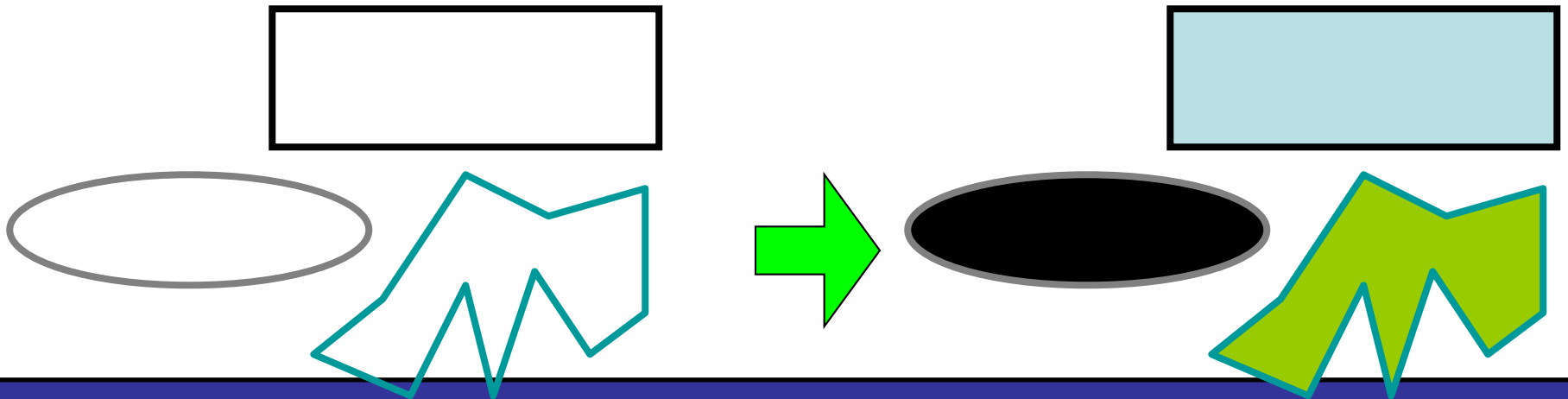


Filling Graphical Shapes

Dr. Subhadip Basu

CSE Dept., JU
subhadip@cse.jdvu.ac.in



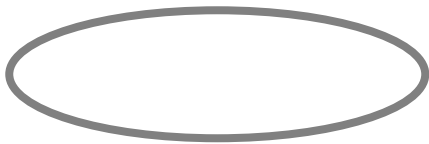


We know how to draw outlines

Can we just fill the “inside”?

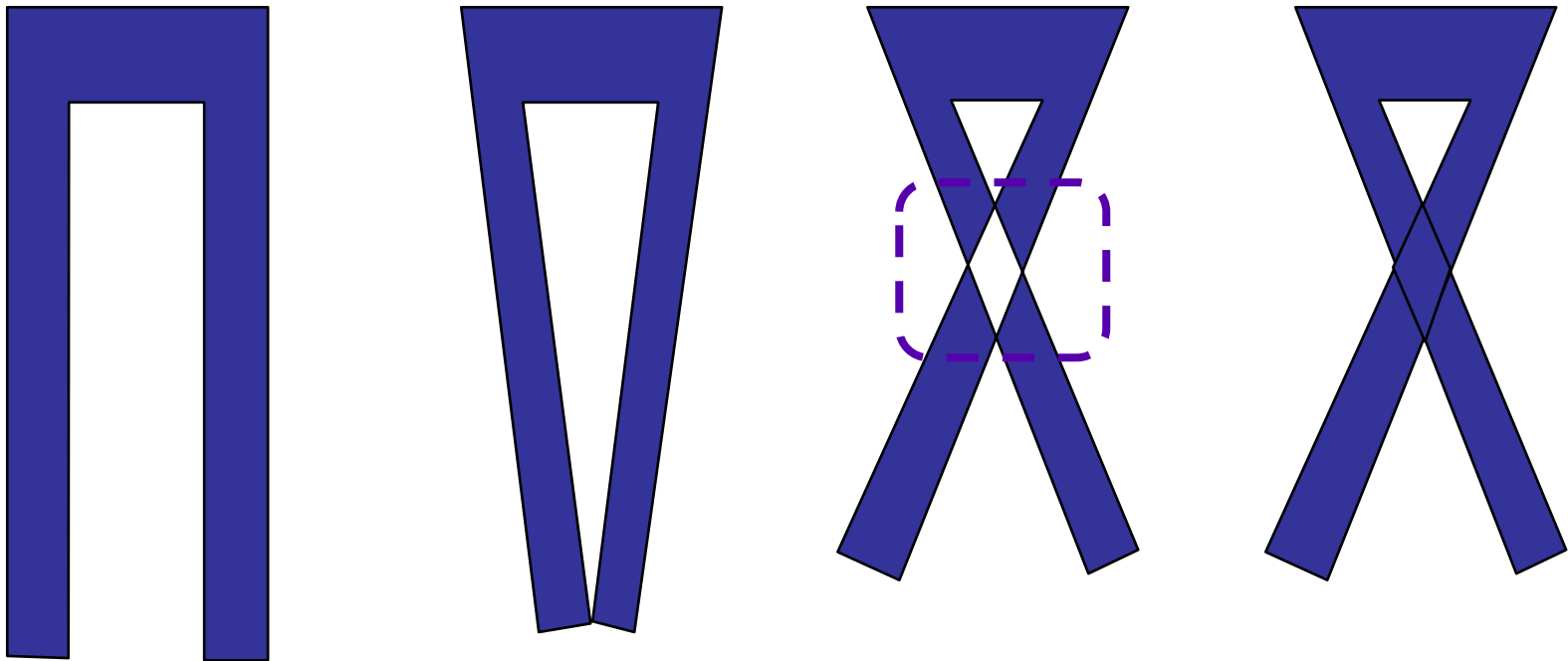
...but how do we know the difference between the inside and outside?

- Can we determine if a point is inside a shape?





A Filling Anomaly





One Approach: The Odd-Even Rule

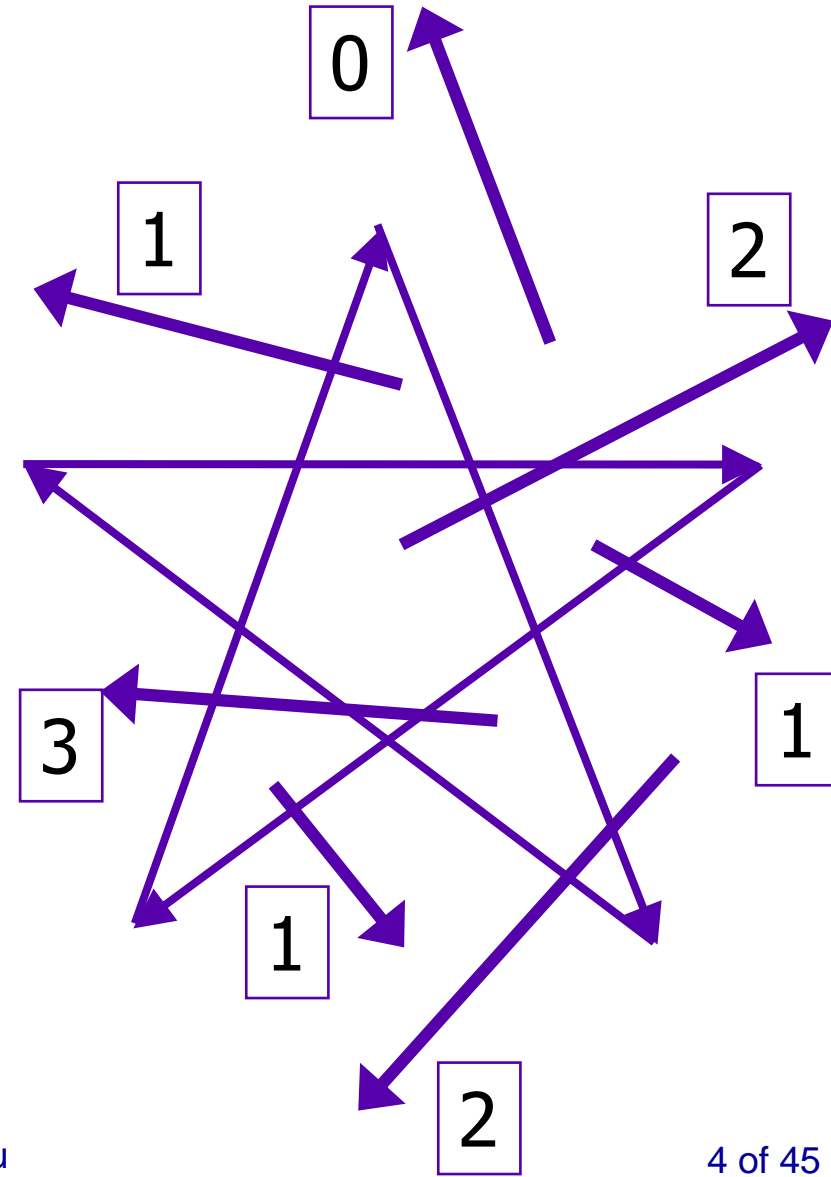
Choose an arbitrary point

Draw ray to a distant point

- Don't intersect any vertices
- What's a "distant" point?

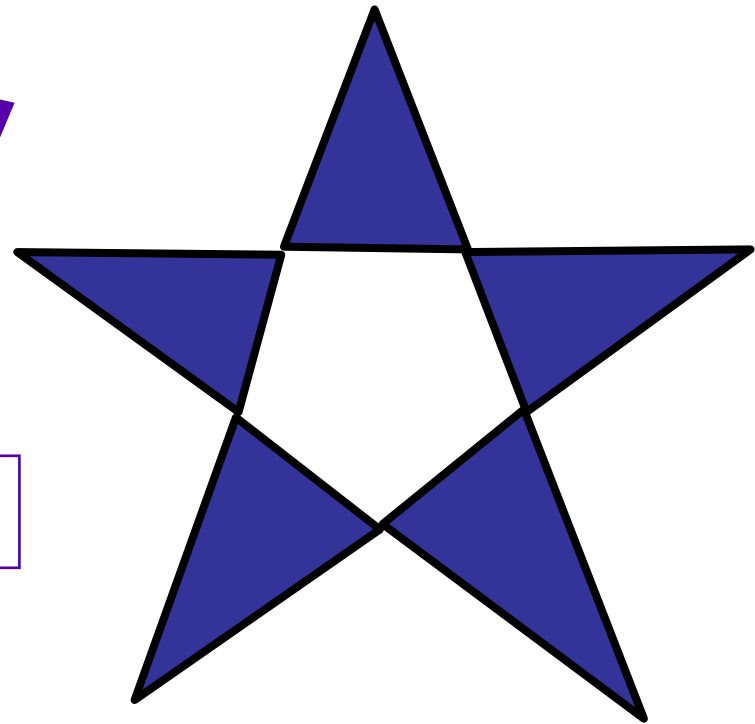
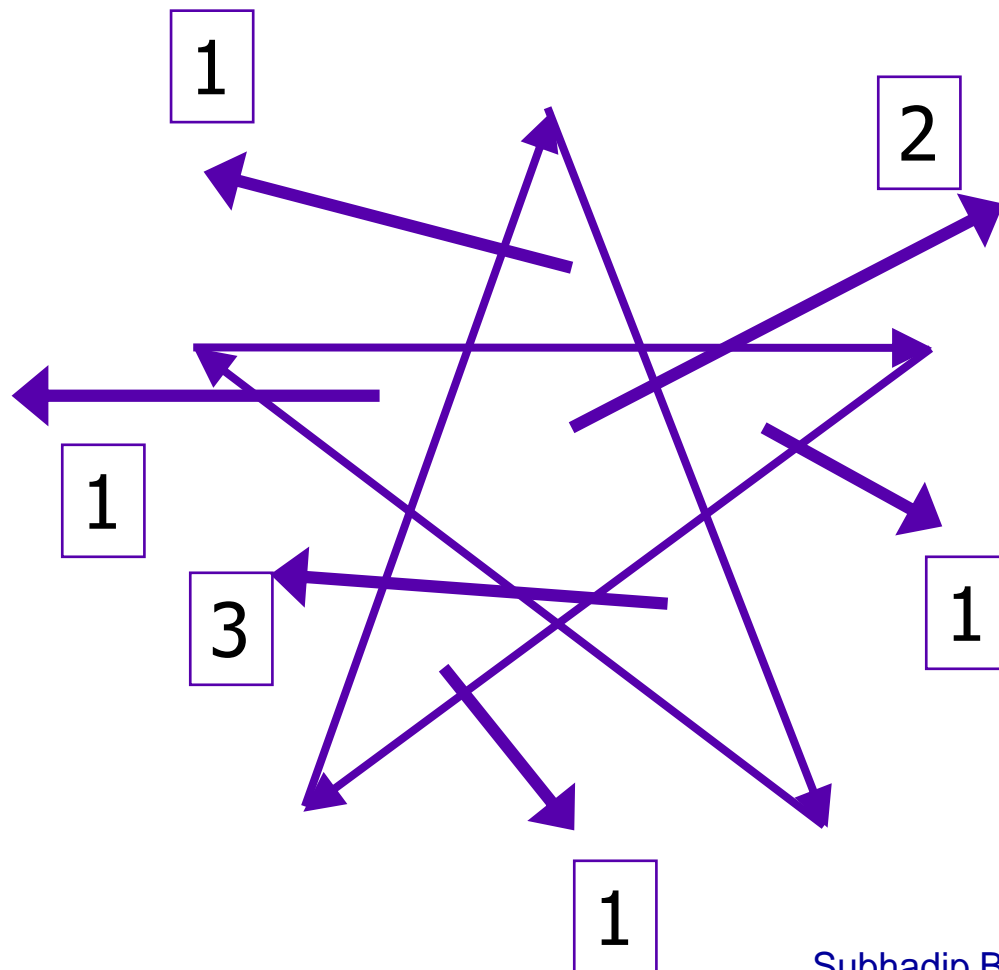
Count edges crossed

- Odd count means interior
- Even count means exterior





Odd-Even Result





Another approach: Nonzero Winding Rule

Choose a point

Draw ray to a distant point

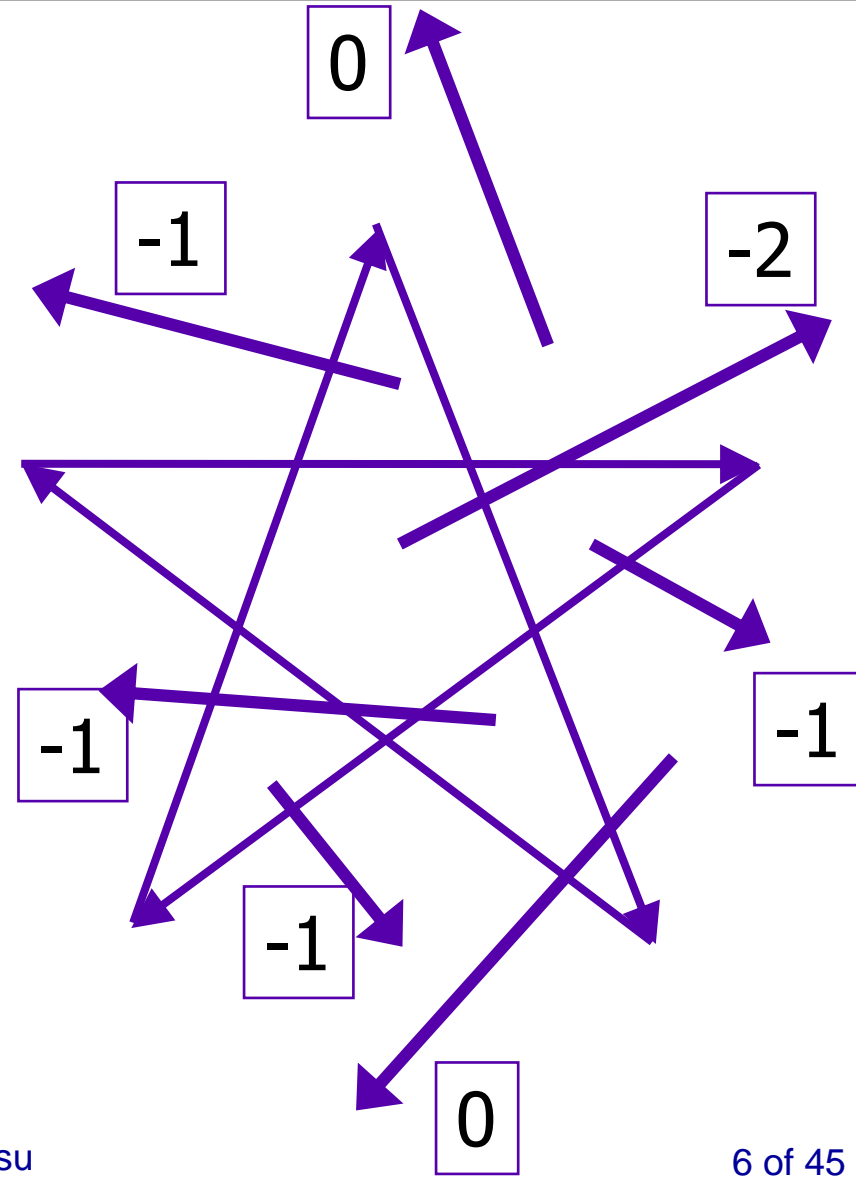
- Don't intersect any vertices

Consider edges crossed
(right hand rule)

- Subtract 1 when ray to edge is clockwise
- Add 1 when ray to edge is counter-clockwise

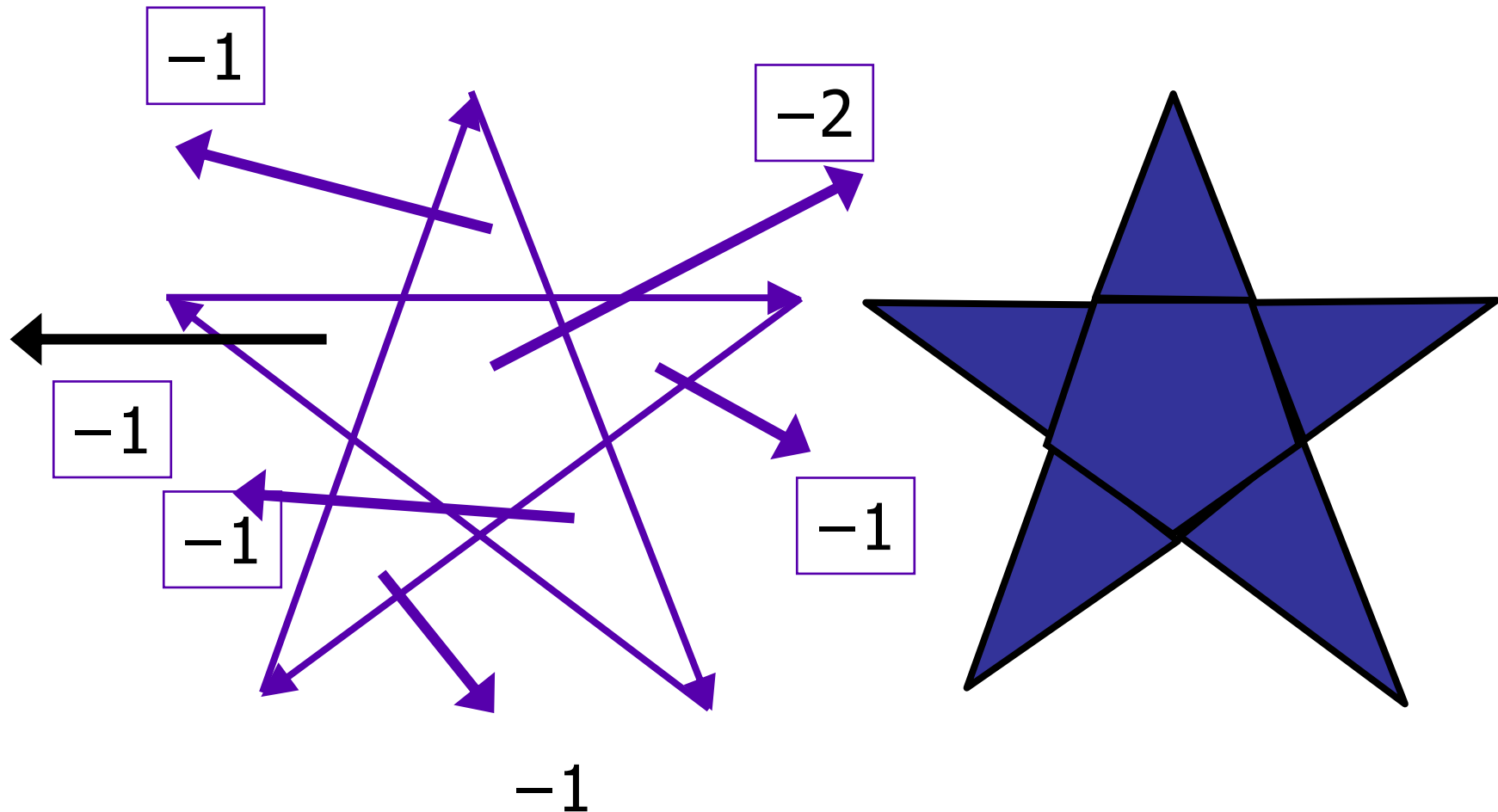
Nonzero count means interior

- Count = "Winding number"



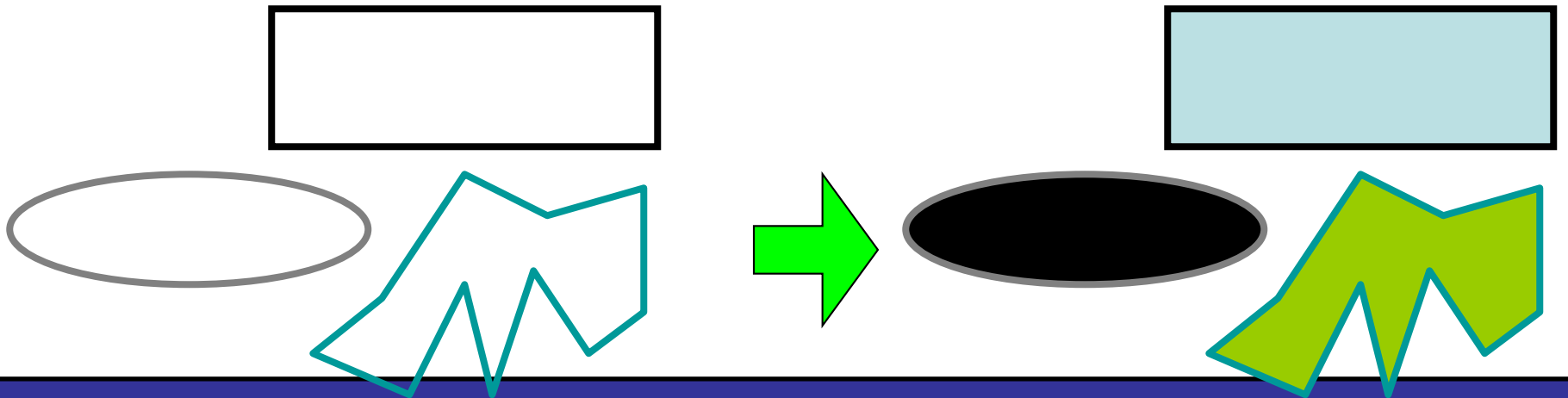


Nonzero Winding Example



Filling Graphical Shapes

Part - 2



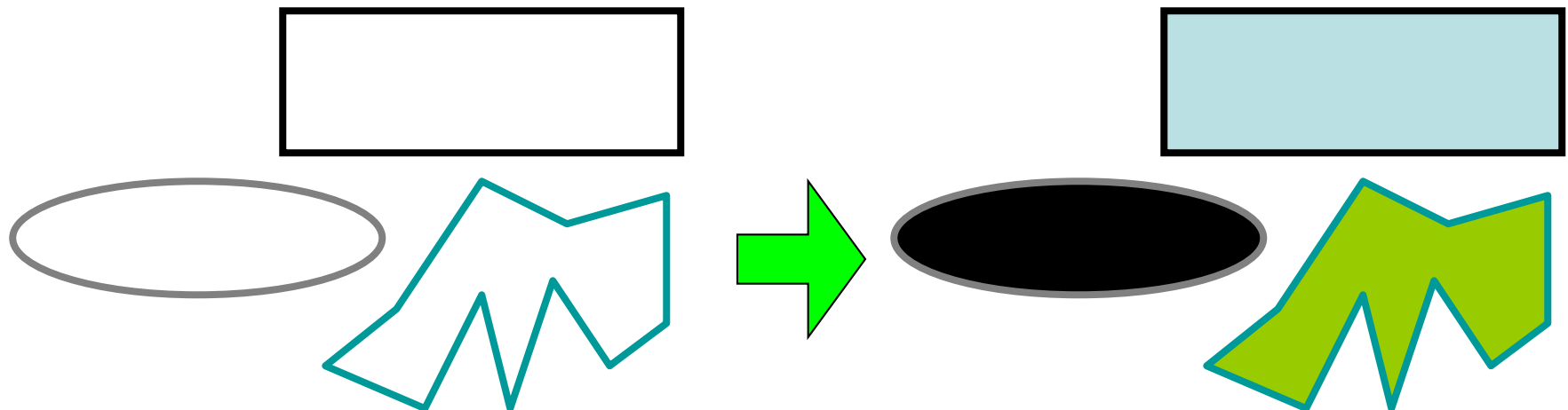


We know two approaches for inside/outside determination

Odd-even rule

- “Alternate” rule in MS API documentation

Non-zero winding rule

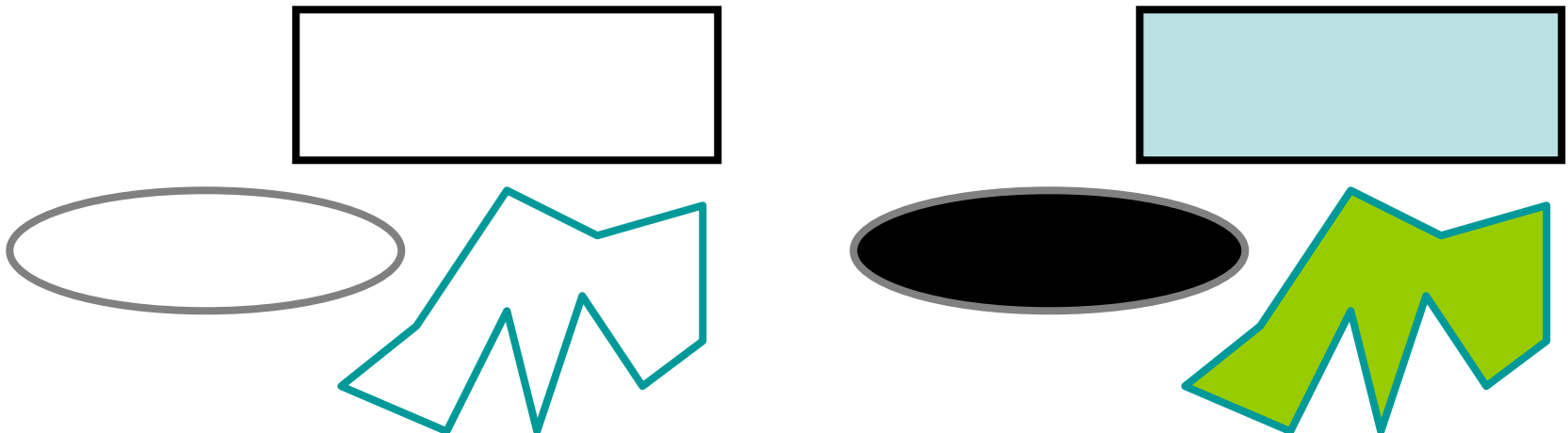




So how do we employ these rules?

One approach

- Vector edge-based fill algorithm

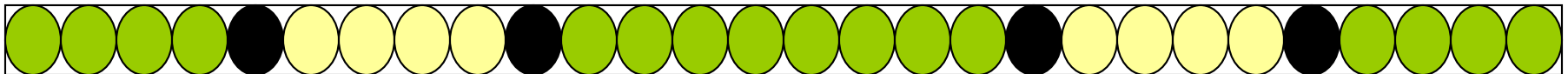
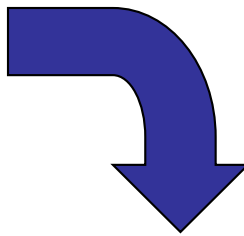
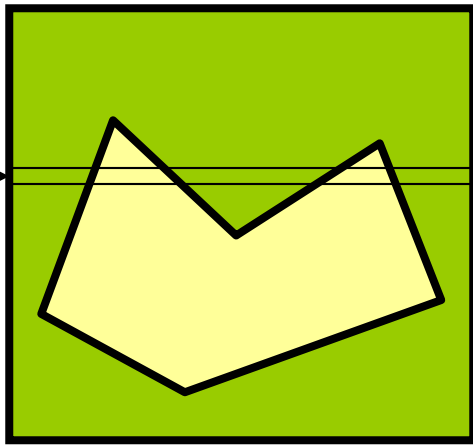




The Scan Line Approach

Scan Line

the rasterized line segment that forms a horizontal slice of the image





Fills one scan line at a time using either the Odd/Even or Non-Zero Winding rule

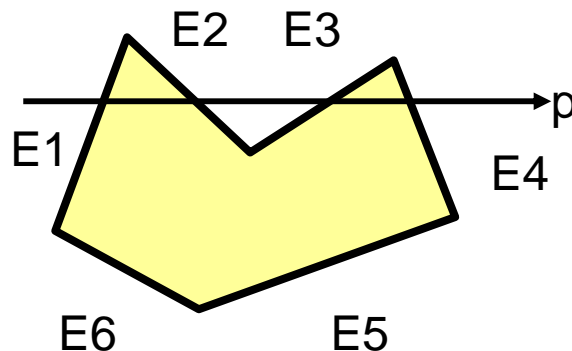
Assumption:

The polygon region is *closed*: image has edge pixels on each scan line ()

Requirement

ray/edge intersection points can be calculated

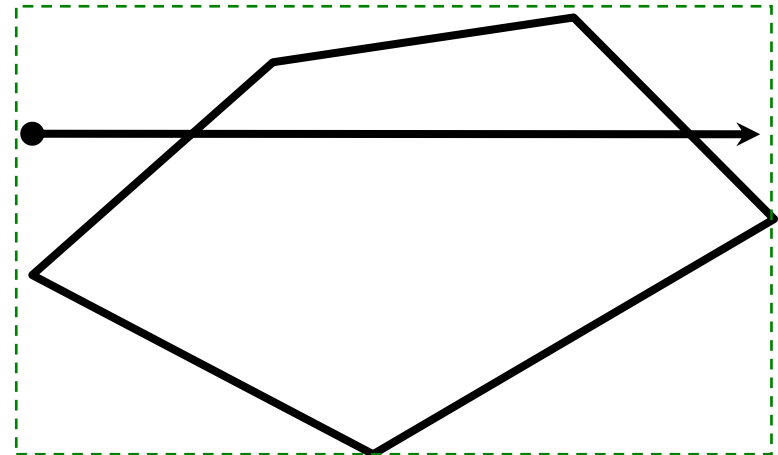
- from polygon edge & ray equations





For each scan line

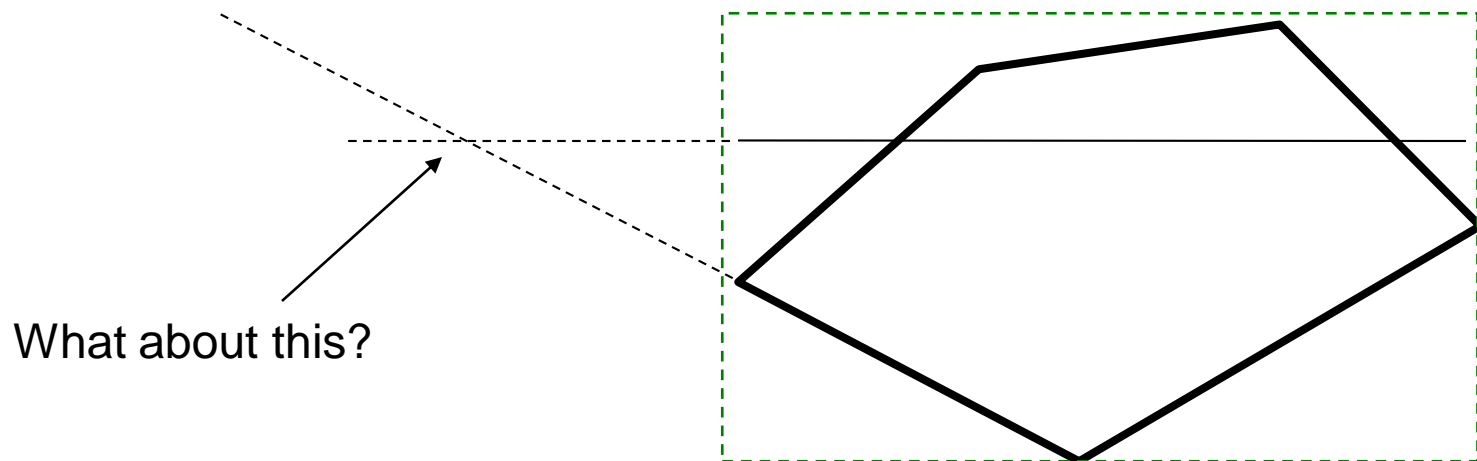
1. Start outside (U.L. of bounding rectangle)
2. Scan line from left to right
3. Determine the intersection with all boundaries
4. Sort the intersection points
5. At each intersection, compute in/out





For each scan line

1. Start outside (U.L. of bounding rectangle)
2. Scan line from left to right
3. **Determine the intersection with all boundaries**
4. Sort the intersection points
5. At each intersection, toggle in/outness





Parametric Line equations – one way to determine intersections

For line E from (x_1, y_1) to (x_2, y_2)

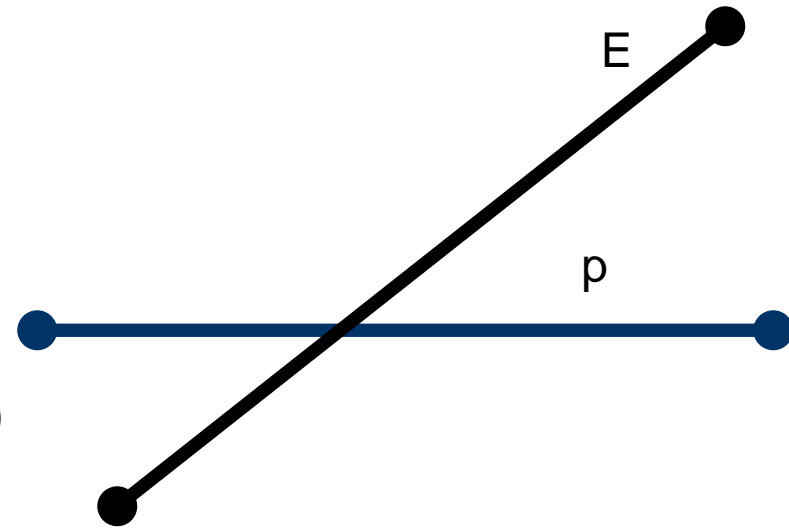
$$x = x_1 + (x_2 - x_1) * t, t \in [0, 1]$$

$$y = y_1 + (y_2 - y_1) * t$$

For a line p from (x_3, y_3) to (x_4, y_4)

$$x = x_3 + (x_4 - x_3) * u, u \in [0, 1]$$

$$y = y_3 + (y_4 - y_3) * u$$





Parametric Line equations

At the intersection

$$x_3 + (x_4 - x_3) * u = x_1 + (x_2 - x_1) * t$$

$$y_3 + (y_4 - y_3) * u = y_1 + (y_2 - y_1) * t$$

2 equations, 2 unknowns (u and t)



Exercise

At the intersection

$$x_3 + (x_4 - x_3) * u = x_1 + (x_2 - x_1) * t$$

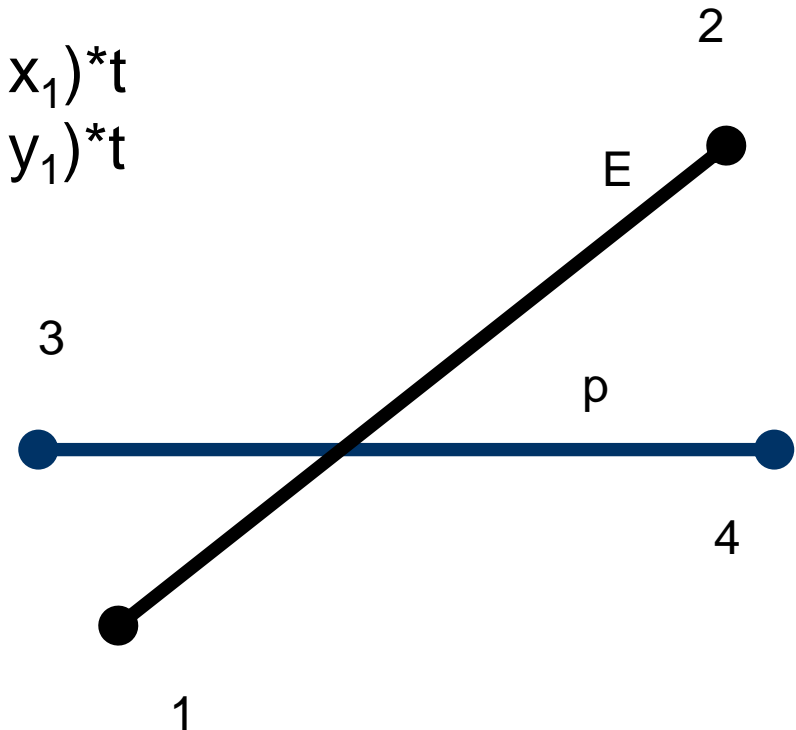
$$y_3 + (y_4 - y_3) * u = y_1 + (y_2 - y_1) * t$$

$$x_1 = 100, y_1 = 100$$

$$x_2 = 200, y_2 = 200$$

$$x_3 = 150, y_3 = 150$$

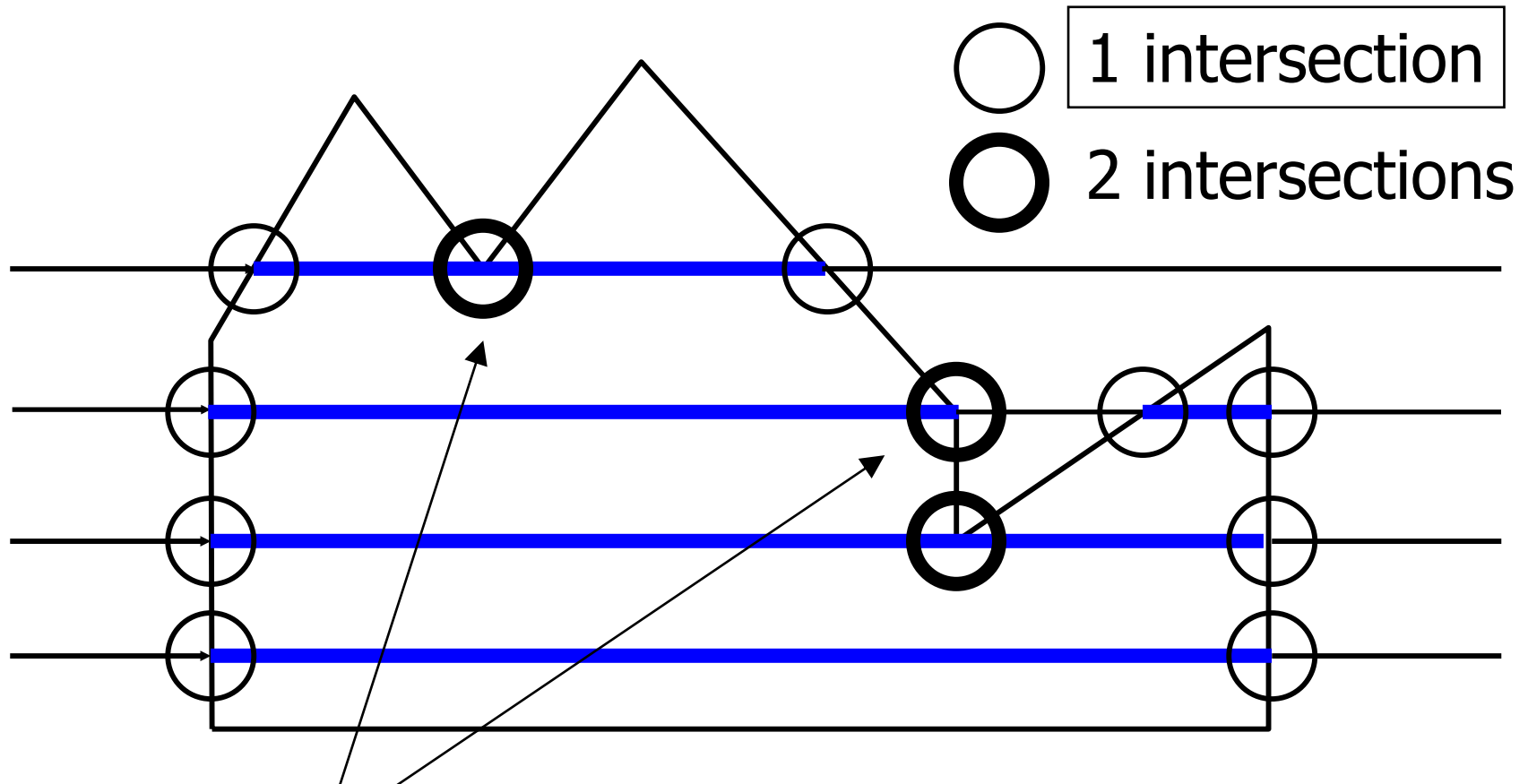
$$x_4 = 200, y_4 = 150$$



What are u and t at the intersection?

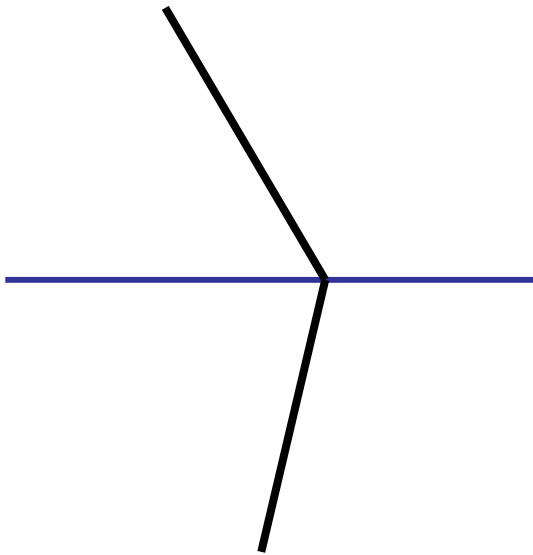


Vertex Complications

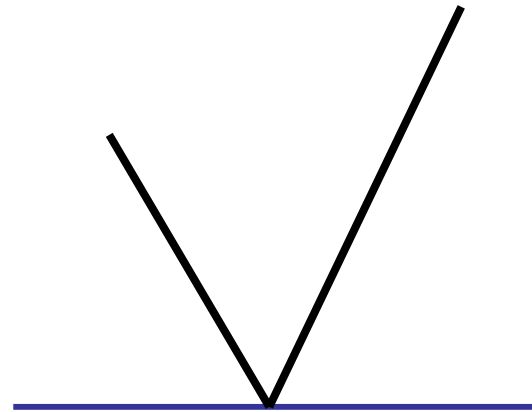




Vertex Intersection Types



Inside/outside
change

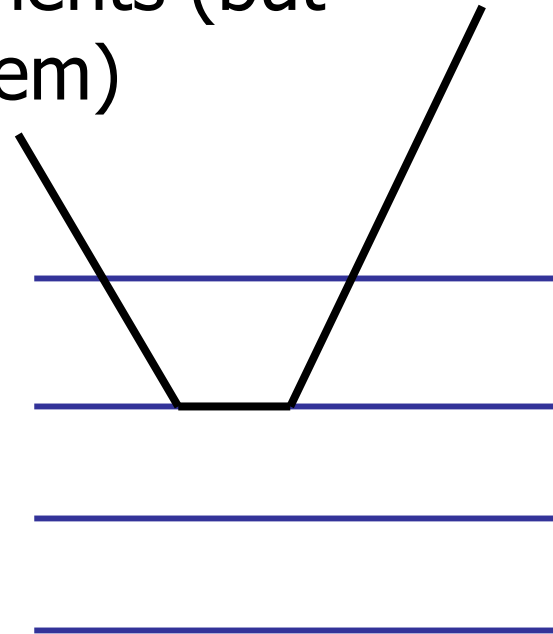
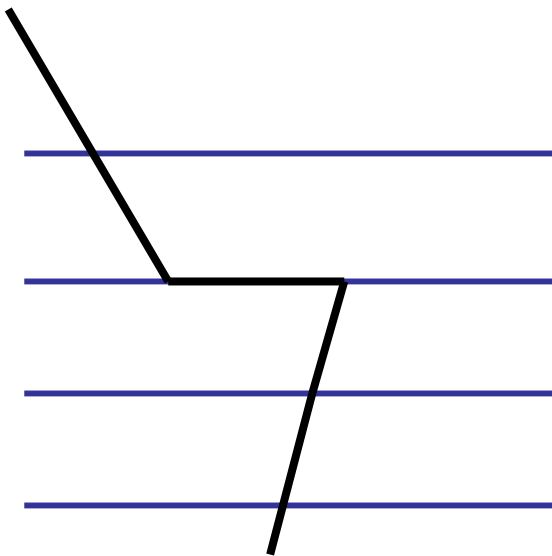


No change



Horizontal edge situation

Ignore horizontal segments (but
don't fill over them)



Check monotonicity of adjacent edges



Calculating Intersections is expensive

All edge intersections must be computed for every scan line

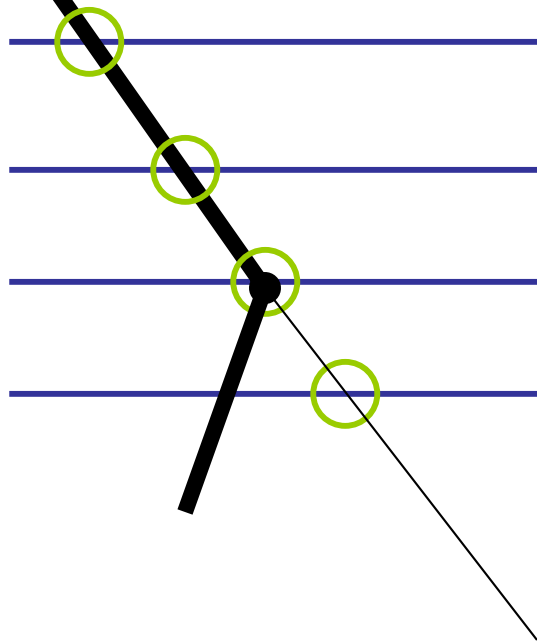
Alternate: Coherence

- For linear segments
 - Each scan line is similar to previous
 - Intersections can be calculated incrementally



Incremental Calculation

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



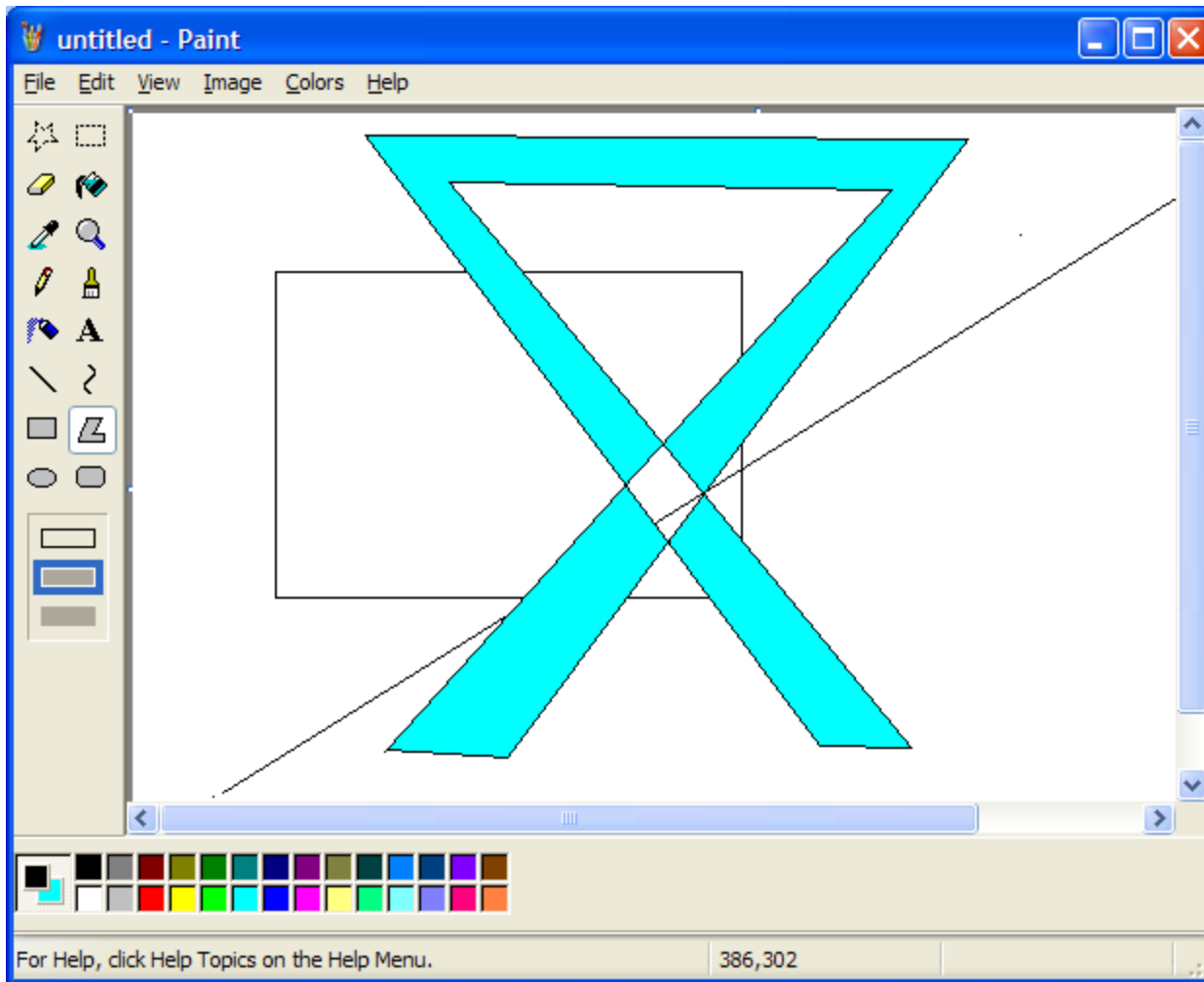
$$y_{k+1} - y_k = 1$$

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

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

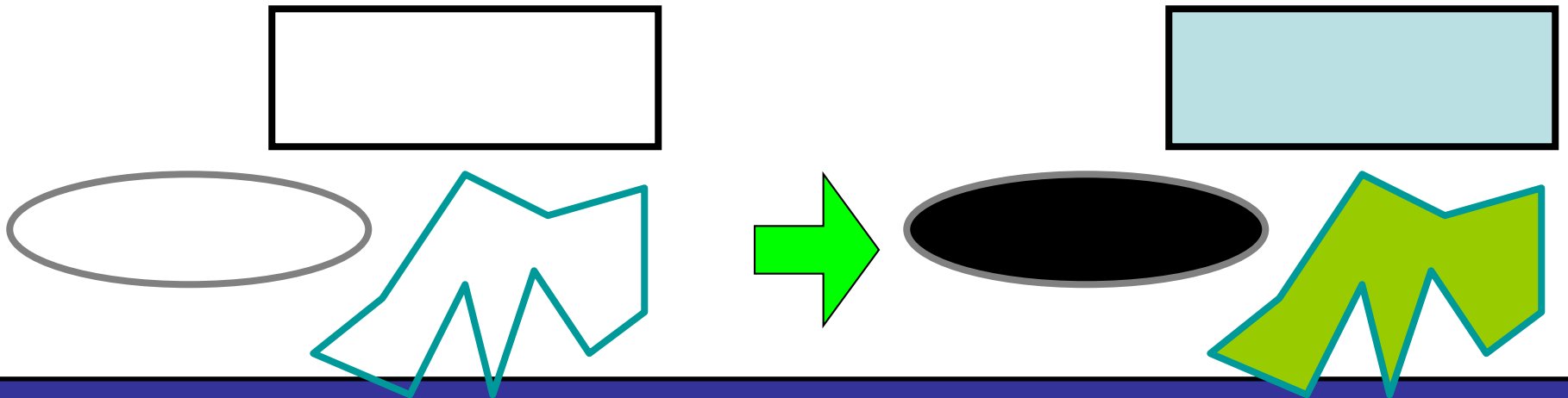


Scanline fill MS Paint example



Filling Graphical Shapes

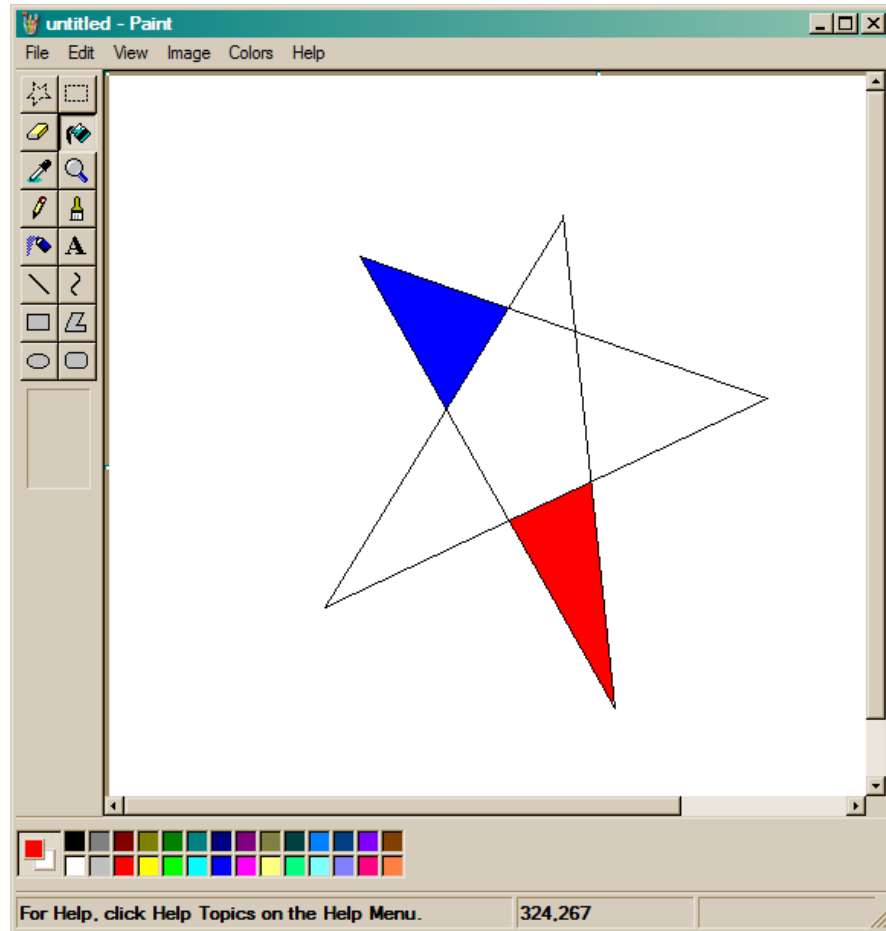
Pixel-based methods





Boundary Fill

MS Paint

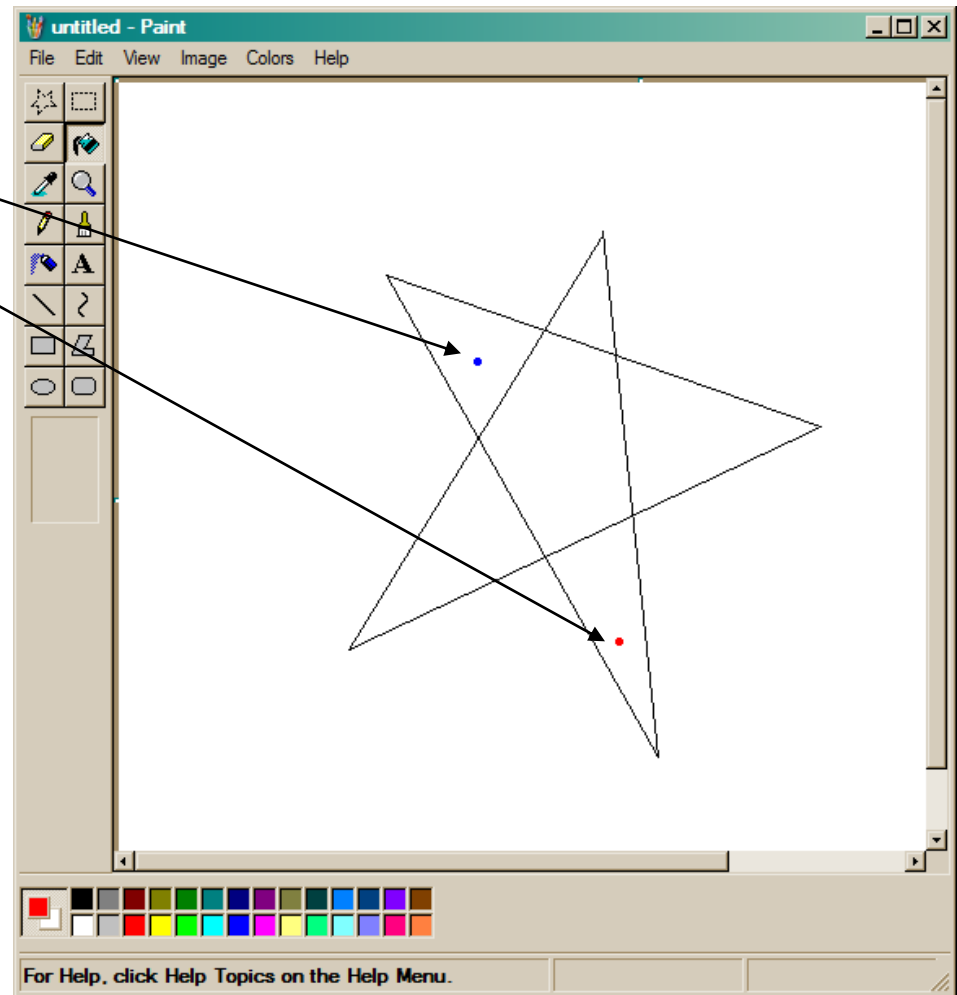




Boundary Fill

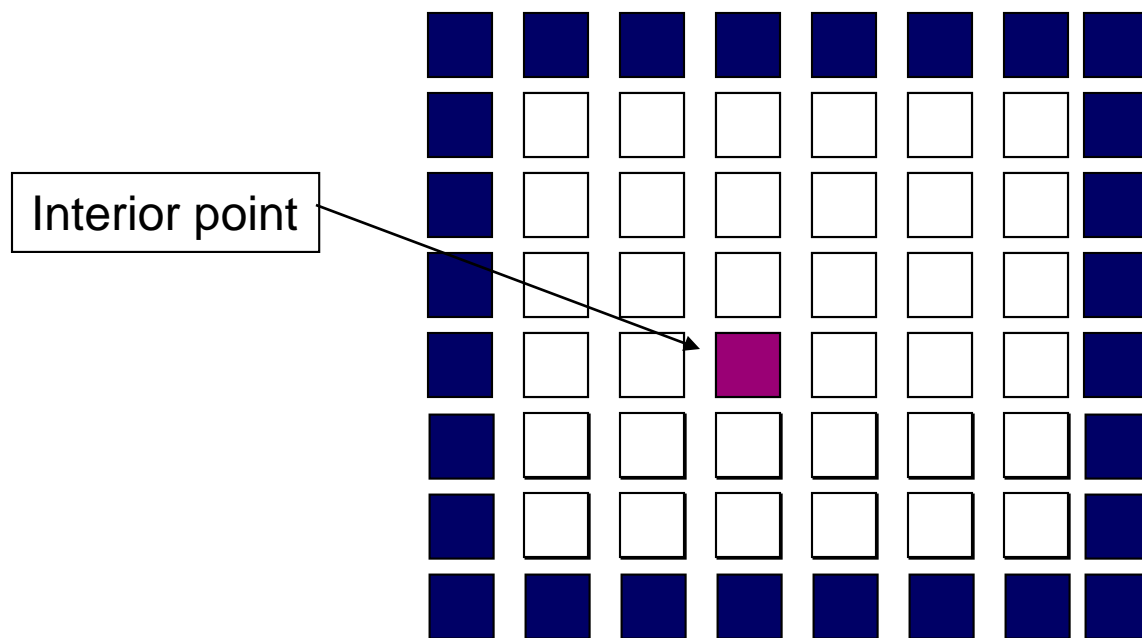
Start at interior point
“Paint” interior
outward toward
boundary

- How?
- Note: Boundary encounter determined by boundary pixel color





Boundary fill





Boundary Fill Algorithm

Don't fill if current position is:

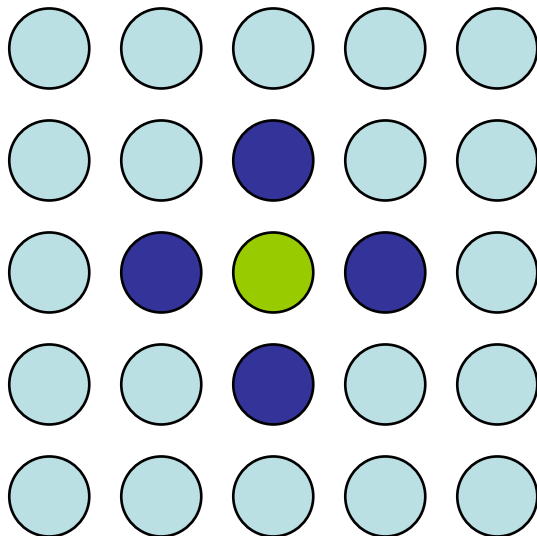
- Boundary color
- Current fill color

Otherwise

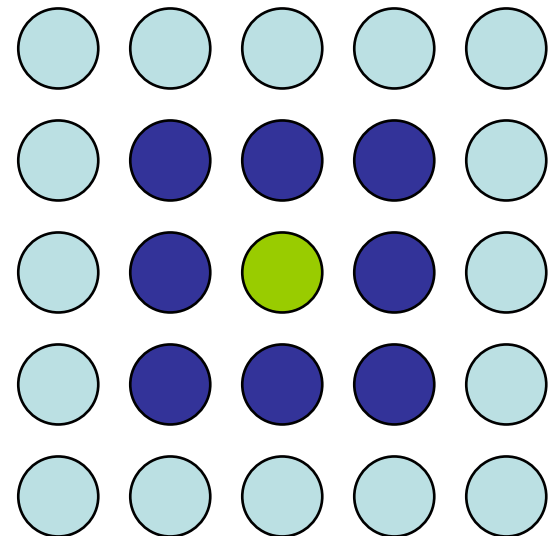
- Set fill color
- **Recursively** try neighbors
 - North, East, South, West
 - Could also be NSEW, NEWS, etc.
 - Each neighbor recursively performs algorithm until “stop”



Boundary Fill Patterns



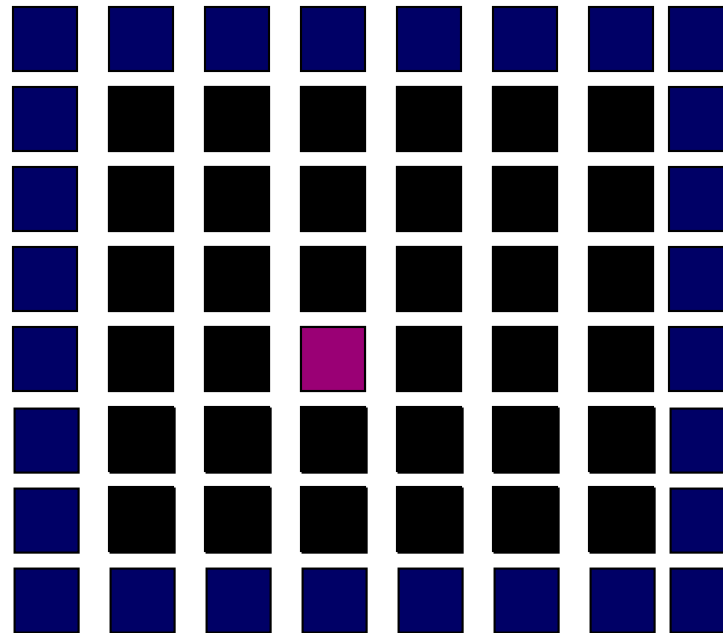
4-connected



8-connected



4-connected EWNS boundary fill





Boundary Fill Algorithm (cont.)

```
void BoundaryFill4(int x, int y,  
                  color newcolor, color edgecolor)  
{  
    int current;  
    current = ReadPixel(x, y);  
    if(current != edgecolor && current != newcolor)  
    {  
        BoundaryFill4(x+1, y, newcolor, edgecolor);  
        BoundaryFill4(x-1, y, newcolor, edgecolor);  
        BoundaryFill4(x, y+1, newcolor, edgecolor);  
        BoundaryFill4(x, y-1, newcolor, edgecolor);  
    }  
}
```



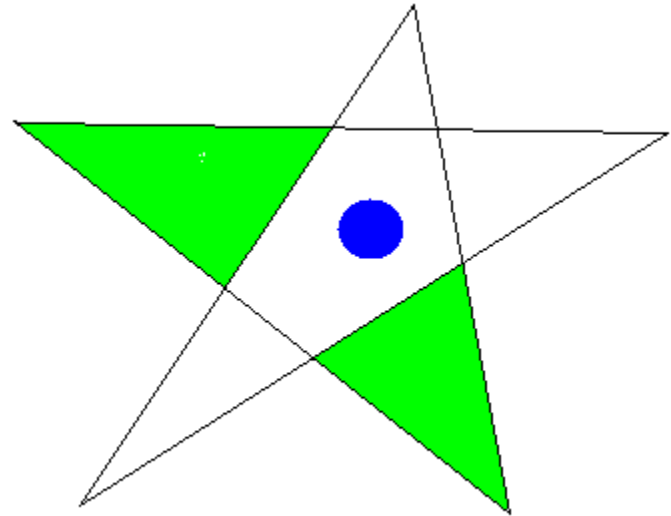
Boundary Fill Problems

Recursive algorithm

- Stack space issue

Recursion stops only on

- Boundary color
- Current fill color



What if some other fill color is already present?

- i.e. pixels already in area to be filled



Similar to boundary fill

But replaces “interior” color (i.e. not boundary)

- Floods through an area
- Fill area must be - initially – a consistent color
- Only one significant color is filled over

Paint bucket tool

- MS Paint actually uses flood fill



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);
    }
}
```



Fill Algorithm Summary

Vector edge model

- Scan line fill
 - Even-odd or non-zero winding

Pixel model

- 4- or 8- connected recursion
- Boundary fill: overwrites all but boundary or current fill color
- Flood fill: overwrites only initially selected color



How do we simulate intermediate shades of grey in an 8-bit image?

How can we use small number of coloured inks to simulate the huge range of colours possible in a 24-bit image?



Printing Grayscale Images

Problem: How to print a grayscale image when the only color ink available is black (the ink) and white (the paper)?



✂ This is the problem faced by **newspapers** and any other print **media**.

✂ Newspapers print at 80-100 DPI (dots per inch)

✂ Magazines print at 200-300 DPI

✂ **Color** printing is a more complicated variant of this problem.

✂ The solution is to use a technique known as **halftoning**.

✂ The process of generating binary pattern of black and white dots from an image.



Halftoning

Photographs are images with continuous shades of black, white, and gray. These shades are *tones*.

Halftoning is a technique, originally developed for the printing industry, to reproduce continuous tone images using printing techniques capable of black and white only, not shades of gray.



Need for Digital Image Halftoning

Examples of reduced grayscale/color resolution

- Laser and inkjet printers (**\$9.3B revenue in 2001 in US**)
- Facsimile machines
- Low-cost liquid crystal displays

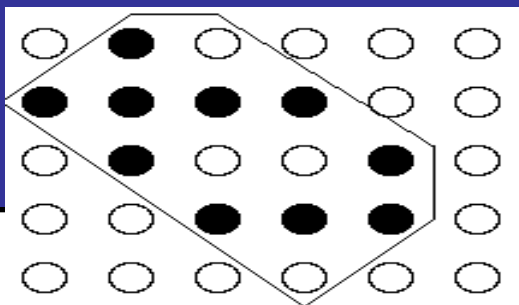
Halftoning is wordlength reduction for images

- Grayscale: 8-bit to 1-bit (**binary**)
- Color displays: 24-bit RGB to 12-bit RGB (**e.g. PDA/cell**)
- Color displays: 24-bit RGB to 8-bit RGB (**e.g. cell phones**)
- Color printers: 24-bit RGB to CMY (**each color binarized**)

Halftoning tries to reproduce full range of gray/ color while preserving quality & spatial resolution.



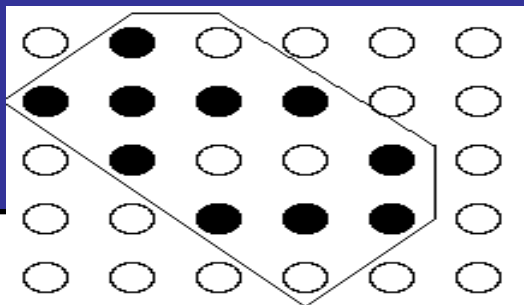
Halftoning



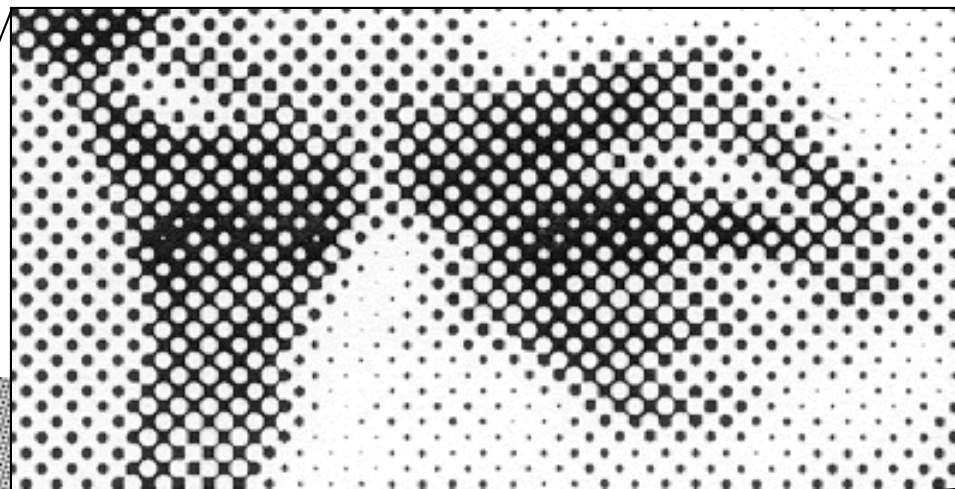
- The process of transforming a grayscale image to a halftone.
- Create the **illusion** of gray-scale by varying the average dot density in local regions of the image
- Dots are always something that can be counted and it must be possible to hold a measuring stick to them.
- Takes advantage of the fact that the eye **integrates** intensity of small image regions.
- Sacrifices **spatial** resolution for **gray-scale** resolution (unless the output device can be over-sampled as most ink-jet printers are these days)
- Spatial refers to image itself, based on direct manipulation pixels.



Halftoning



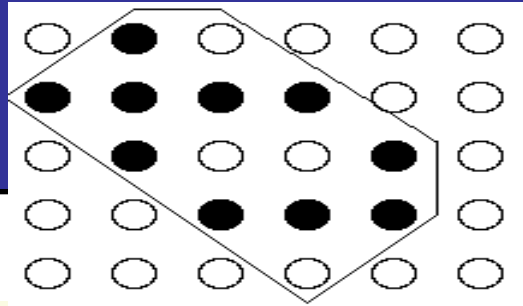
Consider a grayscale image (left) which is halftoned (right) for printing. The right image looks like a grayscale image but is actually only black and white!



Zooming in on the image reveals “pixel” sizes of differing sizes and shapes.



Halftoning



- Detail is rendered by local modulation of this texture.



- The perception of levels of gray intermediate to black or white depends on a local average of the binary texture.





Digital Halftoning

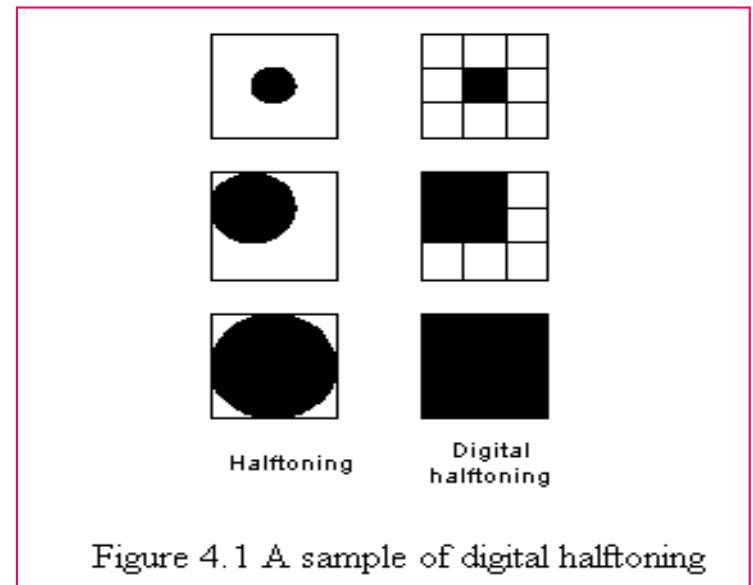
The previous example is from the domain of **print media** which uses physical filters, lights, and film to generate the halftoned images.

Digital halftoning cannot be done this way since digital images consist of identically shaped pixels (usually rectangular) which are either black or white.

The main problem is “***Should this pixel be white or black?***”

Possible solution to this problem

- Bi-level Thresholding





Bilevel Thresholding

Re-quantize the image using a 1 bit color

If the gray-level of a pixel is less than some **threshold** value then set the output to black otherwise set the output to white.

- The **threshold** value may be the “center of the available gray-scale range”
- The **threshold** value may by the “average of all pixels in the image”

$$g(k, l) = \begin{cases} 1 & \text{if } f(k, l) \geq T \\ 0 & \text{otherwise} \end{cases}$$

The T value can be choosen e.g
inspecting the histogram image f .
or

$$T(k, l) = \frac{1}{2}(\max\{f(k, l)\} + \min\{f(k, l)\})$$



Bi-level Thresholding Example



Original Image



Absolute Threshold



Adaptive Threshold