Chapter 1

Bitmaps

Let's talk about another image format: the Bitmap. The Bitmap, denoted with file type `.bmp`, is an early file format for storing images on computers. Rows of different colors are stored as a grid of pixels. Each pixel is represented by a specific number of bits, which determines the color depth of the image. For example, a 24-bit bitmap can represent over 16 million colors, while an 8-bit bitmap can only represent 256 colors.

Bitmaps are uncompressed, meaning they can take up a lot of storage space compared to other image formats like JPEG or PNG. However, they are simple to read and write in code, making them useful for certain applications where image quality is more important than file size.

To understand how bitmaps work in practice, it helps to think of an image as nothing more than numbers stored in memory. Each pixel in a bitmap corresponds to a color value, and these values are usually written in hexadecimal (base-16) form. If you need a review, look back at Chapter **??**

In a common 24-bit bitmap, each pixel is made up of three color components:

- Red
- Green
- Blue

The intensity of each color component is represented by 8 bits. In decimal, this means each component can have a value from `0` to 255, for a total of 256 possible values. In hexadecimal, this range is represented from `00` to `FF`.

Each pixel's color is then represented by a 6-digit hexadecimal number, where the first two digits represent the red component, the next two represent green, and the last two represent blue. A total of $16,777,216$ different colors can be represented $(256 \times 256 \times 256)$.

Together, the intensity of each component determines the final color of the pixel. For example:

- `0xFF0000` represents pure red
- `0x00FF00` represents pure green
- `0x0000FF` represents pure blue

- `0xFFFFFF` represents white
- `0x000000` represents black

Recall that the prefix `0x` indicates that the number is in hexadecimal format. When creating or manipulating bitmap images in code, you can directly set the color values of individual pixels using their hexadecimal representations. This allows for precise control over the image's appearance.

1.1 Bitmap Structure

When traversing a bitmap, it's important to understand its structure. A bitmap file typically consists of a header followed by the pixel data. The header contains metadata about the image, such as its width, height, and color depth. The pixel data is stored in a grid format, with each pixel represented by its color value.

In a 24-bit bitmap, each pixel is represented by 3 bytes (one byte for each color component). The pixel data is usually stored in a bottom-up order, meaning the first row of pixel data corresponds to the bottom row of the image.

1.1.1 Bitmap Headers

The bitmap header is typically constructed with a file section and info section. Following those comes the pixel data, represented as bytes.

```
| File Header (?) | Info Header (usually 40 bytes) | Pixel Data (rowSize*height)
↪  |
```

```
struct BITMAPFILEHEADER
{
    WORD bfType; //specifies the file type
    DWORD bfSize; //specifies the size in bytes of the bitmap file
    WORD bfReserved1; //reserved; must be 0
    WORD bfReserved2; //reserved; must be 0
    DWORD bfOffBits; //species the offset in bytes from the bitmapfileheader to
    ↪  the bitmap bits
};
struct BITMAPINFOHEADER
{
    DWORD biSize; //specifies the number of bytes required by the struct
    LONG biWidth; //specifies width in pixels
    LONG biHeight; //species height in pixels
    WORD biPlanes; //specifies the number of color planes, must be 1
```

```
    WORD biBitCount; //specifies the number of bit per pixel
    DWORD biCompression;//spcifies the type of compression
    DWORD biSizeImage; //size of image in bytes
    LONG biXPelsPerMeter; //number of pixels per meter in x axis
    LONG biYPelsPerMeter; //number of pixels per meter in y axis
    DWORD biClrUsed; //number of colors used by th ebitmap
    DWORD biClrImportant; //number of colors that are important
};
```

You can refer to the BMP file format specification and the bitmap article for more information. Let's create variables to hold important fields:

- `int w = bih.biWidth;` - width of the image in pixels
- `int h = bih.biHeight;` - height of the image in pixels
- `int size = bih.biSizeImage;` - the total size of the pixel data in bytes
- Note that we create `bytesPerPixel = 3` - the number of bytes per pixel (3 for 24-bit RGB)

Let's also preemptively build a pixel structure that can contain the information of an individual value at coordinate (x, y):

```
struct PIXEL
{
    // Each value is in range 0 to 255 represented as a byte
    BYTE b; // 1 byte
    BYTE g; // 1 byte
    BYTE r; // 1 byte
};
```

1.1.2 Padding

Remember that bytes are stored in a contiguous block of memory, as one long string of bytes. The computer does not inherently know where one row ends and the next begins (a concept called **row-major order**)[1]. We need to calculate the starting index of each pixel based on its row and column position. But, there is an issue: each row of pixel data in a bitmap must be aligned to a 4-byte boundary.

Padding exists in bitmap images to ensure that each row of pixel data is aligned to a 4-byte boundary in memory, which was a design choice made to improve performance and

[1] In row-major order, 2D arrays like bitmaps are stored sequentially in memory, row by row. To review this concept, refer back to the Vectors and Matrices Chapter, which explains how matrices can be analyzed in row-major order.

simplicity on early computer systems. Processors and hardware are more efficient when reading data that begins at predictable, aligned memory addresses, and forcing each row to occupy a size divisible by four bytes guarantees this alignment.

Because bitmap pixels do not always naturally fill a multiple of four bytes—especially in formats like 24-bit images where each pixel uses three bytes—extra, non-image bytes are added to the end of each row to reach the required alignment. These padding bytes do not represent color information and are ignored when displaying the image, but they ensure that each row starts at a consistent location in memory, making bitmap files easier and faster for software and hardware to process.

This means that if the width of the image (in bytes) is not a multiple of 4, we need to add padding bytes at the end of each row to ensure proper alignment. Padding, then, is extra bytes added to the end of each row of pixels so that the row size is a multiple of 4 bytes.

Let's look at an example. If we create a 4 pixel bitmap, it takes up

$$4 \text{ pixels} \times 3 \text{ bytes} = 12 \text{ bytes}$$

The memory looks something like

```
... | ?? | ?? | [B G R] [B G R] [B G R] [B G R] | ?? | ?? | ...
```

Since memory is contiguous, the question marks represent unrelated memory that does not belong to the bitmap's pixel data. Attempting to access memory outside the bounds of the image means the program is reading or writing data it does not own. Modern operating systems protect memory by dividing it into regions assigned to each program. If a program tries to access memory outside of its permitted region, the operating system immediately stops the program and reports an error known as a `Segmentation Fault`. This mechanism prevents programs from corrupting other data and helps ensure overall system stability.

12 bytes is evenly divisible by 4, so *no padding is needed*.

However, if we create a 3 pixel bitmap, it would take up:

$$3 \text{ pixels} \times 3 \text{ bytes} = 9 \text{ bytes}$$

9 is not evenly divisible by 4, so 3 bytes of memory must be added as padding for row-alignment.

```
... | ?? | ?? | [B G R] [B G R] [B G R] [ P ] [ P ] [ P ] | ?? | ?? | ...
```

Note that we read the pixels in `B G R` order in memory, but hex colors are read usually to

humans via `R G B` order. This gets into a concept called *little endian*. Put simply in a stack overflow post:

> RGB is a byte-order. But a deliberate implementation choice of most vanilla Graphics libraries is that they treat colours as unsigned 32-bit integers internally, with the three (or four, as alpha is typically included) components packed into the integer.
>
> On a little-endian machine (such as x86) the integer `0x01020304` will actually be stored in memory as `0x04030201`. And thus `0x00BBGGRR` will be stored as `0xRRGGBB00`!

So how can we create an equation for finding a pixel at the coordinates (x, y) if memory is in a contiguous line?

The calculation `rawRowSize = width * bytesPerPixel` provides a raw row estimate, but if padding is included, we need to round up by 4. The multiples of 4 look like 0, 4, 8, 12, 16, 20, 24, 28, 32..

- If `rawRowSize` is already one of these, we want to keep it.
- If not, we want to round upwards to the next multiple.

One way to accomplish this rounding is to take advantage of integer division. By adding a small offset before dividing, we can ensure that any value which is not already divisible by 4 is pushed into the next group of four bytes. Specifically, adding 3 to the raw row size guarantees that the result will round upward when divided by 4 using integer arithmetic.

This gives us the following expression for the true number of bytes in a single row, including padding:

$$\texttt{rowSize} = \left\lfloor \frac{\texttt{rawRowSize} + 3}{4} \right\rfloor \times 4$$

In C++, this is commonly written as:

```
int rowSize = ((width * bytesPerPixel + 3) / 4) * 4;
```

This value represents the number of bytes that must be traversed in memory to move from the beginning of one row of pixels to the beginning of the next. Recall that our `bytesPerPixel` is 3, but can also be calculated by `int bytesPerPixel = bih.biBitCount / 8;`.

Once the padded row size is known, we can compute the location of any pixel at coordinates (x, y). Because bitmap pixel data is stored row by row in a contiguous block of

memory, the offset to the start of row y is given by

$$y \times rowSize$$

Within that row, each pixel occupies `bytesPerPixel` bytes, so the offset to column x is:

$$x \times \texttt{bytesPerPixel}$$

Together, these two offsets combined yields the final memory index for pixel (x, y):

$$\texttt{index}(x, y) = y \times \texttt{rowSize} + x \times \texttt{bytesPerPixel}$$

In code, this calculation appears as:

```
int offset = y * rowSize + x * bytesPerPixel;
```

This formula will come in handy later in our code!

Let's write the full code for bitmap analysis. Starting with store the bitmap that our program reads into a memory allocated array, we can load our bitmap into a file, and read all the file information from it. We also can then utilize our equations `rowSize` and `idx` to get the individual pixel at $(x, y) \in (w, h)$.

```
// program arguments: ./bitmap_reader inputname.bmp outputname.bmp
if (argc !=3) return 1;
string inputname = argv[1];
string outputname = argv[2];


FILE *inputfile = fopen(inputname.c_str(), "rb"); // read byte only mode

BITMAPFILEHEADER bfh;
BITMAPINFOHEADER bih;

fread(&bfh.bfType, 2, 1, inputfile);
fread(&bfh.bfSize, 4, 1, inputfile);
fread(&bfh.bfReserved1, 2, 1, inputfile);
fread(&bfh.bfReserved2, 2, 1, inputfile);
fread(&bfh.bfOffBits, 4, 1, inputfile);
fread(&bih, sizeof(BITMAPINFOHEADER), 1, inputfile);

int w = bih.biWidth;
int h = bih.biHeight;
int bytesPerPixel = bih.biBitCount / 8; // 3 BYTES, stored in BGR order
int rowSize = ((w * bytesPerPixel + 3) / 4) * 4;
int size = rowSize * abs(h);
bih.biSizeImage = size;
```

```
bfh.bfSize = bfh.bfOffBits + bih.biSizeImage;

fseek(inputfile, bfh.bfOffBits, SEEK_SET); // offset from header
BYTE* data = (BYTE *)malloc(size);
fread(data, size, 1, inputfile);
fclose(inputfile);

// ----- FORCE STANDARD BMP HEADER HERE -----
bih.biSize = 40;
bih.biSizeImage = size;
bfh.bfOffBits = 54;
bfh.bfSize = 54 + size;
```

From there, we can set up an output file:

```
FILE *outfile = fopen(outputname.c_str(), "wb"); // creates a new file in write
↪  bytes mode
BYTE* out = (BYTE *) malloc(size);

fwrite(&bfh.bfType,     2, 1, outfile);
fwrite(&bfh.bfSize,     4, 1, outfile);
fwrite(&bfh.bfReserved1,2, 1, outfile);
fwrite(&bfh.bfReserved2,2, 1, outfile);
fwrite(&bfh.bfOffBits,  4, 1, outfile);
fwrite(&bih, sizeof(BITMAPINFOHEADER), 1, outfile);
```

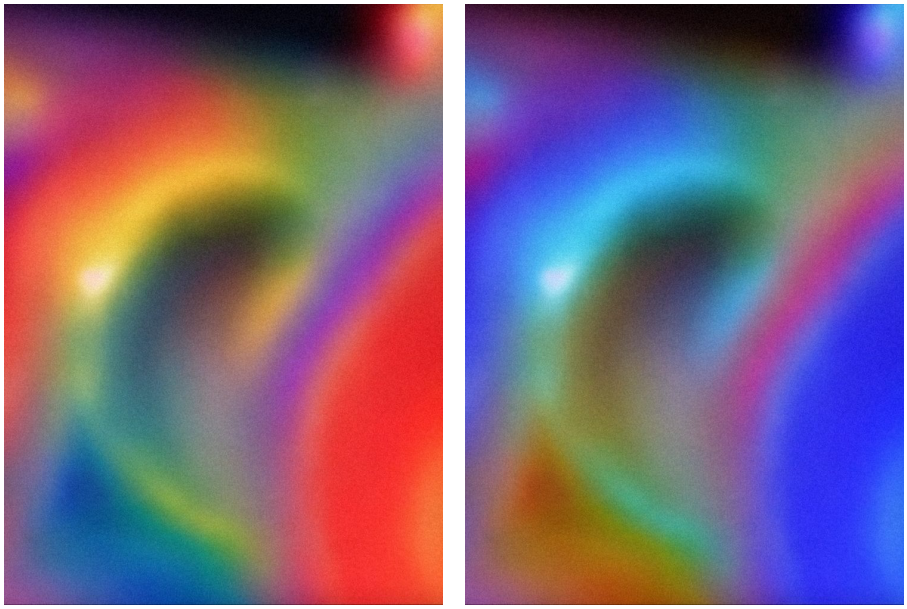
And then, we can loop through to the `w` and `h` to analyze each individual pixel. What we will do as a first test of pixel alteration is *swap red and blue values* in our output.

```
for (int x = 0; x < w; x++)
{
    for (int y = 0; y < h; y++)
    {
        int idx = y * rowSize + x * 3;

        PIXEL p;
        BYTE B = data[idx];
        BYTE G = data[idx + 1];
        BYTE R = data[idx + 2];
        p = { B, G, R };

        PIXEL o;
        o = { R, G, B }; // Swaps red and blue values
        out[idx + 0] = o.b;
        out[idx + 1] = o.g;
        out[idx + 2] = o.r;
    }
}
```

Note that we search our data array at `idx` to locate the blue component, and adding 1 and 2 respectively gets the next two bytes (green and red). This works because

- Memory is a linear array of bytes
- Pixels are stored contiguously
- The order is fixed by the BMP format

After compiling, running this using `./bitmap_reader cow.bmp output.bmp` and `./bitmap_reader gradient.bmp output.bmp` produces the following outputs (see Figures **??** and **??**):



(a) Input bitmap of a gradient. (b) Output bitmap.

Figure 1.1: Running `bitmap_reader.cpp` on a gradient, swapping red and blue channels.

(a) Input image of a highland cow. (b) Output bitmap.

Figure 1.2: Running `bitmap_reader.cpp` on a gradient, swapping red and blue channels.

The entire base code for swapping the red and blue channels is in your Digital Resources, as well as a basic copy paste input-output bitmap. A lot of this stays the same of other pixel-based modifications, such as the next Exercise.

Exercise 1 Eliminate the Green component

Now try it yourself:

Alter both `cow.bmp` and `gradient.bmp` such that the green channel is eliminated. Keep the red and blue channels the same (don't swap them).

Working Space

Answer on Page 17

1.2 Creating a Bitmap with C++

What if we aren't given an input file? Can we create a bitmap?

Of course, let's create a simple 3x3 bitmap of the following colors:

```
[ Red ] [ White ] [ White ]
[ Black ] [ Blue ] [ White ]
[ Black ] [ Black ] [ Green ]
```

Although the image is conceptually two-dimensional, the bitmap file stores its pixel data as a one-dimensional sequence of bytes.

There are, however, a few constraints:

- 14-bit file header
- 40-byte info header
- pixel data with required row padding

```
| File Header (14) | Info Header (40) | Pixel Data |
```

Calcuations:

- Each pixel $= 3$ `bytes`
- Row $= 3$ `pixels` $\times 3$ `bytes per pixel` $= 9$ `pixels`
- 3 `padding bytes` $\implies 12$ `bytes per row`
- 12 `bytes` $\times 3$ `rows` $= 36$ `bytes`

Here is the basic bitmap struct with filled in values:

```c
#pragma pack(push, 1)
struct BITMAPFILEHEADER {
    uint16_t bfType = 0x4D42; // 'BM'
    uint32_t bfSize;
    uint16_t bfReserved1 = 0;
    uint16_t bfReserved2 = 0;
    uint32_t bfOffBits = 54; // 14 + 40
};

struct BITMAPINFOHEADER {
    uint32_t biSize = 40;
    int32_t  biWidth = 3;
    int32_t  biHeight = 3;
    uint16_t biPlanes = 1;
    uint16_t biBitCount = 24;
    uint32_t biCompression = 0; // BI_RGB
    uint32_t biSizeImage;
```

```
    int32_t  biXPelsPerMeter = 0;
    int32_t  biYPelsPerMeter = 0;
    uint32_t biClrUsed = 0;
    uint32_t biClrImportant = 0;
};
#pragma pack(pop)
```

These structures match the on-disk layout of a bitmap header exactly. The `#pragma pack` directive ensures that no padding bytes are inserted by the compiler.

Recall that we can reuse our calculations from the first example to find rowsize and other variables, but this time, we define them instead of fetching them from the input:

```
const int width = 3;
const int height = 3;
const int bytesPerPixel = 3;
const int rowSize = ((width * bytesPerPixel + 3) / 4) * 4; // 12
const int imageSize = rowSize * height; // 36
```

Now we can create a new file and write the file headers individually:

```
FILE* f = fopen("custom_made.bmp", "wb");

BITMAPFILEHEADER bfh;
BITMAPINFOHEADER bih;

bih.biSizeImage = imageSize;
bfh.bfSize = bfh.bfOffBits + imageSize;

fwrite(&bfh, sizeof(bfh), 1, f);
fwrite(&bih, sizeof(bih), 1, f);
```

At this point, the file contains only metadata. No pixel values have been written yet.

Recall that bitmap pixel data is stored bottom-up, meaning the first row written corresponds to the bottom row of the image.

```
Memory order:
Row 2 (bottom)
Row 1
Row 0 (top)
```

Let's write the third row first:

```
// ROW 3 (bottom)
// Black      Black          Green
row[0] = 0;   row[1] = 0;   row[2] = 0;      // Black
row[3] = 0;   row[4] = 0;   row[5] = 0;      // Black
row[6] = 0;   row[7] = 255; row[8] = 0;      // Green
fwrite(row, rowSize, 1, f);
```

Each group of three bytes represents one pixel in BGR order. Any remaining bytes in the row serve as padding and are ignored when the image is displayed.

Row 2:

```
// ROW 2
// Black     Blue           White
row[0] = 0;   row[1] = 0;   row[2] = 0;      // Black
row[3] = 255; row[4] = 0;   row[5] = 0;      // Blue
row[6] = 255; row[7] = 255; row[8] = 255;    // White
fwrite(row, rowSize, 1, f);
```

Row 1:

```
// ROW 1 (top)
// Red        White          White
row[0] = 0;   row[1] = 0;   row[2] = 255;    // Red
row[3] = 255; row[4] = 255; row[5] = 255;    // White
row[6] = 255; row[7] = 255; row[8] = 255;    // White
fwrite(row, rowSize, 1, f);
```

And that's it! The compiler automatically converts the 255s to `0xFF`. Close the file and run the program, and you get an extremely small output. Enlarge the output and you get:

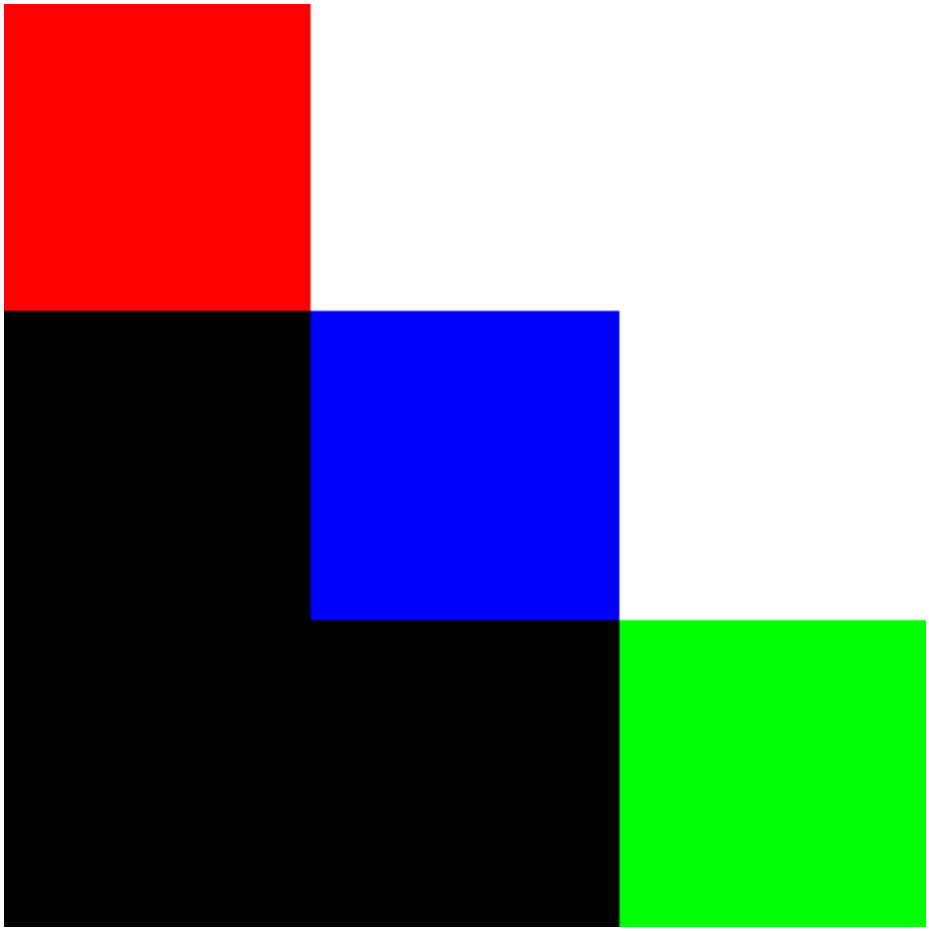


Figure 1.4: Your custom bitmap!

Exercise 2 8 Pixel Colorful Bitmap

Working Space

Create a 4×2 bitmap with the following layout:

```
[ Yellow ] [ Magenta ] [ Cyan ] [
↪  White ]
[ Black  ] [ Red     ] [ Green ] [
↪  Blue  ]
```

Bitmap Requirements:

- Width: 4 pixels
- Height: 2 pixels
- Color depth: 24-bit (RGB, stored as BGR)
- Compression: None (BI_RGB)
- Row padding: Rows must be aligned to 4-byte boundaries

Answer the following questions before writing any code:

1. How many bytes does each pixel use?
2. How many bytes are required for one row before padding?
3. How many padding bytes are required per row?
4. What is the total size of the pixel data?
5. What is the total file size?

Answer on Page 18

1.3 Summary

In this chapter, we experimented with bitmaps. Specifically,

- loading a bitmap in C++
- creating and copying bitmap file and info headers
- creating a pixel structure to work with hexadecimal bytes and hexcodes
- swapping the channels of a bitmap
- creating a bitmap from scratch in C++

Appendix A

Answers to Exercises

Answer to Exercise 1 (on page 9)

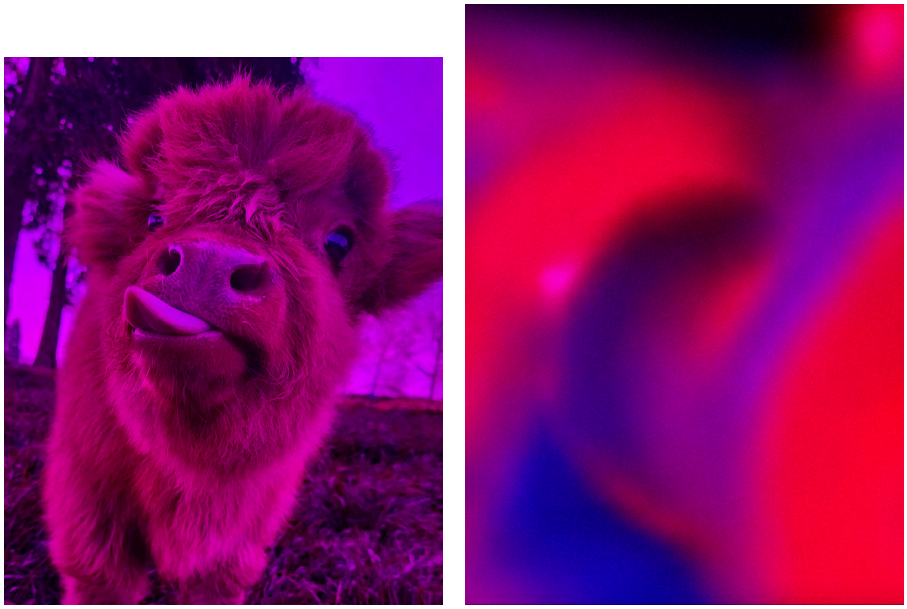
The only thing that changes in our for loop is the writing of the output pixel.

```
// CODE ABOVE STAYS THE SAME
for (int x = 0; x < w; x++)
{
    for (int y = 0; y < h; y++)
    {
        int idx = y * rowSize + x * 3;

        PIXEL p;
        BYTE B = data[idx];
        BYTE G = data[idx + 1];
        BYTE R = data[idx + 2];
        p = { B, G, R };

        out[idx + 0] = p.b;
        out[idx + 1] = 0; // NOTE THE CHANGE HERE
        out[idx + 2] = p.r;
    }
}
```

What is happening here?

In an RGB image, each pixel's final color is the combination of red, green, and blue intensities. If the green value is removed, every pixel changes from (R,G,B) to (R,0,B). Colors that relied heavily on green—such as greens, yellows, and many skin tones—lose a major part of their intensity and appear much darker or shifted in hue. Pure green areas become black, yellow areas (red + green) become red, and cyan areas (green + blue) become blue.

Visually, the image often takes on a magenta or purplish tint (see Figures 1.3a and 1.3b), because magenta is the combination of red and blue with no green. Overall brightness decreases, since green contributes significantly to the brightness in human vision. Refer to Figures 1.2a and 1.1a for the original bitmaps.



(a) Output bitmap - cow. (b) Output bitmap - gradient.

Figure 1.3: Running `bitmap_no_green.cpp` on both provided bitmaps.

Answer to Exercise 2 (on page 14)

1. How many bytes does each pixel use? - 3 `bytes per pixel`
2. How many bytes are required for one row before padding? 4 `pixels`$\times$3 `bytes per pixel` $=$ 12 `bytes`
3. How many padding bytes are required per row? 0 `padding bytes`
4. What is the total size of the pixel data? 12 bytes per row $\times$ 2 `rows` $=$ 24 `bytes`
5. What is the total file size, including header structs? 14 `bytes for BFH`+40 `bytes for BIH` $=$ 54 `bytes` $\implies$ $24 + 54 = 78$ `bytes`

Remember that we need to write the rows bottom-up, so the second row is written first followed by the first row.

Let's establish the colors needed. Remember that we need to swap to BGR order:

- Black $= (0, 0, 0)$
- Red $= (0, 0, 255)$
- Green $= (0, 255, 0)$

- Blue = (255, 0, 0)
- Yellow = (0, 255, 255) (R=255,G=255,B=0 $\implies$ BGR = 0,255,255)
- Magenta = (255, 0, 255) (R=255,B=255 $\implies$ BGR = 255,0,255)
- Cyan = (255, 255, 0) (G=255,B=255 $\implies$ BGR = 255,255,0)
- White = (255, 255, 255)

Overall the program looks like this:

```cpp
#include <cstdio>
#include <cstdint>
#include <cstring>


#pragma pack(push, 1)
struct BITMAPFILEHEADER {
    uint16_t bfType = 0x4D42; // 'BM'
    uint32_t bfSize;
    uint16_t bfReserved1 = 0;
    uint16_t bfReserved2 = 0;
    uint32_t bfOffBits = 54;
};

struct BITMAPINFOHEADER {
    uint32_t biSize = 40;
    int32_t  biWidth = 4;
    int32_t  biHeight = 2;
    uint16_t biPlanes = 1;
    uint16_t biBitCount = 24;
    uint32_t biCompression = 0;
    uint32_t biSizeImage;
    int32_t  biXPelsPerMeter = 0;
    int32_t  biYPelsPerMeter = 0;
    uint32_t biClrUsed = 0;
    uint32_t biClrImportant = 0;
};
#pragma pack(pop)

int main(int argc, char const *argv[])
{
    FILE* f = fopen("fourbytwo.bmp", "wb");

    BITMAPFILEHEADER bfh;
    BITMAPINFOHEADER bih;

    const int width = 4;
    const int height = 2;
    const int bytesPerPixel = 3;
    const int rowSize = ((width * bytesPerPixel + 3) / 4) * 4; // 12
    const int imageSize = rowSize * height;                    // 24
```

```
    bih.biSizeImage = imageSize;
    bfh.bfSize = bfh.bfOffBits + imageSize;

    fwrite(&bfh, sizeof(bfh), 1, f);
    fwrite(&bih, sizeof(bih), 1, f);
    uint8_t row[rowSize];

    // ----------------------------
    // Write bottom row first:
    // [ Black ] [ Red ] [ Green ] [ Blue ]
    // ----------------------------
    std::memset(row, 0, rowSize);

    // Black
    row[0] = 0;   row[1] = 0;   row[2] = 0;
    // Red (BGR = 0,0,255)
    row[3] = 0;   row[4] = 0;   row[5] = 255;
    // Green (BGR = 0,255,0)
    row[6] = 0;   row[7] = 255; row[8] = 0;
    // Blue (BGR = 255,0,0)
    row[9] = 255; row[10]= 0;   row[11]= 0;

    fwrite(row, rowSize, 1, f);

    // ----------------------------
    // Write top row:
    // [ Yellow ] [ Magenta ] [ Cyan ] [ White ]
    // ----------------------------
    std::memset(row, 0, rowSize);

    // Yellow (BGR = 0,255,255)
    row[0] = 0;   row[1] = 255; row[2] = 255;
    // Magenta (BGR = 255,0,255)
    row[3] = 255; row[4] = 0;   row[5] = 255;
    // Cyan (BGR = 255,255,0)
    row[6] = 255; row[7] = 255; row[8] = 0;
    // White (BGR = 255,255,255)
    row[9] = 255; row[10]= 255; row[11]= 255;

    fwrite(row, rowSize, 1, f);

    fclose(f);
    return 0;
}
```

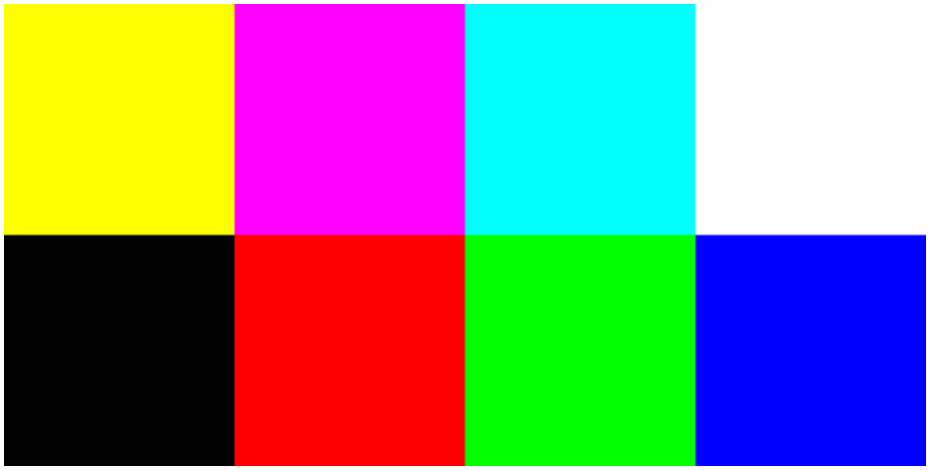
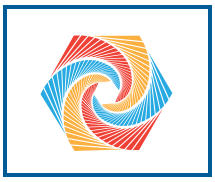


Figure 1.5: Exercise Output of a 4 by 2 bitmap.



Index