Assignment 5 – Color Blindness Simulator

Amish Minocha

CSE 13S – Spring 2023

Purpose

Deuteranopia is the most severe form of red-green color blindness, which is the world's most common form of color-blindness affecting about 4% of the population. What it means is that someone with this form of color blindness would have severe difficulty in differentiating between red and green colors due to the complete lack of green sensitive light sensors in the eyes of those affected by this condition. The purpose of this assignment is to convert a picture initially displayed using a conventional array of colors into a picture that resembles one that a person with deuteranopia would view as. Instead of using the stdio functions like fopen, fprintf, or scanf, we will be utilizing the Unix file I/O system since the pictures in question will be manipulated in the form of binary files. Using the lower-level Unix functions will reduce the overhead and hence make for a more efficient program. Apart from this, we will also employ bitwise operations to read, manipulate, and write binary numbers that represent the 'rgb' value for each pixel in the picture file.

How to Use the Program

To start the program type the following commands to compile and then run the file:

```
make
./colorb
```

The file 'colorb.c' is the main file that comprises the main() function. This function contains all the calls to the other functions and allows us to test and run the program. Once you run the command provided above, you will receive an error prompting you to run the command again and include 'option arguments' at the end. The option argument is a singular character with a hyphen symbol ('-') before it, like '-i' or '-o'. So, retype the command but specify the options -i and -o with their respective parameters for the input and output files. You may also type in -h to display the help message. Correct usage looks as follows:

```
./colorb -i img1.bmp -o img1cb.bmp
```

Program Design

We will first create an io.c file which consists of all the read and write functions declared in the corresponding header file. These functions will perform specific read/write functions using the Unix file I/O system along with bitwise operations depending on the size integer we are dealing with (like 8, 16, or 32 bits). For the read-open and read-close functions, we will be allocating and freeing a Buffer (object defined in io.c) to read from a particular file. Similarly, for the write-open and write-close functions, we will be doing the same but to write to a particular file. We will then write the remaining functions to read and write 8, 16, and 32 bit unsigned integers. We will utilize little-endian ordering to read/write the bytes which means we write the least-significant bytes first. After this, we will create a file called bmp.c in which we will define all the functions declared in bmp.h. In this file, we will define two data structures: color and bmp, and proceed to write the functions to help us manipulate the rgb values, read, write, and more. Finally, we will create our last file called colorb.c and in here we will construct our main method. This will consist of the getopt() function to read the option arguments and have our file functions to read the file path of the image file and

the output file. We will also have a -h option argument to print the help message. We use the '?' case and exit the program in the situation the user enters an option that we haven't specified. We also exit the program if the user fails to enter either an input file or an outputfile the program exits. Finally, we simply call the functions we created in the other .c files in a specific order with the opens and creates followed by the closes and the frees.

Data Structures

- Buffer: This is a structure we create that consists of an integer fd which is the file descriptor from open() and creat(). It also has another integer variable called offset which we'll use to control the offset of the structure. Furthermore, we have the num_remaining integer and finally, the unsigned integer array of 8 bits that has a length of BUFFER_SIZE.

- Color: This is a simple data structure we create which consists of three unsigned integers of 8 bits red, green, blue. We use this for the (r,g,b) values to depict the specific color.

-BMP: This is another data structure we create which consists of two 32-bit unsigned integers 'height' and 'width' for the image. It also has a 'Color' array called palette with a size of 'MAX-COLORS' and a 3 dimensional array of unsigned ints (8 bits) called a. This structure will represent our bitmap.

- getopt(int argc, char *argv[], char *string): function in colorb.c that will take three arguments. argc returns the number of arguments passed (including the function name). argv is a character string array that stores each of the arguments. the string will store all the valid character arguments the user can enter, example: "i:o:h" will allow the user to enter characters 'i', 'o', or 'h', with additional parameters for i and o since they are followed by ':'.

- switch(opt); Switch case data structure that sets a character's corresponding boolean value to true (1). This happens for all the characters the user enters. At the end of the while loop, all the boolean values are appropriately set to either 1 or 0 and move on to the if/else statement.

- if/else statement(s): After the getopt() while loop, we have an if statement that checks to see if either -h is an option. This fulfills the conditions required to print out the help message. We have an else block right after this which consists of all the if statements (written in the specified order) to ensure the name of out input and outfile file. Having them inside of an else block ensures that when we print out the 'help statements' nothing else is printed.

-calloc(): We use this function to allocate memory dynamically. We will use this for our buffer data structure to ensure that we get an empty usable buffer each time with a pointer pointing to the entire data structure. We use the free() function to free the pointer pointing to the buffer and then set the pointer equal to NULL, as it is good practice, to be used again.

Algorithms

We use the bmp-reduce-pallet function that was given to us to manipulate the pixels of the image to reflect the altered image. This was given to us through the assignment document.

Function Descriptions

Buffer *read_open(const char *filename)

- Inputs: Name of the filename (character string)
- Outputs: Returns NULL if unsuccessful else returns the pointer to the new buffer it creates.
- Purpose: To open a file and prepare it to read data from it by creating and initializing a buffer that is dynamically allocated using calloc()
- Explanation: We use the default Unix file I/O functions to perform the operations in the function. We use O_RDONLY as our second permission parameter to indicate that we will only be reading from the file. We use calloc() to allocate the memory for our buffer and set all its elements to zero. We don't need to allocate memory for our array since its size is constant and known during compile time, so its

memory will automatically be allocated during the calloc() call for the entire structure. Finally, we return the pointer to the actual buffer.

Buffer *write_open(const char *filename)

- Inputs: Name of the filename (character string)
- Outputs: Returns NULL if unsuccessful else returns the pointer to the new buffer it creates.
- Purpose: To open a file and prepare to write data to it by creating and initializing a buffer that is dynamically allocated using calloc()
- Explanation: We use the default Unix file I/O functions to perform the operations in the function. We use the creat() function to create a file filename and use the code 0664 which is an octal number since it starts with '0'. Furthermore, we set all the other elements to 0 and return the pointer to the buffer.

void *read_close(Buffer **pbuf)

- Inputs: A buffer.
- Outputs: None.
- Purpose: To close the file and free the buffer so it can be used again.
- Explanation: We close the file and then call the free funtion on the buffer and set it to NULL. We also do this for the datatypes we allocate the buffer to.

void *write_close(Buffer **pbuf)

- Inputs: A buffer.
- Outputs: None.
- Purpose: To close the file and free the buffer so it can be used again.
- Explanation: We close the file and then call the free funtion on the buffer and set it to NULL. We also do this for the datatypes we allocate the buffer to. We use the peudocode given to us for the read-int8 function to check to see if the buffer is full and flush out the elements. Finally, we free the buffer and set the pointer to NULL.

bool *read_uint8(Buffer *buf, uint8_t *x)

- Inputs: A buffer and a pointer to an unsigned integer of 8 bits.
- Outputs: Boolean value (true or false)
- Purpose: To read an 8 bit value (1 byte) into the buffer. Stores the next byte in the memory location *x.
- Explanation: The pseudocode for this function is given to us by the assignment document.

bool *read_uint16(Buffer *buf, uint8_t *x)

- Inputs: A buffer and a pointer to an unsigned integer of 16 bits.
- Outputs: Boolean value (true or false)
- Purpose: To read a 16 bit value (2 bytes) into the buffer. Stores the next two bytes in the memory location *x.
- Explanation: For this, we simply call read-int8 twice and check to see if it returns false for either of the time.

bool *read_uint32(Buffer *buf, uint8_t *x)

- Inputs: A buffer and a pointer to an unsigned integer of 32 bits.
- Outputs: Boolean value (true or false)
- Purpose: To read a 32 bit value (4 bytes) into the buffer. Stores the next four bytes in the memory location *x.
- Explanation: For this, we simply call read-int16 twice and check to see if it returns false for either of the time.

bool *write_uint8(Buffer *buf, uint8_t x)

- Inputs: A buffer and a pointer to an unsigned integer of 8 bits.
- Outputs: Boolean value (true or false)
- Purpose: To write an 8 bit value (1 byte) to the buffer. Sets the next free byte in the buffer to x and increments offset.
- Explanation: The pseudocode for this function is given to us by the assignment document.

bool *write_uint16(Buffer *buf, uint8_t x)

- Inputs: A buffer and a pointer to an unsigned integer of 16 bits.
- Outputs: Boolean value (true or false)
- Purpose: To write a 16 bit value (2 byte) to the buffer. Sets the next two free bytes in the buffer to x and increments offset.
- Explanation: For this, we simply call write-int8 twice, first with x and then with x shifted right by 8 bits.

bool *write_uint32(Buffer *buf, uint8_t x)

- Inputs: A buffer and a pointer to an unsigned integer of 32 bits.
- Outputs: Boolean value (true or false)
- Purpose: To write a 32 bit value (4 byte) to the buffer. Sets the next four free bytes in the buffer to x and increments offset.
- Explanation: For this, we simply call write-int16 twice, first with x and then with x shifted right by 16 bits.

void bmp_write(const BMP *bmp, Buffer *buf)

- Inputs: A bitmap structure that is passed in as a constant so it can't be moddified along with a Buffer structure.
- Outputs: None.
- Purpose: To write a bitmap file using bmp and buf structures.
- Explanation: The pseudocode for this function is entirely given in the assignment document. For the tabular data, we use the 'write_uintxx' function to write the data depending on its size in bits. For the for loops, we use ¡= instead of ¡ along with the exact parameters provided in the pseudocode.

void bmp_create(Buffer *buf)

- Inputs: A buffer structure buf.
- Outputs: None.

- Purpose: To create a new BMP (bitmap) structure with a file read into it.
- Explanation: The pseudocode for this function is entirely given in the assignment document. For the tabular data, we use the 'read_uintxx' function to write the data depending on its size in bits. We use calloc() to create the BMP structure and return NULL if the struct itself is NULL. If we encounter bytes we have to skip, we simply assign them to dedicated variables that we never use (namely skip-8, skip-16, skip-32). We also use assert() statements to verify that that the information read into the bmp struct is accurate. For the for loops, we use ¡= instead of ¡ along with the exact parameters provided in the pseudocode.

void bmp_free(BMP **bmp)

- Inputs: A pointer to a BMP structure bmp.
- Outputs: None.
- Purpose: To free a bmp structure and all the associated elements of it.
- Explanation: The code for this function is entirely given in the assignment document. We free the individual elements of the array first and then we free the pointer that points to the bmp structure itself. Finally, we set the pointer to NULL so our program crashes if we were to reference it again.

void bmp_reduce_palette(BMP *bmp)

- Inputs: A BMP structure bmp.
- Outputs: None.
- Purpose: To convert the individual pixels in a bmp structure from normal to those reflected by color blindness.
- Explanation: The code for this function is entirely given in the assignment document. I do not understand the workings or math behind this algorithm, simply copied it.

Results

The program works perfectly for all the .bmp files provided. Valgrind passes with 0 errors and 0 memory leaks for all bmp files and so does scan-build. No memory is left unattended on the heap after the program runs.

Makefile: Referred to Jess' and Ben's docs on how to have multiple executables. Used previous assignments Makefile as a template. Initially printed help message to stderr but fixed that to reflect printed help message to stdout instead (figured out to do this from class discord). iotest ran with no errors or memory leaks after completing my read functions. Created another .c file to test my write files locally first before pushing it to the pipeline to verify.

Finally, also used the cb.sh script to run all the files at once and to also test valgrind and scan-build on all .bmp files. No known or obvious errors or bugs because of the generously provided pseudocode from the assignment doc.

Images produced are provided below:

Figure 1: Pizza (original vs converted)

Figure 2: Froot Loops (original vs converted)

Figure 3: Apples (original vs converted)

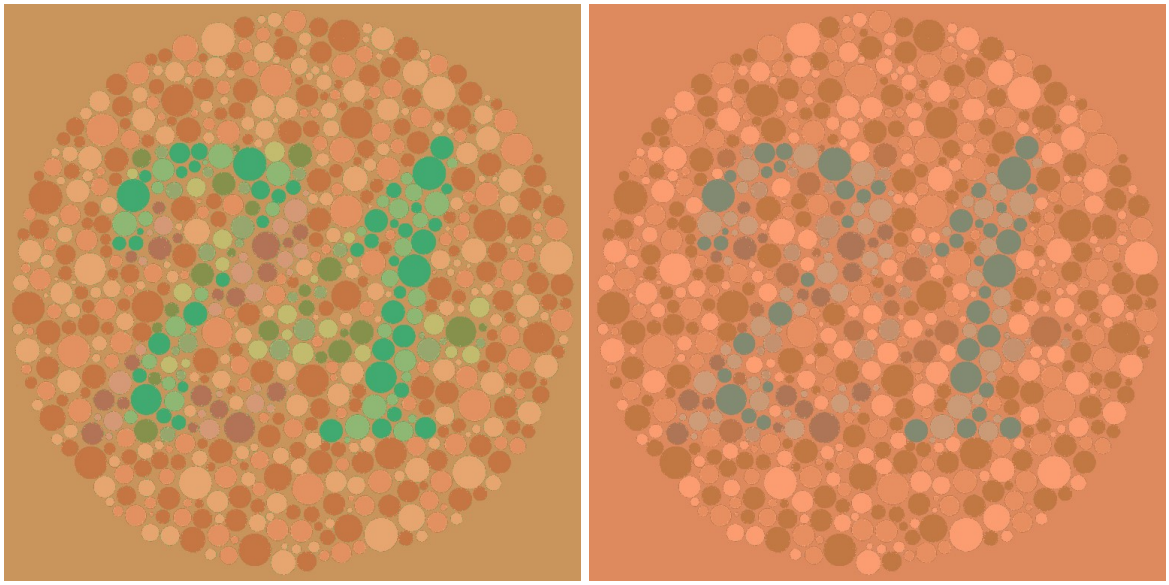

Figure 4: Ishihara (original vs converted)