Assignment 5
Color Blindness Simulator

by Dr. Kerry Veenstra
edits by TBD
CSE 13S, Spring 2023
Document version 1 (changes in Section 11)

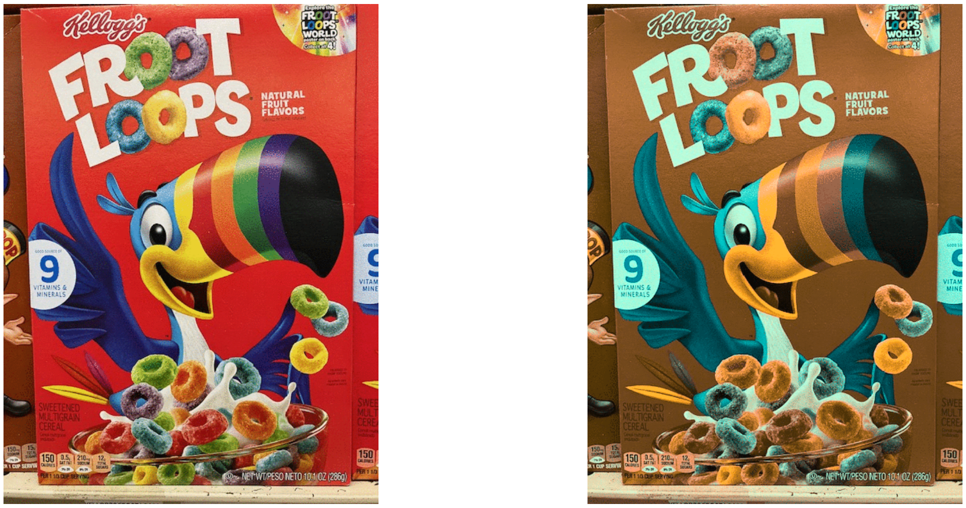
Due Sunday May 28th, 2023, at 11:59 pm
Draft Due Wednesday May 24th, 2023, at 11:59 pm

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

- ☐ Use "unbuffered file-I/O" functions to read and write binary files (Section 2).



(a) Original photo.

(b) Simulating *deuteranopia*.

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

□ 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).

2 Unbuffered File I/O

> I'd spell `creat` with an `e`.
>
> *Ken Thompson, when asked what he would do differently if he were redesigning the UNIX operating system[1].*

The file-I/O functions that you've been using, like `fopen()`, `fprintf()`, `fscanf()`, and `fclose()`, provide convenience and a wealth of features, but they also have overhead. Sometimes, when writing high-performance programs, we want to minimize that overhead. So instead of using the "FILE *" functions, we use Unix *system calls* to access files using an interface of our own design.

The Unix file-I/O functions include `open()`, `read()`, `creat()`, `write()`, and `close()`. One difference is that instead of a `FILE *`, the Unix file-I/O functions use an `int` file descriptor. Another difference is that these Unix file-I/O functions read and write raw bytes of data (`uint8_ts`). There is no formatting! You specify the number of bytes that you would like to transfer into or out of a file, and the `read()` or `write()` function returns the number of bytes that it actually transferred (or an error code).

Take a look at the manual page for `write()`: in Ubuntu enter this command:

```
$ man 2 write
```

(We specify chapter "2" because there's also a `write` command in chapter 1 that does not interest us.) From the manual page for the `write()` system call you learn:

- To use the function you need to
  `#include <unistd.h>`
- The function prototype is
  `ssize_t write(int fd, const void *buf, size_t count);`
- The return value is the number of bytes actually written or `-1` to indicate an error (such as a full storage disk).

`fd` is a file descriptor (described next), `buf` points to a byte buffer, and `count` is the number of bytes from the buffer to write to the file (although the return value may be less than that, if the function chooses not to write that many bytes).

To get a file descriptor, you use either the `open()` system call (for reading a file) or the `creat()` system call (for writing a file). The details regarding the use of these system calls are covered in lecture.

We say that the system calls support *unbuffered* file I/O, but your functions will provide a *bit* of buffering—just not the complexity that the 216-byte FILE structure supports. Instead of a pointer to `FILE`, your file-access functions will use a pointer to `Buffer`, a type that is defined below.

The structure has fields that let access functions convert between large blocks of data and individual bytes. The functions of this section manage the buffers, and the functions of the Section 3 manage the individual bytes. We mention here what the structure's fields are for.

The `a[]` field is the actual buffer: it is a place for up to `BUFFER_SIZE` bytes. The purpose of the `offset` and `num_remaining` fields depends on whether this is a read buffer or a write buffer. When reading from a file, after a block of data is read into `a[]`, the `offset` field contains the index of the next byte to take from the buffer, and the `num_remaining` field stores the number of bytes remaining in `a[]` until the buffer needs to be refilled. When writing to a file, bytes are stored sequentially in `a[]`, the `offset` field contains the index of the next free location to place a byte into the buffer, and the `num_remaining` field is unused. The `fd` field holds the file descriptor that will be used by either `read()` or `write()`.

```
/* put in io.h */
typedef struct buffer Buffer;

/* put in io.c */
struct buffer {
    int fd;                   // file descriptor from open() or creat()
    int offset;               // offset into buffer a[]
                              //     next valid byte (reading)
                              //     next empty location (writing)
    int num_remaining;        // number of bytes remaining in buffer (reading)
    uint8_t a[BUFFER_SIZE];   // buffer
};
```

2.1 Functions

The `read_open()` and `read_close()` functions allocate and free a `Buffer` that can be used for reading from a file. The `write_open()` and `write_close()` functions allocate and free a `Buffer` that can be used for writing to a file. (See Section 3.1 for the related functions that read and write data.)

`Buffer *read_open(const char *filename);`

Open the file `filename` using the `open(filename, O_RDONLY)` system call. If unsuccessful (the return value < 0) then return `NULL`. Otherwise `malloc()` or `calloc()` a new `Buffer`, set the `fd` field to the return value from `open()`, and set the rest of the fields to `0`. Return the pointer to the new `Buffer`. Whenever your program calls `read_open()`, it will need to call a corresponding `read_close()` later.

`void read_close(Buffer **pbuf);`

Call `close((*pbuf)->fd)` to close the file. Free the Buffer. Set `*pbuf = NULL`.

`Buffer *write_open(const char *filename);`

Open the file `filename` using the `creat(filename, 0664);` system call. If unsuccessful (the return value < 0) then return `NULL`. Otherwise `malloc()` or `calloc()` a new `Buffer`, set the `fd` field to the return value from `creat()`, and set the rest of the fields to `0`. Return the pointer to the new `Buffer`. Whenever your program calls `write_open()`, it will need to call a corresponding `write_close()` later.

`void write_close(Buffer **pbuf);`

Write any accumulated bytes that are in the buffer `a[]` to the file indicated by `(*pbuf)->fd`. (See Section 3.) Then call `close((*pbuf)->fd)` to close the file. Free the Buffer. Set `*pbuf = NULL`.

3 Marshaling/Serialization

> **marshal** ˈmär-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"[2]. 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, 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 through a `Buffer` that already has been initialized with `read_open()` or `write_open()`. Each call to a read function retrieves data from a read `Buffer`; each call to a write function stores data to a write `Buffer`. Since at some point the read buffer will become empty (or the write buffer will become full), when necessary, the functions below also will refill the read buffer from a file (or write the contents of a write buffer to a file).

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.

`bool read_uint8(Buffer *buf, uint8_t *x);`

If the buffer is empty (`buf->num_remaining == 0`), then refill it from an open file using `buf->fd`.

```
ssize_t rc = read(buf->fd, buf->a, sizeof(buf->a));
if (rc < 0) TODO report error
if (rc == 0) return false;  // end of file
buf->num_remaining = rc;
buf->offset = 0;
```

Then set `*x` to the next byte in the buffer, increment `buf->offset`, and return `true`.

`bool read_uint16(Buffer *buf, uint16_t *x);`

To deserialize a `uint16_t`, call `read_uint8()` twice to read two bytes. If either call returns `false`, we've gotten to the end of the file, and so this function returns `false` as well. Otherwise, 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 `*x` to the resulting `uint16_t` and return `true`.

bool read_uint32(Buffer *buf, uint32_t *x);

To deserialize a `uint32_t`, Call `read_uint16()` twice to read two `uint16_t` values (four bytes). If either call returns `false`, we've gotten to the end of the file, and so this function returns `false` as well. Otherwise, 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 `*x` to the resulting `uint32_t` and return `true`.

void write_uint8(Buffer *buf, uint8_t x);

If the buffer is full (`buf->offset == BUFFER_SIZE`), then call `write()` as many times as necessary to empty it. (`write()` may not write *all* bytes of the buffer, and so you need to check its return value and call it in a loop.)

```
uint8_t *start = buf->a;
int num_bytes = buf->offset;

do {
    ssize_t rc = write(buf->fd, start, num_bytes);
    if (rc < 0) TODO report error
    start += rc;        // skip past the bytes that were just written
    num_bytes -= rc;    // how many bytes are left?
} while (num_bytes > 0);

buf->offset = 0;
```

Then set `*x` to the next byte in the buffer, increment `buf->offset`, and return `true`.

void write_uint16(Buffer *buf, uint16_t x);

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

void write_uint32(Buffer *buf, 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[3] 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:

```
typedef struct color {
    uint8_t red;
    uint8_t green;
    uint8_t blue;
} Color;

typedef struct bmp {
    uint16_t    height;
    uint16_t    width;
    Color       palette[MAX_COLORS];
    uint8_t   **a;
} BMP;
```

4.1 Functions

You will write functions to serialize and deserialize a BMP file: `bmp_write()` serializes a BMP image and creates a new BMP file. `bmp_create()` deserializes a BMP file, creating a BMP in memory. `bmp_free()` frees the BMP once you are finished with it.

We provide you with `bmp_adjust_palette()`, which changes the colors of a BMP image to simulate deuteranopia.

`void bmp_write(const BMP *bmp, Buffer *buf)`

Write a BMP file. The pseudocode below represents data being written to the file as rows of a table. The left column indicates the number of bits to write, using `write_uint8()`, `write_uint16()`, or `write_uint32()`. The right column gives the value to write.

BMP File Format: Writing

```
int32_t rounded_width = (width + 3) & ~3;
int32_t image_size = height * rounded_width;
int32_t file_header_size = 14
int32_t bitmap_header_size = 40
int32_t num_colors = 256
int32_t palette_size = 4 * num_colors
int32_t bitmap_offset = file_header_size + bitmap_header_size + palette_size
int32_t file_size = bitmap_offset + image_size
```

8	`'B'`
8	`'M'`
32	`file_size`
16	`0`
16	`0`
32	`bitmap_offset`
32	`bitmap_header_size`
32	`bmp->width`
32	`bmp->height`
16	`1`
16	`8`
32	`0`
32	`image_size`
32	`2835`
32	`2835`
32	`num_colors`
32	`num_colors`

```
for i from 0 to num_colors - 1
```

8	`bmp->palette[i].blue`
8	`bmp->palette[i].green`
8	`bmp->palette[i].red`
8	*(skip one byte)*

```
for y from 0 to bmp->height - 1
    for x from 0 to bmp->width - 1
```

8	`bmp->a[x][y]`

```
    for x from bmp->width to rounded_width - 1
```

8	`0`

BMP *bmp_create(Buffer *buf)

Create a new BMP struct, read a BMP file into it, and return a pointer to the new struct. The pseudocode below represents data being read from the file as rows of a table. The left column indicates the number of bits to read, using `read_uint8()`, `read_uint16()`, or `read_uint32()`. The right column indicates which variable should receive the value. When the table indicates that data should be skipped, just read the data into a variable whose value will be ignored.

BMP File Format: Reading

```
BMP *bmp = calloc(1, sizeof(BMP));
// TODO check for bmp == NULL
```

8	`uint8_t type1`
8	`uint8_t type2`
32	*(skip four bytes)*
16	*(skip two bytes)*
16	*(skip two bytes)*
32	*(skip four bytes)*
32	`uint32_t bitmap_header_size`
32	`bmp->width`
32	`bmp->height`
16	*(skip two bytes)*
16	`uint16_t bits_per_pixel`
32	`uint32_t compression`
32	*(skip four bytes)*
32	*(skip four bytes)*
32	*(skip four bytes)*
32	`uint32_t colors_used`
32	*(skip four bytes)*

```
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 from 0 to num_colors - 1
```

8	`bmp->palette[i].blue`
8	`bmp->palette[i].green`
8	`bmp->palette[i].red`
8	*(skip one byte)*

```
// Each row must have a multiple of 4 pixels.  Round up to next multiple of 4.
uint32_t rounded_width = (bmp->width + 3) & ~3

// Allocate pixel array
bmp->a = calloc(rounded_width, sizeof(bmp->a[0]));
for x from 0 to rounded_width - 1
    bmp->a[x] = calloc(bmp->height, sizeof(bmp->a[x][0]));

// read pixels
for y from 0 to bmp->height - 1
    for x from 0 to rounded_width - 1
```

8	`bmp->a[x][y]`

```
return bmp;
```

void bmp_free(BMP **bmp)

```
    uint32_t rounded_width = ((*bmp)->width + 3) & ~3
    for i from 0 to rounded_width - 1
        free((*bmp)->a[i])
    free((*bmp)->a);
    free(*bmp);
    *bmp = NULL;
```

void bmp_reduce_palette(BMP *bmp);

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.)

```
int constrain(int x, int a, int b) {
    return x < a ? a :
           x > b ? b : x;
}

void bmp_reduce_palette(BMP *bmp) {
    for (int i = 0; i < MAX_COLORS; ++i)
    {
        int r = bmp->palette[i].red;
        int g = bmp->palette[i].green;
        int b = bmp->palette[i].blue;

        int new_r, new_g, new_b;

        double SQLE = 0.00999  * r + 0.0664739 * g + 0.7317   * b;
        double SELQ = 0.153384 * r + 0.316624  * g + 0.057134 * b;

        if (SQLE < SELQ) {
            // use 575-nm equations
            new_r =  0.426331  * r + 0.875102  * g +  0.0801271 * b + 0.5;
            new_g =  0.281100  * r + 0.571195  * g + -0.0392627 * b + 0.5;
            new_b = -0.0177052 * r + 0.0270084 * g +  1.00247   * b + 0.5;
        } else {
            // use 475-nm equations
            new_r =  0.758100   * r + 1.45387   * g + -1.48060  * b + 0.5;
            new_g =  0.118532   * r + 0.287595  * g +  0.725501 * b + 0.5;
            new_b = -0.00746579 * r + 0.0448711 * g +  0.954303 * b + 0.5;
        }

        new_r = constrain(new_r, 0, UINT8_MAX);
        new_g = constrain(new_g, 0, UINT8_MAX);
        new_b = constrain(new_b, 0, UINT8_MAX);

        bmp->palette[i].red     = new_r;
        bmp->palette[i].green   = new_g;
        bmp->palette[i].blue    = new_b;
    }
}
```

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 should support these command-line options. -i and -o are required.

- -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 `io_test.c`.

- 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.
- Your program must pass the static analyzer, `scan-build`,

9 Submission

For the **report draft**, you must submit a commit ID on canvas *before Wednesday May 24th at 11:59 Pacific Time.* You must have a PDF called `report.pdf`.

For the entire project, you must submit a commit ID on canvas *before Sunday May 28th 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
- `Makefile` — your Makefile

10 Supplemental Readings

- *The C Programming Language* by Kernighan & Ritchie

11 Revisions

Version 1 Original.

References

[1] Brian W. Kernighan and Rob Pike. *The UNIX Operating Environment*, page 204. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1984.

[2] 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`.

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

[4] 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.[4] 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:

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

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 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 $\mathbf{V'}$).

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

$$\mathbf{Q'} = \mathbf{TV'} \qquad \mathbf{Q} = \mathbf{TV} \tag{4}$$

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

(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:

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

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

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

apply the associative property of matrix multiplication

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

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

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

A.3 Deriving C

The variable $\mathbf{C}$ of the LMS multiplicative transformation $\mathbf{Q}' = \mathbf{C}\mathbf{Q}$ 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

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

so

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

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

$$\boxed{r_{\text{new}} = 0.426331\ r + 0.875102\ g + 0.0801271\ b}$$

$$\boxed{g_{\text{new}} = 0.2811\ r + 0.571195\ g - 0.0392627\ b}$$

$$\boxed{b_{\text{new}} = -0.0177052\ r + 0.0270084\ g + 1.00247\ b}$$

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

$$\boxed{r_{\text{new}} = 0.7581\ r + 1.45387\ g - 1.4806\ b}$$

$$\boxed{g_{\text{new}} = 0.118532\ r + 0.287595\ g + 0.725501\ b}$$

$$\boxed{b_{\text{new}} = -0.00746579\ r + 0.0448711\ g + 0.954303\ b}$$