CS 430 – Computer Graphics
Fall 2016
Assignment 1

In this assignment you will demonstrate your ability to:

1. Read in data from a file and organize it into structures for drawing
2. Set up a software frame buffer to render to
3. Implement a line drawing algorithm to render the lines into the software frame buffer.
4. Output the software frame buffer as an XPM file.

For this assignment you may assume that all endpoints that define the lines are within the bounds of the software frame buffer.

Make sure you give yourself adequate time. The programming components in particular can be quite time consuming.

As a reminder you may use the programming language of your choice, though I recommend C/C++ and also make sure that your program can run on the Drexel tux cluster to insure its system independence.

Submission Guidelines

1. Assignments must be submitted via Bd Learn
2. Submit a single compressed file (zip, tar, etc..) containing:
    a. A PDF file containing your answer to the theory question(s).
    b. A README text file (**not** Word or PDF) that explains
        i. Features of your program
        ii. Language and OS used
        iii. Compiler or interpreter used
        iv. Name of file containing main()
        v. How to compile/link your program
    c. Your source files and any necessary makefiles, scripts files, etc… to compile and run your program.

Theory Question(s)

1. What is the implicit equation of a(2D, line?) How can the equation be used to characterize the location of an arbitrary 2D point relative to the line? That is, how can it be used to tell where an arbitrary point is relative to the line? (4pts)

   The implicit equation for a 2D line is $f(x, y) = ax + by + c = 0$.

   To tell where an arbitrary point is relative to the line, given 2 end points, the algorithm chooses the integer y corresponding to the pixel center that is closest to the ideal (fractional) y for the same x; on successive columns y can remain the same or increase by 1. the algorithm keep track of the y coordinate (which increases by m = Δy/Δx, each time the x increases by one), the algorithm keeps an error bound at each stage, which represents the negative of the distance from the point where the line exits the pixel to the top edge of the pixel.

   Source: https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm

2. For the line segment specified by endpoints (10,7), (1,5) use the DDA algorithm to fill out the table below and then use its results to draw the line. (8pts)

X (floating point)	Y (floating point)	Pixel(X)	Pixel(Y)
2.0	5.22	2	5
3.0	5.44	3	5
4.0	5.66	4	6
5.0	5.88	5	6
6.0	6.10	6	6
7.0	6.32	7	6
8.0	6.55	8	7
9.0	6.77	9	7
10.0	6.99	10	7

(x1, y1) to (x2, y2) are (1,5) (10,7)

dx = x2- x1 = 10-1 = 9

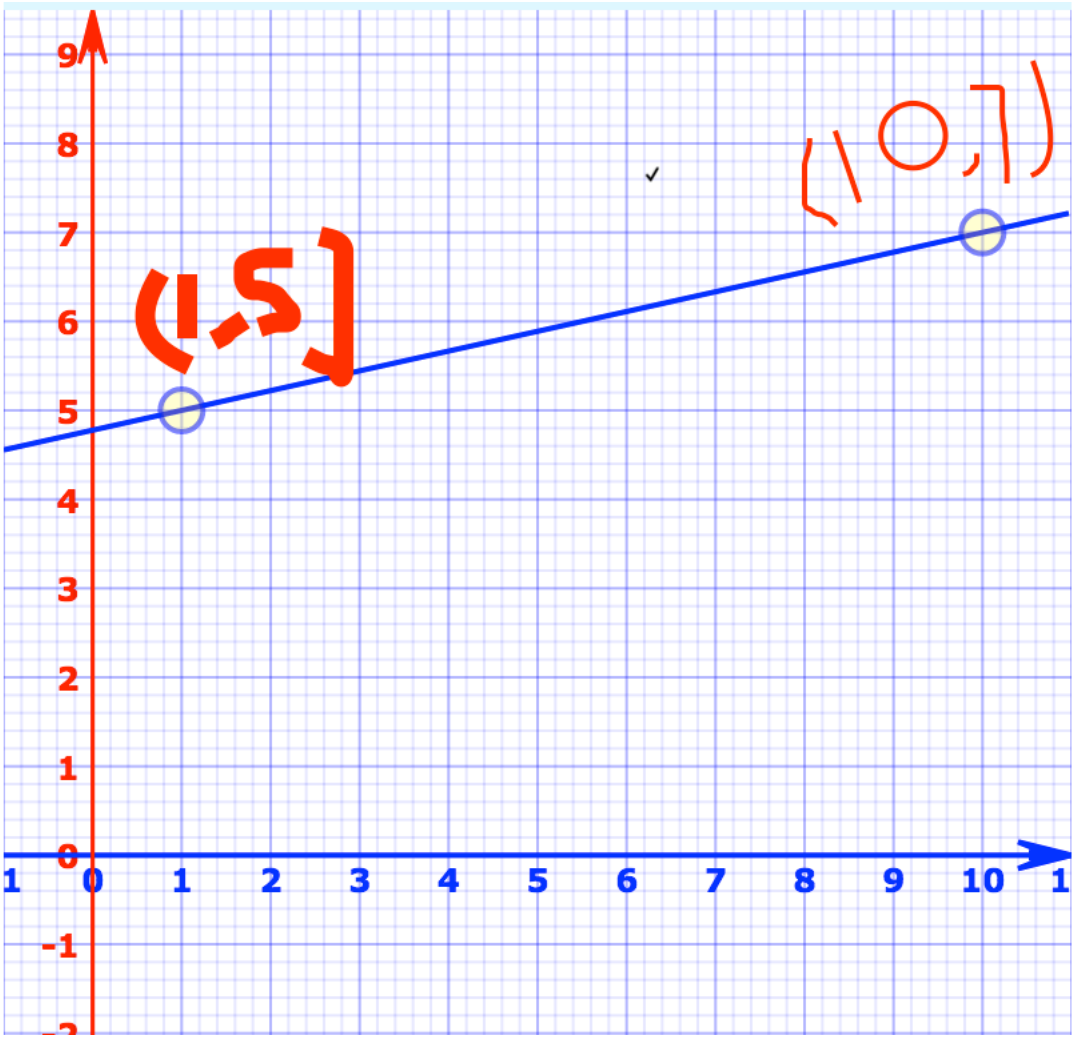
dy = y2 –y1 = 7-5 = 2

m = dy /dx = 2/9 = 0.22
since m is less than 1
deltaX = 1
deltaY = m = 0.22
xIncrement = x1 + deltaX
yIncrement = y1 + deltaY

First point
xIncrement = 1 + 1 = 2
yIncrement = 5 + 0.22 = 5.22

Second point
xIncrement = 2+1 = 3
yIncrement = 5.22+ 0.22 = 5.44

and so on…



3. For the line segment specified by endpoints (10,7), (1,5) use the Bresenham algorithm to complete the table below and use the table to draw the line. (8pts)

D (current)	Pixel(X)	Pixel(Y)	D (updated)
0	2	5.22	(2,5.22)
1	3	5.44	(3,5.44)
2	4	5.66	(4,5.66)
3	5	5.88	(5,5.88)
4	6	6.10	(6,6.10)
5	7	6.32	(7,6.32)
6	8	6.54	(8,6.54)
7	9	6.76	(9,6.76)
8	10	6.98	(10,6.98)

(x1, y1) to (x2, y2) are (1,5) (10,7)

dx = x2- x1 = 10-1 = 9

dy = y2 –y1 = 7-5 = 2

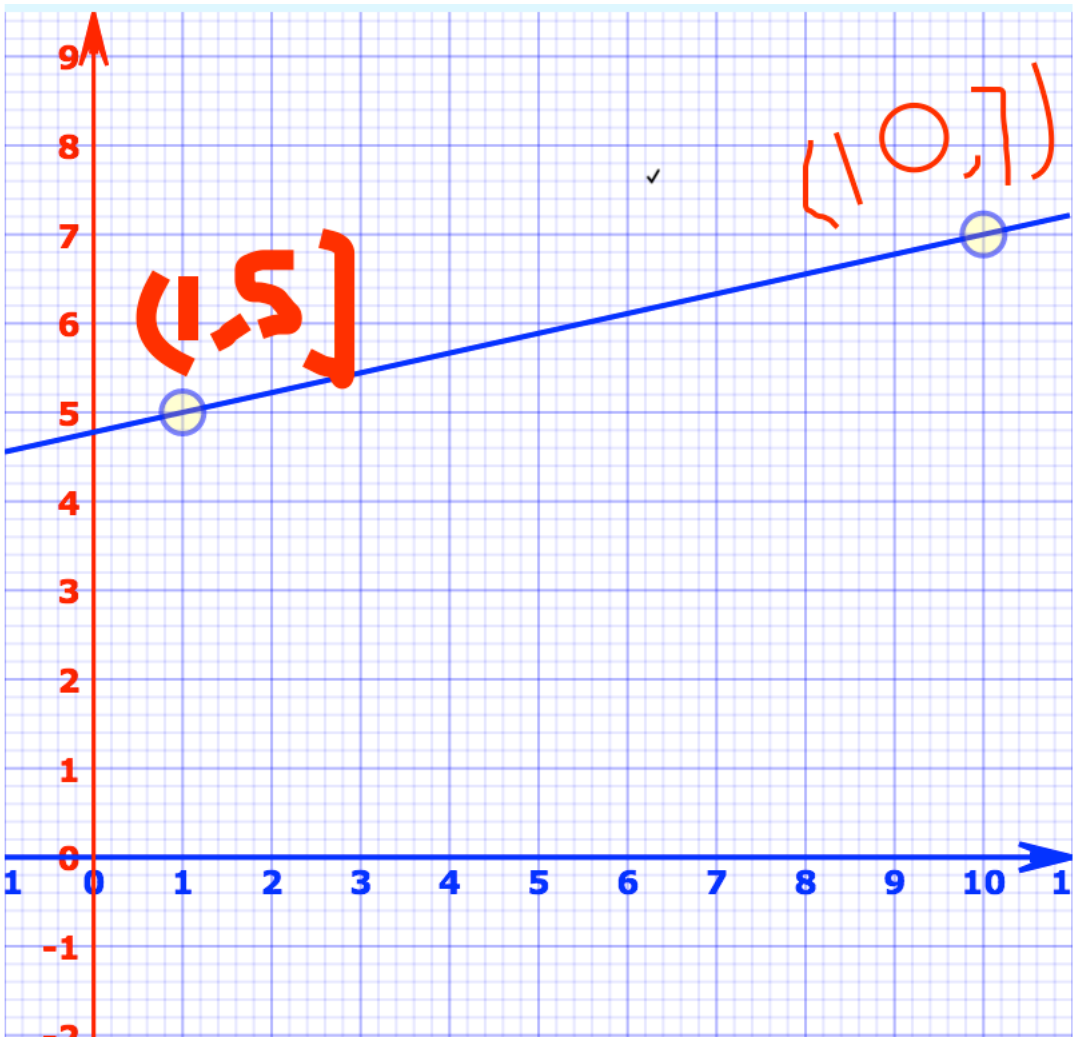
m = dy /dx = 2/9 = 0.22

Xk = dx/9 =9/9 = 1

Yk = dy/9 = 2/9 = 0.22

D	x	y	update(x,y)
0	1+1 = 2	5+0.22 =5.22	(2,5.22)
1	2+1 = 3	5.22+0.22=5.44	(3,5.44)
2	3+1 = 4	5.44+0.22 =5.66	(4,5.66)

and so on….


(1,5]
[10,7)

4. Derive the decision equation(s) (init, and the two directions) used in the Bresenham algorithm when the slope m<-1 (8pts)

Distance between pixel-to-right and ideal pixel is: d1 = y - y(k) and the distance between pixel-to-right-and-up and ideal pixel is: d2 = (y(k)+1) - y. So we can simply choose subsequent pixels based on the following:
if (d1<=d2) then choose pixel-to-right: (x(k)+1, y(k))
if (d1>d2) then choose pixel-to-right-and-up: (x(k)+1, y(k)+1)

If we evaluate d1-d2 as follows:
d1-d2 = y - y(k) - ((y(k)+1) - y) = y - y(k) - (y(k)+1) + y and now substitute using y = m*(x(k)+1) + b, we get
d1-d2 = m*(x(k)+1) + b - y(k) - (y(k)+1) + m*(x(k)+1) + b = 2*m*(x(k)+1) - 2*y(k) + 2*b - 1
The last equation can be reduced by the slope m and substituting as follows:
m = dY/dX where dY = abs(By - Ay) and dX = abs(Bx - Ax), so now we have
d1-d2 = 2*(dY/dX)*(x(k)+1) - 2*y(k) + 2*b - 1 or if we expand the first term (multiply), then:
d1-d2 = 2*(dY/dX)*x(k) + 2*(dY/dX) - 2*y(k) + 2*b - 1

This last equation can be simplified by creating a new decision variable P(k) = dX * (d1-d2). This will remove the divisions (all integer operations for faster and efficient computing) and will still keep the same sign for the decision because dX variable is always positive (dX = abs(Bx - Ax) - absolute value).
If we now evaluate a new decision variable P(k), we get:
P(k) = dX * (2*(dY/dX)*x(k) + 2*(dY/dX) - 2*y(k) + 2*b - 1) or further:
P(k) = 2*dY*x(k) + 2*dY - 2*dX*y(k) + 2*dX*b - dX

P(k) = 2*dY*x(k) - 2*dX*y(k) + 2*dY + 2*dX*b - dX or
P(k) = 2*dY*x(k) - 2*dX*y(k) + c where c is always constant value (it depends only on the input endpoints) and is equal to 2*dY + dX*(2*b - 1)

By evaluating an incremental change of the decision function P = dP = P(k+1) - P(k) we can evaluate by substitution dP as follows:
P(k+1) - P(k) = 2*dY*x(k+1) - 2*dX*y(k+1) + c - 2*dY*x(k) + 2*dX*y(k) - c
= 2*dY*x(k+1) - 2*dY*x(k) - 2*dX*y(k+1) + 2*dX*y(k)
= 2*dY*(x(k+1) - x(k)) - 2*dX*(y(k+1) - y(k))

Since we are processing pixel one by one in the x direction, the change in the x direction is (x(k+1) - x(k)) = 1, so if we substitute this into our dP derivation, we get:
dP = 2*dY - 2*dX*(y(k+1) - y(k))
For the y direction, there are two possibilities; the term (y(k+1) - y(k)) can be only 0 or 1, depending on if we choose pixel-to-right or pixel-to-right-and-up, so now our dP derivation looks like:
dP = 2*dY - 2*dX*(0) = 2*dY if pixel-to-right is chosen
dP = 2*dY - 2*dX*(1) = 2*dY - 2*dX if pixel-to-right-and-up is chosen

From line equation at the starting pixel y(0) = m*x(0) + b we get term for b intercept b = y - m*x(0).
Substituting b and slope m = dY/dX into equation P(0) we get:
P(0) = 2*dY*x(0) - 2*dX*y(0) + 2*dY + dX*(2*(y(0) - (dY/dX)*x(0)) - 1)
= 2*dY*x(0) - 2*dX*y(0) + 2*dY + 2*dX*(y(0) - (dY/dX)*x(0)) - dX

= 2*dY*x(0) - 2*dX*y(0) + 2*dY + 2*dX*y(0) - 2*dY*x(0) - dX
P(0) = 2*dY - dX

Reference:https://salonibaweja10.wordpress.com/tag/bresenhams-line-algorithm/

Introduction:

For the majority of our assignments we will be writing out into an image format called XPM and reading in from a simplified postscript (PS) file format. So first let's get an idea of what each of these plaintext files should contain…..

PS File Format

For most of the assignments you will need to read in files in the Simplified Postscript file format (*.ps). There are many commands supported in the .ps format, but in each assignment you will need to only support some small subset.

- The text of the supported commands will be bounded by two delimiters where %%%BEGIN marks the beginning and %%%END marks the end
- There may be blank lines anywhere between these delimiters
- All text outside these delimiters should be ignored
- Postscript assumes that the origin (0, 0) is the lower left corner of the window
    - Note this is different than XPM
- For this assignment you only need to support lines in the following format
    - x1 y1 x2 y2 Line
- Here's an example PS file
  %%%BEGIN
  375 100 300 230 Line
  499 0 0 250 Line
  170 450 400 350 Line
  350 300 120 400 Line
  %%%END

NOTE: Be careful that the input file is formatted for the OS you are using. One easy way to convert is to choose "Save As" in Textpad and choose our OS.

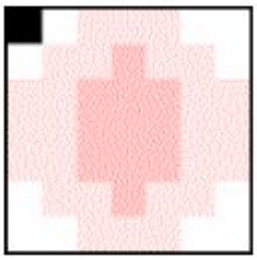
XPM Files Format

We will be writing XPM files to act as our "software pixel buffer" throughout the course.

You can find a brief description of it in the XPM handbook (you'll just need Chapter 1 and 2 at most)

- The XPM format assumes pixel (0, 0) is in the upper left-hand corner
    - This is different from most other applications that assume (0,0) is the bottom-corner. So you will have to compensate for that.
- Put data for "width height ncolors cpp" in the first string. For example:
    - /* width height num_colors chars_per_pixel */
    - "7 7 4 1"

- Then you have “ncolors” strings associating characters with colors. The character ‘c’ in the string associates the previous symbol in the string with the subsequent RGB hexadecimal color in that same string. For example:
    - /*colors */ ○ "- c
    #ffffff", ○ "# c
    #ffe0e0", ○ "a c
    #ffb7b7", ○ "X c
    #010101",
- And last you have “height” number of strings each with “width” * “chars_per_pixel” characters, where every “chars_per_pixel” length string must be one of the previously defined groups in the “colors” section. For example:
    - "X-###--",
    - "-##a##-",
    - "##aaa##",
    - "##aaa##",
    - "##aaa##",
    - "-##a##-",
    - "--###--"
- Here’s an example image /*
    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##-", "--###--"
    };



- Several free programs can view, read and write XPM files. One program for Windows is irfanview. You must install all the plugins. You can use gqview for Linux

The actual programming assignment!

Write a program that accepts the following command arguments. Defaults are in parenthesis. You should be able to process any subset of the options in any arbitrary order.

a. [-f] The next argument is the input "Postscript" file (hw1.ps)

For now we will hard-code the size of your frame buffer to be 500x500 (that is 500 pixels wide by 500 pixels high) and that you may assume that all the vertices specified in the file are within those bounds.

Your program should print output images in the XPM file format to stdout (`cout`) such that it can be piped to a file. All pixels should be initialized to white. Draw your objects in black.

Your general program flow should be:

1. Read in line segment data (PS)
2. Scan covert (i.e. draw) lines into software frame buffer using either the DDA or the Bresenham Algorithm.
3. Output your image (the frame buffer and header information necessary to make an XPM file) via `cout`

Bonus

For ten additional points, allow the user to pass a algorithm. If that flag is omitted, use DDA for line

`[-x]` flag to indicate to use Breshnham's line drawing drawing.

Grading Scheme:

In order to get any credit at all you must be able to generate at least read in a PS file and write out a valid XPM file

1. Theory Questions (28pts)
2. Can read in PS file and output a valid XPM file (35pts)
3. Successfully handles seven basic single line test cases (3pts each = 21pts)
4. Successfully handles a multi-line test case (10pts)
5. Program easy to compile and run based on README (6pts)
6. Bonus (10pts)

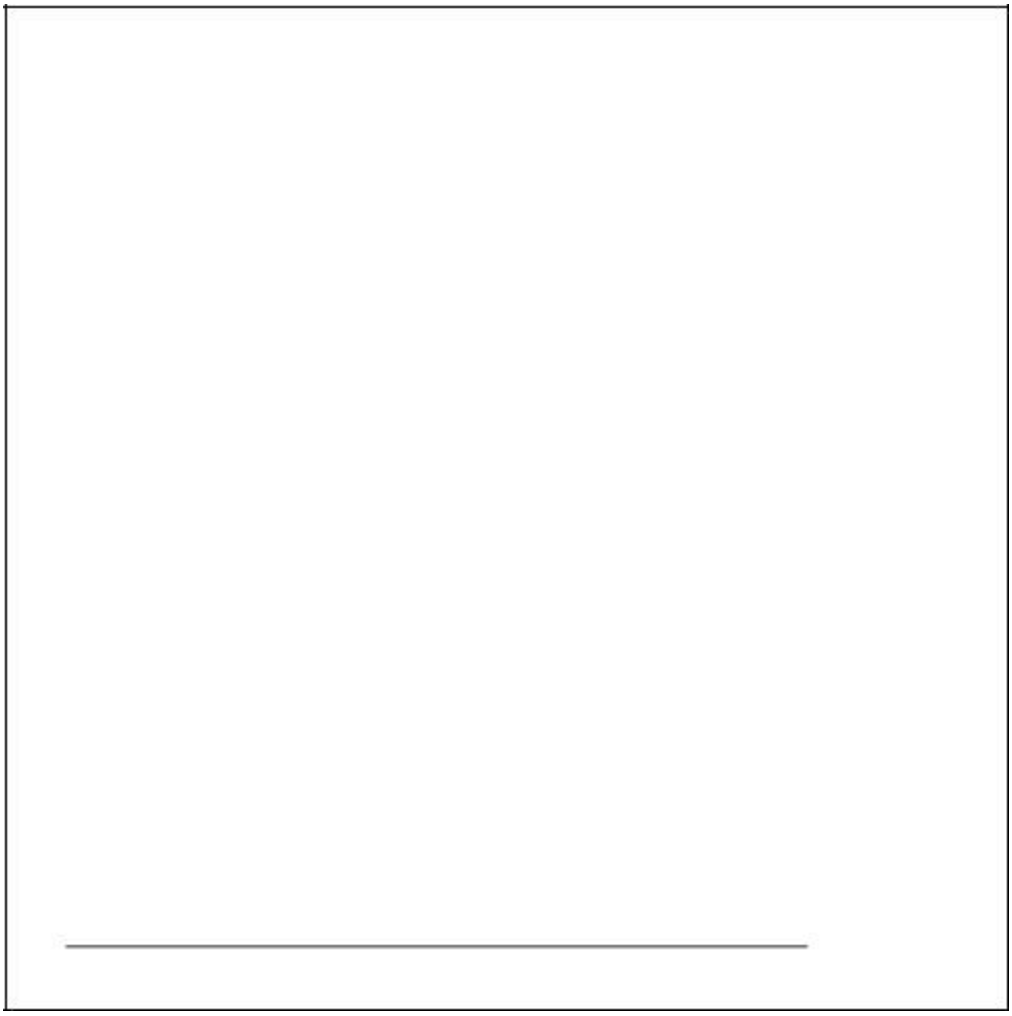
Common deductions:

1. Nothing drawn (-100pts)
2. Missing files (including readme) (-5pts each)
3. Cannot compile/run out-of-the-box on TUX (-10pts)

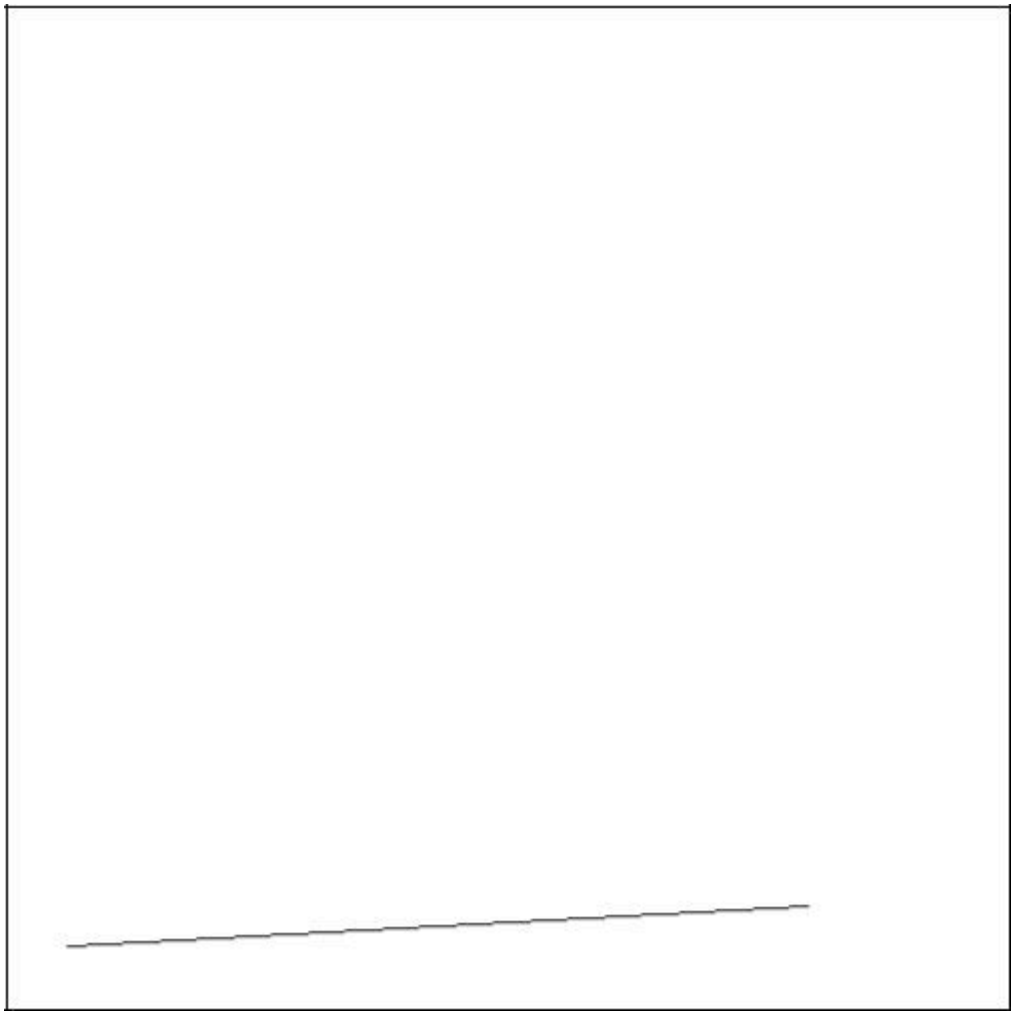
Example Output

Here is the output of a few of the test cases (note: border is not include in the actual output):

./A1 -f hw1_1.ps > out1.xpm



./A1 -f hw1_2.ps > out2.xpm



./A1 -f hw1_8.ps > out8.xpm