Assignment 5 – Color Blindness Simulator

Aditya Bhaskar

CSE 13S – Spring 2023

Purpose

The purpose of this program is to process BMP (Bitmap) image files by applying a color manipulation algorithm called "colorb." The program takes an input BMP image file and generates an output file with modified colors based on the colorb algorithm. The colorb program allows users to enhance or alter the colors of BMP images, providing greater flexibility and creative control over the visual appearance of the images. By running the colorb program, users can transform ordinary images into visually appealing and customized versions with adjusted color schemes. This program is designed for professionals and enthusiasts working with digital images in various industries, such as graphic design, advertising, photography, and multimedia. It provides a convenient tool for manipulating colors in BMP files, enabling users to achieve desired artistic effects or match specific color palettes. With its command-line interface and support for batch processing, the program allows users to efficiently process multiple BMP images in one go, saving time and effort. It also includes options for specifying input and output filenames, ensuring flexibility and ease of integration into existing workflows. The colorb program can be utilized as part of a larger image processing pipeline or as a standalone tool, depending on the specific requirements of the industry professional. Its ability to read and write BMP files makes it compatible with various image editing software and platforms.

Overall, this program empowers industry professionals to manipulate the colors of BMP images, providing them with a powerful tool to enhance their creative work, achieve specific visual effects, and deliver impactful visual content.

How to Use the Program

Using the colorb Program: A User Guide

The colorb program provides you with a convenient way to manipulate the colors of BMP (Bitmap) image files. This user guide will walk you through the process of compiling and running the program, as well as explain the available options and their effects on the program's output.

- **Compiling the program:** To compile the colorb program, follow these steps:
  - Ensure that you have the Clang compiler installed on your system. If not, please install Clang before proceeding.
  - Download the source code files for the colorb program, which include colorb.c, bmp.c, io.c, bmp.h, io.h, iotest.c, and Makefile.
  - Navigate to the directory (in this case the VM) where the program files and the Makefile are located.
  - Open a terminal or command prompt and navigate to the directory where you saved the source code files.
  - Run the command make all or simply make to compile both the colorb program and the I/O test program (iotest).
  - After successful compilation, you will have an executable file named colorb and iotest in the same directory.

- **Running the Program:** To run the colorb program, follow these steps:
  - Open a terminal or command prompt and navigate to the directory where the colorb executable is located.
  - The basic command to run the program is as follows:

    ```
    ./colorb -i input_file -o output_file [options]
    ```

    Replace **input_file** with the path and filename of the BMP image file you want to process. Similarly, replace **output_file** with the desired path and filename of the resulting modified BMP image file.
  - Additionally, you can include various options to customize the program's behavior. Here are the available options:
    - **-i input_file:** Specifies the input BMP image file. Replace **input_file** with the path and filename of the BMP image you want to process. This option is required.
    - **-o output_file:** Specifies the output file where the modified BMP image will be saved. Replace **output_file** with the desired path and filename for the output file. This option is required.
    - **-h:** Prints a help message to the standard output, providing information about the program's usage and available options.
  - After specifying the input file, output file, and any desired options, press Enter to execute the program.
  - The colorb program will process the input BMP image according to the specified options and generate the modified image file as the output. The program will display the progress and any relevant messages during the process.
- **Cleaning Up:** After you have finished using the colorb program, you may want to remove the files generated during compilation. To clean up the files, follow these steps:
  - Open a terminal or command prompt and navigate to the directory where the colorb program is located.
  - Run the command make clean. This command will delete all object files and compiled programs, including the colorb executable.

Program Design

In this section, we will explore the organization and structure of the colorb program, providing insights into the main data structures and algorithms used. This information will help you understand how the program is organized and make it easier for you to maintain and troubleshoot any issues that may arise.

- **Main Data Structures:** The colorb program primarily utilizes the following data structures:
  - BMPImage Structure: This structure represents a BMP image and contains information about its width, height, color depth, and pixel data. It is defined in the bmp.h header file and is used throughout the program for image manipulation.
  - Pixel Structure: This structure represents a single pixel in a BMP image and contains the red, green, and blue color components. It is used within the BMPImage structure to store the pixel data for each image.
- **Main Algorithms:** The colorb program employs several algorithms to manipulate the colors of BMP images. Here are the main algorithms used:

- Reading and Writing BMP Files: The program uses algorithms to read the content of a BMP image file and load it into memory as a BMPImage structure. It also includes algorithms to write the modified BMPImage structure back to a BMP file, preserving the file format and header information.
- Color Manipulation: The program provides algorithms to modify the colors of BMP images based on the specified options. These algorithms can perform operations such as adjusting brightness, applying grayscale conversion, and swapping color channels.
- Command-Line Argument Parsing: The program uses algorithms to parse the command-line arguments provided by the user. These algorithms extract the input and output file paths, as well as any additional options, enabling the program to perform the requested color modifications.
- Error Handling and Reporting: The program incorporates algorithms for error handling and reporting. These algorithms help identify and handle various error conditions, such as invalid file formats, missing files, or incorrect command-line options. They provide meaningful error messages to the user, facilitating troubleshooting and issue resolution.

- **Code Organization:** The colorb program is organized into multiple source code files to enhance modularity and maintainability. Here's an overview of the key files and their purpose:
  - colorb.c: This file contains the main() function, which serves as the entry point of the program. It handles command-line argument parsing, coordinates the execution of color manipulation algorithms, and manages error handling and reporting.
  - bmp.c and bmp.h: These files define the functions and data structures related to BMP image manipulation. They include algorithms for reading and writing BMP files, as well as functions to modify the colors of the image pixels.
  - io.c and io.h: These files provide the serialization and deserialization functions for the BMPImage structure. They handle the conversion between the in-memory representation of a BMP image and its binary file format.
  - iotest.c: This file contains additional test cases to verify the correctness of the I/O functions implemented in io.c. It is used for testing purposes and can be expanded to include more comprehensive test scenarios.
  - Makefile: This file contains the instructions for compiling the program and managing dependencies. It ensures that the program is built correctly, includes necessary compiler flags, and allows for easy compilation and cleanup.

By understanding the organization and structure of the colorb program, we will be better equipped to maintain and troubleshoot issues. The use of clear data structures, well-defined algorithms, and modular code organization helps ensure the program's maintainability and makes it easier to identify and resolve any issues that may arise in the future.

Data Structures and Functions

- **IO.C:**
  - **Data Structure:**
    - **struct buffer:** Represents a buffer containing data to be read from or written to. It consists of the following fields:
      - **int fd:** File descriptor representing the open file associated with the buffer.
      - **uint8_t a[BUFFER_SIZE]:** Array to store the data in the buffer.
      - **size_t num_remaining:** Number of bytes remaining in the buffer.
      - **size_t offset:** Current offset within the buffer.
  - **Functions:**

* **Buffer *read_open(const char *filename):** Opens a file for reading and initializes a new Buffer structure. Returns a pointer to the created buffer.
* **void read_close(Buffer **pbuf):** Closes the file associated with the buffer and frees the memory allocated for the buffer structure.
* **bool read_uint8(Buffer *buf, uint8_t *x):** Reads a single byte from the buffer and stores it in the memory location pointed to by x. Returns true if successful, false otherwise.
* **bool read_uint16(Buffer *buf, uint16_t *x):** Reads two bytes from the buffer, interprets them as a little-endian 16-bit unsigned integer, and stores the result in the memory location pointed to by x. Returns true if successful, false otherwise.
* **bool read_uint32(Buffer *buf, uint32_t *x):** Reads four bytes from the buffer, interprets them as a little-endian 32-bit unsigned integer, and stores the result in the memory location pointed to by x. Returns true if successful, false otherwise.
* **Buffer *write_open(const char *filename):** Opens a file for writing and initializes a new Buffer structure. Returns a pointer to the created buffer.
* **void write_close(Buffer **pbuf):** Writes any remaining data in the buffer to the associated file, closes the file, and frees the memory allocated for the buffer structure.
* **void write_uint8(Buffer *buf, uint8_t x):** Writes a single byte to the buffer.
* **void write_uint16(Buffer *buf, uint16_t x):** Writes a little-endian 16-bit unsigned integer to the buffer.
* **void write_uint32(Buffer *buf, uint32_t x):** Writes a little-endian 32-bit unsigned integer to the buffer.

The functions transfer data between different functions by accepting a Buffer pointer as one of the parameters. This allows the functions to access and modify the buffer's fields **(fd, a, num_remaining, offset)** as necessary to perform the read or write operations. The buffer itself is created and passed between functions to maintain the state and progress of the read/write operations.

The options passed into the functions are not explicitly stored within the data structure. Instead, the functions rely on the input parameters passed to them (such as filename, x) to perform the desired operations.

- **BMP.C:**
  - **Data Structures:**
    * **Color:** Represents a single color in the BMP image. It consists of three **uint8_t** values for the red, green, and blue components.
    * BMP: Represents a BMP image. It contains the following members:
      - **height and width:** The height and width of the image in pixels.
      - **palette:** An array of Color structures representing the color palette of the image. It has a maximum size of **MAX_COLORS**.
      - **a**: A two-dimensional array of **uint8_t** values representing the pixel data of the image.
    * **Buffer:** Represents a buffer used for reading or writing data from/to a file.
  - **Functions:**
    * **bmp_create(Buffer *buf):** Reads data from a Buffer and creates a BMP image structure. It reads various fields from the buffer, such as the image type, size, header information, color palette, and pixel data, and stores them in the BMP structure.
    * **bmp_free(BMP **bmp):** Frees the memory allocated for a BMP image structure. It iterates over the a array and frees each row before freeing the main a array and the BMP structure itself.
    * **bmp_write(const BMP *bmp, Buffer *buf):** Writes the BMP image data to a Buffer. It calculates the required file size, offset, and other header fields based on the BMP structure and writes them to the buffer. It then writes the color palette and pixel data to the buffer.

* **constrain(int x, int a, int b):** A utility function that constrains the value x between a and b. It is used in the **bmp_reduce_palette** function to ensure color values are within the valid range.
* **bmp_reduce_palette(BMP *bmp):** Reduces the color palette of a BMP image. It iterates over each color in the palette and performs color transformation based on specific equations. The transformed color values are constrained to the valid range and stored back into the palette.

The functions receive the necessary data as function parameters, such as the Buffer object for reading or writing, and the BMP structure for image-related operations. Some functions also modify the BMP structure directly to update the image data.

- **COLORB.C:**
  - **Data Structures:**
    * **Buffer:** A data structure that represents a buffer used for reading or writing data.
    * **BMP:** A data structure that represents a BMP image. It contains attributes such as height, width, and palette. The actual image data is stored in the a array.
  - **Variable Declarations:**
    * **input_filename and output_filename:** Pointers to store the input and output file names respectively.
  - **Command-line Argument Parsing:**
    * The program takes command-line arguments and parses them to determine the input and output file names. The -i option is used to specify the input file name, and the -o option is used to specify the output file name.
  - **Command-line Argument Handling:**
    * The program checks if the required input and output filenames are provided. If not, it displays an error message and usage instructions.
  - **Reading the BMP Image:**
    * The program opens the input file using the provided input filename and creates a Buffer object **(read_buffer)** to read data from the file.
    * The **read_open()** function reads the input file and populates the **read_buffer**.
    * The **bmp_create()** function is called, passing the **read_buffer** as a parameter, to create a BMP object (bmp) and populate it with the image data.
    * After reading the image data, the input file is closed using the **read_close()** function.
  - **Modifying the BMP Image:**
    * The program calls the **bmp_reduce_palette()** function, passing the bmp object as a parameter, to modify the BMP image. This function reduces the color palette.
  - **Writing the Modified BMP Image:**
    * The program opens the output file using the provided output filename and creates a Buffer object **(write_buffer)** to write data to the file.
    * **The write_open()** function prepares the output file for writing.
    * The **bmp_write()** function is called, passing the bmp object and the **write_buffer** as parameters, to write the modified BMP image data to the **write_buffer**.
    * After writing the image data, the output file is closed using the **write_close()** function.
  - **Freeing Allocated Memory:**
    * The program calls the **bmp_free()** function, passing the address of the bmp object, to free the allocated memory for the bmp object and its associated data.
  - **Output and Error Handling:**

* The program displays appropriate error messages if any errors occur during file operations or image manipulation.
* If the program runs successfully, it displays a message indicating the completion of the color manipulation process and the name of the output file.

The program reads the input BMP image, modifies it, and writes the modified image to an output file. It uses data structures such as Buffer and BMP to store and manipulate the image data, and transfers the data between functions using function parameters.

Overall, the data structures facilitate the storage and manipulation of image data and options throughout the program, allowing the necessary information to be passed between functions and ensuring coherent processing of the BMP images.

Algorithms

```
# Pseudocode for IO.C

class Buffer:
    def __init__(self, fd, a, num_remaining, offset):
        self.fd = fd
        self.a = a
        self.num_remaining = num_remaining
        self.offset = offset

def read_open(filename):
    fd = open(filename, "rb")
    if fd is None:
        return None
    buf = Buffer(fd, bytearray(), 0, 0)
    return buf

def read_close(pbuf):
    if pbuf is None or pbuf.fd is None:
        return
    pbuf.fd.close()

def read_uint8(buf):
    if buf.num_remaining == 0:
        buf.a = buf.fd.read(BUFFER_SIZE)
        if not buf.a:
            return False
        buf.num_remaining = len(buf.a)
        buf.offset = 0

    x = buf.a[buf.offset]
    buf.offset += 1
    buf.num_remaining -= 1
    return True, x

def read_uint16(buf):
    success1, byte1 = read_uint8(buf)
    success2, byte2 = read_uint8(buf)
    if not success1 or not success2:
        return False, None
```

```
    x = (byte2 << 8) | byte1
    return True, x

def read_uint32(buf):
    success1, value1 = read_uint16(buf)
    success2, value2 = read_uint16(buf)
    if not success1 or not success2:
        return False, None

    x = (value2 << 16) | value1
    return True, x

def write_open(filename):
    fd = open(filename, "wb")
    if fd is None:
        return None
    buf = Buffer(fd, bytearray(), 0, 0)
    return buf

def write_close(pbuf):
    if pbuf is None or pbuf.fd is None:
        return
    pbuf.fd.write(pbuf.a[:pbuf.offset])
    pbuf.fd.close()

def write_uint8(buf, x):
    if buf.offset == BUFFER_SIZE:
        buf.fd.write(buf.a)
        buf.offset = 0
    buf.a[buf.offset] = x
    buf.offset += 1

def write_uint16(buf, x):
    write_uint8(buf, x & 0xFF)
    write_uint8(buf, (x >> 8) & 0xFF)

def write_uint32(buf, x):
    write_uint16(buf, x & 0xFFFF)
    write_uint16(buf, (x >> 16) & 0xFFFF)
```

```
# Pseudocode for BMP.C

class Color:
    def __init__(self, red, green, blue):
        self.red = red
        self.green = green
        self.blue = blue

class BMP:
    def __init__(self, height, width):
        self.height = height
```

```
        self.width = width
        self.palette = [Color(0, 0, 0)] * MAX_COLORS
        self.a = [[0] * height for _ in range(width)]

def bmp_create(buf):
    bmp = BMP(0, 0)
    # Read BMP header and populate bmp attributes
    # ...
    return bmp

def bmp_free(bmp):
    # Free allocated memory for bmp.a and bmp itself
    # ...

def bmp_write(bmp, buf):
    # Write BMP data to buf
    # ...

def constrain(x, a, b):
    # Constrain value x between a and b
    # ...

def bmp_reduce_palette(bmp):
    for i in range(MAX_COLORS):
        # Apply color transformation equations
        # ...
```

```
# Pseudocode for COLORB.C

def main(argv):
    input_filename = None
    output_filename = None

    # Parse command-line arguments
    for i in range(1, len(argv)):
        if argv[i] == "-i":
            if i + 1 < len(argv):
                input_filename = argv[i + 1]
            else:
                print("Error: Input filename not provided.")
                print("Usage: colorb -i <input_file> -o <output_file>")
                print("Options:")
                # Print the available options
                return 1
        elif argv[i] == "-o":
            if i + 1 < len(argv):
                output_filename = argv[i + 1]
            else:
                print("Error: Output filename not provided.")
                print("Usage: colorb -i <input_file> -o <output_file>")
                print("Options:")
                # Print the available options
                return 1
```

```
        elif argv[i] == "-h":
            print("Usage: colorb -i <input_file> -o <output_file>")
            print("Options:")
            # Print the available options
            return 0

    # Check if required options are provided
    if input_filename is None or output_filename is None:
        print("Error: Input and output filenames are required.")
        print("Usage: colorb -i <input_file> -o <output_file>")
        print("Options:")
        # Print the available options
        return 1

    # Read BMP image
    read_buffer = read_open(input_filename)
    if read_buffer is None:
        print(f"Failed to open input file: {input_filename}")
        return 1

    bmp = bmp_create(read_buffer)
    read_close(read_buffer)
    if bmp is None:
        print(f"Failed to read BMP image: {input_filename}")
        return 1

    # Modify the BMP image (e.g., reduce the palette)
    bmp_reduce_palette(bmp)

    # Write modified BMP image
    write_buffer = write_open(output_filename)
    if write_buffer is None:
        print(f"Failed to open output file: {output_filename}")
        bmp_free(bmp)
        return 1

    bmp_write(bmp, write_buffer)
    write_close(write_buffer)
    bmp_free(bmp)
    print(f"Color manipulation complete. Output file: {output_filename}")
```

Results

Simulated Photos results

Screenshots of program output, Apples. 1, Cereal. 1, Color-chooser. 1, Froot-loops. 1, Ishihara. 1, Pizza. 1, and Produce. 1.

Error Handling

Handling errors in the provided code is limited, but there are a few error scenarios that are addressed:

- Missing Input File: If the user does not provide an input filename using the -i option, an error message is printed: "Error: Input filename not provided." The program then displays the usage instructions and available options.
- Missing Output File: If the user does not provide an output filename using the -o option, a similar error message is printed: "Error: Output filename not provided." The usage instructions and available options are displayed as well.
- Failed File Operations: There are a few points in the code where file operations (open, close, read, write) are performed. If any of these operations fail, an error message is printed indicating the specific failure. For example, if opening the input file fails, the error message will be "Failed to open input file: filename". Similarly, if opening the output file fails, the error message will be "Failed to open output file: filename". These error messages indicate that the file operation encountered an issue.
- Failed BMP Operations: If there is an error in reading or creating the BMP image, an error message is printed. For instance, if reading the BMP image fails, the error message will be "Failed to read BMP image: filename". These error messages indicate that there was an issue with the BMP image manipulation.

These error messages provide some information about the encountered errors, allowing users to identify the problem areas. However, the error handling in the code is minimal and doesn't include detailed error codes or exceptions to provide more specific information about the errors.

Figure 1: Screenshot of apples-colorb.

Figure 2: Screenshot of cereal-colorb.

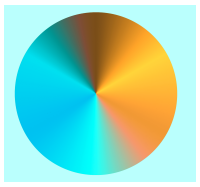

Figure 3: Screenshot of color-chooser-colorb.

Figure 4: Screenshot of froot-loops-colorb.

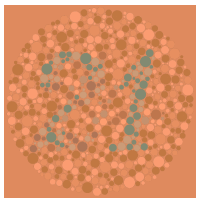

Figure 5: Screenshot of ishihara-9-colorb.

Figure 6: Screenshot of pizza-colorb.

Figure 7: Screenshot of produce-colorb.