CS 543: Computer Graphics

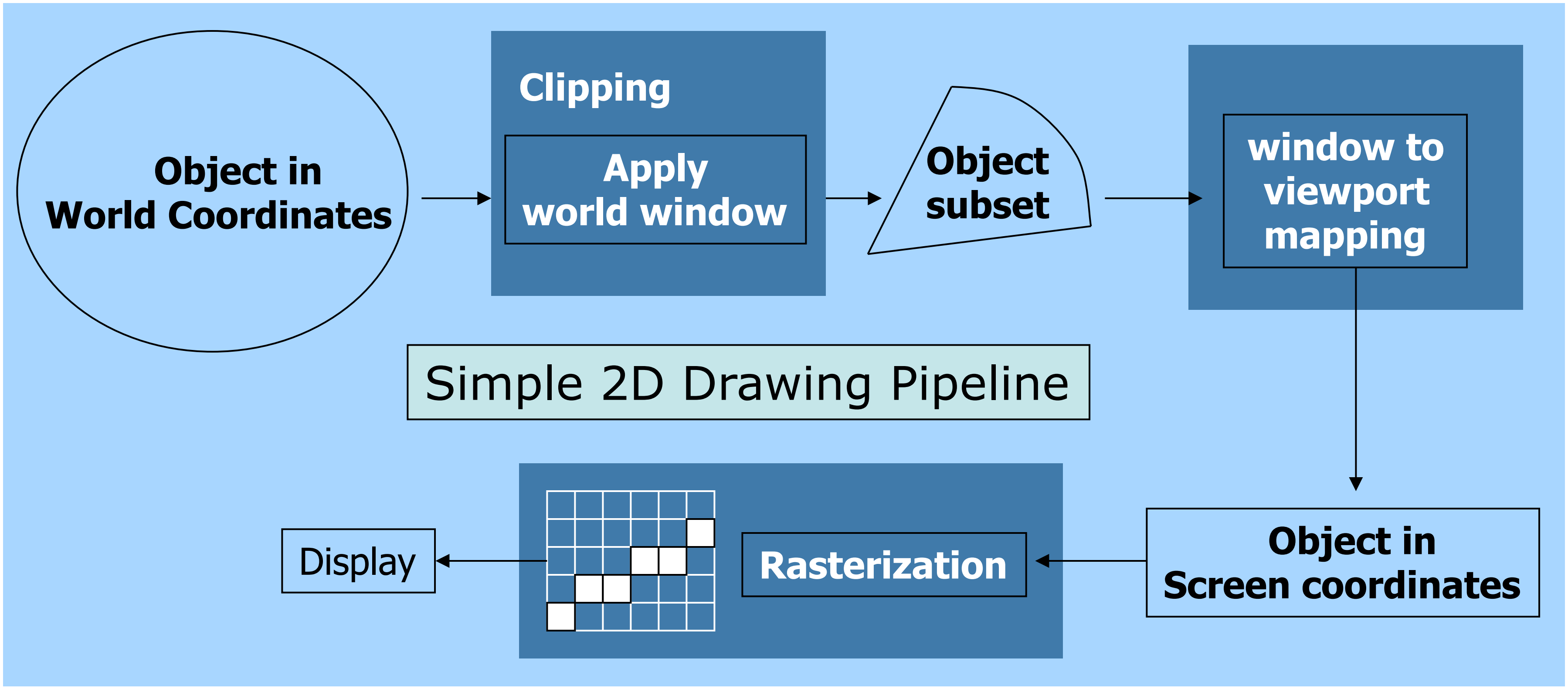
Rasterization

Robert W. Lindeman
Associate Professor
Interactive Media & Game Development
Department of Computer Science
Worcester Polytechnic Institute
gogo@wpi.edu

(with lots of help from Prof. Emmanuel Agu :-)

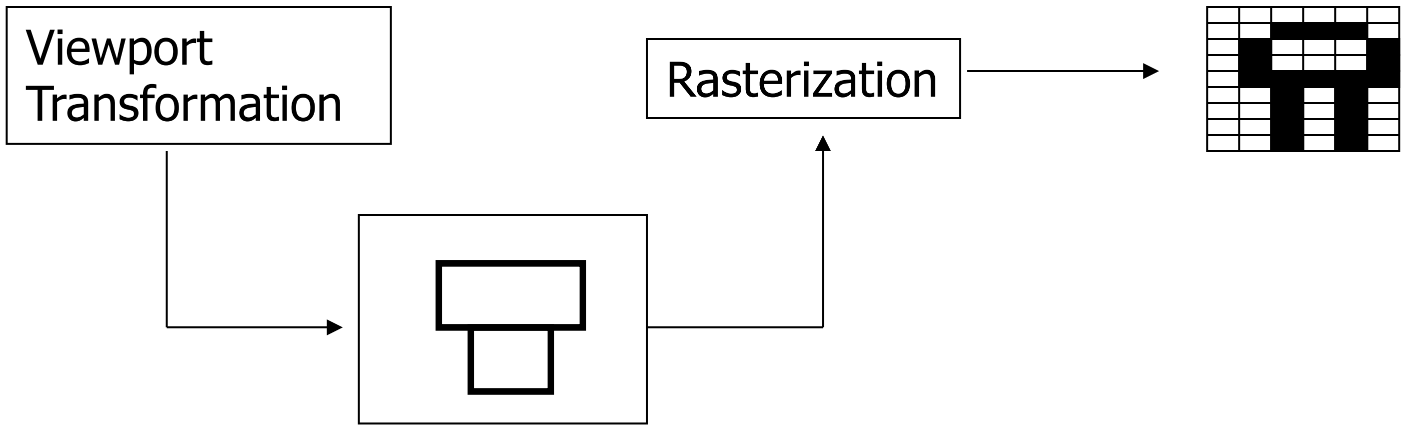
2D Graphics Pipeline

- Simplified


Object in World Coordinates
Clipping
Apply world window
Object subset
window to viewport mapping
Simple 2D Drawing Pipeline
Object in Screen coordinates
Rasterization
Display

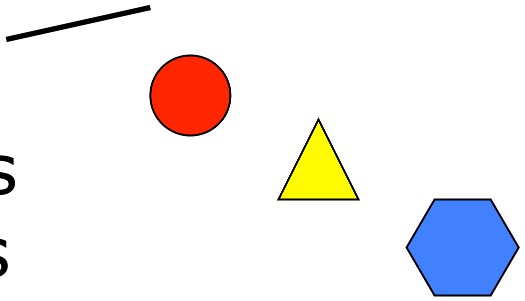
Rasterization (Scan Conversion)

- Convert high-level geometry description to pixel colors in the frame buffer
- Example: given vertex x, y coordinates, determine pixel colors to draw line
- Two ways to create an image
  - Scan existing photograph
  - Procedurally compute values (rendering)



Rasterization

- A fundamental computer graphics function
- Determine the pixels' colors, illuminations, textures, *etc.*
- Implemented by graphics hardware
- Rasterization algorithms
  - Lines
  - Circles
  - Triangles
  - Polygons

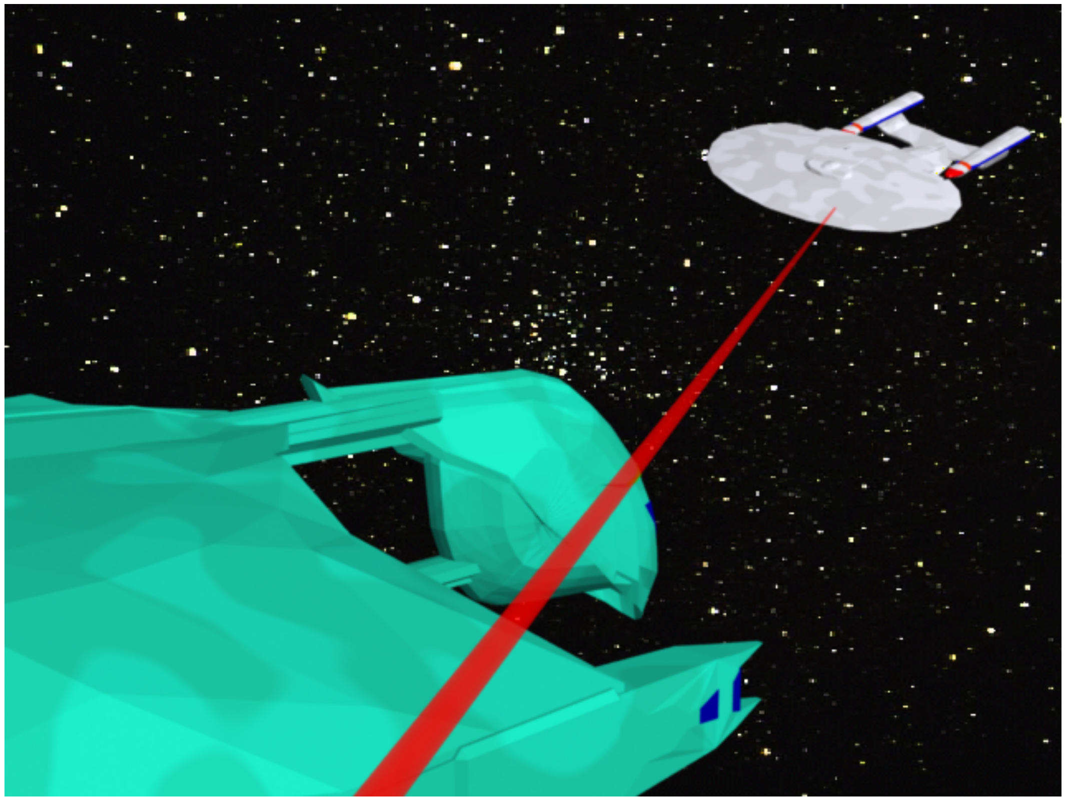


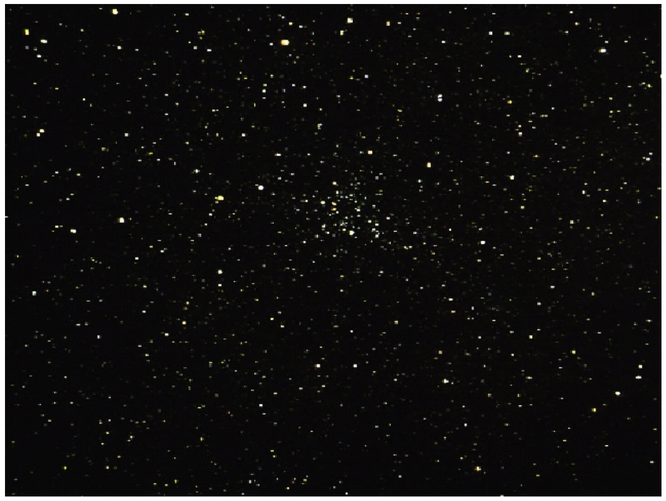
Rasterization Operations

- Drawing lines on the screen
- Manipulating pixel maps (pixmaps): copying, scaling, rotating, *etc.*
- Compositing images, defining and modifying regions
- Drawing and filling polygons
    - Previously **`gl.drawArrays( )`**
- Aliasing and antialiasing methods

WPI

Compositing Example



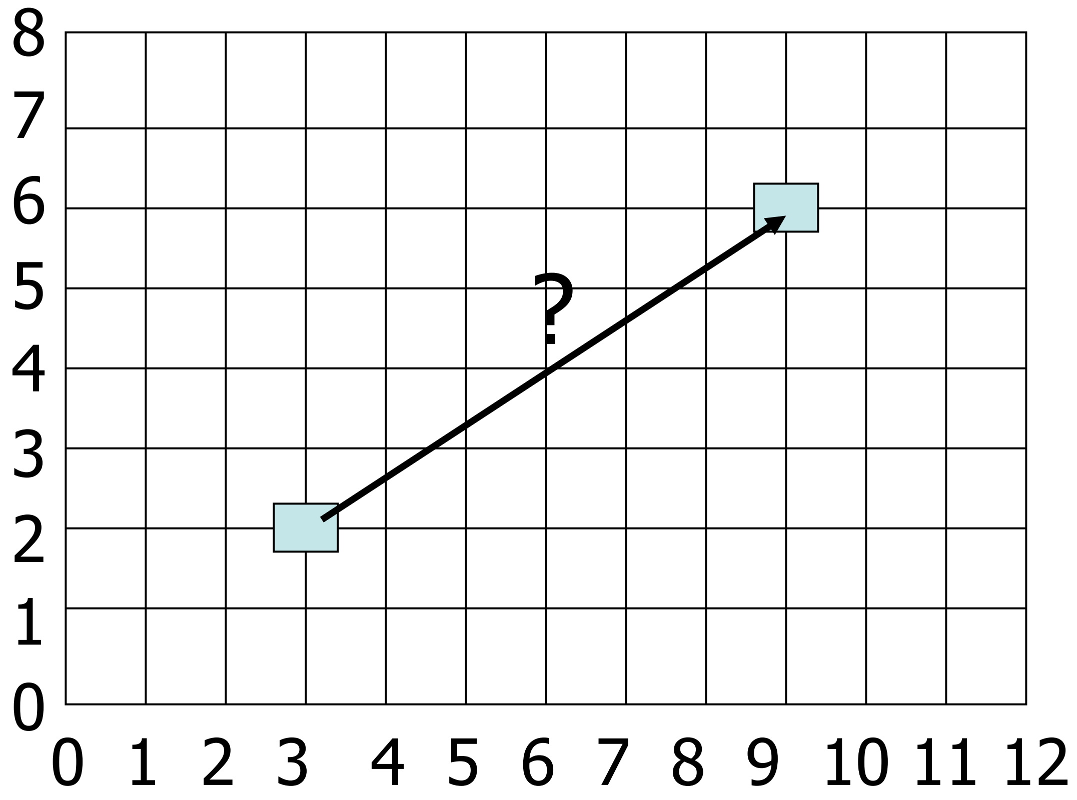


Line Drawing Algorithm

- Programmer specifies (x, y) values of end pixels
- Need algorithm to figure out which intermediate pixels are on line path
- Pixel (x, y) values constrained to integer values
- Actual computed intermediate line values may be floats
- Rounding may be required, *e.g.*, computed point (10.48, 20.51) rounded to (10, 21)
- Rounded pixel value is off actual line path (jaggy!!)
- Sloped lines end up having jaggies, but vertical, horizontal lines don't

WPI

Line Drawing Algorithm (cont.)

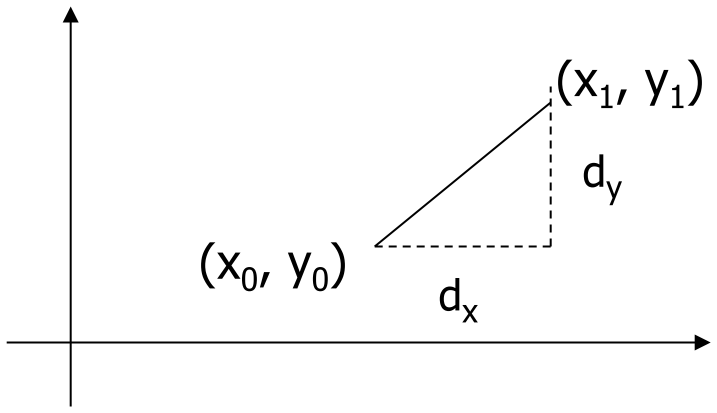


Line: (3,2) -> (9,6)

Which intermediate pixels should we light?

Line Drawing Algorithm (cont.)

- Slope-intercept line equation
  - $y = mx + b$
  - Given two end points (x_0, y_0), (x_1, y_1), how do we compute *m* and *b*?



$$m = \frac{d_y}{d_x} = \frac{y_1 - y_0}{x_1 - x_0}$$

$$b = y_0 - m * x_0$$

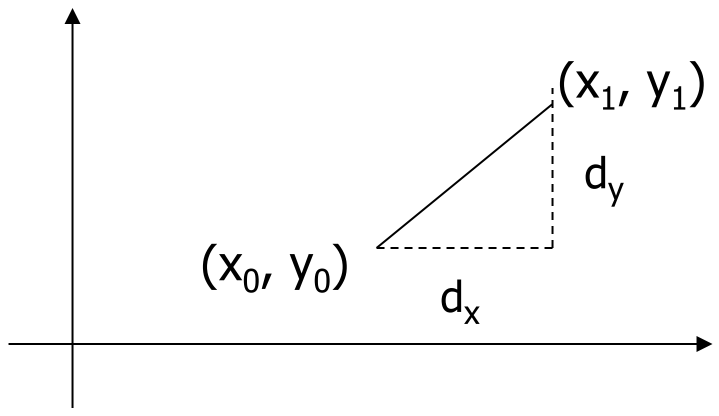
Line Drawing Algorithm (cont.)

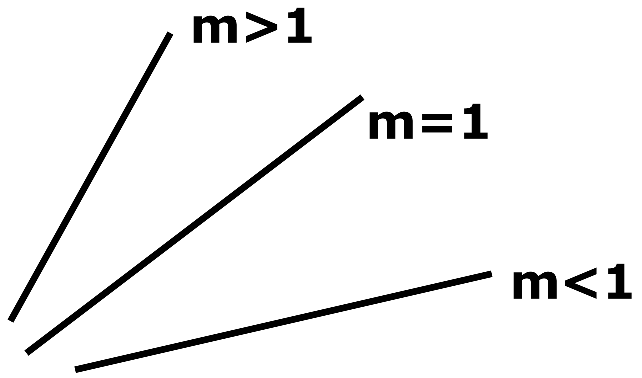
- Numerical example of finding slope m:
- $(A_x, A_y) = (23, 41), (B_x, B_y) = (125, 96)$

$$m = \frac{B_y - A_y}{B_x - A_x} = \frac{96 - 41}{125 - 23} = \frac{55}{102} = 0.5392$$

Line Drawing Algorithm: Digital Differential Analyzer (DDA)

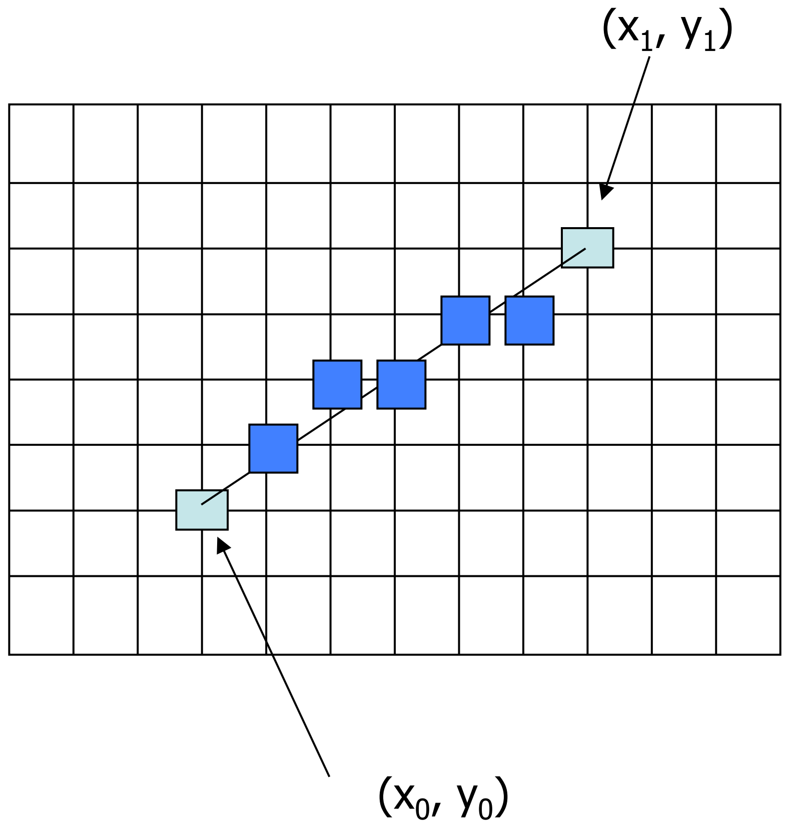
- Walk through the line, starting at (x_0,y_0)
- Constrain x, y increments to values in [0,1] range
- Case a: x is incrementing faster ($m < 1$)
  - Step in $x=1$ increments, compute and round y
- Case b: y is incrementing faster ($m > 1$)
  - Step in $y=1$ increments, compute and round x





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

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



$x = x_0$ $y = y_0$

Illuminate pixel (x, round(y))

$x = x_0 + 1$ $y = y_0 + m$

Illuminate pixel (x, round(y))

$x = x + 1$ $y = y + m$

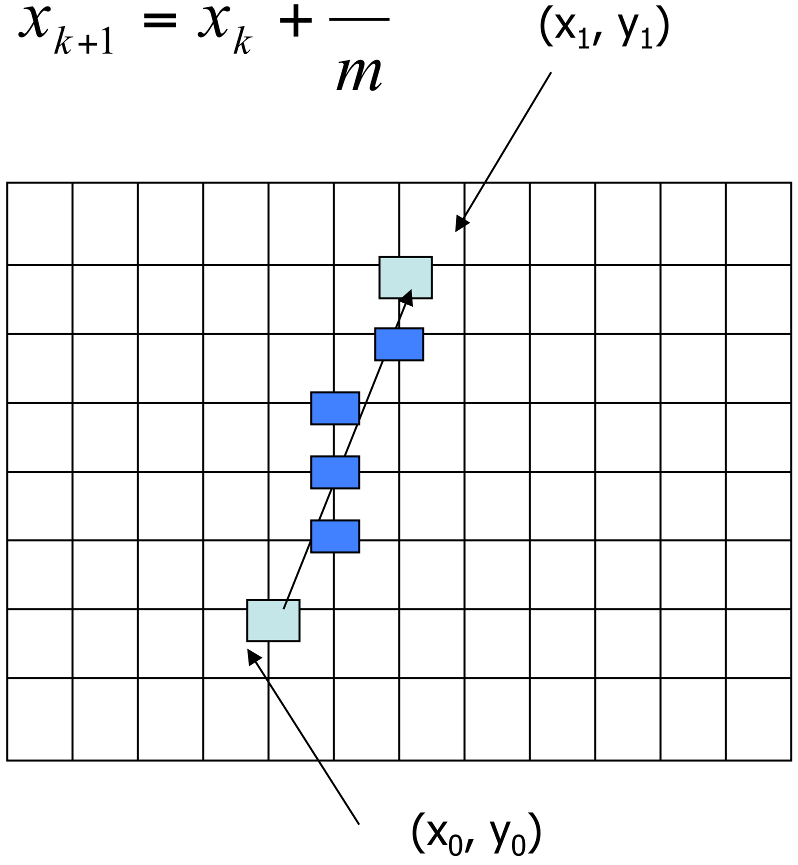
Illuminate pixel (x, round(y))

…

Until $x == x_1$

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

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



$x = x_0$ $y = y_0$

Illuminate pixel (round(x), y)

$y = y_0 + 1$ $x = x_0 + 1/m$

Illuminate pixel (round(x), y)

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

Illuminate pixel (round(x), y)

…

Until $y == y_1$

DDA Line Drawing Algorithm Pseudocode

```
compute m;
if m < 1
  float y = $y_0$;         // initial value
  for( int x = $x_0$; x <= $x_1$; x++, y += m )
    setPixel( x, round(y) );
else // m > 1
  float x = $x_0$;         // initial value
  for( int y = $y_0$; y <= $y_1$; y++, x += 1/m )
    setPixel( round(x), y );
```

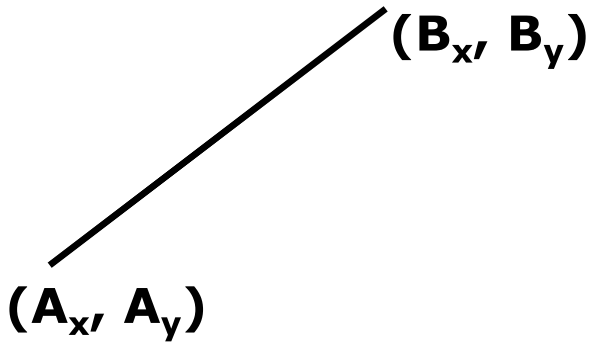
- Note: `setPixel(x, y)` writes current color into pixel in column x and row y in frame buffer

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's algorithm (Hill, 9.4.1)
- Bresenham's 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

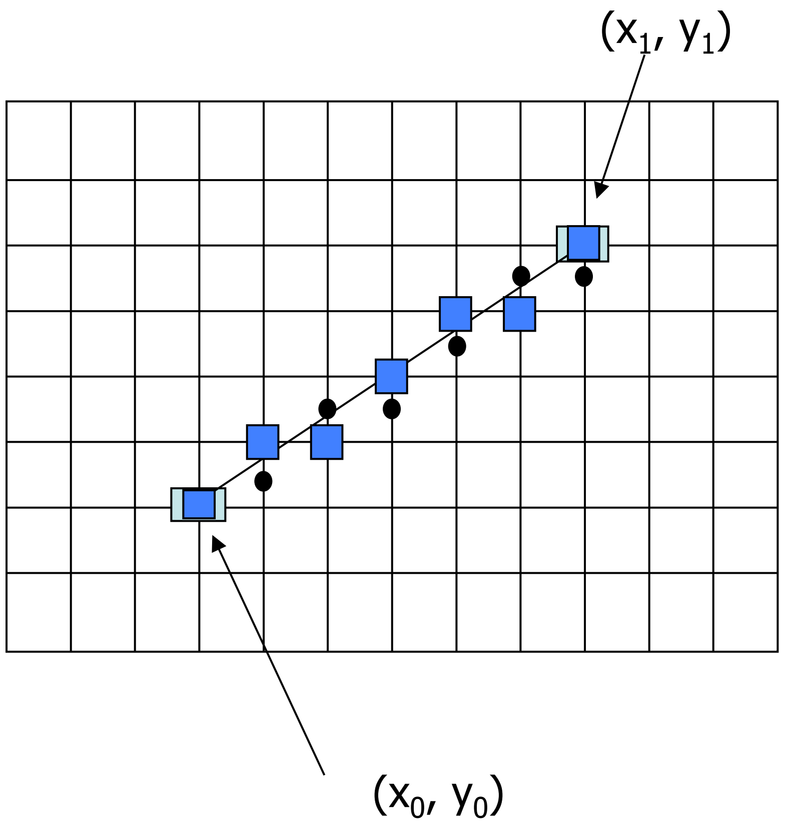
- Problem
  - Given endpoints (A_x, A_y) and (B_x, B_y) of a line, want to determine best sequence of intervening pixels
- First make two simplifying assumptions (remove later):
  - $(A_x < B_x)$ and
  - $(0 < m < 1)$
- Define
  - Width $W = B_x - A_x$
  - Height $H = B_y - A_y$



Bresenham's Line-Drawing Algorithm (cont.)

- Based on assumptions
  - W, H are positive
  - 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 (cont.)



What Pixels do we need to turn on?

Consider pixel midpoint $M(M_x, M_y)$

$M = (x_0 + 1, Y_0 + ½)$

Build equation of line through and compare to midpoint

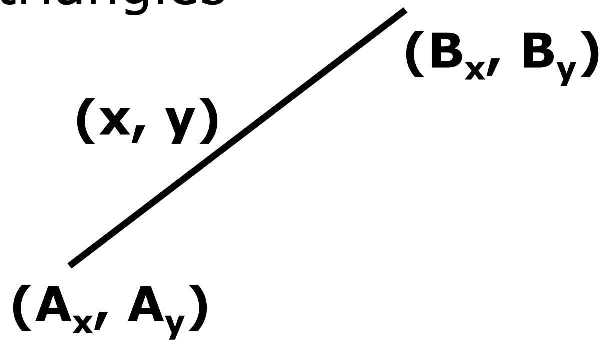
If midpoint is above line, y stays same
If midpoint is below line, y increases + 1

…

Bresenham's Line-Drawing Algorithm (cont.)

- To get a good line equation, use similar triangles

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



$$H(x - A_x) = W(y - A_y)$$
$$-W(y - A_y) + H(x - A_x) = 0$$

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

$$F(x, y) = -2W(y - A_y) + 2H(x - A_x)$$

Bresenham's Line-Drawing Algorithm (cont.)

- So, $F(x, y) = -2W(y - A_y) + 2H(x - A_x)$
- 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

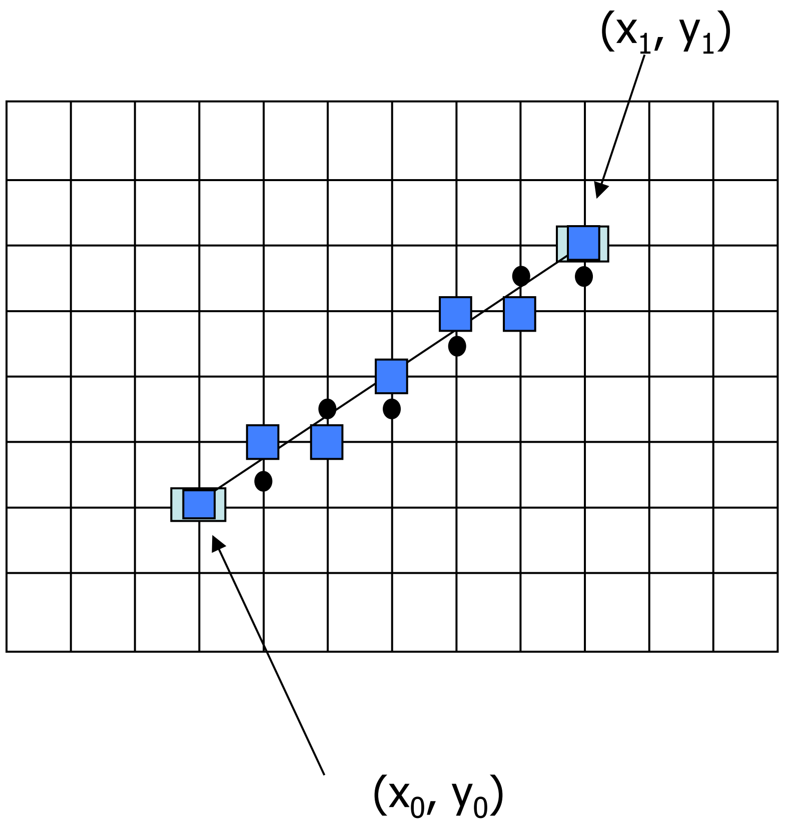
Bresenham's Line-Drawing Algorithm (cont.)

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

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

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

Bresenham's Line-Drawing Algorithm (cont.)



What Pixels do we need to turn on?

Consider pixel midpoint $M(M_x, M_y)$

$M = (x_0 + 1, Y_0 + ½)$

If $F(M_x, M_y) < 0$, M lies above line, shade lower pixel (same y as before)

If $F(M_x, M_y) > 0$, M lies below line, shade upper pixel ($y = y + 1$)

…

Bresenham's Line-Drawing Algorithm (cont.)

- We can compute $F(x, y)$ incrementally
  - Initially, midpoint $M = (A_x + 1, A_y + ½)$
    - $F(M_x, M_y) = -2W(y - A_y) + 2H(x - A_x)$
      $= 2H - W$
  - Can compute $F(x, y)$ for next midpoint incrementally
  - If we increment $x + 1$, y stays same
    $F(M_x, M_y) \mathrel{+}= 2H$
  - If we increment $x +1$, $y + 1$
    $F(M_x, M_y) \mathrel{+}= 2(W - H)$

Bresenham's Line-Drawing Algorithm (cont.)

```
Bresenham( IntPoint a, IntPoint 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 += 2H;
    else  {
      y++;
      F += 2(H - W)
    }
  }
}
```

- Recall: F is the equation of a line

Bresenham's Line-Drawing Algorithm (cont.)

- Final words: we developed algorithm with restrictions

  $0 < m < 1$ and $Ax < Bx$

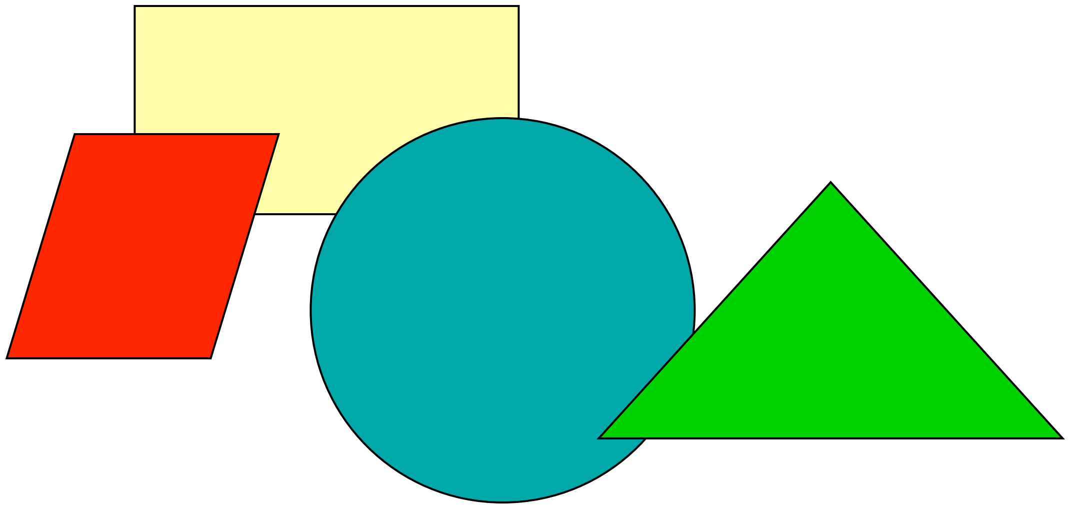
- Can add code to remove restrictions
  - To get the same line when $A_x > B_x$ (swap and draw)
  - Lines having $m > 1$ (interchange x with y)
  - Lines with $m < 0$ (step x++, decrement y not incr)
  - Horizontal and vertical lines (pretest a.x = b.x and skip other tests)

Rasterization So Far...

- Raster graphics
  - Line-drawing algorithms (DDA, Bresenham's)
- Now
  - Defining and filling Regions
  - Polygon drawing and filling
  - Antialiasing

Defining and Filling Regions of Pixels

- First, understand how to define and fill any defined regions
- Next, how to fill regions bounded by a polygon



Methods of Defining Regions

- Pixel-defined
  - Specifies pixels in color or geometric range
- Symbolic
  - Provides property that pixels in region must have
- Examples of symbolic regions
    - 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:** 4-adjacent, plus diagonals
- **4-connected:** If there is unbroken path of 4-adjacent pixels
- **8-connected:** Unbroken path of 8-adjacent pixels

Recursive Flood-Fill Algorithm

- Recursive algorithm
- Starts from initial pixel of color **`initColor`**
- 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 initColor, change it to newColor
  - Do same recursively for all 4 neighbors

Recursive Flood-Fill Algorithm (cont.)

```
void floodFill( short x, short y,
                             short initColor )  {
  if( getPixel( x, y ) == initColor )  {
    setPixel( x, y );
    floodFill( x - 1, y, initColor ); // left
    floodFill( x + 1, y, initColor ); // right
    floodFill( x, y + 1, initColor ); // up
    floodFill( x, y - 1, initColor ); // down
  }
}
```

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

Recursive Flood-Fill Algorithm (cont.)

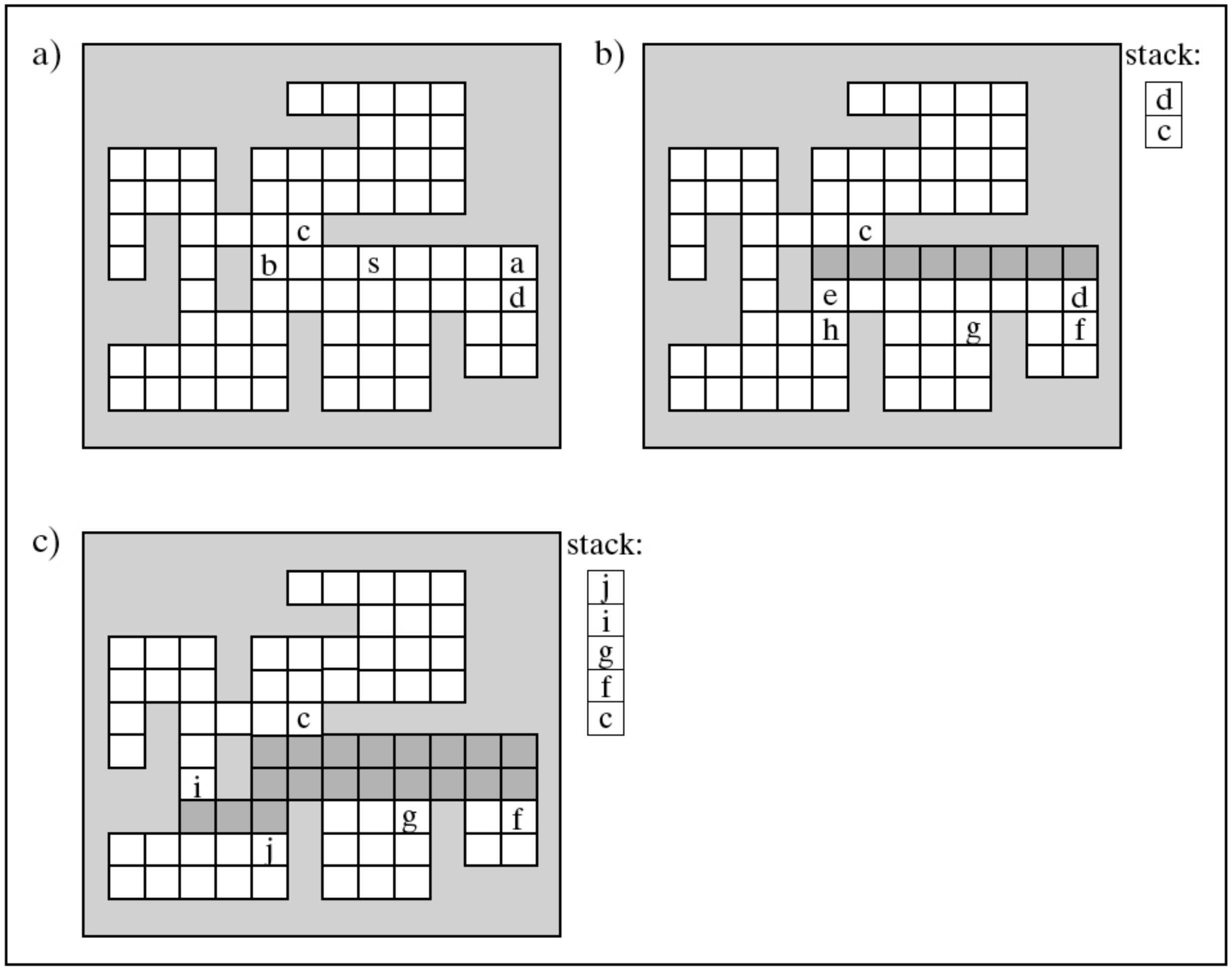
- Okay, now you try it.

Recursive Flood-Fill Algorithm (cont.)

- This version defines region using initColor
- Can also have version defining region by boundary
- Recursive flood-fill is somewhat blind
  - Some pixels may be retested several times before algorithm terminates
- *Region coherence* is likelihood that an interior pixel will be adjacent to another interior pixel
- Coherence can be used to improve algorithm performance
- A *run* is a group of adjacent pixels lying on same scan line
- Exploit runs of pixels

Region Filling Using Coherence

- Start at seed s



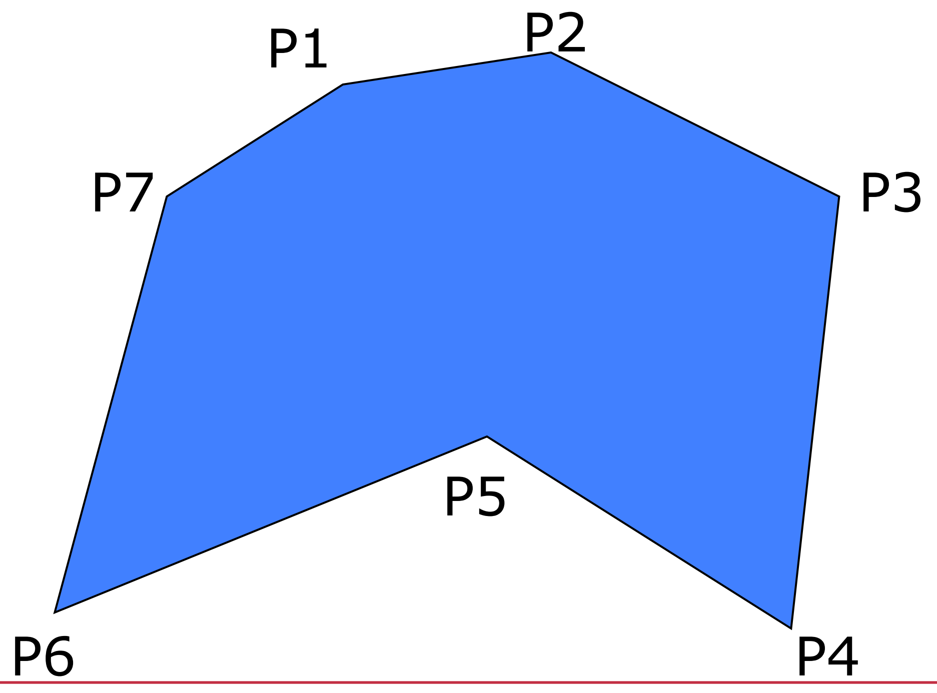
Region Filling Using Coherence Pseudocode

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

Note: Most efficient if there is **span coherence** (pixels on scan line have same value) and **scan-line coherence** (consecutive scan lines are similar)

Filling Polygon-Defined Regions

- **Problem:** Region defined by Polygon P with vertices $P_i = (X_i, Y_i)$, for i = 1…N, specifying sequence of P's vertices



Filling Polygon-Defined Regions (cont.)

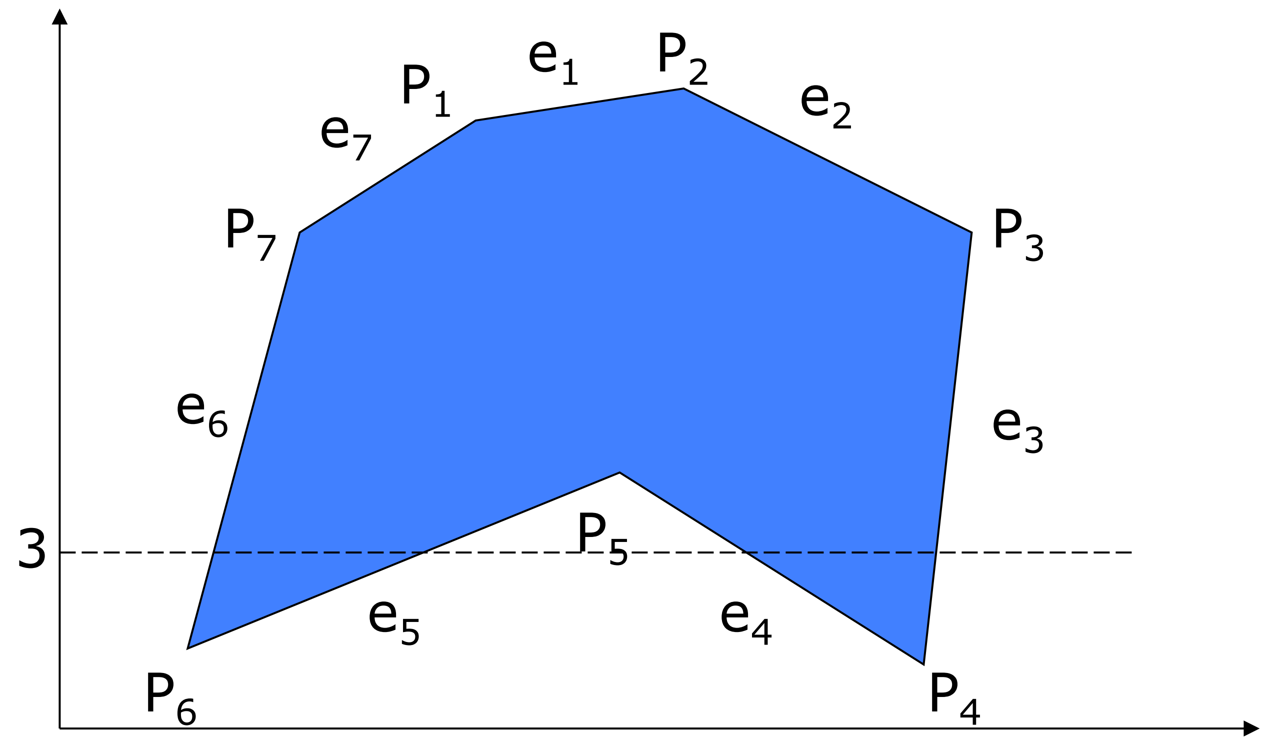
- **Solution:** Progress through frame buffer, scan line by scan line, filling in appropriate portions of each line
- Filled portions defined by intersection of scan line and polygon edges
- Runs lying between edges inside P are filled

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
}
```

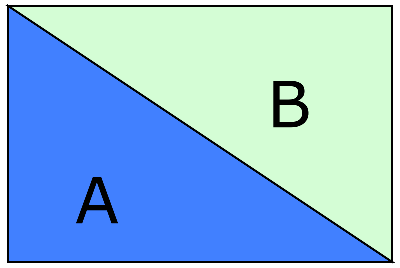
Filling Polygon-Defined Regions (cont.)

- **Example:** scan line y = 3 intersects 4 edges e_3, e_4, e_5, e_6
- Sort x values of intersections and fill runs in pairs
- **Note:** At each intersection, use inside-outside (parity), or vice versa



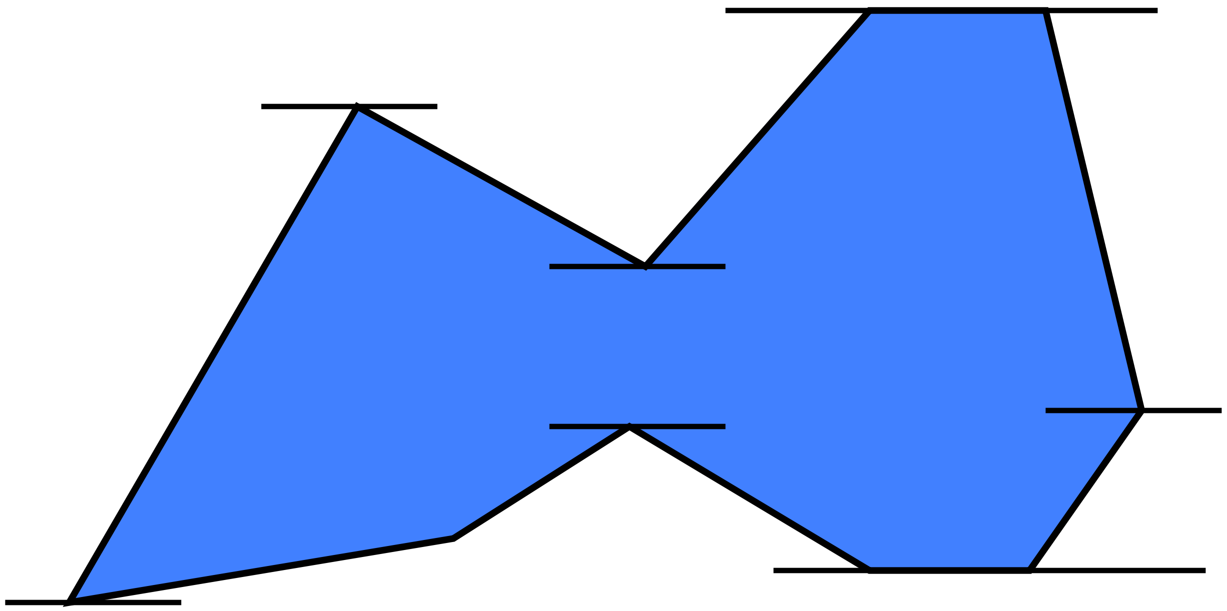
Filling Polygon-Defined Regions (cont.)

- What if two polygons A, B share an edge?
- Algorithm behavior could result in
  - Setting edge first in one color and then 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**



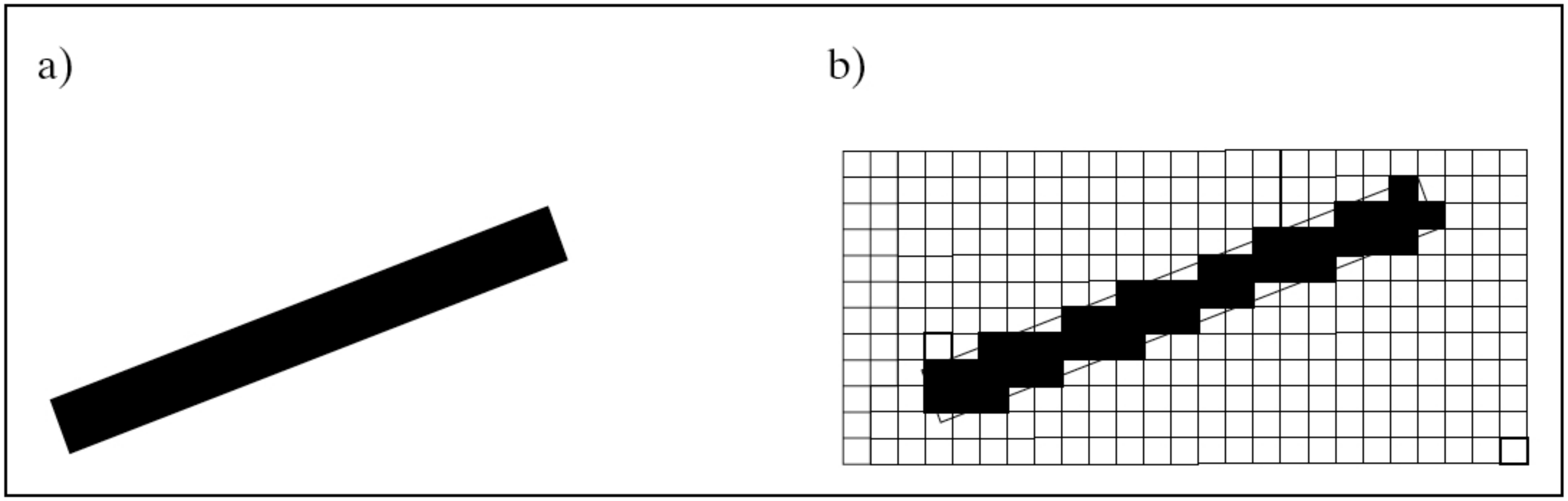
Filling Polygon-Defined Regions (cont.)

- How do we handle cases where scan line intersects with polygon endpoints?
- 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 (cont.)

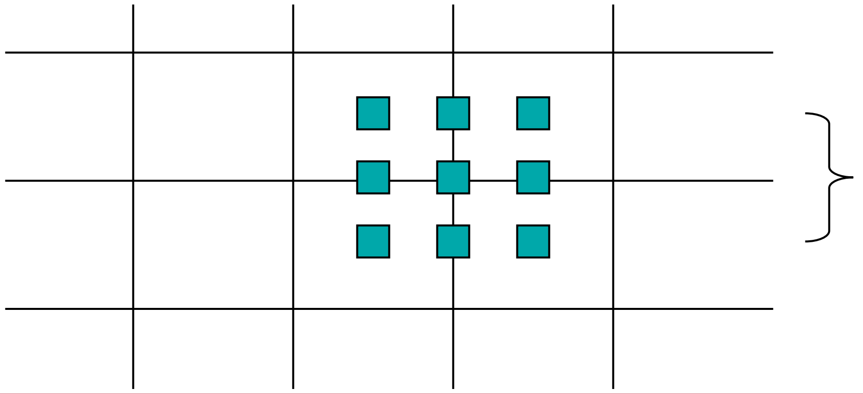
- Aliasing effects
  - Distant objects may disappear entirely
  - Objects can blink on and off in animations
- Antialiasing techniques involve some form of *blurring* to reduce contrast, smooth image
- Three main antialiasing techniques
  - Prefiltering
  - Supersampling
  - Postfiltering

Prefiltering

- Basic idea
  - Compute area of polygon coverage
  - Use proportional intensity value
- Example: if polygon covers 1/2 of the pixel
  - use 1/2 polygon color
  - add it to 1/2 of adjacent region color
- Cons: computing pixel coverage can be time consuming

Supersampling

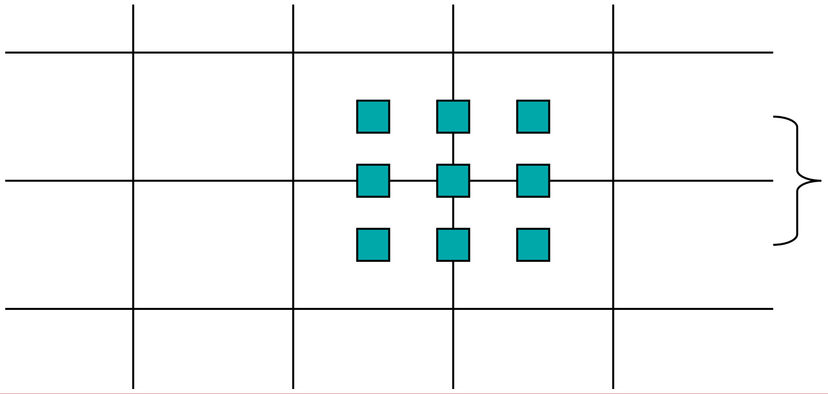
- Useful if we can compute color of any (x,y) value on the screen
- Increase frequency of sampling
- Instead of (x,y) samples in increments of 1, sample (x,y) in fractional (*e.g.*, 1/2) increments
- Find average of samples
- Example: Triple sampling = increments of 1/2 = 9 color values averaged for each pixel



Average 9 (x, y) values to find pixel color

Postfiltering

- Supersampling uses average
- Gives all samples equal importance
- Postfiltering: 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
  - 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 (cont.)

- 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( )`**

WPI

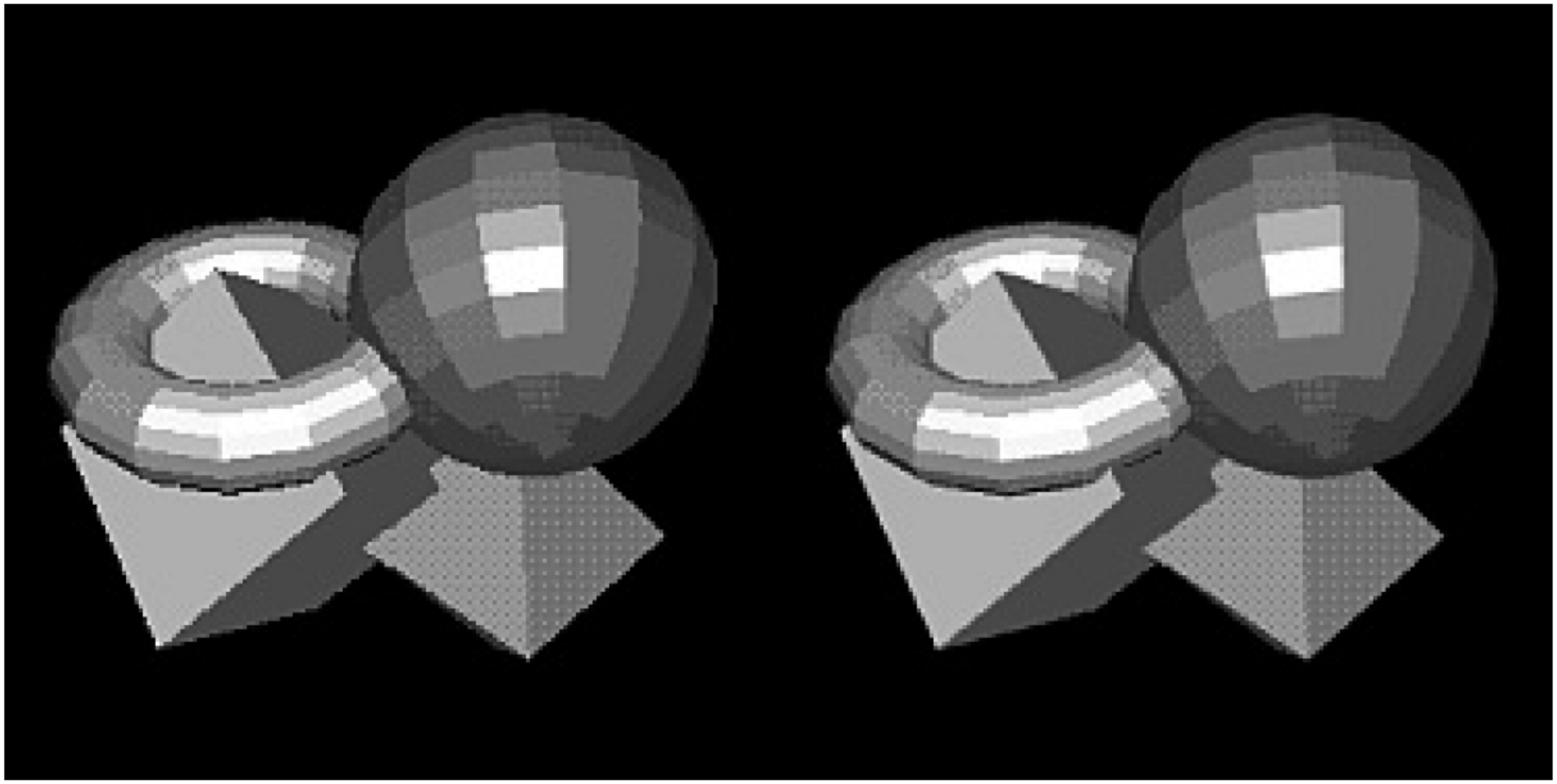
OpenGL Antialiasing Sample Code

```
glClear( GL_ACCUM_BUFFER_BIT );
for( int i = 0; i < 8; i++ )  {
  cam.slide(f*jitter[i].x, 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
…
```

- **`jitter[]`** stores randomized slight displacements of camera,
- Factor, **f** controls amount of overall sliding

Antialiasing Example



Antialiasing Example