Assignment 6
Color Blindness Simulator

by Dr. Kerry Veenstra

CSE 13S, Fall 2023
Document version 3 (changes in Section 11)

Due Sunday Nov 19th, 2023, at 11:59 pm
Draft Due Friday Nov 17th, 2023, at 11:59 pm

1 Introduction

About 4% of people are *color blind*, meaning that they have some decreased ability to see differences in colors compared to other people. The most common type of color blindness is *red-green* color blindness, which refers to an inability or a reduced ability to distinguish between red and green. The most severe form of red-green color blindness is *deuteranopia*, which is caused by a complete lack of green-sensitive light sensors (cone cells) in one's eyes, leaving just the red-sensitive and blue-sensitive cone cells. Presented with two colors that differ only in their stimulation of green cone cells, someone with deuteranopia will be unable to detect any difference between the colors. Such a condition affects how someone interacts with the world around them.

If user-interface designers can be made aware of how their color choices affect people with color blindness, they can design their products to avoid color confusion for those with the most common types of color blindness. To understand which pairs of colors are affected by deuteranopia, in this assignment, you will write an image-processing program that allows someone with normal color vision to appreciate the range of colors experienced by someone who has deuteranopia. As part of this exercise, you will

- ☐ Read and write binary files (Section 2).
- ☐ Get practice in reading and writing the data of a binary file into and out of a C data structure—a process called "marshaling" or "serialization" (Section 3 and Section 4).
- ☐ Use a simple "shell script" to avoid needing to type complex commands (Section 5).

(a) Original photo.

(b) Simulating *deuteranopia*.

Figure 1: A comparison of a normal photo and a simulation of *deuteranopia*.

2 Binary File I/O

You've used `fopen(filename, "r")` to prepare to read a text file. The `"r"` parameter is the specified "mode", where `"r"` means to open the text file for reading. After opening the text file for reading, you can use various other I/O functions, such as `fscanf()` for reading formatted text data and `fgets()` for reading the next line of text data. Similarly, you've used `fopen(filename, "w")` to prepare to write a text file, along with other I/O functions, such as `fprintf()` and possibly `fputs()`.

2.1 C Language's Special Treatment of Text Files

Programs written in the C programming language may treat text data specially. That is to say, some operating systems store text data in memory differently than it is stored in files. For example, Microsoft Windows represents the end of a text line in memory as `'\n'` (a byte with a hexadecimal value of `0xa0`), but it represents the end of a text line in a file as `'\r' '\n'` (a two-byte sequence `0x0d 0x0a`). So when reading a text file under Windows, the `fgetc()` function reads the byte sequence `'\r' '\n'` from the text file and converts it into the single byte `'\n'`, which the function returns to represent end-of-line in-memory. Such treatment of text files lets you compile the same program *unchanged* under both Windows and Linux, representing the end of a text line in memory as `'\n'`, independent of the operating system's file representation.[1]

This special treatment of the characters of *text* files would interfere with reading and writing *binary* files. The next example talks about reading a binary executable program file, such as what an operating system loader or a debugger would do, but with text mode. Although you won't be writing parts of an operating system or a compiler suite in this class, consider this next example from the point of view of someone who is.

Look at the following x64 assembly language (this is just example binary data; you won't be writing assembly language in this class):

```
0:  04 0d                       add   al,0xd
2:  0a 04 25 78 56 34 12        or    al,BYTE PTR ds:0x12345678
```

The generated nine bytes of machine code (in hexadecimal) `04 0d 0a 04 25 78 56 34 12` contain the sequence `0d 0a` (which you may recognize as the Windows text-file sequence for end-of-line `'\r' '\n'` that we were just talking about). If we were to write a Windows program that reads a binary executable file after opening it with `fopen(filename, "r")`—using the *text* mode `"r"`—the program would treat the machine code as text and would convert it into the eight-byte sequence `04 0a 04 25 78 56 34 12`, which means completely different machine code:

```
0:  04 0a                       add   al,0xa
2:  04 25                       add   al,0x25
4:  78 56                       js    0x5c
6:  34 12                       xor   al,0x12
```

So when writing C programs that read and write files, we need some way to distinguish between *text* files, some of whose bytes may get converted, and *binary* files, whose bytes are read and written without change. The next section describes how to read and write binary files.

Since copy/paste from a PDF may be difficult, the resource file `asgn6-text.txt` contains text from the boxed code listings of this document.

[1] Another example of the difference between Windows and Linux text files is that under Windows the end of a file can be indicated by the byte `0x1a`, also known as Ctrl-Z. Linux text files do not consider `0x1a` bytes to be special.

2.2 Reading and Writing Binary Files

By default, the C programming language assumes that files are text. So when you write a program to read a *binary* file, you open it using mode `"rb"` instead of `"r"`:

```
/* Example C code for opening a binary file for reading. */

FILE *fin = fopen(filename, "rb");  // Use "rb" to open a binary file for reading.
if (fin == NULL) {
    /* can't open the input file; report a fatal error */
}
```

Similarly, when your program is preparing to *write* a `binary` file, use mode `"wb"` instead of `"w"`:

```
/* Example C code for opening a binary file for writing. */

FILE *fout = fopen(filename, "wb");  // Use "wb" to open a binary file for writing.
if (fout == NULL) {
    /* can't open the output file; report a fatal error */
}
```

These examples show comparing the resulting `FILE *` pointer with NULL. This is because `fopen()` returns NULL when there is an error.

Once a binary file has been opened for reading, it can be read one byte at a time using `fgetc()`. The `fgetc()` function will return each byte of the file as an `int` value between `0` and `255`. Once all bytes have been read, `fgetc()` will return EOF to mark the end of the file.

Here is an example of opening and reading a binary file. Notice that the return value from `fgetc()` is stored in a variable of type `int`. We don't store the value in a `uint8_t` because `fgetc()` might return EOF, whose value (usually -1) cannot be stored in a `uint8_t`. Also, notice that we call `fclose()` after we finish reading the file.

```
/* Example of code for reading a binary file. */

FILE *fin = fopen("filename", "rb");
if (fin == NULL) {
    /* TODO:  report fatal error */
}

while (1) {
    int ch = fgetc(fin);  // ch is an int because the value might be 0--255 or EOF
    if (ch == EOF) break;
    /* 0 <= ch <= 255 */
    /* TODO:  do something with ch. */
}

if (fclose(fin) == EOF) {
    /* TODO:  report fatal error */
}
```

Writing a binary file is similar. Once a binary file has been opened for writing, it can be written one byte at a time using `fputc()`. The `fputc()` function accepts the next byte for the file as a value between `0` and `255`. Since `fputc()` never accepts `EOF` as input, the value can be stored in a variable of type `uint8_t`.

Here is an example of opening and writing a buffer of size `file_size` into a binary file. If there is an error while writing a byte, then `fputc()` will return `EOF`. We must call `fclose()` after we finish writing the file.

```
/* Example of code for writing uint8_t buffer[] to a binary file. */

FILE *fout = fopen("filename", "wb");
if (fout == NULL) {
    /* TODO:  report fatal error */
}

for (int i = 0; i < file_size; ++i) {
    int result = fputc(buffer[i], fout);
    if (result == EOF) {
        /* TODO:  report fatal error */
    }
}

if (fclose(fout) == EOF) {
    /* TODO:  report fatal error */
}
```

3 Marshaling/Serialization

marshal 'mär-shəl
transitive verb
3. **to lead ceremoniously or solicitously, usher:**
Marshaling her little group of children down the street.

merriam-webster.com

Different computers can use different in-memory representations of the same type of data. In fact, the same computer, running a different compiler, may represent the same data types differently. For example, on different computers integers may have different sizes, floating-point numbers may use different internal representations, and even worse, pointers that contain the addresses of data values on one computer probably won't point to the same data values on another computer.

The classic example of a difference between memory representations is called "endianness." If a data value, such as an `int`, is represented in memory using more than one byte, and a computer needs to access individual bytes of the data value, then there are two choices. A computer that considers the least-significant byte of the `int` to have a lower address than the most-significant byte is called "little-endian." A computer that considers the most-significant byte of the `int` to have a lower address than the least-significant byte is called "big-endian"[1]. So if a program were to send its in-memory representation of a struct to a computer that uses a different endianness, almost surely the intended data values will not be communicated.

So although it is tempting allow computers to communicate quickly by sending each other big blocks of memory, instead, to ensure that the *meaning* of the bytes is the same on both computers, the sending computer "marshals" or "serializes" its memory data into an agreed-upon byte sequence that it sends to the other computer. Then the receiving computer "unmarshals" or "deserializes" the byte sequence into its own in-memory data format.

Such a byte sequence can be used for communication, as described above, but it also can be used to define the format of a binary file, such as an audio file or an image file, which is what you will do in this assignment.

3.1 Functions

The six read and write functions below transfer `uint8_t`, `uint16_t`, and `uint32_t` data values. When you look at the descriptions of the functions, you'll see that only the `read_uint8()` function and the `write_uint8()` function read from or write to files. The other read and write functions, for the larger data types, call the `read_uint8()` or `write_uint8()` functions in the proper order to "marshal" the data into or out of the file. For this assignment we will use the "little-endian" byte order, meaning that the first byte of a data value's byte sequence will contain the data value's least-significant bits. It is the responsibility of the functions below to use the little-endian byte order.

`void read_uint8(FILE *fin, uint8_t *px);`

Read a `uint8_t` data value (a byte) from open file `fin` by calling `fgetc()`, and assign the value to `*px` (the memory location pointed to by `px`). Report a fatal "unexpected end of file" error if `EOF` is returned. `fin` must contain the result of a successful call to `fopen(filename, "rb")`.

```
def read_uint8(fin, uint8_t *px):
    int result = fgetc(fin);
    if result == EOF:
        /* report a fatal error */
    *px = (uint8_t) result;
```

`void read_uint16(FILE *fin, uint16_t *px);`

To deserialize a `uint16_t`, call `read_uint8()` twice to read two bytes. Copy the second byte into a new `uint16_t` variable and shift the new variable to the left by 8 bits. Next, use the binary "or" (`|`) operation to "or" the first byte into the `uint16_t` variable. You've now "unmarshaled" or "deserialized" the `uint16_t` value from the byte sequence. Set `*px` to the resulting `uint16_t`.

`void read_uint32(FILE *fin, uint32_t *px);`

To deserialize a `uint32_t`, call `read_uint16()` twice to read two `uint16_t` values (four bytes). Copy the second `uint16_t` into a new `uint32_t` variable and shift the new variable to the left by 16 bits. Next, use the binary "or" (`|`) operation to "or" the first `uint16_t` into the `uint32_t` variable. You've now "unmarshaled" or "deserialized" the `uint32_t` value from the byte sequence. Set `*px` to the resulting `uint32_t`.

`void write_uint8(FILE *fout, uint8_t x);`

Write a `uint8_t` data value (a byte) to open file `fout` using `fputc()`. `fout` must contain the result of a successful call to `fopen(filename, "wb")`. If `fputc()` returns `EOF`, then report a fatal "unable to write file" error.

```
def write_uint8(fout, uint8_t x):
    int result = fputc(x, fout);
    if result == EOF:
        /* report a fatal error */
```

`void write_uint16(FILE *fout, uint16_t x);`

To serialize the `uint16 x`, call `write_uint8()` twice: first with `x`, and then with `x >> 8`.

`void write_uint32(FILE *fout, uint32_t x);`

To serialize the `uint32 x`, call `write_uint16()` twice: first with `x`, and then with `x >> 16`.

4 Reading/Writing Windows BMP Image Files

Images that use the Windows BMP format[2] can be read by both Windows computers and Macs. Version 3 of the format for Microsoft Windows 3.x is very easy to read and write, and so that's the version that we use. You will write functions to read and write Windows BMP files into and out of a BMP type:

```
    #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;
```

4.1 Functions

You will write functions to deserialize and serialize a BMP file: `bmp_create()` deserializes a existing BMP file, creating a BMP image in memory. `bmp_write()` serializes a BMP image that is in memory, creating a new BMP file. `bmp_free()` frees the BMP image in memory once you are finished with it. You also will write `bmp_reduce_palette()`, which changes the colors of a BMP image to simulate deuteranopia. And you will write a utility function that rounds a `uint32_t` up to the next multiple of some number.

`uint32_t round_up(uint32_t x, uint32_t n)`

BMP files always store a multiple of 4 pixels per row, even when the image's actual number of pixels per row is not a multiple of 4.[2] So when reading and writing BMP files, your algorithms need to round a `uint32_t` value up to the next multiple of 4. This utility function does that necessary rounding.

```
def round_up(x, n):
    while x % n != 0:
        x = x + 1
    return x
```

[2] The values of any extra pixels in each row are ignored. Since we can set any extra pixels to any value, we use 0.

BMP *bmp_create(FILE *fin)

Create a new BMP struct, read a BMP file into it, and return a pointer to the new struct. When told to skip data, just read the data into a variable whose value will be ignored.

```
def bmp_create(fin):
    BMP *pbmp = calloc(1, sizeof(BMP));
    report fatal error if pbmp == NULL

    // read data from the input file
    read uint8 type1
    read uint8 type2

    skip uint32 (that is, read a uint32 into a variable, but you won't use the value)
    skip uint16
    skip uint16
    skip uint32

    read uint32 bitmap_header_size
    read uint32 pbmp->width
    read uint32 pbmp->height

    skip uint16

    read uint16 bits_per_pixel
    read uint32 compression

    skip uint32
    skip uint32
    skip uint32

    read uint32 colors_used

    skip uint32

    verify type1 == 'B'
    verify type2 == 'M'
    verify bitmap_header_size == 40
    verify bits_per_pixel == 8
    verify compression == 0
    uint32_t num_colors = colors_used
    if (num_colors == 0) num_colors = (1 << bits_per_pixel)
    for i in range(0, num_colors):
        // read bytes from the input file
        read uint8 pbmp->pallete[i].blue
        read uint8 pbmp->pallete[i].green
        read uint8 pbmp->pallete[i].red
        skip uint8

    // Each row must have a multiple of 4 pixels.  Round up to next multiple of 4.
    uint32_t rounded_width = round_up(pbmp->width, 4)

    // Allocate pixel array
    pbmp->a = calloc(width, sizeof(pbmp->a[0]));
    for x in range(0, width):
        pbmp->a[x] = calloc(pbmp->height, sizeof(pbmp->a[x][0]));

    // read pixels
    for y in range(0, pbmp->height):
        for x in range(0, pbmp->width):
            read uint8 pbmp->a[x][y]

        // skip any extra pixels per row
        for x in range(pbmp->width, rounded_width):
            skip uint8

    return pbmp;
```

void bmp_write(const BMP *pbmp, FILE *fout)

Write a BMP image from memory to a file.

```
def bmp_write(pbmp, fout):
    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 + bitmap_header_size + palette_size
    uint32_t file_size = bitmap_offset + image_size

    // write data to output file
    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

    // write the palette
    for i in range(0, num_colors):
        write uint8 pbmp->palette[i].blue
        write uint8 pbmp->palette[i].green
        write uint8 pbmp->palette[i].red
        write uint8 0

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

        // if needed, write extra pixels to make a multiple of 4 pixels per row
        for x in range(pbmp->width, rounded_width):
            write uint8 0
```

void bmp_free(BMP **ppbmp)

```
    for i in range(0, (*ppbmp)->width):
        free((*ppbmp)->a[i]);
    free((*ppbmp)->a);
    free(*ppbmp);
    *ppbmp = NULL;
```

uint8_t constrain(double x);

Round a `double` value to an integer, and then constrain it to a range the fits in a `uint8_t`.

```
def constrain(x):
    x = round(x)
    if x < 0:
        x = 0
    if x > UINT8_MAX:
        x = UINT8_MAX
    return (uint8_t) x
```

void bmp_reduce_palette(BMP *pbmp);

Adjust the color palette of a bitmap image to simulate deuteranopia using the C code below. (See Appendix A if you are interested in the source of the equations.)

```
def bmp_reduce_palette(pbmp):
    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 +  0.0801271 * b)
            g_new = constrain( 0.281100  * r + 0.571195  * g + -0.0392627 * b)
            b_new = constrain(-0.0177052 * r + 0.0270084 * g +  1.00247   * b)
        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;
```

5 Using Simple Shell Scripts

We've provided you with a number of test images in the `bmps` directory. You can run `colorb` on all of them manually using commands like this:

```
$ ./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 it is easier to use a "shell script."

We've provided a script called `cb.sh` that looks in the `bmps` directory and runs `colorb` on all BMP files whose name ends in `-orig.bmp`. Each corresponding output file has the same base but ends in `-colorb.bmp`.

```
$ ./cb.sh
```

`cb.sh` is a text file. You can edit it to see what it does.

You may want to run `colorb` on your own files, but if you don't have BMP files, no worries! You can use a program called `convert` from the ImageMagick software suite to convert images to and from the BMP format. You can install the ImageMagick software suite on your Ubuntu VM using this command:

```
sudo apt install imagemagick
```

Then the following command converts from nearly any other image format into the BMP format that `colorb` requires. Change the input filename `file-orig.gif` into whatever file you have (any base name and any extension, such as `.gif`, `.png`, `.jpg`, etc.). You can change the output filename, too, except that it must end in `.bmp`. The `BMP3:` file-version prefix is required, too.

```
$ convert file-orig.gif -colors 256 -alpha off -compress none BMP3:file-orig.bmp
```

To convert back into a `.gif` (or whatever), use a command like this:

```
$ convert file-colorb.bmp file-colorb.gif
```

6 Command line options

Your program must support these command-line options. `-i` and `-o` must be specified on the command line.

- `-i` : Sets the name of the input file. Requires a filename as an argument.
- `-o` : Sets the name of the output file. Requires a filename as an argument.
- `-h` : Prints a help message to `stdout`.

7 Program Output and Error Handling

If any invalid options or files are specified, your program should report an error and exit cleanly. Your `bmp_create()` function should include the listed verification steps, reporting an error if one of them fails, but other than those checks, your code can assume that an input BMP file is valid.

8 Testing your code

To get you started on testing your code, we have provided you `iotest.c`. This runs a series of tests on your `io.c` code. The current implementation only tests *reads*; it works by writing a bunch of known data to a test file, and then using your code to read the same file and making sure the results are the same as what it wrote to the file. While not required, to test your code more thoroughly you can add write tests (i.e. write using your code, then read using `read()` and make sure it matches) to that same file.

- You will receive a folder of BMP images. Your program should be able to successfully process these images.
- Your program should have no *memory leaks*. Make sure you `free()` before exiting. `valgrind` should pass cleanly with any combination of the specified command-line options, including on an error condition. Note that `valgrind` does report errors other than memory leaks, such as invalid reads/writes and use of uninitialized data. These errors are generally *worse* than leaks and you should fix them as well.

9 Submission

For the **report draft**, you must submit a commit ID on canvas *before Friday Nov 17th at 11:59 Pacific Time*. Your draft report must be a PDF named `report.pdf`.

For the entire project, you must submit a commit ID on canvas *before Sunday Nov 19th at 11:59 pm Pacific Time*.

Your submission must have these files. You must have run `clang-format` on the `.c` and `.h` files.

- `bmp.c` – contains your BMP functions (Section 4.1)
- `bmp.h` — provided
- `colorb.c` — contains your `main()` function
- `io.c` — contains your serialization/deserialization functions (Section 3.1)
- `io.h` — provided
- `iotest.c` — provided, but you can (and should!) add more test cases.
- `Makefile` — your Makefile
  - The compiler must be `clang`, and the compiler flags must include `-Werror -Wall -Wextra -Wconversion -Wdouble-promotion -Wstrict-prototypes -pedantic`.
  - `make` and `make all` must build both `colorb` and `iotest`.
  - `make colorb` must build your program.
  - `make iotest` must build the I/O test program using `iotest.c`.
  - `make format` must format all C and header files using `clang-format`.
  - `make clean` must delete all object files and compiled programs.

10 Supplemental Readings

- *The C Programming Language* by Kernighan & Ritchie. Section 7.5 File Access. Section B1 Input and Output <stdio.h>. Section B1.1 File Operations.

11 Revisions

Version 1 Original.

Version 2 Suggest using copy/paste with `asgn6-text.txt`. Box and number the code listings of the functions `read_uint8()` and `write_uint8()`. Number the `bmp_reduce_palette()` listing. Add section numbers to Supplemental Readings.

Version 3 In `bmp_reduce_palette()` replace `new_r` with `r_new`, `new_g` with `g_new`, and `new_b` with `b_new`. In `bmp_write()` replace seven instances of `int32_t` with `uint32_t`.

References

[1] Danny Cohen. On holy wars and a plea for peace. *Internet Experiment Note*, (137), April 1980. URL: `https://www.rfc-editor.org/ien/ien137.txt`.

[2] FileFormat.info. Microsoft windows bitmap file format summary. `https://www.fileformat.info/format/bmp/egff.htm`.

[3] Hans Brettel, Françoise Viénot, and John D. Mollon. Computerized simulation of color appearance for dichromats. *Journal of the Optical Society of America A*, 14(10):2647–2655, October 1997.

Appendix A: Color Transformations

This appendix documents the equations that are in the `bmp_reduce_palette()` function of Section 4. Read it if you are interested.

A.1 Background

Brettel et al.[3] teach how to convert an RGB (red/green/blue) representation of a color into a *new* RGB color that lets people with normal color vision experience the "color confusion" that someone with color blindness can experience. Their approach is to divide the conversion into three steps: (1) transform the RGB color into the more physiologically relevant LMS color space, which models the color response of the three kinds of cone cells of the human eye, (2) manipulate the LMS color to simulate the unique cone cells of someone who has color blindness, and (3) transform the manipulated LMS color back into an RGB value.

These three steps can be combined into a single matrix multiplication of an RGB color value:

$$\mathbf{V}' = \mathbf{R}\mathbf{V}$$

where

$$\mathbf{V}' = \begin{pmatrix} R_{V'} \\ G_{V'} \\ B_{V'} \end{pmatrix} = \text{converted color} \qquad \mathbf{V} = \begin{pmatrix} R_V \\ G_V \\ B_V \end{pmatrix} = \text{original color}$$

and $\mathbf{R}$ is the desired transformation matrix.

A.2 Deriving the Matrix R

Equations from the reference define colors in the LMS color space ($\mathbf{Q}$ and $\mathbf{Q}'$) along with conversions to and from the colors in the RGB color space ($\mathbf{V}$ and $\mathbf{V}'$).

$$\mathbf{Q}' = \begin{pmatrix} L'_Q \\ M'_Q \\ S'_Q \end{pmatrix} = \text{ converted color in LMS space} \qquad \mathbf{Q} = \begin{pmatrix} L_Q \\ M_Q \\ S_Q \end{pmatrix} = \text{ orginal color in LMS space} \tag{3}$$

$$\mathbf{Q}' = \mathbf{T}\mathbf{V}' \qquad \mathbf{Q} = \mathbf{T}\mathbf{V} \tag{4}$$

$$\mathbf{V}' = \mathbf{T}^{-1}\mathbf{Q}' \qquad \mathbf{V} = \mathbf{T}^{-1}\mathbf{Q} \tag{5}$$

(The equation numbers used in this section match those of the reference.)

Table 1 from the reference provides the values of $\mathbf{T}$:

$$\mathbf{T} = \begin{bmatrix} 0.1992 & 0.4112 & 0.0742 \\ 0.0353 & 0.2226 & 0.0574 \\ 0.0185 & 0.1231 & 1.3550 \end{bmatrix}$$

At this point we merely lack a conversion in LMS color space from $\mathbf{Q}$ into $\mathbf{Q}'$. The reference uses a matrix multiplication:

$$\mathbf{Q}' = \mathbf{C}\mathbf{Q}$$

Combine equations from above into a conversion from $\mathbf{V}$ into $\mathbf{V}'$

$$\mathbf{V}' = \mathbf{T}^{-1}(\mathbf{C}(\mathbf{T}\mathbf{V}))$$

apply the associative property of matrix multiplication

$$\mathbf{V}' = (\mathbf{T}^{-1}\mathbf{C}\mathbf{T})\mathbf{V}$$

and recall that we are looking for the $\mathbf{R}$ of the original equation $\mathbf{V}' = \mathbf{R}\mathbf{V}$, which reveals

$$\mathbf{R} = \mathbf{T}^{-1}\mathbf{C}\mathbf{T}$$

A.3 Deriving C

The variable $\mathbf{C}$ of the LMS multiplicative transformation $\mathbf{Q}' = \mathbf{CQ}$ is defined by the equations:

$$\begin{aligned} L_{Q'} &= L_Q \\ M_{Q'} &= -(aL_Q + cS_Q)/b \\ S_{Q'} &= S_Q \end{aligned} \tag{10}$$

or in matrix form

$$\begin{pmatrix} L_{Q'} \\ M_{Q'} \\ S_{Q'} \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ -\dfrac{a}{b} & 0 & -\dfrac{c}{b} \\ 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} L_Q \\ M_Q \\ S_Q \end{pmatrix}$$

so

$$\mathbf{C} = \begin{bmatrix} 1 & 0 & 0 \\ -\dfrac{a}{b} & 0 & -\dfrac{c}{b} \\ 0 & 0 & 1 \end{bmatrix}$$

The values of a, b, and c come from a cross product of two vectors $\mathbf{E} = (L_E, M_E, S_E)$ and $\mathbf{A} = (L_A, M_A, S_A)$.

$$\begin{aligned} a &= M_E S_A - S_E M_A \\ b &= S_E L_A - L_E S_A \\ c &= L_E M_A - M_E L_A \end{aligned} \tag{8}$$

To obtain the values of $\mathbf{E}$ and $\mathbf{A}$, I needed to physically measure the plots in Figures 2(a) and 2(b) of the reference. Those measurements result in

$$\mathbf{E} = \begin{bmatrix} L_E \\ M_E \\ S_E \end{bmatrix} = \begin{bmatrix} 0.54 \\ 0.22 \\ 0.77 \end{bmatrix}$$

$$\mathbf{A}_{575} = \begin{bmatrix} L_A \\ M_A \\ S_A \end{bmatrix} = \begin{bmatrix} 0.47 \\ 0.18 \\ 0 \end{bmatrix}$$

$$\mathbf{A}_{475} = \begin{bmatrix} L_A \\ M_A \\ S_A \end{bmatrix} = \begin{bmatrix} 0 \\ 0.151 \\ 1.41 \end{bmatrix}$$

So depending on whether one uses $\mathbf{A}_{575}$ or $\mathbf{A}_{475}$, the value of $\mathbf{C}$ is either

$$\mathbf{C}_{575} = \begin{bmatrix} 1 & 0 & 0 \\ 0.382978 & 0 & 0.0171318 \\ 0 & 0 & 1 \end{bmatrix}$$

or

$$\mathbf{C}_{475} = \begin{bmatrix} 1 & 0 & 0 \\ 0.254701 & 0 & 0.107092 \\ 0 & 0 & 1 \end{bmatrix}$$

A.4 Converting RGB Colors

The reference teaches that if $S_Q/L_Q < S_E/L_E$ then we use $\mathbf{A}_{575}$. Otherwise, we use $\mathbf{A}_{475}$. To avoid division by zero, we multiply both sides of the inequality by $L_Q L_E$, and then the inequality check becomes $S_Q L_E < S_E L_Q$.

Recalling that $\mathbf{Q} = \mathbf{TV}$, or

$$\begin{aligned} L_Q &= 0.1992\ R_V + 0.4112\ G_V + 0.0742\ B_V \\ M_Q &= 0.0353\ R_V + 0.2226\ G_V + 0.0574\ B_V \\ S_Q &= 0.0185\ R_V + 0.1231\ G_V + 1.3550\ B_V \end{aligned}$$

and using $L_E = 0.54$ and $S_E = 0.77$ from above, we get

$$\boxed{S_Q L_E = 0.00999\ R_V + 0.0664739\ G_V + 0.7317\ B_V}$$

$$\boxed{S_E L_Q = 0.153384\ R_V + 0.316624\ G_V + 0.057134\ B_V}$$

If $S_Q L_E < S_E L_Q$ then

$$\mathbf{R} = \mathbf{T}^{-1}\mathbf{C}_{575}\mathbf{T} = \begin{bmatrix} 0.426331 & 0.875102 & 0.0801271 \\ 0.281100 & 0.571195 & -0.0392627 \\ -0.0177052 & 0.0270084 & 1.00247 \end{bmatrix}$$

so

$$\boxed{R_{V'} = 0.426331\ R_V + 0.875102\ G_V + 0.0801271\ B_V}$$

$$\boxed{G_{V'} = 0.2811\ R_V + 0.571195\ G_V - 0.0392627\ B_V}$$

$$\boxed{B_{V'} = -0.0177052\ R_V + 0.0270084\ G_V + 1.00247\ B_V}$$

Otherwise $S_Q L_E \geq S_E L_Q$, and then

$$\mathbf{R} = \mathbf{T}^{-1}\mathbf{C}_{475}\mathbf{T} = \begin{bmatrix} 0.758100 & 1.45387 & -1.48060 \\ 0.118532 & 0.287595 & 0.725501 \\ -0.00746579 & 0.0448711 & 0.954303 \end{bmatrix}$$

so

$$\boxed{R_{V'} = 0.7581\ R_V + 1.45387\ G_V - 1.4806\ B_V}$$

$$\boxed{G_{V'} = 0.118532\ R_V + 0.287595\ G_V + 0.725501\ B_V}$$

$$\boxed{B_{V'} = -0.00746579\ R_V + 0.0448711\ G_V + 0.954303\ B_V}$$