ECE 252: Programming Methodology I
Department of Electrical and Computer Engineering
Prof. Jorge Ortiz
Rutgers University

RUTGERS
School of Engineering
ELECTRICAL AND COMPUTER ENGINEERING

ECE 252: Programming Methodology (Spring 2023) – Project 1

Due: March 18, 2024 (11:59:59 PM)

Rationale and learning expectations: This project reinforces all the constructs, data types, and operations supported in C that are covered throughout the first half of the course. This includes data types, pointers, loops, operators, and compilation. The goal is to combine these into a comprehensive program that transforms images passed as input to your program.

In this project, your task is to create an image-processing library that uses matrix operations to manipulate images. Images are represented as matrices of single-byte values, where each pixel at matrix location `(i, j)` is assigned a value between `0` and `255`. For color images, there are three channels: `red (R), green (G), and blue (B)`. The information for each image matrix consists of three matrices, each with the same height and width dimensions as the original image and representing the three color channels. These matrices are passed to an external library for file output and display.

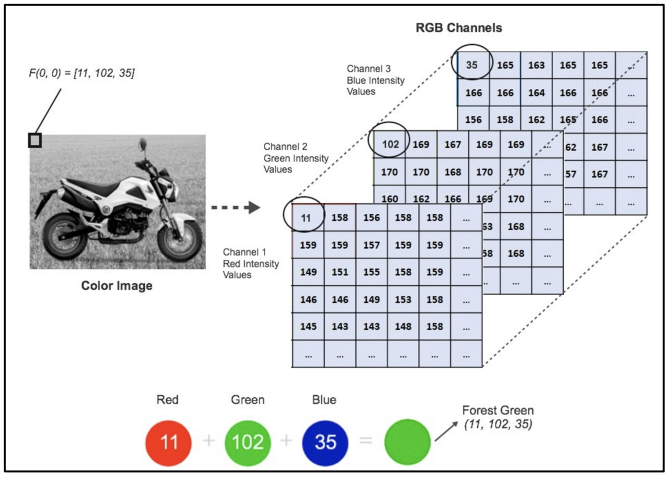

Figure 1: This illustrates how a computer 'sees' an image. A color image is a set of three matrices that represent the pixel intensity in (R)ed, (G)reen, (B)lue. In this project, we will manipulate the image by performing arithmetic operations on individual pixel values and observing the outs.

To complete the project, you must write various functions that bridge the general image library and the `RGB` matrix view. These functions will use the matrix view to create new images, including image addition, subtraction, scalar and multiplicative scaling. By updating the individual pixel values in the image's matrix, you can manipulate the image in various ways to achieve the desired result. We will provide you with the starting code and required files for the project, and the parts of the code that require completion will be labeled with `//YOUR CODE HERE` to signify where your input is required.

Environment Setup

Code link (mac/unix): https://rutgers.box.com/s/wt67fcf3kzo079qbrul3eb930fgv1gbc

Code link (Windows): https://rutgers.box.com/s/u3c1dyq7bnxbidkwh0fwjt7wf44fara5

1. Set up visual studio code for C/C++: Visual Studio Code C/C++ setup.
2. Download the code and set up your environment.
3. Rename the file **`FIRSTNAME_LASTNAME_NETID`** with your first name, last name, and netid. Modify the **`src/imageutil.c`** with your solutions.
4. Test by running make. To learn more about make, read these:
   - Makefile tutorial.

Part I (40 points)

The code consists of several header files and the main file you will be updating. The four files are **`imageutil.c, imageutil.h, stb_image.h, stb_image_write.h, stb_image_resize.h, main.c`** The code you will write will all be in imageutil.c. Notice at the top of the file, we import all the necessary files to run the code. These header files contain the function definitions to open, write, and manipulate a PNG file. **We will work ONLY with `PNG` files for this project.**

The main object that you will use is shown below (`imageutil.h`):

```

// imatrix struct, wraps a byte array of pixel data for an image
typedef struct imatrix{
    int width;          // Width of the image in pixels
    int height;         // Height of the image in pixels
    uint8_t** r;        // Red channel pixel data
    uint8_t** g;        // Green channel pixel data
    uint8_t** b;        // Blue channel pixel data

    // Function pointers for matrix operations
    // Pointer to add function
    struct imatrix* (*add)(struct imatrix* this, struct imatrix* m2);

    // Pointer to subtract function
    struct imatrix* (*subtract)(struct imatrix* this, struct imatrix* m2);

    // Pointer to multiply function
    struct imatrix* (*multiply)(struct imatrix* this, struct imatrix* m2);

    // Pointer to scale function
    struct imatrix* (*scale)(struct imatrix* this, int width, int height, float alpha);

    // Function pointers for image input/output

    // Pointer to function that writes image data to RGB format
    void (*write_image_to_rgb)(struct imatrix* this);

    // Pointer to function that writes RGB data to image format
    void (*write_rgb_to_image)(struct imatrix* m);

    // Pointer to function that sets the RGB image
```

```
32      struct imatrix* (*set_rgb_image)(struct imatrix* this, uint8_t*
33                                       new_rgb_image, int width,
34                                       int height, int channels);
35
36      // Internal reference to image data in RGB format
37      uint8_t* rgb_image; // Pointer to RGB image data
38  } imatrix;
```

This structure consists of several fields. Three pointers, `r, g, b`, will be used to represent an image as three separate matrices, one for each channel. The structure also consists of a pointer `rgb_image`, a pointer to the image data in a contiguous memory block.

When you read an image as a command-line argument, the code loads the `PNG` images using the library. **Your job in this first task will be to implement `init_rgb()`**. You should use `malloc()` for dynamic memory allocation to initialize the `r,g,b` two-dimensional arrays.

Part II (20 points)

In part 2, you will write the code to scale the pixel values of an image. The associated function for this part of the project is **`imatrix* scale(imatrix* this, int width, int height, float alpha)`**. Please read the documentation for this function and complete the code. You should test your code by compiling it with the **`Makefile`** and passing in the necessary parameters on an image of your choice.

Part III (20 points)

Here complete the two add and subtraction functions, **`imatrix* subtract(imatrix* m1, imatrix* m2,`** **`imatrix* add(imatrix* m1, imatrix* m2`**. These two functions take two **`imatrix`** and add/subtract them from each other. The result is written in a file. To blend images together, try scaling them before adding or subtracting them.

Part IV (20 points)

Finally, scaling and rotating an image can be done through matrix multiplication. Complete the function for matrix multiply **`imatrix* dot(imatrix* m1, imatrix* m2)`**, following the specification in the comments section above the function.

Note, the dot product is used for many operations in image processing, such as translation and rotation, convolutions, and others. However, in this assignment, the operation is meaningless. We simply want to see the functionality implemented correctly. If you want to take rotation and translation, you can make **`m2`** a `KxK` matrix (if **`m1`** is `NxK`), in order to perform the translation/rotation.

Submission

Please zip your code and submit it via canvas. Do not include any images or testing you did. You must include the file with your name and netid (as specified in Section and your modified **`imageutil.c`**.

Assignment Rubric

1. **Compilation and Execution:** (30 points)
   This criterion evaluates the code's ability to compile and execute correctly. Points are assigned based on the code's compilation success and its ability to run without any runtime errors or crashes.
   Point Distribution:
   - Code compiles successfully: 10 points
   - Code executes without runtime errors or crashes: 20 points

2. **Unit Test Results:** (40 points)
   This element assesses the results of the unit tests for each function. Points are assigned based on the number of tests passed and the accuracy of the output.
   Point Distribution:
   - All unit tests pass: 40 points
   - Some unit tests pass: 20-30 points
   - A mix of passing and failing unit tests: 10-20 points
   - Most unit tests pass: 30-35 points

3. **Partial Credit for Correctness:** (20 points)
   This criterion awards points for achieving partial correctness in the functions. Points are assigned based on the level of correctness and the number of cases in which the function produces partial correct output.
   Point Distribution:
   - No partial credit awarded: 0 points
   - Minimal partial credit for partially correct output in some cases: 5-10 points
   - Moderate partial credit for partial correctness in multiple cases: 10-15 points
   - Significant partial credit for substantial partial correctness in most cases: 15-18 points
   - Full credit for achieving partial correctness in all relevant cases: 20 points

4. **Handling of Edge Cases and Corner Cases:** (10 points)
   This element evaluates the code's ability to handle edge cases and corner cases effectively. Points are assigned based on the code's accuracy in handling these special scenarios.
   Point Distribution:
   - Does not handle any edge or corner cases correctly: 0 points
   - Handles a few basic edge cases but fails to handle most corner cases: 2-4 points
   - Handles some common edge and corner cases, but may miss a few: 4-7 points
   - Handles most edge and corner cases effectively but may have minor issues: 7-9 points
   - Handles a wide range of edge and corner cases accurately and comprehensively: 10 points