Assignment 5

Color Blindness Simulator

By Atharva Tawde

CSE 13S Spring 2023

Due May 28th, 2023 by 11:59 PM

Purpose

This assignment aims to facilitate the usage of matrix manipulation and file IO in order to filter a bitmap image into its *deuteranopic* counterpart. The resulting images contain dulled yellows, reds and greens, as *deuteranopia* refers to the condition where a person is *red-green* blind.

How to Use the Program

The program is written in C, and thus, will use the CLANG compiler to generate an executable that can be run later on. The specific CLANG command is stored in a Makefile, which can be executed using `make`. Arguments can be specified before executing the file in order to select the input image that undergoes the filter transformation and the name of the output image. For example, `./colorb -i input.bmp -o output.bmp` will read and process the bitmap data stored in `input.bmp`, apply the deuteranopic transformation, and write the modified data to `output.bmp`.

Table of Arguments

Argument	Function
`-i`	Sets the file to read from (input file). Requires a filename as an argument. Must be included in order to run as expected.
`-o`	Sets the file to write to (output file). Requires a filename as an argument. Must be included in order to run as expected.
`-h`	Prints a help message to `stdout`.

Program Design

Each abstract data structure will be written in C and stored in a file specific to the structure. For example, the BMP processing implementation is declared in `bmp.h`, and defined in `bmp.c`. The program that the user directly interacts with is located in `colorb.c`, and is linked to all the data type files.

Data Structures

A `struct` is defined to hold the buffer data as it is read and written to the BMP file. A buffer has methods to instantiate itself and read and write to itself in 8, 16 and 32 bit increments. Another `struct` is defined to hold bitmap information, called `BMP`. `BMP` includes methods to instantiate itself, read in bitmap information from an existing image, apply the deuteranopic filter and write in bitmap information to an output file.

Algorithms

There are bitwise algorithms used to read and write data to a file: `read_uint8` and `write_uint8`. `read_uint8` saves data at a certain point in the buffer into a pointer to be processed at a later time. It is able to extract 8-bit segments of data from the buffer. In contrast, `write_uint8` writes 8-bit data at a certain point in the buffer. Both processes update the location of the offset within the buffer (functions as a pointer to the next bit in the buffer).

Write 8-Bit Data

Define `write_uint8(Buffer, x)`.

- If the buffer pointer is at the end of the buffer
  - Write the contents of the buffer to the file specified.
- Else
  - Fill the buffer up with the 8-bit piece of data `x`.
  - Update the buffer pointer by 8 bits.

Read 8-Bit Data

Define `read_uint8(Buffer, *x)`.

- If the buffer is empty
  - Read the contents of the file specified.
- Else
  - Fill the buffer up with the next 8 bits of data and let `x` point to this data.
  - Update the buffer pointer by 8 bits.

Pseudocode

`colorb.c`

Parse through the user-delivered arguments using `getopt()`.
Initialize three booleans to track the presence of each argument.

Argument	`-i`	`-o`	`-h`
Function	Set input file.	Set output file.	Help section.

Set a specific boolean to `true` in order to track the presence of an argument. For example, if `-i` is present, set `bool_i` to `true`.

Instantiate two Buffers `r` and `w`.

While `getopt()` is not finished with parsing (has not returned -1)
    Switch through each option
        Case `i`
            Set `bool_i` to `true`.
            Set `r` to the provided argument.
        Case `o`
            Set `bool_o` to `true`.
            Set w to the provided argument.
        Case `h`
            Set `bool_h` to `true`.

Instantiate a BMP `bmp`.

Parse through the `.bmp` input file.
Create a BMP `bmp` with this data.
Apply the filter to bmp using `bmp_reduce_palette()`.
Write `bmp` to the output destination.
Free `bmp`.
Close file specified in `r`.
Close file specified in `w`.

`io.c`

Include `io.h` (which declares all File IO methods).

Define a new type `struct Buffer`.

A buffer contains the file `fd` it operates on, the position of the next available bit `offset` in the buffer, and the number of items `num_remaining` left to read from the file.

Define `read_open(const char *filename)`.

Open the file `filename` in read only mode and store the value in `f`.

If `f` is 0

Return `NULL` to indicate a failed opening.

Else

Allocate space for a new Buffer `b`.

Set `b.fd` to `f`.

Set `b.num_remaining` to `0`.

Set `b.offset` to `0`.

Return `b`.

Define `read_close(Buffer **pbuf)`.

Close the file `pbuf.fd`.

Free `pbuf`.

Set `pbuf` to `NULL`.

Define `read_uint8(Buffer *buf, uint8_t *x)`.

If `buf` is empty

Read and populate `buf`.

Set the number of remaining bytes to be read to the amount of bytes successfully read from the file.

Set the current position of the pointer to 0.

Return false if there are any errors with reading the file.

Else

Set the value of `x` to the value stored in the buffer at the current position.

Update the current position of the buffer by 8.

Update the remaining amount of bits to read by 8.

Return true.

Define `read_uint16(Buffer *buf, uint16_t *x)`.

Call `read_unit8()` for the bits in the current position and 8 bits ahead of the current position to read 16 bits at once. Offset the latter and bitwise OR the two results together.

Define `read_uint32(Buffer *buf, uint32_t *x)`.

Call `read_unit32()` for the bits in the current position and 16 bits ahead of the current position to read 32 bits at once. Offset the latter and bitwise OR the two results together.

Define `write_open(const char *filename)`.

Create a file with the permission code `0664`.
If file creation failed
Return `NULL`
Else
Allocate space for a new Buffer `b`.
Set `b.fd` to `f`.
Set `b.num_remaining` to `0`.
Set `b.offset` to `0`.
Return `b`.

Define `write_close(Buffer **pbuf)`.

Write everything left in `pbuf` to the file.

Close the file `pbuf.fd`.
Free `pbuf`.
Set `pbuf` to `NULL`.

Define `write_uint8(Buffer *buf, uint8_t x)`.

If `buf` is full
Set the pointer `start` to the pointer of the first value in `buf`.
Set `num_bytes` to the number of bytes filled up in `buf`.

Do
Write the values within `buf` to the file specified.
If there is a problem with this operation.
Print an error.
Update `start` to how many bits were successfully written.
Subtract `num_bytes` by how many bits were successfully written.
While `num_bytes` is greater than 0.

Populate the buffer byte at the current position to the value passed in.
Update the current position in the buffer by 8.

Define `write_uint16(Buffer *buf, uint16_t x)`.

Call `write_uint8()` on the first 8 bits of the value, and a second time on the last 8 bits of the value passed in.

Define `write_uint32(Buffer *buf, uint32_t x)`.

Call `write_uint16()` on the first 16 bits of the value, and a second time on the last 16 bits of the value passed in.

`bmp.c`

Include `bmp.h` (which declares all BMP methods).

Define a new type `struct Color`.

- Declare an 8 bit integer `red`.
- Declare an 8 bit integer `green`.
- Declare an 8 bit integer `blue`.

Define a new type `struct BMP`.

- Declare a 32 bit unsigned integer `height`.
- Declare a 32 bit unsigned integer `width`.
- Declare an array of `Color`s of `MAX_COLORS` (256).

Define `bmp_create(Buffer *buf)`.

- Allocate space for a new BMP bmp.
- Use `read_unit8()`, `read_unit16()`, `read_unit32()` to extract required data for bitmap reconstruction in the future.
- Populate bmp with data.

Define `bmp_free(BMP **bmp)`.

- Free every pixel within the image stored by `bmp`.
- Free the pointer of the list of pixels.
- Free `bmp` itself.

Define `bmp_write(const BMP *bmp, Buffer *buf)`.

Set default values as listed in the table below.

Variable	Value
`rounded_width`	`bmp`'s width rounded to the nearest multiple of four.
`image_size`	`bmp`'s height x `rounded_width`
`file_header_size`	14
`bitmap_header_size`	40
`num_colors`	256
`palette_size`	4 x `num_colors`
`bitmap_offset`	`file_header_size +`

	`bitmap_header_size + palette_size`
`file_size`	`bitmap_offset + image_size`

Write values to the buffer that comply with the bitmap format, filling it with the new values of each pixel.

Define `constrain(int x, int a, int b)`.

Return `x` bounded by `a` and `b`.

Define `bmp_reduce_palette(BMP *bmp)`.

Calculate the deuteranopic transformation to `bmp`.

Update the pixel to the post-filter color.

Results

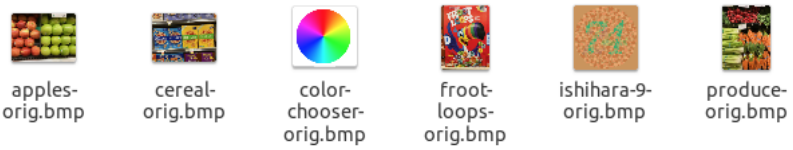
Reflection

While writing this program, I learned how to manage unbuffered input and output in order to construct a bitmap. Although the implementation of the filter was abstracted for us, it was informative to be able to go through the process of writing those pixels back. I also learned how to use the octal dump `od` command to verify the similarity between two bitmaps, which is a very nice debugging tool. I also learned a little bit about the BMP format itself, and how specific the format is required to be in order to create a valid image.

Program Output

The program allows the user to enter four possible arguments `-i` *`input_file`*, `-o` *`output_file`*, `-h`. `-i` or `-o` are required in order for the program to run and they require the name of a text file passed in next to them in order to create Buffers to read and write to during the execution of the program. Refer to the Table of Arguments to see functionality.

Below is a picture of the directory that `colorb` is working on in this example.



If all arguments are provided properly, then no output will display in the terminal. Rather the new images will be created and stored in the specified directory.

If `-i` or `-o` are missing, the program lets the user know about the failure to detect the presence of these arguments.

```
vboxuser@CoolBox:~/development/atawde/asgn5$ ./colorb
colorb:  -i option is required
Usage: colorb -i infile -o outfile
       colorb -h
```

During a successful execution, the deuteranopic filter is applied to the image specified as expected.

Before	After
	
`apples-orig.bmp`	`apples-colorb.bmp`

References

[Documentation for `read()` from `unistd.h`]

[Explanation of the Windows BMP File Format]