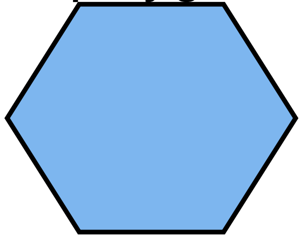
Computer graphics
Polygon Filling

Polygon Filling

Types of filling

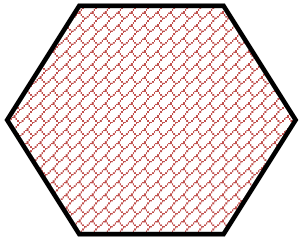
- **Solid-fill**

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.

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

The polygon can be displayed by drawing a line between **(x_1, y_1)**, and **(x_2, y_2)**, then a line between **(x_2, y_2)**, and **(x_3, y_3)**, and so on until the end vertex. In order to close up the polygon, a line between **(x_n, y_n)**, and **(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.

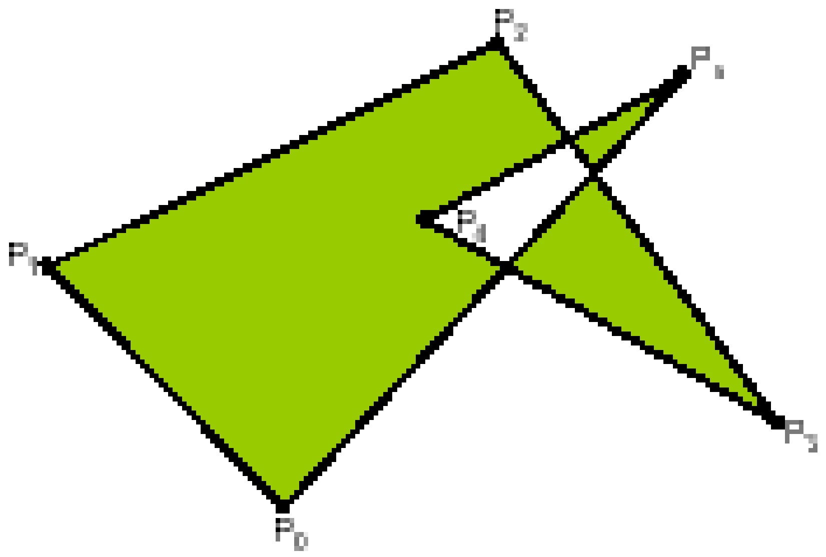
Polygon Representation

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.

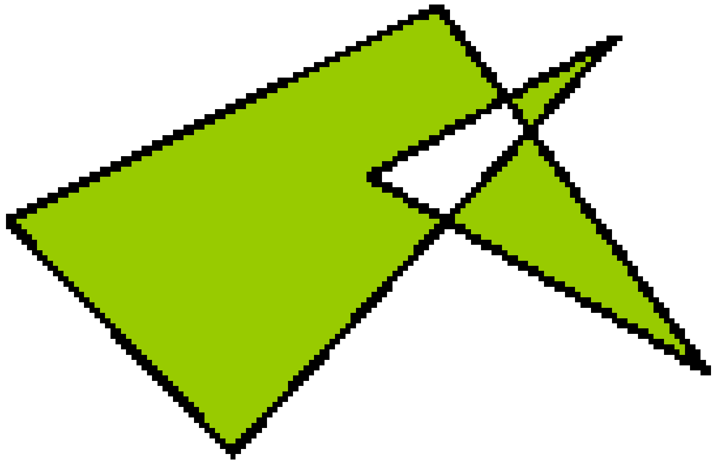
Filling General Polygons

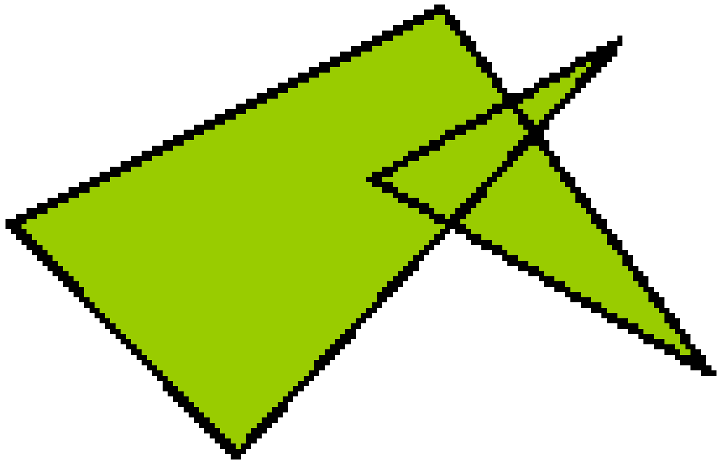
- Specifying the interior
  - Must be able to determine which points are inside the polygon
    - Need a fill rule



Filling General Polygons

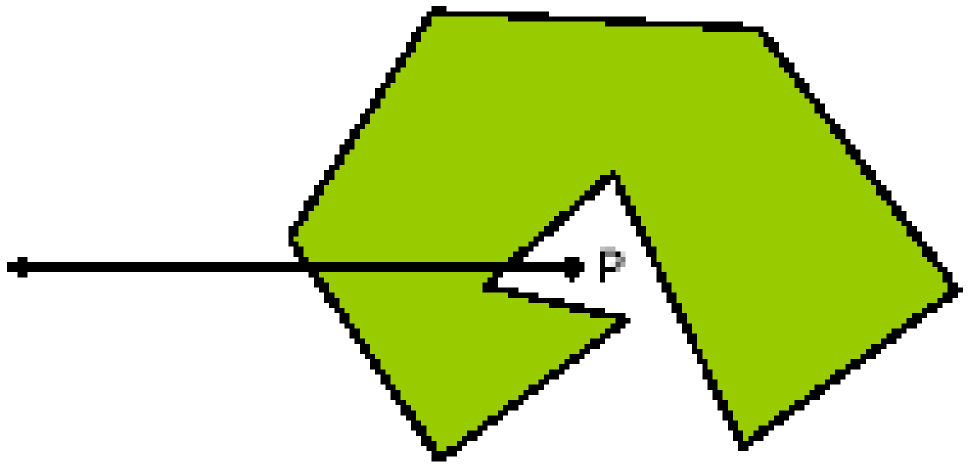
- Specifying the interior
  - There are two commonly used fill rules
    - Even-odd parity rule
    - Non-zero winding rule


Filled using even-odd parity rule


Filled using none-zero winding rule

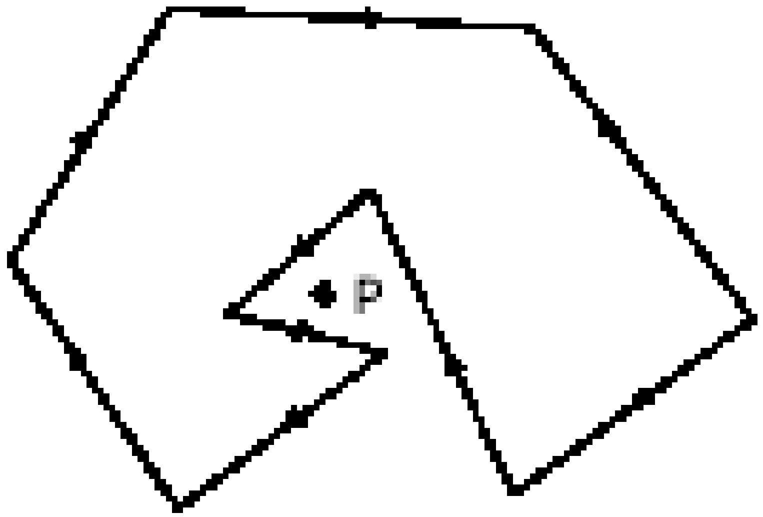
Even-odd Parity Rule

- To determine if a point P is inside or outside
  - Draw a line from P to infinity
  - Count the number of times the line crosses an edge
    - If the number of crossing is odd, the point is inside
    - If the number of crossing is even, the point is outside



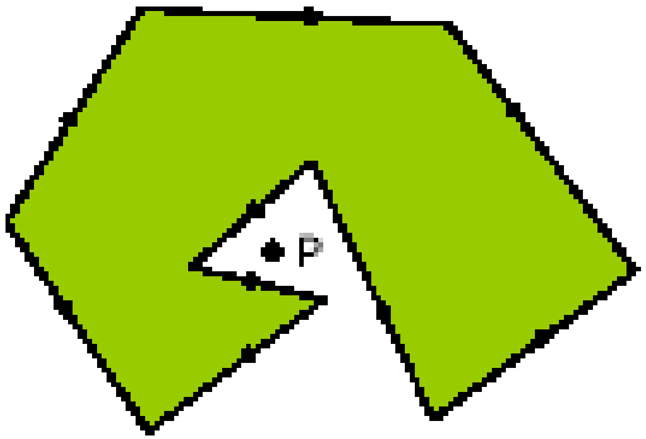
Non-zero Winding Number Rule

- The outline of the shape must be directed
  - The line segments must have a consistent direction so that they formed a continuous, closed path



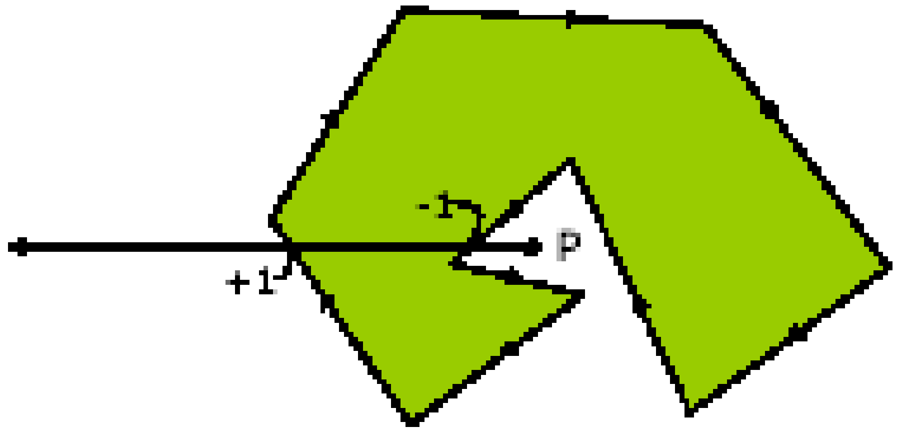
Non-zero Winding Number Rule

- To determine if a points is inside or outside
  - Determine the winding number (i.e. the number of times the edge winds around the point in either a clockwise or counterclockwise direction)
    - Points are outside if the winding number is zero
    - Point are inside if the winding number is not zero



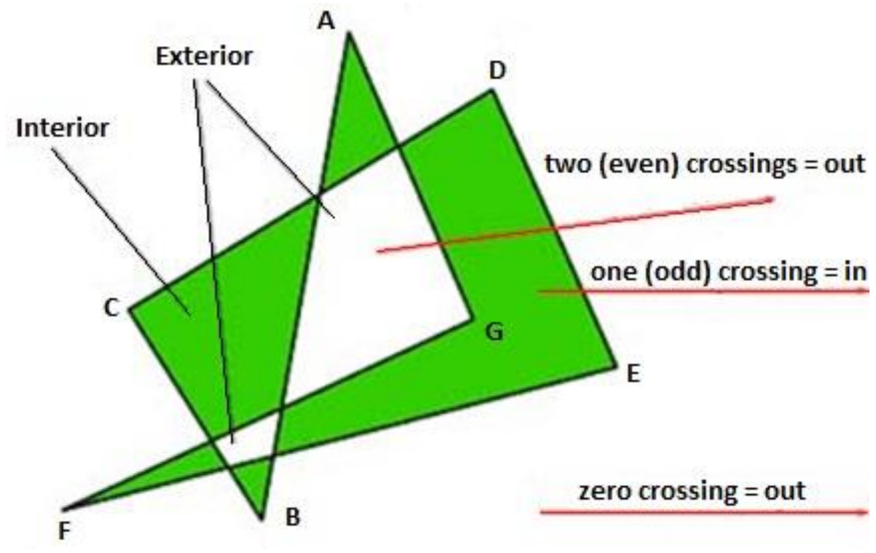
Non-zero Winding Number Rule

- To determine the winding number at a point P
  - Initialize the winding number to zero and draw a line (e.g. horizontal) from P to infinity
  - If the line crosses an edge directed bottom to up
    - Add 1 to the winding number
  - If the line crosses an edge directed top to bottom
    - Subtract 1 from the winding number

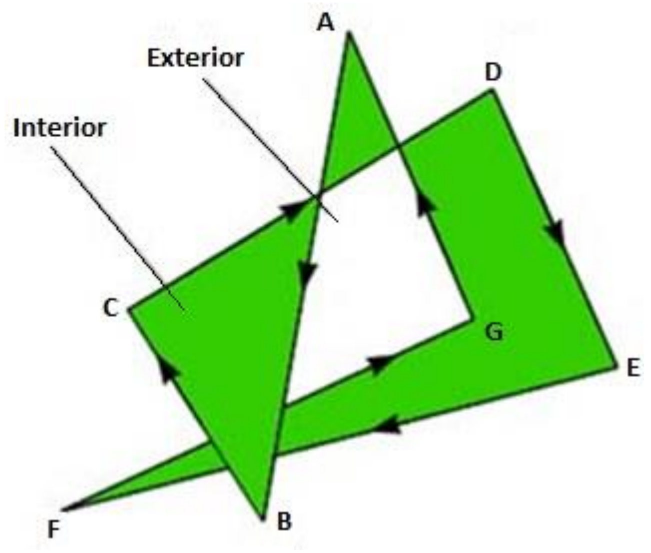


Comparison between Odd Even Rule and Nonzero Winding Rule

- For standard polygons and simple object both rule gives same result but for more complicated shape both rule gives different result.



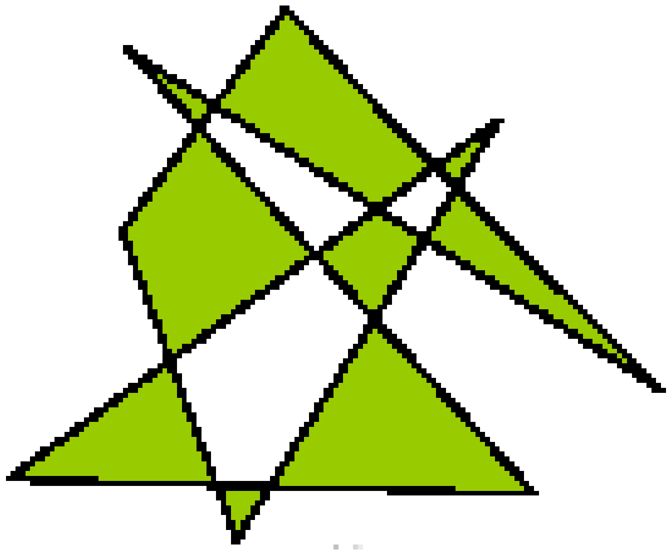
Odd Even Rule

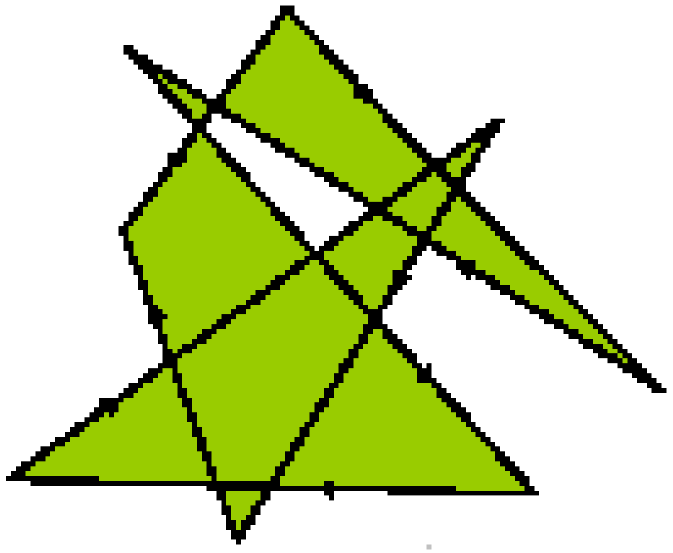


Nonzero Winding Rule

Inside-Outside Tests

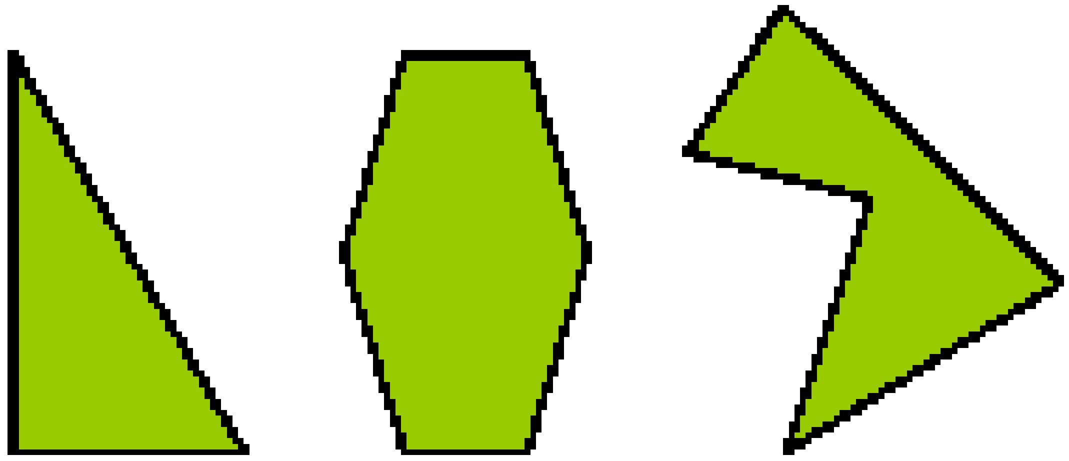
- The non-zero winding number rule and the even-odd parity rule can give different results for general polygons
  - When polygons self intersect
  - When polygons have interior holes


Even-odd parity


Non-zero winding

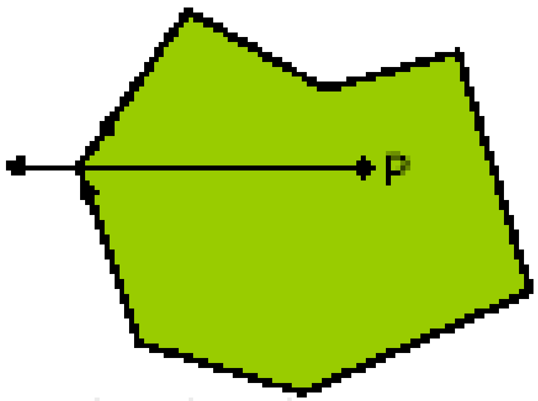
Inside-Outside Tests

- Standard polygons
  - Standard polygons (e.g. triangles, rectangles, octagons) do not self intersect and do not contain holes
  - The non-zero winding number rule and the even-odd parity rule give the same results for standard polygons

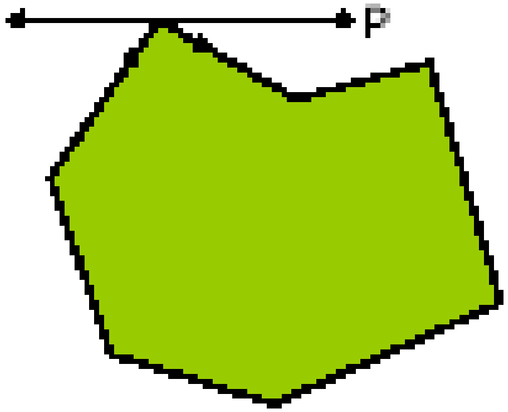


Shared Vertices

- Edges share vertices
  - If the line drawn for the fill rule intersects a vertex, the edge crossing would be counted twice
    - This yields incorrect and inconsistent even-odd parity checks and winding numbers



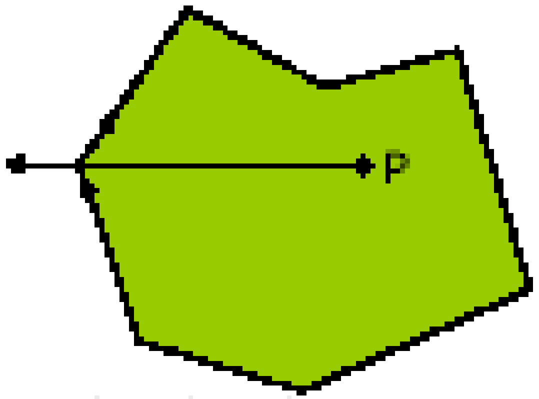
Line pierces the outline
- Should count as one crossing



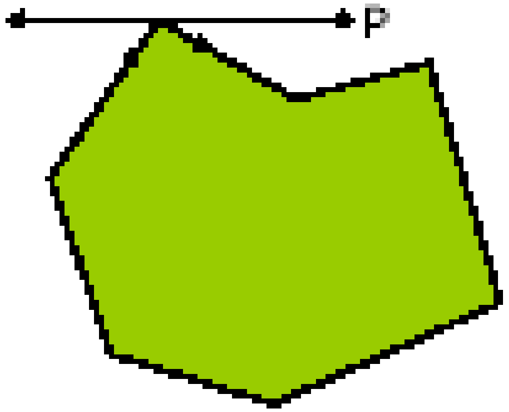
Line grazes the outline
- Should count as no crossings

Dealing with Shared Vertices

1. Check the vertex type (piercing or grazing)
   - If the vertex is between two upwards or two downwards edges, the line pierces the edge
     - Process a single edge crossing
   - If the vertex is between an upwards and a downwards edge, the line grazes the vertex
     - Don't process any edge crossings



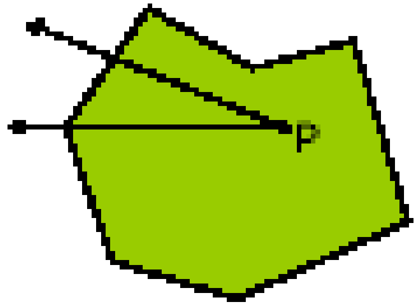
Vertex between two upwards edges
- Process a single crossing



Vertex between upwards and downwards edges
- Process no crossings

Dealing with Shared Vertices

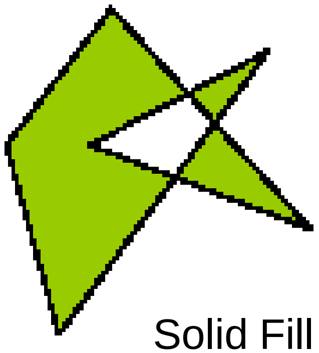
2. Ensure that the line does not intersect a vertex
   - Use a different line if the first line intersects a vertex
     - Could be costly if you have to try several lines



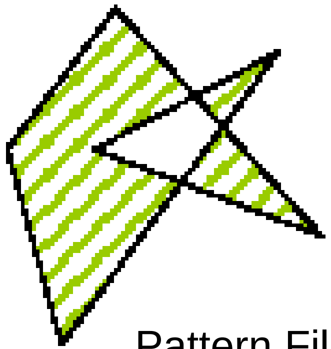
   - If using horizontal scan line for the inside-outside test
     - Preprocess edge vertices to make sure that none of them fall on a scan line
       - Add a small floating point value to each vertex y-position

Fill Options

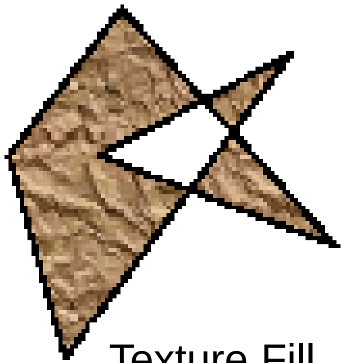
- How to set pixel colors for points inside the shape?



Solid Fill



Pattern Fill



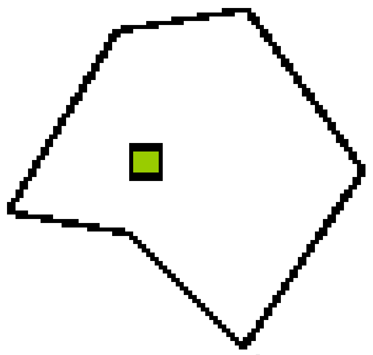
Texture Fill

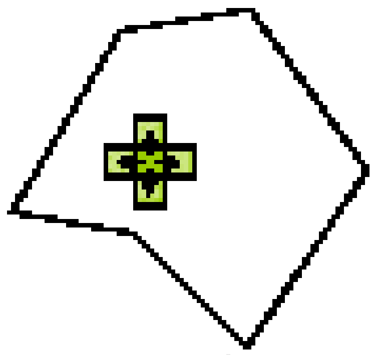
Filled-Area Primitives

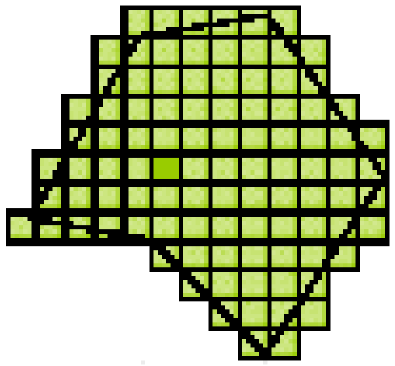
- In practical we often use polygon which are filled with some colour or pattern inside it.
- There are two basic approaches to area filling on raster systems.
  - One way to fill an area is to determine the overlap intervals for scan line that cross the area.
  - Another method is to start from a given interior position and paint outward from this point until we encounter boundary.

Seed Fill

- Approach
  - Select a seed point inside a region
  - Move outwards from the seed point, setting neighboring pixels until the region is filled

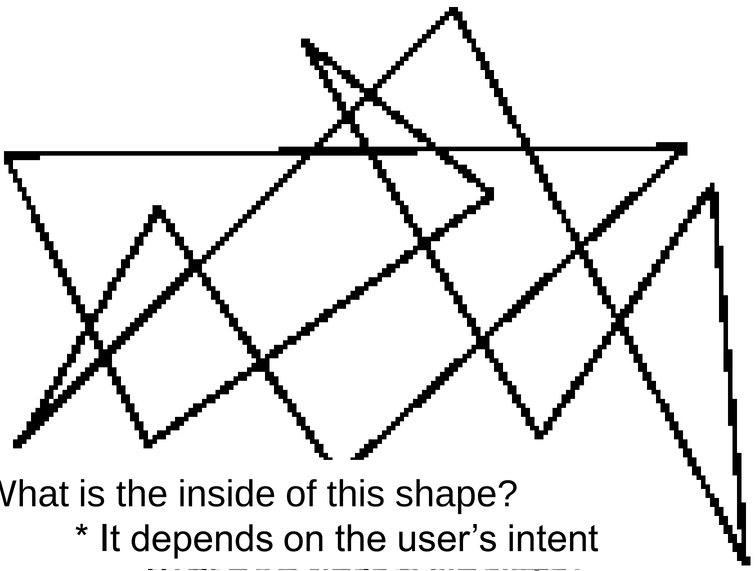

Seed point


Move outwards to neighbors


Stop when the region is filled

Selecting the Seed Point

- Difficult to place the seed point automatically
  - Seed fill works best in an interactive application where the user sets the seed point



What is the inside of this shape?

* It depends on the user’s intent

▪ It depends on the user's intent

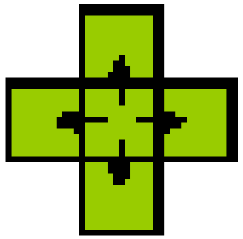
Seed Fill

- Basic algorithm

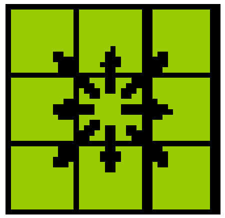
```
select seed pixel
initialize a fill list to contain seed pixel
while (fill list not empty) {
    pixel ← = get next pixel from fill list          setPixel(pixel)
    for (each of the pixel's neighbors) {
        if (neighbor is inside region AND neighbor not set)
    add neighbor to fill list
    }
}
```

Which neighbors should be tested?

- There are two types of 2D regions
  - 4-connected region (test 4 neighbors)
    - Two pixels are 4-connected if they are vertical or horizontal neighbors

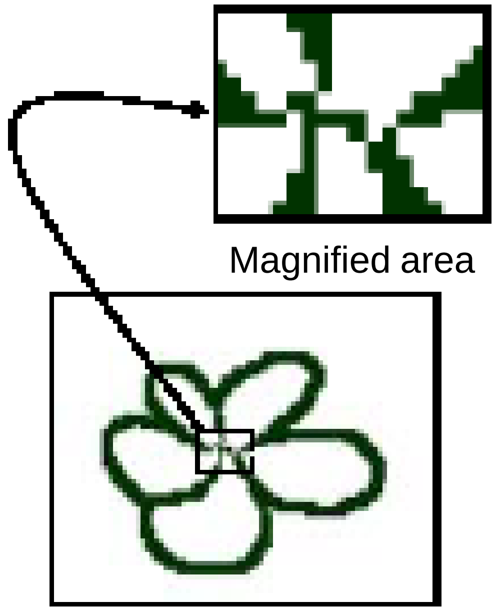


    - 8-connected region (test 8 neighbors)
    - Two pixels are 8-connected if they are vertical, horizontal, or diagonal neighbors



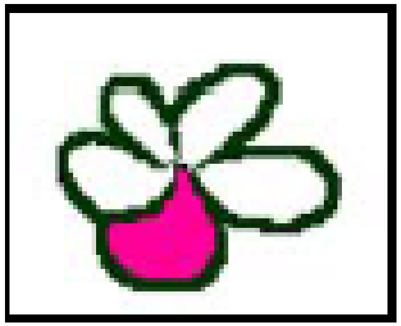
Which neighbors should be tested?

- Using 4-connected and 8-connected neighbors gives different results

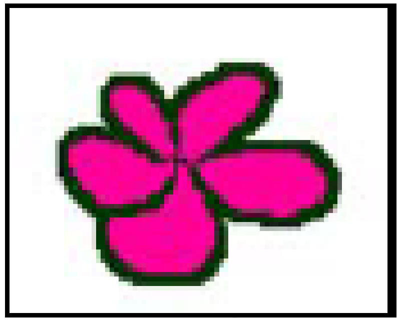


Magnified area

Original boundary



Fill using 4-connected neighbors



Fill using 8-connected neighbors

When is a Neighbor Inside the Region?

- There are two types of tests, resulting in two filling approaches
  - Boundary fill
  - Flood fill

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.
- A boundary-fill procedure accepts as input the coordinate of the interior **point (x, y)**, a **fill color**, and a **boundary color**.

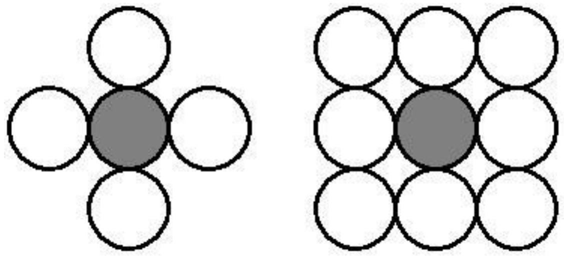
Boundary Fill Algorithm

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**.

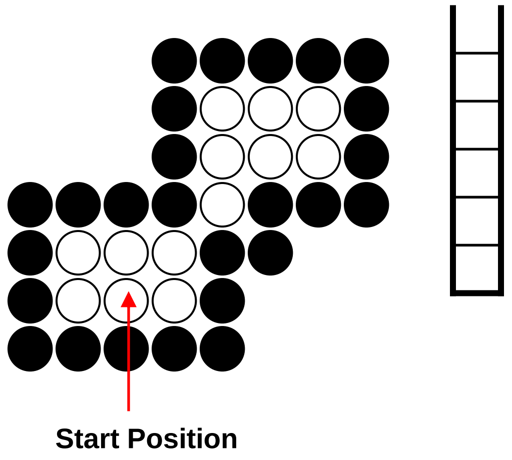
Boundary Fill Algorithm

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.



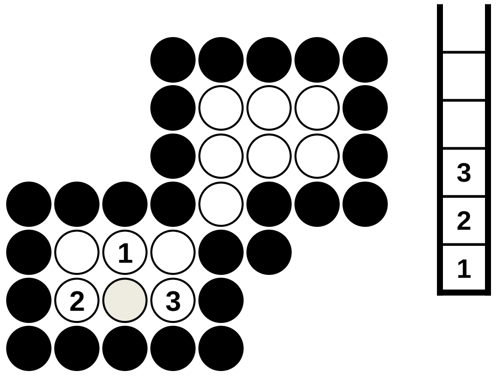
Boundary Fill Algorithm

4-connected (Example)



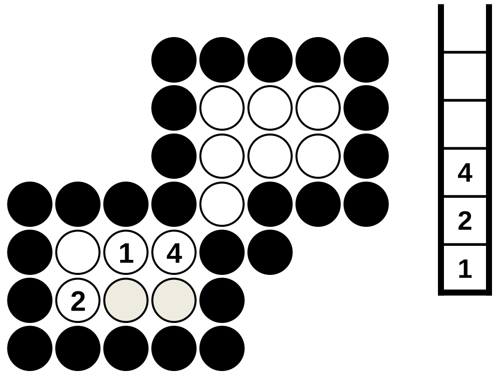
Boundary Fill Algorithm

4-connected (Example)



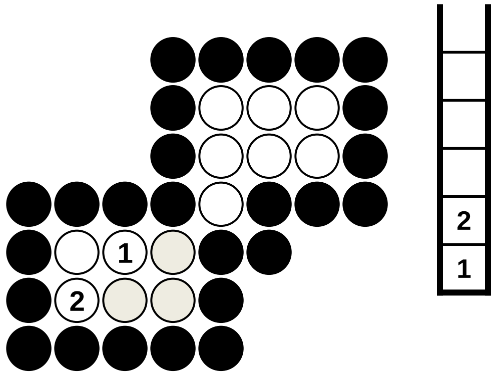
Boundary Fill Algorithm

4-connected (Example)



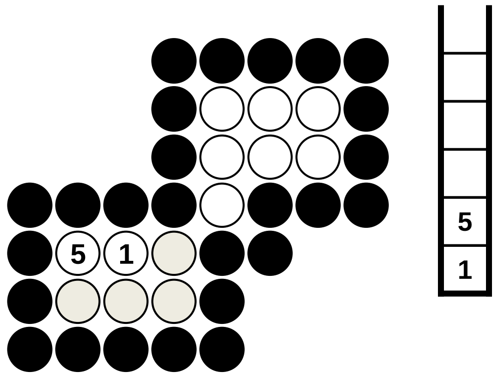
Boundary Fill Algorithm

4-connected (Example)



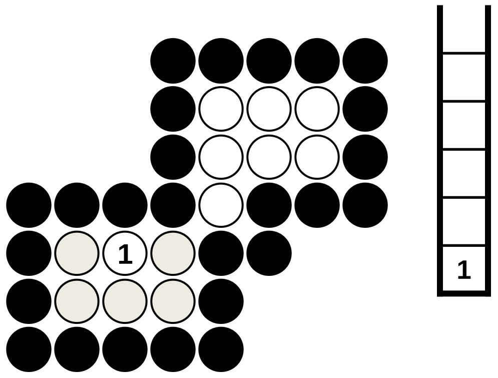
Boundary Fill Algorithm

4-connected (Example)



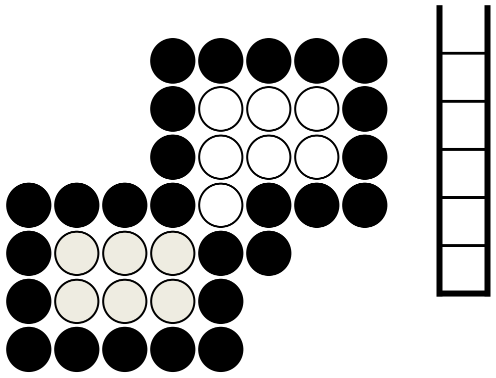
Boundary Fill Algorithm

4-connected (Example)



Boundary Fill Algorithm

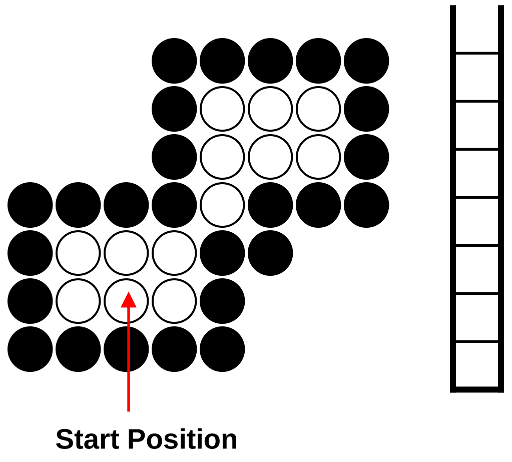
4-connected (Example)



```
void boundaryFill4(int x, int y, int fill_color,int boundary_color)
{
    if(getpixel(x, y) != boundary_color &&
       getpixel(x, y) != fill_color)
    {
        putpixel(x, y, fill_color);
        boundaryFill4(x + 1, y, fill_color, boundary_color);
        boundaryFill4(x, y + 1, fill_color, boundary_color);
        boundaryFill4(x - 1, y, fill_color, boundary_color);
        boundaryFill4(x, y - 1, fill_color, boundary_color);
    }
}
```

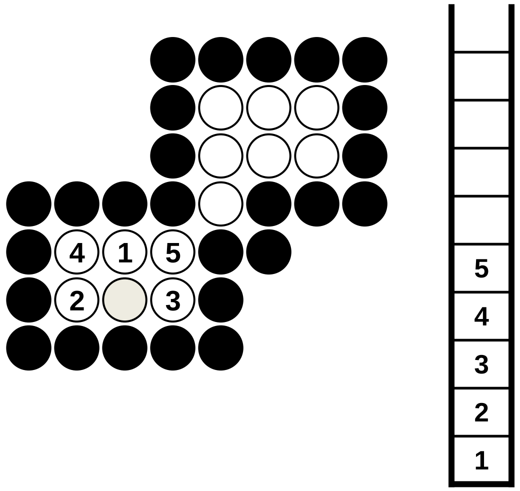
Boundary Fill Algorithm

8-connected (Example)



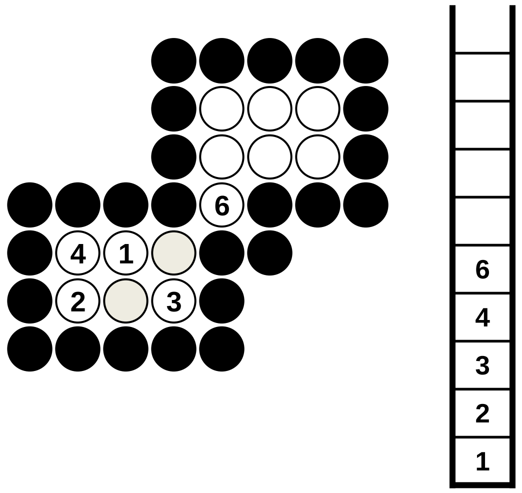
Boundary Fill Algorithm

8-connected (Example)



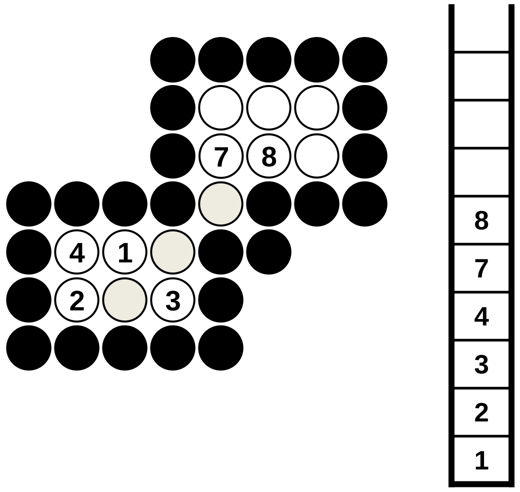
Boundary Fill Algorithm

8-connected (Example)



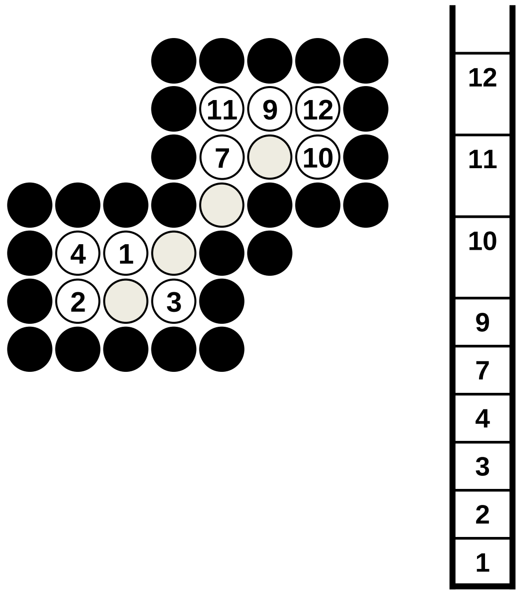
Boundary Fill Algorithm

8-connected (Example)



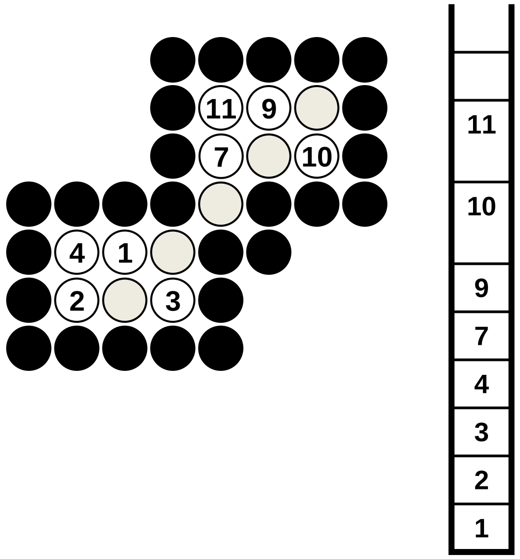
Boundary Fill Algorithm

8-connected (Example)



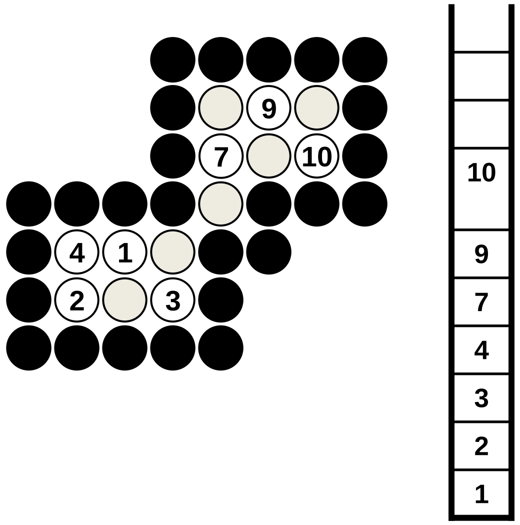
Boundary Fill Algorithm

8-connected (Example)



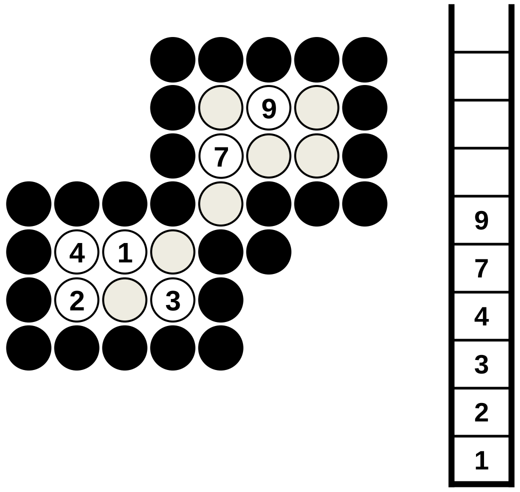
Boundary Fill Algorithm

8-connected (Example)



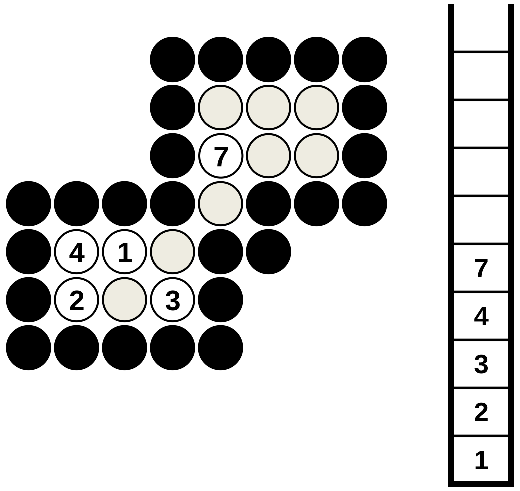
Boundary Fill Algorithm

8-connected (Example)



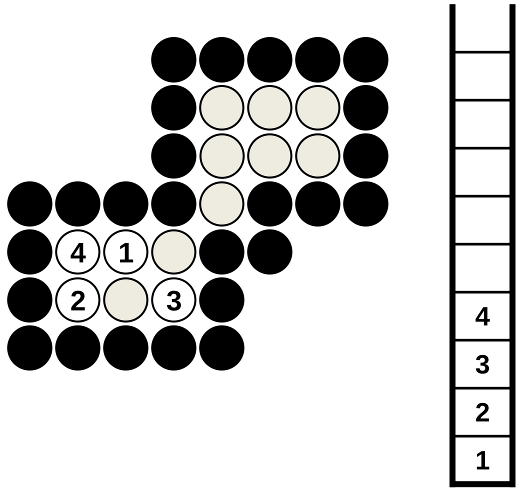
Boundary Fill Algorithm

8-connected (Example)



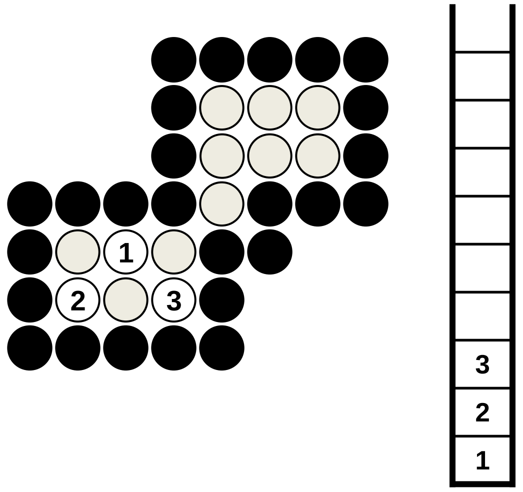
Boundary Fill Algorithm

8-connected (Example)



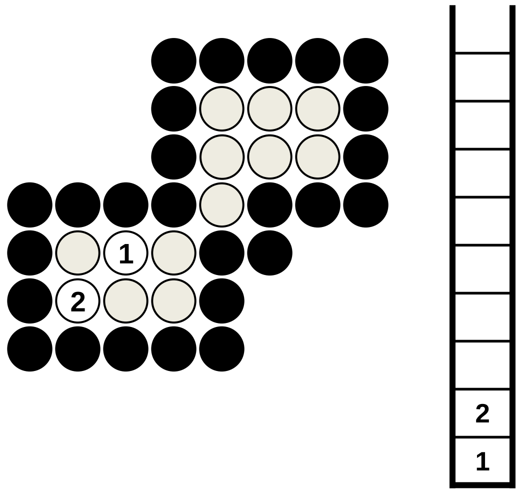
Boundary Fill Algorithm

8-connected (Example)



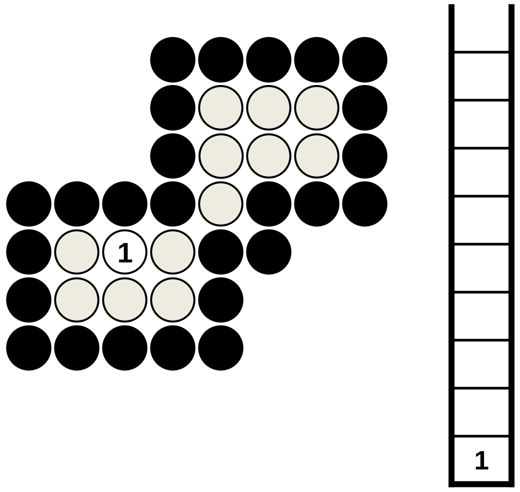
Boundary Fill Algorithm

8-connected (Example)



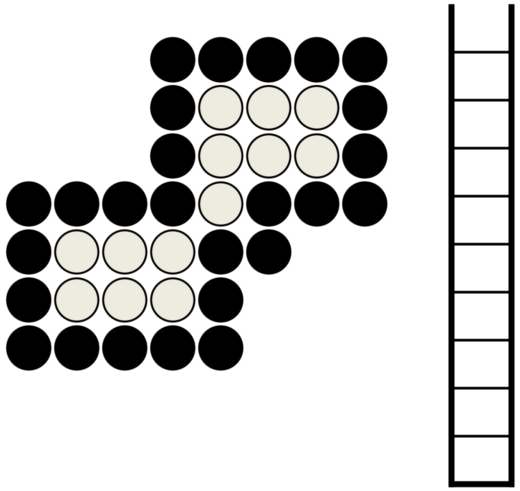
Boundary Fill Algorithm

8-connected (Example)



Boundary Fill Algorithm

8-connected (Example)



```
void boundaryFill8(int x, int y, int fill_color,int boundary_color)
{
    if(getpixel(x, y) != boundary_color &&
       getpixel(x, y) != fill_color)
    {
        putpixel(x, y, fill_color);
        boundaryFill8(x + 1, y, fill_color, boundary_color);
        boundaryFill8(x, y + 1, fill_color, boundary_color);
        boundaryFill8(x - 1, y, fill_color, boundary_color);
        boundaryFill8(x, y - 1, fill_color, boundary_color);
        boundaryFill8(x - 1, y - 1, fill_color, boundary_color);
        boundaryFill8(x - 1, y + 1, fill_color, boundary_color);
        boundaryFill8(x + 1, y - 1, fill_color, boundary_color);
        boundaryFill8(x + 1, y + 1, fill_color, boundary_color);
    }
}
```

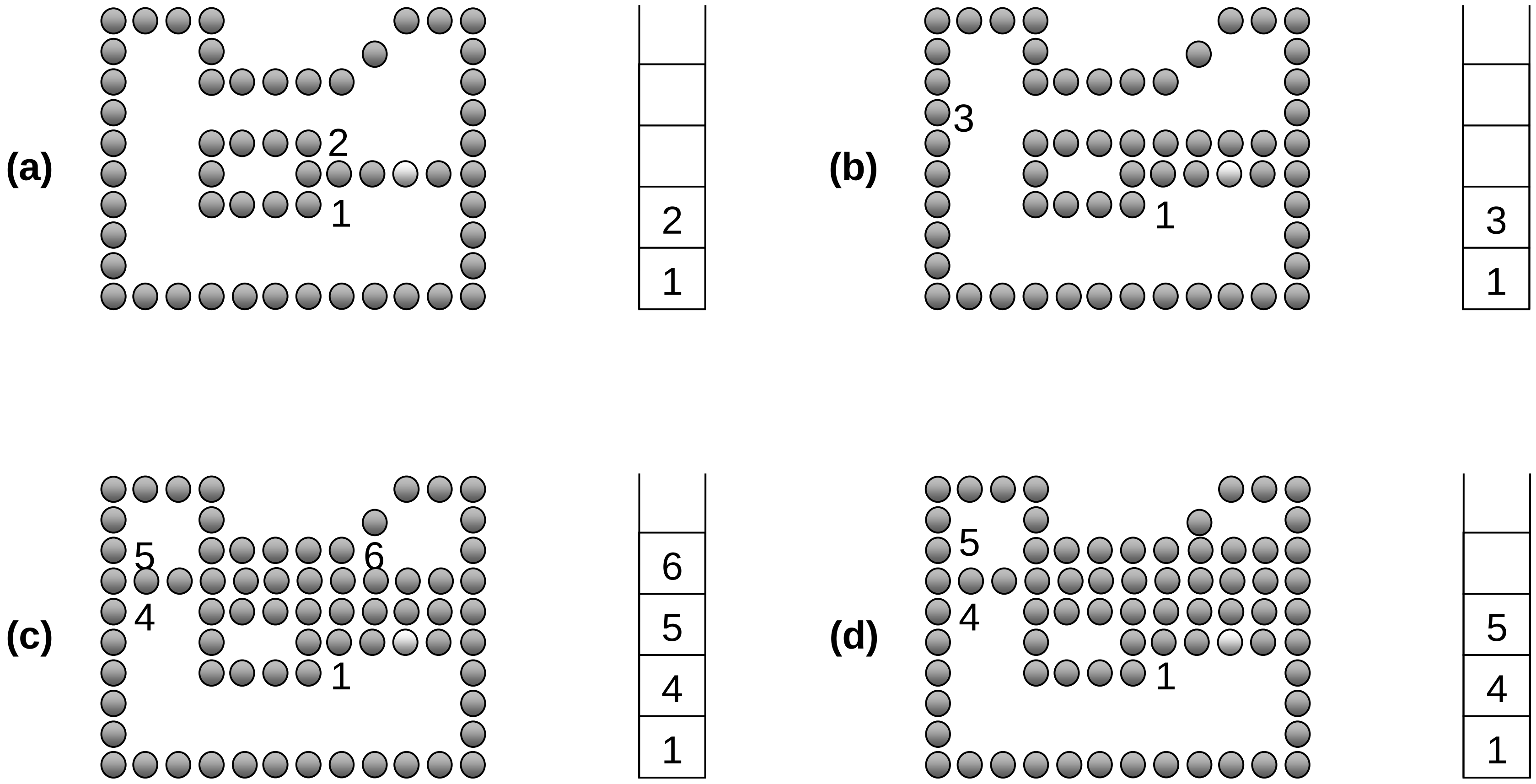
Problem of Staking and Efficient Method

- Same procedure can be modified according to 8 connected region algorithm by including four additional statements to test diagonal positions.
- This procedure requires considerable stacking of neighbouring points more, efficient methods are generally employed.
- Efficient method fill horizontal pixel spans across scan lines, instead of proceeding to 4 connected or 8 connected neighbouring points.
- Then we need only stack a beginning position for each horizontal pixel span, instead of stacking all unprocessed neighbouring positions around the current position.
- Starting from the initial interior point with this method, we first fill in the contiguous span of pixels on this starting scan line.

Contd.

- Then we locate and stack starting positions for spans on the adjacent scan lines.
- Spans are defined as the contiguous horizontal string of positions bounded by pixels displayed in the area border colour.
- At each subsequent step, we unstack the next start position and repeat the process.
- For e.g.

Contd.



Boundary Fill Algorithm

Since the previous procedure requires considerable stacking of neighboring pixels, more **efficient methods** are generally employed.

These methods (**Span Flood-Fill**) **fill horizontal pixel spans across scan lines**, instead of proceeding to 4-connected or 8-connected neighboring pixels.

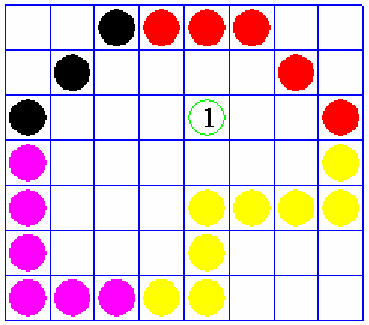
Then we need only stack a beginning position for each horizontal pixel spans, instead of stacking all unprocessed neighboring positions around the current position.

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 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.

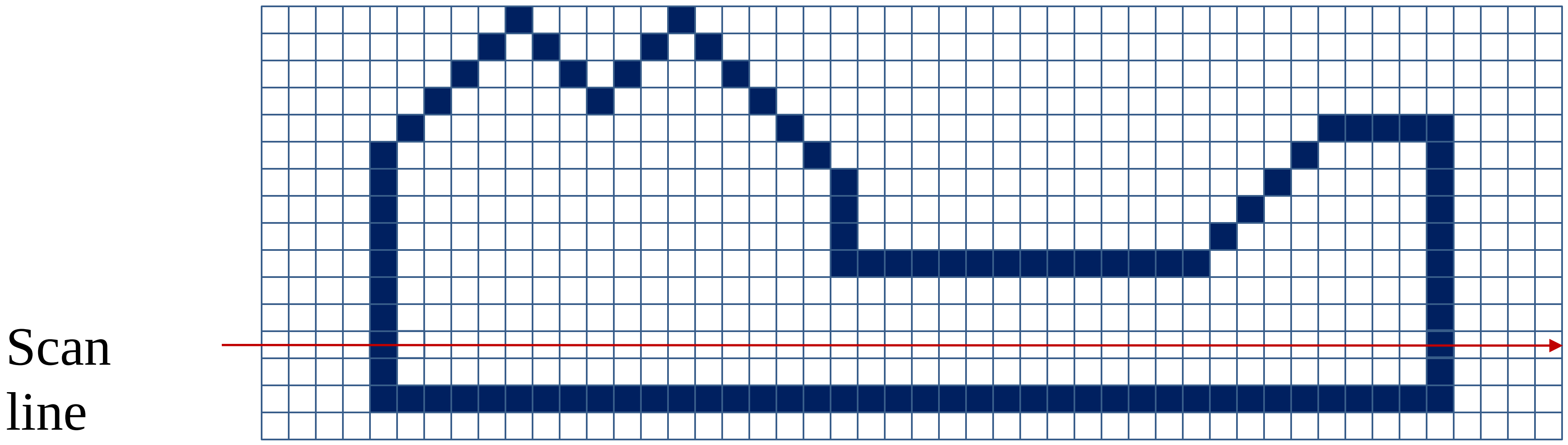
Introduction to Flood-Fill Algorithm

- Sometimes it is required to fill in an area that is not defined within a single colour boundary.
- In such cases we can fill areas by replacing a specified interior colour instead of searching for a boundary colour.
- This approach is called a flood-fill algorithm. Like boundary fill algorithm, here we start with some seed and examine the neighbouring pixels.
- However, here pixels are checked for a specified interior colour instead of boundary colour and they are replaced by new colour.
- Using either a 4-connected or 8-connected approach, we can step through pixel positions until all interior point have been filled.

```
Procedure floodfill (x, y,fill_ color, old_color:
integer)
    If (getpixel (x, y)=old_color)
   {
    setpixel (x, y, fill_color);
    fill (x+1, y, fill_color, old_color);
     fill (x-1, y, fill_color, old_color);
    fill (x, y+1, fill_color, old_color);
    fill (x, y-1, fill_color, old_color);
     }
}
```

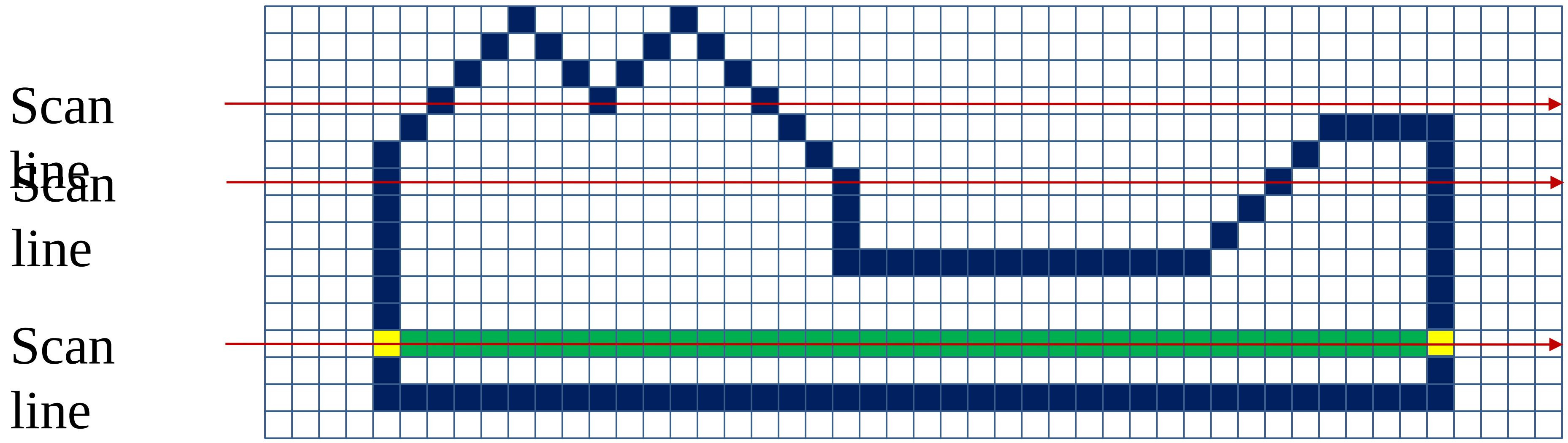
Scan-Line Polygon Fill Algorithm

- For each scan-line crossing a polygon, the algorithm locates the intersection points are of scan line with the polygon edges.
- This intersection points are stored from left to right.
- Frame buffer positions between each pair of intersection point are set to specified fill color.



Contd.

- Scan line intersects at vertex are required special handling.
- For vertex we must look at the other endpoints of the two line segments which meet at this vertex.
  - If these points lie on the same (up or down) side of the scan line, then that point is counts as two intersection points.
  - If they lie on opposite sides of the scan line, then the point is counted as single intersection.



Edge Intersection Calculation with Scan-Line

- Coherence methods often involve incremental calculations applied along a single scan line or between successive scan lines.
- In determining edge intersections, we can set up incremental coordinate calculations along any edge by exploiting the fact that the slope of the edge is constant from one scan line to the next.
- For above figure we can write slope equation for polygon boundary as follows.
- $m = \frac{y_{k+1} - y_k}{x_{k+1} - x_k}$
- Since change in y coordinates between the two scan lines is simply
- $y_{k+1} - y_k = 1$

Contd.

- So slope equation can be modified as follows
- $m = \frac{y_{k+1} - y_k}{x_{k+1} - x_k}$
- $m = \frac{1}{x_{k+1} - x_k}$
- $x_{k+1} - x_k = \frac{1}{m}$
- $x_{k+1} = x_k + \frac{1}{m}$
- Each successive x intercept can thus be calculated by adding the inverse of the slope and rounding to the nearest integer.

Edge Intersection Calculation with Scan-Line for parallel execution

- For parallel execution of this algorithm we assign each scan line to separate processor in that case instead of using previous x values for calculation we use initial x values by using equation as.
- $x_k = x_0 + \frac{k}{m}$
- Now if we put $m = \frac{\Delta y}{\Delta x}$ in incremental calculation equation $x_{k+1} = x_k + \frac{1}{m}$ then we obtain equation as.
- $x_{k+1} = x_k + \frac{\Delta x}{\Delta y}$
- Using this equation we can perform integer evaluation of x intercept.

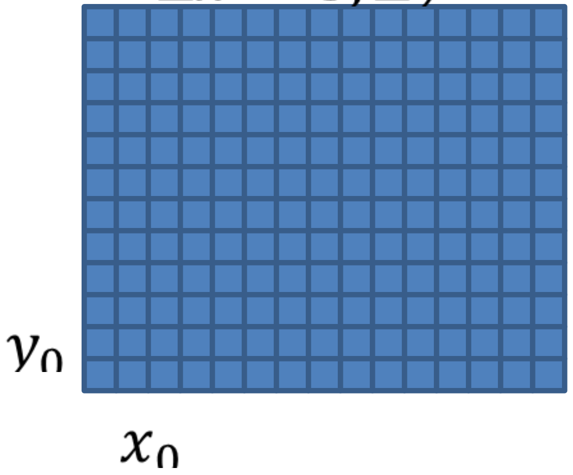
Simplified Method for Edge Intersection Calculation with Scan-Line

1. Suppose $m = 7/3$
2. Initially, set counter to 0, and increment to 3 (which is Δx).
3. When move to next scan line, increment counter by adding Δx
4. When counter is equal to or greater than 7 (which is Δy), increment the $x - intercept$ (in other words, the $x - intercept$ for this scan line is one more than the previous scan line), and decrement counter by 7(which is Δy).

$\Delta x = 3, \Delta y =$

Counter

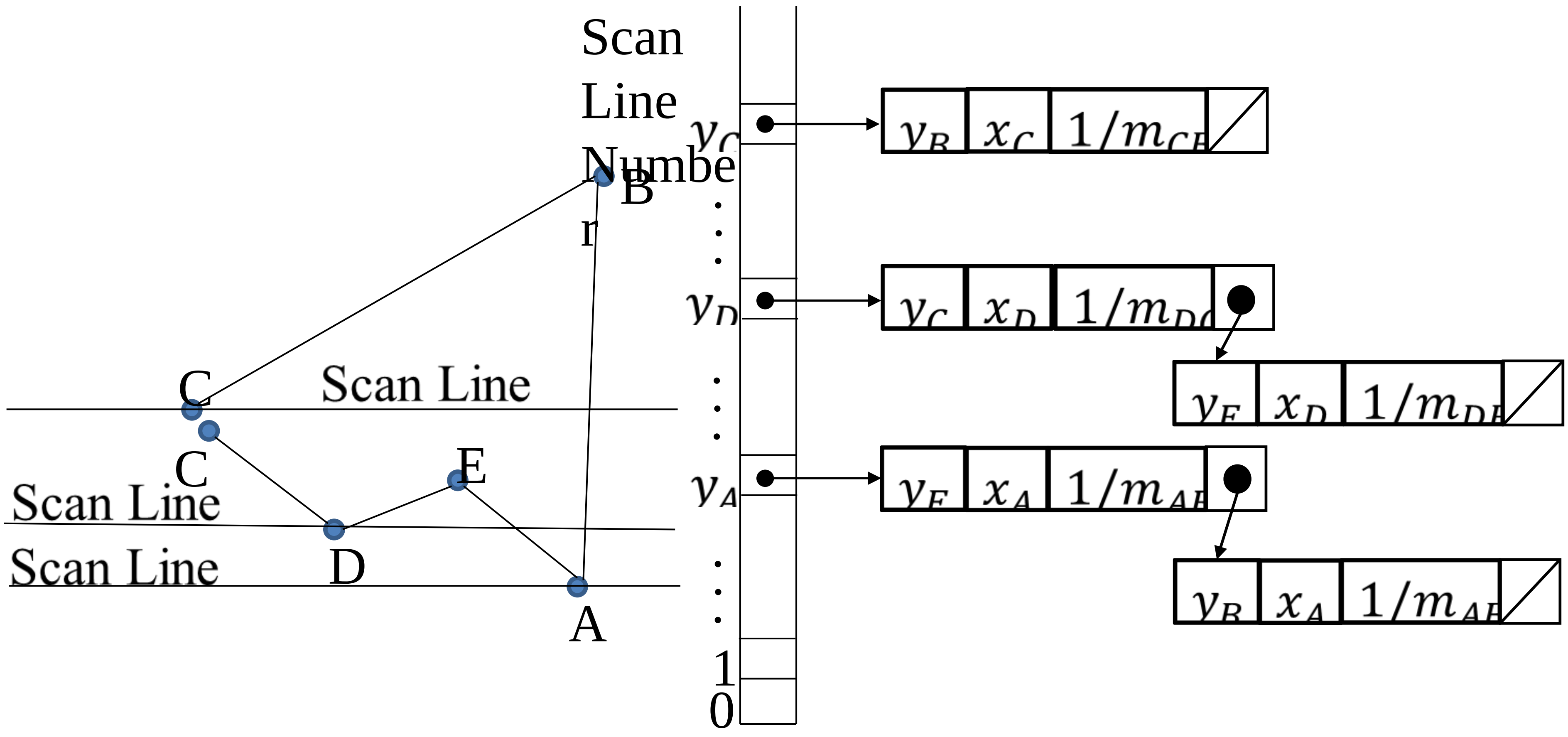
=[illegible]



Use of Sorted Edge table in Scan-Line Polygon Fill Algorithm

- To efficiently perform a polygon fill, we can first store the polygon boundary in a sorted edge table.
- It contains all the information necessary to process the scan lines efficiently.
- We use bucket sort to store the edge sorted on the smallest y value of each edge in the correct scan line positions.
- Only the non-horizontal edges are entered into the sorted edge table.

Contd.


Scan Line Number
B
C
Scan Line
C
E
Scan Line
Scan Line
D
A
y_C
y_D
y_A
1
0
y_B | x_C | $1/m_{CB}$
y_C | x_D | $1/m_{DC}$
y_E | x_D | $1/m_{DE}$
y_E | x_A | $1/m_{AE}$
y_B | x_A | $1/m_{AB}$

Character Generation

- We can display letters and numbers in variety of size and style.
- The overall design style for the set of character is called typeface.
- Today large numbers of typefaces are available for computer application for example Helvetica, Arial etc.
- Originally, the term font referred to a set of cast metal character forms in a particular size and format.
- Example: 10-point Courier Italic or 12- point Palatino Bold.

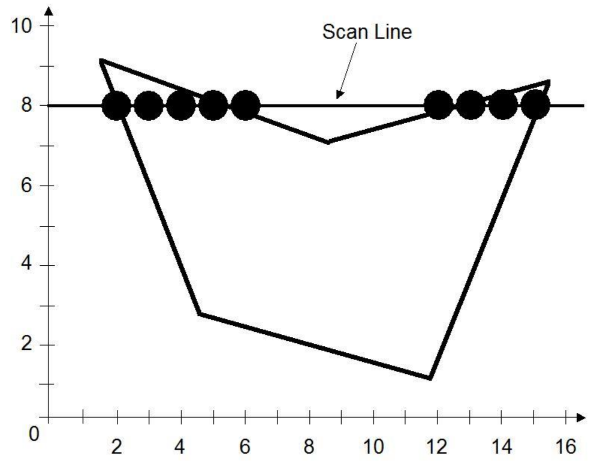
The Scan-Line Polygon Fill Algorithm

The scan-line polygon-filling algorithm involves

• the **horizontal scanning** of the polygon from its **lowermost** to its **topmost** vertex,

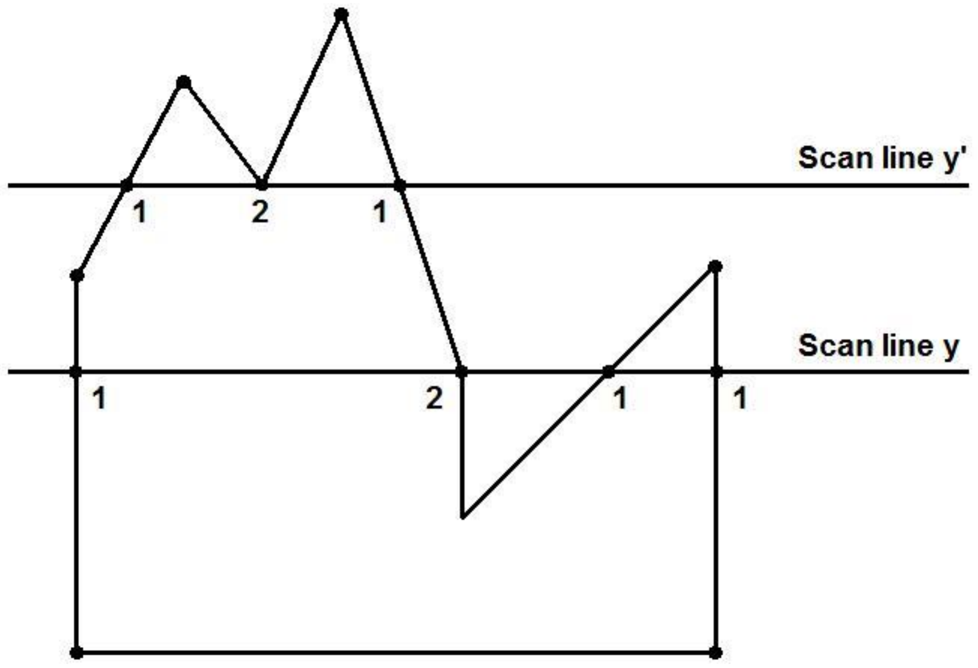
• identifying which edges intersect the scan-line,

• and finally drawing the interior horizontal lines with the specified fill color. process.



The Scan-Line Polygon Fill Algorithm

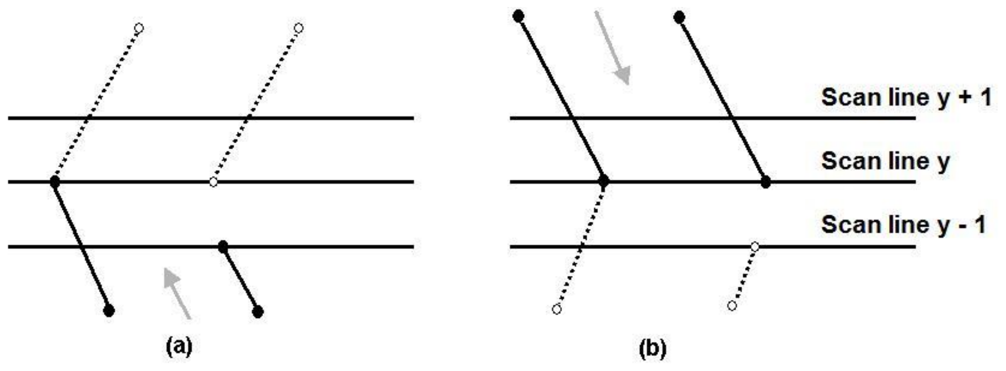
Dealing with vertices



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)



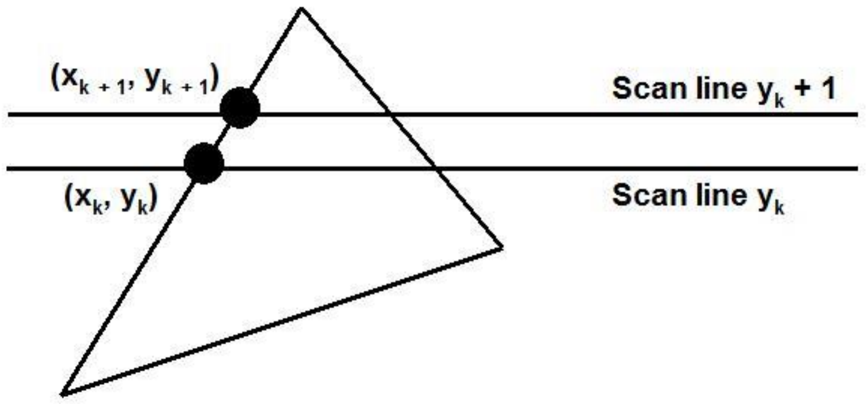
The Scan-Line Polygon Fill Algorithm

Determining Edge Intersections

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

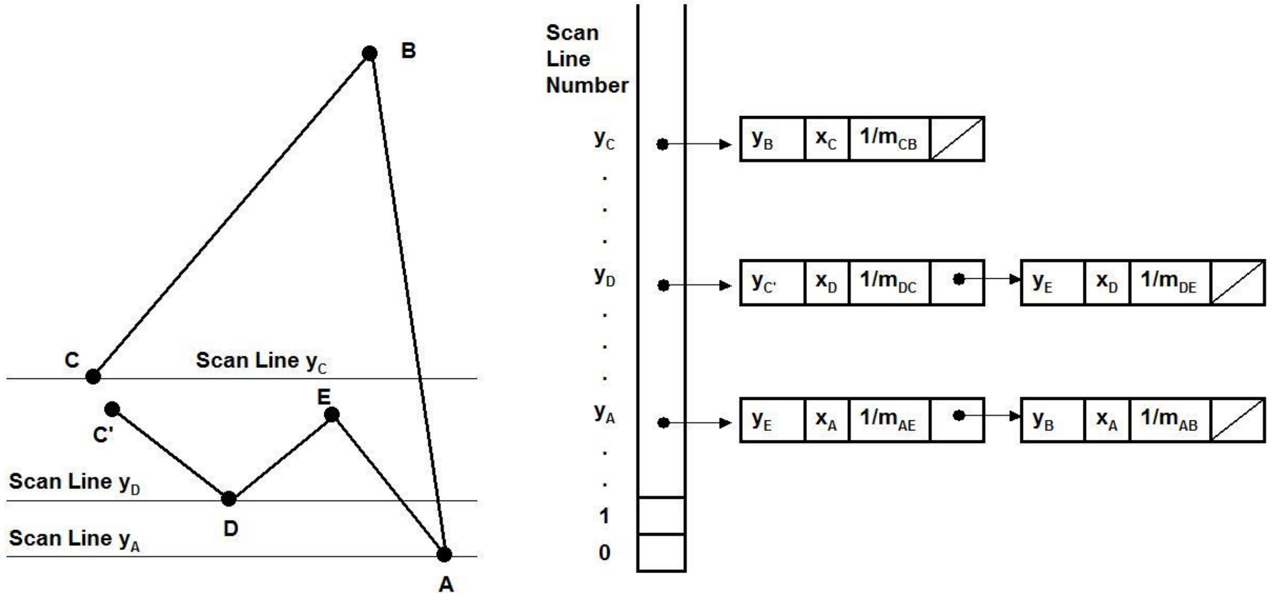
$$y_{k+1} - y_k = 1$$

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



The Scan-Line Polygon Fill Algorithm

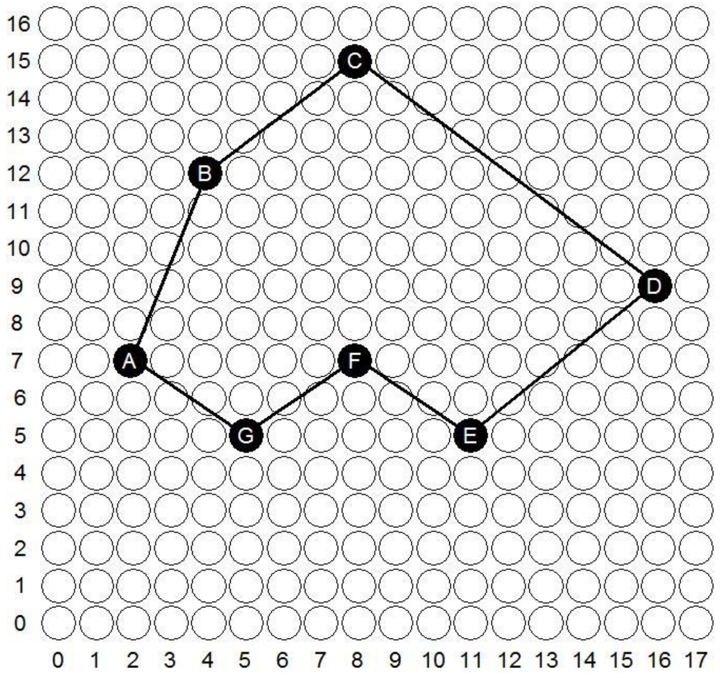
- Each **entry** in the table for a particular scan line contains the **maximum y** value for that edge, the **x-intercept** value (**at the lower vertex**) for the edge, and the **inverse slope** of the edge.

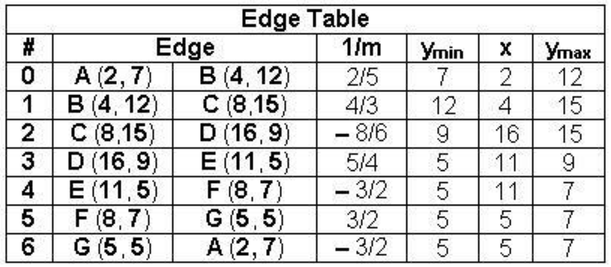

B
C
Scan Line y_C
E
C'
Scan Line y_D
D
Scan Line y_A
A
Scan Line Number
y_C
y_D
y_A
1
0
y_B | x_C | $1/m_{CB}$
$y_{C'}$ | x_D | $1/m_{DC}$
y_E | x_D | $1/m_{DE}$
y_E | x_A | $1/m_{AE}$
y_B | x_A | $1/m_{AB}$

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)}



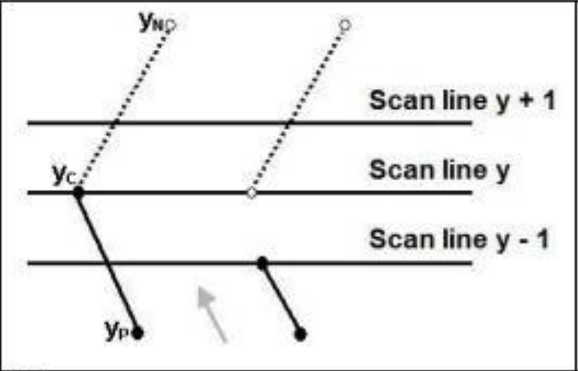
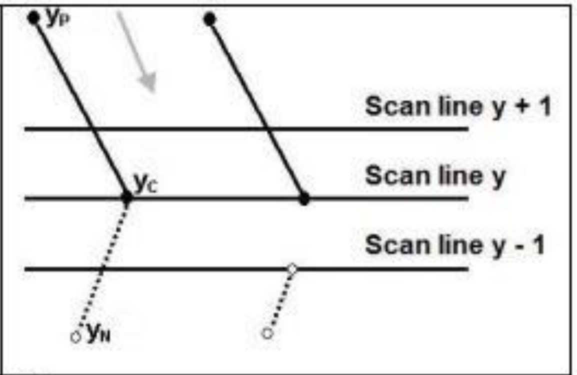


Edge Table

#	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

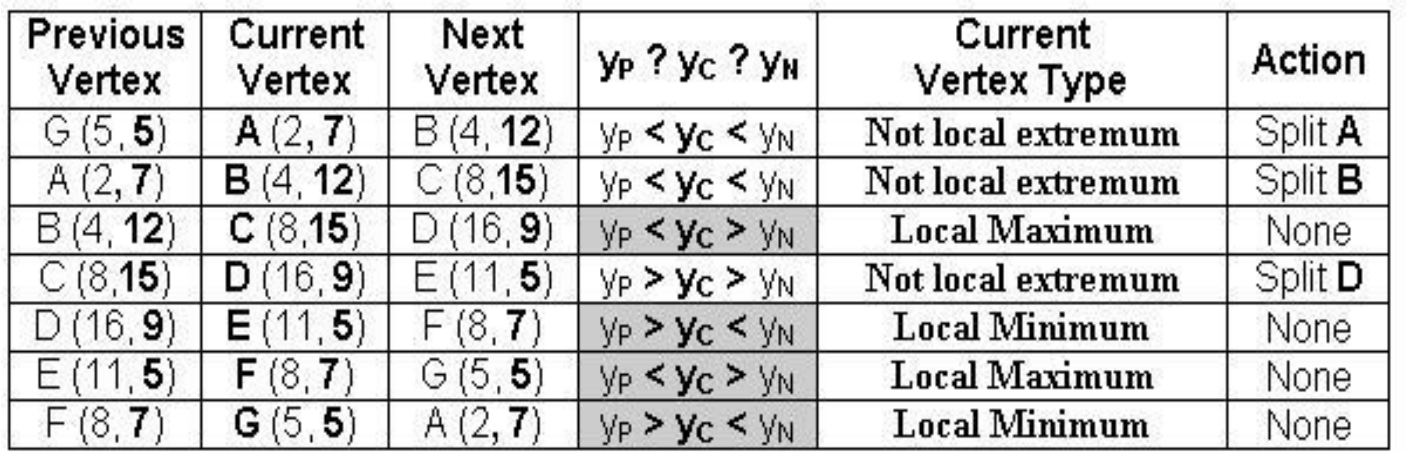
The Scan-Line Polygon Fill Algorithm

(Example)

 y_N Scan line y + 1 y_C Scan line y Scan line y - 1 y_P	 y_P Scan line y + 1 y_C Scan line y Scan line y - 1 y_N
If $y_P < y_C < y_N$ Then y_C is decreased by one. The new edges become $(x_P, y_P) \rightarrow (x'_C, y_C - 1)$ 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**)

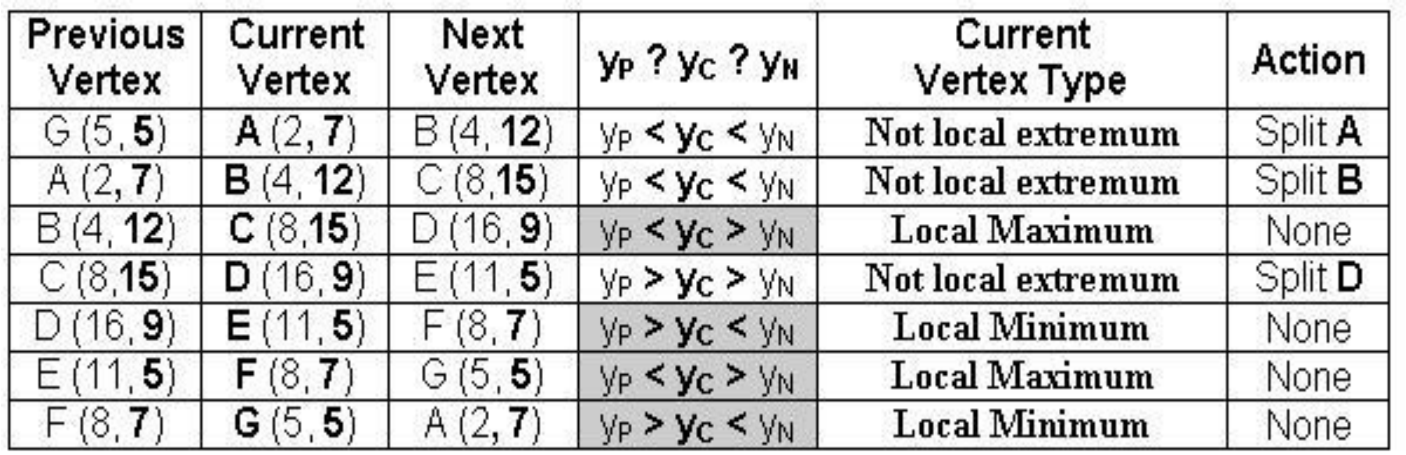
$$m = (5 - 7)/(5 - 2) = -2/3$$

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

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**)

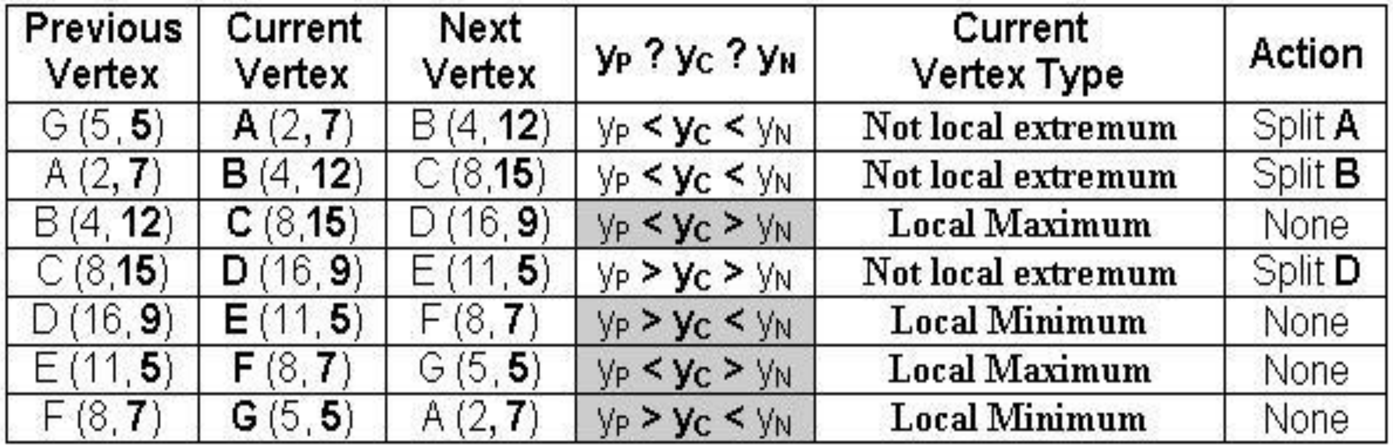
$$m = (7 - 12)/(2 - 4) = 5/2$$

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

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**)

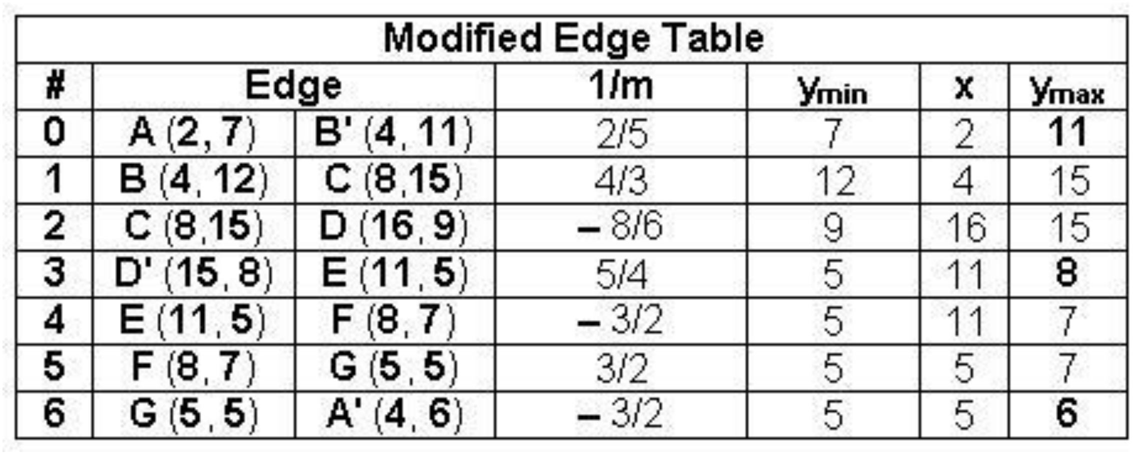
$$m = (5 – 9)/(11 – 16) = 4/5$$

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

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

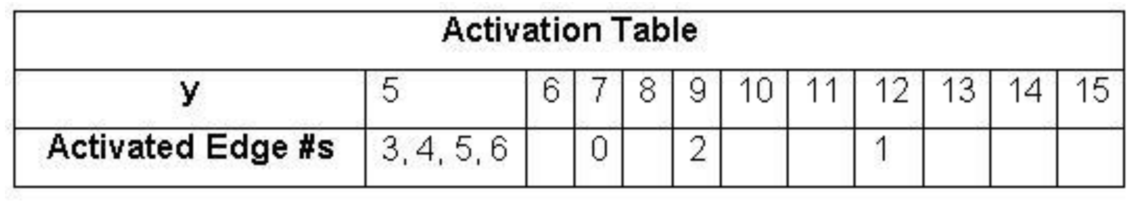
The Scan-Line Polygon Fill Algorithm

(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

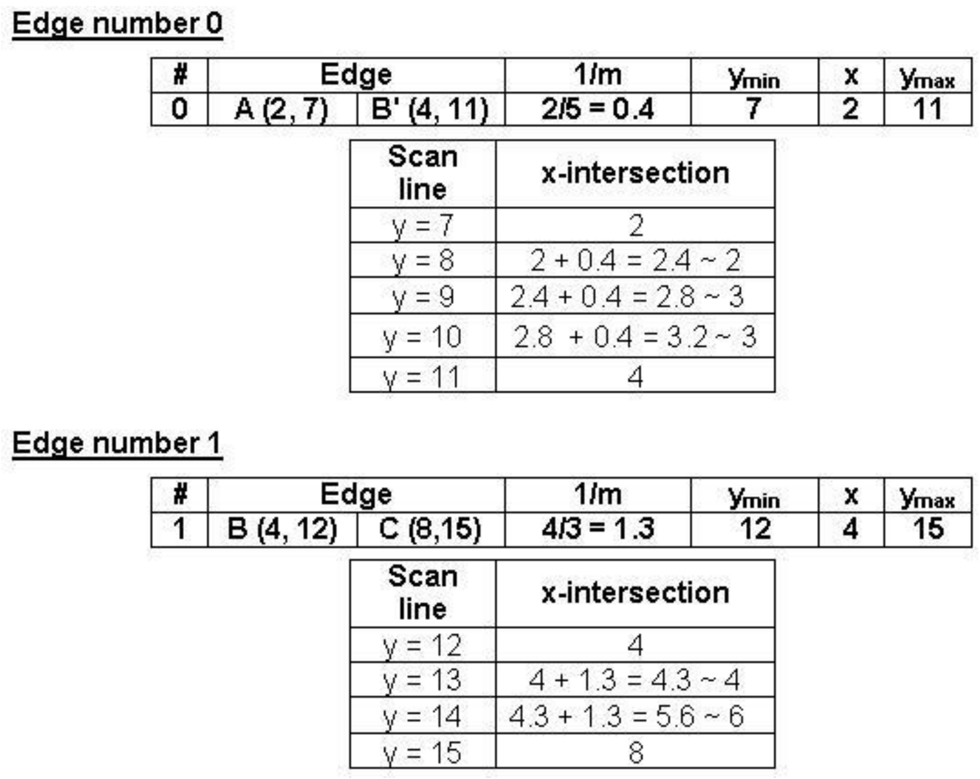


Activation Table

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

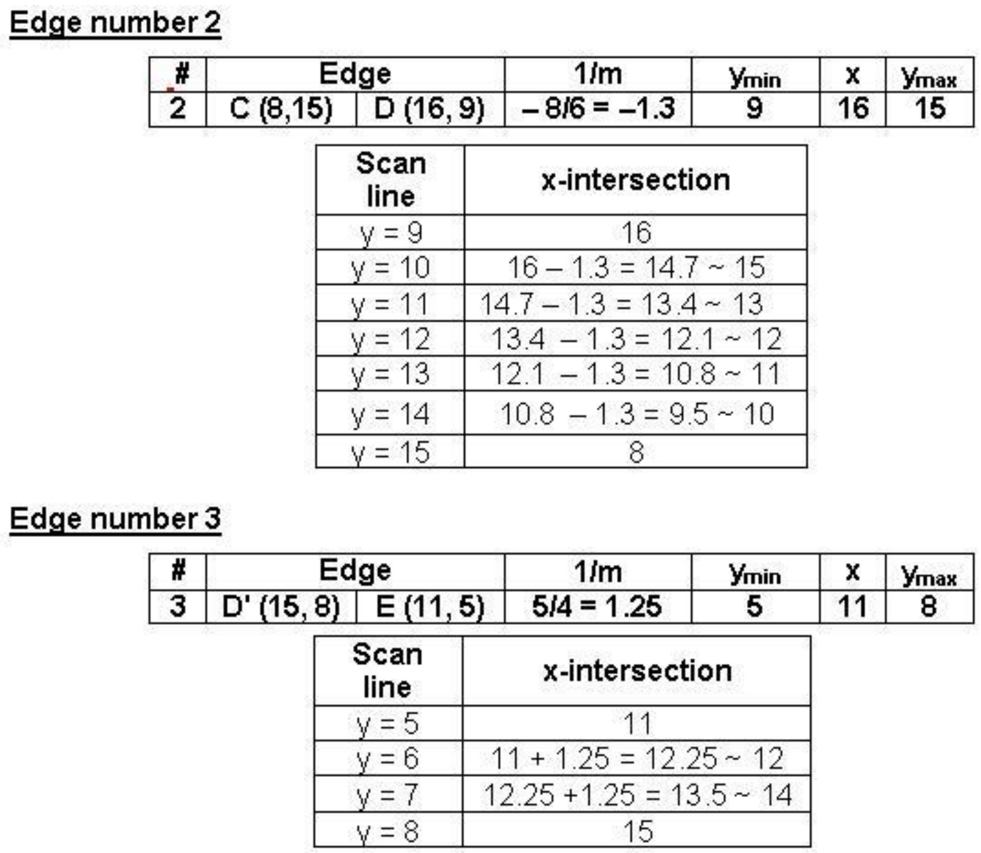
Edge number 1

#	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

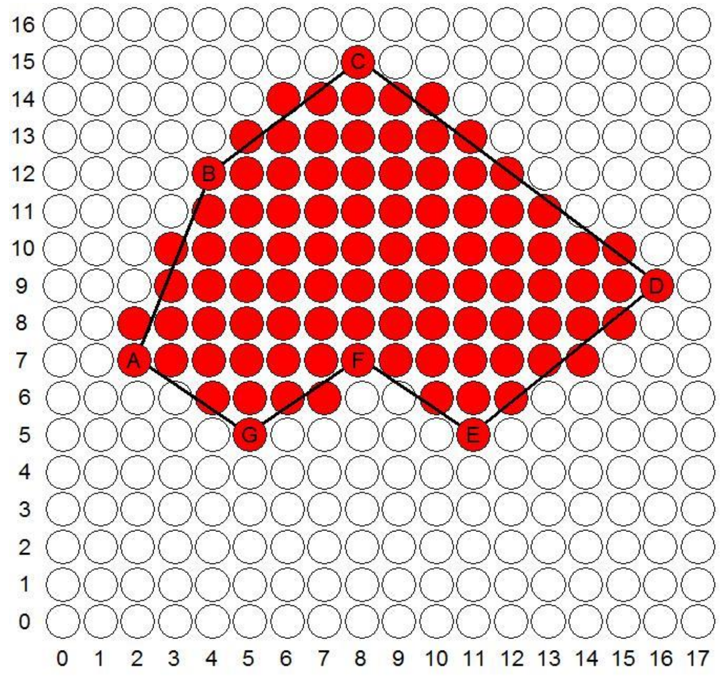
#	Edge		1/m	y_{min}	x	y_{max}
6	G (5, 5)	A' (4, 6)	– 3/2 = –1.5	5	5	6

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

The Scan-Line Polygon Fill Algorithm

(Example)

Scan line	x-intersections							x-intersections pair Ascending order
	Edge#							
	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)



Contd.

- Now, the terms font and typeface are often used interchangeably, since printing is no longer done with cast metal forms.
- Methods of character generation are:
  - Bitmap Font/ Bitmapped Font
  - Outline Font
  - Stroke Method
  - Starbust Method

Bitmap Font/ Bitmapped Font

- A simple method for representing the character shapes in a particular typeface is to use rectangular grid patterns.
- In frame buffer, the 1 bits designate which pixel positions are to be displayed on the monitor.

0	0	1	1	1	1	0	0
0	1	1	1	1	1	1	0
1	1	0	0	0	0	1	1
1	1	0	0	0	0	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	0	0	0	0	1	1
1	1	0	0	0	0	1	1

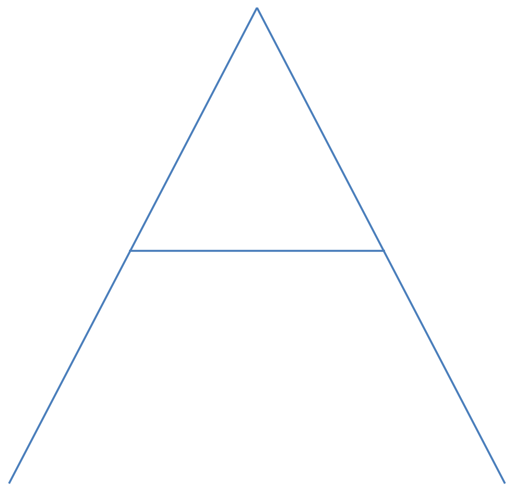
- Bitmap fonts are the simplest to define and display.
- Bitmap fonts require more space.
- It is possible to generate different size and other variation from one set but this usually does not produce good result.

Outline Font

- In this method character is generated using curve section and straight line as combine assembly.
- To display the character we need to fill interior region of the character.
- This method requires less storage since each variation does not required a distinct font cache.
- We can produce boldface, italic, or different sizes by manipulating the curve definitions for the character outlines.
- But this will take more time to process the outline fonts, because they must be scan converted into the frame buffer.

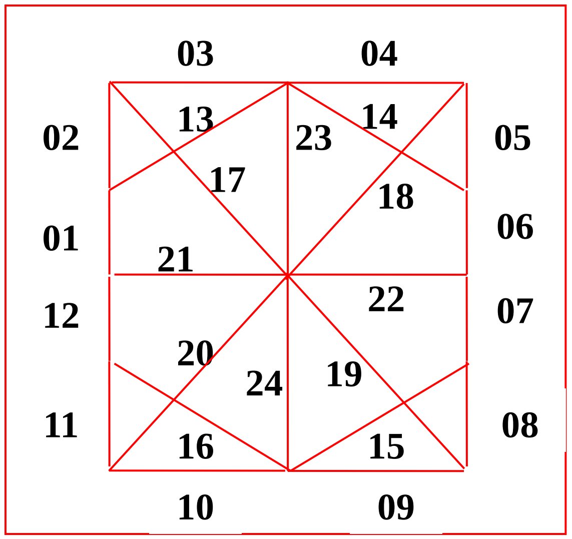
Stroke Method

- It uses small line segments to generate a character.
- The small series of line segments are drawn like a stroke of a pen to form a character.
- We can generate our own stroke method by calling line drawing algorithm.
- Here it is necessary to decide which line segments are needs for each character and then draw that line to display character.
- It support scaling by changing length of line segment.



Starbust Method

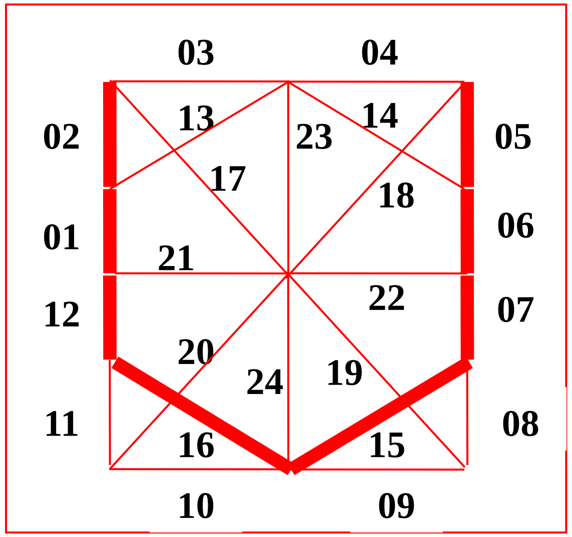
- In this method a fix pattern of lines (24 line) segments are used to generate characters.



- We highlight those lines which are necessary to draw a particular character.
- Pattern for particular character is stored in the form of 24 bit code.
- In which each bit represents corresponding line having that number.
- We put bit value 1 for highlighted line and 0 for other line.

Contd.

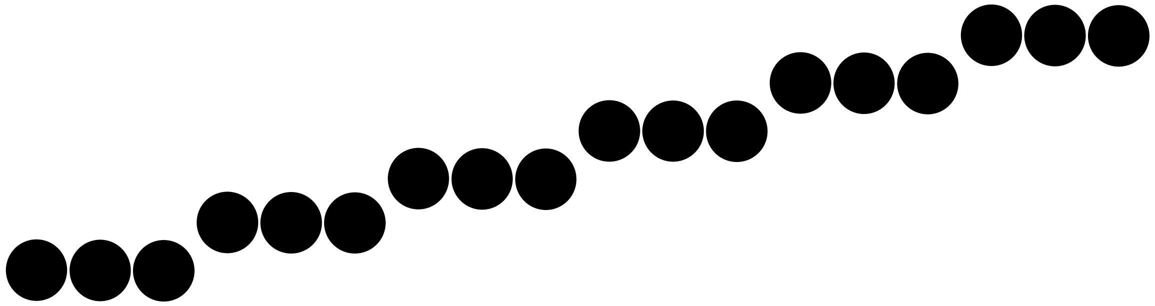
- Example letter V



- Code for letter V = 1 1 0 0 1 1 1 0 0 0 0 1 0 0 1 1 0 0 0 0 0 0 0 0
- This technique is not used now a days because:
  - It requires more memory to store 24 bit code for single character.
  - It requires conversion from code to character.
  - It doesn’t provide curve shapes.

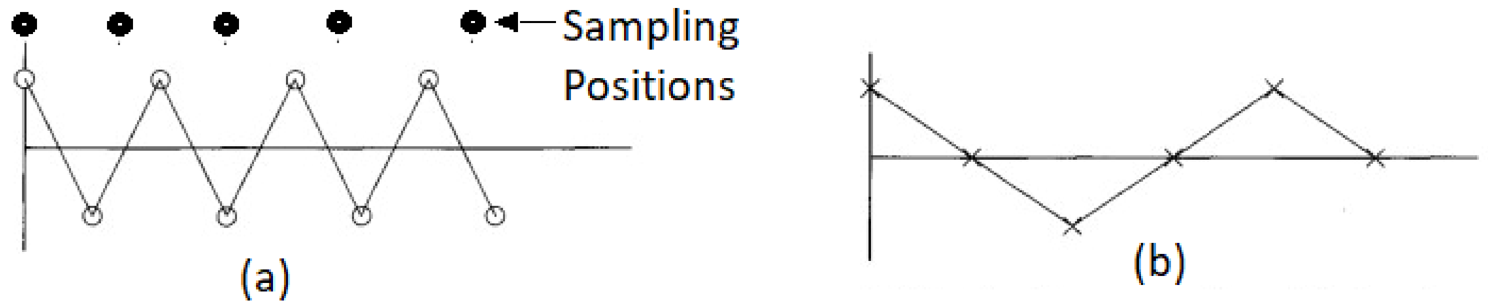
Aliasing

- Primitives generated in raster graphics by various algorithms have stair step shape or jagged appearance.
- Jagged appearance is due to integer calculation by rounding actual values.
- This distortion of actual information due to low frequency sampling is called **aliasing.**



Antialiasing

- Minimise effect of aliasing by some way is known as **antialiasing.**
- In periodic shape distortion may be occurs due to under sampling.



- To avoid losing information from such periodic objects, we need to set the sampling frequency to at least twice that of the highest frequency occurring in the object.
- This is referred to as the **Nyquist sampling frequency** (or Nyquist sampling rate):

$$f_s = 2f_{max}$$

Contd.

- In other words sampling interval should be no larger than one-half the cycle. Which is called **nyquist sampling interval.**

$$\Delta x_s = \frac{\Delta x_{cycle}}{2}$$

- One way to solve this problem is to display image on higher resolution.
- But it has also limit that how much large frame buffer we can maintain along with maintaining refresh rate 30 to 60 frame per second.
- And also on higher resolution aliasing will remains up to some extents.
- With raster systems that are capable of displaying more than two intensity levels (color or grayscale), we can apply antialiasing methods to modify pixel intensities.
- By appropriately varying the intensities of pixels along the boundaries of primitives, we can smooth the edges to lessen the aliasing effect.

Antialiasing Methods

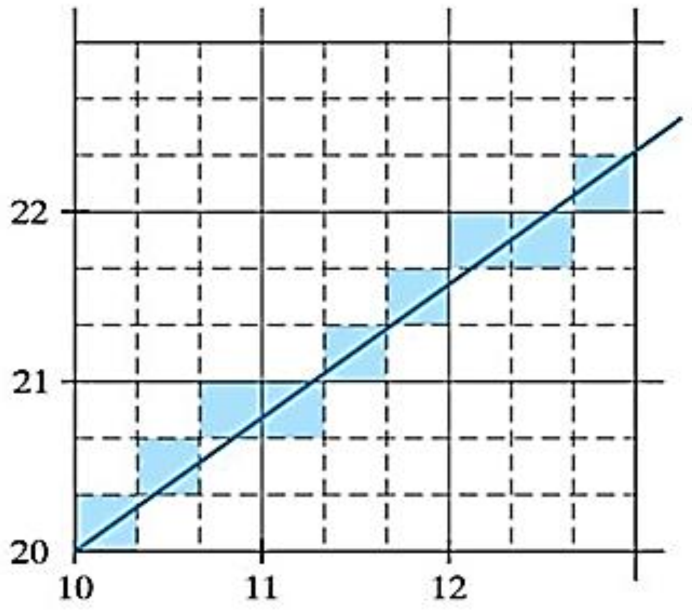
1. Supersampling Straight Line Segments
2. Pixel-Weighting Masks
3. Filtering Techniques
4. Pixel Phasing

https://youtu.be/R3AgvpOA2qw

Antialiasing Methods

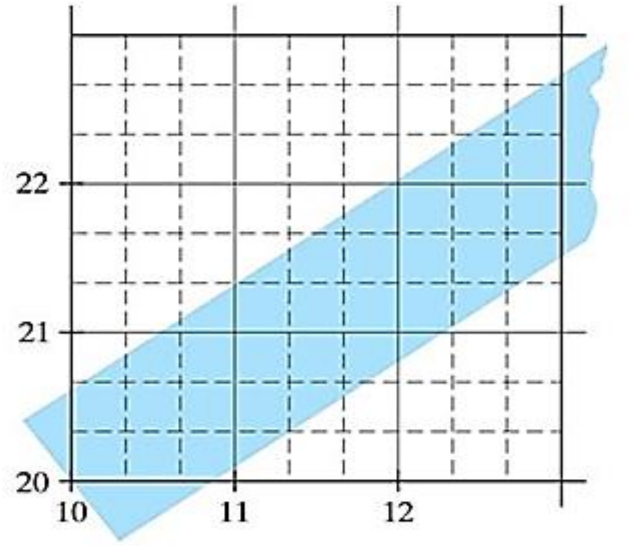
Supersampling Straight Line Segments

- For the greyscale display of a straight-line segment, we can divide each pixel into a number of sub pixels and determine the number of sub pixel along the line path.
- Then we set intensity level of each pixel proportional to number of sub pixel along the line path.
- E.g. in figure area of each pixel is divided into nine sub pixel and then we determine how many number of sub pixel are along the line (it can be 3 or 2 or 1 as we divide into 9 sub pixel).
- Based on number 3 or 2 or 1 we assign intensity value to that pixel.



Contd.

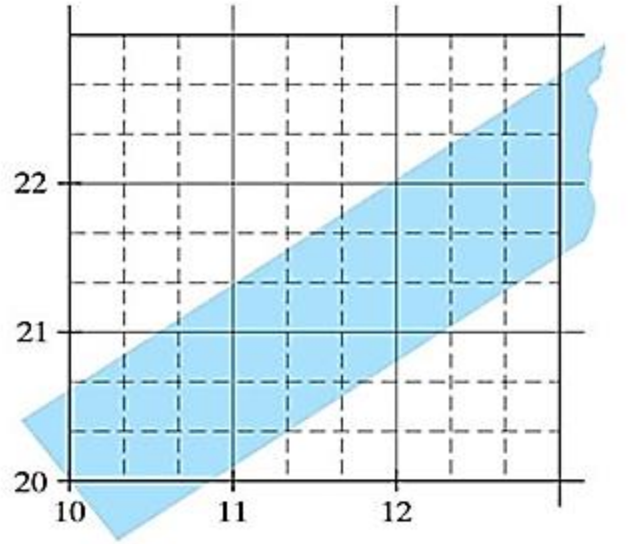
- We can achieve four intensity levels by di pixels and five intensity levels by dividing
- Lower intensity gives blurred effect antialiasing.
- Other way is we considered pixel areas treated the line as a mathematical entity wi



- Actually, displayed lines have a width approximately equal to that of a pixel.
- If we take finite width of the line into account, we can perform supersampling by setting each pixel intensity proportional to the number of sub pixels inside the polygon representing the line area.

Contd.

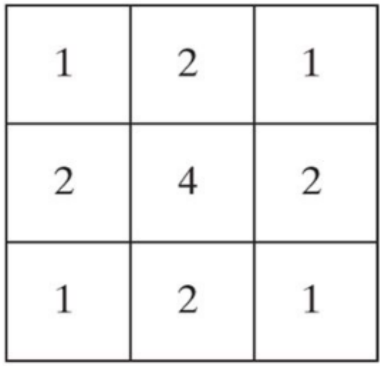
- A sub pixel can be considered to be inside the line if its lower left corner is inside the polygon boundaries.
- Advantage of this is that it having number of intensity equals to number of sub pixel.



- Another advantage of this is that it will distribute total intensity over more pixels.
- E.g. in figure pixel below and left to (10, 20) is also assigned some intensity levels so that aliasing will reduce.
- For color display we can modify levels of color by mixing background color and line color.

Pixel-Weighting Masks

- Supersampling method are often implemented by giving more weight to sub pixel near the center of pixel area.
- As we expect centre sub pixel to be more important in determining the overall intensity of a pixel.
- For the 3 by 3 pixel subdivisions we have considered so far, a weighting scheme as in fig. could be used.
- The center sub pixel here is weighted four times that of the corner sub pixels and twice that of the remaining sub pixels.
- By averaging the weight of sub pixel which are along the line and assign intensity proportional to average weight will reduce aliasing effect.



1	2	1
2	4	2
1	2	1

Filtering Techniques

- It is more accurate method for antialiasing.
- Common example of filter is rectangular, conical and Gaussian filters.
- Methods for applying the filter function are similar to applying a weighting mask, but now we integrate over the pixel surface to obtain the weighted average intensity.
- For reduce computation we often use table look up.

Pixel Phasing

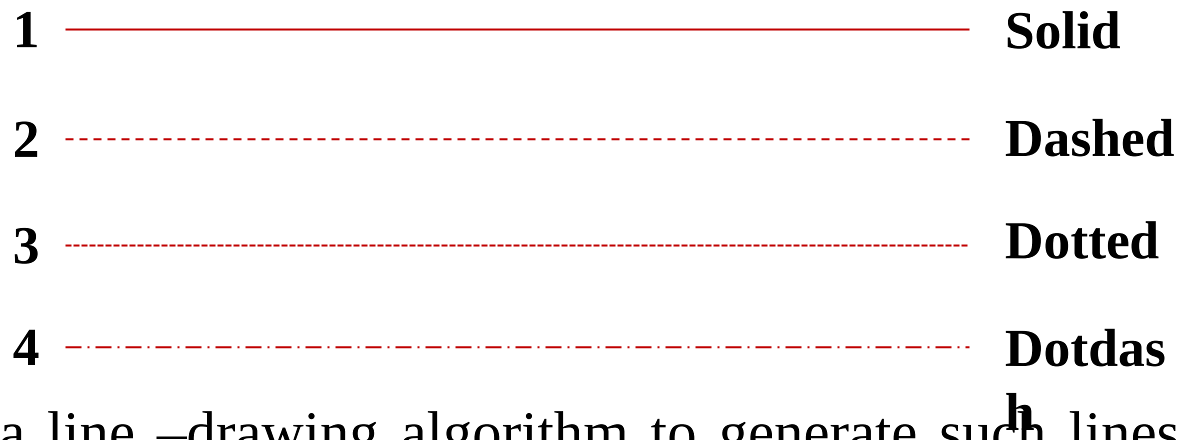
- On raster system if we pass the electron beam from the closer sub pixel so that overall pixel is shifted by factor ¼, ½, or ¾ to pixel diameter.
- Where beam strike at that part of pixel get more intensity then other parts of the pixel and gives antialiasing effect.
- Some systems also allow the size of individual pixels to be adjusted as an additional means for distributing intensities.

Line Attributes

- Basic attributes of a straight line segment are:
  - Type
  - Dimension
  - color
  - pen or brush option.

Line Type

- Possible selection for the line-type attribute includes solid lines, dashed lines, and dotted lines etc.



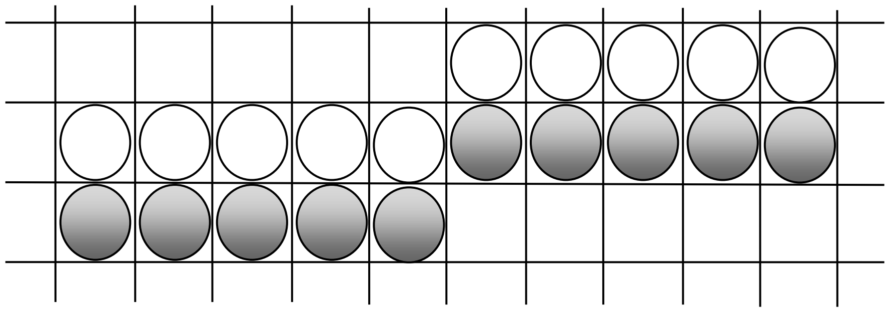
- We modify a line –drawing algorithm to generate such lines by setting the length and spacing of displayed solid sections along the line path.
- To set line type attributes in a PHIGS application program, a user invokes the function: **setLinetype(It)**
- Where parameter lt is assigned a positive integer value of 1, 2, 3, 4… etc. to generate lines that are, respectively solid, dashed, dotted, or dotdash etc.

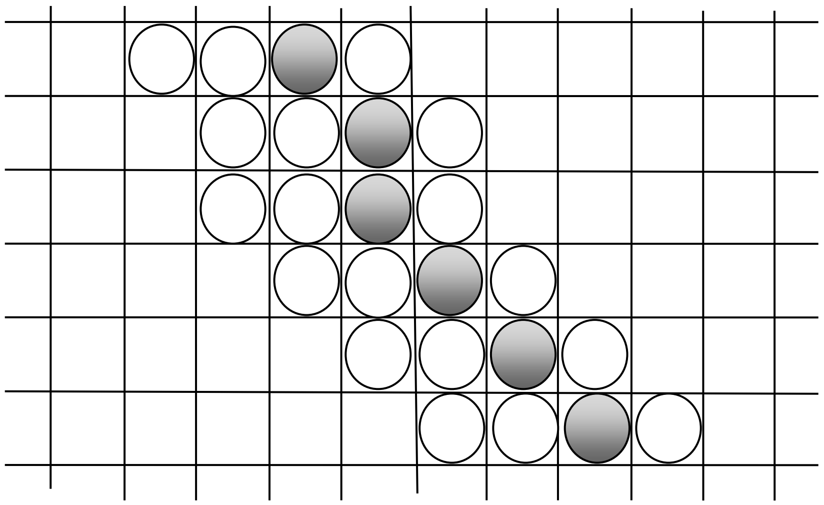
Line Width

- Implementation of line-width options depends on the capabilities of the output device.
- A heavy line on a video monitor could be displayed as adjacent parallel lines, while a pen plotter might require pen changes.
- To set line width attributes in a PHIGS application program, a user invokes the function: **setLinewidthScalFactor (lw)**
- Line-width parameter lw is assigned a positive number to indicate the relative width of the line to be displayed.
- Values greater than 1 produce lines thicker than the standard line width and values less than the 1 produce line thinner than the standard line width.

Contd.

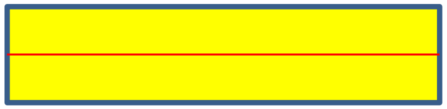
- In raster graphics we generate thick line by plotting
  - above and below pixel of line path when slope |m|<1. &
  - left and right pixel of line path when slope |m|>1.

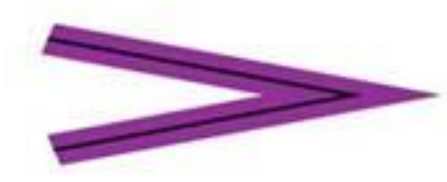


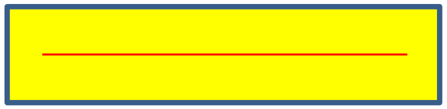


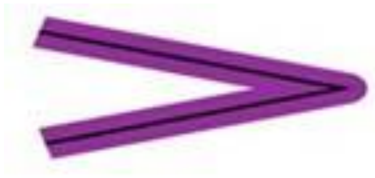
Line Width at Endpoints and Join

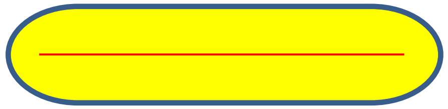
- As we change width of the line we can also change line end and join of two lines which are shown below

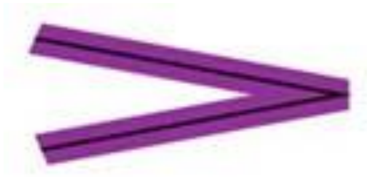

Butt caps


Miter join


Projecting square caps


Round join


Round caps


Bevel join

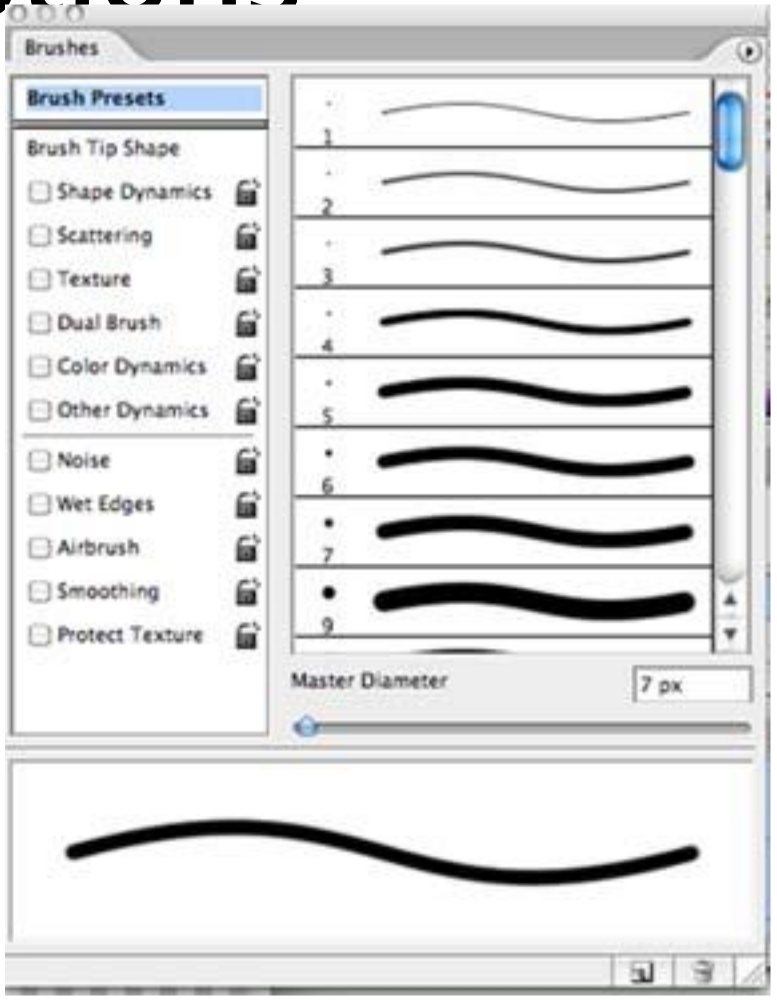
Line color

- The name itself suggests that it is defining color of line displayed on the screen.

- By default system produce line with current color but we can change this color by following function in PHIGS package as follows: **setPolylinecolorIndex (lc)**
- In this lc is constant specifying particular color to be set.

Pen and Brush Options

- In some graphics packages line is displayed with pen and brush selections.
- Options in this category include shape, size, and pattern.
- These shapes can be stored in a pixel mask that identifies the array of pixel positions that are to be set along the line path.
- Also, lines can be displayed with selected patterns by superimposing the pattern values onto the pen or brush mask.



Area-Fill Attributes

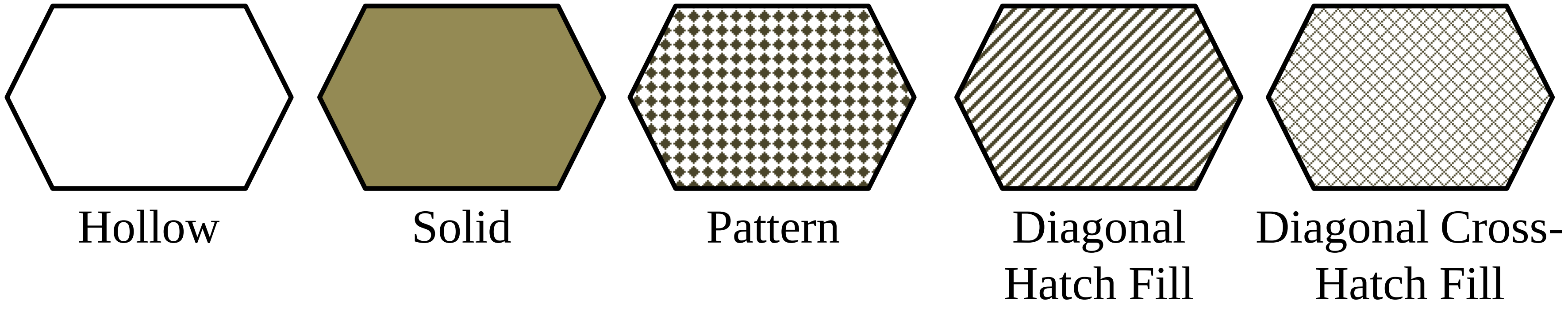
- For filling any area we have choice between solid colors or pattern to fill all these are include in area fill attributes. Which are:
  - Fill Styles
  - Pattern Fill
  - Soft Fill

Fill Styles

- Area are generally displayed with three basic style.
  1. hollow with color border
  2. filled with solid color
  3. filled with some design
- In PHIGS package fill style is selected by following function: **setInteriorStyle (fs)**
- Value of fs include hollow ,solid, pattern etc.

Contd.

- Another values for fill style is hatch, which is patterns of line like parallel line or crossed line.



- For setting interior color in PHIGS package we use: **setInteriorColorIndex (fc)**
- Where fc specify the fill color.

Pattern Fill

- We select the pattern with **setInteriorStyleIndex (pi)**
- Where pattern index parameter pi specifies position in pattern table entry.
- For example:
  - **SetInteriorStyle(pattern) ;**
  - **setInteriorStyleIndex (2) ;**
  - **fillArea (n, points);**

Pattern Table

Index(pi)	Pattern(cp)
1	$\begin{bmatrix} 4 & 0 \\ 0 & 4 \end{bmatrix}$
2	$\begin{bmatrix} 2 & 1 & 2 \\ 1 & 2 & 1 \\ 2 & 1 & 2 \end{bmatrix}$

Contd.

- We can also maintain separate table for hatch pattern.
- We can also generate our own table with required pattern.
- Other function used for setting other style as follows: **setpatternsize (dx, dy)**
- **setPaternReferencePoint (position)**
- We can create our own pattern by setting and resetting group of pixel and then map it into the color matrix.

Soft Fill

- Soft fill is filling layer of color on back ground color so that we can obtain the combination of both color.
- It is used to recolor or repaint so that we can obtain layer of multiple color and get new color combination.
- One use of this algorithm is soften the fill at boundary so that blurred effect will reduce the aliasing effect.
- For example if we fill **t** amount of foreground color then pixel color is obtain as:
- $p = tF + (1 - t)B$
- Where F is foreground color and B is background color

Contd.

- If we see this color in RGB component then:
- $p(pixel) = \left(p_r, p_g, p_b\right)$
- $f(forground) = (f_r, f_g, f_b)$
- $b(backgound) = (b_r, b_g, b_b)$
- Then we can calculate t as follows:
- $t = \frac{P_k - B_k}{F_k - B_k}$
- If we use more then two color say three at that time equation becomes as follow:
- $p = t_0 F + t_1 B_1 + (1 - t_0 - t_1) B_2$
- Where the sum of coefficient t_0, t_1, and $(1 - t_0 - t_1)$ is 1.

Character Attributes

- The appearance of displayed characters is controlled by attributes such as:
  - Font
  - Size
  - Color
  - Orientation.
- Attributes can be set for entire string or may be individually.

Text Attributes

- In text we are having so many style and design like italic fonts, bold fonts etc.
- For setting the font style in PHIGS package we have function:

  setTextFont (tf)
- Where tf is used to specify text font.
- For setting color of character in PHIGS we have function:

  setTextColorIndex (tc)
- Where text color parameter tc specifies an allowable color code.
- For setting the size of the text we use function:

  setCharacterheight (ch)
- Where ch is used to specify character height.

Contd.

- For scaling the character we use function:
  <u>setCharacterExpansionFacter (cw)</u>
- Where character width parameter cw is set to a positive real number that scale the character body width.
- Spacing between character is controlled by function:
- **<u>setCharacterSpacing (cs)</u>**
- Where character spacing parameter cs can be assigned any real value.

Contd.

- The orientation for a displayed character string is set according to the direction of the character up vector:

  setCharacterUpVector (upvect)

- Parameter upvect in this function is assigned two values that specify the x and y vector components.
- Text is then displayed so that the orientation of characters from baseline to cap line is in the direction of the up vector.
- For setting the path of the character we use function:

  setTextPath (tp)

- Where the text path parameter tp can be assigned the value: right, left, up, or down.

Contd.

- For setting the alignment we use function:

  setTextAlignment (h, v)

- Where parameter h and v control horizontal and vertical alignment respectively.
- For specifying precision for text display is given with function:

  setTextPrecision (tpr)

- Where text precision parameter tpr is assigned one of the values: string, char, or stroke.
- The highest-quality text is produced when the parameter is set to the value stroke.

Marker Attributes

- A marker symbol display single character in different color and in different sizes.
- For marker attributes implementation by procedure that load the chosen character into the raster at defined position with the specified color and size.
- We select marker type using function: **setMarkerType (mt)**
- Where marker type parameter mt is set to an integer code.

Contd.

- Typical codes for marker type are the integers 1 through 5, specifying, respectively:
  1. a dot (.)
  2. a vertical cross (+)
  3. an asterisk (*)
  4. a circle (o)
  5. a diagonal cross (x).
- Displayed marker types are centred on the marker coordinates.

Contd.

- We set the marker size with function: **<u>SetMarkerSizeScaleFactor (ms)</u>**
- Where parameter marker size ms assigned a positive number according to need for scaling.
- For setting marker color we use function: **<u>setPolymarkerColorIndex (mc)</u>**
- Where parameter mc specify the color of the marker symbol.

Thank You