EECS402, Fall 2021, Project 3

Overview:

Working with, and modifying pictures on a computer is big business. Images might be modified by performing image sharpening algorithms in order to better make out a criminal's face from a fuzzy surveillance camera photo. A company might charge a small fee for removing the "red eye" problem that flash photography suffers from, so that you can make better prints from your photos. Team members located in different cities might "mark up" an image during teleconferences in order to share their thoughts on images or graphs. And sometimes, you just want to be able to put conversation bubbles on a photo to add some humor. Each of these situations requires knowledge of how computers deal with imagery. This project will introduce you to a straight forward image format, and allow you to modify an image in a few specific ways.

Due Date and Submitting:

This project is due on **Thursday, November 4, 2021 at 4:30pm**. Early submissions are allowed, with bonus points added according to the policy described in detail in the course syllabus.

For this project, you must submit several files. You must submit each header file (with extension .h) and each source file (with extension .cpp) that you create in implementing the project. In addition, you must submit a valid UNIX Makefile, that will allow your project to be built using the command "make" resulting in an executable file named "proj3.exe". Also, your Makefile must have a target named “clean” that removes all your .o files and your executable (but not your source code!).

When submitting your project, be sure that **every** file (both .h and .cpp files, and the Makefile and typescript file!!!) are attached to the submission email. The submission system will respond with the number of files accepted as part of your submission, and a list of all the files accepted – it is ***your responsibility*** to ensure all files were attached to the email and were accepted by the system. If you forget to submit a file on accident, we will not allow you to add the file after the deadline, so please take the time to carefully check the submission response email to be completely sure every single file you intended to submit was accepted by the system.

Detailed Description:

In the previous project, you developed classes for representing a Color, a Color Image, and a Row/Column Location. This project will use those same concepts but will focus on the use of dynamic allocation of arrays and file input/output, as well as separating your implementation into multiple files. Of course, we’ve also talked about detecting and overcoming stream Input/Output issues, and you’ll be expected to manage that as well.

Background: .ppm Imagery

Since you will be reading and writing images, you need some background on how images work. For this project, we will use a relatively simple image format, called PPM imagery. These images, unlike most other formats, are stored in an ASCII text file, which you are already familiar with. More complicated image

formats (like .gif and .jpg) are stored in a binary file and use sophisticated compression algorithms to make the file size smaller. A .ppm image can contain the exact same image as a .gif or .jpg, but it would likely be significantly larger in file size. Since you already know how to read and write text files, the only additional information you need is the format of the .ppm file.

Most image types start with two special characters, which are referred to as that image type's "magic number" (not to be confused with the magic numbers we've talked about as being bad style in programming). A computer program can determine which type of image it is based on the value of these first two characters. For a .ppm image, the magic number is "P3", which simply allows an image viewing program to determine that this file is a PPM image and should be read using the PPM format.

Since a 100 pixel image may be an image of 25 rows and 4 columns, or 10 rows and 10 columns (or any other such combination) you need to know the specific size of the image. Therefore, after the magic number, the next two elements of the .ppm file are the width of the image, followed by the height of the image. Obviously, both of these values should be integers, since they both are in units of "number of pixels". (note: width comes first, and height comes second! People always get this mixed up, so take care with the order…)

The next value is also an integer, and is simply the maximum value in the color descriptions. For this project, you will use 255 as the maximum number. With a maximum of 10, you are only allowed 10 shades of gray, and 10^3 unique colors which would not allow you to generate a very photographic looking image, but if your maximum value is 255, you could get a much wider range of colors (255^3).

The only thing left is a description of each and every pixel in the image. The pixel in the upper left corner of the image comes first. The rest of the first row follows, and then the first pixel of the second row comes after that. This pattern continues until every pixel has been described (in other words, there should be rows*cols color values in the .ppm file). As mentioned above, each pixel is described with three integers (red, green, blue), so a 4 row by 4 column color image requires 4*4*3=48 integers to describe the pixels.

You can't use static allocation here, you dynamic alloction

A very very small image of a red square on a blue background would be stored in a .ppm file as follows:

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

Maximum value

Once you create these images, you can view them many ways. There are many freely available programs that will display PPM images directly (I often use one called "IrfanView" on Windows (should be able to download this free from download.com) and either "xv" or ImageMagick's "display" on Linux).

Another alternative is to convert the image to a JPEG image, which will allow you to display the image via a web browser. One way to convert a PPM to JPG is to use the Linux command "cjpeg" like this:

```
% cjpeg inFile.ppm > outFile.jpg
```

Or using Linux's ImageMagick to convert like this:

```
% convert inFile.ppm outFile.jpg
```

Note: The "%" character shown in the commands is just meant to be the Linux prompt – it is not part of the command you would type in.

Required Functionality

For this project, you are only required to implement a few algorithms to modify an image. However, after completing the project, you will be able to add any number of your own algorithms to modify imagery in any number of ways.

Following are descriptions of the algorithms you are required to implement. First, you will need to allow rectangles to be drawn on an image. Rectangle outlines may be placed on an image to draw attention to a specific area, or filled rectangles may be placed on an image to block out a specific area. Both of these operations will be supported in this project.

Second, and more interestingly, an image may be annotated with a "pattern". A pattern, while rectangular overall, contains a description of a shape that is to be placed on an image. A pattern consists of a rectangle of only zeros and ones. When a pattern is placed over an image, the values in the pattern each fall over a specific pixel in the original image. A value of one in a pattern indicates that the pixel under it should be modified to be a certain color that is specified by the user. A zero in a pattern indicates that the pixel under it should NOT be affected by the pattern. Its original value is left intact, resulting in a sort of transparency.

Patterns are contained in text files of the following format: The first value is an integer representing the number of columns in the rectangular pattern. The second value is an integer representing the number of rows in the rectangular pattern. What follows is a collection of zeros and ones that is (rows * columns) in length. For example, here is the contents of a pattern file that defines a pattern of the letter 'T':

```
6 8
1 1 1 1 1 1
1 1 1 1 1 1
0 0 1 1 0 0
0 0 1 1 0 0
0 0 1 1 0 0
0 0 1 1 0 0
0 0 1 1 0 0
0 0 1 1 0 0
```

This is pattern file

The placement of such patterns on an image will be supported in this project. This capability allows you to annotate an image with any shape you wish, regardless of what it looks like.

The final image modification algorithm you will implement is the insertion of another (presumably smaller) PPM image at a specified location within the image being modified. This insertion simply reads another PPM image from a file and inserts the image contents where the user desires. PPM images are, by definition, rectangular. Since oftentimes, the image you want to insert is not rectangular, you must support a transparency color, such that any pixel in the image to be inserted which is the transparency color does not change the original image, but pixels that are not the transparency color will be used to replace the pixel value in the original image. Note that this is very similar to the use of a pattern, described above, except that patterns can only be one color, while inserted images can have as many colors as the PPM allows.

At any stage, you must allow the user to output a PPM image file in its current state from the main menu. The user may want to output an image after each change made, or just once when all updates have been performed. Since the option of outputting an image is available on the main menu, this functionality will be supported in this project.

There are examples of all required functionality available in the sample output of the project.

Implementation and Design

All of your global constants must be declared and initialized in a file named "constants.h". This file will not have a corresponding .cpp file, since it will not contain any functions or class definitions. Make sure you put all your global constants in this file, and avoid magic numbers. Since you now know about dynamic allocation, the image pixels will be allocated using the new operator, using exactly the amount of space required for the image (for example, a smaller image will use less memory than a larger image). Therefore, there is no practical limit to the size of the image allowed.

I'll leave the majority of the design up to you, and remember I will be looking at your design during grading. If you find you want some global functions, you may use them. Each individual class will be contained in a .h and a .cpp file (named with the class name before the dot). ALL class member variables MUST be private. Your member functions may be public. Each global function will be contained in a .h and a .cpp file named the same as the function. Do not put multiple global functions in a single file (unless they are overloaded using the same name, and therefore belong in the same file). Remember, when submitting, you must submit ALL .h files, .cpp files, and your Makefile. Do not include your .o files or your executable in your submission.

While you might want to make use of your framework from the previous project, there are some important changes to note: 1) The maximum color value will now be 255 (instead of 1000). All clipping and “max color values” should use 255 instead of 1000. If you didn’t use magic numbers, this should be rather straightforward. 2) The ColorImageClass developed in the previous project had a matrix of pixels that was statically allocated with a specified size – for this project, the size will not be known at compile time, and

you must use dynamic allocation to allocate exactly the number of pixels needed – no more and no less. 3) You'll have to add functionality as required for this project. Thinking about the functionality to write and read images to/from files - when developing this functionality, remember that a ColorImageClass object should write/read image-related attributes to/from files. It is really each individual pixel's responsibility to write/read its own color to/from the file. In other words, the ColorImageClass write/read methods should not write/read color RGB values (the ColorImageClass shouldn't even know the details of what a ColorClass has as attributes, etc). Instead, the ColorImageClass should call a member function of the ColorClass to write those values. Always think about this type of thing when designing your project.

You'll see that when choosing to annotate an image with a rectangle, there are three different methods you must support – specifying the rectangle via: 1) the upper-left and lower-right locations directly; 2) specifying the upper-left corner and a width and height; and 3) specifying the center of a rectangle and a width extent and height extent from the center (i.e. half-width and half-height). At first glance, this seems like tedious "make work", but the reason for requiring three methods to do the same thing is to make you think about your design of your Rectangle class. One thing to remember is that, regardless of which method I use to specify the rectangle, the resulting rectangle can be described using a pre-defined set of attributes. For example, even if the user uses method 3 to specify a rectangle, internally in your program, it can be stored, described, and used via an upper-left corner and a lower-right corner. In other words, there is no need to have attributes in your rectangle class to support each input method – simply convert the values the user input to the attributes you will store all rectangles as. This is another type of thing we will be looking at in your design, so its worth understanding and doing correctly.

While you have more flexibility in your project design, your menu and the way your project operates must match that shown in the sample output. Do not change the actions associated with menu options, orderings, expected user inputs, etc. I must be able to input the exact same values I would to my solution, in the exact same order, and have the program act accordingly. Do not add additional prompts that the user has to respond to, re-order menu options, change the number of items requested for input, etc.

A Quick Detail:

The "open" member function of the file stream classes (ifstream, ofstream) take the name of a file in as a parameter of type "c-string", NOT C++ string. You'll have filenames stored as C++ strings, though, so you'll need to convert it to a C-string so the compiler will be happy. Do this using a member function of the string class called "c_str()". For example, your code would look something like this:

```
ifstream inFile;
string fname;
//get the name of the file stored in fname somehow
inFile.open(fname.c_str());
```

Error Handling

Since we've talked about error handling for stream input/output, you'll need to ensure you handle potential issues when dealing with input. There are several things that can go wrong during the input/output, and you should consider all of those cases. In addition to being an object-oriented program using dynamic allocation and file I/O, this project will focus on error checking, and many of our test cases

during grading will be “nitpicky” to check that you detected and handled errors that might come up appropriately.

Here's how to handle errors that come up. For the initial prompt for the main image, if the image can't be loaded, print an error message and allow the program to end. If other files can't be read or written during the program (pattern files, other images, etc.), output a ***descriptive*** error message and continue the program. The program should ***not*** exit in this case – the reasoning is that the user may have spent hours annotating an image, etc., and if they make a simple typo when trying to type the name of a pattern file (for example) you don’t want the user to have to start over. In any case, make sure you print a descriptive error message. Saying "Error found when trying to read magic number - expected P3 but found P5" is far better than just saying "Error reading image" which doesn't describe the error that occurred at all, and doesn’t provide the user any insight as to how they can fix the problem. Make sure you consider all the different things that could go wrong when reading a PPM file, which may or may not be in a proper format (there’s quite a few things to consider that could go wrong when reading an image file).

"Specific Specifications"

These "specific specifications" are meant to state whether or not something is allowed. A "no" means you definitely may NOT use that item. We have not necessarily covered all the topics listed, so if you don’t know what each of these is, it’s not likely you would “accidentally” use them in your solution. Those types of restrictions are put in place mainly for students who know some of the more advanced topics and might try to use them when they’re not expected or allowed. In general, you can assume that you should not be using anything that has not yet been covered in lecture (as of the first posting of the project).

- Use of Goto: No
- Global Variables / Objects: No
- Global Functions: Yes (as necessary)
- Use of Friend Functions / Classes: No
- Use of Structs: No
- Use of Classes: Yes – required!
- Public Data In Classes: **No** (all data members must be private)
- Use of Inheritance / Polymorphism: No
- Use of Arrays: Yes
- Use of C++ "string" Type: Yes
- Use of C-Strings: No (except as noted to satisfy the “open” method)
- Use of Pointers: Yes – required! All matrices for images/patterns must use dynamic allocation
- Use of STL Containers: **No**
- Use of Makefile / User-Defined Header Files / Multiple Source Code Files: Yes – required!
- Use of exit(): No
- Use of overloaded operators: No
- Use of float type: **No** (That is, all floating point values should be type double, not float)