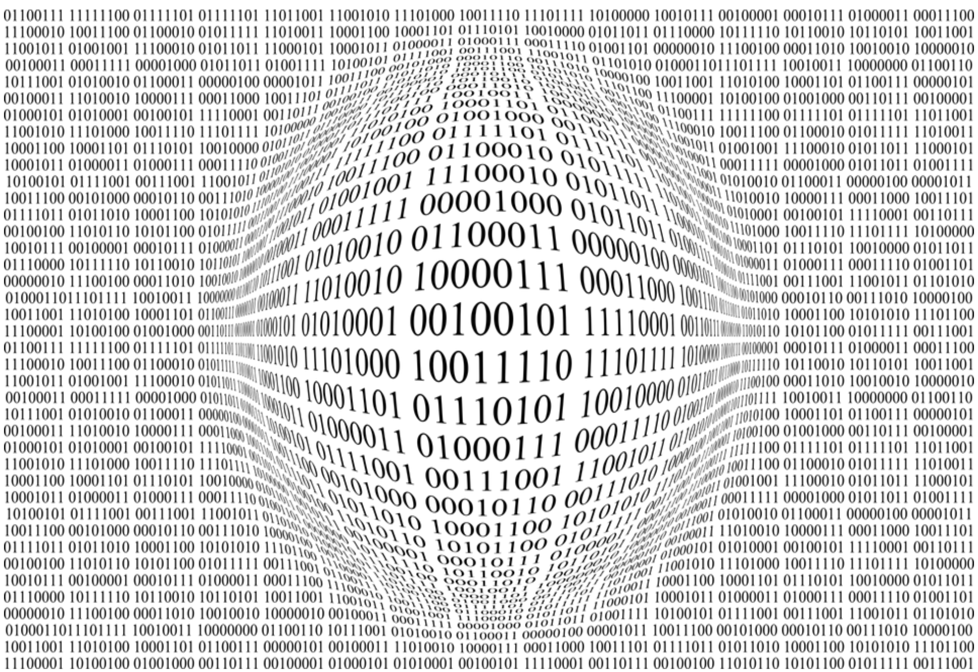
More Binary Representations



- Numbers
  - Signed integers
  - Biased integers
  - Fixed-point fractions
  - Floating point
- "Finiteness"
- Pixels
  - On screen
  - In files

Signed Integers

- Obvious method is to encode the sign of the integer using one bit.
- Conventionally, the most significant bit is used for the sign.
- This encoding of signed integers is called "SIGNED MAGNITUDE"

$$v = -1^S \sum_{i=0}^{n-2} 2^i b_i$$

S	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	1	1	1	1	1	0	0	0	0	1

-2017

- The Good
  - Easy to negate, easy to take absolute value
- The Bad
  - Two ways to represent "0", +0 and -0
  - Add/subtract is complicated; depends on the signs
- Not frequently used in practice
  - With one important exception that we'll discuss shortly

2's Complement Notation

- The 2's complement representation for signed integers is the most commonly used signed-integer representation.
- It is a simple modification of unsigned integers where the most significant bit is a negative power of 2.

$$v = -2^{n-1} b_{n-1} + \sum_{i=0}^{n-2} 2^i b_i$$

-2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	1	1	1	1	1	0	0	0	0	1

Still a "sign bit"
(It must be "1" for
the number to < 0)

$$\begin{array}{r} -2048 \\ +2017 \\ \hline -31 \end{array}$$

- Huh?
  - Negative numbers seem hard to "read" (for humans)
  - Nonsymmetric range:

    For 12 bits the range is $-2048 \le x \le 2047$

Why 2's Complement?

- In the two's complement representation for signed integers, the same binary "addition procedure" (mod 2^n) works for adding any combination of positive and negative numbers.
- Don't need a separate "subtraction procedure" (carries only, no borrows)
- The "addition procedure" also handles unsigned numbers!
- In 2's complement adding is adding regardless of operand signs.
- You NEVER need to subtract when you use 2's-complement.
- Just form the 2's -complement of the subtrahend

$$\begin{aligned} 55_{10} &= 000000110111_2 \\ +10_{10} &= 000000001010_2 \\ \hline 65_{10} &= 000001000001_2 \end{aligned}$$

$$\begin{aligned} 55_{10} &= 000000110111_2 \\ +-10_{10} &= 111111110110_2 \\ \hline 45_{10} &= 1000000101101_2 \end{aligned}$$

Ignore this "carry"

2's Complement Tricks

- **Negation** - changing the sign of a number
  1. Invert every bit (i.e. 1 → 0, 0 → 1)
  2. Add 1

Example: $42_{10} = 000000101010_2$
$-42_{10} = 111111010101_2 + 1 = 111111010110_2$

- **Sign-Extension** - aligning different sized 2's complement integers
  - Simply copy the sign bit into higher positions

Example: 16-bit version of 42: $42_{10} = 0000000000101010_2$
16-bit version of -42: $-42_{10} = 1111111111010110_2$

Class Exercise

10's-complement Arithmetic (so you'll never need to borrow again)

Step 1) Write down two 3-digit numbers, where you'll subtract the second from the first

Step 2) Form the 9's-complement of each digit in the second number (the subtrahend)

Step 3) Add 1 to it (the subtrahend)

Step 4) Add this number to the first

Step 5) If your result is less than 1000, form the 9's complement of the sum, add 1, and remember your result is negative, otherwise subtract 1000

Helpful Table of the 9's complement for each digit

$0 \rightarrow 9$
$1 \rightarrow 8$
$2 \rightarrow 7$
$3 \rightarrow 6$
$4 \rightarrow 5$
$5 \rightarrow 4$
$6 \rightarrow 3$
$7 \rightarrow 2$
$8 \rightarrow 1$
$9 \rightarrow 0$

What did you get? Why weren't you taught to subtract this way?

Fixed-point numbers

- You can always assume that the boundary between 2 bits is a "binary point".
- If you **align** binary points between addends, there is no effect on how operations are preformed.

-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}
1	1	1	1	1	1	0	1	0	1	1	0

$$
\begin{aligned}
11111101.0110 &= -2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^0 + 2^{-2} + 2^{-3} \\
&= -128 + 64 + 32 + 16 + 8 + 4 + 1 + 0.25 + 0.125 \\
&= -2.625
\end{aligned}
$$

OR

$$
\begin{aligned}
11111101.0110 &= -42 \times 2^{-4} \\
&= -42 \;/\; 16 \\
&= -2.625
\end{aligned}
$$

Repeated Binary Fractions

Not all fractions can be represented exactly using a **finite representation**. You've seen this before in decimal notation where the fraction 1/3 (among others) requires an infinite number of digits to represent (0.3333...).

In binary, a great many fractions that you've grown attached to require an infinite number of bits to represent exactly.

Example: $1/10 = 0.1_{10} = 0.0\overline{0011}\ldots_2 = 0.1\overline{9}\ldots_{16}$

$1/5 = 0.2_{10} = 0.\overline{0011}\ldots_2 = 0.\overline{3}\ldots_{16}$

$1/3 = 0.3_{10} = 0.\overline{01}\ldots_2 = 0.\overline{5}\ldots_{16}$

Finite Representations

- Computers use a finite set of bits (or certain fixed-sized bit clusters) to represent numbers.
- In fact, **everything** that a realizable computer does is limited by a finite set of bits.
- Through your mastery of mathematics, you've gradually grown used to infinite representations. So much so that finite representations seem odd
- One type of infinity that you've grown used to: **Infinite digits**

```
   ...0000000042.0000000000...
   ...0000000000.0000000000...001000
10000000...0000000000.0
```

- The concept an infinite supply of zero digits is conceptually elegant, but difficult to physically implement

Side Effects of being Finite

These examples assume a finite 16-bit representation

- You can "overflow"

$32767_{10} + 1_{10} = -32768_{10}$

$$
\begin{array}{r}
0111\ 1111\ 1111\ 1111_2 \\
+\ 0000\ 0000\ 0000\ 0001_2 \\
\hline
1000\ 0000\ 0000\ 0000_2
\end{array}
$$

$-20000_{10} - 20000_{10} = 25536_{10}$

$$
\begin{array}{r}
1011\ 0001\ 1110\ 0000_2 \\
+\ 1011\ 0001\ 1110\ 0000_2 \\
\hline
1\ 0110\ 0011\ 1100\ 0000_2
\end{array}
$$

- Certain numbers can't be negated

$-32768_{10} = -32768_{10}$

$$
\begin{array}{r}
1000\ 0000\ 0000\ 0000_2 \\
0111\ 1111\ 1111\ 1111_2 \\
+\ \ 0000\ 0000\ 0000\ 0001_2 \\
\hline
1000\ 0000\ 0000\ 0000_2
\end{array}
$$

Bias Notation

There is yet one more way to represent signed integers, which is surprisingly simple. It involves subtracting a fixed constant from a given unsigned number. This representation is called "Bias Notation".

$$v = \sum_{i=0}^{n-1} 2^i b_i - Bias$$

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	0	0	1	0	1	1	0

Example of Bias 127:

$$\begin{array}{rrr} & 9 \times 2^4 = & 144 \\ + & 6 \times 2^0 = & 6 \\ - & & 127 \\ \hline & & 23 \end{array}$$

Adding 2 numbers requires a subtraction to fix the result

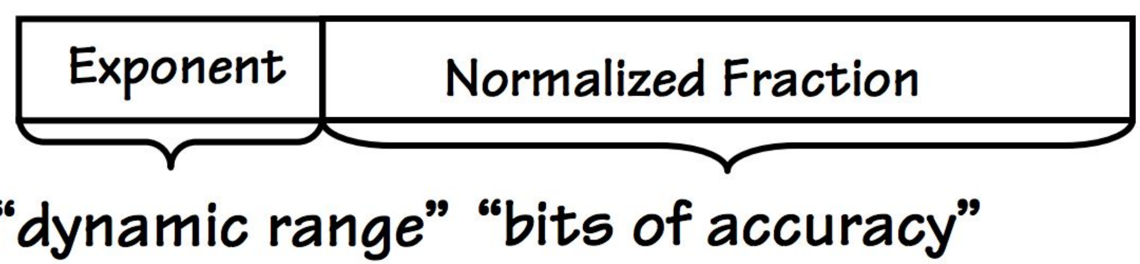
$$\begin{array}{rl} & 150 \\ + & 150 \\ - & 127 \\ \hline & 173 = 46 + 127 \end{array}$$

Why? Monotonicity when viewed as an unsigned number

Floating Point Numbers

Another way to represent numbers is to use a notation similar to **Scientific Notation**. This format can be used to represent numbers with fractions (3.90×10^{-4}), very small numbers (1.60×10^{-19}), and large numbers (6.02×10^{23}). This notation uses two fields to represent each number. The first part represents a normalized fraction (called the significand), and the second part represents the exponent (i.e. the position of the "floating" binary point).

$$\text{Normalized} \quad \text{Fraction} \times 2^{\text{Exponent}}$$



IEEE 754 Format

- Single precision format

This is effectively a signed magnitude fixed-point number with a "hidden" 1.

The 1 is hidden because it provides no information after the number is "normalized"

S	Exponent	Significand
1	8	23

The exponent is represented in bias 127 notation. Why?

$$v = -1^{s} \times 1.\text{Significand} \times 2^{\text{Exponent}-127}$$

- Example: $52.25 = 00110100.01000000_2$

Normalize: $001.10100010000000_2 \times 2^5$

(127+5)

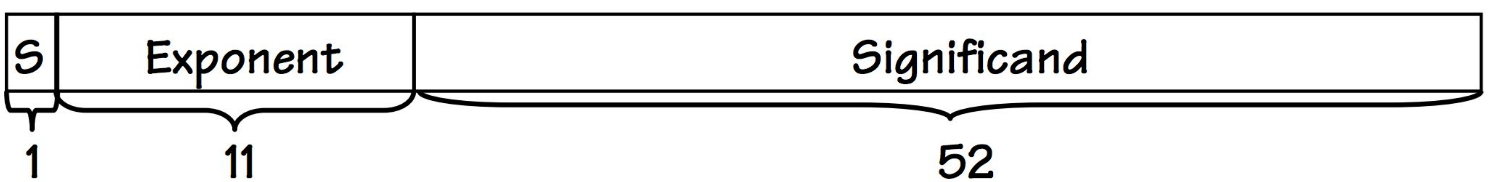
0 10000100 10100010000000000000000

0100 0010 0101 0001 0000 0000 0000 0000

52.25 = 0x42510000

IEEE 754 Limits and Features

- Single precision limitations
  - A little more than 7 decimal digits of precision
  - Minimum positive normalized value: $\sim 1.18 \times 10^{-38}$
  - Maximum positive normalized value: $\sim 3.4 \times 10^{38}$
- Inaccuracies become evident after multiple single precision operations
- Double precision format

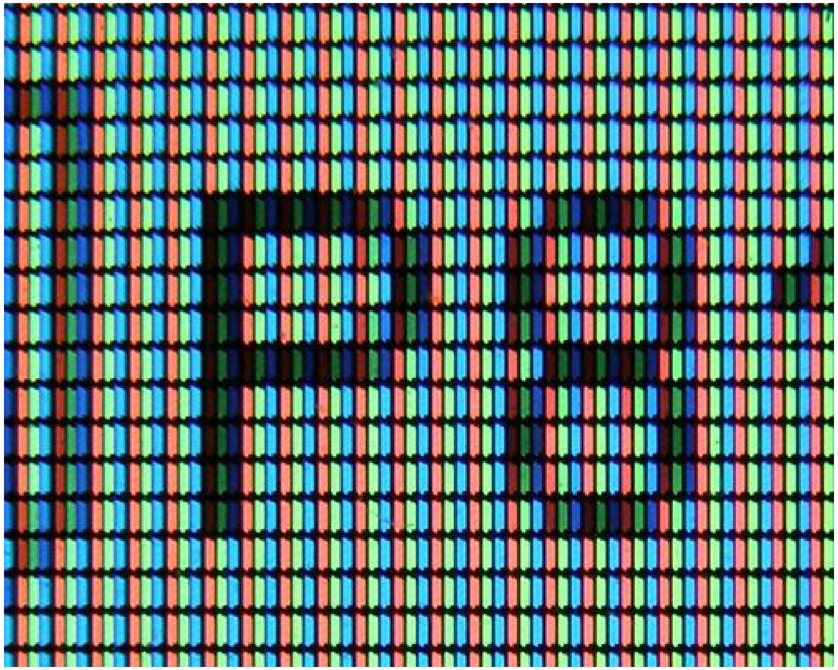


$$v = -1^{s} \times 1.\text{Significand} \times 2^{\text{Exponent}-1023}$$

Bits You can See

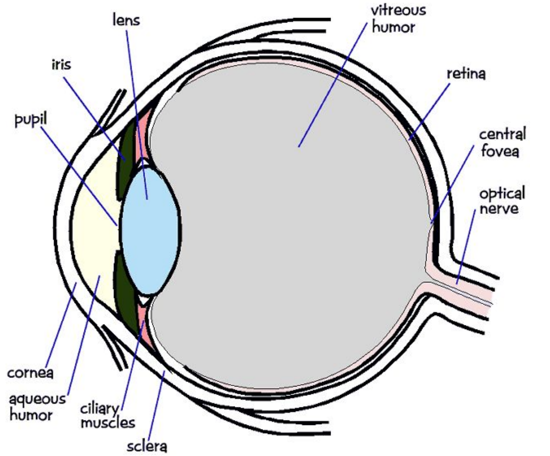
The smallest element of a visual display is called a "pixel". Pixels have three independent color components that generate most of the ***perceivable*** color range.

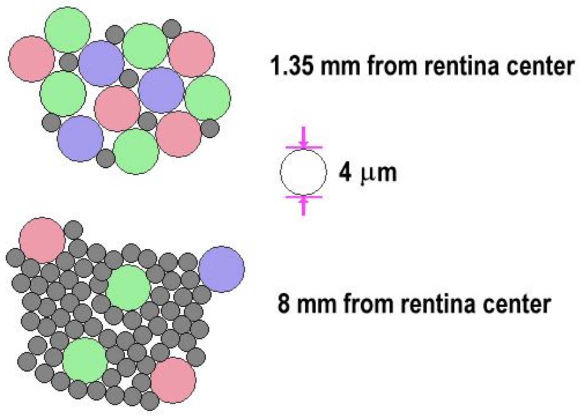
- Why three and what are they
- How are they represented in A computer?
- First, let's discuss this notion of perceivable



It starts with the Eye

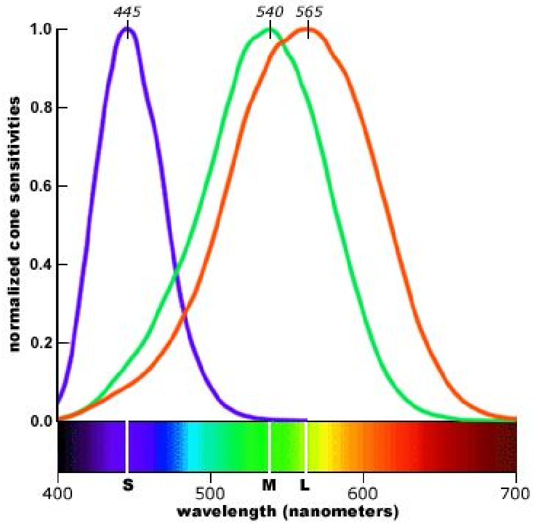
- The photosensitive part of the eye is called the retina.
- The retina is largely composed of two cell types, called rods and cones.
- Cones are responsible for color perception.
- Cones are most dense within the fovea.
- There are three types of cones, referred to as S, M, and L whose spectral sensitivity varies with wavelength.

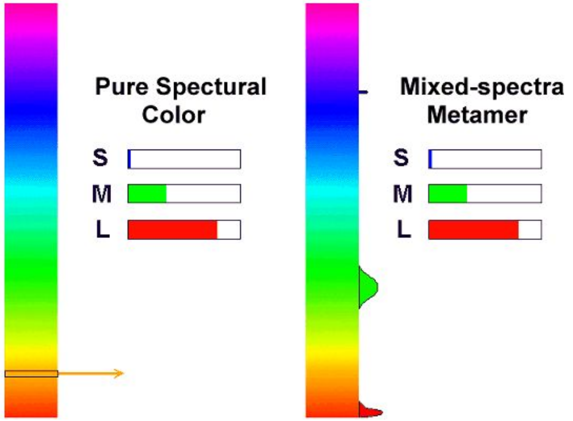




Why we see in color

- Pure spectral colors simulate all cones to some extent.
- Mixing multiple colors can stimulate the cones to respond in a way that Is indistinguishable from a pure color.
- Perceptually identical sensations are called metamers.
- This allows us to use just three colors to generate all others.





How colors Are Represented

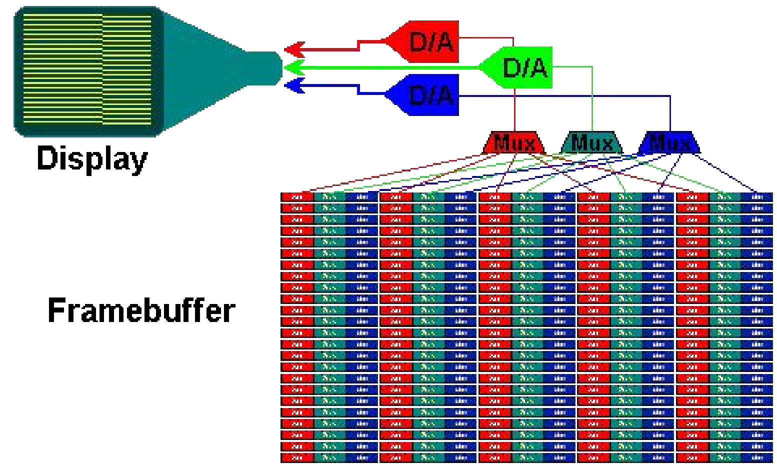
- Each pixel is stored as three primary parts
- Red, green, and blue
- Usually around 8-bits per channel
- Pixels can have individual R,G,B components or they can be stored indirectly via a "look-up table"

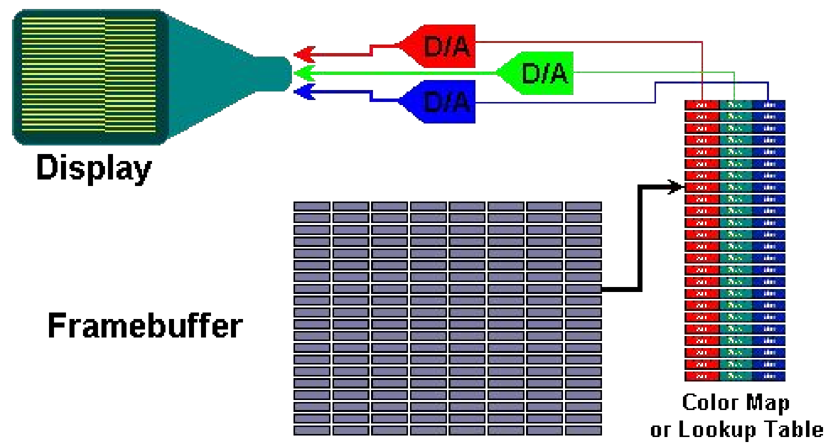
8-bits	8-bits	8-bits

3 - 8-bit unsigned binary integers (0,255)

-OR-

3 - fixed point 8-bit values (0-1.0)





Color Specifications

Web colors:

Name	Hex	Decimal Integer	Fractional
Orange	#FFA500	(255, 165, 0)	(1.0, 0.65, 0.0)
Sky Blue	#87CEEB	(135, 206, 235)	(0.52, 0.80, 0.92)
Thistle	#D8BFD8	(216, 191, 216)	(0.84, 0.75, 0.84)

Colors are stored as binary too. You'll commonly see them in Hex, decimal, and fractional formats.

Summary

- **ALL** modern computers represent signed integers using a two's-complement representation
- Two's-complement representations eliminate the need for separate addition and subtraction units
- Addition is **identical** using either unsigned and two's-complement numbers
- Finite representations of numbers on computers leads to anomalies
- Floating point numbers have separate fractional and exponent components.