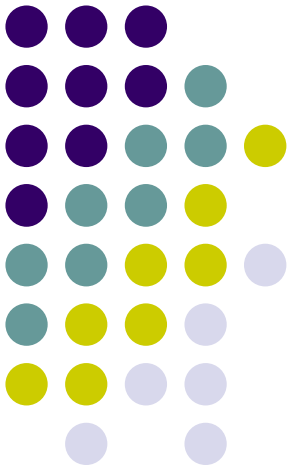
Computer Graphics (CS 543) Lecture 10: Rasterization and Antialiasing

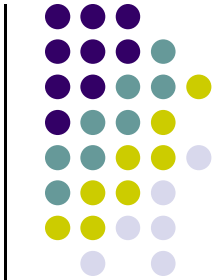
Prof Emmanuel Agu

Computer Science Dept.

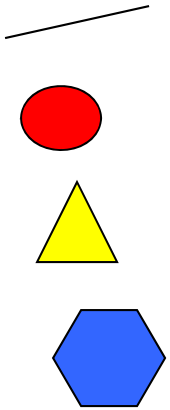
Worcester Polytechnic Institute (WPI)



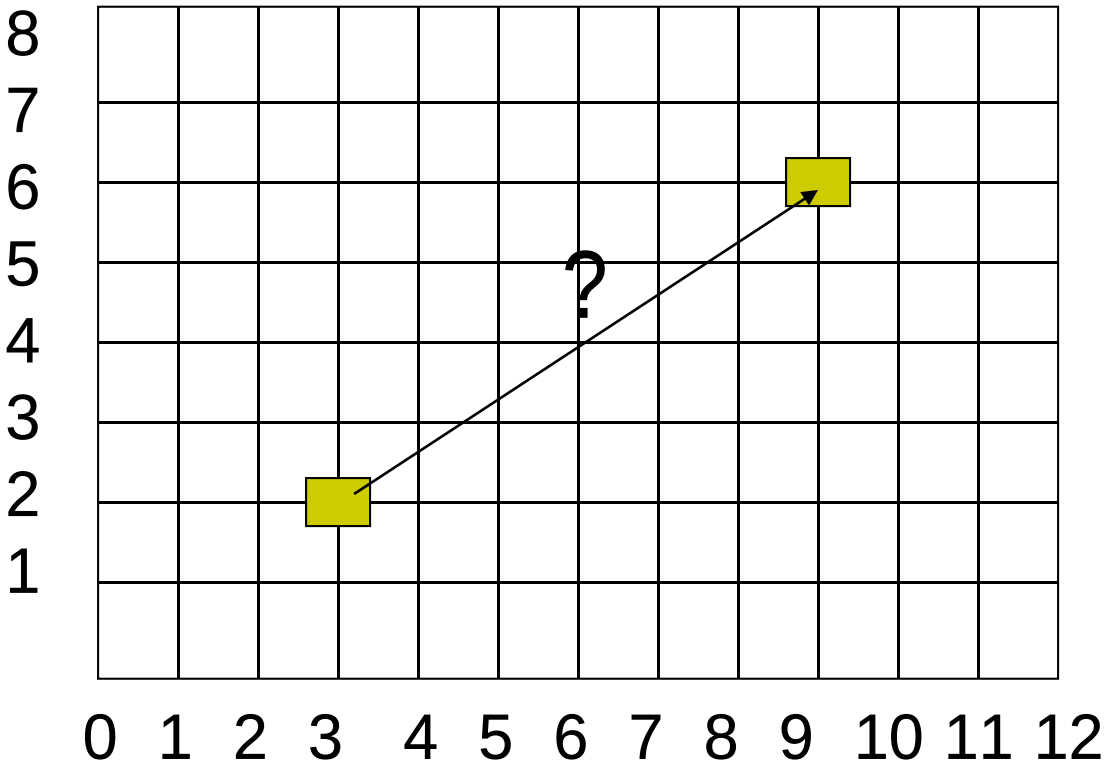
Recall: Rasterization



- Rasterization (scan conversion)
  - Determine which pixels that are inside primitive specified by a set of vertices
- Implemented by graphics hardware
- Rasterization algorithms
  - Lines
  - Circles
  - Triangles
  - Polygons



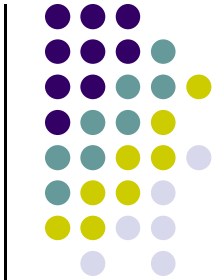
Recall: Line Drawing Algorithm



Line: (3,2) -> (9,6)

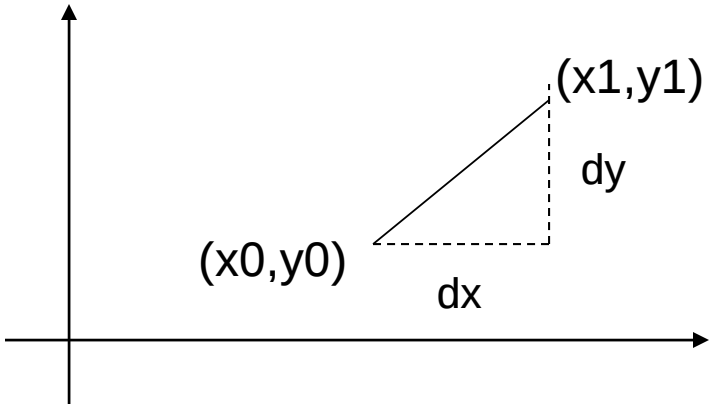
Which intermediate pixels to turn on?

Line Drawing Algorithm

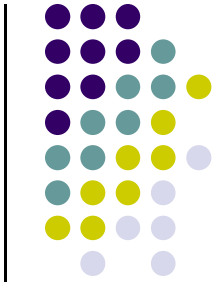


- Slope-intercept line equation
  - y = mx + b
  - Given two end points (x0,y0), (x1, y1), how to compute m and b?

$$m = \frac{dy}{dx} = \frac{y1 - y0}{x1 - x0} \qquad b = y0 - m * x0$$



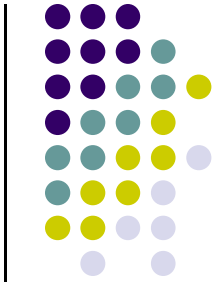
Line Drawing Algorithm



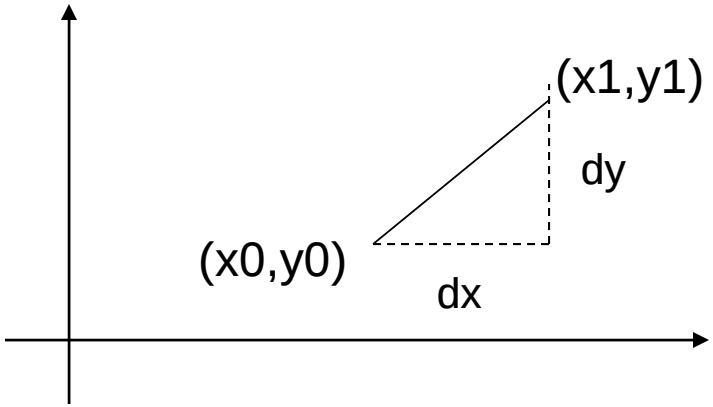
- Numerical example of finding slope m:
  - (Ax, Ay) = (23, 41), (Bx, By) = (125, 96)

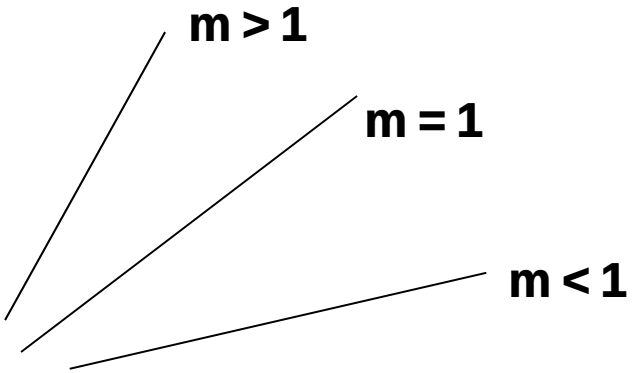
$$m = \frac{By - Ay}{Bx - Ax} = \frac{96 - 41}{125 - 23} = \frac{55}{102} = 0.5392$$

Digital Differential Analyzer (DDA): Line Drawing Algorithm



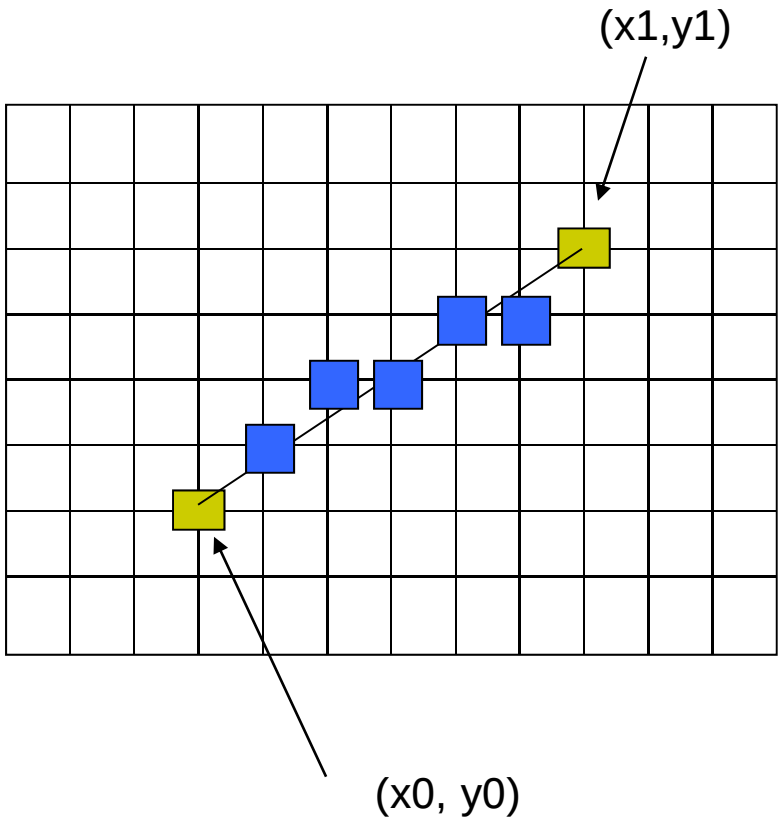
- Walk through the line, starting at (x0,y0)
- Constrain x, y increments to values in [0,1] range
- Case a: x is incrementing faster (m < 1)
  - Step in x=1 increments, compute and round y
- Case b: y is incrementing faster (m > 1)
  - Step in y=1 increments, compute and round x





DDA Line Drawing Algorithm (Case a: m < 1)

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



x = x0 y = y0

Illuminate pixel (x, round(y))

x = x0 + 1 y = y0 + 1 * m

Illuminate pixel (x, round(y))

x = x + 1 y = y + 1 * m

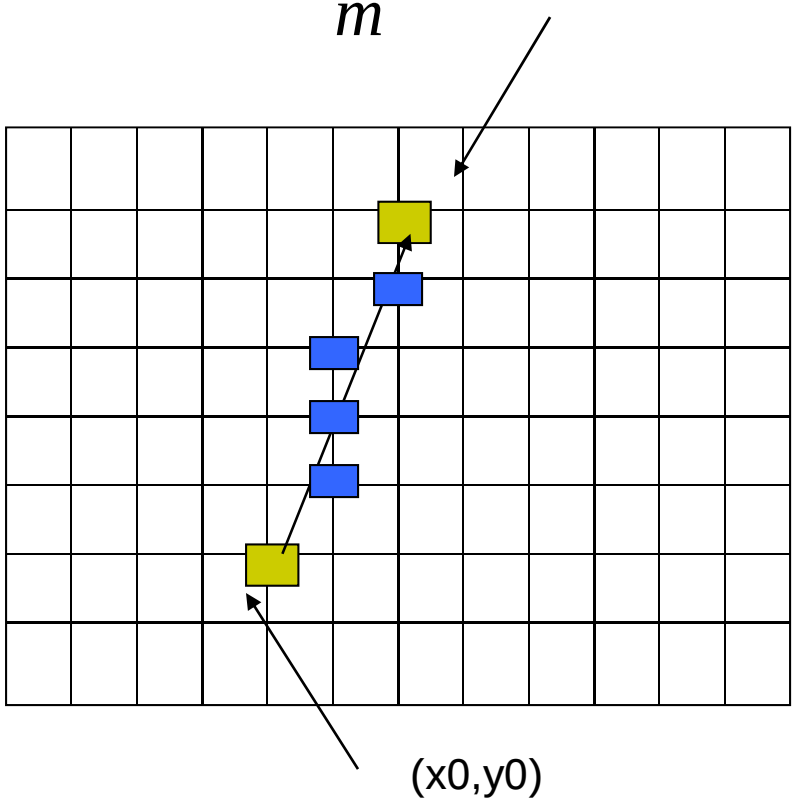
Illuminate pixel (x, round(y))

...

Until x == x1

DDA Line Drawing Algorithm (Case b: m > 1)

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



x = x0 y = y0

Illuminate pixel (round(x), y)

y = y0 + 1 x = x0 + 1 * 1/m

Illuminate pixel (round(x), y)

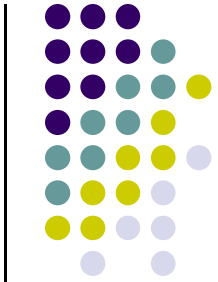
y = y + 1 x = x + 1 /m

Illuminate pixel (round(x), y)

...

Until y == y1

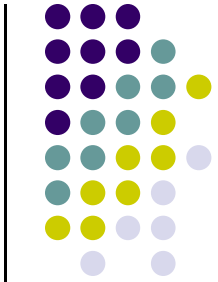
DDA Line Drawing Algorithm Pseudocode



```
compute m;
if m < 1:
{
   float y = y0;                // initial value
   for(int x = x0;x <= x1; x++, y += m)
                                setPixel(x, round(y));
}
else // m > 1
{
   float x = x0;        // initial value
   for(int y = y0;y <= y1; y++, x += 1/m)
               setPixel(round(x), y);
}
```

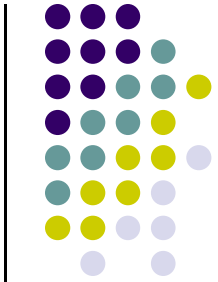
- Note: **`setPixel(x, y)`** writes current color into pixel in column x and row y in frame buffer

Line Drawing Algorithm Drawbacks

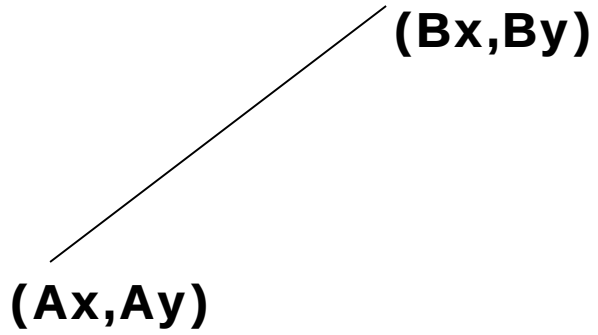


- DDA is the simplest line drawing algorithm
  - Not very efficient
  - Round operation is expensive
- Optimized algorithms typically used.
  - Integer DDA
  - E.g.Bresenham algorithm (Hill)
- Bresenham algorithm
  - Incremental algorithm: current value uses previous value
  - Integers only: avoid floating point arithmetic
  - Several versions of algorithm: we'll describe midpoint version of algorithm

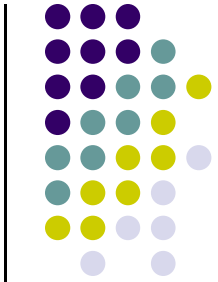
Bresenham's Line-Drawing Algorithm



- Problem: Given endpoints (Ax, Ay) and (Bx, By) of a line, want to determine best sequence of intervening pixels
- First make two simplifying assumptions (remove later):
  - (Ax < Bx) and
  - (0 < m < 1)
- Define
  - Width W = Bx – Ax
  - Height H = By - Ay

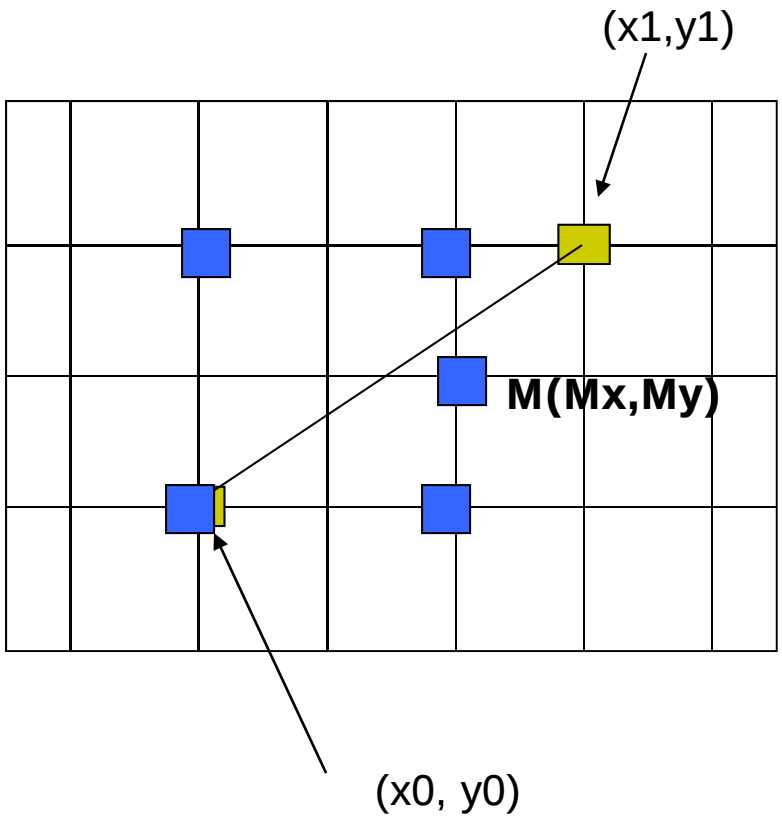


Bresenham's Line-Drawing Algorithm



- Based on assumptions:
  - W, H are +ve
  - H < W
- As x steps in +1 increments, y incr/decr by <= +/–1
- y value sometimes stays same, sometimes increases by 1
- Midpoint algorithm determines which happens

Bresenham's Line-Drawing Algorithm



What Pixels to turn on or off?

Consider pixel midpoint M(Mx, My)

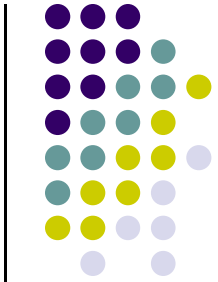
M = (x0 + 1, Y0 + ½)

Build equation of line through and compare to midpoint

If midpoint is above line, y stays same
If midpoint is below line, y increases + 1

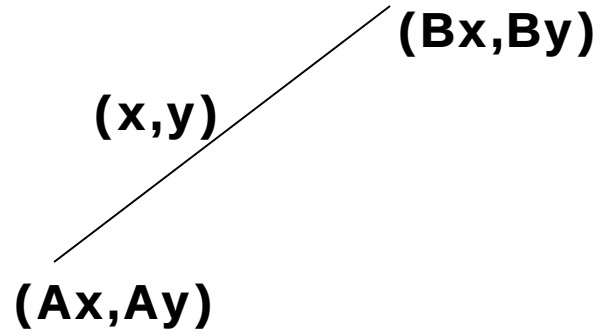
...

Bresenham's Line-Drawing Algorithm



- Using similar triangles:

$$\frac{y - Ay}{x - Ax} = \frac{H}{W}$$

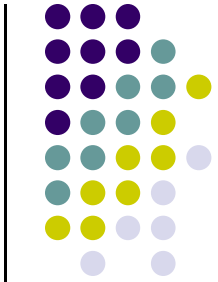


$$H(x - Ax) = W(y - Ay)$$
$$-W(y - Ay) + H(x - Ax) = 0$$

- Above is ideal equation of line through (Ax, Ay) and (Bx, By)
- Thus, any point (x,y) that lies on ideal line makes eqn = 0
- Double expression (to avoid floats later), and give it a name,

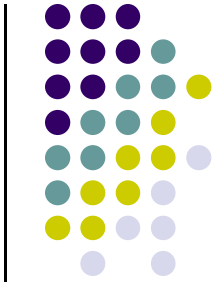
$$F(x,y) = -2W(y - Ay) + 2H(x - Ax)$$

Bresenham's Line-Drawing Algorithm



- So, F(x,y) = -2W(y – Ay) + 2H(x – Ax)
- Algorithm, If:
  - F(x, y) < 0, (x, y) above line
  - F(x, y) > 0, (x, y) below line
- Hint: F(x, y) = 0 is on line
- Increase y keeping x constant, F(x, y) becomes more negative

Bresenham's Line-Drawing Algorithm

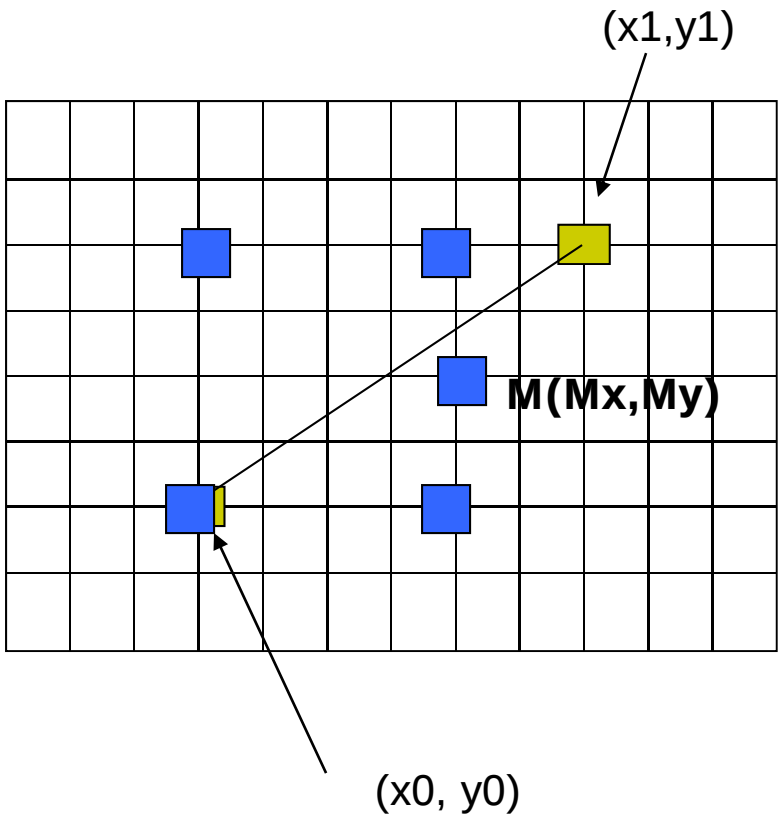


- Example: to find line segment between (3, 7) and (9, 11)

$$F(x,y) = -2W(y - Ay) + 2H(x - Ax)$$
$$= (-12)(y - 7) + (8)(x - 3)$$

- For points on line. E.g. (7, 29/3), F(x, y) = 0
- A = (4, 4) lies below line since F = 44
- B = (5, 9) lies above line since F = -8

Bresenham's Line-Drawing Algorithm



What Pixels to turn on or off?

Consider pixel midpoint M(Mx, My)

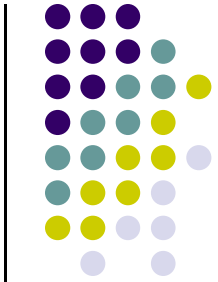
M = (x0 + 1, Y0 + ½)

If F(Mx,My) < 0, M lies above line,
shade lower pixel (same y as before)

If F(Mx,My) > 0, M lies below line,
shade upper pixel

...

Can compute F(x,y) incrementally



Initially, midpoint M = (Ax + 1, Ay + ½)

F(Mx, My) = -2W(y – Ay) + 2H(x – Ax)

= 2H – W

Can compute F(x,y) for next midpoint incrementally

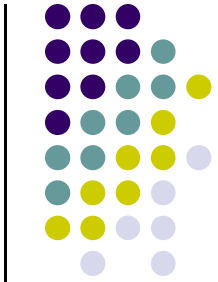
If we increment x + 1, y stays same, compute new F(Mx,My)

F(Mx, My) += 2H

If we increment x +1, y + 1

F(Mx, My) -= 2(W – H)

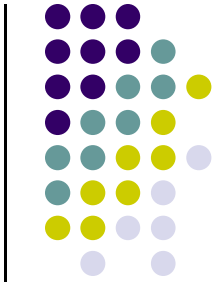
Bresenham's Line-Drawing Algorithm



```
Bresenham(IntPoint a, InPoint b)
{ // restriction: a.x < b.x and 0 < H/W < 1
      int y = a.y, W = b.x – a.x, H = b.y – a.y;
      int F = 2 * H – W;   // current error term
      for(int x = a.x; x <= b.x; x++)
      {
        setpixel at (x, y); // to desired color value
            if F < 0
                F = F + 2H;
            else{
                Y++,  F = F  + 2(H – W)
            }
      }
}
```

- Recall: F is equation of line

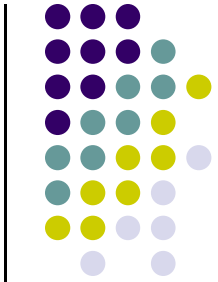
Bresenham's Line-Drawing Algorithm



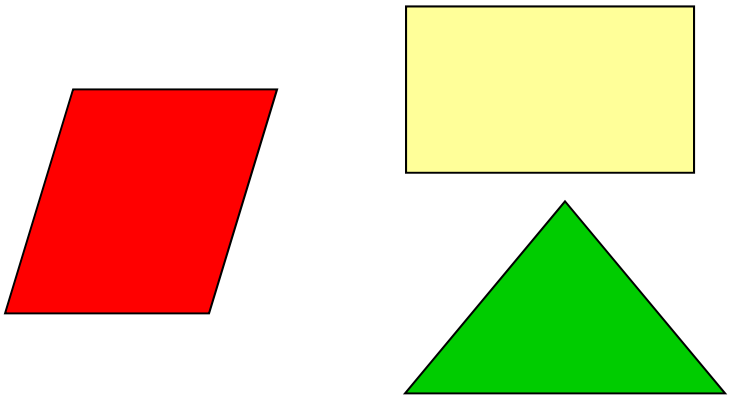
- Final words: we developed algorithm with restrictions

  $0 < m < 1$ and $Ax < Bx$
- Can add code to remove restrictions
  - To get the same line when $Ax > Bx$ (swap and draw)
  - Lines having $m > 1$ (interchange x with y)
  - Lines with $m < 0$ (step x++, decrement y not incr)
  - Horizontal and vertical lines (pretest a.x = b.x and skip tests)

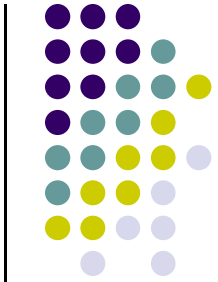
Defining and Filling Regions of Pixels



- Methods of defining region
  - Pixel-defined: specifies pixels in color or geometric range
  - Symbolic: provides property pixels in region must have
  - Examples of symbolic:
    - Closeness to some pixel
    - Within circle of radius *R*
    - Within a specified polygon

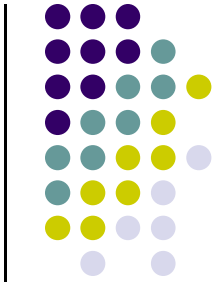


Pixel-Defined Regions

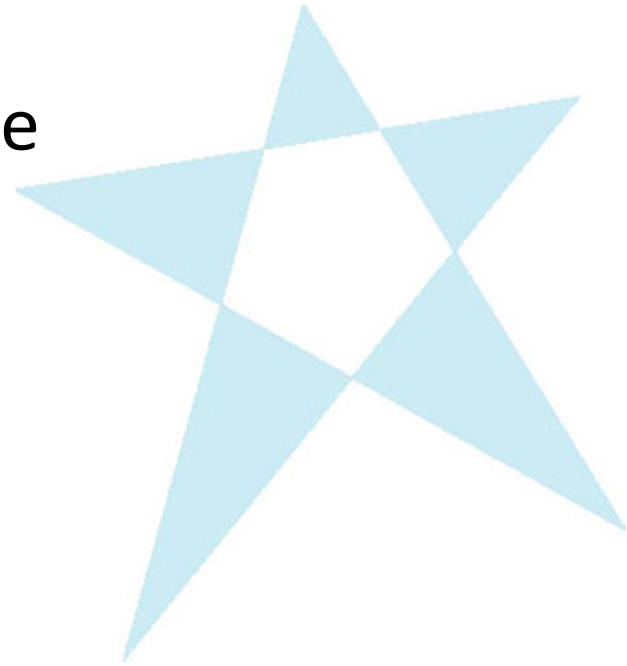


- **Definition:** Region R is the set of all pixels having color C that are connected to a given pixel S
- **4-adjacent:** pixels that lie next to each other horizontally or vertically, NOT diagonally
- **8-adjacent:** pixels that lie next to each other horizontally, vertically OR diagonally
- **4-connected:** if there is unbroken path of 4-adjacent pixels connecting them
- **8-connected:** unbroken path of 8-adjacent pixels connecting them

Polygon Scan Conversion

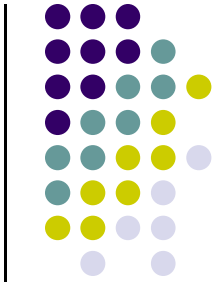


- Scan Conversion = Fill
- How to tell inside from outside
  - Convex easy
  - Nonsimple difficult
  - Odd even test
    - Count edge crossings
  - Winding number

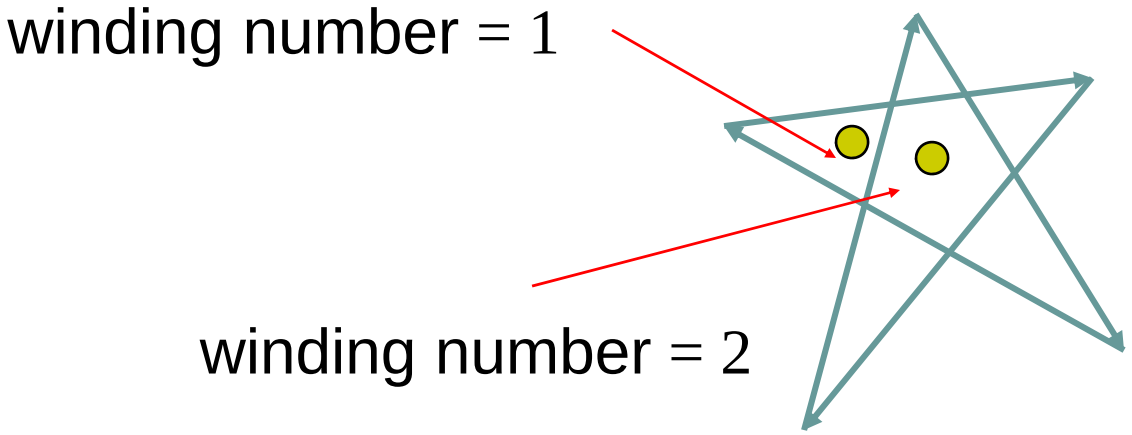


odd-even fill

Winding Number

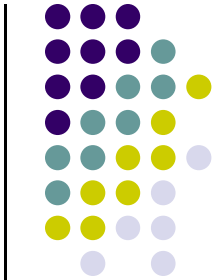


- Count clockwise encirclements of point



- Alternate definition of inside: inside if winding number $\neq 0$

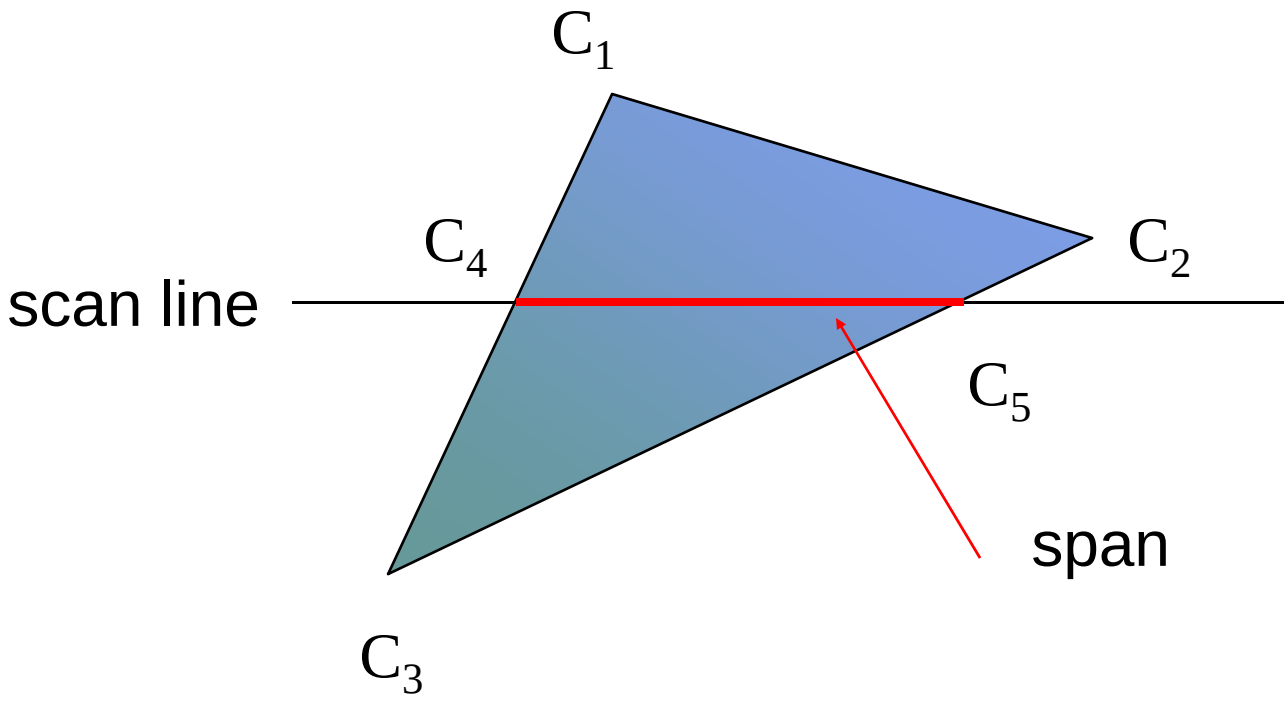
Filling in the Frame Buffer



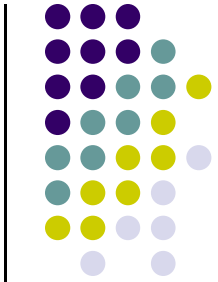
- Fill at end of pipeline
  - Convex Polygons only
  - Nonconvex polygons assumed to have been tessellated
  - Shades (colors) have been computed for vertices (Gouraud shading)
  - Combine with z-buffer algorithm
    - March across scan lines interpolating shades
    - Incremental work small

Using Interpolation

C_1 C_2 C_3 specified by **glColor** or by vertex shading
C_4 determined by interpolating between C_1 and C_2
C_5 determined by interpolating between C_2 and C_3
interpolate between C_4 and C_5 along span

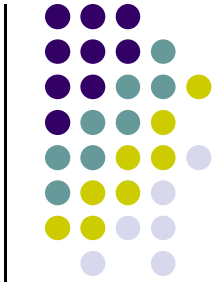


Recursive Flood-Fill Algorithm



- Recursive algorithm
- Starts from initial pixel of color, **`intColor`**
- Recursively set 4-connected neighbors to **`newColor`**
- **Flood-Fill**: floods region with **`newColor`**
- **Basic idea:**
  - start at “seed” pixel (x, y)
  - If (x, y) has color intColor, change it to newColor
  - Do same recursively for all 4 neighbors

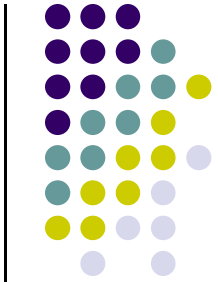
Recursive Flood-Fill Algorithm



- **Note:** getPixel(x,y) used to interrogate pixel color at (x, y)

```
void floodFill(short x, short y, short intColor)
{
   if(getPixel(x, y) == intColor)
   {
        setPixel(x, y);
        floodFill(x – 1, y, intColor); // left pixel
        floodFill(x + 1, y, intColor); // right pixel
        floodFill(x, y + 1, intColor); // down pixel
        floodFill(x, y – 1, intColor); // fill up
   }
}
```

Recursive Flood-Fill Algorithm

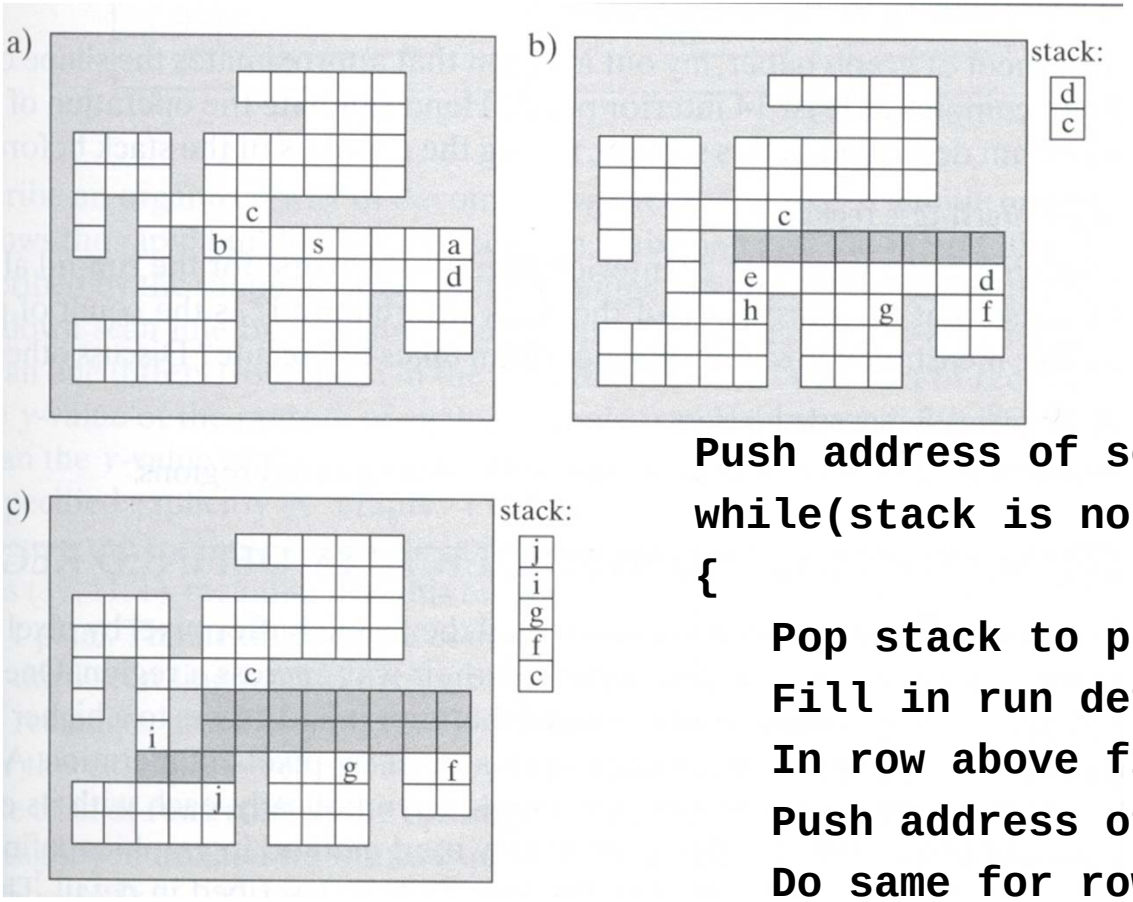


- This version defines region using intColor
- Can also have version defining region by boundary
- Recursive flood-fill is somewhat blind and some pixels may be retested several times before algorithm terminates
- Region coherence is likelihood that an interior pixel mostly likely adjacent to another interior pixel
- Coherence can be used to improve algorithm performance
- A run: group of adjacent pixels lying on same scanline
- Fill runs(adjacent, on same scan line) of pixels

Note: algorithm most efficient if there is **span coherence** (pixels on scanline have same value) and **scan-line coherence** (consecutive scanlines are similar)

Region Filling Using Coherence

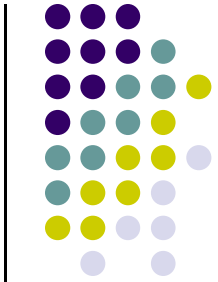
- Example: start at s, initial seed



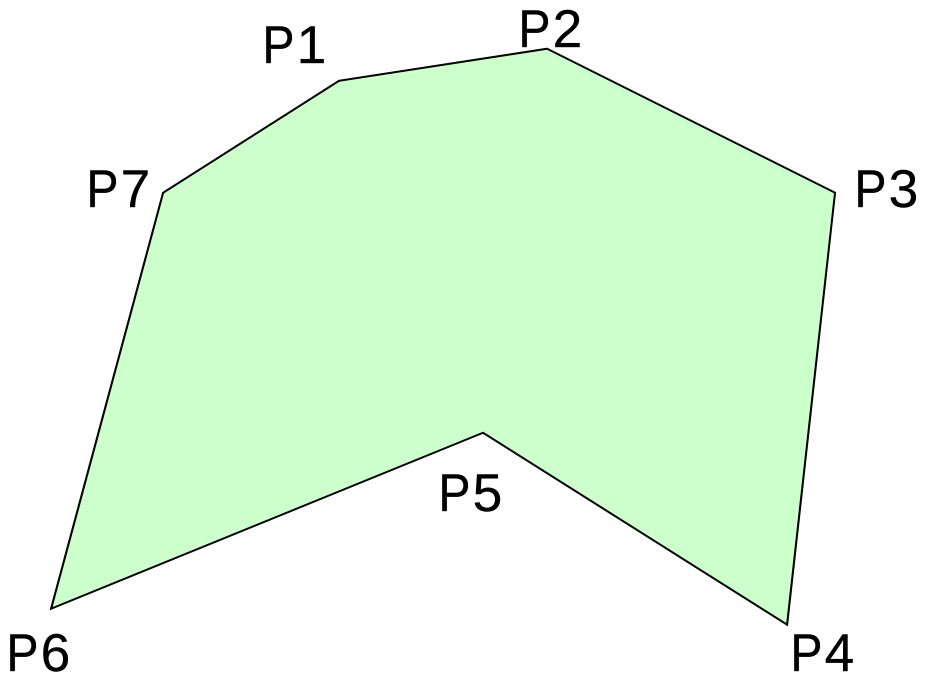
Pseudocode:

```
Push address of seed pixel onto stack
while(stack is not empty)
{
    Pop stack to provide next seed
    Fill in run defined by seed
    In row above find reachable interior runs
    Push address of their rightmost pixels
    Do same for row below current run
}
```

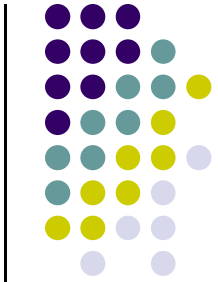
Filling Polygon-Defined Regions



- **Problem:** Region defined by Polygon P with vertices Pi = (Xi, Yi), for i – 1…N, specifying sequence of P's vertices

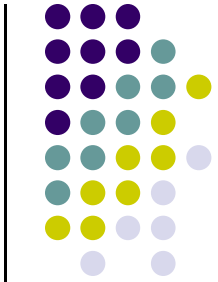


Filling Polygon-Defined Regions



- **Solution:** Progress through frame buffer scan line by scan line, filling in appropriate portions of each line
- Filled portions defined by intersection of scan line and polygon edges
- Runs lying between edges inside P are filled

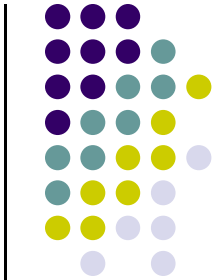
Filling Polygon-Defined Regions



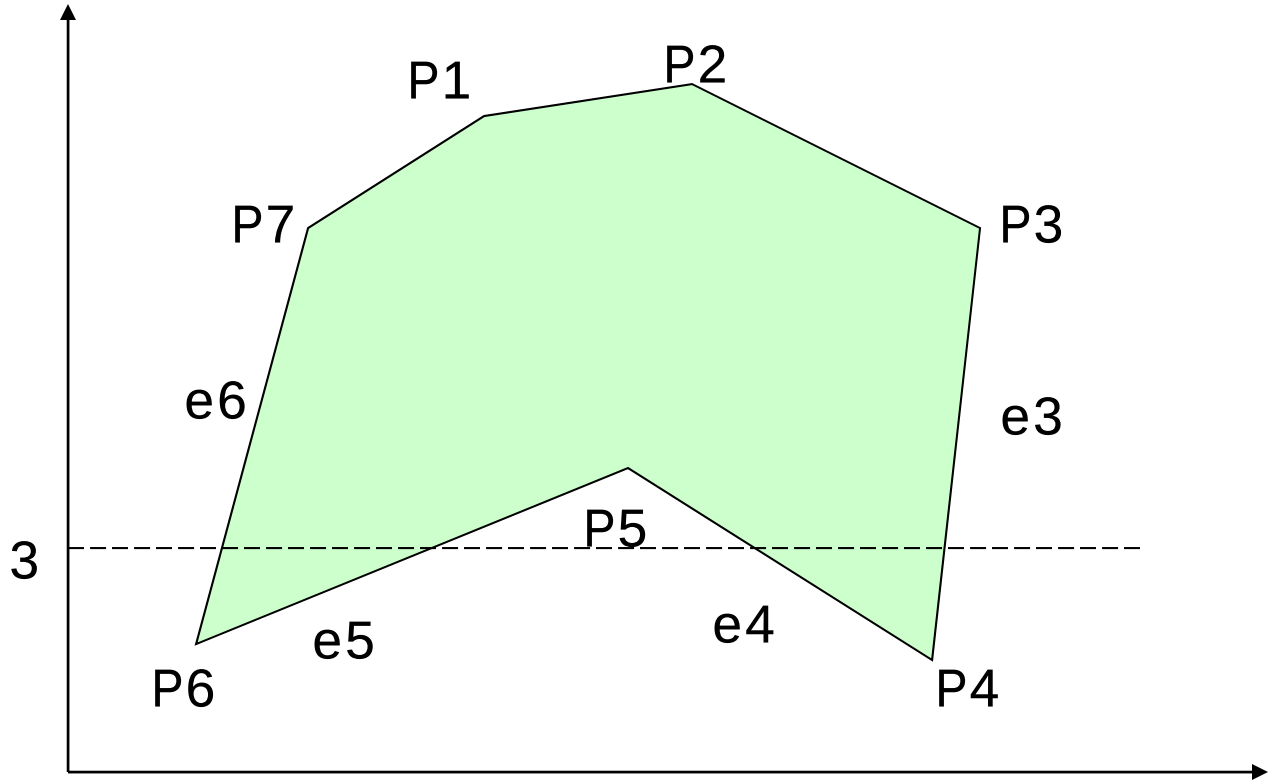
- **Pseudocode:**

```
for(each scan Line L)
{
   Find intersections of L with all edges of P
   Sort the intersections by increasing x-value
   Fill pixel runs between all pairs of intersections
}
```

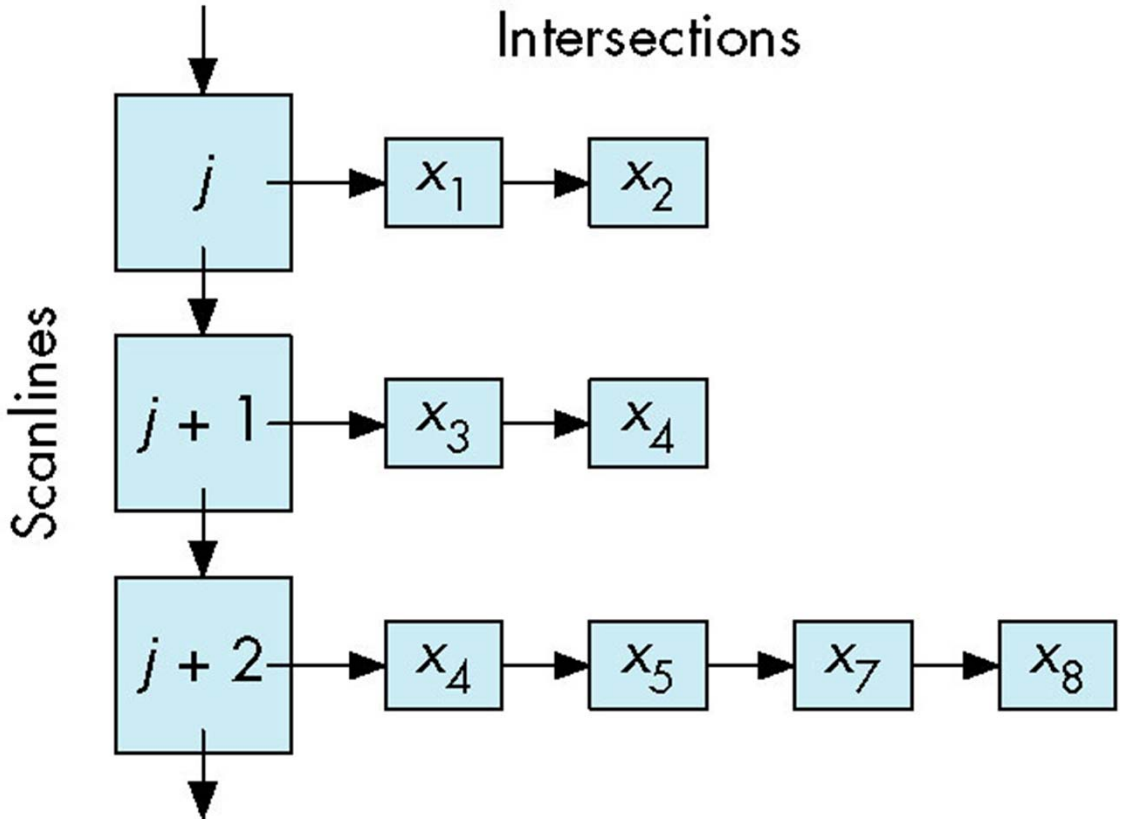
Filling Polygon-Defined Regions



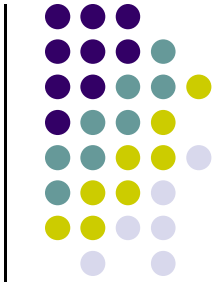
- **Example:** scan line y = 3 intersects 4 edges e3, e4, e5, e6
- Sort x values of intersections and fill runs in pairs
- **Note:** at each intersection, inside-outside (parity), or vice versa



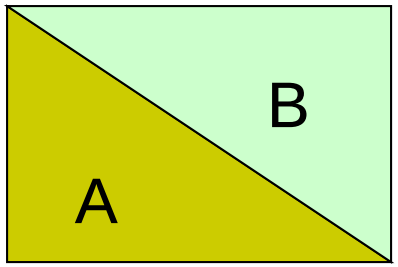
Data Structure



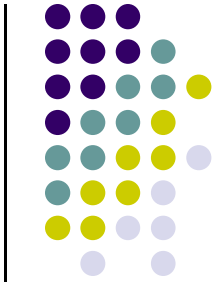
Filling Polygon-Defined Regions



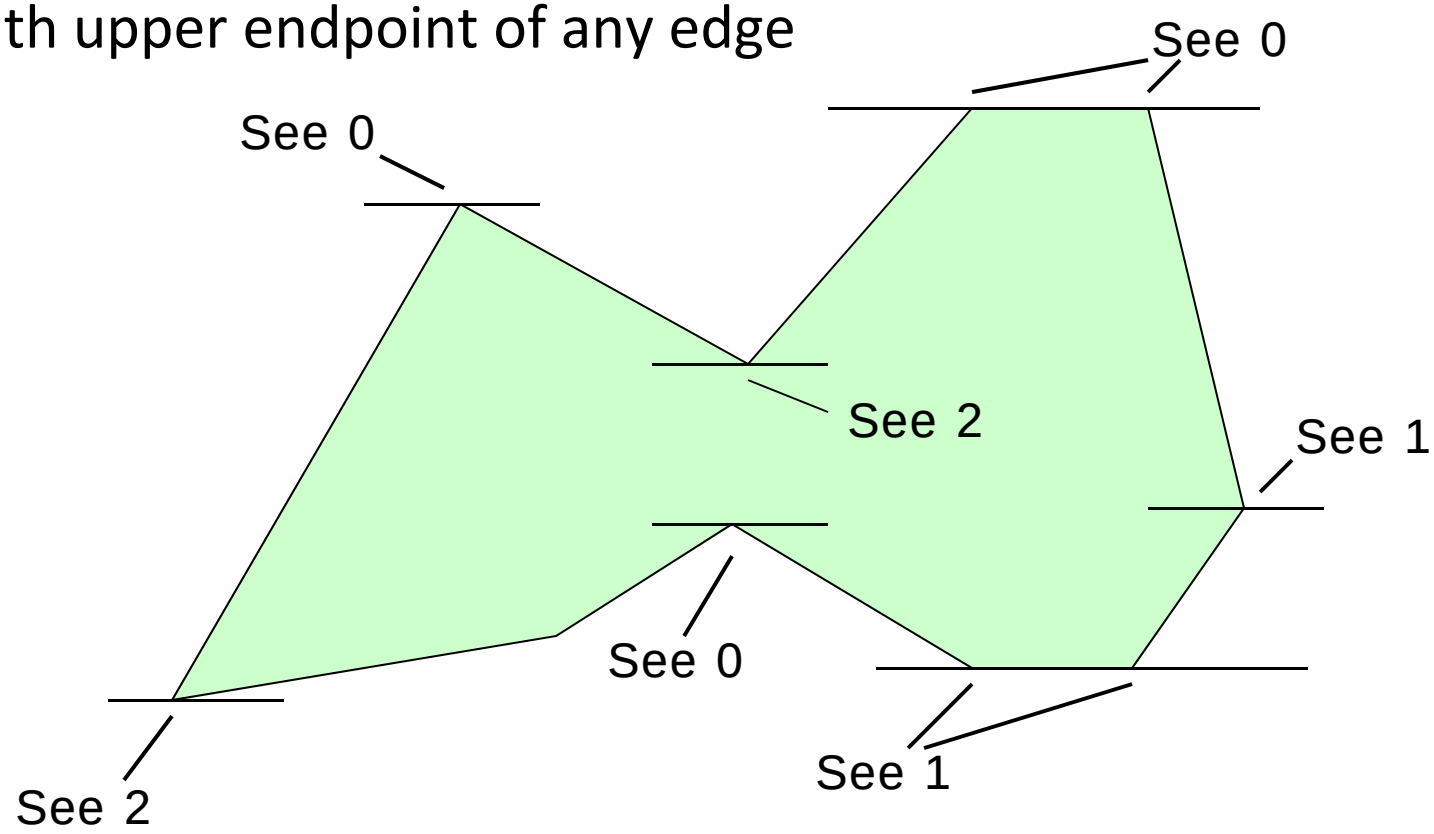
- What if two polygons A, B share an edge?
- Algorithm behavior could result in:
  - setting edge first in one color and the another
  - Drawing edge twice too bright
- **Make Rule:** when two polygons share edge, each polygon owns its left and bottom edges
- E.g. below draw shared edge with color of polygon **B**



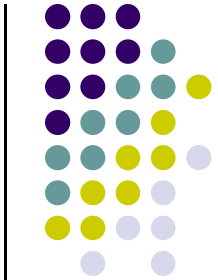
Filling Polygon-Defined Regions



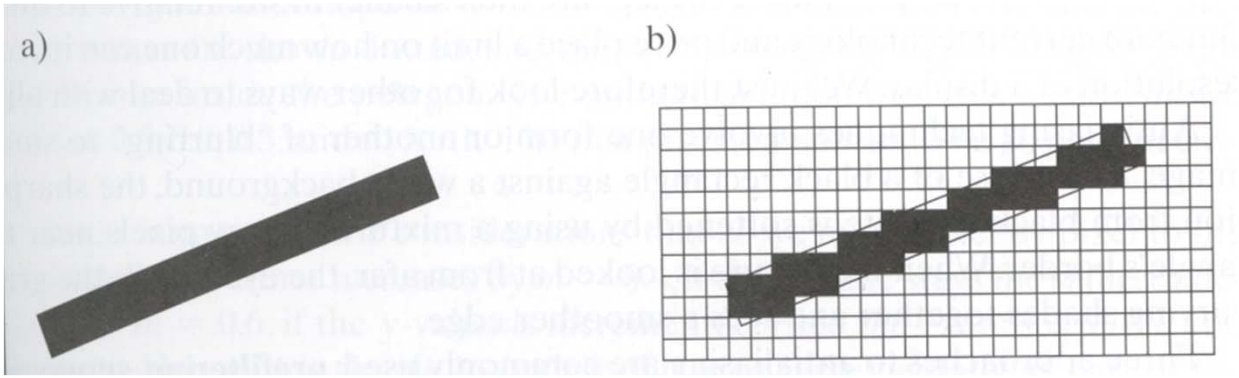
- How to handle cases where scan line intersects with polygon endpoints to avoid wrong parity?
- **Solution:** Discard intersections with horizontal edges and with upper endpoint of any edge



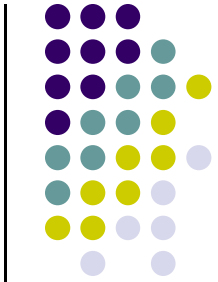
Antialiasing



- Raster displays have pixels as rectangles
- Aliasing: Discrete nature of pixels introduces "jaggies"

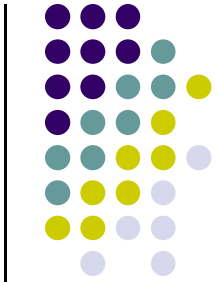


Antialiasing



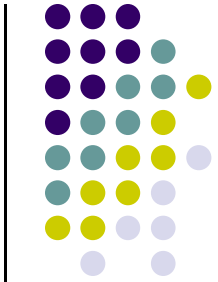
- Aliasing effects:
  - Distant objects may disappear entirely
  - Objects can blink on and off in animations
- Antialiasing techniques involve some form of blurring to reduce contrast, smoothen image
- Three antialiasing techniques:
  - Prefiltering
  - Postfiltering
  - Supersampling

Prefiltering

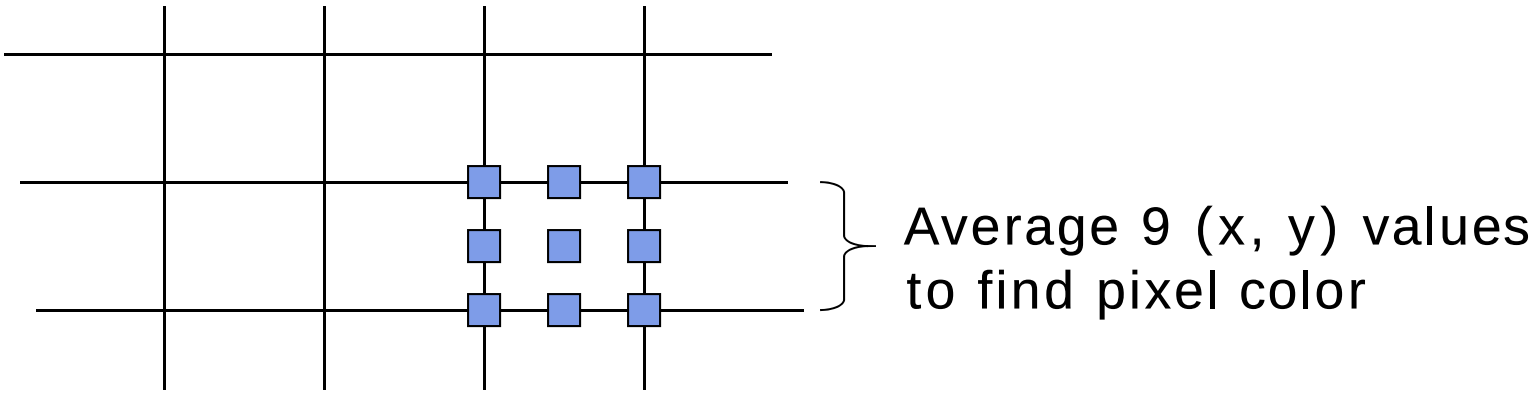


- Basic idea:
  - compute area of polygon coverage
  - use proportional intensity value
- Example: if polygon covers ¼ of the pixel
  - use ¼ polygon color
  - add it to ¾ of adjacent region color
- Cons: computing polygon coverage can be time consuming

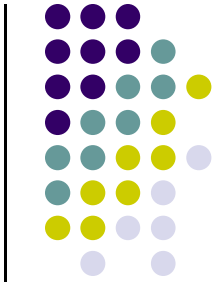
Supersampling



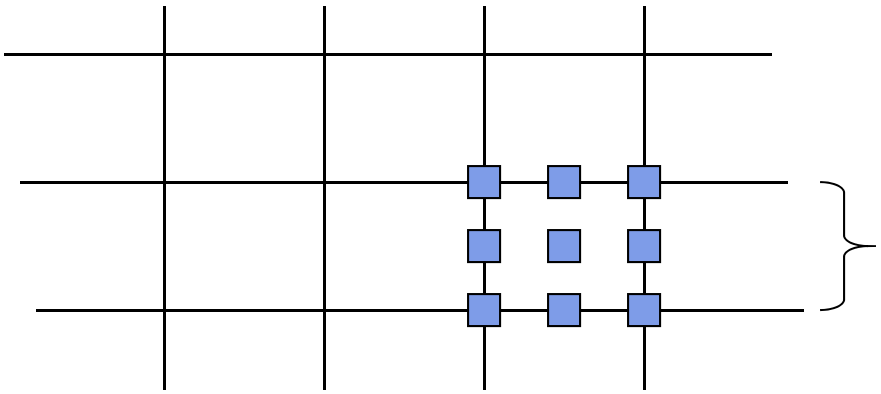
- Useful if we can compute color of any (x,y) value on the screen
- Increase frequency of sampling
- Instead of (x,y) samples in increments of 1
- Sample (x,y) in fractional (e.g. ½) increments
- Find average of samples
- Example: Double sampling = increments of ½ = 9 color values averaged for each pixel



Postfiltering



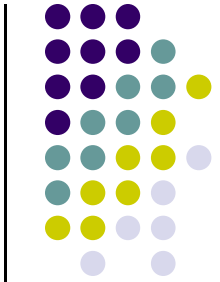
- Supersampling uses average
- Gives all samples equal importance
- Post-filtering: use weighting (different levels of importance)
- Compute pixel value as weighted average
- Samples close to pixel center given more weight



Sample weighting

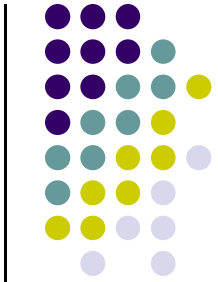
1/16	1/16	1/16
1/16	1/2	1/16
1/16	1/16	1/16

Antialiasing in OpenGL



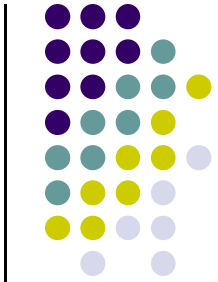
- Many alternatives
- Simplest: accumulation buffer
- Accumulation buffer: extra storage, similar to frame buffer
- Samples are accumulated
- When all slightly perturbed samples are done, copy results to frame buffer and draw

Antialiasing in OpenGL



- First initialize:
  - **`glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_ACCUM | GLUT_DEPTH);`**
- Zero out accumulation buffer
  - **`glClear(GLUT_ACCUM_BUFFER_BIT);`**
- Add samples to accumulation buffer using
  - **`glAccum( )`**

Antialiasing in OpenGL



- Sample code
- jitter[] stores randomized slight displacements of camera,
- factor, f controls amount of overall sliding

```
glClear(GL_ACCUM_BUFFER_BIT);
for(int i=0;i < 8; i++)
{
   cam.slide(f*jitter[i], f*jitter[i].y, 0);
   display( );
   glAccum(GL_ACCUM, 1/8.0);
}
glAccum(GL_RETURN, 1.0);
```

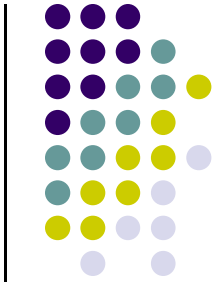
jitter.h

-0.3348, 0.4353

0.2864, -0.3934

.....

References



- Angel and Shreiner, Interactive Computer Graphics, 6th edition
- Hill and Kelley, Computer Graphics using OpenGL, 3rd edition, Chapter 9