ECE220: Computer Systems & Programming **Fall 2022 ZJUI**
Machine Problem 8 **due: Sunday 20 November, 11:59:59 p.m.**

Flood Fill with Recursion

Your task this week is to implement several varieties of flood fill operations using recursion. These operations can be applied to sample images. The objective for this week is for you to gain some experience with recursion by solving a problem using programming knowledge gained from the class.

Read this document completely before you begin so that you know how to use what you've been given.

Background

An image is composed of an array of pixels. Each pixel has three bytes of information in the RGB format. These bytes represent the red, blue, and green levels of the pixel. The values for each pixel range from 0 to 255.

In a flood fill operation, the user selects one pixel in the image along with an RGB color. In a painting analogy, the chosen color floods out in all directions from the chosen pixel until it reaches a boundary. In the simplest case, such a boundary might simply be a pixel of another color, such as white.

One can implement flood fill recursively, although the number of stack frames needed may be too large for typical image sizes, so you will implement a version that limits the range of the flooding to about 35 pixels, thus limiting the total number of pixels to about 3,850.

The Image Arrays

Your functions will be given arrays for each of the three channels in the image: red, green, and blue. These arrays will be passed as pointers to 8-bit pixels (that is, as `uint8_t*`). However, they represent two-dimensional arrays of specified `height` and `width` (these values are also provided to your functions). To make use of the arrays, you must calculate the appropriate offset into the one-dimensional C arrays based on the two indices of the pixel that you want to access. Recall from class that you should make use of the expression `x + y * width` to access the pixel at (x, y). For example, given an array `uint8_t* red` for the red channel, you can access the (x, y) pixel in the array with the expression `red[x + y * width]`.

Pieces

In addition to code, we have provided a few sample images in the `Images` subdirectory. The `Examples` subdirectory contains a few images processed by the "gold" version of the program (the version that I write before asking you to do the assignment). The images' names tell you the arguments to pass when you run the program.

There is more code in the package this time, but you still need to look at only three files:

`mp8.h` This header file provides function declarations and descriptions of the functions that you must write for this assignment.

`mp8.c` The source file for helper functions, including wrapper functions that call the recursive functions in `mp8recurse.c`. Function headers for all functions are provided to help you get started.

`mp8recurse.c` The source file for your recursive functions. Function headers for all functions are provided to help you get started.

Other files provided to you include

`main.c` A source file that interprets commands and calls your functions.

`Makefile` A file that describes how to build your program, allowing you to type "`make`" instead of re-typing the full compiler command every time.

`imageData.h` `imageData.c` Some utility functions written by a former TA for the class.

`lodepng.h` `lodepng.c` A package for loading and storing PNG images.

You need not read any of these, although you are welcome to do so.

Details

You should read the descriptions of the functions in the header file and peruse the function headers in the source file before you begin coding. Here are the tasks that you need to complete for this assignment:

1. Write a recursive flood fill, stopping at white pixels.
2. Write a second recursive flood fill that stops at grey (near-white) pixels.
3. Write a recursive flood fill that stops at any color far enough from the color at the initial flood point, and is limited to a 35-pixel range.

The structure for each of these tasks is almost identical, and involves implementing one function (per task) in the `mp8recurse.c` file and one function (per task) in the `mp8.c` file. The first function is a recursive flooding function that uses an array to track its progress (and to avoid infinite loops). The second function is a wrapper function that prepares the array to track the flood's progress, calls the recursive flooding function, and then uses the progress array to create the final image from the original image. For tasks 2 and 3, you must also create a small helper function (in `mp8.c`) that decides whether two colors are within a certain distance of one another. The total is thus seven functions.

All function signatures appear in `mp8.h`. The wrapper functions and utility function must be written in `mp8.c`. For the wrapper functions, the first five parameters to each describe the input image: the width in pixels, the height in pixels, and the arrays containing the red, green, and blue channels of the pixels. The next five or six parameters describe the flood: the starting x and y position, the red, green, and blue components of the flood color, and (for tasks 2 and 3 only) the distance squared value. Finally, the last three parameters are arrays through which your wrapper functions define the red, green, and blue color channels for the final output image.

The recursive functions that your wrapper functions must call appear in `mp8recurse.c`. The prefixes of the names match those of the wrapper functions, so `basicFlood` should call `basicRecurse`, and so forth.

Step 1: The Wrapper Structure

Start by writing `basicFlood`, the wrapper function for the first task. The wrapper should use `outRed` to record whether each pixel has been visited by the flood. Initially, no pixels have been visited, so the first step is to fill the array with 0s (use `memset`). Then call `basicRecurse` (once). Finally, walk through the whole image: pixels marked in the marking array (`outRed`) should be filled with the flood color.

Pixels not marked in the marking array should be copied from the original image. Keep in mind that you are overwriting `outRed` as you fill in the final image.

Step 2: Your First Recursion

Now you're ready for your first recursive function! Be sure to get the stopping conditions right, or your program will crash due to infinite recursion. Use the marking array to mark any pixel visited, and check it before making any recursive calls. Check adjacent pixels in the following order: up (negative y), right (positive x), down (positive y), and left (negative x). Don't recurse into white pixels (RGB = 0xFFFFFF).

When you complete this function, you can compile the program and use command "1" to test and debug your two functions. The file `Images/E.png` has mostly true white pixels. The files with pictures of numbers (`Images/seven.png` and `Images/eight.png`) do not have as many—see what happens with your fill.

Step 3: Defining a Distance Metric for Color

People who care about color do not use Euclidean distance in RGB space. People who want simplicity for a programming assignment do, however. Your next step is to write the short function `colorsWithinDistSq` in `mp8.c`. This function must return 1 if the two RGB colors are within the specified distance squared of one another. In other words, if $(R_1 - R_2)^2 + (G_1 - G_2)^2 + (B_1 - B_2)^2 \leq (\text{distance})^2$. Otherwise, the function should return 0. Note that you do not need to take any square roots.

Step 4: Stopping Near White

Now that you have a way to measure distance in RGB space, you can write a new wrapper (`greyFlood`) and a new recursive routine (`greyRecurse`) that stop when they run into pixels that are within a given parameter (the new `distSq` parameter) of white (RGB=0xFFFFFF). Use `colorsWithinDistSq` to check.

Once those functions are complete, you can test them by compiling and using command "2". To get a feeling for the actual variation within an image that looks like a white boundary at first glance, try using `Images/eight.png` as an input, starting the flood at (20,32) for `eight` and (32,10) for `seven`, and seeing what happens with various `distSq` values.

Step 5: Stemming the Flood

You are now ready to write the final version of the flooding algorithm, which stops when it goes more than 35 pixels away from the starting point (make this check the first stopping condition, do NOT use square roots, and do not mark pixels that are too far away). This version also stops when the RGB color goes too far away from the color at the starting point (use the `colorsWithinDistSq` function again, but in the opposite sense). You also need a new wrapper (`limitedFlood`), but the code only differs in that it calls `limitedRecurse`.

Once these two functions are complete, you can test them using command "3". Now you can see how close to white (RGB=0xFFFFFF) the background in `Images/numbers.png` really is by flooding at (100,100) with distance squared values of 10 and 100. Try (90,100), too. Or look at some regions in `Images/tajmahal.png`, such as (300,430) versus (300,440) with distance squared of 500.

The Interface

When you have completed one or more of the functions, you can type "`make`" (no quotes) to compile your code into an "`mp8`" executable program. The make program shows you the compiler commands that it executes on your behalf, so you can see what is happening and will receive any syntax errors or other errors or warnings from the compiler. Fix any warnings!

Once you have compiled your program, you can use the interface built into `main.c` to execute your functions. A set of images has been provided to you in the `Images` subdirectory along with several copies processed by a correct version of the program in the `Examples` subdirectory (take a look).

The `mp8` program takes seven or eight command line arguments:

```
./mp8 <input file> <output file> <command #> <x> <y> <RGB> [<distSq>]
```

The commands are the same as the task numbers above. The flood starts at (x, y), which must be within the chosen input image, and fills with the color specified by RGB (a 3-byte hex value; for example, "AABBCC" means 0xAA for red, 0xBB for green, and 0xCC for blue). For the second and third tasks, you must also specify the distance squared that defines the boundary of the flooding.

Specifics

- Your code must be written in C and must be contained in the `mp8.c` and `mp8recurse.c` files in the **`mp8`** subdirectory of your repository. Functions must appear in the correct files, as in the distributed versions. We will NOT grade files with any other names, nor will we grade files with functions moved between the files. **Changes made to any other files WILL BE IGNORED during grading.** If your code does not work properly without such changes, you are likely to receive 0 credit.
- You must implement the `basicFlood`, `basicRecurse`, `colorsWithinDistSq`, `greyFlood`, `greyRecurse`, `limitedFlood`, and `limitedRecurse` functions correctly.
- Your routines return values and outputs must be correct.
- Your code must be well-commented. Follow the commenting style of the code examples provided in class and in the textbook.

Grading Rubric

We put a fair amount of emphasis on style and clarity in this class, as reflected in the rubric below.

Functionality (85%)

- 10% - `basicFlood` works correctly.
- 20% - `basicRecurse` works correctly.
- 5% - `colorsWithinDistSq` works correctly.
- 10% - `greyFlood` works correctly.
- 15% - `greyRecurse` works correctly.
- 10% - `limitedFlood` works correctly.
- 15% - `limitedRecurse` works correctly.

Comments, Clarity, and Write-up (15%)

- 5% - introductory paragraphs explaining what you did for files and functions (even if it's just the required work)
- 10% - code is clear and well-commented, and compilation generates no warnings (note: any warning means 0 points here)

Note that some categories in the rubric may depend on other categories and/or criteria. For example, if you code does not compile, you will receive no functionality points. If your `colorsWithinDistSq` function fails, tasks 2 and 3 are also likely to fail. As always, your functions must be able to be called many times and produce the correct results, so we suggest that you avoid using any static storage (or you may lose most/all of your functionality points).