CS 430 – Computer Graphics

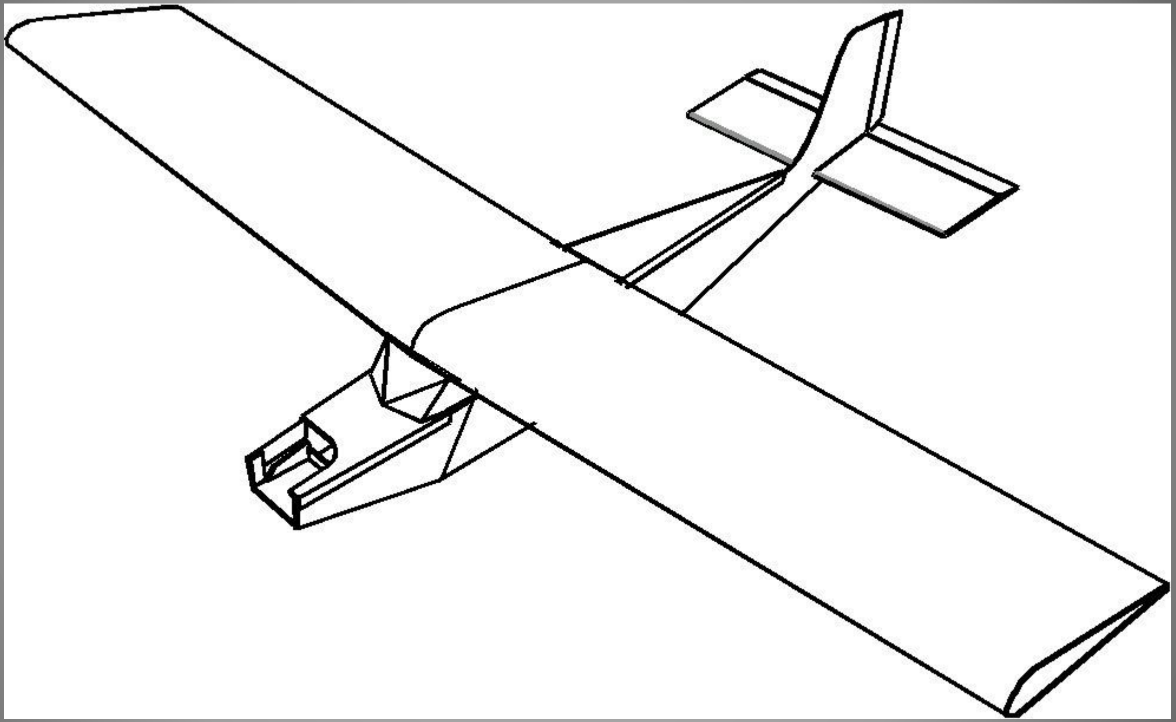
Lecture 1 – Part 3

Line Drawing

Motivation

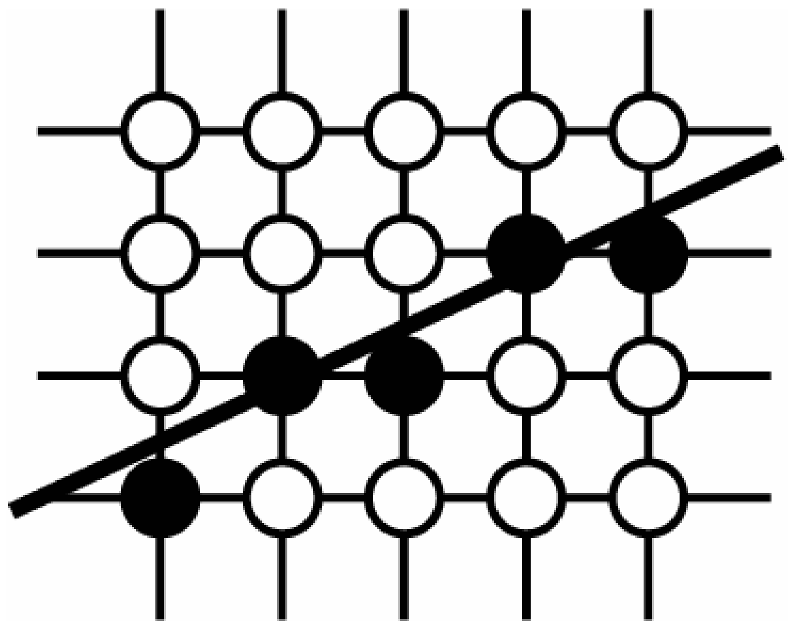
- Most graphics applications allow us to specify vertex attributes (locations, colors, etc..) and then it is the job of the graphics pipeline to draw the desired shape
  - The user specifies the shape type (i.e. line, point, triangle, etc..)
- In this part of the lecture we will start with how to render the pixels of a line given the endpoints

Line Drawing



Scan-Conversion Algorithms

- Scan-Conversion
  - Compute pixel coordinates for *ideal* line on 2D raster grid



Drawing a Line

- We are typically given two endpoints: (x_s, y_s) and (x_e, y_e)
  - From which we can compute the slope

$$m = \frac{\Delta y}{\Delta x} = \frac{(y_e - y_s)}{x_e - x_s}$$

- Then we start at point (x_s, y_s) and compute the points along the way until we reach (x_e, y_e)

Drawing a Line

- Let's assume we choose to move along the x-direction by some fixed integer amount n
- Then given the current point (x_i, y_i) we can find the next point (x_{i+1}, y_{i+1}) by:
  - Compute the next x position x_{i+1} as:
    $$x_{i+1} = x_i + n$$
  - Then we could compute the next y position, y_{i+1} as
    $$y_{i+1} = mx_{i+1} + b$$
  - Or since $\Delta y = y_{i+1} - y_i$ we can compute y_{i+1}
    $$y_{i+1} = y_i + \Delta y = y_i + mn$$
  - And the round y_{i+1} to an integer value.
- Problem:
  - This is expensive (inefficient)
  - Involves a lot of multiplications

Drawing a Line

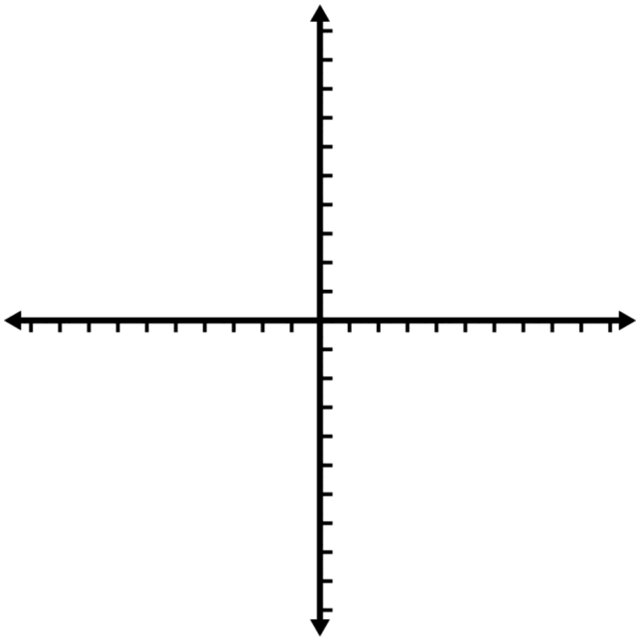
- If we decide to increment x such that $n = 1$, we can make this:

$$x_{i+1} = x_i + 1$$
$$y_{i+1} = y_i + \mathrm{m}$$

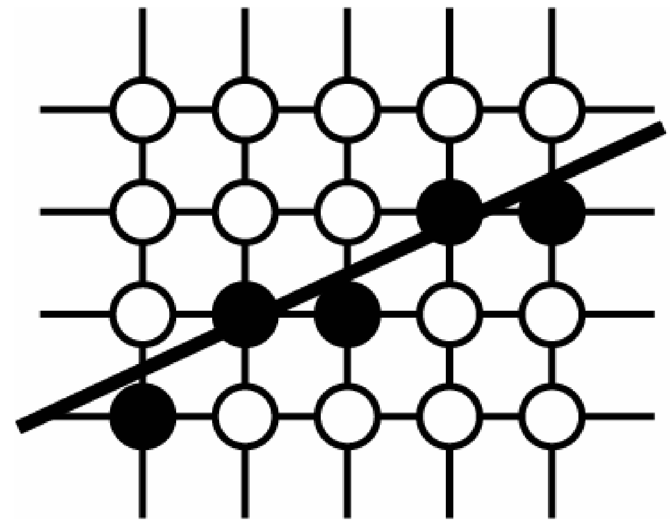
- No more multiplication!

Deciding on Increment Direction

- We don't want our line to have too harsh of jumps
- What would happen if $m = 10$ and we decide to increment in the x-direction by 1?



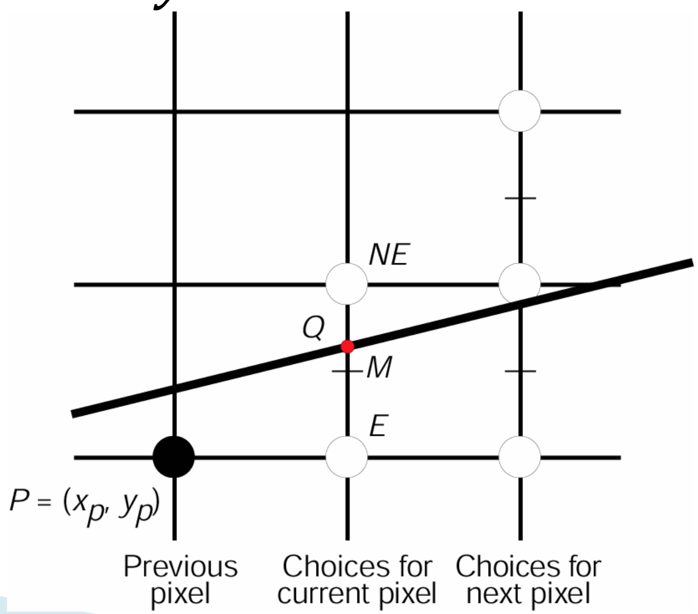
Digital Differential Analyzer (DDA)



- If |m| is less than 1
  - $\Delta x = 1$ and $\Delta y = m$
    - And start at left-most point
    - End at right-most point
- Else
  - $\Delta y = 1$ and $\Delta x = 1/m$
    - And start at the bottom-most point
    - End at top-most point
- Check for vertical line case
  - $m = \infty$
- Compute
  - $x_{k+1} = \mathrm{x_k} + \Delta x$
  - $y_{k+1} = \mathrm{y_k} + \Delta y$
  - Round (x,y) for pixel location (but must keep track of float values too)
- Issue
  - We'd like to avoid floating point operations!

Bresenham's Algorithm

- Another incremental scan-conversion algorithm
  - But performs integer-only arithmetic!
- Based on the *implicit* equation of a line:
  - $f(x, y) = ax + by + c = 0$



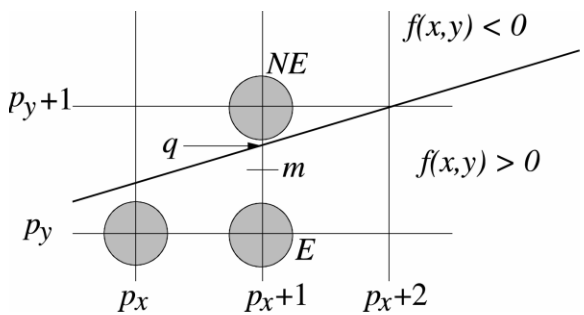
1994 Foley/VanDam/Finer/Huges/Phillips ICG

Bresenham's Algorithm

- Just like with the DDA algorithm we'll have to deal with different slope situations
- For the following example let's assume the slope is $0 \leq m \leq 1$
  - So just like in DDA we want to use $\Delta x = 1$
- What we're about to show is called the *midpoint* approach where we
  - Compute the midpoint and observe the cost
  - Base on the cost, decide where we should go

Bresenham's Algorithm

- Let's assume we are currently at pixel (p_x, p_y)
- Since (for $0 \leq m \leq 1$) we're incrementing in the x direction we have two choices of pixels to go to next:
  - East (E) : $(p_x + 1, p_y)$
  - North East (NE): $(p_x + 1, p_y + 1)$
- Let M be midway between E and NE:
  - $M = \left(p_x + 1, p_y + \frac{1}{2}\right)$

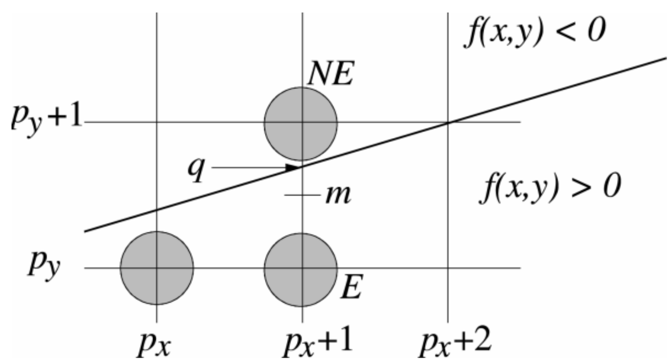


Bresenham's Algorithm

- We can then evaluate the implicit function at M to determine if it's above or below the line
  - If $f(x, y) < 0$ then we're above the line
  - If $f(x, y) > 0$ then we're below the line
  - If $f(x, y) = 0$ then we're one the line
- What does it mean to be "above" the line?
  - If it's a vertical line or slope $m < 0$, then compute a vector V from the endpoint with the smaller y coordinate to the one with the larger y coordinate.
  - Otherwise compute a vector V from endpoint with the smaller x value to the one with the larger
  - The direction of the normal to V, $N = \left(V_y, -V_x\right)$, is going towards points with $f(x, y) > 0$

Bresenham's Algorithm

- So we evaluate $f(M)$ in order to make our decision
- I like to think that since we want $f(x,y) = 0$ that we want to go the opposite way of $f(M)$ so that we're closer to $f(x,y) = 0$
- Therefore
  - If $f(M) > 0$ then the midpoint is below the lien so we should go NE →$(p_x + 1, p_y + 1)$
  - If$f(M) < 0$ so go E → $(p_x + 1, p_y)$



Bresenham's Algorithm

- But to compute the midpoint we had to divide by 2!

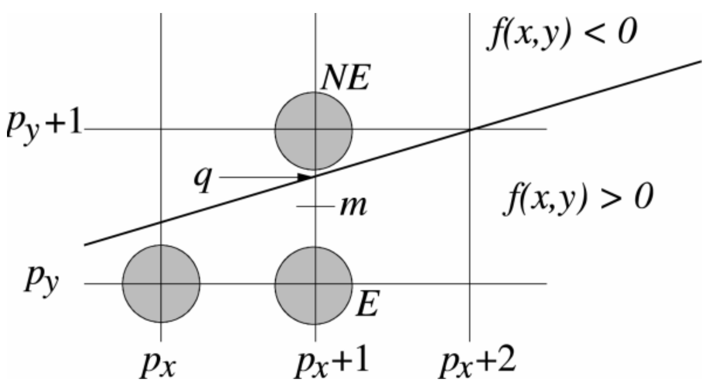
$$M_y = p_y + \frac{1}{2}$$

  - Wasn't the whole point of not doing DDA to avoid doing division/multiplication?
- Let's see how we can get around this…
- Returning to the implicit function of a line….

$$f(x, y) = ax + by + c = 0$$

- Lets modify this by multiplying each side by two:

$$f(x, y) = 2ax + 2by + 2c = 0$$

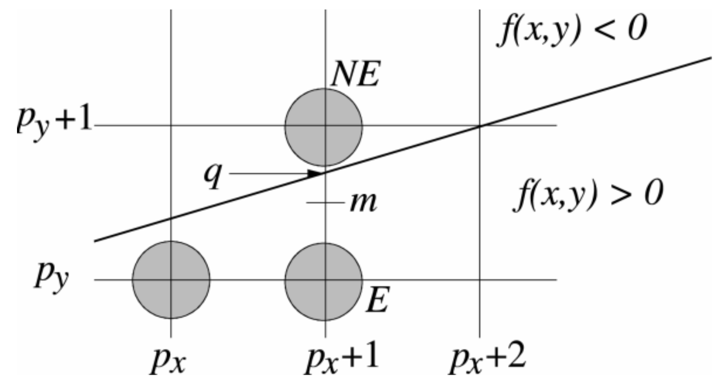


Pics/Math courtesy of Dave Mount @ UMD-CP

Bresenham's Algorithm

$$f(x, y) = 2ax + 2by + 2c = 0$$

- Now let's create a *decision variable* D where D is the value of f at the midpoint $M = \left(p_x + 1, p_y + \frac{1}{2}\right)$:
  - $f(M) = f\left(p_x + 1, p_y + \frac{1}{2}\right)$
  - $= 2a(p_x + 1) + 2b\left(p_y + \frac{1}{2}\right) + 2c$
  - $= 2ap_x + 2bp_y + (2a + b + 2c)$
- But what are a, b, c?



Bresenham's Algorithm

- Given two points q and r let
  - $d_x = r_x - q_x$ //the x component of the line
  - $d_y = r_y - q_y$ //the y component of the line
- From point slope form we have

$$(y - r_y) = \frac{d_y}{d_x}(x - r_x)$$

- Converting this to a general equation

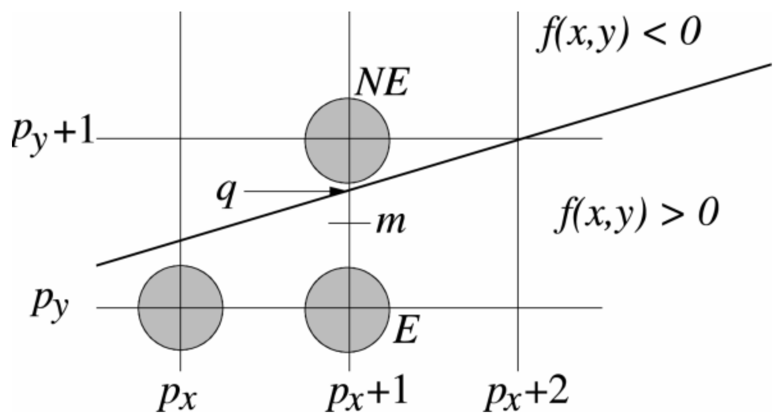
$$f(x, y) = ax + by + c = 0$$

  we get:
  - $f(x, y) = d_y x - d_y r_x - d_x y + d_x r_y = 0$
  - $f(x, y) = d_y x - d_x y + (d_x r_y - d_y r_x) = 0$
- So
  - $a = d_y$
  - $b = -d_x$
  - $c = d_x r_y - d_y r_x$

Bresenham's Algorithm

- Therefor for a line with slope $0 \leq m \leq 1$ our decision rule on where to go next is:

$$D = f(M) = 2d_y p_x - 2d_x p_y + \left(2d_y - d_x + 2(d_x r_y - d_y r_x)\right)$$

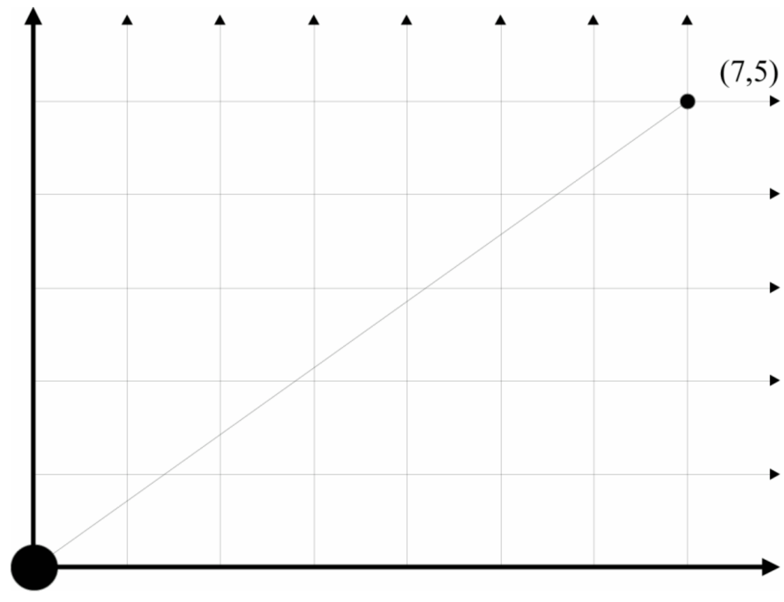


Generalized Algorithm

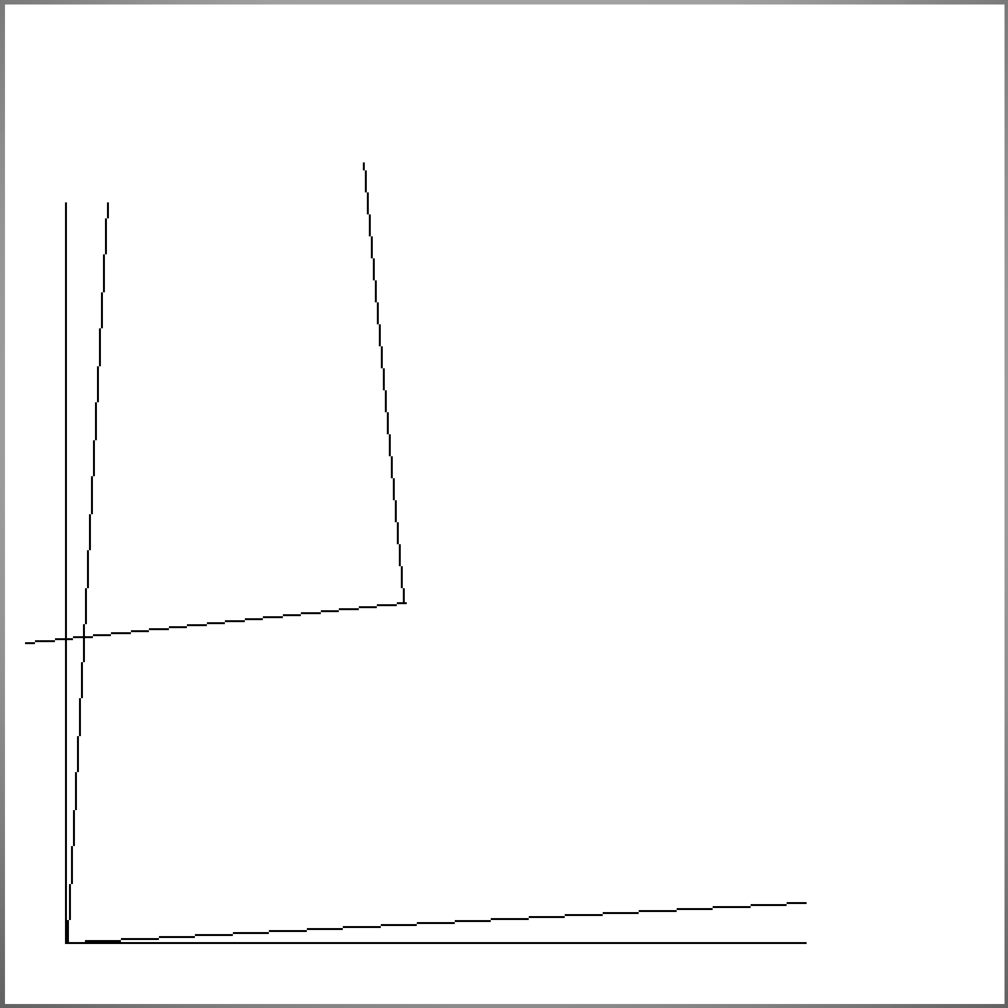
- 5 Different Cases:
  1. Vertical line
  2. $0 \leq m \leq 1$
     - If $q_x > r_x$, swap points
     - This is what we just did (always increment x, conditionally increment y)
  3. $m > 1$
     - If $q_y > r_y$, swap points
     - Always increment y, conditionally increment x
  4. $-1 \leq m < 0$
     - If $q_x > r_x$, swap points
     - Always increment x, conditionally decrement y
  5. $m < -1$
     - If $q_y > r_y$, swap points (vertical scan)
     - Always decrement y, conditionally increment x
- The $D = f(M)$ equations will be different for each case
  - Do the math!

Example

- Let's apply the algorithm for the line that's specified by endpoints (0,0) and (7,5)
- What's our values for a, b, c, d_x, d_y?
- Ok let's do it!



Assignment 1



Assignment Details

- Throughout this course we will
  - Read in data from a simple graphics form file, PostScript (.PS)
  - Simulate writing to a frame buffer by creating XPM images.
  - As we go we will
    - Draw lines
    - Draw polygons
    - Clip lines and polygons
    - Fill polygons
    - Simulate viewports
    - Allow for 2D and 3D transformations
    - Allow for 3D projection
    - Perform 3D depth cueing
    - Perform 3D depth buffering

Data Structures

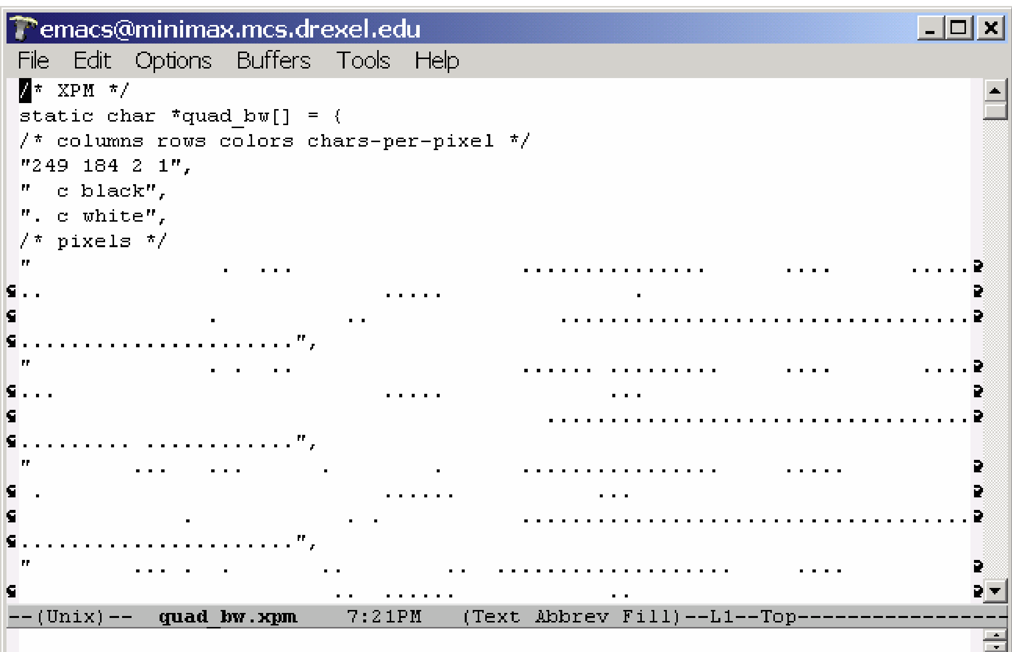
- Think about how to structure your assignments early
  - Need to read in augmented postscript file
  - Need to write out XPM file
- Also need data structures for
  - Frame buffer data
  - Drawable objects
    - 2D Lines
    - 3D Lines
    - 2D Polygons
    - 3D Polygons
    - 3D Camera objects
- Might be a good opportunity to flex your OOP skills
  - Inheritances, polymorphism etc..

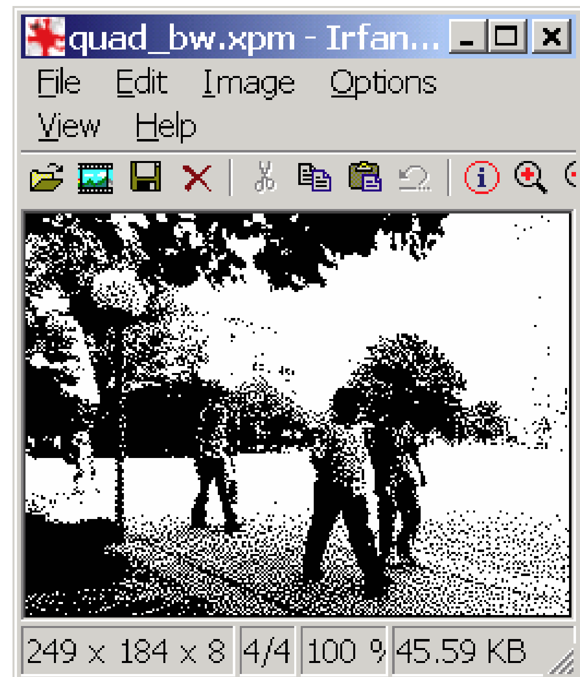
Programming Assignment 1

- Input PostScript-like file
- Output B/W (for now) XPM
- Create data structure to hold points and lines
- Implement line drawing

XPM Format

- Encoded pixels
- Actually C Code as an ASCII Text file





XPM Format

- View image (run code) as
  - Linux: display
  - Windows: Use IrfanView program (will also need all the plugins)
  - Mac : Install and run Xquartz. Then do ssh -x onto tux
- Or convert to image type that's more readable on your machine via:

  http://www.files-conversion.com/image/xpm

XPM Basics

- X PixelMap (XPM)
- Native file format in X Windows
- Files are C source code
- Read by compiler instead of a viewer
- Successor of X BitMap (XBM) B/W format
- Supports color

XPM: Defining Colors

- Each pixel specified by an ASCII char
- *Key* describes the context this color should be used within
  - Use “c” for color
- Colors are specified as:
  - color name
  - “#” followed by the RGB code in hex
- RGB – 24 bits (2 characters 0-f for each color)

XPM: Example

- Array of C strings
- The XPM format assumes the origin (0,0) is the upper-left hand corner
- First string is "`width height ncolors cpp`"
- Then you have "`ncolors`" strings associating characters with colors
- And last you have "`height`" strings of "`width`"* "`cpp`" characters

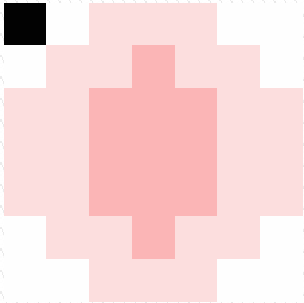
```
/* XPM */
static char *sco100[] = {

/* width height num_colors chars_per_pixel */
"7 7 4 1",

/* colors */
"- c #ffffff",
"# c #ffe0e0",
"a c #ffb7b7",
"x c #010101",

/* pixels */
"x-###--",
"-##a##-",
"##aaa##",
"##aaa##",
"##aaa##",
"-##a##-",
"--###--"

};
```



Simplified Post Script File Format

- We will only handle the command (for now!) of
  - `x1 y1 x2 y2 Line`
- The text of these commands are between the `%%%BEGIN` and `%%%END` markers
- Postscript assumes the origin (0,0) is at the lower-left corner of the window)
- Example:

```
%%%BEGIN
375 100 300 230 Line
499 0 0 250 Line
170 450 400 350 Line
350 300 120 400 Line
%%%END
```

Assignment Tasks

- Your program will allow you to
  - Read data from a PostScript file that specifies endpoints for lines.
  - Draw lines into a virtual frame buffer
  - Save the software frame buffer to an XPM file