

CSC 112 Lab 4: Image Processing

Fall 2009

Due: Friday, September 25th, 9:00am

Purpose

This lab is intended to:

- Provide a chance to work with dynamic memory allocation, particularly array allocation
- Reinforce the notions of function calls, parameter passing, and looping control structures
- Introduce simple file input/output and command line arguments
- Provide an interesting problem that provides immediate feedback on program correctness

Introduction to PPMA Images

At the simplest level, every image representation needs the following information: image height, image width, number of color values, and, for each picture element (pixel), the red, green, and blue color channel values. The PPMA file format is an ASCII file format that represents this information as follows:

```
P3
Width Height
MaxColorValue
Red Green Blue   Red Green Blue   Red Green Blue   Red Green Blue
Red Green Blue   Red Green Blue   Red Green Blue   Red Green Blue
...
```

For example, the following image is a 4x2 (width 4, height 2) alternating black and white image.



and its PPMA representation is:

```
P3
4 2
255
0 0 0 255 255 255 0 0 0 255 255 255
255 255 255 0 0 0 255 255 255 0 0 0
```

Notice that black is all 0's (no expression for all colors) and white is all 255's (full expression of all colors).

For this lab, I have provided some example PPMA images on the class webpage. If you prefer, feel free to download jpg images of your choice. Compiling the program `ppmb_to_ppma.cpp` and running it as part of the script `jpeg2ppma.sh` using the command line shown below will convert a jpeg to a ppma file:

```
./jpeg2ppma.sh      inputFileName.jpg  outputFileName.ppma
```

To compile `ppmb_to_ppma.cpp`, run:

```
g++ -o ppmb_to_ppma ppmb_to_ppma.cpp
```

`jpeg2ppma.sh` uses `djpeg` (should already be on your system) to decompress a jpeg file to a binary ppm, and then uses `ppmb_to_ppma` to convert the binary ppm to an ASCII ppm.

To view PPMA images, you can use `gimp` (likely already installed on your system),

```
gimp filename.ppma &
```

or `gqview` (likely needs to be installed: `sudo apt-get install gqview`)

```
gqview filename.ppma &
```

Programming Task #1: Input/Output

Download the source code `lab4.cpp`, which is the shell of a program I have begun for you. You should begin this lab by completing the two functions:

```
void readPPMAFile(string filename, image, width, height,
maxColor)
```

```
void writePPMAFile(string filename, image, width, height,
maxColor)
```

Notice that I am leaving it up to you to set the type and parameter passing technique for the variables *image*, *width*, *height*, and *maxColor* for these functions.

Given a string as a filename, `readPPMAFile` should open the file, read each of the ASCII entries (assuming the file format described on page 1), assign values to any variables as appropriate, dynamically allocate an appropriately sized multi-dimensional array to hold the image, and read the red/green/blue values into the array.

Given a string as a filename, `writePPMAFile` should open the file and write out the image using the file description provided on page 1. Note, there is a requirement that no more than 70 characters be on each line of the output file. It is common to just print the representation of 4 pixels per line

Assume the filename provided will be the filename of an existing file, and that the file you are given is a valid PPMA file.

I have already written the code to open and close files for you. A quick explanation of that code is as follows.

For input, one needs to open a file input stream:

```
ifstream inputStream;          //ifstream requires #include <fstream>
inputStream.open(filename.c_str());

// code to read file

inputStream.close();
```

Given that, it is possible to read in from the file just as if you are reading in from a user (using the >> operator), except one uses inputStream instead of cin.

For output, open a file output stream

```
ofstream outputStream;
outputStream.open(filename.c_str());

// code to write file

outputStream.close();
```

Given that, it is possible to write to the file just as if you are writing to the screen (using the << operator), except one uses outputStream instead of cout.

Programming Task #2: MaxMin Filter

Write a function that applies a “max-min” filter to an image. This function should take in the image array, image width, image height, and the max color value.

This filter employs the following technique:

For each pixel:

For each red, green, blue color entry:

If the value of that color entry is greater than or equal to half the maximum color value, set the color entry to the max color value.

Otherwise, set the color entry to 0.

This filter tends to “blow-out” an image – significantly darkening already dark areas and significantly brightening light areas. An example of its application is shown on the next page.

Before filter applied:



After filter applied:



Programming Task #3: Smoothing Filter (Mean and Median)

Write a function that applies a “smoothing” filter to an image. This function should take in the image array, image width, image height, and an integer “option” variable.

If the option variable is 0, you should execute instructions that implement a *mean* smoothing filter. If the option variable is set to 1, you should execute instructions that implement a *median* smoothing filter. Both mean and median smoothing filters employ the same techniques, except for one step.

This filter employs the following technique:

Ask the user what distance they would like to use in smoothing

For each pixel:

For each red, green, blue color entry:

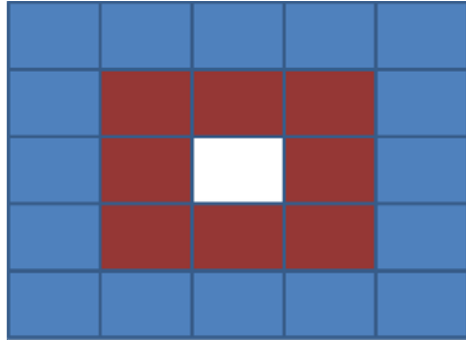
Collect the corresponding color entries for each of the pixels that surround, within the specified distance, the pixel of interest, as well as collect the color entry of the pixel of interest

Replace the color entry of the pixel of interest with a value computed on the collection of color entries.

For the mean smoothing filter, the replacement is: assign to the pixel of interest the (integer) mean value of the collection

For the median smoothing filter, the replacement is: assign to the pixel of interest the median value of the collection.

In the image on the next page, the 8 pixels highlighted in red, as well as the pixel of interest highlighted in white in the middle (for a total of 9 pixels), would be used in computing the mean/median for a user input distance of 1, while the 16 pixels highlighted in blue, the 8 in red, and the 1 highlighted in white (for a total of 25) would be used for a distance of 2. This bounding box continues to grow for larger distances.



Note that there is a function to compute the mean of an array and a function to compute the median of an array provided to you. Also note – each pixel requires looking at other neighboring pixels. You want to use the neighbor’s original pixel color values in updating a pixel of interest, not the neighbors updated color values (if the neighbor has already been updated).

These filters tend to smooth images – reducing the sharpness and contrast between areas in the image. The median filter degrades the image faster than the mean filter (because it uses less information). The greater the distance, the faster the image degrades as well (less local information).

Before filter applied:



After mean, distance 1 filter applied:



After mean, distance 2 filter applied:



After median, distance 1 filter applied:



After median, distance 2 filter applied:



Programming Task #4: main Function

Values can be sent into a program when it is executed on the command line. As an example, when we use `g++`, we send in several command line arguments:

```
g++ -o lab4 lab4.cpp
```

The system tells the main function for `g++` that it has four command line arguments:

- 1) `g++` // note this is the name of the program
- 2) `-o`
- 3) `lab4`
- 4) `lab4.cpp`

Remember that for the main function, there are two parameters: `int main(int argc, char* argv[])`

For the `g++` example above, `argc` (argument count) is set to 4, and the array of character strings called `argv` holds the following values [`g++` | `-o` | `lab4` | `lab4.cpp`] in `argv[0]` through `argv[3]` respectively.

For this lab, the user will enter the input filename of interest, output filename of interest, and the filter to apply on the command line. As an example, the user should be able to type: `./lab4 image.ppm imageAfterFilter.ppm maxmin`

If this is how the program is called, the file `image.ppm` should be loaded, the `maxmin` filter applied, and the resulting image saved out to the file `imageAfterFilter.ppm`.

Allow the user to enter the following filter options: *maxmin*, *meansmooth*, and *mediansmooth*. Code has already been written for you to read these values into strings in the main function. String values can be tested for equality using the `==` operator so that you can choose which functions to execute.

Completing the Lab

Please adhere to all of the following points to receive the maximum credit for this program:

- Turn-in electronically the one C++ source file for this program. Submit your `lab4.cpp` file under the “Lab 4” section of the Assignments Menu on Blackboard by the due date and time. You are not required to tar and gzip your code this time (since it’s only one file).
- Assume valid inputs.
- Use dynamic memory allocation for this program. You are responsible for deleting any memory you dynamically allocate.
- Adhere to documentation and coding style standards