Contents

1 How Cameras Work *3*

1.1 The Light That Shines On the Cow *3*

1.2 Light Hits the Cow *5*

1.3 Pinhole camera *7*

1.4 Lenses *8*

1.5 Sensors *10*

1.6 Aspect Ratio *10*

1.6.1 Printing *12*

2 How Eyes Work *13*

2.1 Eye problems *14*

2.1.1 Glaucoma *14*

2.1.2 Cataracts *15*

2.1.3 Nearsightedness, farsightedness, and astigmatism *15*

2.2 Seeing colors *16*

2.3 Pigments *18*

3 Images in Python *21*

3.1 Adding color *22*

3.2 Using an existing image *25*

4 Representing Natural Numbers *27*

4.1 Base-10 (Decimal) *27*

4.2 Base-2 (Binary) *28*

4.2.1 Bits and Bytes 29
4.3 Base-16 (Hexadecimal) 30
4.3.1 Hex Color Codes 31
4.4 Base-8 (Octal) 31
4.4.1 Why Octal Matters 32

5 Bitmaps ***33***
5.1 Bitmap Structure 34
5.1.1 Bitmap Headers 34
5.1.2 Padding 35
5.2 Creating a Bitmap with C++ 39
5.3 Connection to transformations 43
5.4 Summary 44

A Answers to Exercises ***45***

Index 49

Chapter 1

How Cameras Work

Let's say it is a sunny day and you are standing in a field a few meters from a cow. You use the camera on your phone to take a picture of the cow. How does that whole process work?

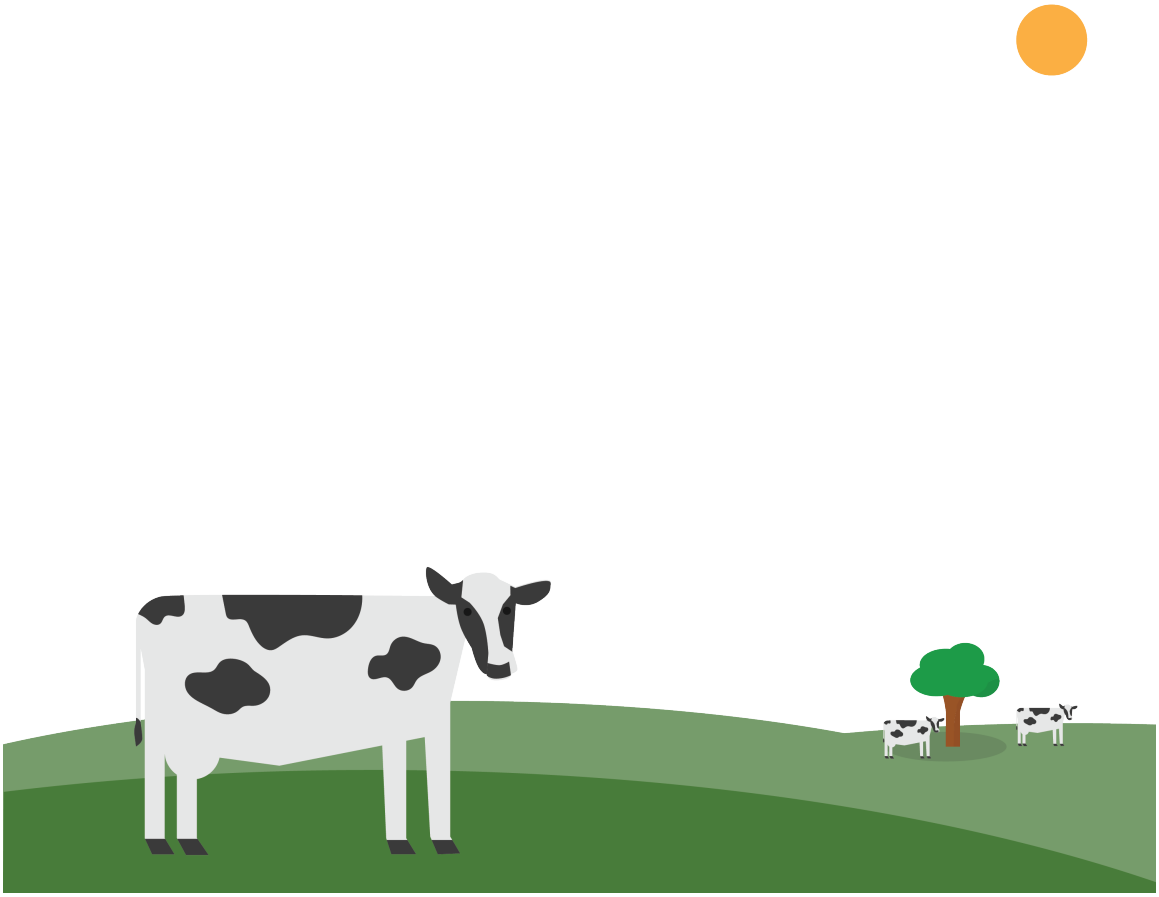


Figure 1.1: A cow in a field.

1.1 The Light That Shines On the Cow

The sun is a sphere of hot gas. About 70% of the gas is hydrogen. About 28% is helium. There's also a little carbon, nitrogen, and oxygen.

Gradually, the sun is converting hydrogen into helium through a process known as "nu-

clear fusion" (which we will be discussing more in a future chapter). A large amount of heat is created in this process. This heat makes the gases glow.

How does heat make things glow? The heat pushes the electrons into higher orbitals. When they come back down to a lower orbital, they release a photon of energy, which travels away from the atom as an electromagnetic wave.

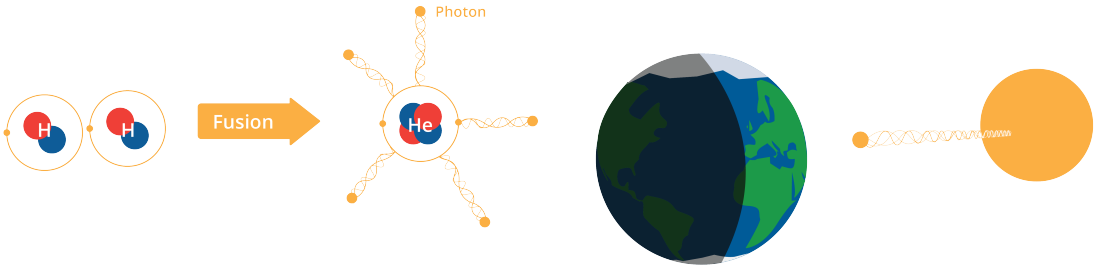


Figure 1.2: Photons are released when the sunlight hits the cow.

Heat is not the only way to push the electrons into a higher orbital. For example, a fluorescent lightbulb is filled with gas. When we pass electricity through the gas, its electrons are moved to a higher orbital. When they fall back to a lower orbital, light is created.

What is the frequency of the wave that the photon travels on? Depending on what orbital it falls from and how far it falls, the photon created has different amounts of energy. The amount of energy determines the frequency of the electromagnetic wave.

Formula for energy of a photon

If you want to know the amount of energy E in a photon, here is the formula:

$$E = \frac{hc}{\lambda}$$

where c is the speed of light, λ is the wavelength of the electromagnetic wave, and h Planck's constant: $6.63 \times 10^{-34} m^2 kg/s$

For example, a red laser light has a wavelength of about 630 nm. So, the energy in each photon is:

$$\frac{(300 \times 10^{6})(6.63 \times 10^{-34})}{630 \times 10^{-9}} = 3.1 \times 10^{-19} \text{ joules}$$

In the sun, there are several kinds of molecules and each has a few different orbitals that the electrons can live in. Thus, the light coming from the sun is made up of electromagnetic waves of many different frequencies.

We can see some of these frequencies as different colors, but some are invisible to humans, such as ultraviolet and infrared.

1.2 Light Hits the Cow

When these photons from the sun hit the cow, the hide and hairs of the cow will absorb some of the photons. These photons will become heat and make the cow feel warm. Some of the photons will not be absorbed – they will leave the cow. When you say "I see the cow," what you are really saying is "I see some photons that were not absorbed by the cow."

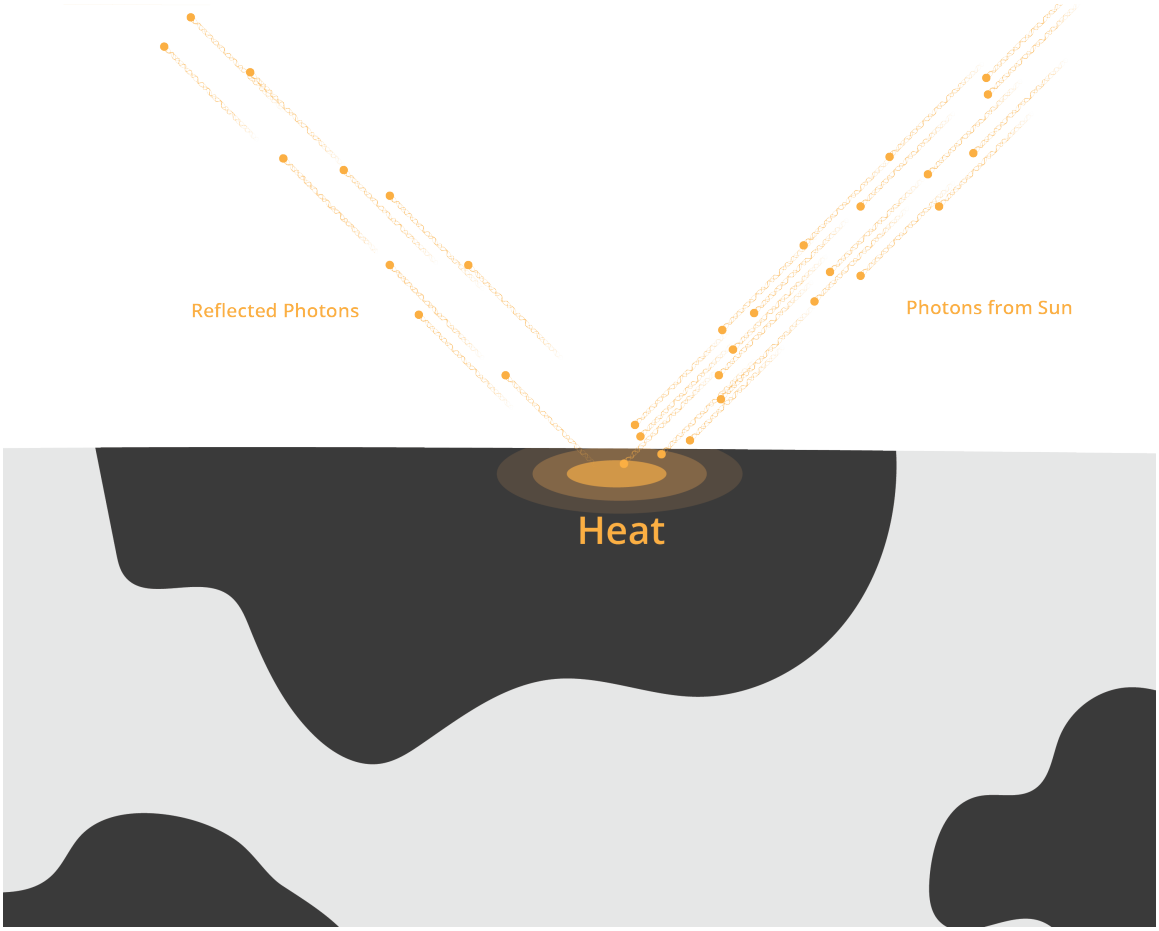


Figure 1.3: Photons hitting the cow create a heated spot, where some are absorbed.

Different materials absorb different amounts of each wavelength. A plant, for example, absorbs a large percentage of all blue and red photons that hit it, but it absorbs only a small percentage of the green photons that hit it. Thus, we say "That plant is green."

White things absorb very small percentages of photons of any visible wavelength. Black things absorb very *large* percentages of photons of any visible wavelength. That is why, on a hot summer day, a black car with black seats and interior will heat up on the inside much hotter than a white car.

Before we go on, let's review: The sun creates photons that travel as electromagnetic waves of assorted wavelengths to the cow. Many of those photons are absorbed, but some are not. Some of those photons that are not absorbed go into the lens of our camera.

1.3 Pinhole camera

The simplest cameras have no lenses. They are just a box. The box has a tiny hole that allows photons to enter. The side of the box opposite the hole is flat and covered with film or some other photo-sensitive material.

The photons entering the box continue in the same direction they were going when they passed through the hole. Thus, the photons that entered from high hit the back wall at a low point. The photons that came from the left hit the back wall on the right. This is how the image is projected onto the back wall, rotated 180 degrees; what was up is down, what was on the left is on the right.

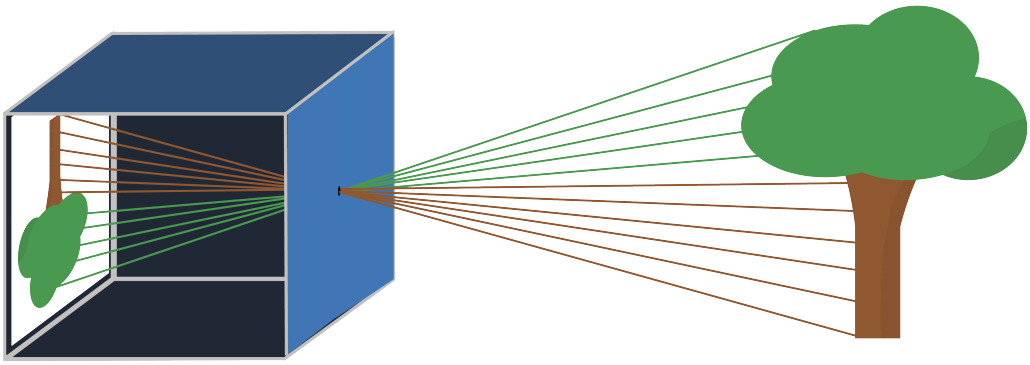


Figure 1.4: A pinhole camera flips the image it "sees" by 180°.

Exercise 1 Height of the image

Working Space

FIXME: cow swap Let's say that that the pinhole is exactly the same height as the shoulder of the cow, and that the shoulder is directly above one hoof. This means the pinhole, the shoulder, and the hoof form a right triangle.

Now, let's say that the camera is being held perpendicular to the ground. The pinhole, the image of the shoulder, and the image of the hoof on the back wall of the camera now also form a right triangle.

These two triangles are similar.

The shoulder is 2 meters from the hoof. The cow is standing 3 meters from the camera. The distance from the pinhole to the back wall of the camera is 3 cm. How tall is the image of the cow on the back wall of the camera?

Answer on Page 45

1.4 Lenses

Now, a quick review: A photon leaves the sun in some random direction. It travels 150 million km from the sun and hits a cow. It is not absorbed by the cow, and heads off in a new direction. It passes through the pinhole and hits the back wall of the camera. That seems incredibly improbable, right?

It actually is relatively improbable, especially if there isn't a lot of light — like you are taking the picture at dusk. To increase the odds, we added a *lens* to the camera. If you focus a lens on a wall and you draw a dot on that wall, the lens is designed such that all the photons from the dot that hit the lens get redirected to the same spot on the back wall

of the camera — regardless of which path it took to get to the lens.

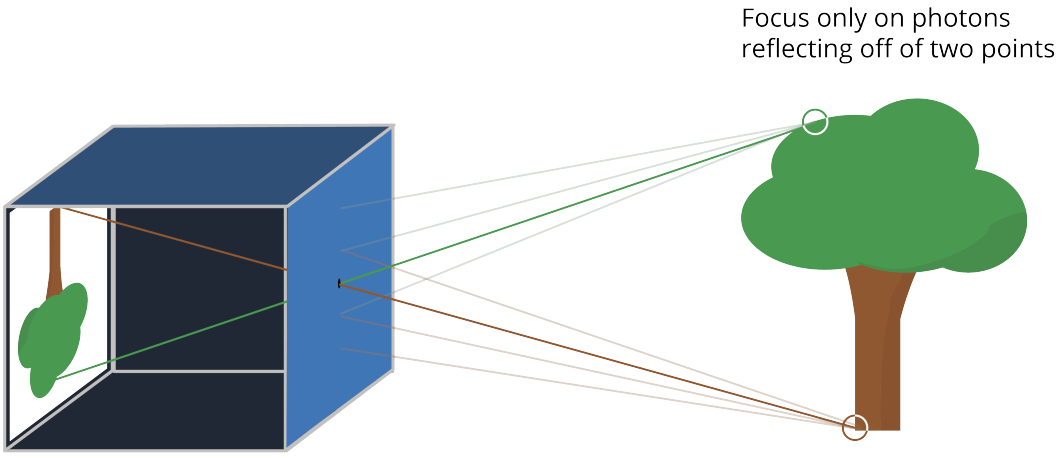


Figure 1.5: A pinhole with a lens allows you to choose the points it reflects.

Note that the image still gets flipped. There is a *focal point* that all the photons pass through.

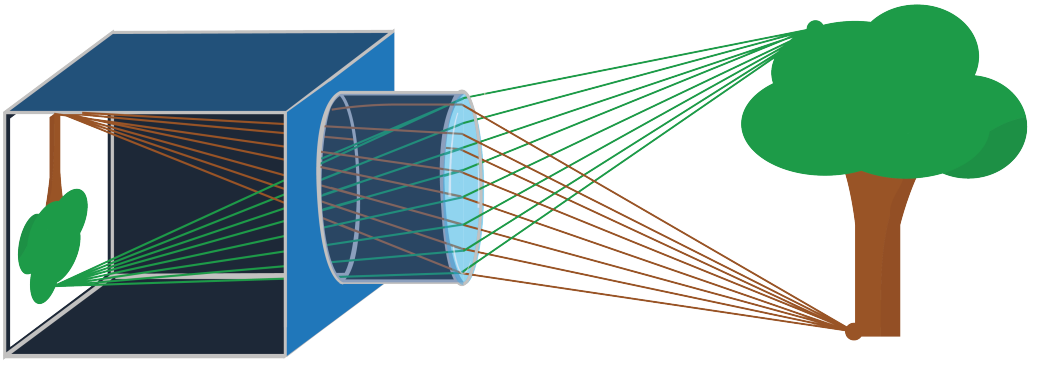


Figure 1.6: A lens still flips the image, but introduces a focal point.

The distance from the lens to its focal point is called the lens's *focal length*. Telephoto lenses, that let you take big pictures of things that are far away, have long focal lengths. Wide-angle lenses have short focal lengths.

1.5 Sensors

The camera on your phone has a sensor on the back wall of the camera. The sensor is broken up into tiny rectangular regions called pixels. When you say a sensor is 6000 by 4000 pixels (most common ratio for photography), we are saying the sensor is a grid of 24,000,000 pixels: 6000 pixels wide and 4000 pixels tall.

Each pixel has three types of cavities that take in photos. One of the cavities measures the amount of short wavelength light, like blues and violets. One of the cavities measures the long wavelength light, like reds and oranges. One of the cavities measures the intensity of wavelengths in the middle, like greens. Thus, if your camera has a *resolution* of 6000×4000, the image is 24,000,000 pixels yields three numbers: intensity of long wavelength, mid wavelength, and short wavelength light, for a total of 72,000,000 numbers. We call these numbers "RGB" for Red, Green, and Blue. The RGB values range from $0 - 255$ for each channel. We will talk more about this when hexadecimal is introduced.

1.6 Aspect Ratio

When you are buying a monitor, TV, or even smartphone, you may hear the term **aspect ratio**. The aspect ratio is a set of two numbers representing the ratio between the width and the height. Conventionally, the width comes first, separated by a colon, and then the height:

$$\text{width} : \text{height}$$

Some of the most common aspect ratios are:

- $16 : 9$ (known as Widescreen)
- $17 : 9$
- $9 : 16$
- $4 : 3$ (known as Fullscreen)
- $4 : 5$
- $3 : 2$
- $1 : 1$ (perfect square, like on a smartwatch)

As we talked about above, the aspect ratio doesn't describe the actual size of the screen or image. Instead, it describes the shape. A movie theatre screen can have the same aspect ratio as your monitor, even if the screen is 360 inches and the monitor is 24 inches.

Aspect ratio describes the proportional relationship between width and height. Resolution describes the total number of pixels used to represent an image.

For example, both of the following resolutions have a $16:9$ aspect ratio:

$$1280 \times 720$$
$$1920 \times 1080$$

Mathematically, the aspect ratio is a dimensionless quantity. Since both width and height are measured in the same units (such as centimeters, inches, or pixels), the units cancel when forming the ratio. This means aspect ratio describes proportion rather than physical measurement. Most often, we determine the aspect ratio of a digital display by comparing the number of horizontal pixels to vertical pixels.

For example:

$$1920 \times 1080 \quad \Rightarrow \quad \frac{1920}{1080} = \frac{16}{9} = 16:9$$

Digital cameras and smartphone cameras capture images using a rectangular sensor. The shape of this sensor determines the default aspect ratio of the image. For example:

- Many digital cameras use a $3:2$ sensor.
- Micro Four Thirds cameras use $4:3$.
- Most video recording uses $16:9$.

Important: when you change the aspect ratio setting on a camera, the device does not change the sensor itself. Instead, it crops part of the image to match the selected ratio. Cropping reduces the number of recorded pixels, which may reduce file size. However, actual file size also depends on compression methods used by the device.

Human vision has a wider horizontal field of view than vertical field of view. Because of this, wider aspect ratios such as $16:9$ often feel more natural and immersive. This is one reason modern TVs and cinema formats favor wider screen shapes. However, with the rise of cellphones, we have switched to consistently using portrait or vertical mode, and video aspect ratios are more consistently becoming $9:16$. This allows for consistency in device quality, and commonly used for vertical videos, like in short-form content, compared to traditional widescreen videos, such as Youtube or traditional horizontal video capturing.

1.6.1 Printing

You can imagine this may cause a problem for printing out photos. Most printing paper is 8.5 inches $\times$ 11 inches or 8 inches $\times$ 10 inches. This causes an issue, as that fits a $8.5 : 11 \iff 17 : 22$ or $4 : 5$ aspect ratio, respectively. We combat this by using very standardized printing sizes (in inches):

- $6 \times 4 \to 3 : 2$,
- $7 \times 5 \to 7 : 5$,
- $10 \times 8 \to 5 : 4$,
- and, rarely, $8.5 \times 11 \to 1.29 : 1$

Going to any framing or printing store will give you a very standardized set of sizes, or some multiple of these.

Most smartphones use $4 : 3$ or $3 : 2$, which will work for 4×6 inches or 8×6 inches, respectively.

Chapter 2

How Eyes Work

Dr. Craig Blackwell has made a great video on the mechanics of the eye. You should watch it: https://youtu.be/Z8asc2SfFHM

Mechanically, your eye works a lot like a camera. The eye is a sphere with two lenses on the front: The outer lens is called the *cornea*, while the second lens is simply called "the lens." Between the two lenses is an aperture that opens wide when there is very little light, and closes very small when there is bright light. The opening is called the *pupil* and the tissue that forms the pupil is called the *iris*. When people talk about the color of your eyes, they are talking about the color of your iris. The blackness at the center of your iris is your pupil.

There are two types of photoreceptor cells in your retina: rods and cones. The rods are more sensitive; in very dark conditions, most of our vision is provided by the rods. The cones are used when there is plenty of light, and they let us see colors.

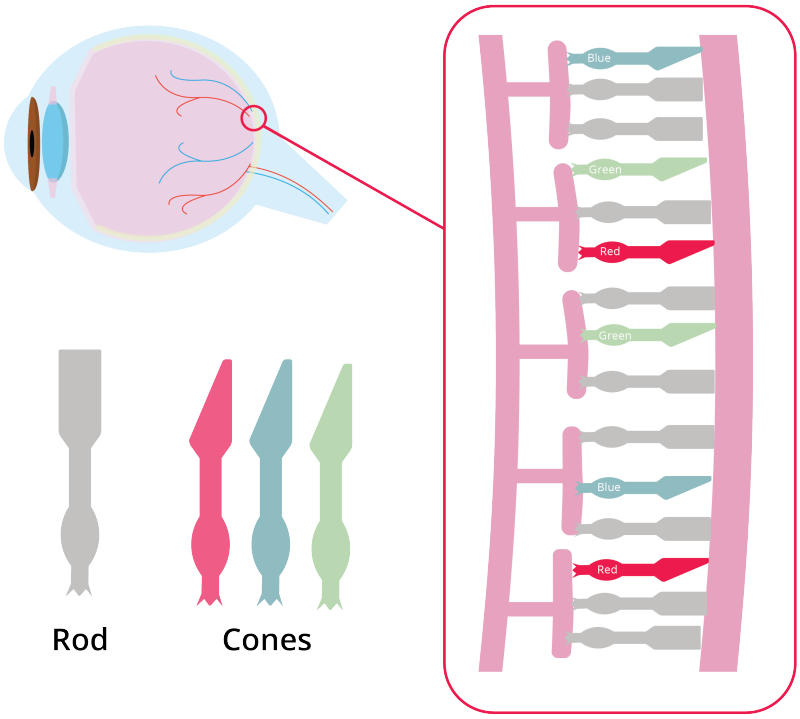


Figure 2.1: The cones and rods of the eye act as sensors and cognitive vision.

The white part around the outside of the eyeball? That is called the *sclera*.

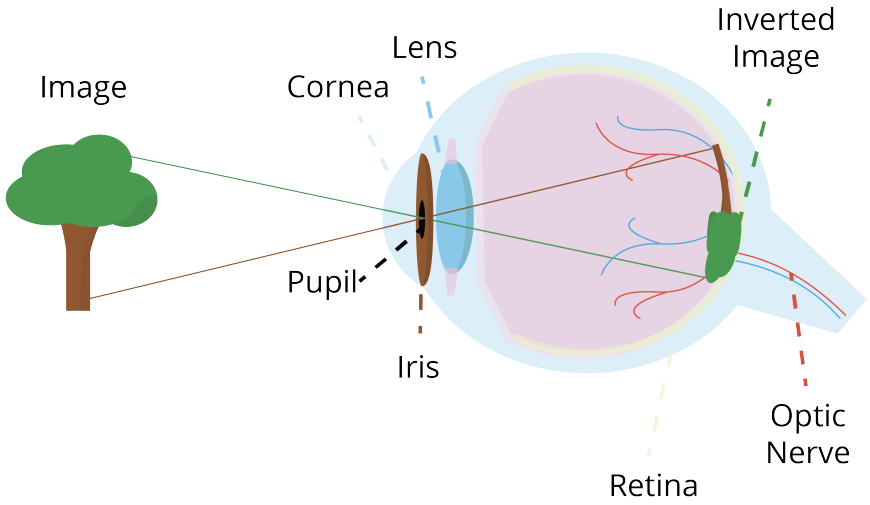


Figure 2.2: The eye works just like a camera's lens.

The walls of the eye are lined inside with the *retina*, which has sensors that pick up the light and send impulses down the optic nerve to your brain.

Just like a camera, the images are flipped when they get projected on the back of the eye (see Figure 2.2).

2.1 Eye problems

Now that you know the mechanics of the eye, let's go over a few things that commonly go wrong with the eye.

2.1.1 Glaucoma

The space between your cornea and lens is filled with a fluid called *aqueous humor*. To feed the cells of the cornea and lens, the aqueous humor carries oxygen and nutrients like blood would, but unlike blood, it is transparent so you can see. Aqueous humor is constantly being pumped into and out of that chamber. If aqueous humor has trouble

exiting, the pressure builds up and can damage the eye. This is known as *glaucoma*. See Figure 2.3.

2.1.2 Cataracts

The lens should be clear. As a person ages, the proteins in the lens break down and clump together, becoming opaque. This can also be accelerated by diabetes, too much exposure to sunlight, obesity, and high blood pressure. From the outside, the eye will look cloudy. This is called a *cataract*, and it makes it difficult for the person to see.

This problem can be corrected, however. The person's cloudy lens is removed and replaced with a clear, manufactured lens.

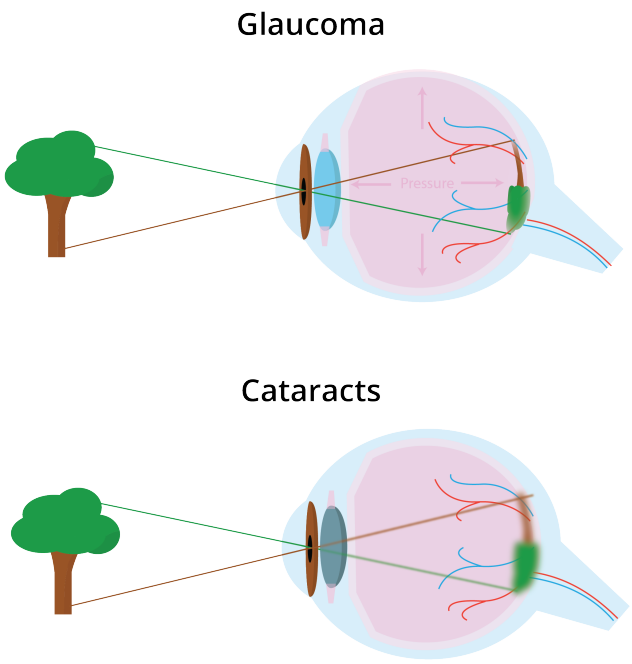


Figure 2.3: Cataracts and Glaucoma represented in the eye cross section.

2.1.3 Nearsightedness, farsightedness, and astigmatism

If you are in a dark room and a tiny LED is turned on, the photons from that LED can pass through your cornea in many different places. If your eye is focusing on that light correctly, all the photons should meet up at the same place on the retina.

FIXME: Diagram here

If the lenses are bending the light too much, the photons meet up before they hit the retina and get smeared a bit across it. To the person, the LED would appear blurry. The eye is said to be *nearsighted* or *myopic*. This signals that near objects appear correctly, but farther objects appear blurry.

If the lenses are not bending it enough, the photons would meet up behind the retina. Once again, they get smeared a bit across the retina and the LED looks blurry to the person. The eye is said to be *farsighted* or *hyperoptic*. The objects which are distant can appear clearly, while closer objects are heavily blurred.

Your lenses are supposed to bend the photons the same amount vertically and horizontally. If one dimension is focused, but the other is myopic or hyperoptic, the eye is said to have *astigmatism*.

Myopia, hyperoptia, and astigmatism can be corrected with glasses or contact lenses. Doctors can also do surgical corrections, usually by changing the shape of the cornea.

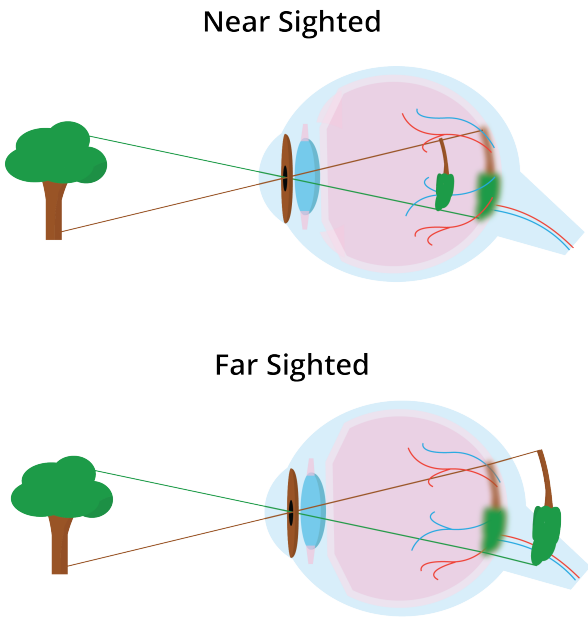


Figure 2.4: Nearsightedness versus farsightedness.

2.2 Seeing colors

TED-Ed has made a good video on how we see color. Watch it here: https://youtu.be/l8_fZPHasdo

When a rainbow forms, you are seeing different wavelengths separating from each other.

In the rainbow:

- Red is about 650 nm.
- Orange is about 600 nm.
- Yellow is about 580 nm.
- Green is about 550 nm.
- Cyan is about 500 nm.
- Blue is about 450 nm.
- Violet is about 400 nm.

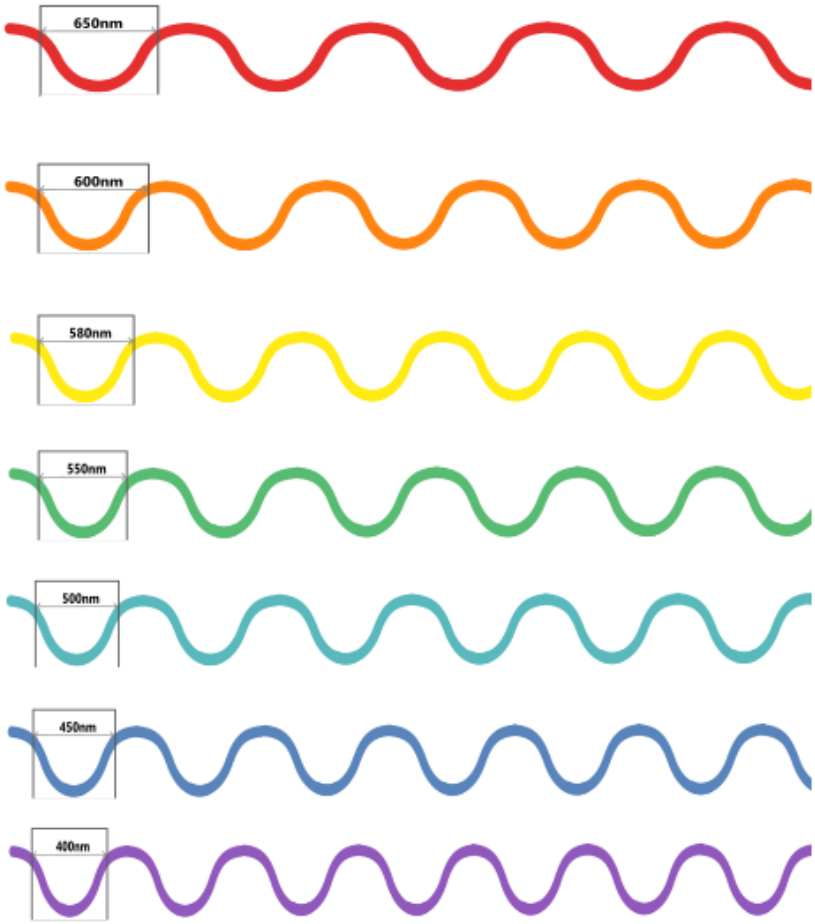


Figure 2.5: All colors have different wavelengths.

If you shine a light with a wavelength of 580 nm on a white piece of paper, you will see yellow.

However, if you shine two lights with wavelengths of 650 nm (red) and 550 nm (green), you will also see yellow.

Why? Our ears can hear two different frequencies at the same time. Why can't our eyes see two colors in the same place?

As mentioned above, the cone photoreceptors in our eyes let us see colors. There are three kinds of cones:

- Red: Cones that let us see the frequencies up to about 700nm.
- Green: Cones that are most sensitive to frequencies near 550nm.
- Blue: Cones that are most sensitive to frequencies near 450nm.

When a wavelength of 580 nm hits your retina, it excites the red and green receptors, and your brain interprets that mix as yellow.

Similarly, when light that contains both 650 nm and 550 nm waves hits your retina, it excites the red and green receptors, and your brain interprets that mix as yellow.

You can't tell the difference!

Now we know why the sensors on the camera are RGB. The camera is recording the scene as closely as necessary to fool your eye.

A TV or a color computer monitor only has three colors of pixels: red, green, and blue. By controlling the mix of them, it creates the sensation of thousands of colors to your eye.

2.3 Pigments

A color printer works in the opposite fashion. Instead of radiating colors, it puts pigments on the paper that absorb certain frequencies. A pigment that absorbs only frequencies near 650 nm (red) will appear to your eye as cyan. This makes sense, because the sensation of cyan is created when your blue and green receptors are activated.

Thus, pigment colors come in:

- Cyan: absorbs frequencies around red
- Magenta: absorbs freqencies around green

- Yellow: absorbs frequencies around blue

If you buy ink for a color printer, you know there is typically a fourth ink: black. If you put cyan, magenta, and yellow pigments on paper, the mix won't absorb all the visible spectrum in a consistent manner. Our eyes are pretty sensitive to this, so we would see brown. This is why we add black ink to get pretty grays and blacks.

We call this approach to color CMYK (as opposed to RGB). If an artist is creating an image to be viewed on a screen, they will typically make an RGB image. If they are creating an image to be printed using pigments, they typically create a CMYK image. (Most of us don't care so much — we just let the computer do conversions between the two color spaces for us.)

Chapter 3

Images in Python

An image is usually represented as a three-dimensional array of 8-bit integers. NumPy arrays are the most commonly used library for this sort of data structure.

If you have an RGB image that is 480 pixels tall and 640 pixels wide, you will need a $480 \times 640 \times 3$ NumPy array.

There is a separate library (imageio) that:

- Reads an image file (like JPEG files) and creates a NumPy array.
- Writes a NumPy array to a file in standard image formats

Let's create a simply python program that creates a file containing an all-black image that is 640 pixels wide and 480 pixels tall. Create a file called `create_image.py`:

```
import NumPy as np
import imageio
import sys

# Check command-line arguments
if len(sys.argv) < 2:
    print(f"Usage {sys.argv[0]} <outfile>")
    sys.exit(1)

# Constants
IMAGE_WIDTH = 640
IMAGE_HEIGHT = 480

# Create an array of zeros
image = np.zeros((IMAGE_HEIGHT, IMAGE_WIDTH, 3), dtype=np.uint8)

# Write the array to the file
imageio.imwrite(sys.argv[1], image)
```

To run this, you will need to supply the name of the file you are trying to create. The extension (like .png or .jpeg) will tell imageio what format you want written. Run it now:

```
python3 create_image.py blackness.png
```

Open the image to confirm that it is 640 pixels wide, 480 pixels tall, and completely black.

3.1 Adding color

Now, let's walk through through the image, pixel-by-pixel, adding some red. We will gradually increase the red from 0 on the left to 255 on the right.

```
import NumPy as np
import imageio
import sys

# Check command-line arguments
if len(sys.argv) < 2:
    print(f"Usage sys.argv[0] <outfile>")
    sys.exit(1)

# Constants
IMAGE_WIDTH = 640
IMAGE_HEIGHT = 480

# Create an array of zeros
image = np.zeros((IMAGE_HEIGHT, IMAGE_WIDTH, 3), dtype=np.uint8)

for col in range(IMAGE_WIDTH):

    # Red goes from 0 to 255 (left to right)
    r = int(col * 255.0 / IMAGE_WIDTH)

    # Update all the pixels in that column
    for row in range(IMAGE_HEIGHT):
        # Set the red pixel
        image[row, col, 0] = r

# Write the array to the file
imageio.imwrite(sys.argv[1], image)
```

When you run the function to create a new image, it will be a fade from black to red as you move from left to right:

Now, inside the inner loop, update the blue channel so that it goes from zero at the top to 255 at the bottom:

```
    # Update all the pixels in that column
    for row in range(IMAGE_HEIGHT):

        # Update the red channel
        image[row,col,0] = r

        # Blue goes from 0 to 255 (top to bottom)
        b = int(row * 255.0 / IMAGE_HEIGHT)
        image[row,col,2] = b

imageio.imwrite(sys.argv[1], image)
```

When you run the program again, you will see the color fades from black to blue as you go down the left side. As you go down the right side, the color fades from red to magenta.

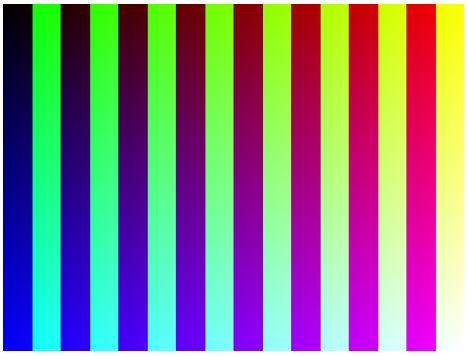
Notice that red and blue with no green looks magenta to your eye.

Next, let's add some stripes of green:

```
import NumPy as np
```

```
import imageio
import sys

# Check command line arguments
if len(sys.argv) < 2:
    print(f"Usage sys.argv[0] <outfile>")
    sys.exit(1)

# Constants
IMAGE_WIDTH = 640
IMAGE_HEIGHT = 480
STRIPE_WIDTH = 40
pattern_width = STRIPE_WIDTH * 2

# Create an image of all zeros
image = np.zeros((IMAGE_HEIGHT, IMAGE_WIDTH, 3), dtype=np.uint8)

# Step from left to right
for col in range(IMAGE_WIDTH):

    # Red goes from 0 to 255 (left to right)
    r = int(col * 255.0 / IMAGE_WIDTH)

    # Should I add green to this column?
    should_green = col % pattern_width > STRIPE_WIDTH

    # Update all the pixels in that column
    for row in range(IMAGE_HEIGHT):

        # Update the red channel
        image[row,col,0] = r

        # Should I add green to this pixel?
        if should_green:
            image[row,col,1] = 255

        # Blue goes from 0 to 255 (top to bottom)
        b = int(row * 255.0 / IMAGE_HEIGHT)
        image[row,col,2] = b

imageio.imwrite(sys.argv[1], image)
```

When you run this version, you will see the previous image in half the stripes. In the other half, you will see that green fades to cyan down the left side, and yellow fades to white down the right side.



3.2 Using an existing image

imageio can also be used to read in any common image file format. Let's read in an image and save each of the red, green, and blue channels out as its own image.

Create a new file called `separate_image.py`:

```
import imageio
import sys
import os

# Check command line arguments
if len(sys.argv) < 2:
    print(f"Usage {sys.argv[0]} <infile>")
    sys.exit(1)

# Read the image
path = sys.argv[1]
image = imageio.imread(path)

# What is the filename?
filename = os.path.basename(path)

# What is the shape of the array?
original_shape = image.shape

# Log it
print(f"Shape of {filename}:{original_shape}")

# Names of the colors for the filenames
colors = ['red','green','blue']

# Step through each of the colors
for i in range(3):
```

```
    # Create a new image
    newimage = np.zeros(original_shape, dtype=np.uint8)

    # Copy one channel
    newimage[:,:,i] = image[:,:,i]

    # Save to a file
    new_filename = f"{colors[i]}_{filename}"
    print(f"Writing {new_filename}")
    imageio.imwrite(new_filename, newimage)
```

Now, you can run the program with any common RGB image type:

```
python3 separate_image.py dog.jpg
```

This will create three images: `red_dog.jpg`, `green_dog.jpg`, and `blue_dog.jpg`.

Chapter 4

Representing Natural Numbers

Natural numbers are positive whole numbers, such as 1, 2, 3, and so on. -5 is not a natural number. π is not a natural number. $\frac{1}{2}$ is not a natural number.

4.1 Base-10 (Decimal)

You are used to seeing the natural numbers represented in a **base-10** *Hindu-Arabic* or the *Decimal* numeral system. That is, when you see 2531 you think "2 thousands, 5 hundreds, 3 tens, and 1 one." Rewritten, this is:

$$2 \times 10^3 + 5 \times 10^2 + 3 \times 10^1 + 1 \times 10^0$$

The *radix* is the amount of number of unique values a base can have before increasing the amount of digits. For example, values 0, 1, 2, $\cdots$, 8, 9 are all the values for base-10. Once 1 is added added to 9, we need to increase the amount of digits in a number.

In any Hindu-Arabic system, the location of the digits is meaningful: 101 is different from 110; the position of a number indicates it's magnitude. Here are those numbers in Roman numerals: CI and CX. Roman numerals didn't have a symbol for zero at all.

The Hindu-Arabic system gave us really straightforward algorithms for addition and multiplication. For addition, you memorized the following table:

	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	10
2	2	3	4	5	6	7	8	9	10	11
3	3	4	5	6	7	8	9	10	11	12
4	4	5	6	7	8	9	10	11	12	13
5	5	6	7	8	9	10	11	12	13	14
6	6	7	8	9	10	11	12	13	14	15
7	7	8	9	10	11	12	13	14	15	16
8	8	9	10	11	12	13	14	15	16	17
9	9	10	11	12	13	14	15 16	17	18	

When you multiplied two number together, you simply multiplied each pair of digits. 254×26 might look like this:

		2	5	4	
		×	2	6	
			2	4	6×4
		3	0		6×5
	1	2			6×2
			8		2×4
	1	0			2×5
+	4				2×2
6	6	0	4		

For multiplication, you memorized this table:

	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	10	12	14	16	18
3	0	3	6	9	12	15	18	21	24	27
4	0	4	8	12	16	20	24	28	32	36
5	0	5	10	15	20	25	30	35	40	45
6	0	6	12	18	24	30	36	42	48	54
7	0	7	14	21	28	35	42	49	56	63
9	0	9	18	27	36	45	54	63	72	81

4.2 Base-2 (Binary)

Binary, on the other hand, has only 2 digits it can have: 0 or 1. This kind of number system is often used for electrical and computer systems because the values can only be **on** or **off**.

Each digit in a binary number represents a power of 2, starting from the rightmost digit (the least significant bit, or LSB). For example:

$$\cdots + 2^3 + 2^2 + 2^1 + 2^0$$

For example,

$$1011_2 = (1 \cdot 2^3) + (0 \cdot 2^2) + (1 \cdot 2^1) + (1 \cdot 2^0) = 8 + 0 + 2 + 1 = 11_{10}$$

[1]

[1] The subscripts represent the radix or base.

Because the radix is 2, the numbers increase differently:

$$
\begin{aligned}
&0 : 0 \\
&1 : 1 \\
&2 : 10 \\
&3 : 11 \\
&4 : 100 \rightarrow (2^2 \cdot 1) + 0 + 0 = 4 \\
&5 : 101 \\
&6 : 110 \\
&7 : 111 \\
&8 : 1000 \\
&\dots \\
&15 : 1111 \cdots
\end{aligned}
$$

Conversion from Decimal to Binary

To convert 13_{10} to binary, repeatedly divide by 2 and record the remainders until the divisor reaches 0 or 1:

$13 \div 2 = 6$ remainder $16 \div 2 = 3$ remainder $03 \div 2 = 1$ remainder $11 \div 2 = 0$ remainder 1

Reading from bottom to top, $13_{10} = 1101_2$.

4.2.1 Bits and Bytes

A singular binary digit (0 or 1) is called a *bit*. A lightbulb, switch, or any sort of *Boolean* value (true or false) can be represented as a bit. A single binary digit is called a *bit*. Eight bits form a *byte*, which is a common unit of storage in computer systems:

$$1 \text{ byte} = 8 \text{ bits}$$

For example, the binary sequence 01001000 represents one byte of data.

We won't go into the depths of it here, but here are the basics of computer architecture. There are codes of 4-bit sequences that form up the *memory*. This can form the sections of it into different data types:

Common Data Types in Memory

Data Type	Size	Bit Length	Typical Use
Character	1 byte	8 bits	ASCII characters, small integers
Integer	4 bytes	32 bits	Whole numbers
Double	8 bytes	64 bits	Decimal (floating-point) values

Strings are formed by listing characters in consecutive memory addresses such that they are marked by a starting memory address and ending memory address.

8-bit binary numbers can be used to signify positive or negative numbers. The most significant bit (MSB) will alter sign, causing the range to be $[-2^7,\ 2^7 - 1]$. If the byte is negative (in terms of bits), invert each bit and add 1 to the inverted result. The other way is to multiply the MSB by negative one, and add until the value is reached, as seen below.

$$-85_{10} = -1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 10101011$$

Binary can be easy converted to *Hexadecimal*.

4.3 Base-16 (Hexadecimal)

Hexadecimal (from the latin *hexa* and *deca* for 16) is a representation of number with a radix of 16. Digits are represented through numeric numbers 0-9 and *A-F*.

Counting to 16 in hex goes as follows:

1
2
3
...
9
A (decimal value 10)
B (decimal value 11)
C (decimal value 12)
D (decimal value 13)
E (decimal value 14)
F (decimal value 15)

Converting between binary and hexadecimal is easier because the hex radix ($16 = 2^4$) is a power of the binary radix (2^1).

It takes 4 binary digits to represent 1 hex digit: 15 = F

For example,

$$0x47 = 0100\ 0111_2$$

$$0xBD = 1011\ 1101_2$$

4.3.1 Hex Color Codes

Hex colors are represented by 6 hexadecimal digits, grouped as two each for red, green, and blue (RGB). Each pair ranges from 00 (0 in decimal) to FF (255 in decimal), allowing 256 shades per color channel. For example:

$$\#FF0000 = \text{Red}, \quad \#00FF00 = \text{Green}, \quad \#0000FF = \text{Blue}$$

By mixing values, you can create millions of unique colors, e.g.:

$$\#FFA500 = \text{Orange}, \quad \#800080 = \text{Purple}, \quad \#FFFFFF = \text{White}, \quad \#000000 = \text{Black}$$

There are $256^3 = 16{,}777{,}216$ possible color combinations in the RGB hex system. The higher the red, green, or blue values are (closer to 255), the brighter the component becomes and the overall color shifts closer to white. Conversely, the lower the values are (closer to 0), the darker the component becomes and the overall color shifts closer to black.

Included in your digital resources are multiple hex videos, as well as an interactive hex color guessing game.

4.4 Base-8 (Octal)

Octal is another representation of radix 8. Digits range from 0 to 7, and the number of digits increase the threshold is passed.

Just as in the above systems, octal increases by the radix —– powers of 8. For example,

$$135_8 = (1 \times 8^2) + (3 \times 8^1) + (5 \times 8^0) = 64 + 24 + 5 = 93_{10}$$

4.4.1 Why Octal Matters

Octal was historically important in computing because early computers often worked with word sizes that were multiples of 3 bits (such as 12, 24, or 36). Since each octal digit corresponds exactly to 3 binary digits, it was a convenient shorthand for binary values before hexadecimal became more widespread. For example:

$$110101_2 = 65_8$$

Even though modern systems mostly use hexadecimal, octal is still used in some contexts, such as Unix file permissions (e.g., `chmod 755`).

Chapter 5

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 4

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.

5.1 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.

5.1.1 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
};
```

5.1.2 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);
```

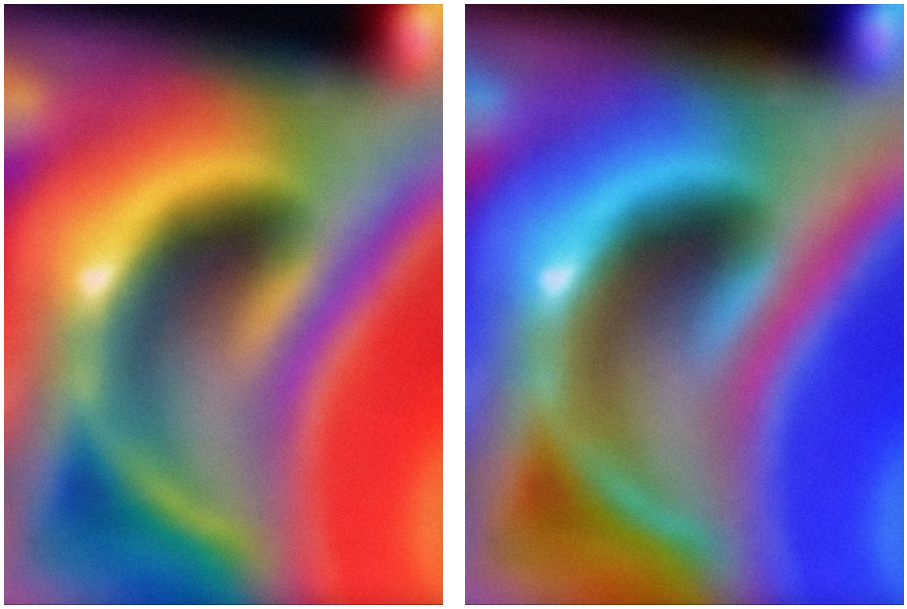
And then, we can loop through to the `w` and `h` to analyze each individual pixel. What we will do as a first test of pixel alteration is *swap red and blue values* in our output.

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

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

Figure 5.2: 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 2 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 45

5.2 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:

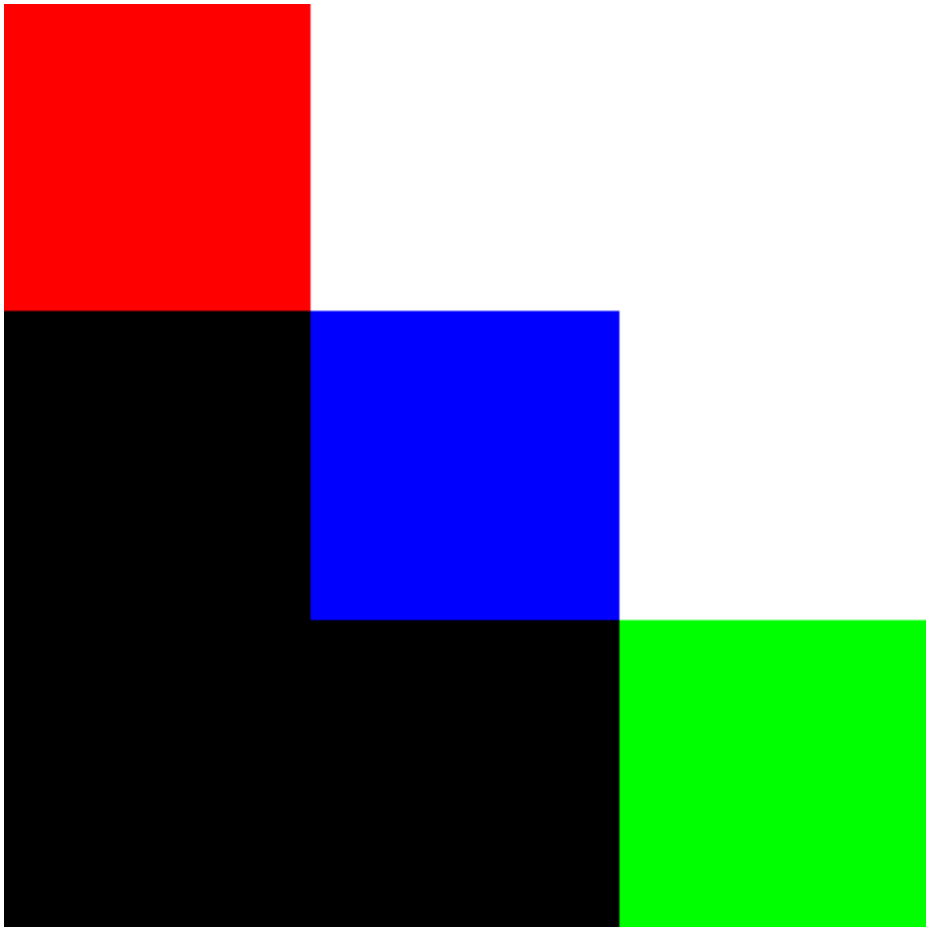


Figure 5.4: Your custom bitmap!

Exercise 3 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 46

5.3 Connection to transformations

What if alter the pixel values in a more complex way? For example, instead of altering the colors themselves, we alter the pixel positions. This is what is called a transfomation, and is a fundamental concept in computer graphics. A transformation is a mathematical operation that changes the position, size, or orientation of an object in a graphical space. In the context of bitmaps, transformations can be applied to the pixel data to achieve various effects, such as rotation, scaling, translation, and shearing.

What if you took a selfie, but then wanted to mirror it horizontally, to get a more representation of your face? This is a common transformation called a horizontal flip. To achieve this, you would iterate through each row of pixels in the bitmap and swap the pixels on the left side with the corresponding pixels on the right side. This can be done by calculating the index of each pixel and performing the necessary swaps. Note that nothing on the vertical direction of the image is changed, only the horizontal positions of the pixels are altered.

Since we are storing our pixel data as (x, y) coordinates, we can easily apply transformations by manipulating these coordinates. For an image with width w and height h, the horizontal flip transformation can be expressed mathematically as:

$$x' = w - 1 - x, \qquad y' = y$$

This operation does not change the pixel values themselves, but instead sets the new position of each pixel according to the transformation rule. The entire image transformation is therefore a systematic coordinate mapping applied across the matrix.

Without having to learn to much linear algebra, we can express this as a simple function: If we have a pixel coordinate expressed as a simple vector $\mathbf{p} = \begin{bmatrix} x \\ y \end{bmatrix}$, we can define a transformation function T that takes this pixel coordinate and maps it to a new coordinate $\mathbf{p}'$:

$$\mathbf{p}' = T\mathbf{p}$$

In this case, the transformation function T is a matrix that encodes the horizontal flip operation that obeys matrix multiplication. We have not talked about this yet, and it is a difficult concept to understand without having introduced linear algebra or matrices. So, we will not go into the details of how to construct this matrix, but just know that a horizontal flip can represented as:

$$T = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$$

This matrix, when multiplied with the pixel coordinate vector $\mathbf{p}$, will yield the new coordinate $\mathbf{p}'$ that corresponds to the horizontally flipped position of the original pixel.

Specifically, the -1 in the first row of the matrix indicates that the x-coordinate is inverted (flipped), while the 1 in the second row indicates that the y-coordinate remains unchanged.

Let's do this with code! The only thing that changes in our for loop is the calculation of the output pixel's position. Instead of writing to the same index, we write to the flipped index:

```
for (int y = 0; y < h; y++)
{
    for (int x = 0; x < w; x++)
    {
        int newX = w - 1 - x; // Calculate the new x-coordinate for the
        ↪ horizontal flip
        int srcIdx = y * rowSize + x * 3;
        int dstIdx = y * rowSize + newX * 3;

        // read the pixel values from the source index
        BYTE B = data[srcIdx];
        BYTE G = data[srcIdx + 1];
        BYTE R = data[srcIdx + 2];

        //output the pixel values to the destination index
        out[dstIdx + 0] = B;
        out[dstIdx + 1] = G;
        out[dstIdx + 2] = R;
    }
}
```

Taking the input bitmap of the cow from 5.2a and running this code produces the horizontally flipped output in 5.6.

Figure 5.6: Output of a horizontally flipped cow bitmap after applying the new pixel position transformation.

Similarly, we can transform the bitmap in other ways, which we will learn more of how to do in the transformations chapter by experimenting with vectors. For now, remember that matrices are a powerful tool for representing transformations, and that by manipulating pixel coordinates, we can achieve a wide variety of effects on bitmap images.

5.4 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 8)

The two triangles are similar; one is 2 m and 3m, the other is x cm and 3 cm.

The image of the cow is 2 cm tall.

Answer to Exercise 2 (on page 39)

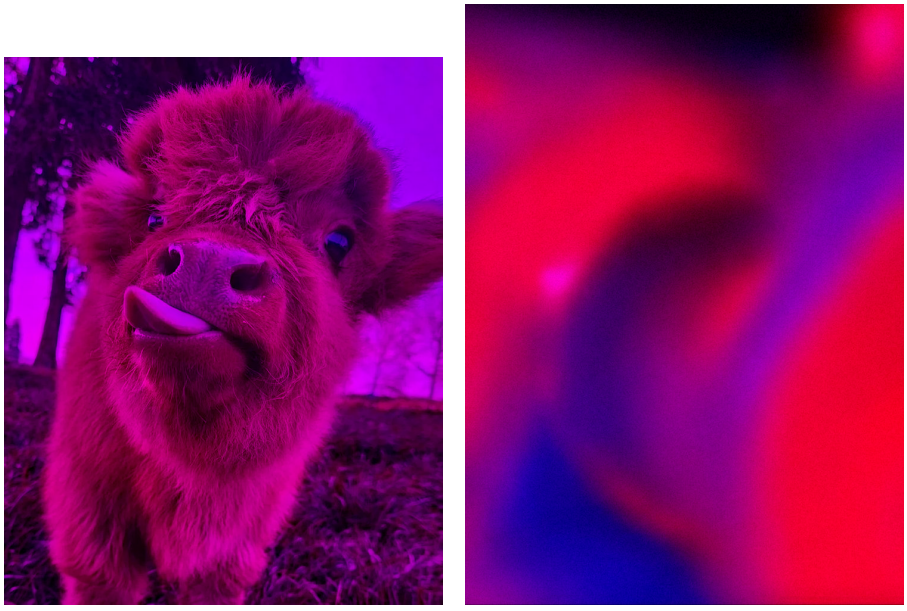
The only thing that changes in our for loop is the writing of the output pixel.

```
// 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 5.3a and 5.3b), 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 5.2a and 5.1a for the original bitmaps.



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

Figure 5.3: Running `bitmap_no_green.cpp` on both provided bitmaps.

Answer to Exercise 3 (on page 42)

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`×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 × 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:

```
#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;
}
```

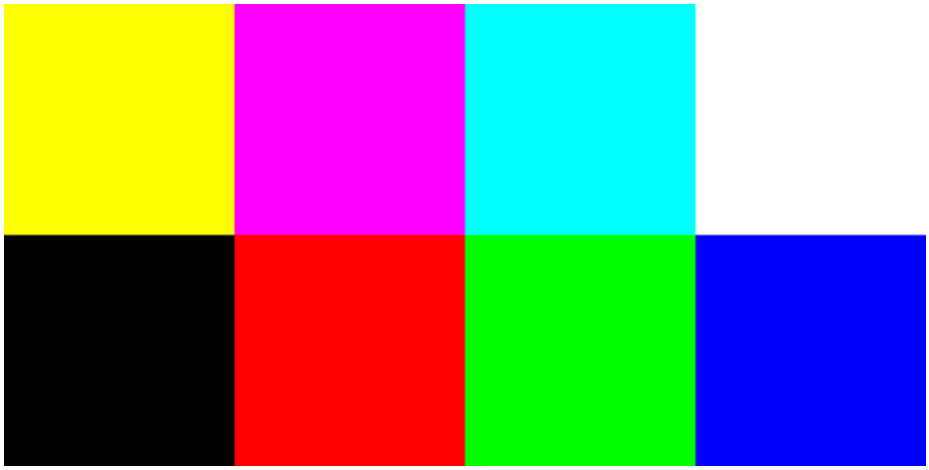
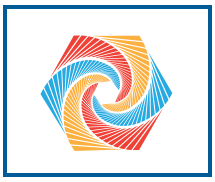


Figure 5.5: Exercise Output of a 4 by 2 bitmap.



Index

aspect ratio, 10

base-2, 28
binary, 28
    counting in, 29
bit, 29
bitmaps, 33
    structure of, 34
byte, 29

camera, 3
camera
    lens, 8
    lens
        focal length, 10
    pinhole, 7
cmyk, 19
color, 16
computer architecture, 29

eye, 13
    astigmatism, 15
    cataract, 15
    farsightedness, 15
    glaucoma, 14
    iris, 13
    myopia, 15
    nearsightedness, 15
    pupil, 13
    retina, 14
    sclera, 13

hex colors, 31
hexadecimal, 30, 31, 33

memory, 29

Octal, 31

padding, 36
photon, 4
    colors of, 6
    wavelength, 6

rgb, 10, 18

segmentation fault, 36
signed bits, 30