# Assignment 6 – Color Blindness Simulator

Johann Lee

CSE 13S – Fall 2023

## Purpose

The purpose of this project is to create a simulation of an image in the eyes of a color-blind person. By giving the program a bmp (bitmap image) file in the viewpoint of a person who does not have color blindness, we can run and create a new bmp that is within the viewpoint of a person who does have color blindness. Note that this program only mimics deutranopia which is the inability or reduced ability to distinguish between red and green. It does not mimic any other types of color blindness. By simulating these images into the perspective of a person with such disability it can help prompt product designers to choose colors that can be distinguished regardless of the way people view colors.

## How to Use the Program

Compiling this program requires the use of make commands while the required c files are present. Using "make" or "make all" will create two binaries called "colorb" and "iotest". "iotest" is meant to test whether io.c is properly working and is not necessary to touch when running "colorb". If needed then it can be ran using "./iotest" and will read out any errors within io.c. "colorb" is the binary that is used to change a bmp into a colorblind one. To get help on this program run either "./colorb" or "./colorb-h". It will print out a message that describes how to use the program. Regardless, instructions will also be written here. "colorb" requires two command line inputs, -i and -o, these being input file and output file respectively. Using these we can write "./colorb -i "inputfile.bmp" -o "outputfile.bmp"". Note that both these files must be bmp files as that is how the program functions and creates an image that can be seen. Now that an output.bmp file is created after running the code it must be opened in an environment that supports visuals. If this is present on a local machine with native visuals, located it within the folders and open the bmp file. If it is present on a virtual machine with no visual capabilities run the command "scp user@remote:"path"/"outputfile.bmp" /"path on local"". This will copy the bmp file onto your local machine from which it can be open and viewed.

## Program Design

### Functions and C files

This project contains a lot of given functions which will be rather briefly discussed and was lcoated in file, "asgn6.pdf by Dr. Kerry Veenstra", a resource provided to us.

- The first file is the io.c file which helped to read and write binary files.
  - void read_uint8(FILE *fin, uint8_t *px)
    - This program was provided to us on page 5 of asgn.pdf and is the base method of reading an 8 bit unsigned integer from a given file and assigned this uint8 to a pointer named px. It uses fgetc on the file saving it to result which is then checked to make sure it is not an EOF. If it is not then it is type casted into a uint8 and saved to px.
  - void read_uint16(FILE *fin, uint16_t *px)

* This does the same thing as the previous function but reads 16 bits instead. We can use the the previous function.

```
temp 1 and temp 2 both uint8

read_uint8(fin, temp1)
read_uint8(fin,  temp2)

px = temp2 << 8
px = px | temp1
```

- void read_uint32(FILE *fin, uint32_t *px)
  * Just as before it has the same function as 16 but reads 32 bits instead. We just need to copy our 16 bit read and change the numbers.

```
temp 1 and temp 2 uint16

read_uint16(fin, temp1)
read_uint16(fin, temp2)

px = temp2 << 16
px = px | temp1
```

- void write_uint8(FILE *fout, uint8_t x)
  * This program was provided to us on page 5 of asgn.pdf and is the base method of writing an 8 bit unsigned integer from an input uint8 x. This is done by using fputc() on x and our file.
- void write_uint16(FILE *fout, uint16_t *x)
  * This does the same thing as the previous function but writes 16 bits instead. We can use the the previous function and shift the second write by 8 bits.

```
write_uint8(fout, x)
write_uint8(fout, x >> 8)
```

- void write_uint32(FILE *fout, uint32_t *x)
  * Just as before it has the same function as 16 but reads 32 bits instead. We just have to change the numbers.

```
write_uint16(fout, x)
write_uint16(fout, x >> 16)
```

- This beings our next C file which is called bmp.c. This will use two given structs written just below that was provided by asgn6.pdf page 6. This helper function file will allow us to create our bmp and shifts colors.

```
#define MAX_COLORS 256

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;
```

  - Our first function uint32_t round_up(uint32_t x, uint32_t n) rounds the value of x up based on input n. This is done by using the modulus operator on x using n and checking it is not equal to 0. If this is true then x is increased by 1. We then return x.

- Our next function BMP *bmp_create(FILE *fin) is meant to create our bmp struct based on the input file given. This was given on asgn6.pdf page 7.
  This is done by first dynamically allocating a BMP and checking to make sure it is not null. We then run a series of reads and skips to saves to various variables using our previously created read functions. After we are done we should end with 8 variables with saved bits. These are type1, type2, bitmap_header_size, pbmp-¿width, pbmp-¿height, bits_per_pixel, compression, and colors_used. We then check to make sure type 1 is 'B', type 2 is 'M', the bitmap header is equal to 40, the bits per pixel is 8, compression is 0, and crate a new variable called num_colors and set it equal to colors_used.
  We now will use these variables to create our BMP. First is an if statement to check num_colors is equal to 0 and if it is true we shift 1 to the left by bits_per _pixel. After this we run a for statement in range of 0 to num_colors while reading pbmp-¿pallete.blue, green, red, and skip each comma separating a new read statement.
  We then have to ensure that each row has a multiple of 4 pixels so we use round_up on pbmp-¿width using 4 and save this to a variable named rounded_width. We then allocate a pixel array using calloc on pbmp-¿[a] and a for loop for 0 through width to create a calloc for every pbmp-¿a[x].
  We then read the pixels with a double for loop in range of 0 through pbmp-¿height and pbmp-¿width. Within this double for loop is a read into pbmp-¿a[x][y] to read the pixels into the pixel array from before. We also have to skip any extra pixels per row so a for loop is using in range of pbmp-¿width through rounded_width and skip an 8 bit. Finally we return our created pbmp.

- After creating our bmp now we must write our bmp using the function void bmp_write(const BMP *pbmp, FILE *fout). There are a lot of repeated lines of creating so the given pseudocode will be provided based on asgn6.pdf page 8.

```
        uint32_t rounded_width = round_up(pbmp->width, 4)
        uint32_t image_size = height * rounded_width;
        uint32_t file_header_size = 14
        uint32_t bitmap_header_size = 40
        uint32_t num_colors = MAX_COLORS
        uint32_t palette_size = 4 * num_colors
        uint32_t bitmap_offset = file_header_size +                    bitm
        uint32_t file_size = bitmap_offset + image_size
```

  Now we will write out these created variables in the same order as a bmp file just as we had read one using the bmp creation function from before.

```
        write uint8 'B'
        write uint8 'M'
        write uint32 file_size
        write uint16 0
        write uint16 0
        write uint32 bitmap_offset
        write uint32 bitmap_header_size
        write uint32 pbmp->width
```

```
        write uint32 pbmp->height
        write uint16 1
        write uint16 8
        write uint32 0
        write uint32 image_size
        write uint32 2835
        write uint32 2835
        write uint32 num_colors
        write uint32 num_colors
```

We must also write the pallette and pixels which we had read earlier using for loops just as we had read them. We also have to round out to multiples of 4 pixels per row by writing extra pixels.

```
        for y in range(0, pbmp->height):
            for x in range(0, pbmp->width):
                write uint8 pbmp->a[x][y]
            for x in range(pbmp->width, rounded_width):
                write uint8 0
```

- Since we created dynamic memory within our bmp function we must also free this memory which will be done using the function void bmp_free(BMP **ppbmp) provided on asgn6.pdf page 9. This function consists of a for loop going through ppbmp's width and freeing each of the rows. Then freeing ppbmp-¿a and finally ppbmp itself. We must also set ppbmp to null to ensure that it is fully freed.
- We also need a function uint8_t constrain(double x) which will be used shortly. This function will round x and check if x is less than 0. If it is then it will set x to 0. It will also check if x is greater than the max uint8 value and if it is it will set it to the max value. It then returns x type casted as an 8 bit unsigned integer.
- Our final function void bmp_reduce_palette(BMP *pbmp) is meant to adjust the color palette of the bitmap to actually mimic deuteranopia. Since the wording for the function is difficult to explain it will be written our first then explained.

```
    for i in range(0, MAX_COLORS):
        r = pbmp->palette[i].red;
        g = pbmp->palette[i].green;
        b = pbmp->palette[i].blue;

    SqLe = 0.00999 * r + 0.0664739 * g + 0.7317 * b
    SeLq = 0.153384 * r + 0.316624 * g + 0.057134 * b

    if SqLe < SeLq:
        r_new = constrain( 0.426331  * r + 0.875102  *
        g_new = constrain( 0.281100  * r + 0.571195  *
        b_new = constrain(-0.0177052 * r + 0.0270084 *
    else:
        r_new = constrain( 0.758100 * r + 1.45387 * g + -1.48060 *b)
        g_new = constrain( 0.118532 * r + 0.287595 * g + 0.725501 *b)
        b_new = constrain(-0.00746579 * r + 0.0448711 * g + 0.954303 *b)

    pbmp->palette[i].red   = r_new;
    pbmp->palette[i].green = g_new;
    pbmp->palette[i].blue  = b_new;
```

As seen first the individual rgb values are created as variables named r,g, and b from which they are given their respective palette colors. These values are then shifted in order to change their color values to mimic deuteranopia in SqLe and SeLq. We then make sure that rgb stays within certain parameters as to not go out of bounds using constrain() and reapply these new values to their respective palettes.

## Main Function

The main function is on the simpler side with the int main containing argv and argc in order to take in command line arguments. We then create file variables "*infile" an "*outfile". We then use getopt in order to create switch cases based on letters "hio" with "i" and "o" taking in parameters. Within the switch cases if it is 'h' we print the help message. If it is 'i' we set infile to optarg and check that it is not null. If it is null we print an error and exit. We do the same for 'o' but instead do it with outfile. Moving on from the switch cases we create a new BMP using our bmp_create and input our infile. We then reduce the palette of the bmp using the respective function and then use bmp_write in order to write our bmp to the outfile. Once we have this done we free our bmp and memory and return 0.