CS 2984
Media Computation

Steve Harrison

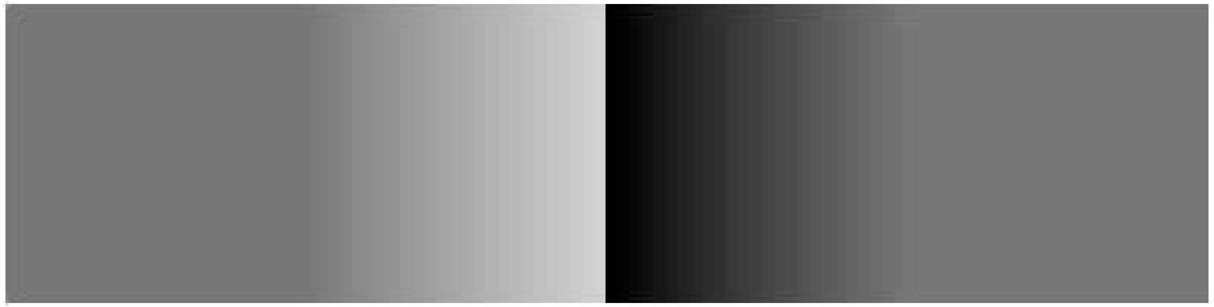
Lecture 2.1 (September 1, 2008)

We perceive light different from how it actually is

- Color is continuous
  - **Visible light is in the wavelengths between 370 and 730 nanometers**
    - That’s 0.00000037 and 0.00000073 meters
- But we *perceive* light with color sensors that peak around 425 nm (blue), 550 nm (green), and 560 nm (red).
    - Our brain figures out which color is which by figuring out how much of each kind of sensor is responding
    - One implication: We perceive two kinds of “orange” — one that’s *spectral* and one that’s red+yellow (hits our color sensors just right)
    - Dogs and other simpler animals have only two kinds of sensors
      - **They do see color. Just less color.**

Luminance vs. Color

- We perceive borders of things, motion, depth via *luminance*
  - **Luminance is not the amount of light, but our perception of the amount of light.**
  - **We see blue as "darker" than red, even if same amount of light.**
- Much of our luminance perception is based on comparison to backgrounds, not raw values.



Luminance is actually *color blind*. Completely different part of the brain.

Digitizing pictures as bunches of little dots

- We digitize pictures into lots of little dots
- Enough dots and it looks like a continuous whole to our eye
  - **Our eye has limited resolution**
  - **Our background/depth acuity is particularly low**
- Each picture element is referred to as a *pixel*

from Friday: The Wooden Mirror

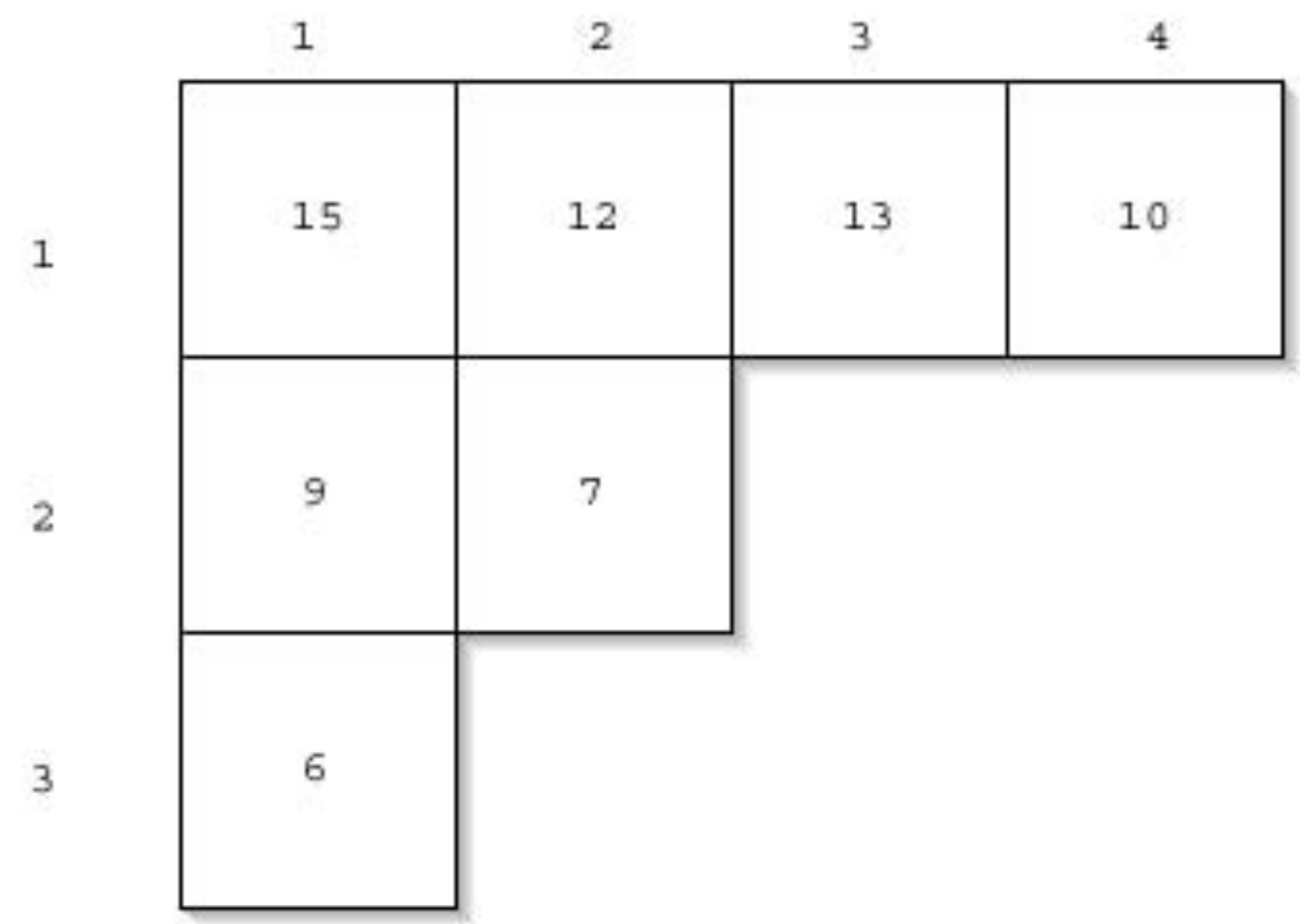
- Video
- How does it work?
- Color ?
- *Look up "DLP"*

Pixels

- Pixels are *picture elements*
  - **Each pixel object knows its color**
  - **It also knows where it is in its picture**

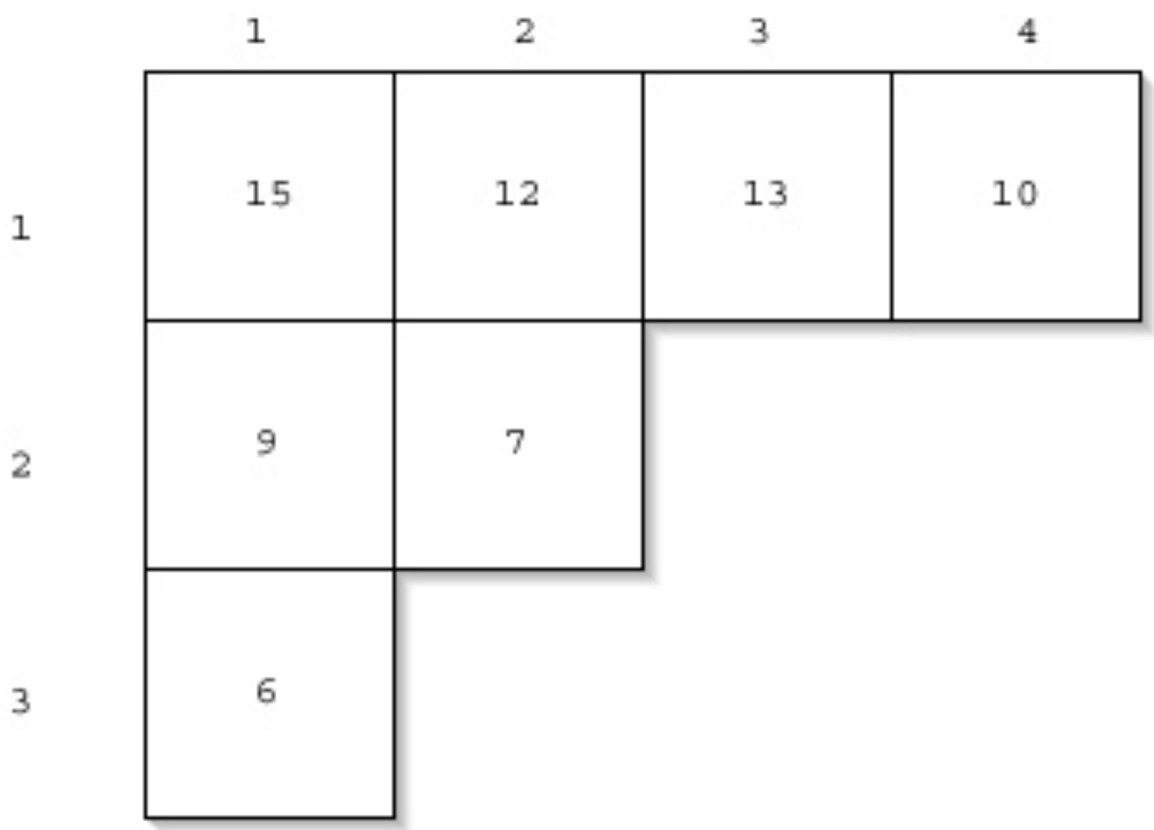
A Picture is a matrix of pixels

- It’s not a continuous line of elements, that is, an *array*
- A picture has two dimensions: Width and Height
- We need a two-dimensional array: a *matrix*



Just the upper left hand corner of a matrix.

Referencing a matrix



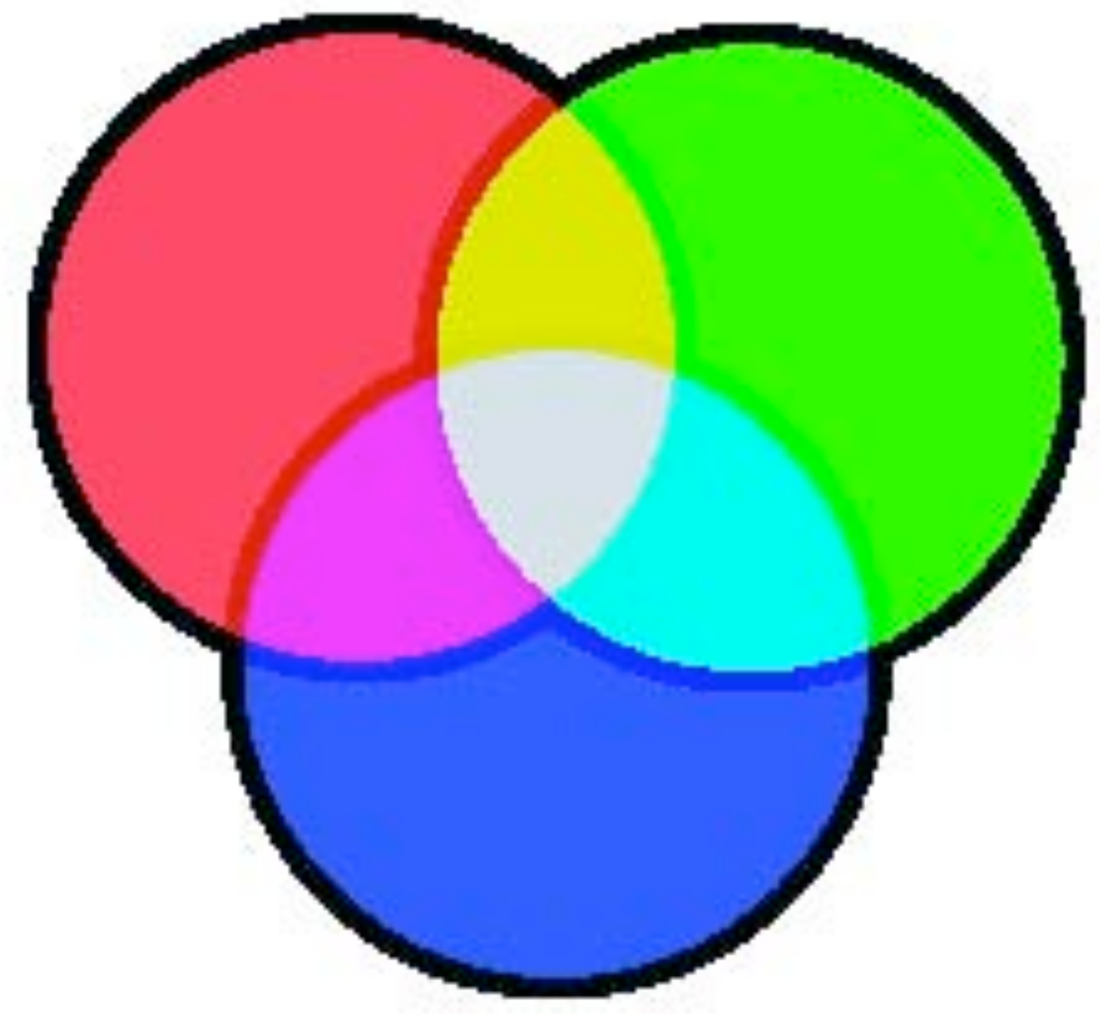
- We talk about positions in a matrix as (x,y), or (horizontal, vertical)
- location (1,1) is the upper left corner
- Element (2,1) in the matrix at left is the value 12
- Element (1,3) is 6

Encoding color

- Each pixel encodes color at that position in the picture
- Lots of encodings for color
  - **Printers use CMYK: Cyan, Magenta, Yellow, and blacK.**
  - **Others use HSB for Hue, Saturation, and Brightness (also called HSV for Hue, Saturation, and Brightness**
- We'll use the most common for computers
  - **RGB: Red, Green, Blue**

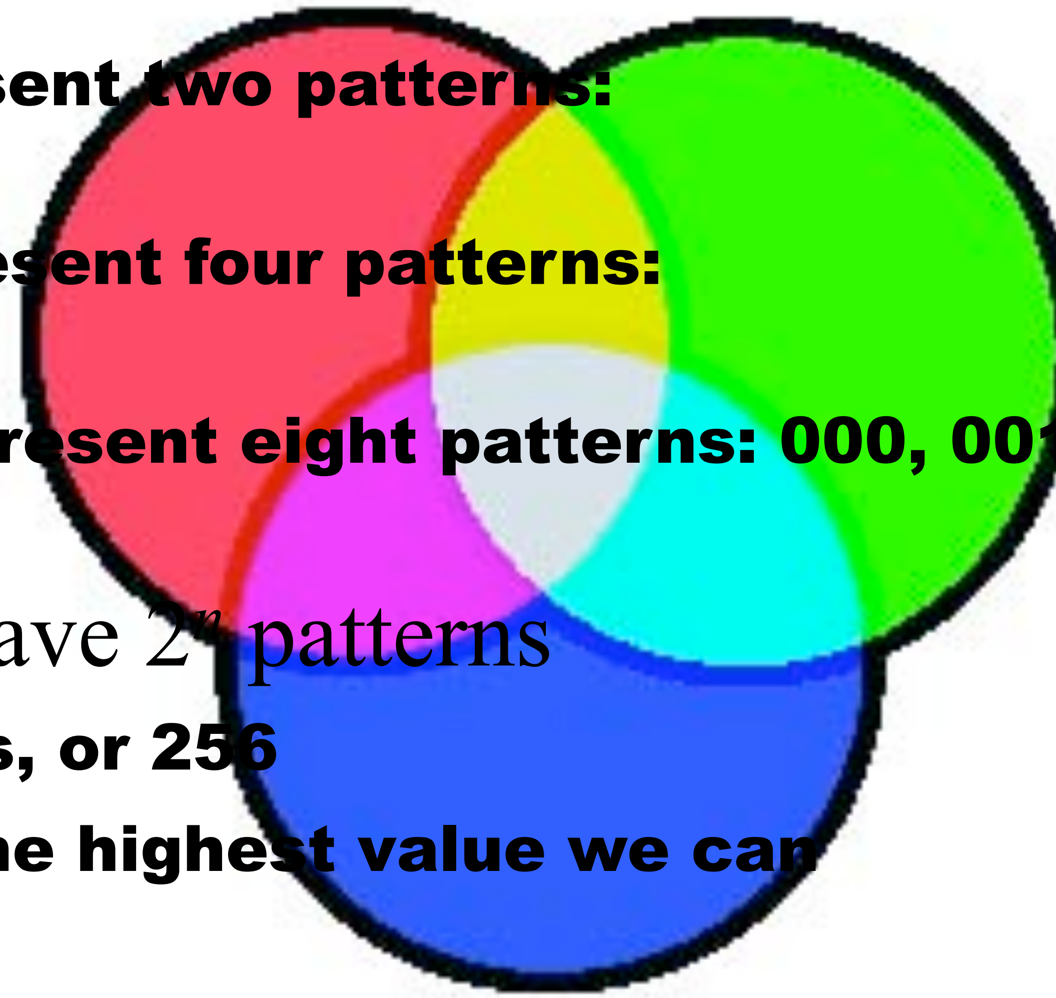
RGB

- In RGB, each color has three component colors:
  - **Amount of redness**
  - **Amount of greenness**
  - **Amount of blueness**
- Each does appear as a separate dot on most devices, but our eye blends them.
- In most computer-based models of RGB, a single *byte* (8 bits) is used for each
  - **So a complete RGB color is 24 bits, 8 bits of each**



How much can we encode in 8 bits?

- Let's walk it through.
  - **If we have one bit, we can represent two patterns: 0 and 1.**
  - **If we have two bits, we can represent four patterns: 00, 01, 10, and 11.**
  - **If we have three bits, we can represent eight patterns: 000, 001, 010, 011, 100, 101, 110, 111**
- General rule: In *n* bits, we can have 2^n patterns
  - **In 8 bits, we can have 2^8 patterns, or 256**
  - **If we make one pattern 0, then the highest value we can represent is 2^8-1, or 255**
  - **Thus the range is from 0 to 255**

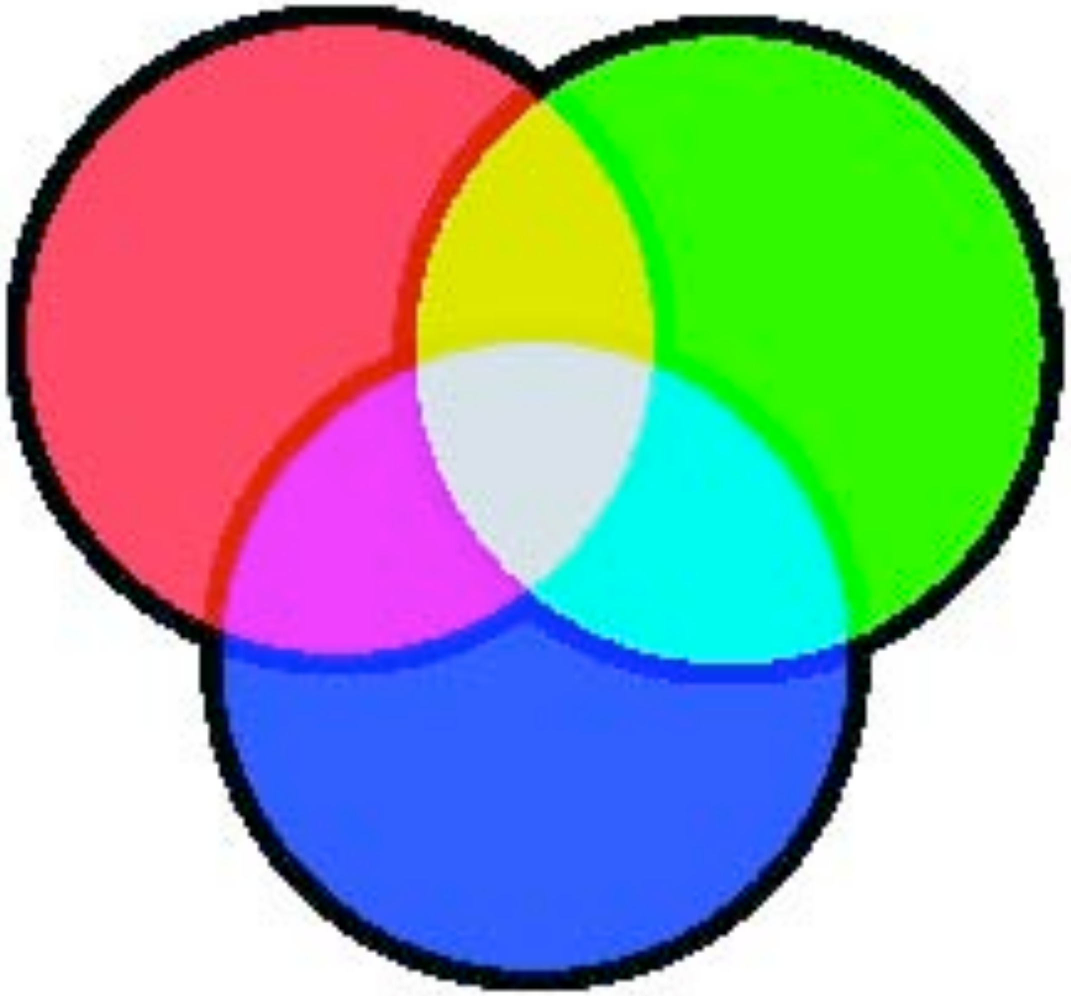


How much can we encode in 8 bits?

- Let's walk it through.
  - **If we have one bit, we can represent two patterns: 0 and 1.**
  - **If we have two bits, we can represent four patterns: 00, 01, 10, and 11.**
  - **If we have three bits, we can represent eight patterns: 000, 001, 010, 011, 100, 101, 110, 111**
- General rule: In *n* bits, we can have 2^n patterns
  - **In 8 bits, we can have 2^8 patterns, or 256**
  - **If we make one pattern 0, then the highest value we can represent is 2^8-1, or 255**
  - **Thus the range is from 0 to 255**

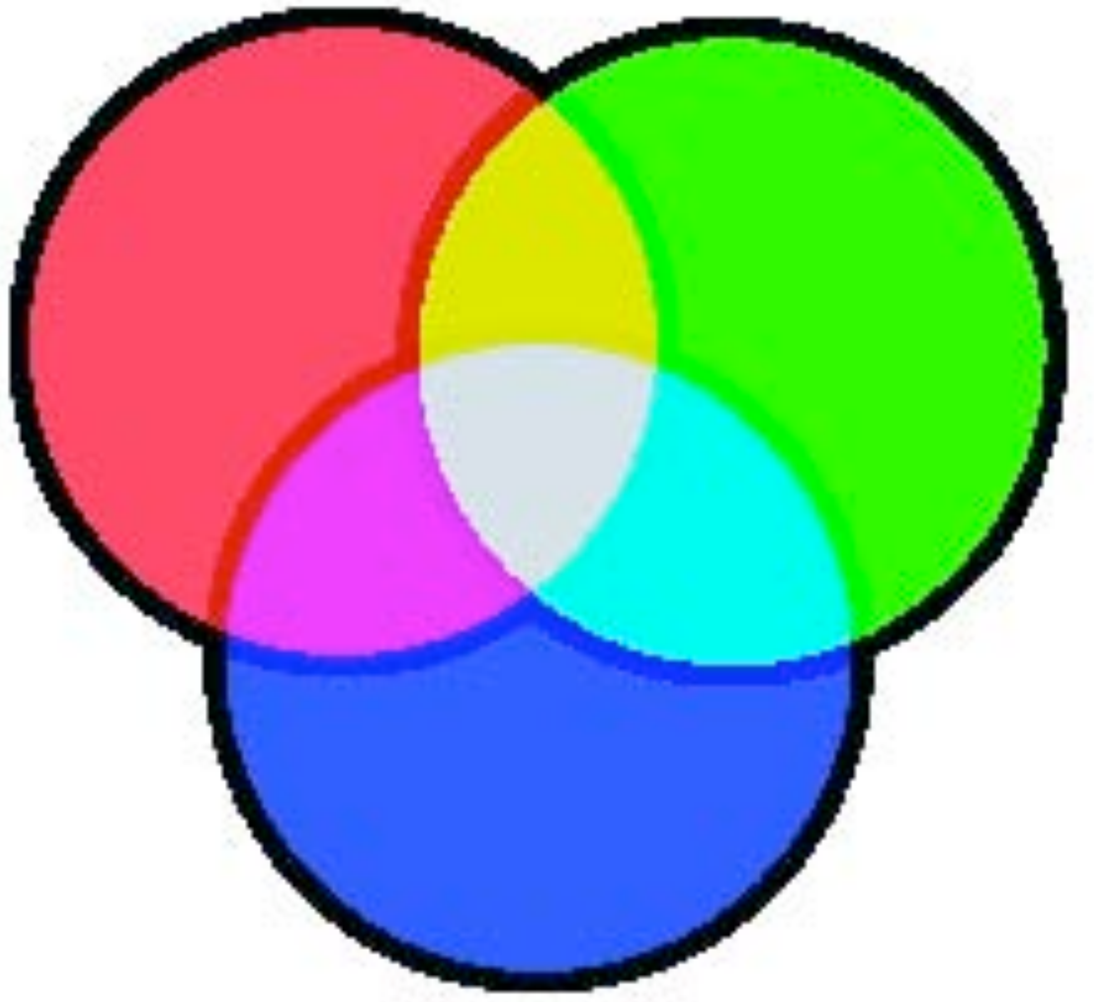
How much can we encode in 8 bits?

- Let's walk it through.
  - **If we have one bit, we can repr 0 and 1.**
  - **If we have two bits, we can rep 00, 01, 10, and 11.**
  - **If we have three bits, we can r 010, 011, 100, 101, 110, 111**
- General rule: In *n* bits, we can
  - **In 8 bits, we can have 2^8 patte**
  - **If we make one pattern 0, then represent is 2^8-1, or 255**
  - **Thus the range is from 0 to 255**



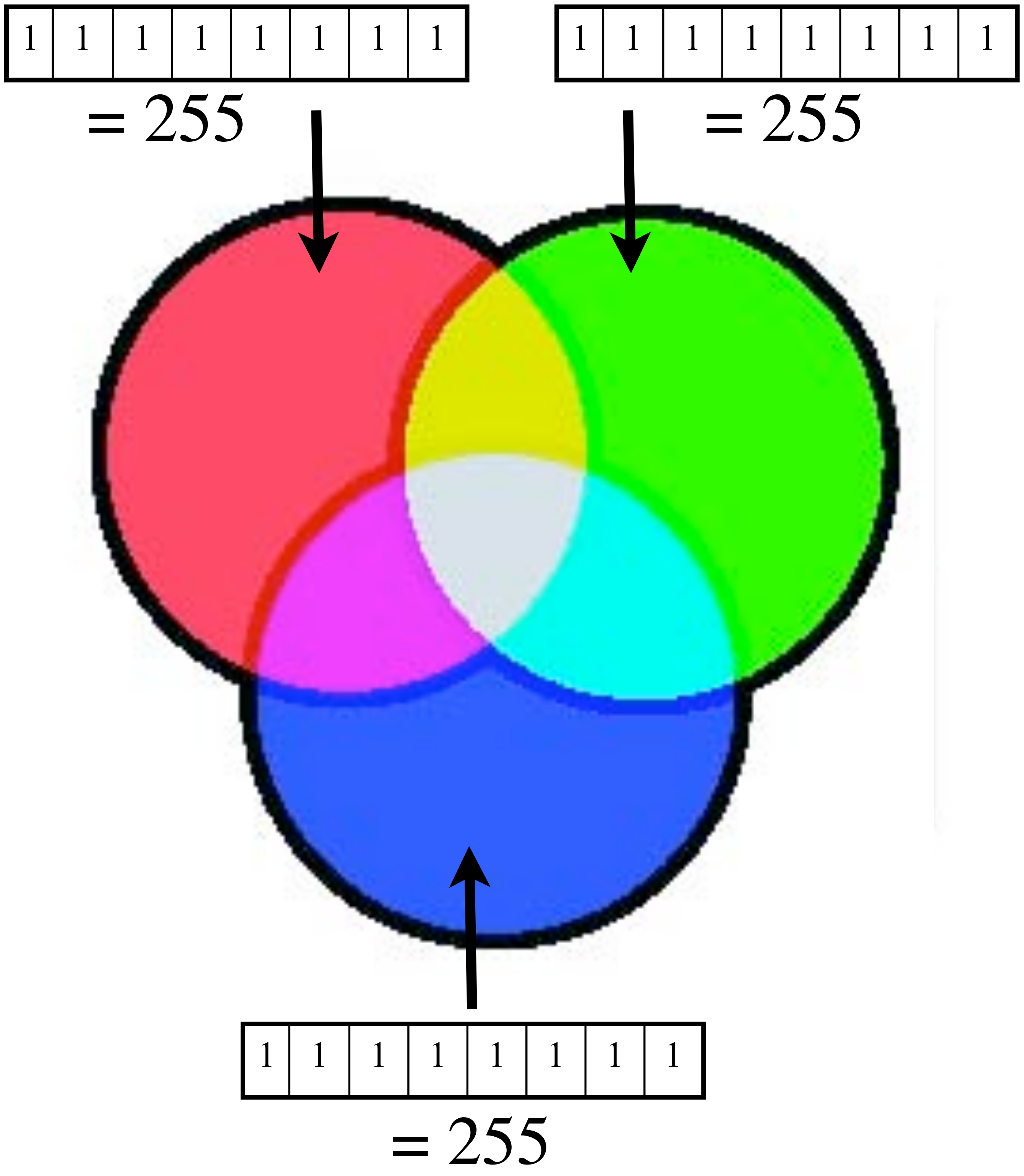
Encoding RGB

- 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 color is in greyscale**
    - (50,50,50) at (2,2)
  - **(0,0,0) (at position (1,2) in example) is black**
  - **(255,255,255) is white**



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 color is in greyscale**
    - (50,50,50) at (2,2)
  - **(0,0,0) (at position (1,2) in example) is black**
  - **(255,255,255) is white**



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 color is in greyscale**
    - (50,50,50) at (2,2)
  - **(0,0,0) (at position (1,2) in example) is black**
  - **(255,255,255) is white**

Another way to say 255…

- Some of you might have seen colors represented in hexadecimal: red = “ff”

1	1	1	1	1	1	1	1

- Its the same thing as 255
- 3 bits can represent 0 to 7
- 4 bits can represent 0 to 15
  - **And one byte is 8 bits which divides evenly into two groups of 4 bits**

1	1	1	1	1	1	1	1

  - **We then need a numbering system that goes from 0 to 16: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f**
  - **Hexadecimal means “base 16”**

Is that enough?

- We're representing color in 24 (3 * 8) bits.
  - That's 16,777,216 (2^{24}) possible colors
  - Our eye can discern millions of colors, so it's probably pretty close
  - But the real limitation is the physical devices: We don't get 16 million colors out of a monitor
- Some graphics systems support 32 bits per pixel
  - May be more pixels for color, or an additional 8 bits to represent 256 levels of translucence

Size of images

	320 x 240 image	**640 x 480 image**	**1024 x 768 monitor**
24 bit color	1,843,200 bytes	7,372,800 bytes	18,874,368 bytes
32 bit color	2,457,600 bytes	9,830,400 bytes	25,165,824 bytes

Reminder: Manipulating Pictures

```
>>> file=pickAFile()
>>> print file
/Users/guzdial/mediasources/barbara.jpg
>>> picture=makePicture(file)
>>> show(picture)
>>> print picture
Picture, filename /Users/guzdial/mediasources/barbara.jpg
height 294 width 222
```

What's a "picture"?

- An encoding that represents an image
  - **Knows its height and width**
  - **Knows its filename**
  - **Knows its window if it's opened (via show and repainted with repaint)**

Manipulating pixels

getPixel(picture,x,y) gets a single pixel.

getPixels(picture) gets *all* of them in an array. (Square brackets is a standard array reference notation—which we'll generally *not* use.)

```
>>> pixel=getPixel(picture,1,1)
>>> print pixel
Pixel, color=color r=168 g=131 b=105
>>> pixels=getPixels(picture)
>>> print pixels[0]
Pixel, color=color r=168 g=131 b=105
```

What can we do with a pixel?

- getRed, getGreen, and getBlue are functions that take a pixel as input and return a value between 0 and 255
- setRed, setGreen, and setBlue are functions that take a pixel as input *and* a value between 0 and 255

We can also get, set, and make Colors

- getColor takes a pixel as input and returns a Color object with the color at that pixel
- setColor takes a pixel as input *and* a Color, then sets the pixel to that color
- makeColor takes red, green, and blue values (in that order) between 0 and 255, and returns a Color object
- pickAColor lets you use a color chooser and returns the chosen color
- We also have functions that can makeLighter and makeDarker an input color

Demonstrating: Manipulating Colors

```
>>> print getRed(pixel)
168
>>> setRed(pixel,255)
>>> print getRed(pixel)
255
>>> color=getColor(pixel)
>>> print color
color r=255 g=131 b=105
>>> setColor(pixel,color)
>>> newColor=makeColor(0,100,0)
>>> print newColor
color r=0 g=100 b=0
>>> setColor(pixel,newColor)
>>> print getColor(pixel)
color r=0 g=100 b=0
```

```
>>> print color
color r=81 g=63 b=51
>>> print newcolor
color r=255 g=51 b=51
>>> print distance(color,newcolor)
174.41330224498358
>>> print color
color r=168 g=131 b=105
>>> print makeDarker(color)
color r=117 g=91 b=73
>>> print color
color r=117 g=91 b=73
>>> newcolor=pickAColor()
>>> print newcolor
color r=255 g=51 b=51
```

We can change pixels directly…

```
>>> file="/Users/guzdial/mediasources/barbara.jpg"
>>> pict=makePicture(file)
>>> show(pict)
>>> setColor(getPixel(pict,10,100),yellow)
>>> setColor(getPixel(pict,11,100),yellow)
>>> setColor(getPixel(pict,12,100),yellow)
>>> setColor(getPixel(pict,13,100),yellow)
>>> repaint(pict)
```

But that's *really* dull and boring…
That's the subject of the next lecture

Use a loop!
Our first picture recipe

```
def decreaseRed(picture):
  for p in getPixels(picture):
    value=getRed(p)
    setRed(p,value*0.5)
```

Used like this:

```
>>> file=pickAFile() <--- barbara.jpg
>>> picture=makePicture(file)
>>> show(picture)
>>> decreaseRed(picture)
>>> repaint(picture)
```

Once we make it work for one picture, it will work for any picture

Think about what we just did

- Did we change the program at all?
- Did it work for all the different examples?
- What was the input variable **picture** each time, then?
  - **It was the value of whatever picture we provided as input!**

```
def decreaseRed(picture):
  for p in getPixels(picture):
    value=getRed(p)
    setRed(p,value*0.5)
```

Read it as a Recipe

```
def decreaseRed(pict):
  for p in getPixels(pict):
    value=getRed(p)
    setRed(p,value*0.5)
```

Read it as a Recipe

```
def decreaseRed(pict):
  for p in getPixels(pict):
    value=getRed(p)
    setRed(p,value*0.5)
```

- Recipe: To decrease the red

Read it as a Recipe

```
def decreaseRed(pict):
  for p in getPixels(pict):
    value=getRed(p)
    setRed(p,value*0.5)
```

- Recipe: To decrease the red
- Ingredients: One picture, name it **pict**

Read it as a Recipe

```
def decreaseRed(pict):
  for p in getPixels(pict):
    value=getRed(p)
    setRed(p,value*0.5)
```

- Recipe: To decrease the red
- Ingredients: One picture, name it **pict**
- Step 1: Get all the pixels of **pict**. For each pixel **p** in the pixels…

Read it as a Recipe

```
def decreaseRed(pict):
  for p in getPixels(pict):
    value=getRed(p)
    setRed(p,value*0.5)
```

- Recipe: To decrease the red
- Ingredients: One picture, name it **pict**
- Step 1: Get all the pixels of **pict**. For each pixel **p** in the pixels…
- Step 2: Get the value of the red of pixel **p**, and set it to 50% of its original value

Let’s use something with known red to manipulate: Santa Claus

What if you decrease Santa's red again and again and again…?

```
>>> file=pickAFile()
>>> pic=makePicture(file)
>>> decreaseRed(pic)
>>> show(pic)
```

(That's the first one)

```
>>> decreaseRed(pic)
>>> repaint(pic)
```

(That's the second)

Increasing Red

```
def increaseRed(picture):
  for p in getPixels(picture):
    value=getRed(p)
    setRed(p,value*1.2)
```

What happened here?!?

Remember that the limit for redness is 255.

If you go *beyond* 255, all kinds of weird things can happen

How does increaseRed differ from decreaseRed?

- Well, it does increase rather than decrease red, but other than that…
  - **It takes the same input**
  - **It can also work for any picture**
    - It's a specification of a *process* that'll work for any picture
    - There's nothing specific to a specific picture here.

Clearing Blue

```
def clearBlue(picture):
  for p in getPixels(picture):
    setBlue(p,0)
```

Again, this will work for any picture.

Try stepping through this one yourself!

Creating a negative

- Let’s think it through
  - **R,G,B go from 0 to 255**
  - **Let’s say Red is 10. That’s very light red.**
    - What’s the opposite? LOTS of Red!
  - **The negative of that would be 245: 255-10**
- So, for each pixel, if we negate each color component in creating a new color, we negate the whole picture.

Recipe for creating a negative

```
def negative(picture):
  for px in getPixels(picture):
    red=getRed(px)
    green=getGreen(px)
    blue=getBlue(px)
    negColor=makeColor( 255-red, 255-green, 255-blue)
    setColor(px,negColor)
```

Original, negative, negative-negative

Converting to greyscale

- We know that if red=green=blue, we get grey
  - **But what value do we set all three to?**
- What we need is a value representing the darkness of the color, the *luminance*
- There are lots of ways of getting it, but one way that works reasonably well is dirt simple—simply take the average:

$$\frac{(red+green+blue)}{3}$$

Converting to greyscale

```
def greyScale(picture):
  for p in getPixels(picture):
    intensity = (getRed(p)+getGreen(p)+getBlue(p))/3
    setColor(p,makeColor(intensity,intensity,intensity))
```

Can we get back again?
Nope

- Converting to greyscale is different than computing a negative.
  - **A negative transformation retains information.**
- With greyscale, we've lost information
  - **We no longer know what the ratios are between the reds, the greens, and the blues**
  - **We no longer know any particular value.**

A comment about Comments

- Starting a line with a “#” makes jython ignore the rest of the line
- Comments are good -- in fact, essential -- to understanding a program
- Use them to explain what is happening, what a variable is supposed to have in it, etc.

Coming attractions

- Project 1 (on website for last week)
  - makeBandWNegative(aFileName)
  - due Friday @ 2:00 PM
  - about "langiappe" (a little bit extra)
    - must tell us! (# use a comment)