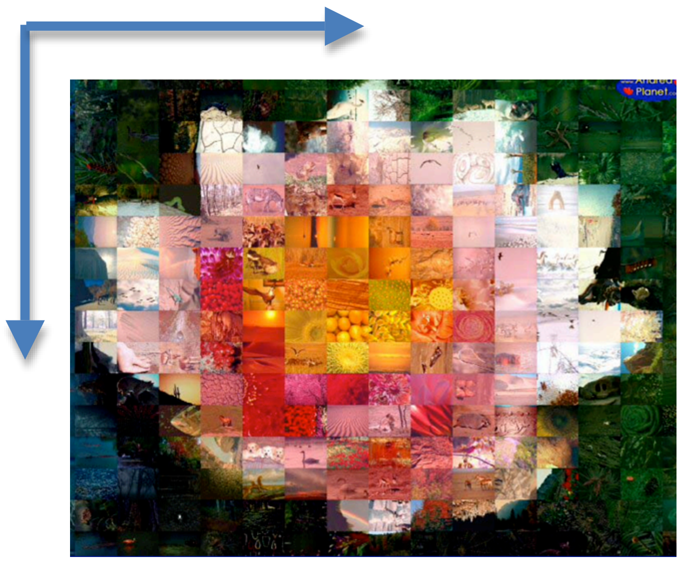
Programming Assignment 4
Image Shrinker
Due 03 April by Midnight

The final two programming assignments (Program 4 and Program 5) will be two phases of a single project to implement a photo mosaic generator/creator. A photo-mosaic is a collection of small images that, when viewed at a distance, produce the illusion of looking like some other, larger image. Here is an example:



The goal of this two-phase project is to implement a C++ program that can create these interesting art pieces for any input image. In order to get started is important to be intentional about some basic design considerations and a few techniques for managing and manipulating digital images. In solving this problem we will be using classes, pointers, and structured data in order to organize an efficient solution.

The images you will manipulate are two-dimensional structures with a width and a height in units of pixels. Think of an image as a 2D array, indexed by rows (horizontal "*scan lines*"), which go across the image, and columns (vertical stripes through the image). Any pixel in an image is located by its (row, col) coordinates, or in C++ array terms, **`myImage[row][col]`**. The positive axes of the coordinate system are shown on the image above, with **`myImage[0][0]`** in the upper left corner. Stored at each position in this two dimensional structure are the details of an individual pixel, represented as a point in a 24-bit RGB color space (8 bits for red, 8 bits for green, and 8 bits for blue). This is a common way to store image data. In this case, each and every pixel in the image consists of three values that are each 8 bits in length. This 8-bit size is no accident: in C++, 8 bits can be stored in one `unsigned char` variable. These three values represent the red, green, and blue components of a single pixel. Since the value has 8 bits and is unsigned, it can hold any value between 0 and 255. A pixel with red, green, and blue all zero will appear black; a pixel with R, G, and B all equal to 255 will appear white. Or put another way, a low value is "dark" and a high value is "bright". Just like an array, the origin (pixel at 0, 0) is normally viewed by convention as the upper left pixel, with rows horizontal and columns vertical. So first, make sure you understand the structure of an image: a 2D array (with a width, height); scanlines as rows; a coordinate frame that has the origin in the upper left, with positive x along the horizontal from left to right, and positive y along the vertical from top to bottom; and at each location a composite pixel object which contains values for R,G,B – and each of those will require 8 bits. The structure/layout has a very natural visual interpretation.

The basic concept of creating a photo-mosaic starts with a ***base*** image. In the example above, the starting base image was an image of a white flower with a red center. The goal is to create a new "mosaic image" from the base image. This is done by taking small sections of the base image (suppose we divide the base image into 64x64 blocks of pixels), and replacing those sections with "filler images" (small, completely different images), each of which "looks like" the pixels in the section it replaces. One straightforward way to find a good filler image for a specific block in the base image is to take the *average* of the base image pixels in the block and compare it to the average value of pixels in a big database of available (small) filler images. Here is the rule:

> *The filler image that has an average value close to the average value of the block is the one selected to replace the block.*

So the base image is systematically replaced, block-by-block, with filler images that are similar to the blocks they replace. The result is a photo-mosaic.

In this first half of the assignment, we will be laying the foundation for this work. The process discussed above for finding the average color in a larger block of pixels can be thought of as a very simple "re-sampling" of the bigger image. This technique is often used to generate smaller versions of big images (e.g., a thumbnail of the original image). In this assignment you will write a program that can:

1. Allow simple loading and saving of a base image.
2. Divide a base image into blocks of a given width and height and find the average color in that block.
3. Create a new image from the process in (2) that will consist of 1 pixel per every block analyzed in the first image.
4. Apply a red, green, or blue tint to an image by increasing or decreasing the appropriate RGB values to every pixel in the image.
5. Invert the image by subtracting every pixel's current value from its max value.

Provided in the code base for this assignment, you will see the following files in addition to the usual:

`globals.h`
Note in this file that the type "unsigned char" has a typedef to the easier to write, "uchar". These types are interchangeable throughout the program.

`pixel.h`
This file contains the `pixel` class that is used to store the red, green, and blue values for a pixel of data. The `pixel` class consists of three public `uchar` variables to store this data. The implementation of this class, with its data members *public*, is an example of a POD (Plain Old Data) Class. In almost all cases, data members should be kept private, with access granted only through member functions. However, in some cases (such as this), no functions are required whatsoever (no constructors or any member functions) and the purpose is simply to aggregate several pieces of data into one object. In these cases, it is acceptable to have the data members publicly accessible,which allows for a smaller memory footprint and easier notation when referencing the data.

`image.h / image.cpp`
This class is provided to do the low-level work of loading and saving images. It also provides a way for the graphical user interface (GUI) to display the images represented in our RGB pixel data. The functions you should be concerned with are these:

`bool loadImage(string filename)`
Loads an image from the specified file. Returns `true` if it loaded successfully, `false` if it did not.

`void saveImage(string filename)`
Attempts to save the image, stored as pixel data, to disk at the specified file location.

`void createNewImage(int width, int height)`
Clears current data in the image object and creates a new blank image of the specified width and height. This function, or the `loadImage()` function, must be called before the pixels of an image object can be accessed.

`int getWidth()`
Returns the width of an image in units of pixels.

`int getHeight()`
Returns the height of an image in units of pixels.

`pixel** getPixels()`
Returns a pointer to a 2D array of `pixel` objects which represent an image. This multi-dimensional array is allocated dynamically at run time, and this function returns a pointer to the array once it has been allocated. Note the return type of `pixel**`, which may seem strange at first glance. Remember that a 1D array name in C++ is equivalent to a pointer to the first value of the array block in memory. So for example, for a 1D array, `pixel[]` is equivalent to `pixel*`. Extending this idea, a 2D dimensional array name can be thought of as a *pointer to a pointer*. Appropriate use of this function will therefore look something like this:

```
    pixel** myPixels = myImage.getPixels();
```

The elements of the 2D array pointed to by the variable myPixels can then be referenced, for example, as

```
    myPixels[0][0].red = 255;
```

This would set the red value of the first pixel in the image to its maximum value.

Feel free to look at the code in `image.cpp` to see how this is done.

Use these objects as you see fit to implement this program.

Supplied for you is a GUI interface written in C++/Qt. You should only need to modify the file `CS215Pgm4.cpp` as well as any `.h` and `.cpp` files that *you add* to the project.

You must implement the following functions in `CS215Pgm4.cpp`:

`bool loadImageFromFile(string filename)`

INPUTS: a string denoting the input file pathname.
OUTPUTS: a `bool` indicating whether or not the image file was opened successfully.

This function is called when the user presses the "open image" button on the interface. The dialog box (the "open file dialog") gets the string from the user and passes that as the parameter to this function, which then must do the work of actually opening the image file and loading it.

`void saveImageToFile(string filename)`

INPUTS: a string denoting the output file pathname.
OUTPUTS: NONE

This function is called when the user presses the "save image" button on the interface. The dialog box (the "save file dialog") gets the string from the user and passes that as the parameter to this function, which then must do the work of actually saving the image.

`image* displayImage()`

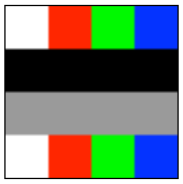
INPUTS: NONE
OUTPUTS: This function should return a pointer to the image object that is to be viewed on the interface. Whenever the interface determines that it needs to refresh the display, it will call this function for a pointer to the image object that is to be displayed in the image area of the GUI.

For example, if other functions make changes to the image object (if a user has loaded an image correctly), the interface will call this function when it refreshes the image and it will expect a pointer to an image object to be returned so that it can display that specific image on the interface. This means that other functions you write, such as the function associated with the "shrink" button (aka `averageRegions` function) or the red/green/blue filters, or the invert function, may need to modify this pointer or the data it points to so that the updates will be displayed on the GUI.

In general, any time you as the programmer want a specific image object to be displayed in the image area of the GUI, this function is the one that the GUI will use to get a pointer to the image to be displayed.

`void averageRegions(int blockWidth, int blockHeight)`

INPUTS: Integers indicating the width and height of the "blocks" to be averaged
OUTPUTS: NONE
This function should create a new `image` object that will consist of one pixel for every block of size `blockWidth` by `blockHeight` pixels in the original image. Then each of these pixels should be set to the *average color* of the pixels in that region in the original image. For example, consider this image:



The image is 64x64 and assume this function is called with `blockWidth` = 16 and `blockHeight` = 16. The new image resulting from the function should be a 4 x 4 pixel image consisting of:

white, red, green, blue
black, black, black, black
gray, gray, gray, gray
white, red, green, blue

Of course it is straightforward to find the average color of the blocks shown because they all solid colors – this is just a simple example. If the image does not divide evenly into the number of blocks specified, just ignore the "remainder pixels" on the right and bottom of the image. For example, if the above example was 70 x 70, this would analyze the first 64 bits (4 blocks) of the pixel data and ignore the last 6 on the top and bottom).

Note that this is a perfect place to consider writing a helper function that could be called from within this one. The second function could calculate the average value of a block of pixels given to it, and return that to the original function to be used.

`void changeRedValues(int value)`

INPUTS: A positive or negative integer representing the desired change in the red component of each pixel.
OUTPUTS: NONE
This function should change the red component of each pixel in the current image by the amount specified, i.e., `newRed = red + value`. RGB values (8-bit uchar) have values between 0 and 255. Note that `value` can be negative in order to decrease the red value.

`void changeGreenValues(int value)`

INPUTS: A positive or negative integer representing the desired change in the green component of each pixel.
OUTPUTS: NONE
This function should change the green component of each pixel in the current image by the amount specified, i.e., `newGreen = green + value`. RGB values (8-bit uchar) have values between 0 and 255. Note that `value` can be negative in order to decrease the green value.

`void changeBlueValues(int value)`

INPUTS: A positive or negative integer representing the desired change in the blue component of each pixel.
OUTPUTS: NONE
This function should change the green component of each pixel in the current image by the amount specified, i.e., `newBlue = blue + value`. RGB values (8-bit uchar) have values between 0 and 255. Note that `value` can be negative in order to decrease the blue value.

`void invertValues()`

INPUTS: NONE
OUTPUTS: NONE
This function "inverts" the individual components of each pixel in the current image. To invert a pixel value, set each component in the pixel to its maximum value (255) minus its current value, i.e.,

```
newRed = maxRed – red;
newGreen = maxGreen – green;
newBlue = maxBlue – blue;
```

This function should apply the inversion to every component of every pixel in the image.

Sample images have been included for your convenience. Feel free to test your program with these or any other images. Please note however, that these images can be in jpg, tif, png, bmp, etc format, but MUST be saved as a 24 bpp (aka 8 bits per RGB value) image. We are limiting our work to this color space. There are many other color spaces that images can be saved in (Grayscale, Palette, CMYK, etc) but these are beyond the scope of this assignment. If an image cannot be loaded for this reason or one similar, a message will be printed out to the console and false will be returned from the load function.

Submission
You will need to submit the source files that you have created along with your modifications to CS215Pgm3.cpp. These files should all be put in a .zip file and upload to the CS Portal.

Example screens shots from the program:

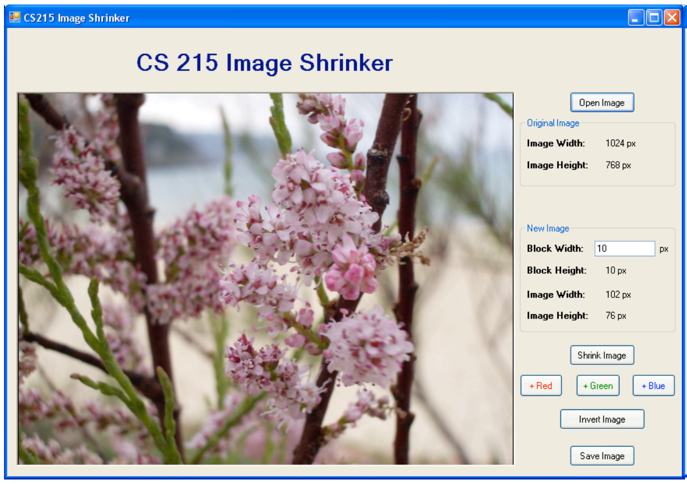


Image Width: 1024 px
Image Height: 768 px

Block Width: 10 px
Block Height: 10 px
Image Width: 102 px
Image Height: 76 px

After loading an image

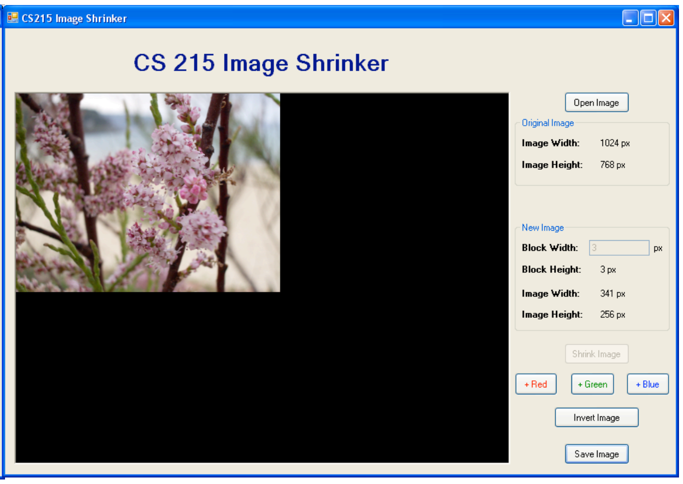


Image Width: 1024 px
Image Height: 768 px

Block Width: 3 px
Block Height: 3 px
Image Width: 341 px
Image Height: 256 px

After shrinking (averaging the pixels) of the image by 3x3 blocks

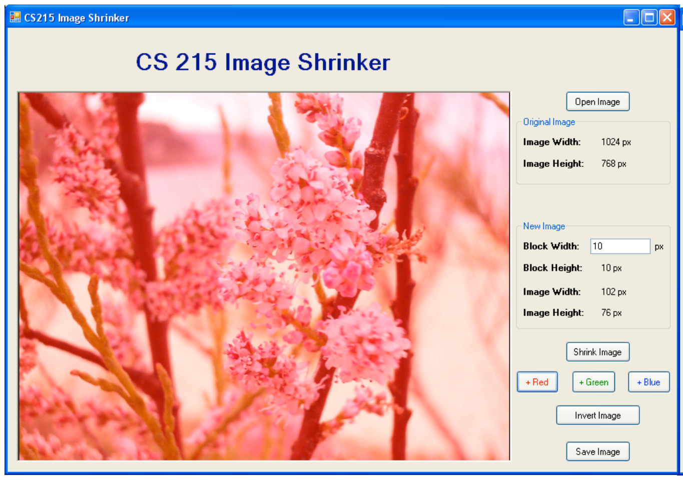


Image Width: 1024 px
Image Height: 768 px

Block Width: 10 px
Block Height: 10 px
Image Width: 102 px
Image Height: 76 px

After reloading and increasing the red component of the image several times

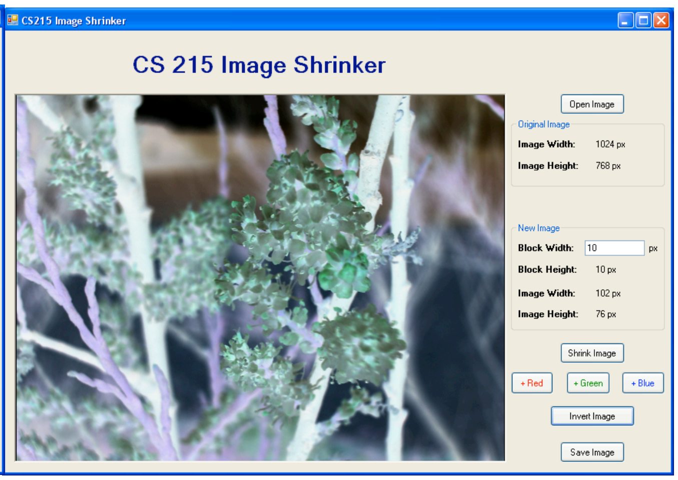


Image Width: 1024 px
Image Height: 768 px

Block Width: 10 px
Block Height: 10 px
Image Width: 102 px
Image Height: 76 px

After reloading and inverting the image.