Assignment 6 – Color Blindness Simulator

Asaveri Rao

CSE 13S – Fall 2023

Purpose

This program's basic function is converting a colorful image to an image that simulates deuteranopia(color-blindness). It utilizes various mechanisms to implement the program and to properly execute the program. One of the major purposes of this program is to read and write binary files, which are one of the mechanisms used to implement the program. The next major purpose behind this assignment is to practice marshalling also known as serializing.

How to Use the Program

This program consists of a few moving parts, and is on the less complex side. The basic premise of this assignment is to convert an element from one type of coloring to another type of coloring, one that simulates color-blindness. There are header files bmp.h and there is a bmp.c file that is parallel to the header file which contains functions that write in and out of bmp files out of bmp type. The main purpose of the bmp.c is to carry out the process of serializing or "marshalling" bmp files. The io.h file has a corresponding io.c file, which contains the serialization functions. There is a file called bmps that contains images that needs to be converted from regular coloring into coloring that simulates color-blindness. Similar to the other assignments, to compile and run the program, you need to have a functioning Makefile. To compile and format you have to run the commands, make format, make clean, make and there should be no errors that compile. After every edit, even if it is minor, the programmer has to run the commands make format, make clean, make to make sure the file is properly formatted and compiled. If the terminal does not return any errors, then that means it has properly compiled. To run the program you have to enter ./colorb -i and enter the name of the files in the bmps directory. For example you could manually enter: "./colorb -i bmps/apples-orig.bmp -o bmps/apples-colorb.bmp", "./colorb -i bmps/cereal-orig.bmp -o bmps/cereal-colorb.bmp", "./colorb -i bmps/froot-loops-orig.bmp -o bmps/froot-loops-colorb.bmp", "./colorb -i bmps/ishihara-9-orig.bmp -o bmps/ishihara-9-colorb.bmp", "./colorb -i bmps/produce-orig.bmp -o bmps/produce-colorb.bmp", "./colorb -i bmps/color-chooser-orig.bmp -o bmps/color-chooser-colorb.bmp". However, for a less complex approach, there is a file called cb.sh that utilizes shell script so to run the program the user can manually enter: ./cb.sh.

Program Design

This program utilizes three major c files. The io.h file that was provided has an io.c file in correspondence, there is a bmp.h file that has a corresponding bmp.c file. As mentioned before io.c, bmp.c, and colorb.c which contains the main function are what carry most of weight for the assignment.

Data Structures

The main data structures used in this program are structs. Overall use of functions, strings, and integers are also utilized. These data structures were used over other possible data structures because these were the most efficient to carry out the main purpose of reading and writing binary files, and to convert images

from one type of coloring to another. I will provide brief psuedo code for each of the c files. Here is the pseudocode for io.c file, which contains the serializing functions: io.c:

```
Function to read an 8-bit unsigned integer from a file:
void read_uint8(FILE *fin, uint8_t *px) {
    Read a byte from file:
    integer result is equal to fgetcharacter(fin);
    Check if  end of file is reached:
    if (result is equal to EOF) {
        Print an error message if the file is not valid;
    } else {
        Store the read value in the provided memory location
        *px is equal to (uint8_t) result;
    }
}
Function to read a 16-bit unsigned integer from a file:
void read_uint16(FILE *fin, uint16_t *px) {
    Declare variables to store two bytes:
    unsigned 8 bit integer byte1, byte2;

    Read the first byte:
    read unsigned 8 bit integer(fin, &byte1);

    Read the second byte:
    read unsigned 8 bit integer(fin, &byte2);

    Combine the two bytes to form a 16-bit integer and shift:
    *px = (uint16_t) (byte1 | (byte2 << 8));
}
Function to read a 32-bit unsigned integer from a file:
void read_uint32(FILE *fin, uint32_t *px) {
    Declare variables to store two 16-bit words:
    unsigned int16_t word1, word2;

    Read the first 16 bits:
    read unsigned int16(fin, &word1);

    Read the second 16 bits:
    read unsigned int16(fin, &word2);

    Combine the two words to form a 32-bit integer and shift:
    *px = (uint32_t) (word1 | (word2 << 16));
}
Write functions:
Function to write an 8-bit unsigned integer to a file:
void write_uint8(FILE *fout, uint8_t x) {
    Write the byte to the file
    int result is equal to fputc(x, fout);

    Check if the write operation was successful:
    if (result is equal to EOF) {
        Print an error message if the file is not valid;
    }
}
```

```
Function to write a 16-bit unsigned integer to a file:
void write_uint16(FILE *fout, uint16_t x) {
    Write the low byte of the 16-bit integer;
    Write the high byte of the 16-bit integer:
    write unsigned 8-bit integer(fout, (uint8_t) (x >> 8));
}
Function to write a 32-bit unsigned integer to a file:
void write_uint32(FILE *fout, uint32_t x) {
    Write the low word of the 32-bit integer:
    write unsigned 16-bit integer (fout, (uint16_t) x);
    Write the high word of the 32-bit integer:
    write unsigned 16-bit integer (fout, (uint16_t) (x >> 16));
}
```

bmp.c pseudocode:

```
Define a Color structure with red, green, and blue components
struct Color {
    unsigned 8-bit integer red;
    unsigned 8-bit integer green;
    unsigned 8-bit integer blue;
}

Define a BMP structure with height, width, palette, and pixel array:
struct BMP {
    unsigned 32-bit integer height;
    unsigned 32-bit integer width;
    Color palette;
    unsigned 8-bit integer **a;
}
Function to round up a number to the nearest multiple of another number:
unsigned 32-bit integer round_up(uint32_t x, uint32_t n) {
while (x is not divisible by n) {
    Increment x by 1
}
Return the updated value of x
}
Function to skip a specified number of bytes in a file
void skip(FILE *fin, uint32_t num_bytes) {
  Loop for the specified number of bytes:
    for unsigned 32 bit integer{
        Read and discard a byte from the file;
}
Function to create a BMP structure from a file:
BMP *bmp_create(FILE *fin) {
Read data from the input file
Verify conditions
Read palette
Allocate pixel array
}

Function to write a BMP structure to a file:
void bmp_write(BMP *pbmp, FILE *fout) {
```

```
Write data to the output file
Write the palette
Write the pixels
write extra pixels to make a multiple of 4 pixels per row
}
Function to free memory allocated for a BMP structure:
void free_bmp(BMP **ppbmp) {
If NULL, return to avoid dereferencing a NULL pointer
Loop through each row of the pixel array and free the memory
for unsigned 32-bit integer {
    Free the memory for the ith row of the pixel array
}
Free the memory for the pixel array itself
Free the memory for the BMP structure
Set the pointer to NULL to avoid potential issues
}

Function to constrain a double value to the range:
unsigned 8-bit integer constrain(double x) {
Round the value of x to the nearest integer
if x is less than zero {
    x is equal to 0;
}
if x is greater than UINT8_MAX) {

    x is equal to UINT8_MAX;
}
}

Function to reduce the palette of a BMP structure:
void bmp_reduce_palette(BMP *pbmp) {
Loop through each color in the palette
Extract the red, green, and blue components of the current color
Calculate two values, SqLe and SeLq, based on the color components
Check a condition based on SqLe and SeLq
 Update the color in the palette with the new components
}
```

Algorithms

This program has the main function located in the colorb.c. The main function is fairly simple. It consists of the being able to execute the command line options -i, -o, and -h that prints out the help messages to stdout. Due to all the major functions are defined in bmp.c, so the main purpose of colorb.c is to call these functions defined in bmp.c and to open and read the files so it can be converted into the color-blind coloring. Here is brief pseudo code that shows the logic behind the main function:

```
int main(int argc, char *argv[]) {
    Check the number of command-line arguments {
        PrintErrorMessage;
        print_help();
        return EXIT FAILURE;
    }
    Validate input and output options{
        PrintErrorMessage;
```

```
        print_help();
        return EXITFAILURE;
    }
    Process file
}

Function to process input&output files
void process_file(const char *input_filename, const char *output_filename) {
    Open and validate the input file
    Create BMP structure from the input file
    Close the input file
    ReducePalette function called;
    Open&validate the output file
    Write the modified BMP data to the output file
    Close the output file
    Free BMP structure
}
```

Error Handling

There are a few errors that can occur, or flags that we have to look out for. If something doesn't run properly or execute correctly, an error message will be printed. More specifically if a file that hasn't been specified or provided is called, or an invalid command line option is input, the program will report an error message and stop executing. Now moving on to the bmpcreate function, there are certain order of verification steps that occur if any complications arise. It checks for type1, type2, bitmap header size, bits per pixel, and compression checks. If all of these are verified and checks properly, that means there are no errors.

Testing

For the testing of this program there are a few different things that needed to be done. There is a provided iotest.c file that contains a few tests that can check the io.c code, it checks for the implementation of the read tests. The program also processes images in a folder more specifically BMP images. Valgrind also needs to be run, to make sure there are no memory leaks.

Results

The program is working and executing successfully. The files were able to be opened, read, written, and a new image with the color-blind coloring was simulated. The functionality of all the header files and colorb.c file also passed successfully in pipeline, which is why the images were generated correctly. Initially, I was producing what looked like an identical image to what professor Veenstra produced, however there was a character difference in a few lines which was messing up the pipeline. This was fixed by changing my int(x) to a rounded x, which made the precision more accurate for the pictures. When I ran make format, make clean, and make along with the other make commands, it compiled successfully with no errors. I also ran valgrind, which produced no memory leaks, and no errors on my terminal, however pipeline is still saying valgrind is producing memory leaks. I attached a screenshot of the valgrind run that says there are no errors. Something that I learned from the previous assignment, and I implemented for this one was adding debugging print statements to see where there was a flaw in my logic. I also attached a few side by side screenshots of the original image and what my program produced.

```
asa@asarao:~/cse13s/asgn6$ valgrind ./cb.sh
==124351== Memcheck, a memory error detector
==124351== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==124351== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==124351== Command: ./cb.sh
==124351==
==124351==
==124351== HEAP SUMMARY:
==124351==     in use at exit: 141,383 bytes in 543 blocks
==124351==   total heap usage: 1,248 allocs, 705 frees, 207,121 bytes allocated
==124351==
==124351== LEAK SUMMARY:
==124351==    definitely lost: 0 bytes in 0 blocks
==124351==    indirectly lost: 0 bytes in 0 blocks
==124351==      possibly lost: 0 bytes in 0 blocks
==124351==    still reachable: 141,383 bytes in 543 blocks
==124351==         suppressed: 0 bytes in 0 blocks
==124351== Rerun with --leak-check=full to see details of leaked memory
==124351==
==124351== For lists of detected and suppressed errors, rerun with: -s
==124351== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figure 1: Valgrind Screenshot with No Memory Leaks

Figure 2: Apples BMP Comparison

Figure 3: Produce BMP Comparison