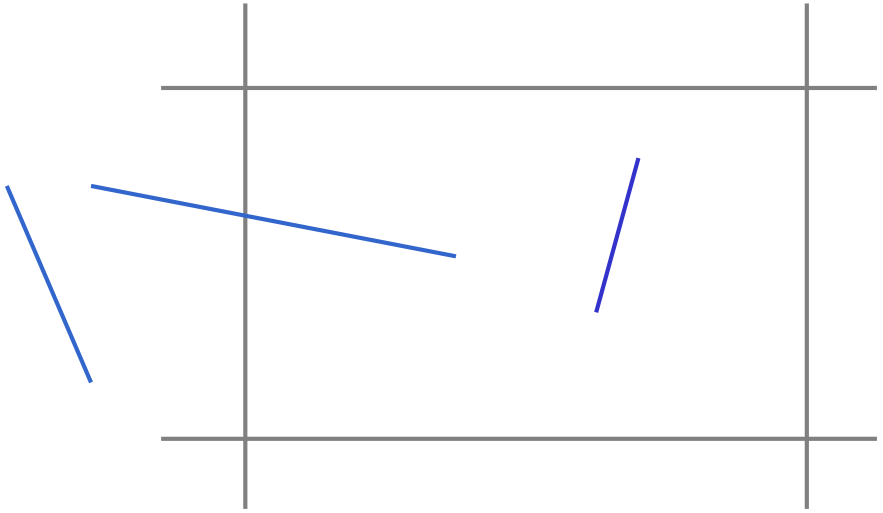
Trivial Accepts

- Big optimization: trivial accept/rejects
- How can we quickly determine whether a line segment is entirely inside the viewport?
- Answer: test both endpoints.



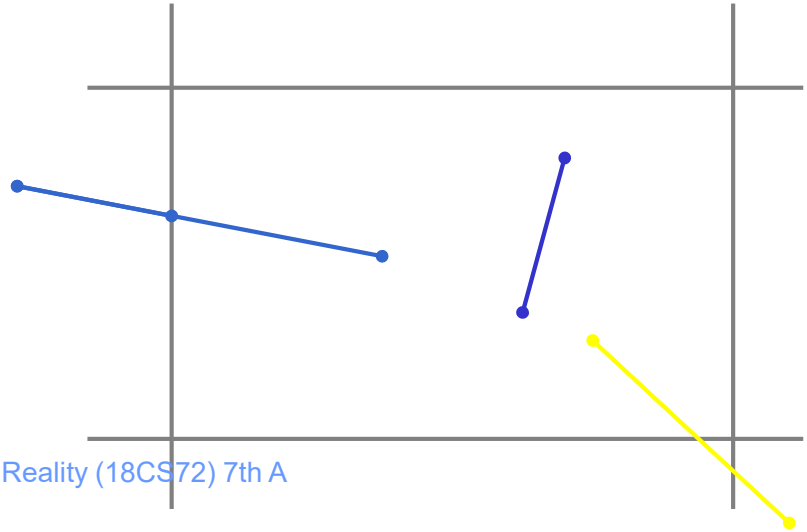
Trivial Rejects

- How can we know a line is outside viewport?
- Answer: if both endpoints on wrong side of same edge, can trivially reject line



Clipping Lines to Viewport

- Discard segments of lines outside viewport
  - Trivially accept lines with both endpoints **inside all edges of the viewport**
  - Trivially reject lines with both endpoints **outside the same edge of the viewport**
  - Otherwise, reduce to trivial cases **by splitting into two segments**



✓ 2D Viewing

2-D viewing: The viewing pipeline, viewing coordinate reference frame, window to view-port coordinate transformation,

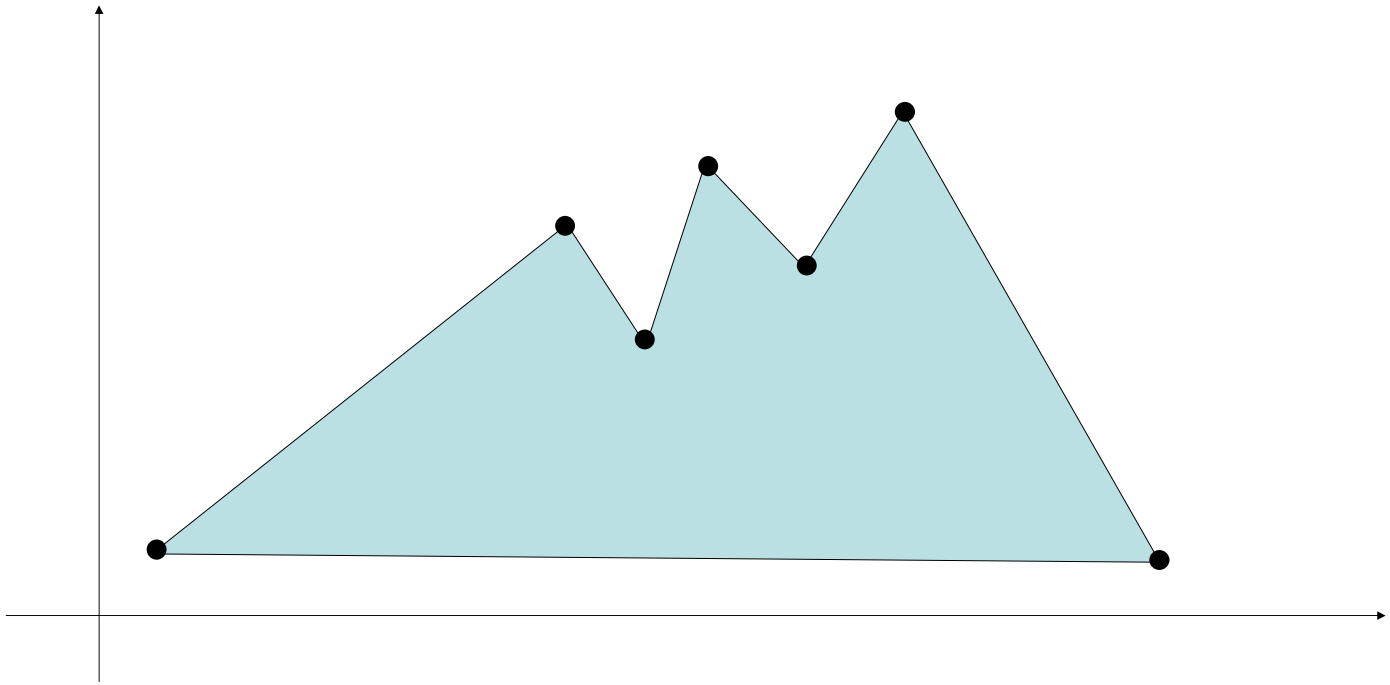
viewing functions

Cohen-Sutherland and Cyrus-beck ***line clipping algorithms***,

Sutherland –Hodgeman ***polygon clipping algorithm***.

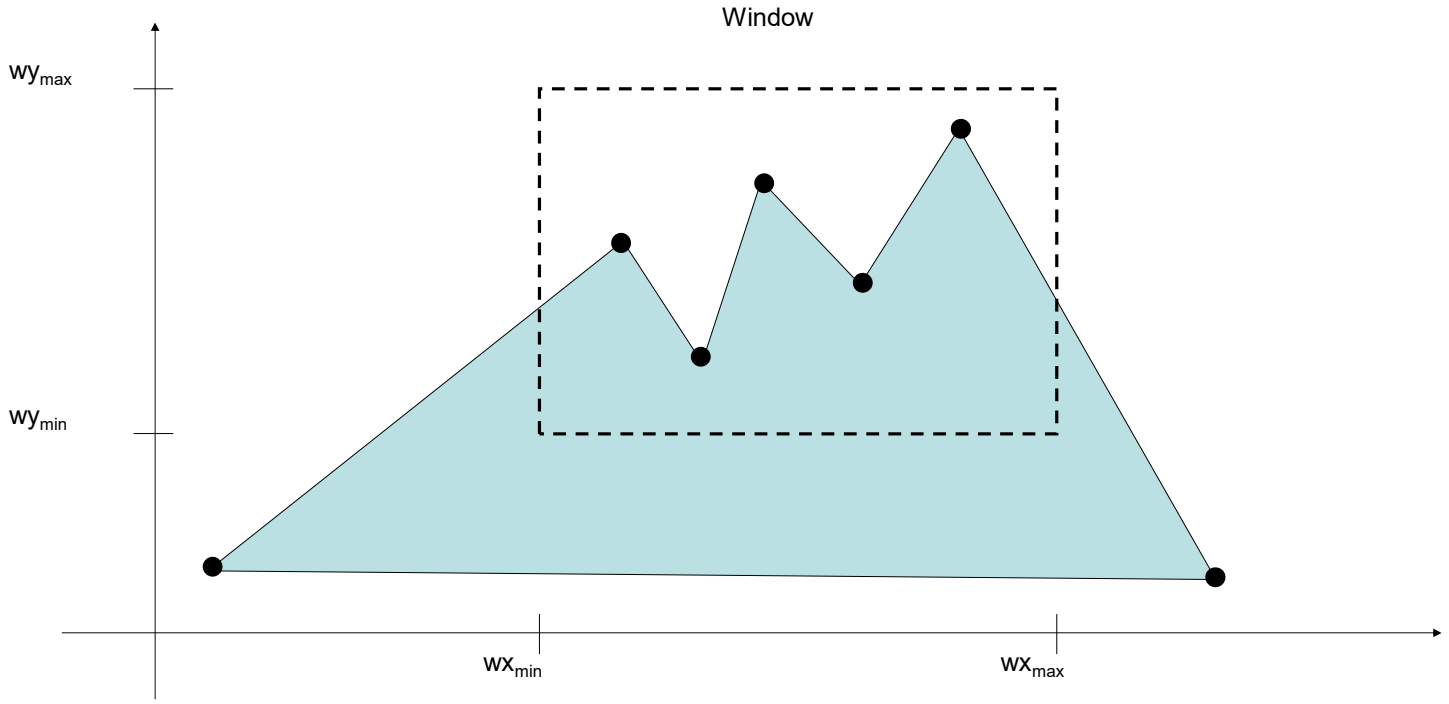
Windowing I

A scene is made up of a collection of objects specified in world coordinates



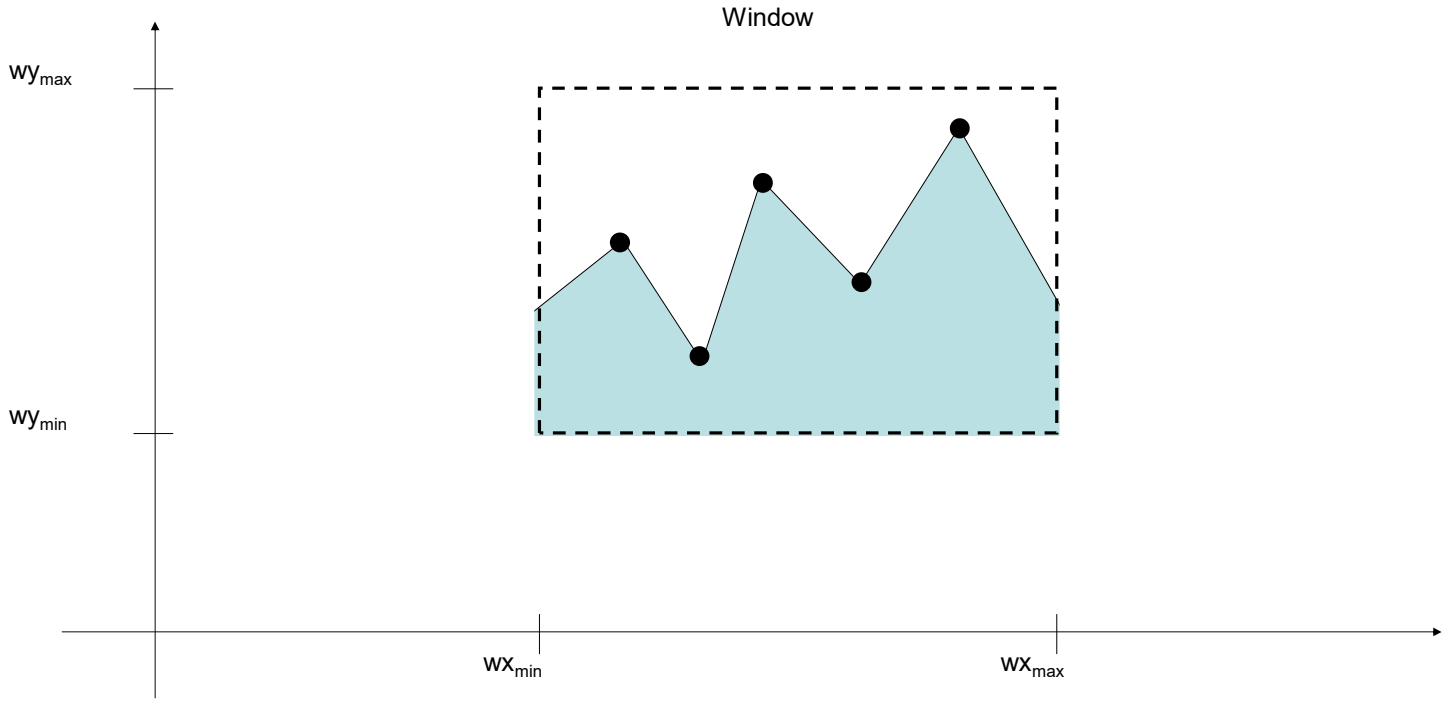
Windowing II

When we display a scene only those objects within a particular window are displayed



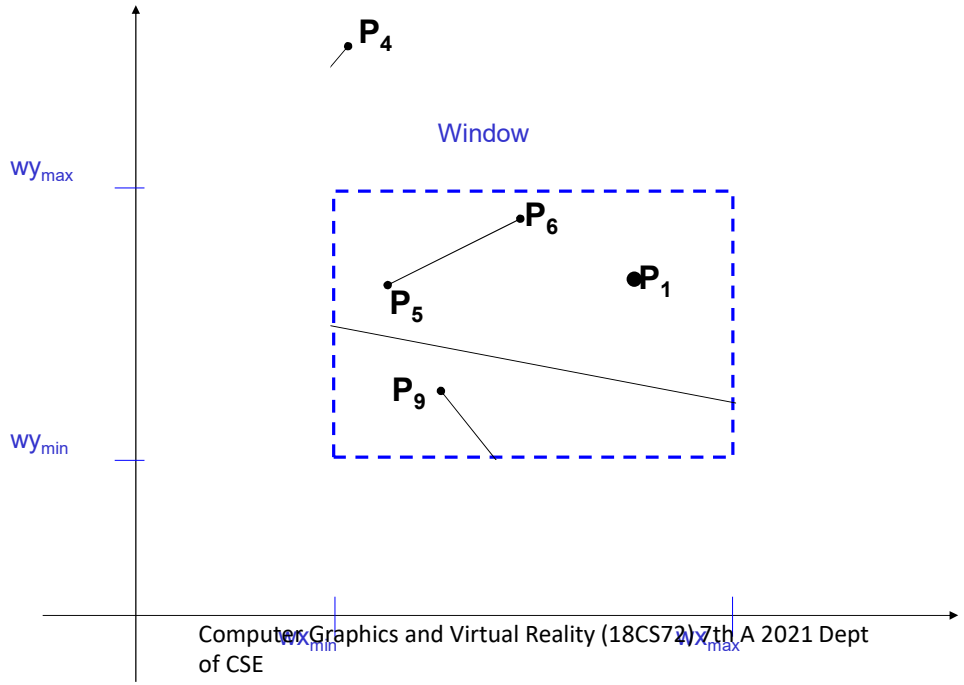
Windowing III

Because drawing things to a display takes time we *clip* everything outside the window



Clipping

For the image below consider which lines and points should be kept and which ones should be clipped

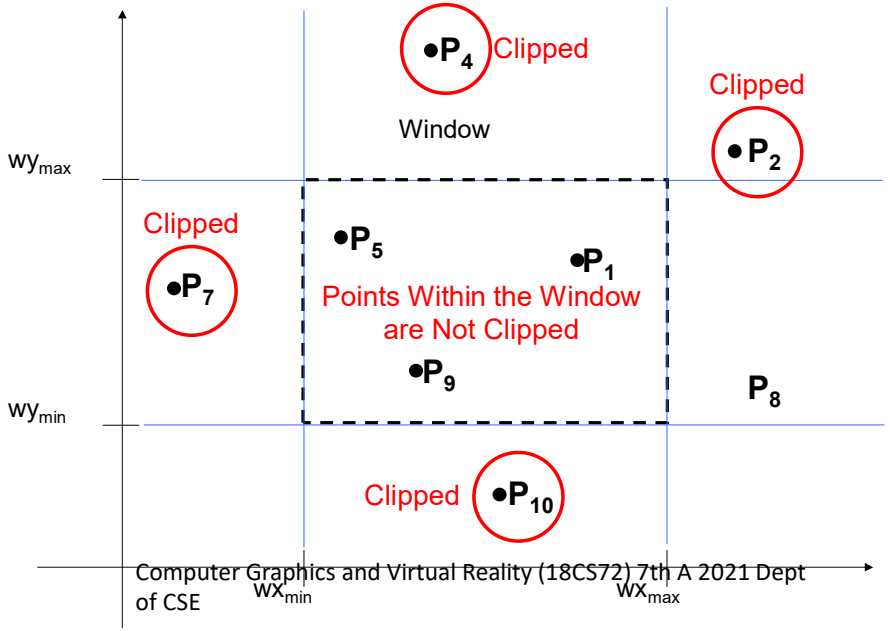


Point Clipping

Easy - a point (x,y) is not clipped if:

$$wx_{min} \le x \le wx_{max} \text{ AND } wy_{min} \le y \le wy_{max}$$

otherwise it is clipped

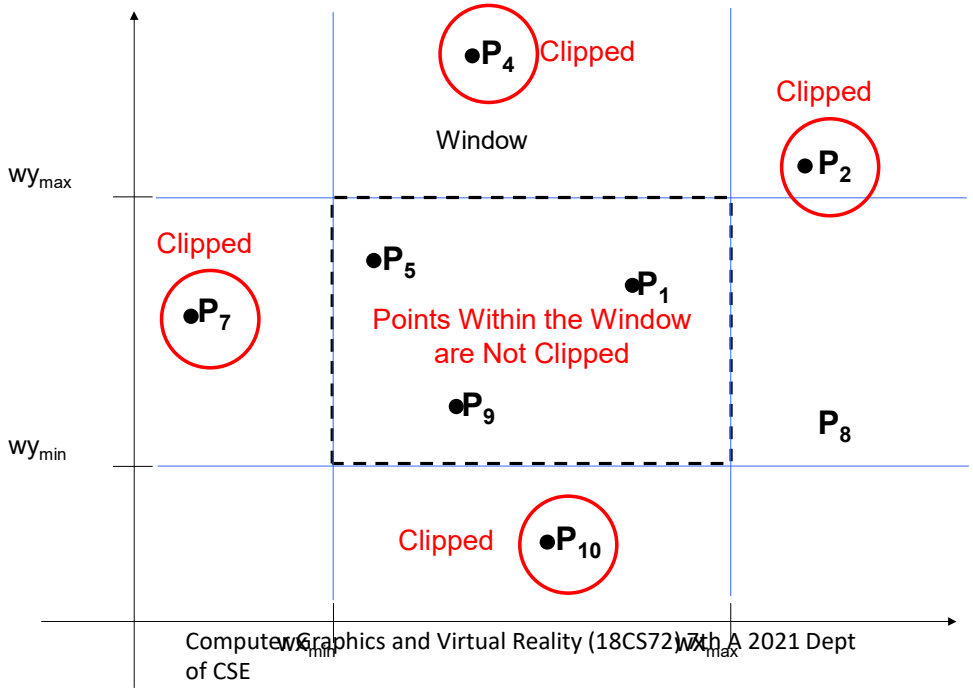


Cohen-Sutherland Clipping Algorithm

Easy - a point (x,y) is not clipped if:

$$wx_{min} \le x \le wx_{max} \text{ AND } wy_{min} \le y \le wy_{max}$$

otherwise it is clipped

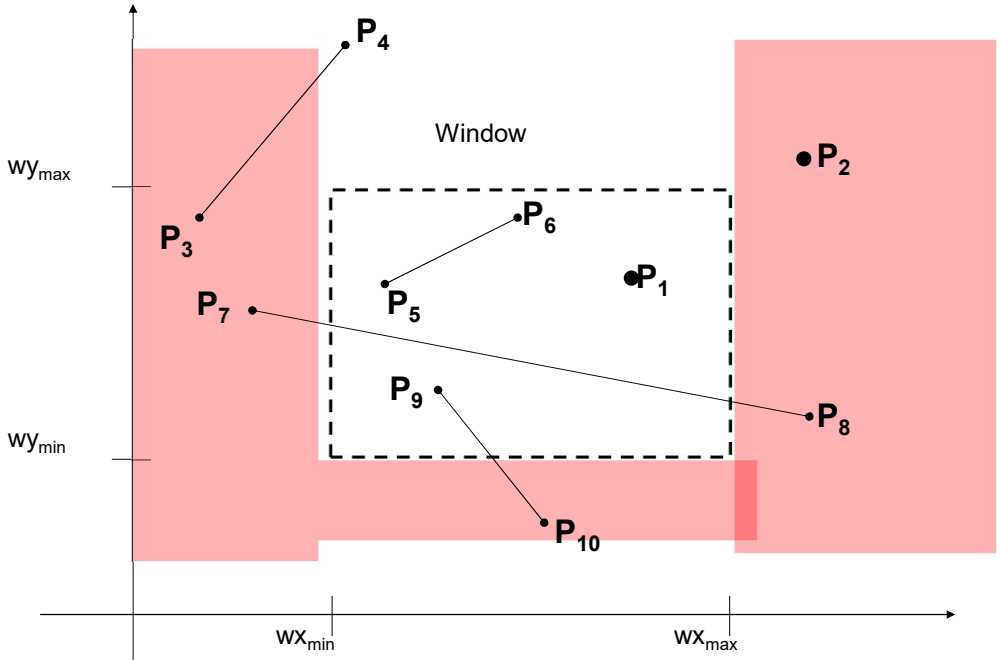


Clipping

Point clipping is easy:

– For point (x,y) the point is not clipped if

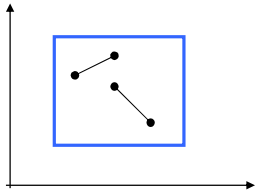
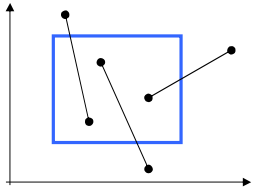
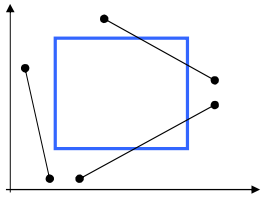
$$wx_{min} \le x \le wx_{max} \text{ AND } wy_{min} \le y \le wy_{max}$$



Before Clipping

Line Clipping

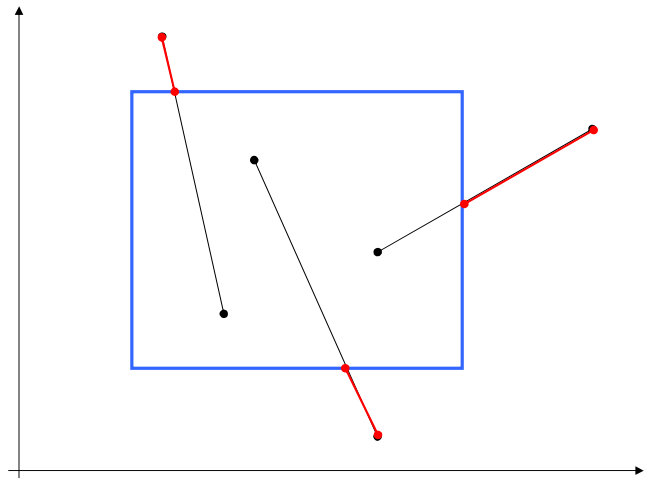
Harder - examine the end-points of each line to see if they are in the window or not

Situation	**Solution**	**Example**
Both end-points inside the window	Don't clip	
One end-point inside the window, one outside	Must clip	
Both end-points outside the window	Don't know!	

Brute Force Line Clipping

Brute force line clipping can be performed as follows:

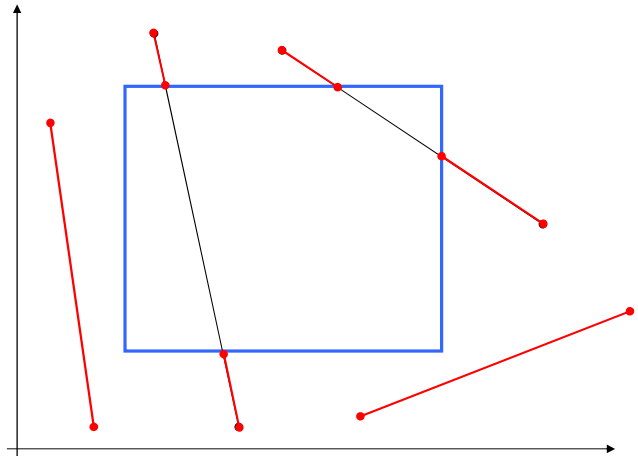
- ***Don't clip*** lines with both end-points within the window
- For lines with ***one end-point inside the window and one end-point outside***,



calculate the intersection point (using the equation of the line) and clip from this point out

Brute Force Line Clipping (cont…)

– For lines with both end-points outside the window test the line for intersection with all of the window boundaries, and clip appropriately



However, calculating line intersections is computationally expensive

Because a scene can contain so many lines, the brute force approach to clipping is much too slow.

Cohen-Sutherland Clipping Algorithm

- An efficient line clipping algorithm
- The key advantage of the algorithm is that it vastly reduces the number of line intersections that must be calculated

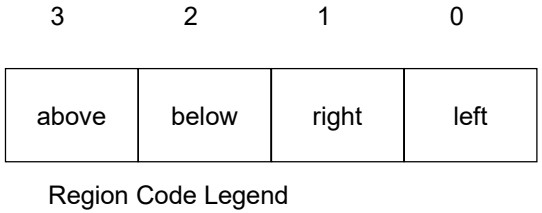
Dr. Ivan E. Sutherland co-developed the Cohen-Sutherland clipping algorithm. Sutherland is a graphics giant and includes amongst his achievements the invention of the head mounted display.

Cohen is something of a mystery – can anybody find out who he was?

Cohen-Sutherland: World Division

World space is divided into regions based on the window boundaries

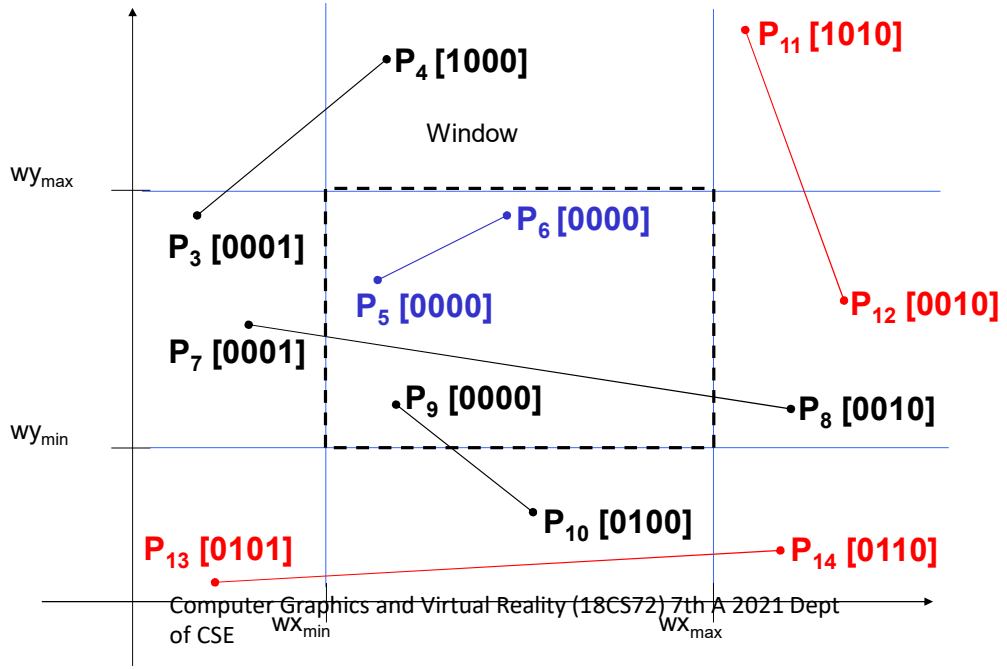
- Each region has a unique four bit region code
- Region codes indicate the position of the regions with respect to the window



1001	1000	1010
0001	0000 Window	0010
0101	0100	0110

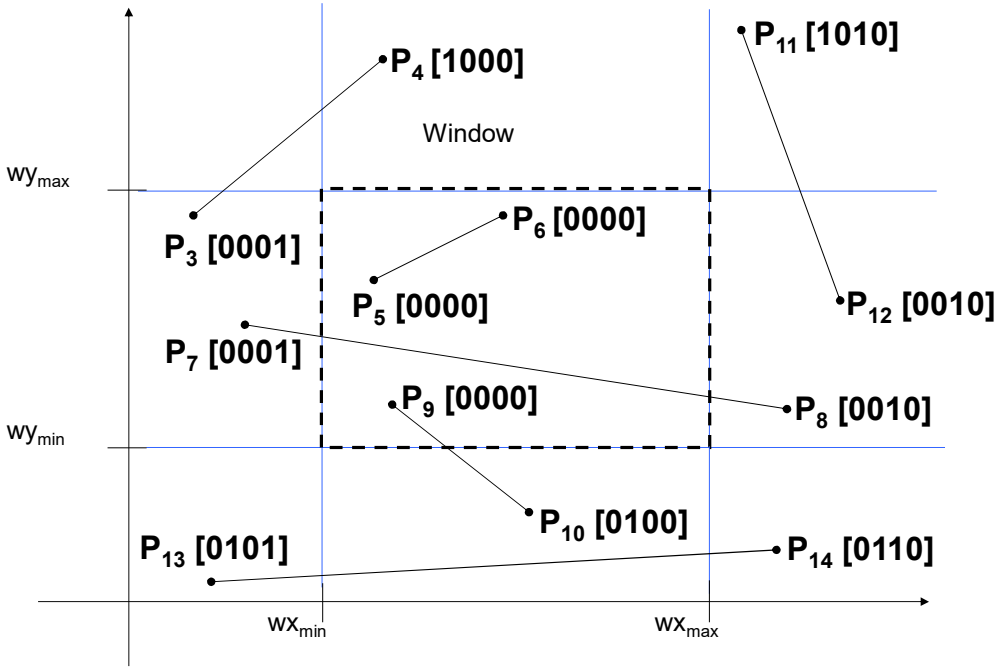
Cohen-Sutherland Clipping Algorithm VI

Let's consider the lines remaining below



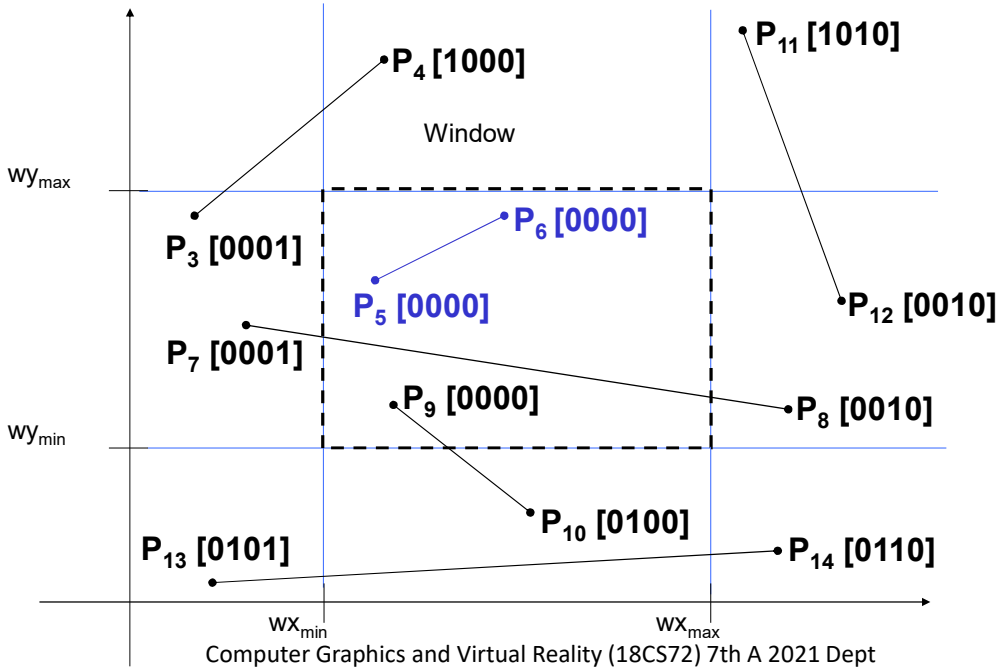
Cohen-Sutherland: Labelling

Every end-point is labelled with the appropriate region code



Cohen-Sutherland: Lines In The Window

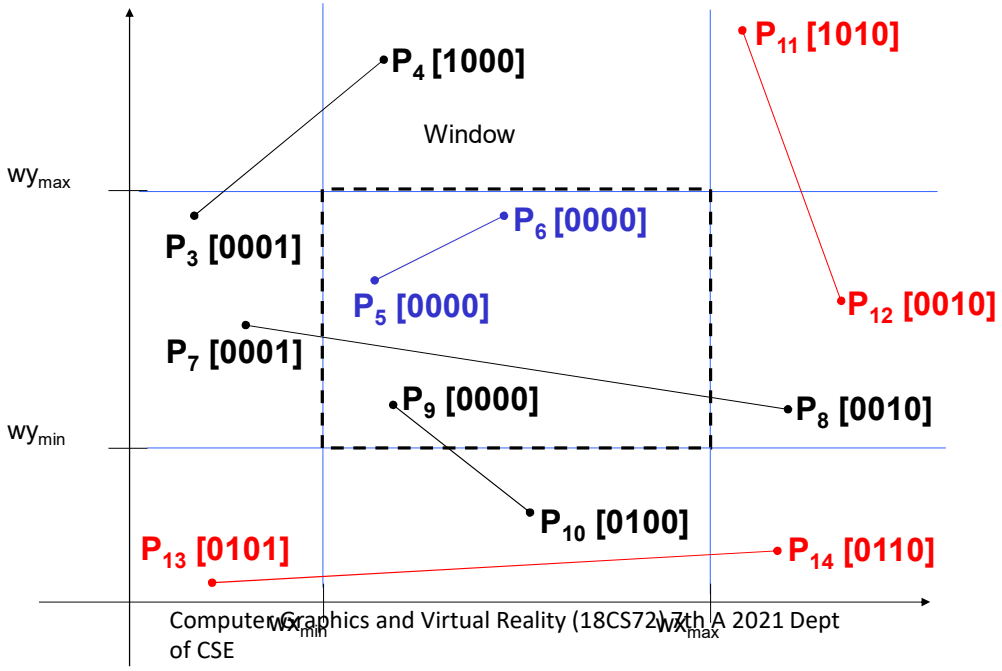
Lines completely contained within the window boundaries have region code [0000] for both end-points so are not clipped



Cohen-Sutherland: Lines Outside The Window

Any lines with a common set bit in the region codes of both end-points can be clipped

–The AND operation can efficiently check this



Cohen-Sutherland: Other Lines

Lines that cannot be identified ***as completely inside or outside*** the window may or may not cross the window interior

These ***lines are processed*** as follows:

- Compare an end-point outside the window to a boundary (choose any order in which to consider boundaries e.g. left, right, bottom, top) and determine how much can be discarded
- If the remainder of the line is entirely inside or outside the window, retain it or clip it respectively

Cohen-Sutherland: Other Lines (cont…)

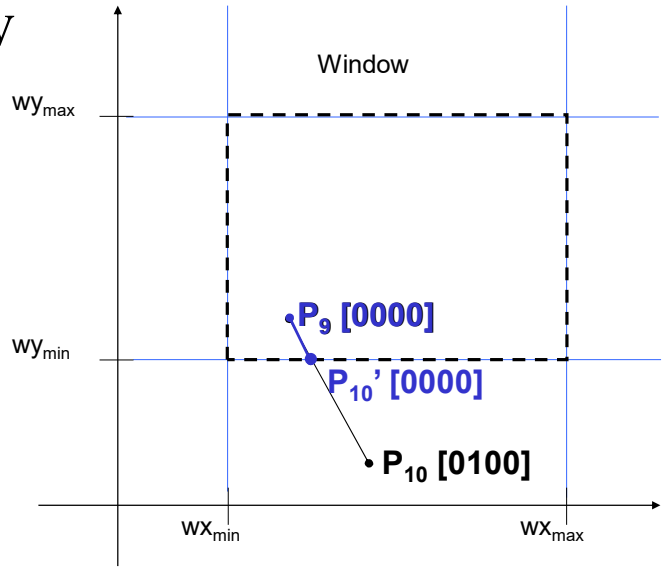
- Otherwise, compare the remainder of the line against the other window boundaries
- Continue until the line is either discarded or a segment inside the window is found

We can use the region codes to determine which window boundaries should be considered for intersection

- To check if a line crosses a particular boundary we compare the appropriate bits in the region codes of its end-points
- If one of these is a 1 and the other is a 0 then the line crosses the boundary

Cohen-Sutherland Examples

Consider the line P_9 to P_{10} below

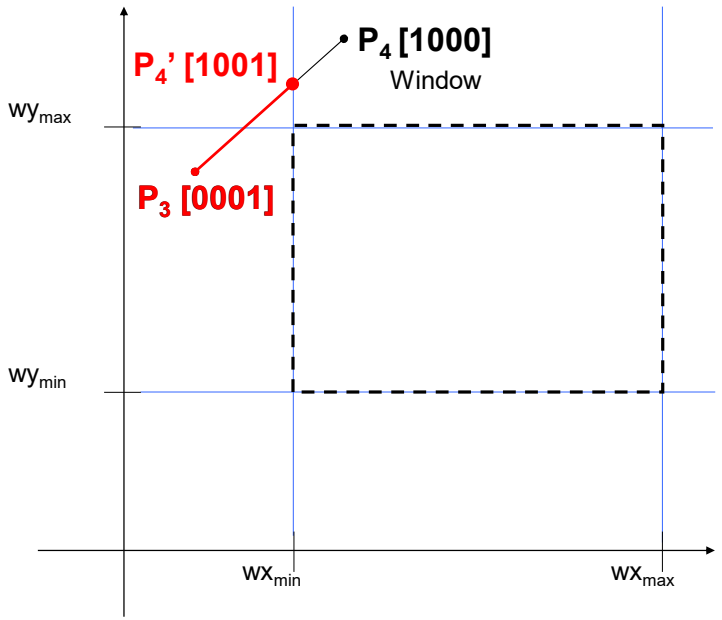


- Start at P_{10}
- From the region codes of the two end-points we know the line doesn't cross the left or right boundary
- Calculate the intersection of the line with the bottom boundary to generate point P_{10}'
- The line P_9 to P_{10}' is completely inside the window so is retained

Cohen-Sutherland Examples (cont…)

Consider the line P_3 to P_4 below

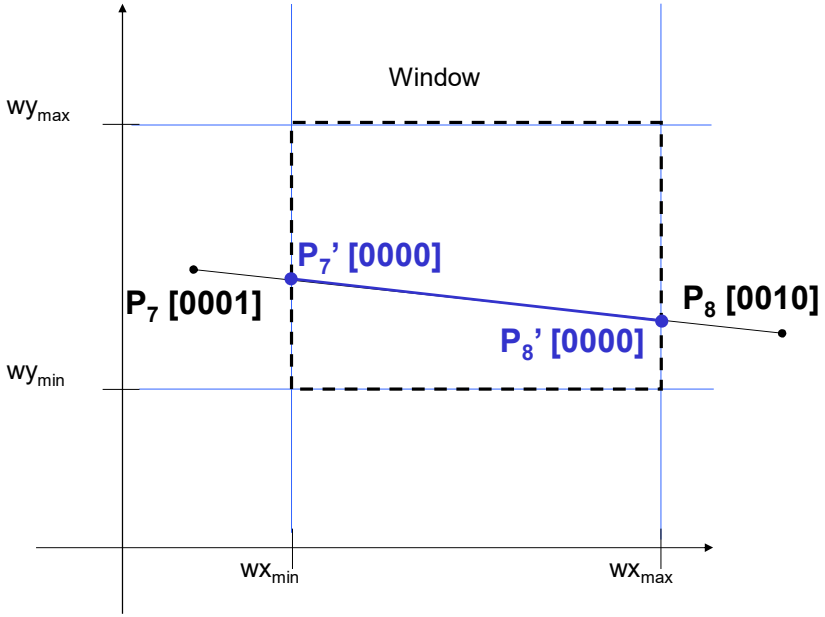
- Start at P_4
- From the region codes of the two end-points we know the line crosses the left boundary so calculate the intersection point to generate P_4'
- The line P_3 to P_4' is completely outside the window so is clipped



Cohen-Sutherland Examples (cont…)

Consider the line P_7 to P_8 below

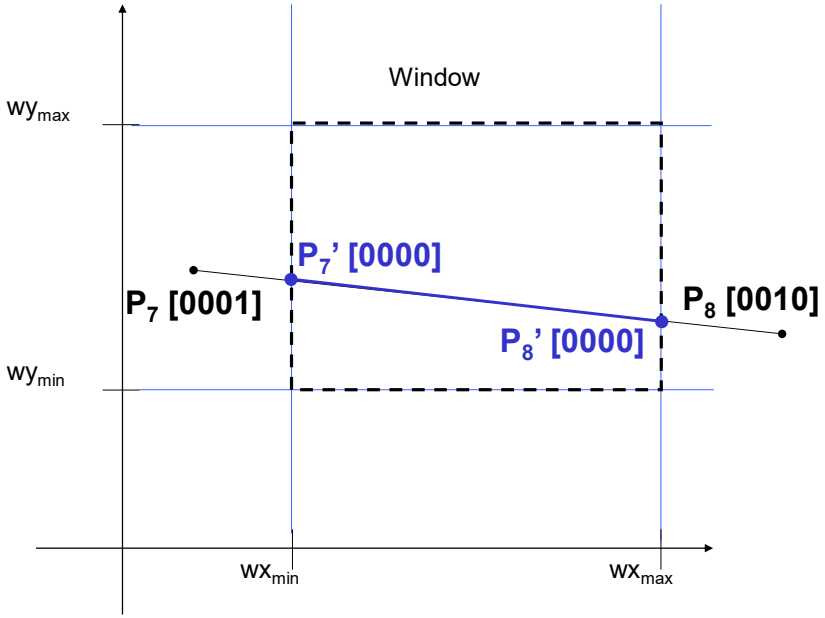
- Start at P_7
- From the two region codes of the two end-points we know the line crosses the left boundary so calculate the intersection point to generate P_7'



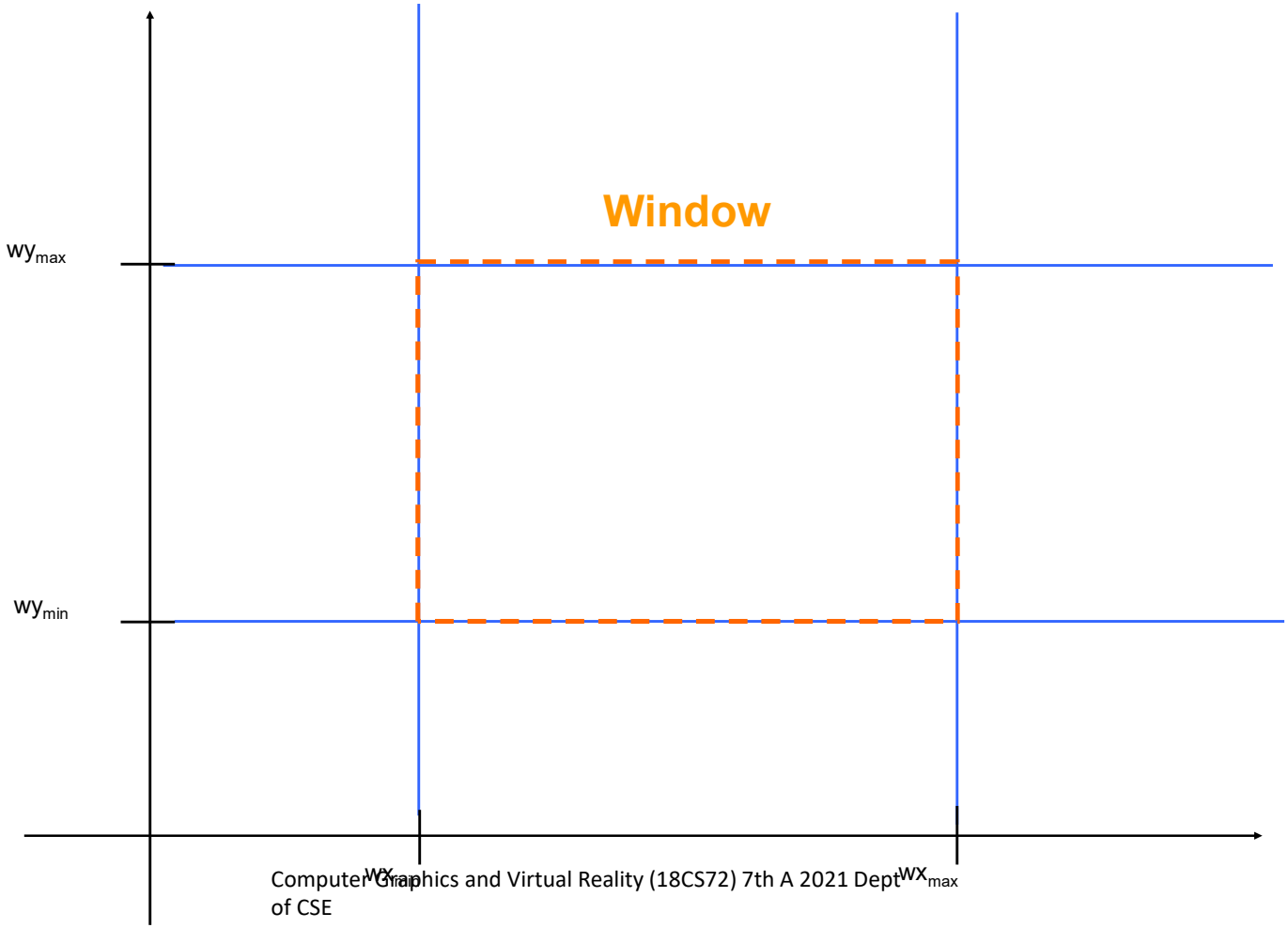
Cohen-Sutherland Examples (cont…)

Consider the line P_7' to P_8

- Start at P_8
- Calculate the intersection with the right boundary to generate P_8'
- P_7' to P_8' is inside the window so is retained



Cohen-Sutherland Worked Example



Calculating Line Intersections

Intersection points with the window boundaries are calculated using the line-equation parameters

- Consider a line with the end-points (x_1, y_1) and (x_2, y_2)
- The y-coordinate of an intersection with a vertical window boundary can be calculated using:

$$y = y_1 + m\,(x_{boundary} - x_1)$$

where $x_{boundary}$ can be set to either wx_{min} or wx_{max}

Calculating Line Intersections (cont…)

– The x-coordinate of an intersection with a horizontal window boundary can be calculated using:

$$x = x_1 + (y_{\text{boundary}} - y_1) / m$$

where $y_{boundary}$ can be set to either wy_{min} or wy_{max}

– m is the slope of the line in question and can be calculated as $m = (y_2 - y_1) / (x_2 - x_1)$

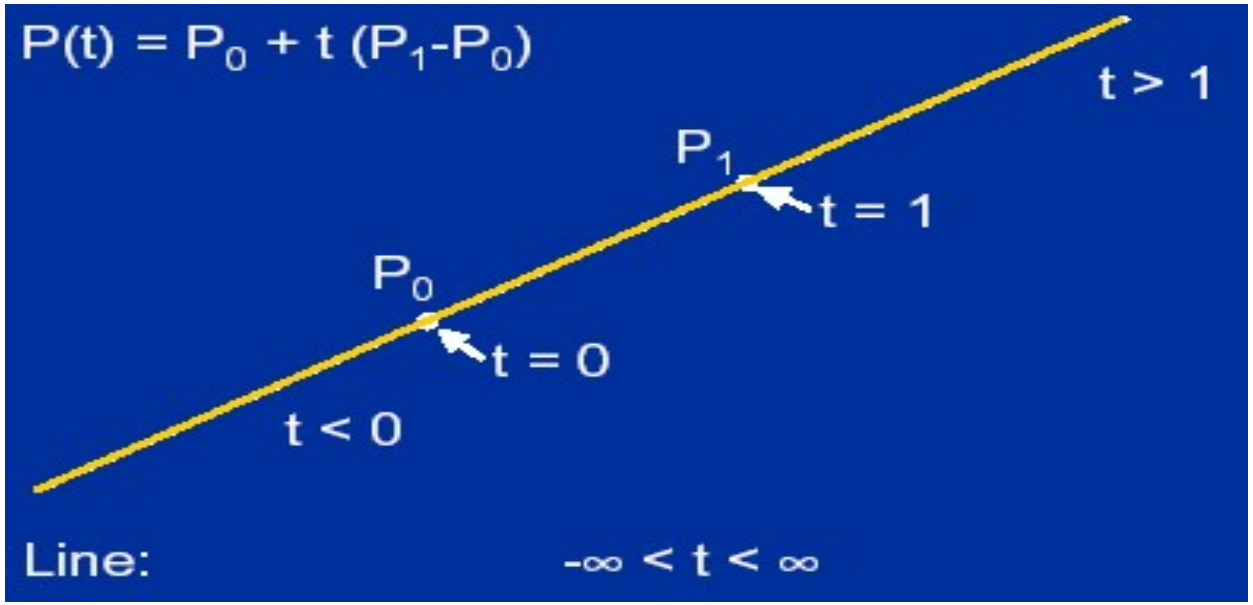
Cohen-Sutherland Line Clipping

- This algorithm can be very efficient if it can accept and reject primitives trivially
  - If clip window is large wrt scene data
    - Most primitives are accepted trivially
  - If clip window is much smaller than scene data
    - Most primitives are rejected trivially
- Good for hardware implementation

Solving Simultaneous Equations

- Equation of a line:
  - Slope-intercept: $y = mx + h$ *difficult for vertical line*
  - Implicit Equation: $Ax + By + C = 0$
  - Parametric: Line defined by two points, P_0 and P_1
    - $P(t) = P_0 + (P_1 - P_0)\, t$
    - $x(t) = x_0 + (x_1 - x_0)\, t$
    - $y(t) = x_0 + (y_1 - y_0)\, t$

Parametric Lines and Intersections



For L_1 :

$x = x0_{l1} + t(x1_{l1} - x0_{l1})$

$y = y0_{l1} + t(y1_{l1} - y0_{l1})$

For L_2 :

$x = x0_{l2} + t(x1_{l2} - x0_{l2})$

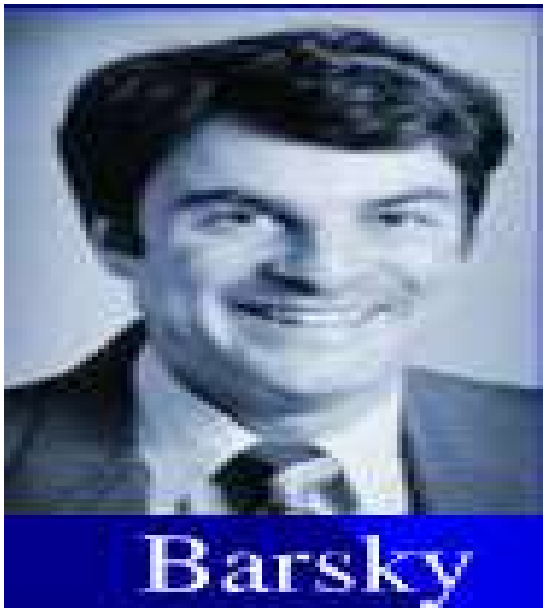
$y = y0_{l2} + t(y1_{l2} - y0_{l2})$

The Intersection Point:

$$x0_{l1} + t_1 (x1_{l1} - x0_{l1}) = x0_{l2} + t_2 (x1_{l2} - x0_{l2})$$

$$y0_{l1} + t_1 (y1_{l1} - y0_{l1}) = y0_{l2} + t_2 (y1_{l2} - y0_{l2})$$

Liang-Barsky Algorithm (1984)



The ONLY algorithm named for **Chinese people** in Computer Graphics course books

Liang, Y.D., and Barsky, B., "A New Concept and Method for Line Clipping", ***ACM Transactions on Graphics***, 3(1):1-22, January 1984.

Liang-Barsky Algorithm (1984)

- Because of horizontal and vertical clip lines:

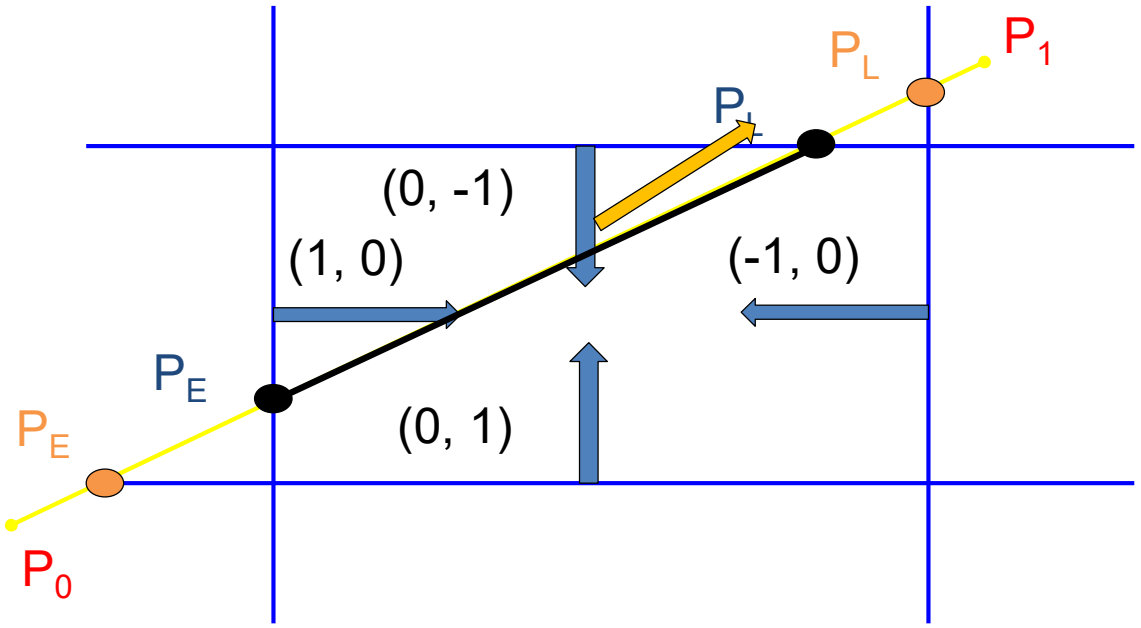
  Many computations reduce
- Normals
- Pick constant points on edges
- solution for t:

$t_L = -(x_0 - x_{left}) / (x_1 - x_0)$

$t_R = (x_0 - x_{right}) / -(x_1 - x_0)$

$t_B = -(y_0 - y_{bottom}) / (y_1 - y_0)$

$t_T = (y_0 - y_{top}) / -(y_1 - y_0)$



Liang-Barsky Algorithm (1984)

Edge	Inner normal	A	P_1-A	$t = -\frac{\boldsymbol{N} \cdot (P_1 - A)}{\boldsymbol{N} \cdot (P_2 - P_1)}$
Left $x=XL$	$(1, 0)$	(XL, y)	$(x_1\text{-}XL, y_1\text{-}y)$	$\frac{(x_1 - XL)}{-(x_2 - x_1)}$
Right $x=XR$	$(-1, 0)$	(XR, y)	$(x_1\text{-}XR, y_1\text{-}y)$	$\frac{-(x_1 - XR)}{x_2 - x_1}$
Bottom $y=YB$	$(0, 1)$	(x, YB)	$(x_1\text{-}x, y_1\text{-}YB)$	$\frac{(y_1 - YB)}{-(y_2 - y_1)}$
Top $y=YT$	$(0, -1)$	(x, YT)	$(x_1\text{-}x, y_1\text{-}YT)$	$\frac{-(y_1 - YT)}{y_2 - y_1}$

Liang-Barsky Algorithm (1984)

Let $\Delta x = x2 - x1$, $\Delta y = y2 - y1$:

$$\begin{cases} r_1 = -\Delta x, & s_1 = x_1 - x_L, \\ r_2 = \Delta x, & s_2 = x_R - x_1, \\ r_3 = -\Delta y, & s_3 = y_1 - y_B, \\ r_4 = \Delta y, & s_4 = y_T - y_1, \end{cases}$$

$$t_k = s_k / r_k \quad , \quad k = L, R, B, T$$

- When $r_k<0$, t_k is entering point; when $r_k>0$, t_k is leaving point. If $r_k=0$ and $s_k<0$, then the line is invisible; else process other edges

Comparison

- Cohen-Sutherland:
  - Repeated clipping is expensive
  - Best used when trivial acceptance and rejection is possible for most lines
- Cyrus-Beck:
  - Computation of t-intersections is cheap
  - Computation of (x,y) clip points is only done once
  - Algorithm doesn't consider trivial accepts/rejects
  - Best when many lines must be clipped
- Liang-Barsky: Optimized Cyrus-Beck
- Nicholl et al.: Fastest, but doesn't do 3D

Liang-Barsky Line Clipping

Clipping: Overview of Steps

- Express line segments in parametric form
- Derive equations for testing if a point is inside the window
- Compute new parameter values for visible portion of line segment, if any
- Display visible portion of line segment

- The relative speed improvement over Sutherland-Cohen algorithm is as follows:
- 36% for 2D lines
  40% for 3D lines
  70% for 4D lines

Liang-Barsky - Algorithm

- Compute entering *t* values, which are q_k/p_k for each $p_k<0$
- Compute leaving *t* values, which are q_k/p_k for each $p_k>0$
- Parameter value for small *t* end of line is:t_{small}= max(0, entering *t*'s)
- parameter value for large t end of line is: t_{large}=min(1, leaving *t*'s)
- if $t_{small}<t_{large}$, there is a line segment - compute endpoints by substituting *t* values

Liang-Barsky Clipping

- Parametric clipping - view line in parametric form and reason about the parameter values
- More efficient, as not computing the coordinate values at irrelevant vertices
- Clipping conditions on parameter: Line is inside clip region for values of *t* such that:

$$x_{\min} \le x_1 + t\Delta x \le x_{\max} \qquad \Delta x = x_2 - x_1$$
$$y_{\min} \le y_1 + t\Delta y \le y_{\max} \qquad \Delta y = y_2 - y_1$$

Liang-Barsky (2)

- Infinite line intersects clip region edges when:

$$t_k = \frac{q_k}{p_k}$$

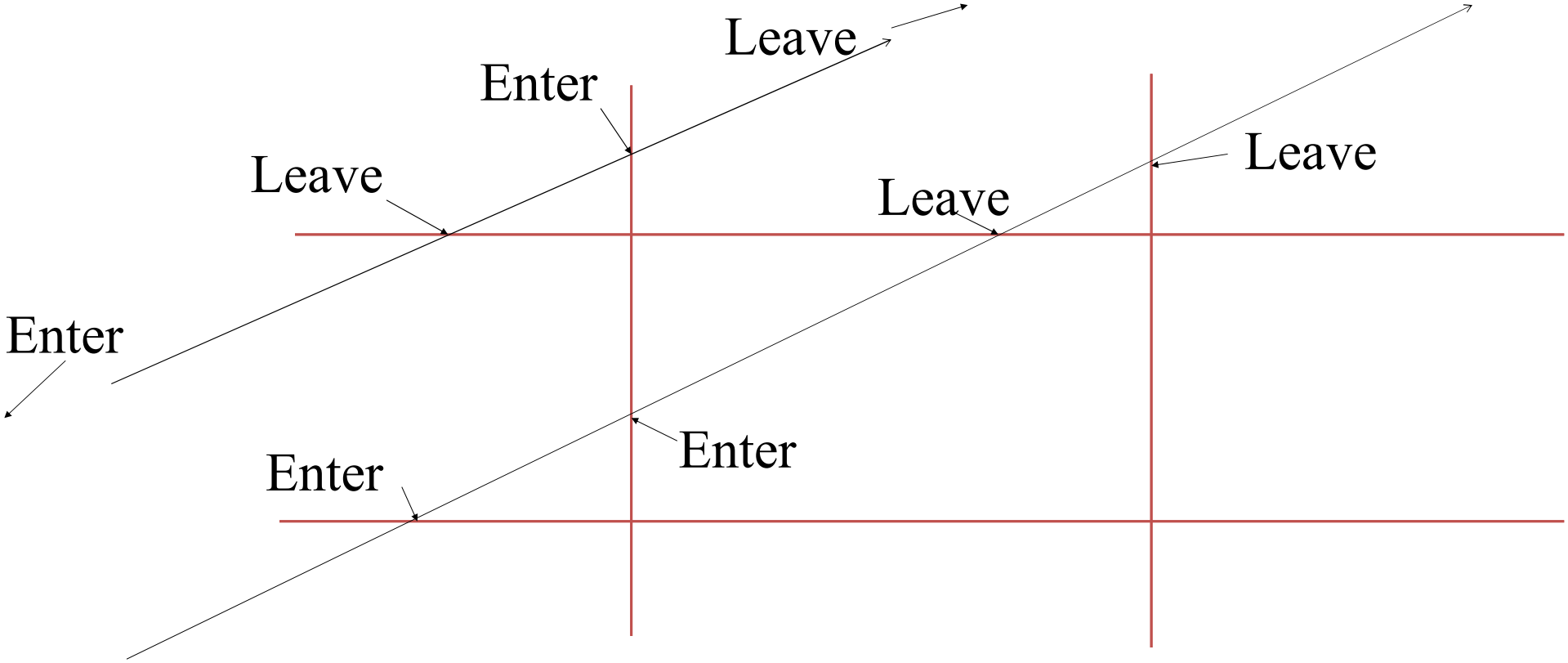
where

$$p_1 = -\Delta x \qquad q_1 = x_1 - x_{\min}$$
$$p_2 = \Delta x \qquad q_2 = x_{\max} - x_1$$
$$p_3 = -\Delta y \qquad q_3 = y_1 - y_{\min}$$
$$p_4 = \Delta y \qquad q_4 = y_{\max} - y_1$$

Liang-Barsky (3)

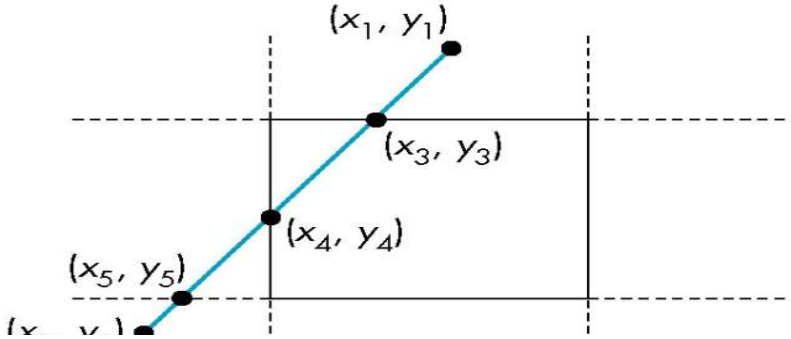
- When p_k<0, as t increases line goes from outside to inside - enter
- When p_k>0, line goes from inside to outside - leave
- When p_k=0, line is parallel to an edge (clipping is easy)
- If there is a segment of the line inside the clip region, sequence of infinite line intersections must go: enter, enter, leave, leave

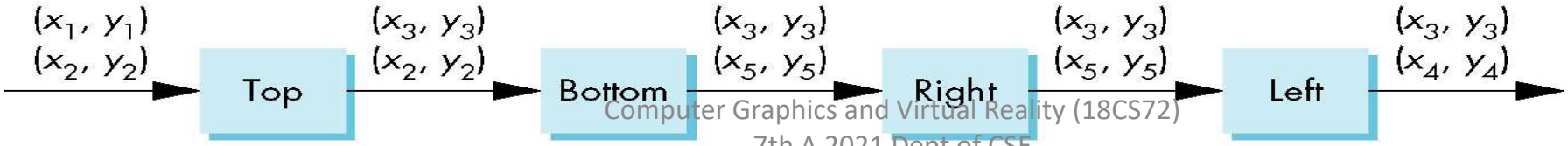
Liang-Barsky (4)



Pipeline Clipping of Line Segments

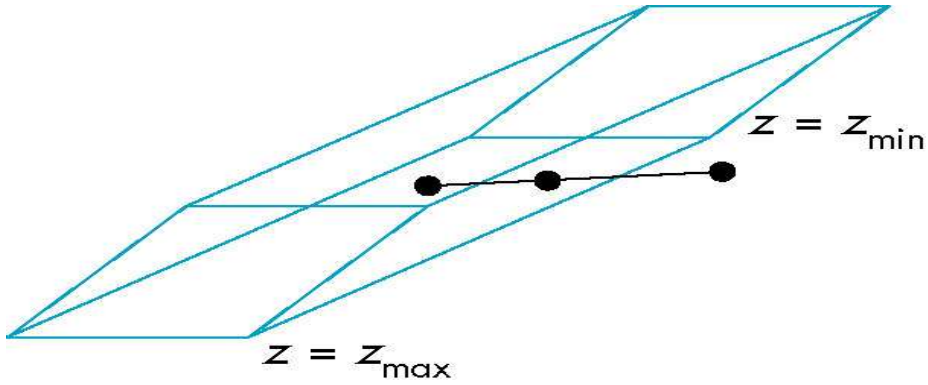
- Clipping against each side of window is independent of other sides
  - Can use four independent clippers in a pipeline



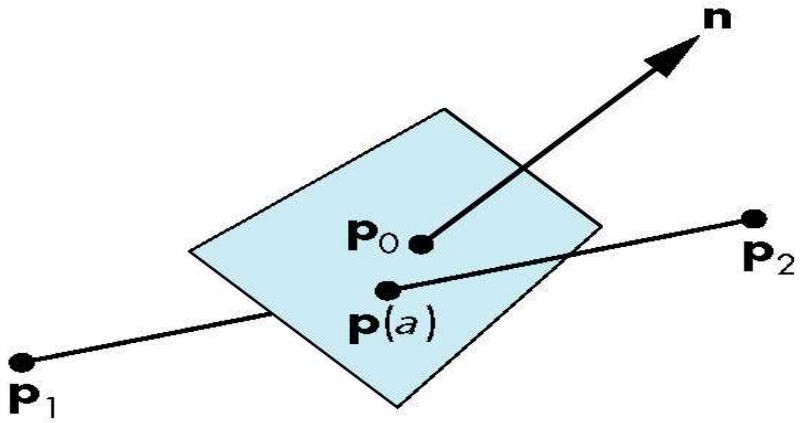


Clipping and Normalization

- General clipping in 3D requires intersection of line segments against arbitrary plane
- Example: oblique view



Plane-Line Intersections



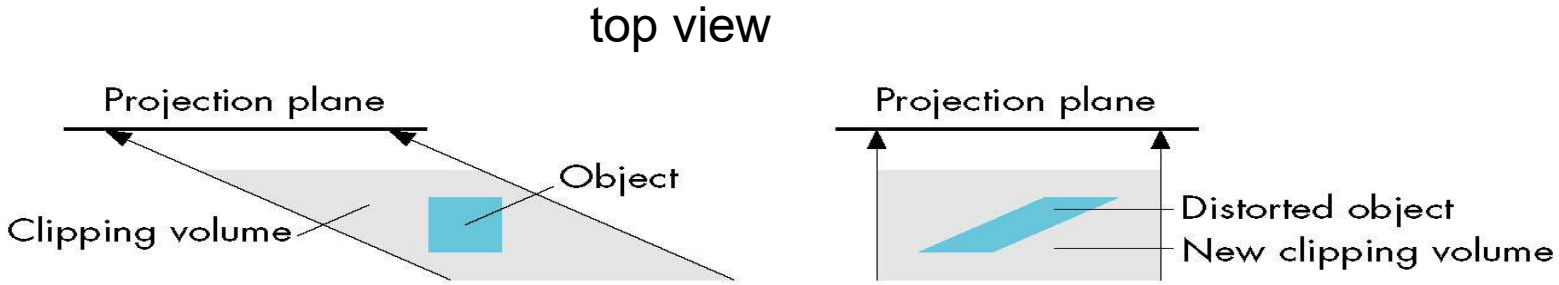
$$((P_0 - (P_1 + t(P_2 - P_1))) \bullet \mathrm{n} = 0$$

$$t = \frac{n \bullet (p_0 - p_1)}{n \bullet (p_2 - p_1)}$$

Point-to-Plane Test

- Dot product is relatively expensive
  - 3 multiplies
  - 5 additions
  - 1 comparison (to 0, in this case)
- Think about how you might optimize or special-case this?

Normalized Form



Normalization is part of viewing (pre clipping) but after normalization, we clip against sides of right parallelepiped

Typical intersection calculation now requires only a floating point subtraction, e.g. is $x > x_{max}$

Clipping Polygons

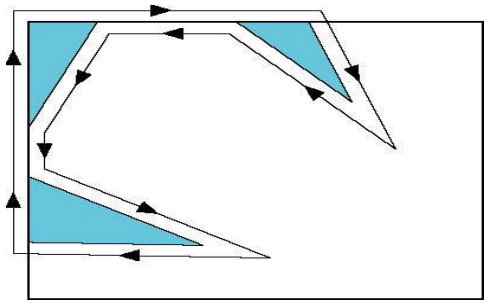
- Clipping polygons is more complex than clipping the individual lines
  - Input: polygon
  - Output: polygon, or nothing

Polygon Clipping

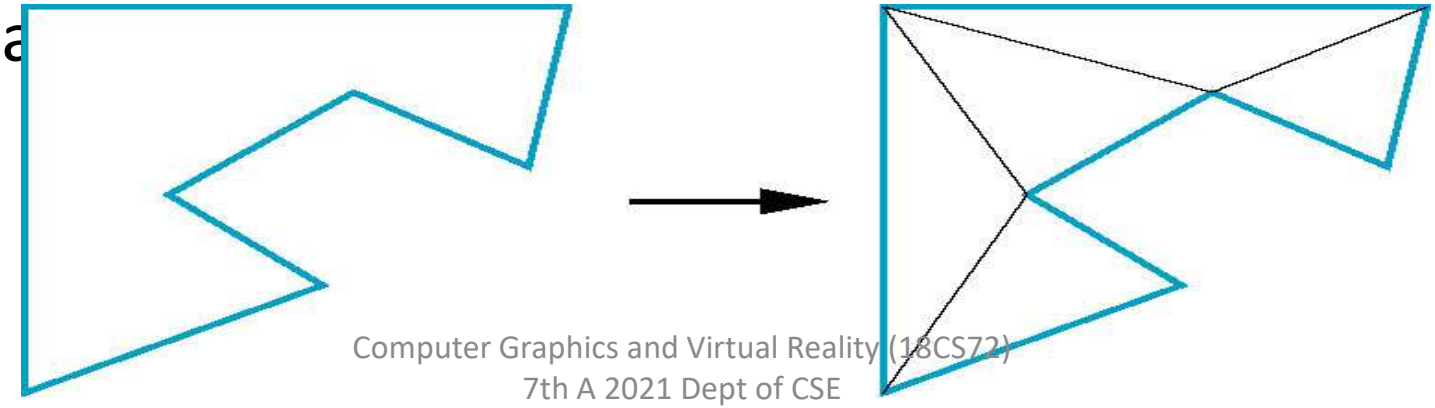
- Not as simple as line segment clipping
  - Clipping a line segment yields at most one line segment
  - Clipping a polygon can yield multiple polygons

- However, clipping a convex polygon can yield at most one other polygon

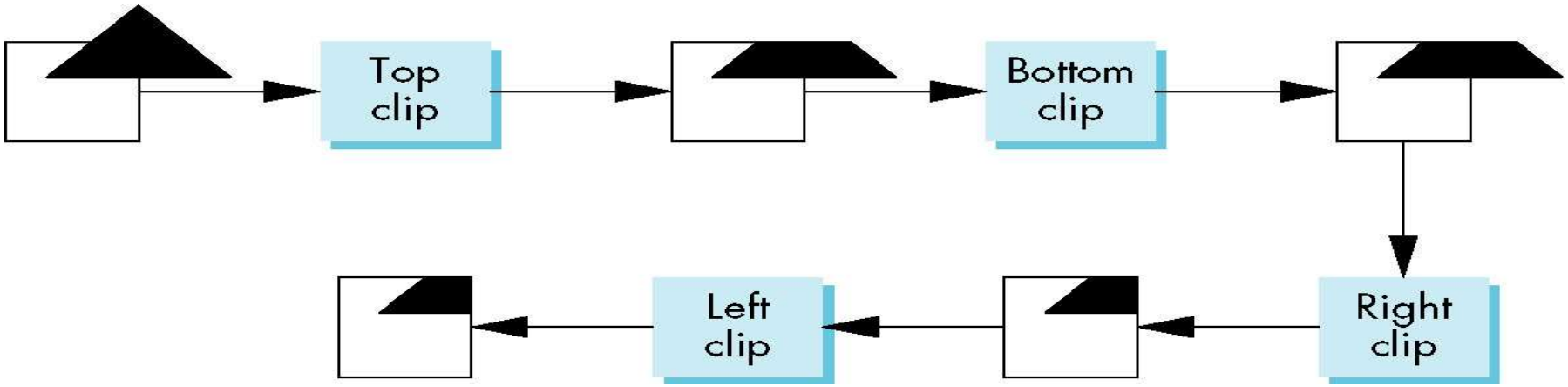
Tessellation and Convexity



- One strategy is to replace nonconvex (*concave*) polygons with a set of triangular polygons (a *tessellation*)
- Also makes fill easier (we will study later)
- Tessellation code in GLU library, the best is Constrained Delauna



Pipeline Clipping of Polygons

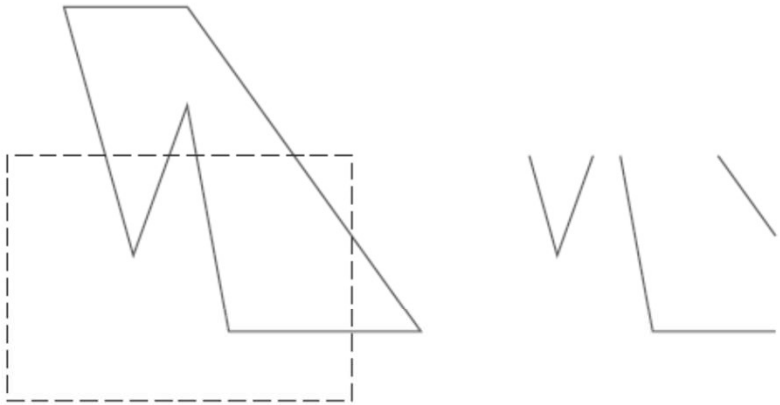


- Three dimensions: add front and back clippers
- Strategy used in SGI Geometry Engine
- Small increase in latency

Polygon Fill-Area Clipping

- To **clip a polygon fill area** -- ***cannot apply a line-clipping method*** to the individual polygon edges directly - would not produce a closed polyline.
- **Line clipper** would often ***produce a disjoint set of lines*** with no complete information about to form a closed boundary around the clipped fill area.
- Figure 19 illustrates a possible output from **a line-clipping procedure applied to the edges of a polygon fill area.**
    - we ***require a procedure that will output one or more closed polylines*** for the boundaries of the clipped fill area,
    - so that the polygons can be scan-converted to fill the interiors with the assigned color or pattern, as in Figure 20.

Polygon Fill-Area Clipping



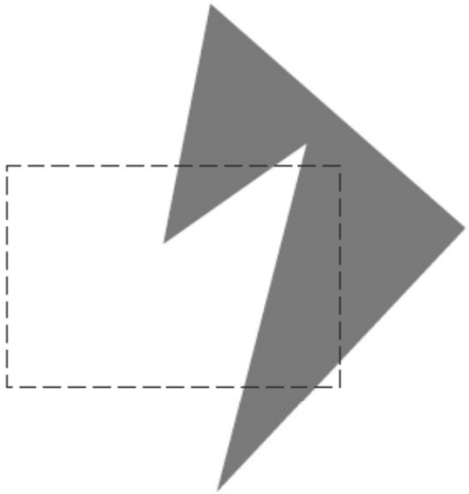
Before Clipping

(a)

After Clipping

(b)

FIGURE 19
A line-clipping algorithm applied to the line segments of the polygon boundary in (a) generates the unconnected set of lines in (b).



Before Clipping

(a)

After Clipping

(b)

FIGURE 20
Display of a correctly clipped polygon fill area.

Polygon Fill-Area Clipping

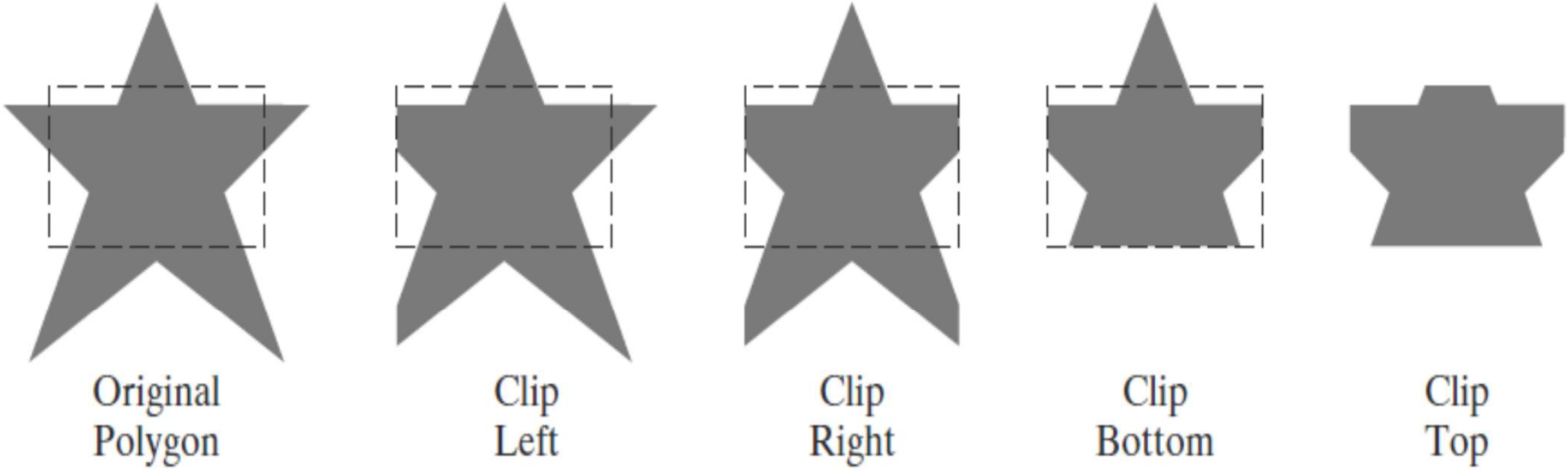


FIGURE 21
Processing a polygon fill area against successive clipping-window boundaries.

Polygon Fill-Area Clipping

- we ***need to maintain a fill area*** as an entity as it is processed through the clipping stages.
- To ***clip a polygon fill area*** --- by ***determining the new shape for the polygon as each clipping-window edge is processed***, as demonstrated in Figure 21.
- the interior fill for the polygon would not be applied until the final clipped border had been determined.

Polygon Fill-Area Clipping

- If **we cannot identify a fill area as being completely inside or completely outside the clipping window,**
    - then ***need to locate the polygon intersection positions*** with the clipping boundaries.
- One way **to implement convex-polygon clipping** - create a new vertex list at each clipping boundary, and then pass this new vertex list to the next boundary clipper.
    - The output of the final clipping stage is the vertex list for the clipped polygon (Figure 23).
- **For concave-polygon clipping**
    - modify this basic approach so that multiple vertex lists could be generated.

Polygon Fill-Area Clipping

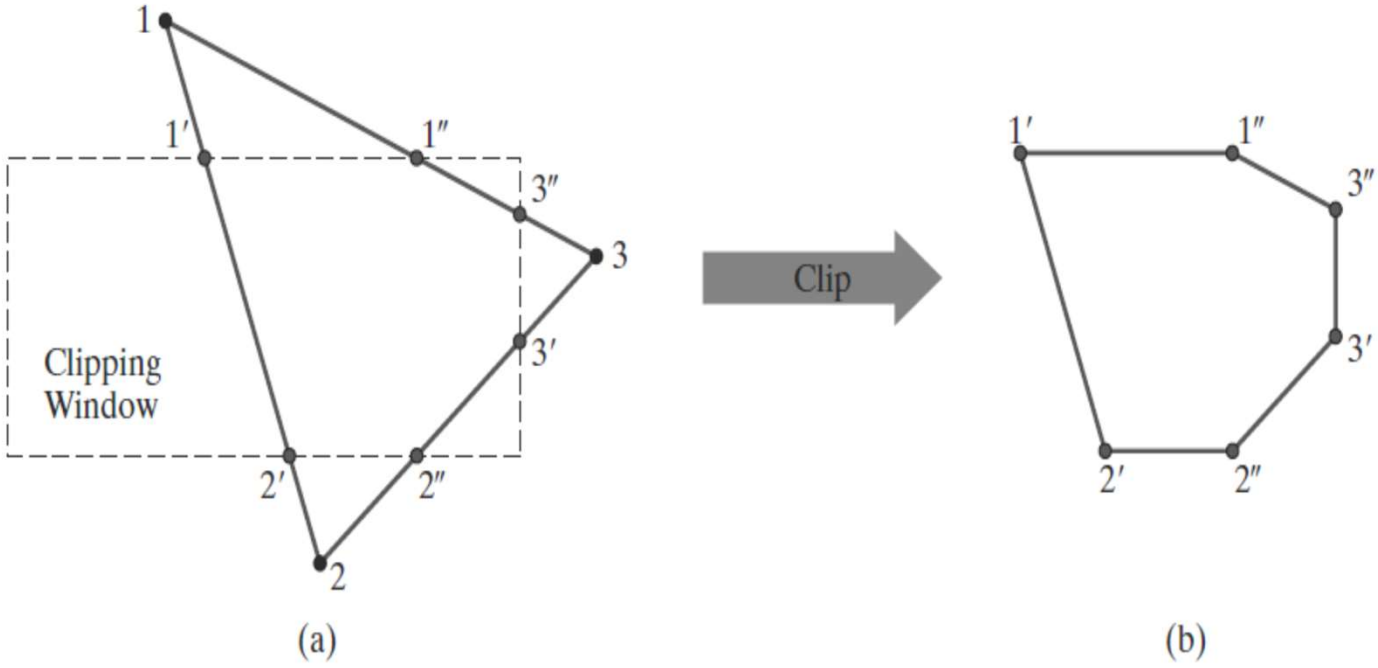
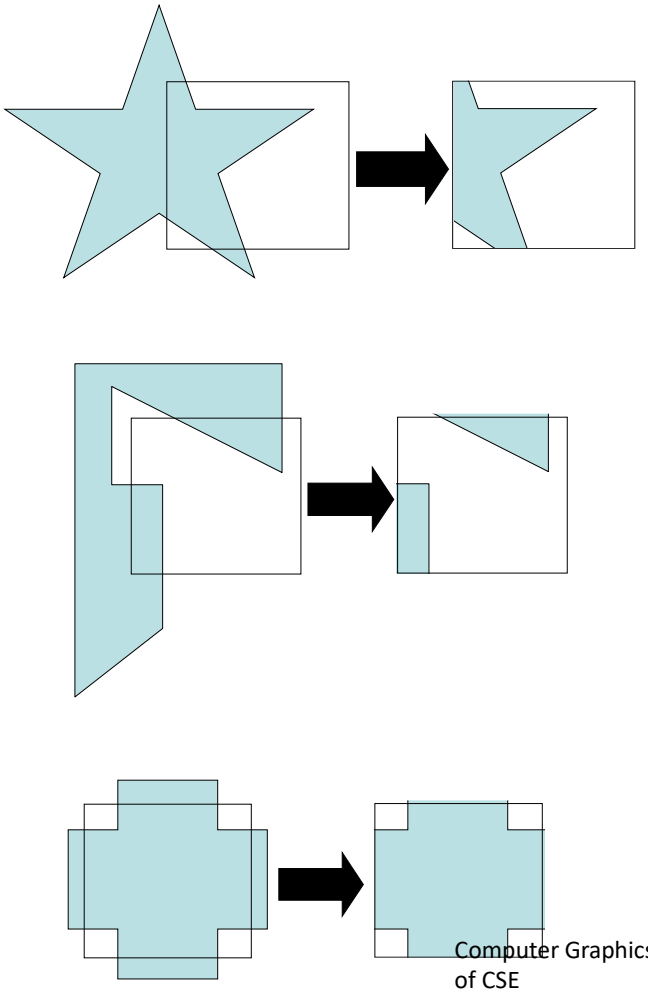


FIGURE 23
A convex-polygon fill area (a), defined with the vertex list {1, 2, 3}, is clipped to produce the fill-area shape shown in (b), which is defined with the output vertex list {1′, 2′, 2″, 3′, 3″, 1″}.

Polygon Fill-Area Clipping

- To **clip a polygon fill area** -- ***cannot apply a line-clipping method*** to the individual polygon edges directly
  - would not produce a closed polyline.
- **Line clipper** would often ***produce a disjoint set of lines*** with no complete information about to form a closed boundary around the clipped fill area.
- Figure 19 illustrates a possible output from a line-clipping procedure applied to the edges of a polygon fill area.
  - we require a procedure that will output one or more closed polylines for the boundaries of the clipped fill area,
  - so that the polygons can be scan-converted to fill the interiors with the assigned color or pattern, as in Figure 20.

Area Clipping



- Similarly to lines, areas must be clipped to a window boundary
- Consideration must be taken as to which portions of the area must be clipped

Polygon Fill-Area Clipping - Sutherland and Hodgman

- An efficient method for clipping a convex-polygon fill area, developed by **Sutherland and Hodgman**
    - send the polygon vertices through each clipping stage so that a single clipped vertex can be immediately passed to the next stage.
    - ***eliminates*** the need for an output set of vertices at each clipping stage, and it ***allows*** the boundary-clipping routines to be implemented in parallel.
    - The ***final output*** is a list of vertices that describe the edges of the clipped polygon fill area.

Polygon Fill-Area Clipping - Sutherland and Hodgman

- Sutherland-Hodgman algorithm produces only one list of output vertices, ***it cannot correctly generate the two output polygons*** in Figure 20(b) that are the result of clipping the concave polygon shown in Figure 20(a).
- However, more processing steps can be added to the algorithm to allow it to produce multiple output vertex lists, so that general concave-polygon clipping could be accommodated.
- **Basic Sutherland-Hodgman algorithm is able to process concave polygons** when the clipped fill area can be described with a single vertex list.

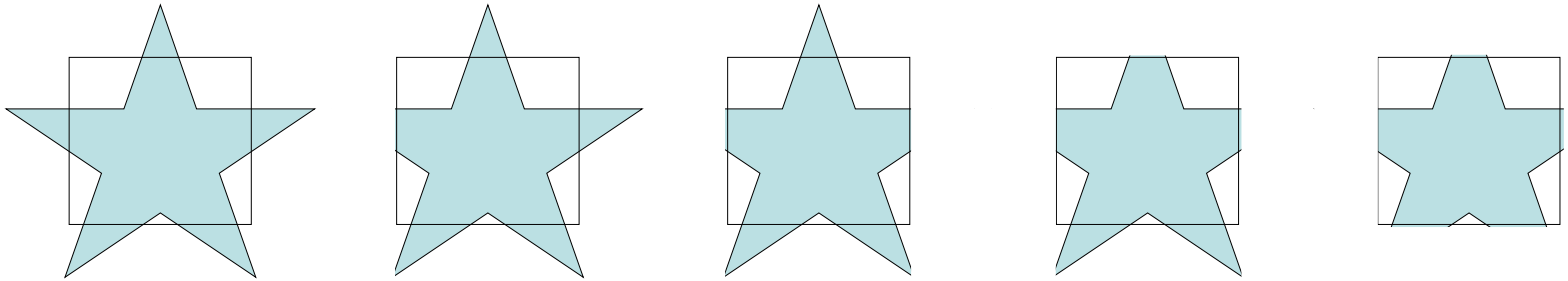
Polygon Fill-Area Clipping - Sutherland and Hodgman

- ***The general strategy in this algorithm***
  - send the pair of endpoints for each successive polygon line segment through the series of clippers (left, right, bottom, and top).
  - As soon as a clipper completes the processing of one pair of vertices, the clipped coordinate values, if any, for that edge are sent to the next clipper.
  - Then the first clipper processes the next pair of endpoints.
  - the individual boundary clippers can be operating in parallel.

Sutherland-Hodgman Area Clipping Algorithm

- A technique for clipping areas developed by Sutherland & Hodgman
- Put simply the polygon is clipped by comparing it against each boundary in turn

Sutherland turns up again. This time with Gary Hodgman with whom he worked at the first ever graphics company
Evans & Sutherland



Sutherland-Hodgman Area Clipping Algorithm (cont…)

To clip an area against an individual boundary:

- Consider each vertex in turn against the boundary
- Vertices inside the boundary are saved for clipping against the next boundary
- Vertices outside the boundary are clipped
- If we proceed from a point inside the boundary to one outside, the intersection of the line with the boundary is saved
- If we cross from the outside to the inside intersection point and the vertex are saved

Polygon Fill-Area Clipping - Sutherland and Hodgman

- There are **four possible cases** that need to be considered when processing a polygon edge against one of the clipping boundaries.
    - One possibility is that the ***first edge endpoint is outside the clipping boundary*** and the ***second endpoint is inside.***
    - ***both endpoints*** could be ***inside*** this clipping boundary.
    - ***first endpoint is inside*** the clipping boundary and the ***second endpoint is outside***.
    - ***both endpoints*** could be ***outside*** the clipping boundary.

Polygon Fill-Area Clipping - Sutherland and Hodgman

To facilitate the passing of vertices from one clipping stage to the next, the output from each clipper can be formulated as shown in Figure 24.

As each successive pair of endpoints is passed to one of the four clippers, an output is generated for the next clipper according to the results of the following tests:

1. If the **first input vertex is outside** this clipping-window border and the second vertex is inside, both the intersection point of the polygon edge with the window border and the second vertex are sent to the next clipper.

2. If **both input vertices are inside** this clipping-window border, only the second vertex is sent to the next clipper.

3. If the **first vertex is inside** this clipping-window border and the **second vertex is outside**, only the polygon edge-intersection position with the clipping-window border is sent to the next clipper.

4. If **both input vertices are outside** this clipping-window border, no vertices are sent to the next clipper.

Polygon Fill-Area Clipping - Sutherland and Hodgman

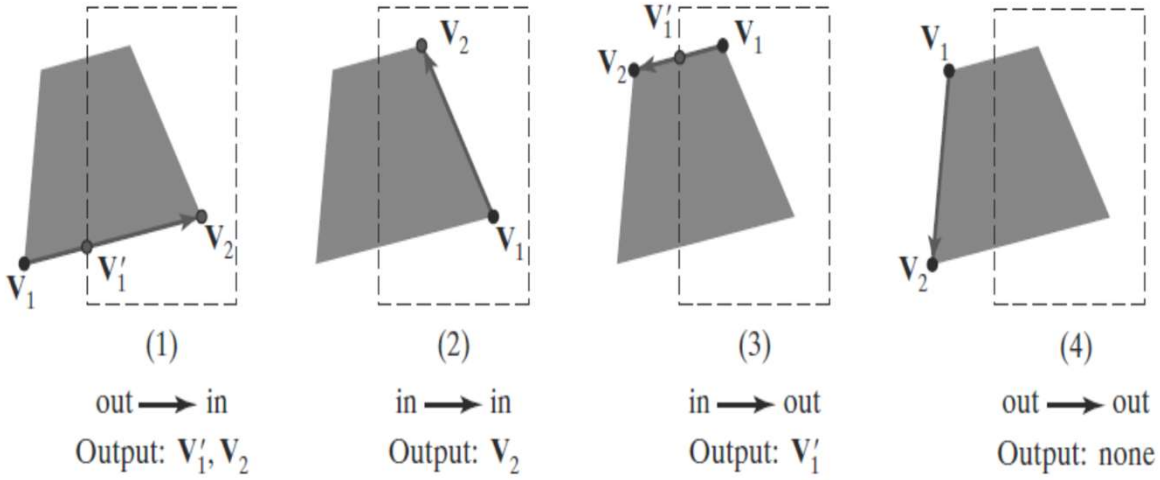


FIGURE 24
The four possible outputs generated by the left clipper, depending on the position of a pair of endpoints relative to the left boundary of the clipping window.

Sutherland and Hodgman -- Example

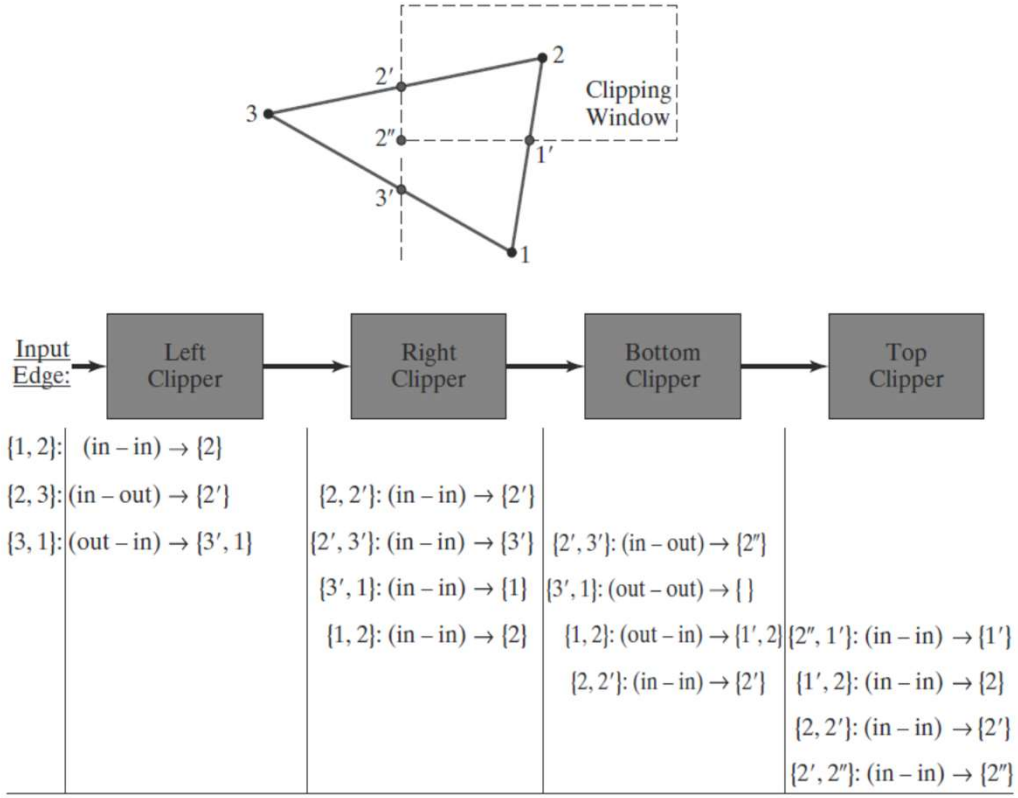


FIGURE 25
Processing a set of polygon vertices, {1, 2, 3}, through the boundary clippers using the Sutherland-Hodgman algorithm. The final set of clipped vertices is {1′, 2, 2′, 2″}.

Polygon Fill-Area Clipping - Sutherland and Hodgman

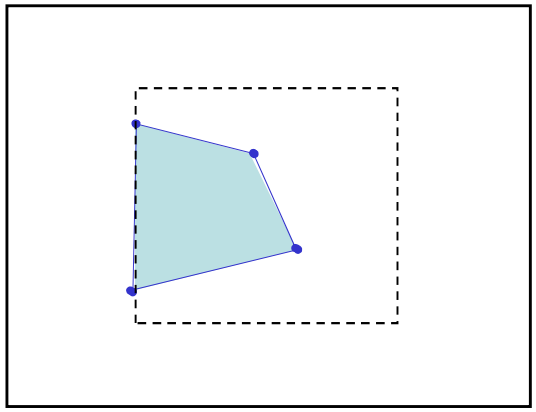
- Figure 25 provides an example of the Sutherland-Hodgman polygon clipping algorithm for a fill area defined with the vertex set

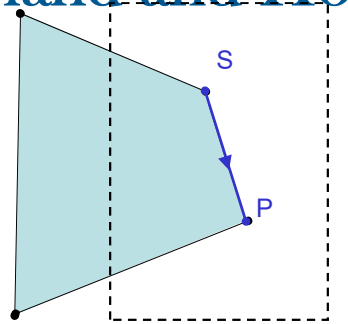
  $\{1, 2, 3\}$.

- As soon as a clipper receives a pair of endpoints, it determines the appropriate output using the tests illustrated in Figure 24.
- These outputs are passed in succession from the left clipper to the right, bottom, and top clippers. The output from the top clipper is the set of vertices defining the clipped fill area. For this example, the output vertex list is $\{1', 2, 2', 2''\}$.

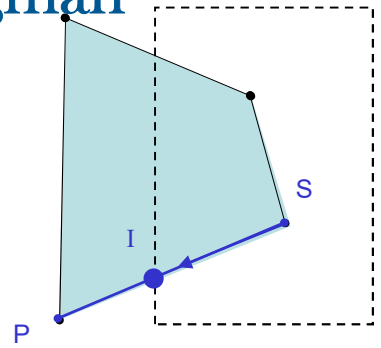
Polygon Fill-Area Clipping - Sutherland and Hodgman

- Each example (Figs Right) shows the point being processed (P) and the previous point (S)
- Saved points define area clipped to the boundary in question.
- Fig (below) – final result

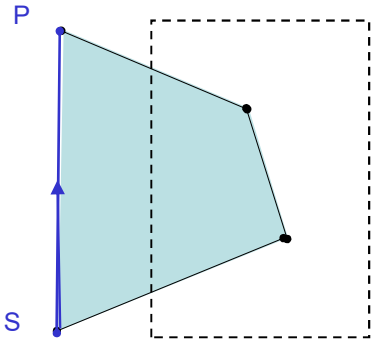




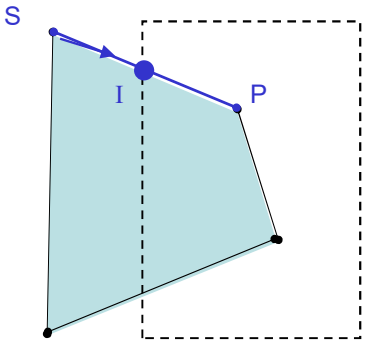
Save Point P



Save Point I



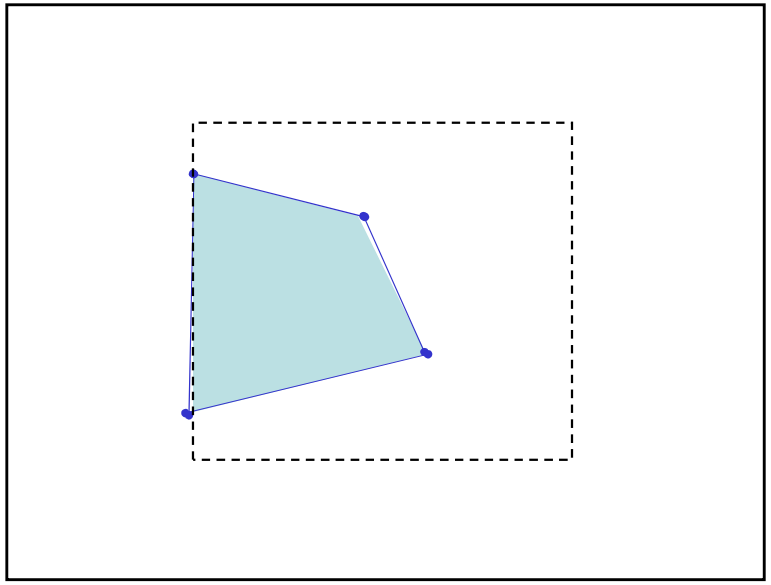
No Points Saved



Save Points I & P

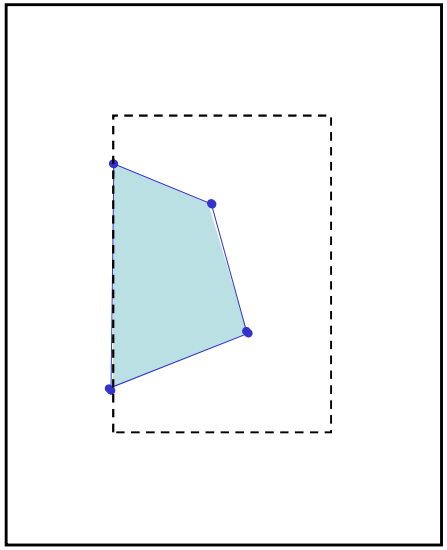
Polygon Fill-Area Clipping - Sutherland and Hodgman

- Fig (below) – final result of Clipping



Sutherland-Hodgman Example(Contd..)

- Fig (below) – final result of Clipping

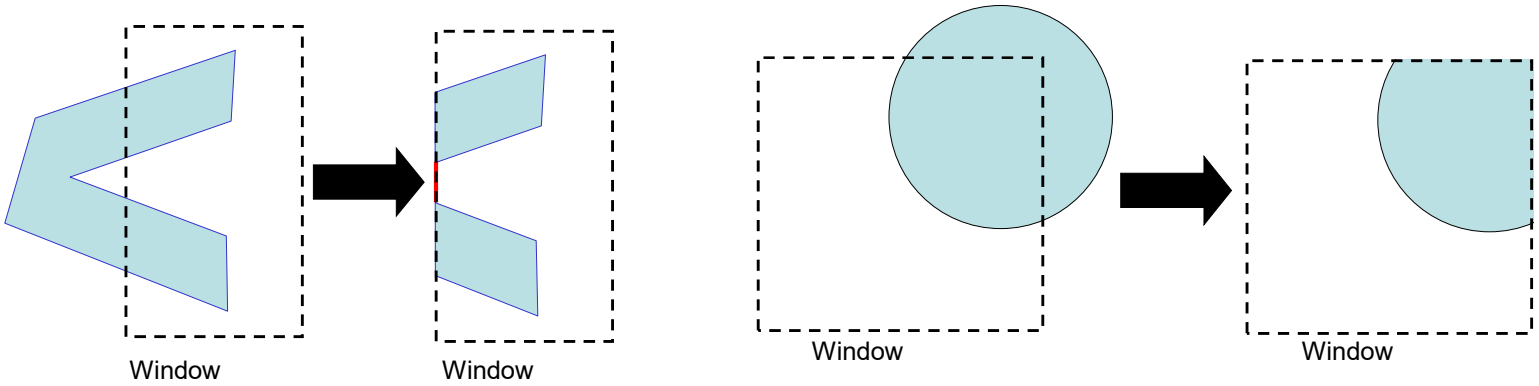


Other Area Clipping Concerns

- A sequential implementation of the Sutherland-Hodgman polygon-clipping algorithm is demonstrated in the set of procedures.
- An input set of vertices is converted to an output vertex list by clipping it against the four edges of the axis-aligned rectangular clipping region.

Other Area Clipping Concerns

Clipping concave areas can be a little more tricky as often superfluous lines must be removed



Clipping curves requires more work

- For circles we must find the two intersection points on the window boundary

Summary

- Objects within a scene must be clipped to display the scene in a window
- Because there are can be so many objects clipping must be extremely efficient
- The Cohen-Sutherland and Liang Barsky algorithms can be used for line clipping
- The Sutherland-Hodgman algorithm can be used for area clipping