CMSC 202 — **Homework 2** — Spring 1996

Sections 101, 102 & 103 — **due: Tuesday, 20 February** — Handout #5

1 Background

A basic computer graphics task is the flood–fill task. The purpose of flood–fill is to fill an arbitrary bounded space on the screen with a given color (we will be using black). For example, it is a flood–fill algorithm that allows colorization of old black and white movies.

To run a flood–fill algorithm, you need a grid of pixels (which are simply dots on the screen) each of which can be either on or off, and a location within that grid at which to start filling. The pixel at the start location should be off. The flood–fill algorithm sets the start location to on, and all of its horizontal and vertical neighbors that are off, and all of their horizontal and vertical neighbors that are off, *etc.* This 'flooding' stops when a pixel that is already on is reached. You can think of the pixels that are initially on as walls, and of the start location as a hose pumping out water. The water fills as much area as it can, but it is stopped by the walls.

As an example, suppose that we were to flood–fill Figure 1 below, starting from the pixel marked with a diamond. The result is shown in Figure 2. Notice that the flooding cannot proceed diagonally; only the horizontal and vertical neighbors of a flooded cell that are not already on will be flooded:

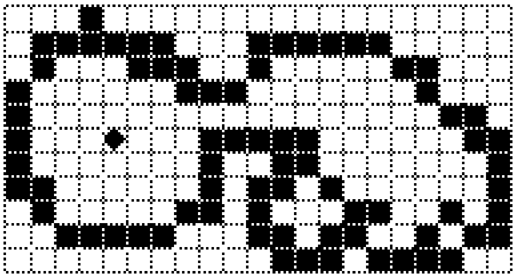

Figure 1

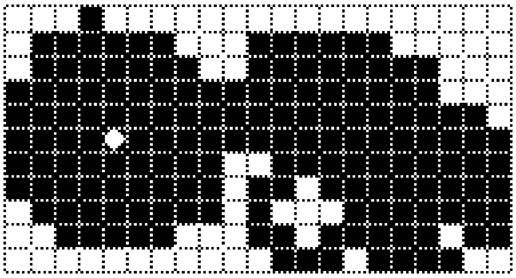

Figure 2

Assignment

You are to implement a flood–fill algorithm. You will write a program that reads in a 'screen' (*i.e.*, a two–dimensional array of pixels) from a file, asks for a point on that screen from which to commence flood–filling, runs the flood–fill algorithm from that point, and writes out the resulting screen to another file. The program also allows the user to view a version of the picture on the screen, using user–selected ASCII characters for pixels.

The purpose of this assignment is to give you experience using recursion. Therefore, your program may not use any of C's iteration constructs (`do`, `while`, `for`, and `goto`); all repetition must be accomplished using recursion. This applies to *every* operation in the program, even file operations.

Example

Here is an example of how your program should work (text entered by the user is shown in *italics*):

```
umbc8[22]% hw2
What is the name of the input PBM file? example.pbm
This screen is 5 pixels wide by 4 pixels high.
Do you want to see the input [n]: y
Please enter characters for '1' and '0': * -
****-
*---*
**--*
*****
Please enter the x and y coordinates of the start pixel: a 5
You must enter two integers; `a 5' is not valid.
Please enter the x and y coordinates of the start pixel: 5 a
You must enter two integers; `5 a' is not valid.
Please enter the x and y coordinates of the start pixel: 8 3
The first integer must lie between 0 and 4, inclusive.
Please enter the x and y coordinates of the start pixel: 0 5
The second integer must lie between 0 and 3, inclusive.
Please enter the x and y coordinates of the start pixel: 0 0
That pixel is already set.
Please enter the x and y coordinates of the start pixel: 2 1
What is the name of the output file? outexample.pbm
Output written to file outexample.pbm
Do you want to see the output [n]: y
Please enter characters for '1' and '0': @ +
@@@@+
@@@@@
@@@@@
@@@@@
umbc8[23]% more outexample.pbm
P1
5 4
1 1 1 1 0
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
umbc8[23]%
```

Note that the pixels are numbered from the upper lefthand corner of the screen, which is 0,0. Your program should verify that the selected start pixel is indeed off, and print a warning message if it is not.

The PBM Format

We will be using a file format called PBM, which stands for Portable Bit Map. A PBM file has three components. The first item in the file is the pair of characters '`P1`'; this serves to identify the file as a PBM file. The second item in the file is a pair of integers that indicate the size of the screen: first the width, then the height (*i.e.*, first the number of pixels in each row, then the number of rows). The remaining lines of the file contain one digit for each pixel in the image. A '`1`' means that the pixel is on, while a '`0`' means that the pixel is off. Here is the sample file used above in PBM format:

```
P1
5 4
1 1 1 1 0
1 00  0 1
11 00 11 111 1
```

Note that neither the rows nor the bits in those rows are required to be separated by whitespace. In particular, a newline is not required after each row. Complete details about the PBM format may be obtained by using '`man pbm`'.

Two other points should be made about the PBM format. First, characters from a # to the end of the line are supposed to be ignored in a PBM file; however, we will not allow such comments in our PBM files. Second, no line is supposed to be over 70 characters long. We will ignore this requirement; however, if you decide you want to use the output of your program as the input to other programs that manipulate PBM files, you might want to observe it.

Error Checking of Input

The following errors should cause program termination after an appropriate message is printed to `stderr`:

- The input PBM file cannot be opened for reading.
- The output PBM file cannot be opened for writing.
- The screen represented by the input PBM file exceeds the maximum allowed size (which we will set at 100 pixels wide by 100 pixels high).

The following errors should cause a warning message to be printed to `stderr`, but should not cause program termination:

- The start pixel is already `on`.
- The start pixel is not a legal location on this screen.
- An input item is not correct (*e.g.*, expected a `y` or an `n` but got something else; expected an integer but got a non–integer, *etc.*).

Resources

I have created a number of PBM files on which you can test your algorithms. They are located in the directory *~mayfield/pbm*, and they all end in the extension '*.pbm*'. More test files may be found in `~anastasi/Public/202/Homework/HW2/PbmFiles/`.

There are a couple of tools that you might find useful when working on this assignment. The '`xv`' program can display pictures that are stored in a variety of formats, including PBM. Because the standard window manager imposes a minimum window width, very small PBM pictures will be stretched when they are displayed by `xv`. You can unstretch such pictures by changing the size of the `xv` window (for instance, by dragging the lower righthand corner of the window with the mouse).

Another tool you might find useful is the '`bitmap`' program. Bitmap allows you to draw a picture on a bitmap screen and flood–fill it. You can use `bitmap` to experiment with flood–filling, to verify that your program produces correct results, and to create new bitmaps for use as input to your program. Unfortunately, `bitmap` does not save its results in PBM format. It uses instead an alternative format called XBM. You can convert a PBM file into an XBM file using the '`pbmtoxbm`' program:

```
umbc8[24]% pbmtoxbm myfile.pbm > myfile.xbm
```

Converting a file from XBM to PBM format is slightly more complicated. First, you must use the '`xbmtopbm`' program to convert from XBM format to a packed version of the PBM format. Then, you use the '`pnmnoraw`' program to convert the packed PBM file into an ordinary PBM file such as those with which we are familiar:

```
umbc8[25]% xbmtopbm myfile.xbm > myfile.junk
umbc8[26]% pnmnoraw myfile.junk > myfile.pbm
umbc8[27]% rm myfile.junk
```

The directory *~mayfield/pbm* contains XBM versions of each sample PBM file, so that you can play with them using `bitmap`.

2 File Structure

Your program must be organized in a modular way. It must be segmented into multiple files. The main function should appear (with little else) in its own file (suggestion: name it `hw2.c`). Other functions must be declared in an appropriate interface file (for example, `hw2_aux.h`, `pdm_ops.h`, *etc.*) and be defined in a corresponding implementation file. The interface files must be **guarded**.

Hints

The flood–fill task itself is a natural for ordinary recursion. You should use tail recursion to read and write the files. Refer to Handout 4 for more information about tail recursion. You should write and test the code to read the input file before attempting to write the rest of the program. If you are having great difficulty writing the necessary code directly, you may try implementing the code using iteration statements, then translating your code into recursive code. Don't submit the iterative code though—you will get no credit for it.

Submission

Submission of this homework will be electronic, using the `submit` program. The submission class is `cs2021`; the submission assignment is `hw2`. You must submit all of your `.c` and `.h` files, together with your Makefile. The Makefile must correctly compile your homework upon issue of the `make` command with no arguments. You may name your files as you wish, but the executable created by the `make` command must be named `hw2`. Your Makefile will be used to compile your submitted code, so be sure it works. A simple Makefile that just does the compilation is fine.

We will automatically compile and run your program on a fixed set of data and commands. You may submit your program more than once, but only the last version you submit will be graded. Submissions made by midnight on the due date (Tuesday, 20 February) will receive the on–time bonus. You may have an automatic extension until Thursday, 22 February; submission of homework will be disabled at midnight on the 22^{nd}.

Make sure that you adhere to the I/O specifications given in this handout (*i.e.*, the order of inputs to your program—the precise wording of the program's questions to the user is unimportant). If you do not adhere to these specifications, the automatic portions of the grading process will fail, and you will receive a poor grade. Don't forget to include a comment at the top of your program stating your sources of help (as described in the Prospectus).

Checklist

You should check to make sure that you have fulfilled each of these requirements:

- Did not use any C iteration constructs (`while`, `for`, *etc.*).
- Wrote in modular form, using implementation and interface files.
- Submitted a working Makefile.
- Submitted all source files needed to compile the program.
- All error messages are written to `stderr`.
- All other messages are written to `stdout`.

- "ASCII version" of PBM files are written to `stdout`.
- Program terminates, with error message on the following conditions:
  - Can't open a PBM file (in or out).
  - Input PBM file exceeds the maximum screen size.
- Program prints error message, but does not terminate on the following conditions:
  - User input is not correct format.
  - Start pixel is already `on`.
  - Start pixel is "off the screen."
- Tested the program against test files.
- Followed all style guidelines.