



COMP120: Creative Computing: Tinkering

4: Tinkering Graphics I

Learning Outcomes

By the end of this workshop, you should be able to:

- ▶ **Explain how** pictures are digitised into raster images by a computer system
- ▶ **Apply** knowledge of colour models to **write** code that manipulates pixels in a surface
- ▶ **Use** functions, arguments, and basic data structures such as arrays

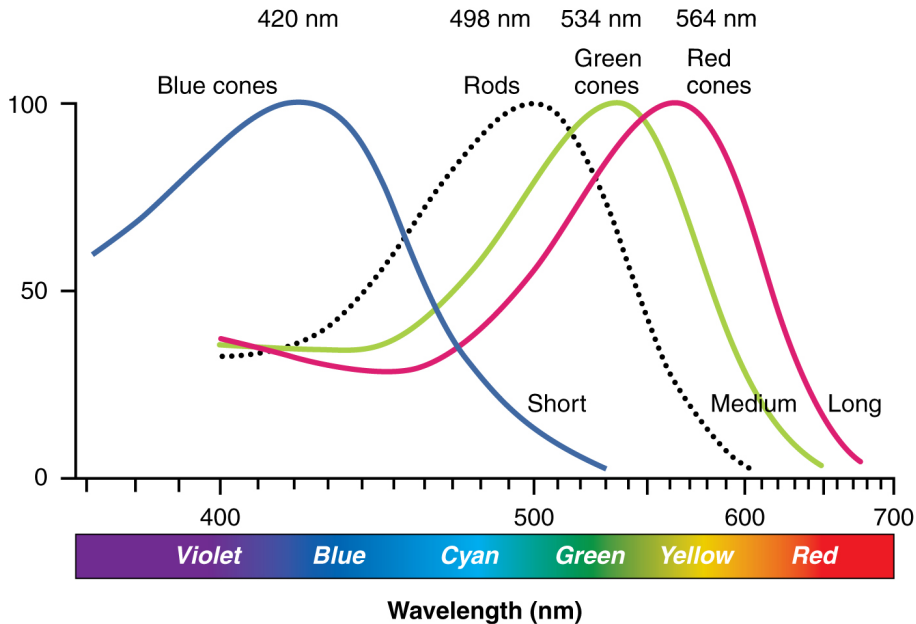
Light Perception

- ▶ Colour is continuous:
 - ▶ Visible light is in the wavelengths between 370nm and 730nm
 - ▶ i.e., 0.00000037 — 0.00000073 meters
- ▶ However, we *perceive* light around three particular peaks:
 - ▶ Blue peaks around 425nm
 - ▶ Green peaks around 550nm
 - ▶ Red peaks around 560nm

Light Perception

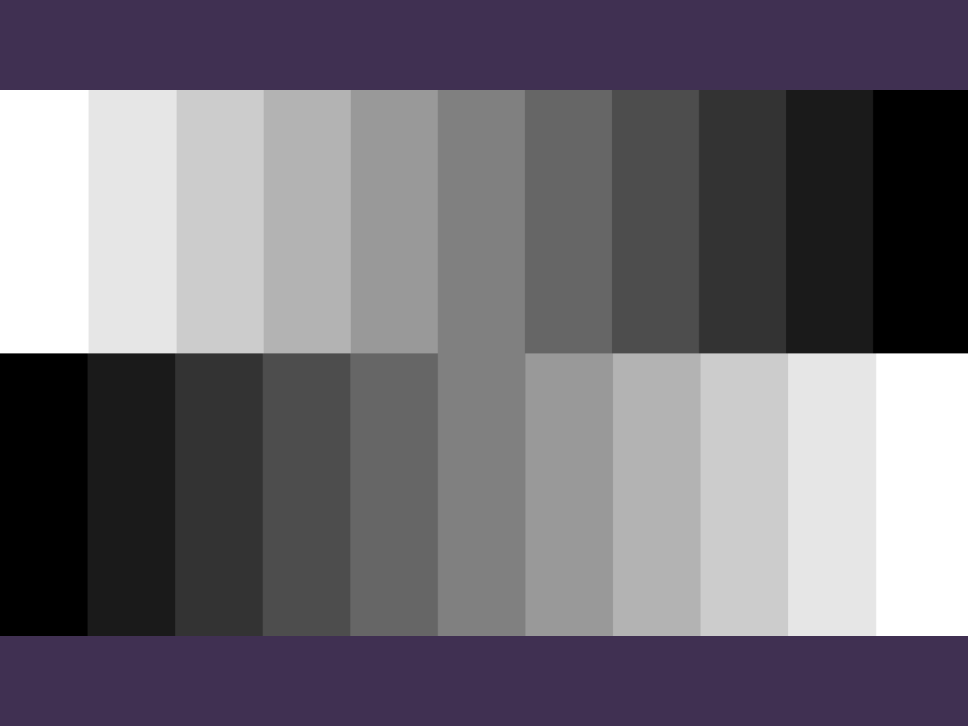
- ▶ Our eyes have three types of colour-sensitive photoreceptor cells called 'cones' that respond to light wavelengths
- ▶ Our perception of colour is based on how much of each kind of sensor is responding
- ▶ An implication of this is perception overlap: we see two kinds of 'orange' — one that's spectral and one that's combinatorial

Normalized absorbance



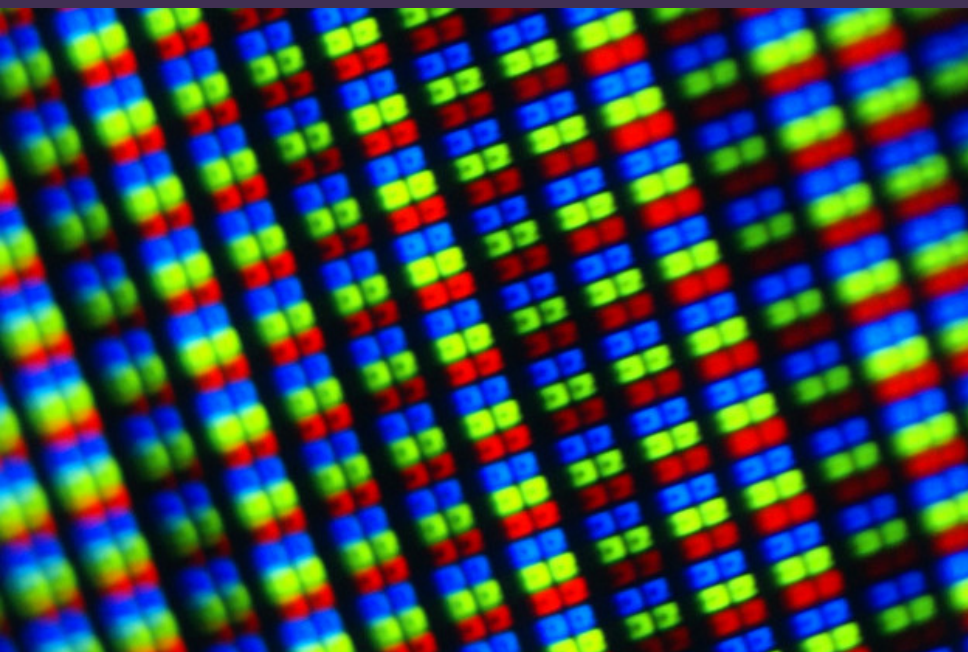
Luminance vs Colour

- ▶ Our eyes have another type of photoreceptor cells called 'rods' that respond to light intensity
- ▶ Our perception, however, is actually luminance: a relativistic contrast of *borders* of things (i.e., motion)
 - ▶ Luminance is *not* the amount of light, but our perception of the amount of light
 - ▶ Much of our luminance perception is based on comparison to background, not raw values
- ▶ An implication of this is perception overlap: we see blue as 'darker' than red when the intensity is actually the same



Resolution

- ▶ We have a limited number of rods and cones in our eyes
- ▶ This means humans perceive vision in a limited resolution — yet, we perceive vision as continuous
- ▶ We take advantage of this human characteristic in computer monitors



Pixels

- ▶ We digitize pictures into many little dots
- ▶ Enough dots and it looks like a continuous whole to our eye
- ▶ Each element is referred to as a *pixel*

Pixels

Pixels must have:

- ▶ a color
- ▶ a position

Pictures and Surfaces

A picture is a *matrix* of pixels

- ▶ It is not a continuous line of elements, that is, a one-dimensional *array*
- ▶ A picture has two dimensions: width and height
- ▶ It's a two-dimensional *array*

0

1

2

3

15

12

13

10

	0	1	2	3
0	15	12	13	10
1	9	7	2	1
2	6	3	9	10

Pictures and Surfaces








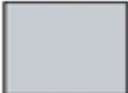
- ▶ (x, y) —or— (horizontal, vertical)
- ▶ $(1, 0) = 12$
- ▶ $(0, 2) = 6$

Encoding Colour

- ▶ Each element in the matrix is a pixel, with the matrix defining its position and the value defining its colour
- ▶ Computer memory stores numbers, so colour must be encoded into a number:
 - ▶ CMYK = cyan, magenta, yellow, black
 - ▶ HSB = hue, saturation, brightness
 - ▶ RGBA = red, green, blue, alpha (transparency)
- ▶ By default, PyGame uses RGBA

Encoding RGB

- ▶ Each component color (red, green, and blue) is encoded as a single byte
- ▶ Colors go from $(0, 0, 0)$ to $(255, 255, 255)$:
 - ▶ If all three components are the same, the colour is in grey-scale
 - ▶ $(0, 0, 0)$ is black
 - ▶ $(255, 255, 255)$ is white
- ▶ Why 255?

	0	1	2	3
0	 255, 30, 30	 30, 30, 255	 30, 255, 30	 0, 0, 0
1	 255, 150, 150	 150, 150, 255	 150, 255, 150	 200, 200, 200

Encoding Bits

- ▶ If we have one bit, we can represent two patterns:
 - ▶ 0
 - ▶ 1
- ▶ If we have two bits, we can represent two patterns:
 - ▶ 00
 - ▶ 01
 - ▶ 10
 - ▶ 11
- ▶ As a general rule: In n bits, we can have 2^n patterns
- ▶ One of these patterns will be 0, so the highest value we can represent is: $2^8 - 1$, or 255

Encoding Bits

- ▶ RGB uses 24-bit color (i.e., $3 * 8 = 24$)
 - ▶ That's 16,777,216 (2^{24}) possible colours
 - ▶ Our eyes cannot discern many colours beyond this
 - ▶ The big issue is the monitor: they can't reliably reproduce 16 million colours
- ▶ RGBA uses 32-bit colour
 - ▶ No additional colour, but offers support for transparency

Encoding Bits

- ▶ Use this information to estimate the size of a bitmap:
 - ▶ $320 \times 240 \times 24 = 230,400$ bytes
 - ▶ $640 \times 480 \times 32 = 1,228,800$ bytes
 - ▶ $1024 \times 768 \times 32 = 3,145,728$ bytes
- ▶ Why do we have smaller numbers here?

Activity #1

In pairs:

- ▶ Setup a basic project in PyGame
- ▶ Refer to the following documentation
 - ▶ `www.pygame.org/docs/ref/surface.html`
 - ▶ `www.pygame.org/docs/ref/pixelarray.html`
- ▶ Manipulate the pixels in a surface using PixelArray
- ▶ 10 minutes

Make a Picture Less Red

```
def decreaseRed(pict):  
    pixelMatrix = getPixels(pict)  
    for pixel in pixelMatrix:  
        value = getRed(pixel)  
        setRedPixel(pixel, value * 0.5)
```

Note: This source code excerpt will not work in PyGame.

Activity #2

In pairs:

- ▶ Define a function that turns all of the red values of pixels into blue values...
- ▶ ...and all of the blue values into red values

Swap Channels

```
def swapRedBlueChannels(pict):  
    pixelMatrix = getPixels(pict)  
    for pixel in pixelMatrix:  
        red_value = getRed(pixel)  
        blue_value = getBlue(pixel)  
        setRedPixel(pixel, blue_value)  
        setBluePixel(pixel, red_value)
```

Note: This source code excerpt will not work in PyGame.

Activity #3

In pairs:

- ▶ Refer to the following documentation:
 - ▶ `//www.pygame.org/docs/ref/image.html`
- ▶ Define a function that loads an image and turns it to greyscale

Make a Picture Grey-scale

```
def loadGrayscale(file):  
    pixelMatrix = getPixels(makePicture(file))  
    for pixel in pixelMatrix:  
        red = getRed(p)  
        green = getGreen(p)  
        blue = getBlue(p)  
  
        pixelValue = (red+green+blue)/3  
  
        setRedPixel(pixel, pixelValue)  
        setGreenPixel(pixel, pixelValue)  
        setBluePixel(pixel, pixelValue)
```

Note: This source code excerpt will not work in PyGame.

Activity #4

In pairs:

- ▶ Refer to the following documentation:
 - ▶ `//www.pygame.org/docs/ref/image.html`
- ▶ Define a function that loads an image and turns it to its negative

Make a Picture Negative

```
def neg(picture):  
    pixelMatrix = getPixels(makePicture(file))  
    for pixel in pixelMatrix:  
        red = getRed(p)  
        green = getGreen(p)  
        blue = getBlue(p)  
  
        setRedPixel(pixel, 255-red)  
        setGreenPixel(pixel, 255-green)  
        setBluePixel(pixel, 255-blue)
```

Note: This source code excerpt will not work in PyGame.

Activity #5

In pairs:

- ▶ Refer to the following documentation:
 - ▶ `//www.pygame.org/docs/ref/time.html`
- ▶ Define a function that loads an image and animates a sunset

Decrease Luminance Over Time

```
def decreaseRed(picture, amount):  
    for p in getPixels(picture):  
        value=getRed(p)  
        setRed(p,value*amount)  
  
amount = 0.1 #tinker with this value  
wait_time = 50 #tinker with this value  
  
for i in range(10):  
    decreaseRed(picture, amount)  
    decreaseGreen(picture, amount)  
    decreaseBlue(picture, amount)  
    wait(50)
```

Note: This source code excerpt will not work in PyGame.

Activity #6

In pairs:

- ▶ Refer to the following documentation:
 - ▶ `https://docs.python.org/2/tutorial/introduction.html#lists`
- ▶ Define a function that animates copying the top half of a picture to its bottom half

Copying Part of a Picture

```
def copyHalf(picture):  
    pixels = getPixels(picture)  
    for index in range(0, len(pixels)/2):  
        sourcePixel = pixels[index]  
        sourceRGBValue = getColor(sourcePixel)  
        destinationPixel = pixels[index + len(pixels)/2]  
        setColor(destinationPixel, sourceRGBValue)  
    repaint(picture)
```

Note: This source code excerpt will not work in PyGame.