Assignment 5 – Color Blindness Simulator

Priya Chaudhari

CSE 13S – Spring 2023

Purpose

The main purpose of this program is to mimic color blindness by manipulating color information in image files inputted by the user. We will be utilizing buffered and unbuffered file inputting and outputting functions to mimic this experience of deuteranopia. This image-processing program uses unbuffered I/O file functions to read in and write to binary files written in io.c and employed by a separate source file called bmp.c.

How to Use the Program

In order to use the program, the user needs to enter the command "make" within the terminal which will compile the Makefile, and then enter one of the 3 main command line options to bring data into my program. Below is a list of all the command line options (inputs and outputs) and what the user would need to enter for a specific purpose. Additionally, there's a special folder within my cse13s/asgn5 directory labeled bmps which contains a collection of images that I will input into my program and it should alter the makeup to replicate what deuteranopia would look like.

1. -i: the input file which my program will read and set the name to, the user must enter their filename as an argument
2. -o: the output file that my program will write to and set the name to, the user must enter their filename as an argument, and it will default to stdout
3. -h: will print a help message to stdout

The following are examples of commands that can be inputted into the terminal once Make is run. My color.c file is responsible for utilizing these commands and altering the images in the bmps directory accordingly. Below in the section "Results", are examples of what occurs when these commands are inputted by the user.

1. $./colorb - ibmps/apples - orig.bmp - obmps/apples - colorb.bmp$ ./colorb -i bmps/cereal-orig.bmp -o bmps/cereal-colorb.bmp
3. $./colorb - ibmps/froot - loops - orig.bmp - obmps/froot - loops - colorb.bmp$ ./colorb -i bmps/ishihara-9-orig.bmp -o bmps/ishihara-9-colorb.bmp
4. $./colorb - ibmps/produce - orig.bmp - obmps/produce - colorb.bmp$ ./colorb -i bmps/color-chooser-orig.bmp -o bmps/color-chooser-colorb.bmp

Program Design

The program is designed with 3 main source files- 2 of which are files (bmp.c and io.c) and one main file which will be the executable for this program (colorb.c).

Main Files:

6. source files (authored by me): bmp.c (contains BMP functions) and io.c (serialization/deserialization functions)
2. executable/main file (authored by me): colorb.c
3. header files (provided and unmodified): io.h, iotest.c, and bmp.h

Main Structures/Algorithms:

1. Unbuffered File I/O Structure: io.c uses standard file I/O functions and operations to describe and work with the Buffer structures
2. Marshaling/Serialization: Serializes and deserializes data when working with different file formats and data types (uint16, 32, and 8) and is a concept utilized with my read and write functions in io.
3. BMP: Reading and writing BMP image files and working with the BMP struct and file format.

Data Structures

1. Buffer Structure: Manages the file unbuffered I/O operations and includes elements like the respective buffer's size, data, position, and file descriptor it belongs to.

```
typedef struct buffer Buffer;
    struct buffer {
        int fd; // file descriptor from open() or creat()
        int offset; // offset into buffer a[]
            // next valid byte (reading)
            // next empty location (writing)
        int num_remaining; // number of bytes remaining in buffer (reading)
        uint8_t a[BUFFER_SIZE]; // buffer
};
```

2. BMP Structure: This is the Windows BMP image file format and includes elements like width, height, color depth, and pixel data. Responsible for reading and writing the BMP image files and manipulating the color blindness process.

```
    typedef struct color {
        uint8_t red;
        uint8_t green;
        uint8_t blue;
    } Color;

    typedef struct bmp {
        uint32_t height;
        uint32_t width;
        Color palette[MAX_COLORS];
        uint8_t **a;
    } BMP;
```

Function Descriptions

io.c

- Buffer *read_open(const char *filename): opens a file to be read and returns a pointer to a new buffer structure. Also responsible for initializing the file descriptor and other parts of the buf struct. Will also return NULL if memory allocation failed.

- void read_close(Buffer **pbuf): closes file with buffer and frees the memory used to prevent memory leaks. Sets pointer to buf as NULL to indicate it's closed.
- bool read_uint8(Buffer *buf, uint8_t *x): reads a single byte from the buffer and returns true if it was successfully read or false if the end of the file has been reached.

```
bool read_uint8(Buffer *buf, uint8_t *x){
    if(buf->num_remaining == 0){
            ssize_t rc = read(buf->fd, buf->a, sizeof(buf->a));
            if (rc < 0){
                    printf("Error handling reading operations for uint8 buffer.");
            }
            if (rc == 0){
                    return false; // end of file
            }

            buf->num_remaining = rc;
            buf->offset = 0;
            }

    //storing next byte in the buffer in *x
    *x = buf->a[buf->offset];    //*x = memory location pointed to by x
    buf->offset++;
    buf->num_remaining--;

    return true;
}
```

- bool read_uint16(Buffer *buf, uint16_t *x): reads 2 bytes from the buffer and deserializes them into a uint16 type value. Again, returns true if both bytes were read and false if the end of the file was reached.
- bool read_uint32(Buffer *buf, uint32_t *x): reads 4 bytes from the buffer and deserializes them into a uint32 type value. Again, returns true if all 4 bytes were read and false if the end of the file was reached.
- Buffer *write_open(const char *filename): opens file for writing and returns pointer to new buf structure. Also responsible for initializing the file descriptor and other parts of the buf struct. Will also return NULL if memory allocation failed.
- void write_close(Buffer **pbuf): writes remaining data in buffer to file and closes it. Also frees the memory and sets pointer to buf to NULL.
- void write_uint8(Buffer *buf, uint8_t x): writes 1 byte to buffer/file. if it's full it'll reset the buffer and add the byte to the buffer and update the offset.

```
void write_uint8(Buffer *buf, uint8_t x){
    if (buf->offset == BUFFER_SIZE){
            uint8_t *start = buf->a;
            int num_bytes = buf->offset;
            do {
                    ssize_t rc = write(buf->fd, start, num_bytes);
                    if (rc < 0){
                            printf("Error handling write operations.");
                            return;
                    }
```

```
                    start += rc; // skip past the bytes that were just written
                    num_bytes -= rc; // how many bytes are left?
    } while (num_bytes > 0);
            buf->offset = 0;  //resetting buffer since it's already empty here
    }

    buf->a[buf->offset] = x; //adding new byte to buf
    buf->offset++; //incrementing offset

}
```

- void write_uint16(Buffer *buf, uint16_t x): serializes the uint16 variable into 2 bytes and writes them to the buffer/file. Calls write_uint8 twice to do this.
- void write_uint32(Buffer *buf, uint32_t x): serializes a uint32_t value into 4 bytes and writes them to the buffer/file. Calls write_uint16 twice to do this.

bmp.c

- BMP *bmp_create(Buffer *buf): reads the contents of a BMP image from a buffer and creates a BMP struct representing the image. Goes theough the header and palette info from the buffer and stores them into the BMp struct and then a pointer to this struct is returned.
- void bmp_free(BMP **buf): frees memory allocated with the BMP struct and frees the double-pointer passed in as a parameter and sets it to NULL, then finally frees the memory.
- void bmp_write(const BMP *bmp, Buffer *buf): writes the contents of a BMP struct to the buffer in BMP file format. Writes the file header, bitmap header, palette, and pixel data to the buffer.
- int constrain(int x, int a, int b): takes the integer x and puts it within the range [a,b].
- void bmp_reduce_palette(BMP *bmp): reduces the color palette of a BMP image using the color transformation algorithm provided. This function is what is mainly responsible to stimulate color blindness with manipulating the RGB values of an image file and modifies the BMP struct by updating the palette with the transformed colors.

colorb.c

- Define necessary header files and libraries
- Define the command-line options and arguments and declare variables for the input file, output file, input buffer, output buffer, and BMP structure
- Implement the main function
- Handle each command-line option: Store the input file name in infile variable, use read_open to open the input file and store the resulting buffer in inputBuf, store the output file name in outfile variable. Use write_open to open the output file and store the resulting buffer in outputBuf. Also, have the help message command line option and include if statements in case there's an error that occurs with reading the files.
- Check if both the input and output file names are assigned (not NULL) with an if statement
- Create a BMP structure from the input buffer with bmp_create and call bmp_reduce_palette
- Write to the output buffer with bmp_write and free with bmp_free
- Finally close input and output file buffers and return 0 to mark the end of the main function

Makefile

- Define compiler and compiler flags : like -Wall -Wextra -Werror -pedantic -Wstrict-prototypes, clang(clang-format), and LDFLAGS
- Define object files, executables, and dependencies
- Define build rules: includes compiling all, colorb.c, iotest.c, and make clean (rm), and make format.

Results

- My code does successfully achieve everything it should, although I do feel there may be room for improvement. I feel I could add more print statements for error handling which may be helpful for the user. I feel my error print statements throughout my program were not as consistent as they could be, and I also should possibly add more memory allocation error print statements. I also feel like I could add more comments throughout my code, as I mostly add comments for functions I have a harder time grasping so I can better conceptualize it with everyday language comments right next to it.
- I learned much from implementing concepts in this assignment, especially file I/O functionality and working with BMP files (not solely working with stdin and stdout file descriptors). I feel I have a much better grasp on using buffers and important factors like knowing when to manipulate the image data, how memory allocation works and its significance, and learning how to write the modified data back to the files. I felt challenged, but ultimately now very grateful to know how to read, write, open, and close files without being reliant on fscanf, fprintf, fopen, close, etc. functions. Knowing how to implement these in a respective source file and integrating them in my main executable file was super interesting to learn. Another lesson I found pretty enlightening was ultimately manipulating the images already given in the bmps directory and the novelty of having that command line was cool to learn.
- The following image is an example of what happens when my program is executed. I used the command line "./colorb -i bmps/froot-loops-orig.bmp -o bmps/froot-loops-colorb.bmp". The following figure shows the original image on the left and the image on the right is the output of the color blindness simulation algorithms.

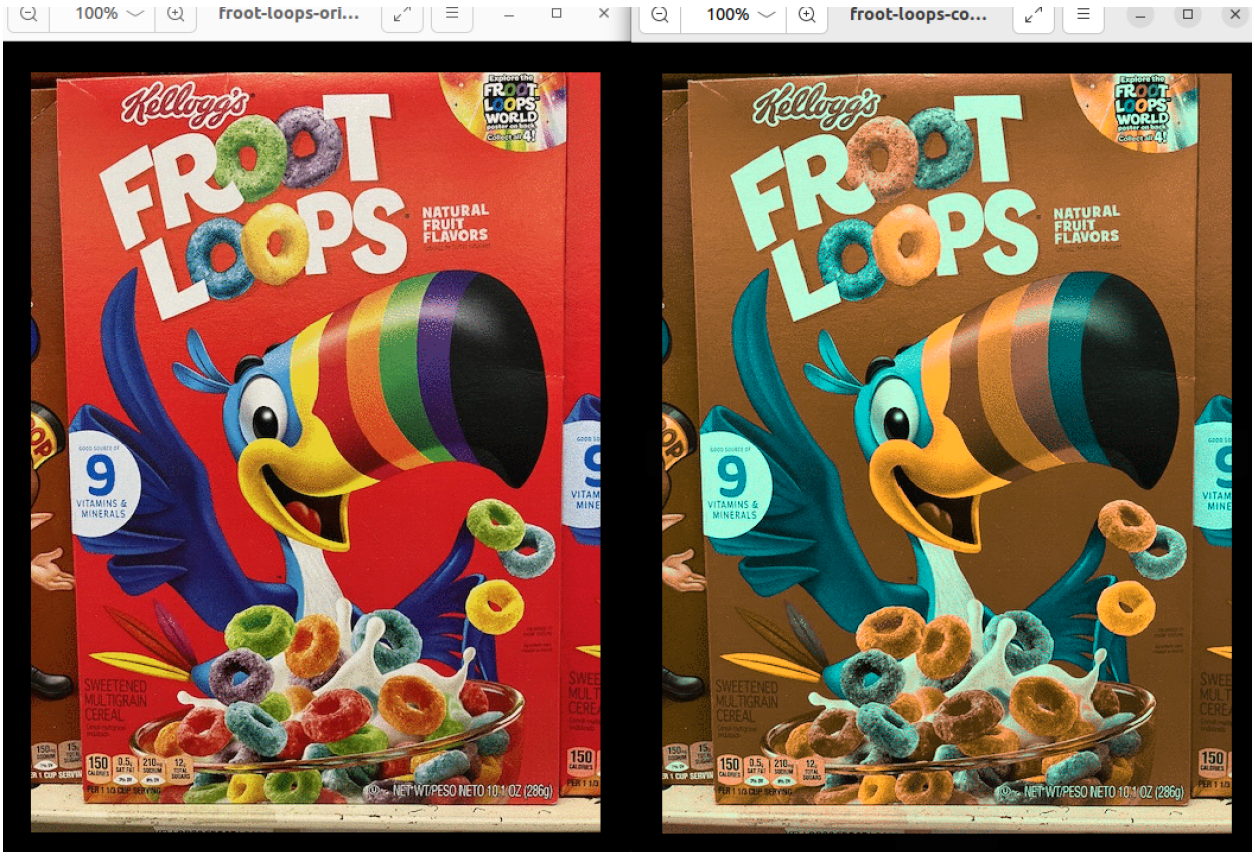

Figure 1: Screenshot of the output of 1 command line.