6.837 Lecture 2

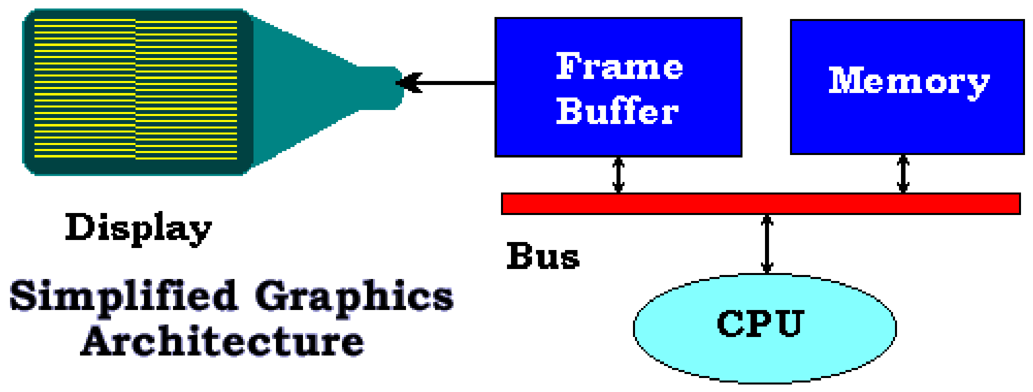
1. Rasters, Pixels and Sprites
2. Review of Raster Displays
3. High-End Graphics Display System
4. A Memory Raster
5. A Java Model of a Memory Raster
6. Example Usage: Rastest.java
7. Lets Talk About Pixels
8. True-Color Frame Buffers
9. Indexed-Color Frame Buffers
10. High-Color Frame Buffers
11. Sprites
12. A Sprite is a Raster
13. An Animated Sprite is a Sprite
14. A Playfield is a Raster and has Animated Sprites
15. PixBlts
16. Seems Easy
17. The Tricky Blits
18. Alpha Blending
19. Alpha Compositing Details
20. Compositing Example
21. Elements of Color
22. Visible Spectrum
23. The Eye
24. The Fovea
25. The Fovea
26. Color Perception
27. Dominant Wavelength
28. Color Algebra
27. Saturated Color Curve in RGB
28. Color Matching
29. CIE Color Space
30. CIE Chromaticity Diagram
31. Definitions:
32. Color Gamuts
33. The RGB Color Cube
34. Color Printing
35. Color Conversion
36. Other Color Systems
35. Next Time : Flooding and Other Imaging Topics

Rasters, Pixels and Sprites

Experiencing **JAVA**phobia?

- Rasters
- Pixels
- Alpha
- RGB
- Sprites
- BitBlts

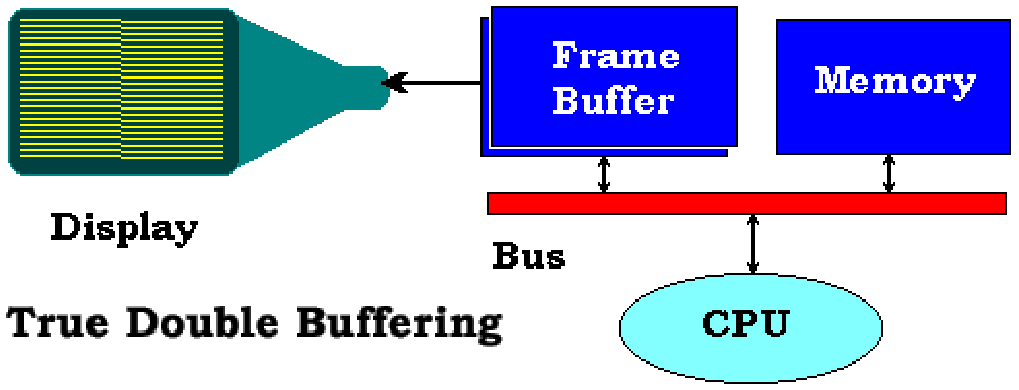
Review of Raster Displays



Simplified Graphics Architecture

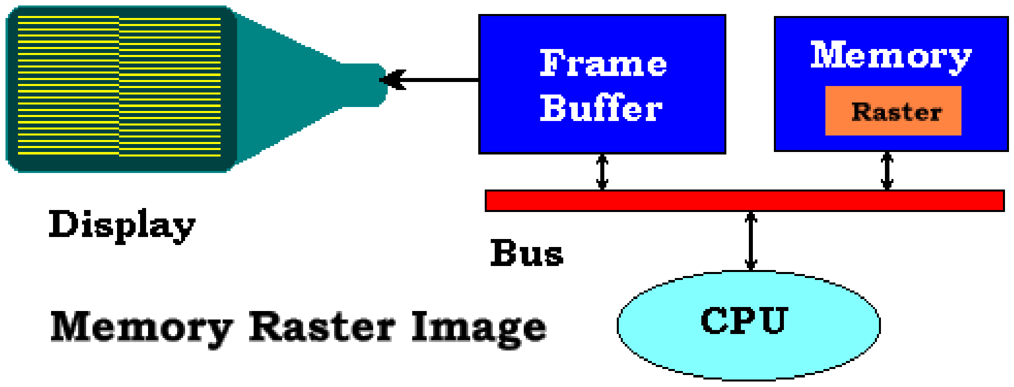
- Display synchronized with CRT sweep
  - Special memory for screen update
- Pixels are the discrete elements displayed
  - Generally, updates are visible

High-End Graphics Display System



- Adds a second frame buffer
- Swaps during vertical blanking
- Updates are invisible
- Costly

A Memory Raster



- Maintains a copy of the screen (or some part of it) in memory
  - Relies on a fast copy
  - Updates are *nearly* invisible
  - Conceptual model of a physical object

A Java Model of a Memory Raster

```
class Raster implements ImageObserver {
    /////////////////////////// Constructors //////////////////////
    public Raster();                 // allows class to be extended
    public Raster(int w, int h);     // specify size
    public Raster(Image img);        // set to size and contents of image

    /////////////////////////// Interface Method /////////////////
    public boolean imageUpdate(Image img, int flags, int x, int y, int w, int h);

    /////////////////////////// Accessors /////////////////////////
    public int getSize( );           // pixels in raster
    public int getWidth( );          // width of raster
    public int getHeight( );         // height of raster
    public int[] getPixelBuffer( ); // get array of pixels

    /////////////////////////// Methods ///////////////////////////
    public void fill(int argb);      // fill with packed argb
    public void fill(Color c);           // fill with Java color
    public Image toImage(Component root);
    public int getPixel(int x, int y);
    public Color getColor(int x, int y);
    public boolean setPixel(int pix, int x, int y);
    public boolean setColor(Color c, int x, int y);
}
```

Download Raster.java here.

Example Usage: Rastest.java

The code on the right demonstrates the use of a Raster object. The running Applet is shown below. Clicking on the image will cause it to be negated.

The source code for this applet can be downloaded here: Rastest.java.

```
import java.applet.*;
import java.awt.*;
import Raster;

public class Rastest extends Applet {
    Raster raster;
    Image output;
    int count = 0;

    public void init() {
      String filename = getParameter("image");
      output = getImage(getDocumentBase(), filename);
      raster = new Raster(output);
      showStatus("Image size: " + raster.getWidth() + " x " + raster.getHeight());
    }

    public void paint(Graphics g) {
      g.drawImage(output, 0, 0, this);
      count += 1;
      showStatus("paint() called " + count + " time" + ((count > 1) ? "s":""));
    }

    public void update(Graphics g) {
      paint(g);
    }

    public boolean mouseUp(Event e, int x, int y) {
      int s = raster.getSize();
      int [] pixel = raster.getPixelBuffer();
      for (int i = 0; i < s; i++) {
        raster.pixel[i] ^= 0x00ffffff;
      }
      output = raster.toImage(this);
      repaint();
      return true;
    }
}
```

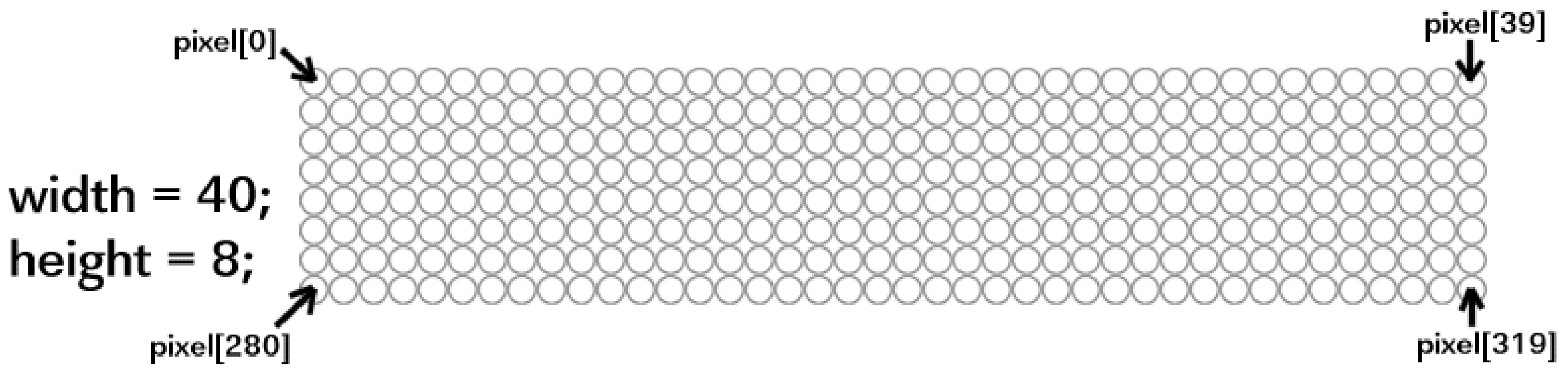
Lets Talk About Pixels

- Pixels are stored as a 1-dimensional array of *int*s
- Each *int* is formatted according to Java's standard pixel model

Alpha	Red	Green	Blue

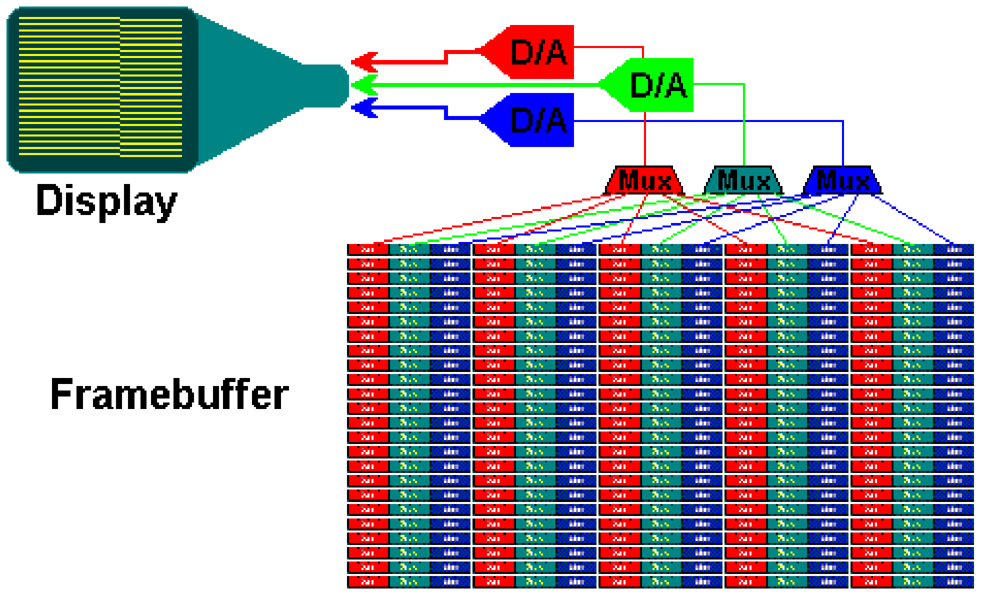
The 4 bytes of a 32-bit *Pixel* int.
if Alpha is 0 the pixel is transparent.
if Alpha is 255 the pixel is opaque.

- Layout of the pixel array on the display:



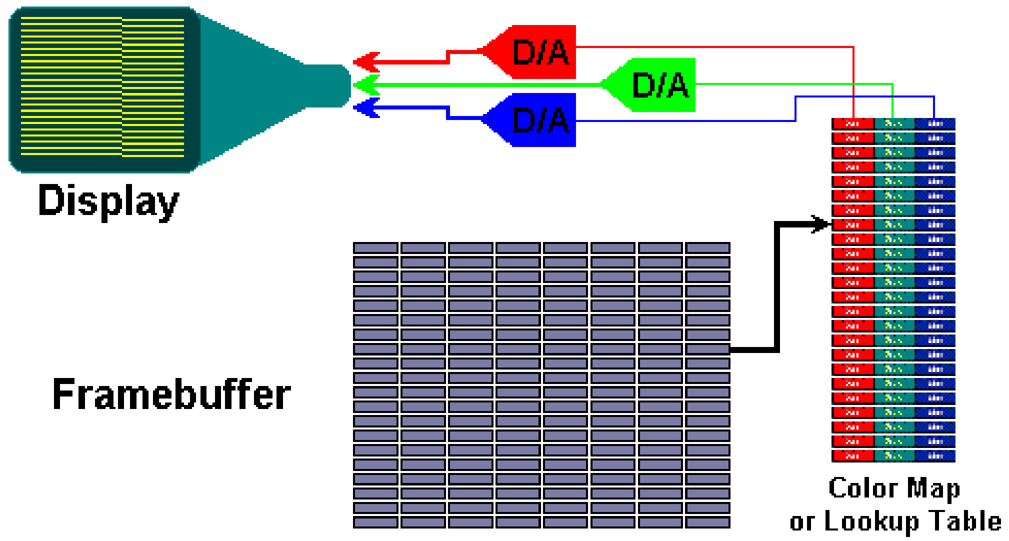
- This is the image format used internally by Java

True-Color Frame Buffers



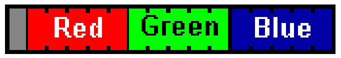
- Each pixel requires at least 3 bytes. One byte for each primary color.
- Sometimes combined with a look-up table per primary
- Each pixel can be one of 2^24 colors
- Worry about your *Endians*

Indexed-Color Frame Buffers



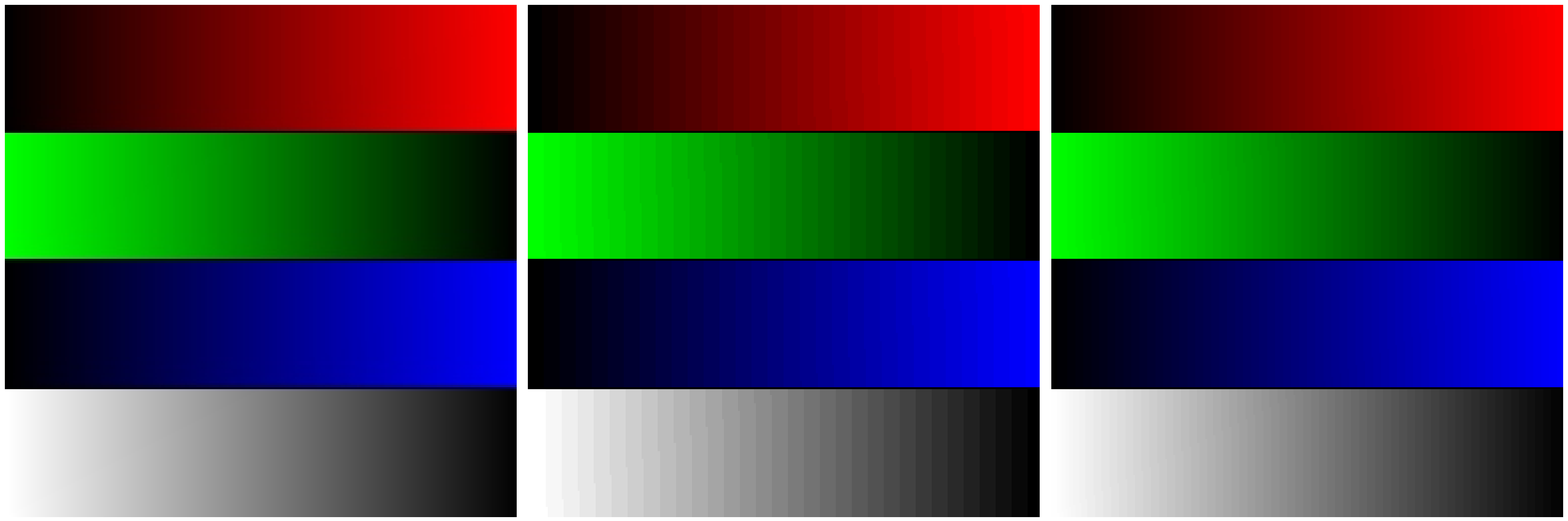
- Each pixel uses one byte
- Each byte is an index into a color map
- If the color map is not updated synchronously then *Color-map flashing* may occcur.
- Color-map Animations
- Each pixel may be one of 2^24 colors, but only 256 color be displayed at a time

High-Color Frame Buffers



Pixels are packed in a short
Each primary uses 5 bits

- Popular *PC/(SVGA)* standard (popular with Gamers)
- Each pixel can be one of 2^15 colors
- Can exhibit worse quantization (banding) effects than Indexed-color



Sprites

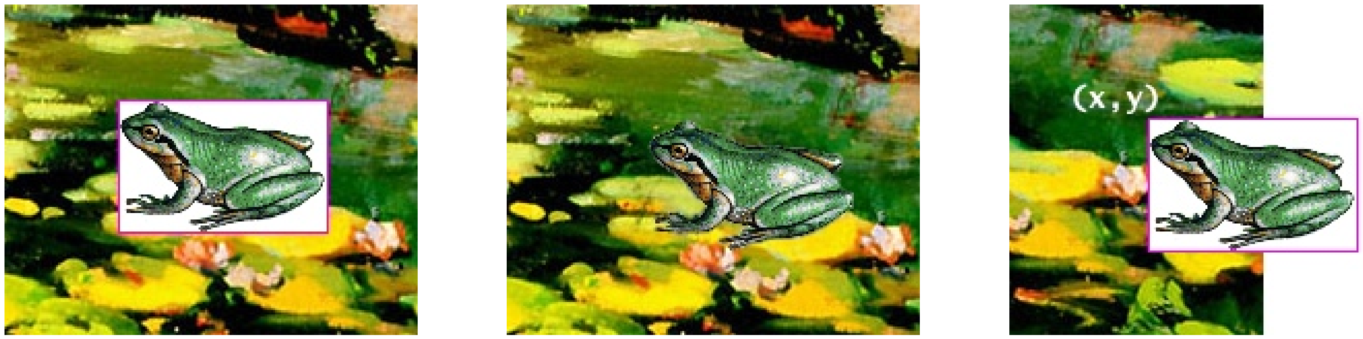
Sprites are rasters that can be overlaid onto a background raster called a playfield.

A sprite can be animated, and it generally can be repositioned freely any where within the playfield.

A Sprite is a Raster

```
        class Sprite extends Raster {
            int x, y;                             // position of
sprite on playfield
            public void Draw(Raster bgnd);  // draws sprite on a
Raster
        }
```

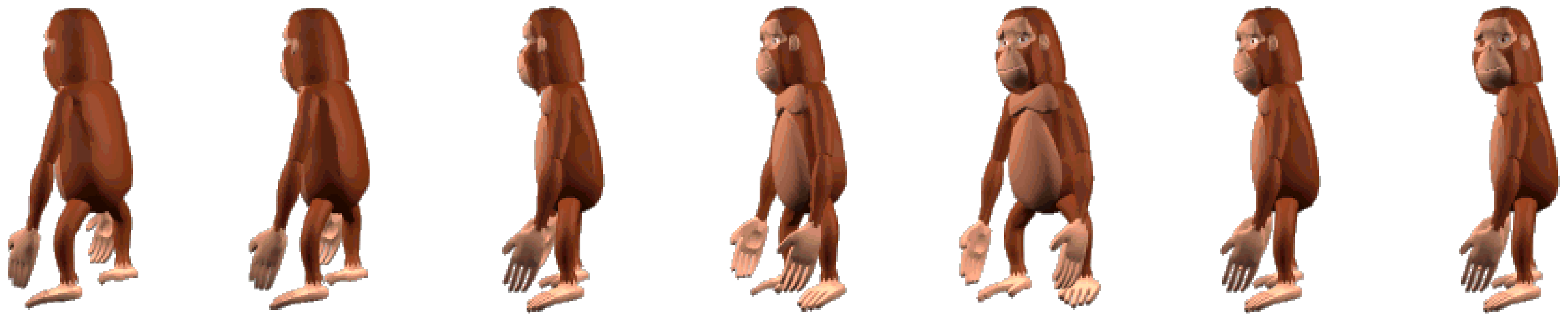
Things to consider:



The Draw( ) method must handle transparent pixels, and it must also handle all cases where the sprite overhangs the playfield.

An Animated Sprite is a Sprite

```
class AnimatedSprite extends Sprite {
    int frames;         // frames in sprite
                        // there are other private variables
        public AnimatedSprite(Image images, int frames);
        public AnimatedSprite(AnimatedSprite s); // copy a sprite
        public void addState(int track, int frame, int ticks, int dx, int dy);
        public void addTrack(int anim, StringTokenizer parse);
        public void Draw(Raster bgnd);
        public void nextState();
        public void setTrack(int t);
        public boolean notInView(Raster bgnd);
        public boolean overlaps(AnimatedSprite s);
}
```

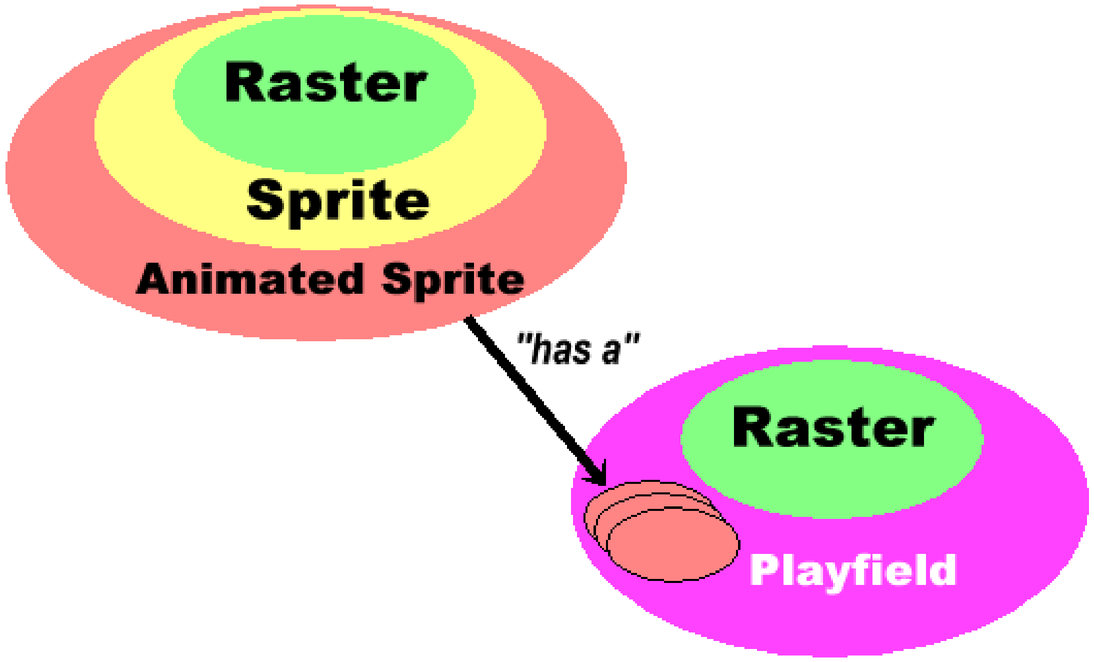


A track is a series of frames. The *frame* is displayed for *ticks* periods. The sprite's position is then moved (*dx*, *dy*) pixels.

A Playfield is a Raster and has Animated Sprites

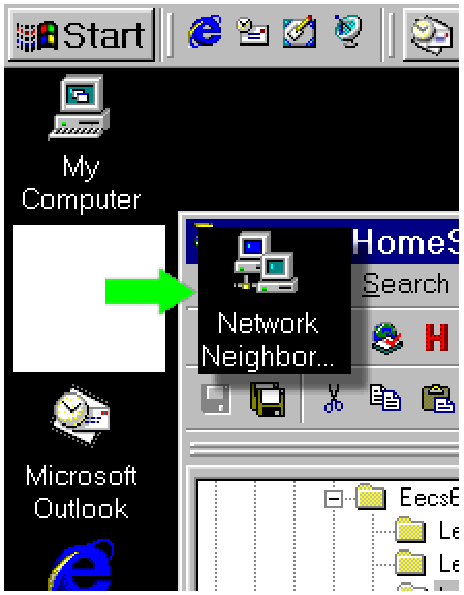
```
class Playfield extends Raster {
    Raster background;          // background image
    Vector sprites;                    // sprites on this playfield

        public Playfield(Image bgnd);
        public void addSprite(AnimatedSprite s);
        public void removeSprite(AnimatedSprite s);
        public void Draw( );
        // Other methods...
}
```



PixBlts

PixBlts are raster methods for moving and clearing sub-blocks of pixels from one region of a raster to another



Very heavily used by window systems:

- moving windows around
- scrolling text
- copying and clearing

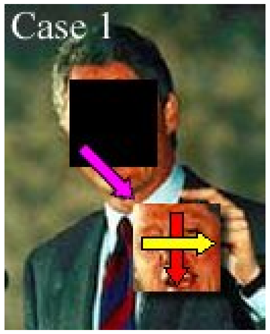
Seems Easy

Here's a PixBlt method:

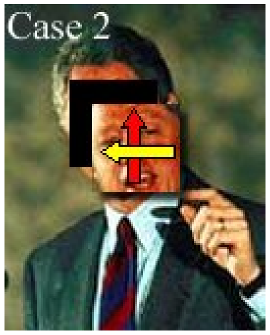
```
public void PixBlt(int xs, int ys, int w, int h, int xd, int yd)
{
    for (int j = 0; j < h; j++) {
        for (int i = 0; i < w; i++) {
            this.setPixel(raster.getPixel(xs+i, ys+j), xd+i, yd+j);
        }
    }
}
```

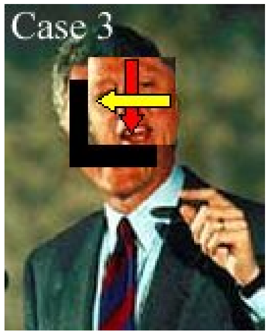
But does this work?
What are the issues?
How do you fix it?

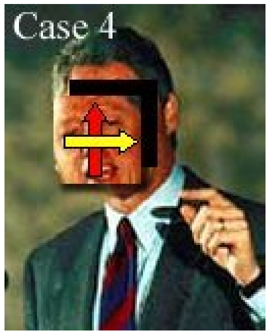
The Tricky Blits

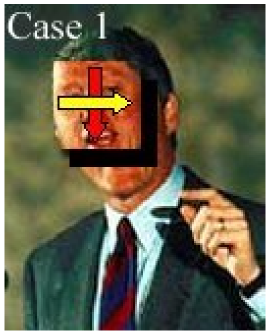


Our PixBlt Method works fine when the source and destination regions do not overlap. But, when these two regions do overlap we need to take special care to avoid copying the wrong pixels. The best way to handle this situation is by changing the iteration direction of the copy loops to avoid problems.



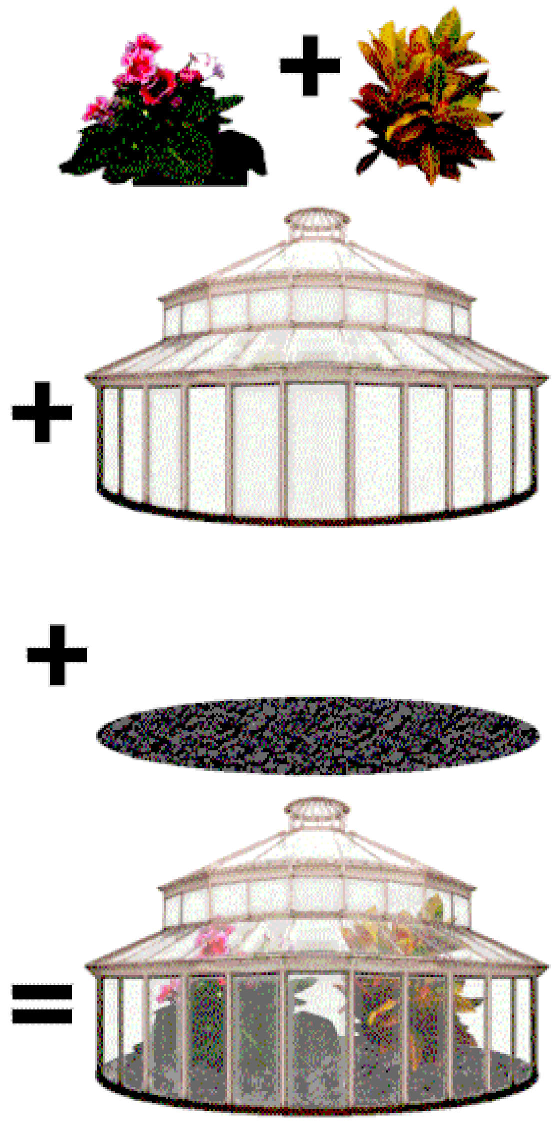






The iteration direction must always be *opposite* of the direction of the block's movement for each axis. You could write a fancy loop which handles all four cases. However, this code is usually so critical to system performace that, generally, code is written for all 4 cases and called based on a test of the source and destination origins. In fact the code is usually unrolled.

Alpha Blending



Alpha blending simulates the *opacity* of *celluloid* layers

General rules:

- Adds one channel to each pixel [&alpha;, r, g, b]
- Usually process layers back-to-front (using the *over* operator)
- 255 or 1.0 indicates an opaque pixel
- 0 indicates a transparent pixel
- Result is a function of foregrond and background pixel colors
- Can simulate partial-pixel coverage for anti-aliasing

See Microsoft's Image Composer for more info on these images

Alpha Compositing Details

Definition of the *Over* compositing operation:

$$\&alpha;_{result} c_{result} = \&alpha;_{fg}\, c_{fg} + (1 - \&alpha;_{fg})\, \&alpha;_{bg} c_{bg}$$

$$\&alpha;_{result} = \&alpha;_{fg} + (1 - \&alpha;_{fg})\, \&alpha;_{bg}$$

Issues with alpha:

Premultiplied (integral) alphas:

- pixel contains (&alpha;, &alpha;r, &alpha;g, &alpha;b)
- saves computation
  $\&alpha;_{result} c_{result} = \&alpha;_{fg}\, c_{fg} + \&alpha;_{bg} c_{bg} - \&alpha;_{fg} \&alpha;_{bg} c_{bg}$
- alpha value constrains color magnitude
- alpha modulates image shape
- conceptually clean - multiple composites are well defined

Non-premultiplied (independent) alphas:

- pixel contains (&alpha;, r, g, b)
- what Photoshop does
- color values are independent of alpha
- transparent pixels have a color
- divison required to get color component back

(An excellent reference on the subject)

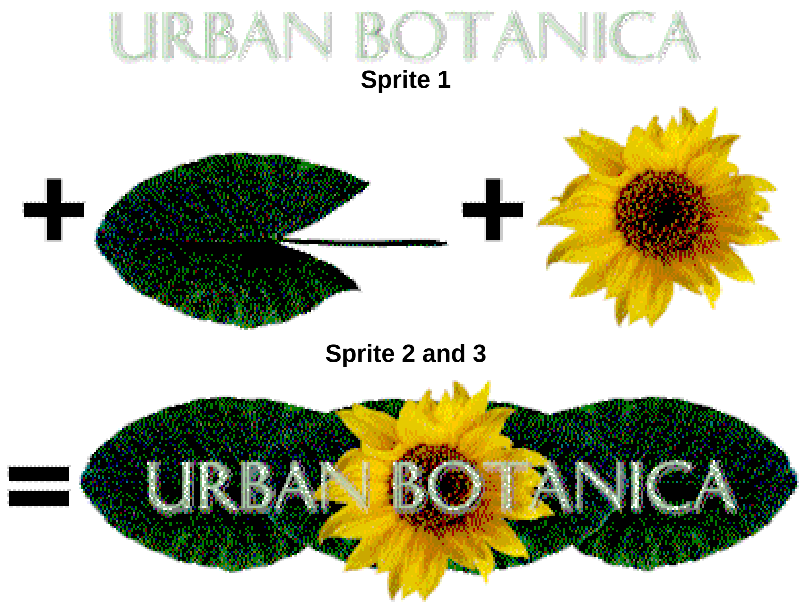
PREVIOUS

NEXT

Compositing Example

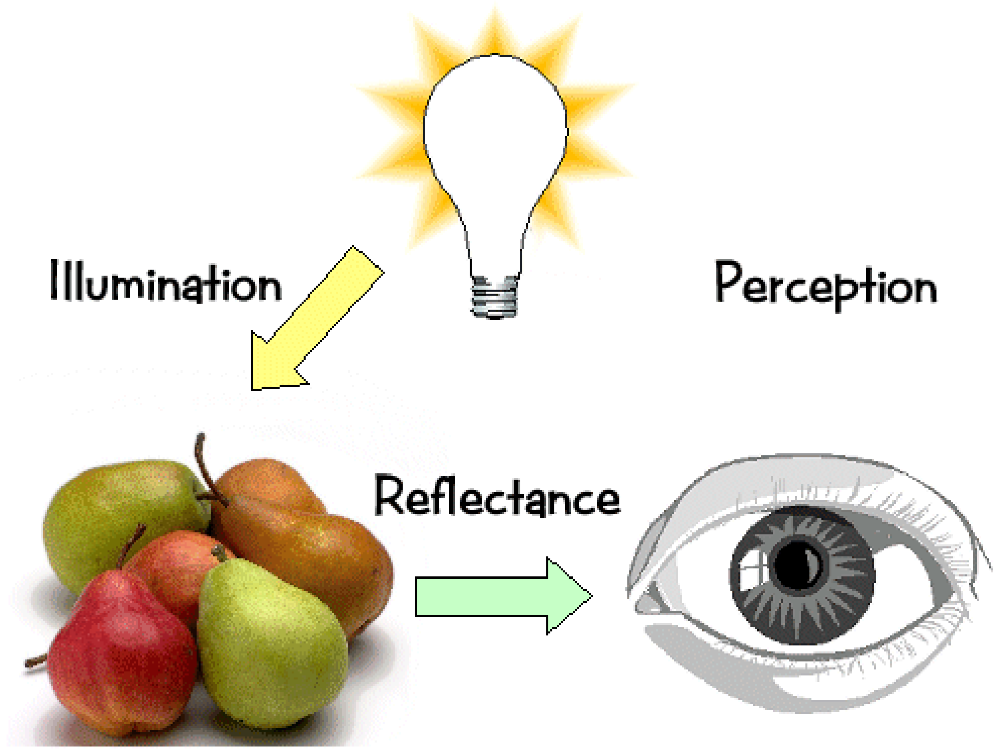
Alpha-blending features:

- Allows image to encode the shape of an object (Sprite)
- Can be used to represent partial pixel coverage for anti-aliasing (independent of the final background color)
- Can be used for transparency effects
- Should be adopted by everyone!
  (what were you planning to do with that extra byte anyway)



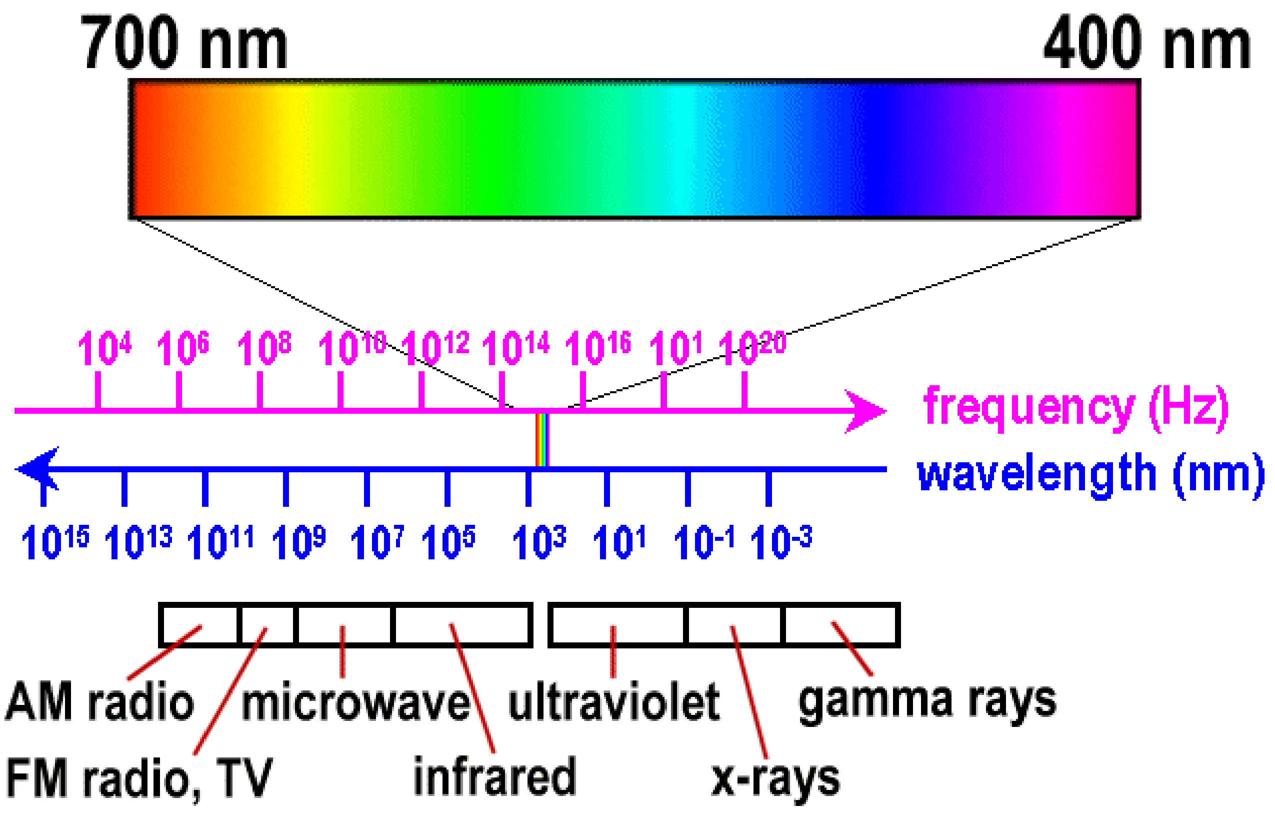
Elements of Color

Hearn & Baker - Chapter 15



Visible Spectrum

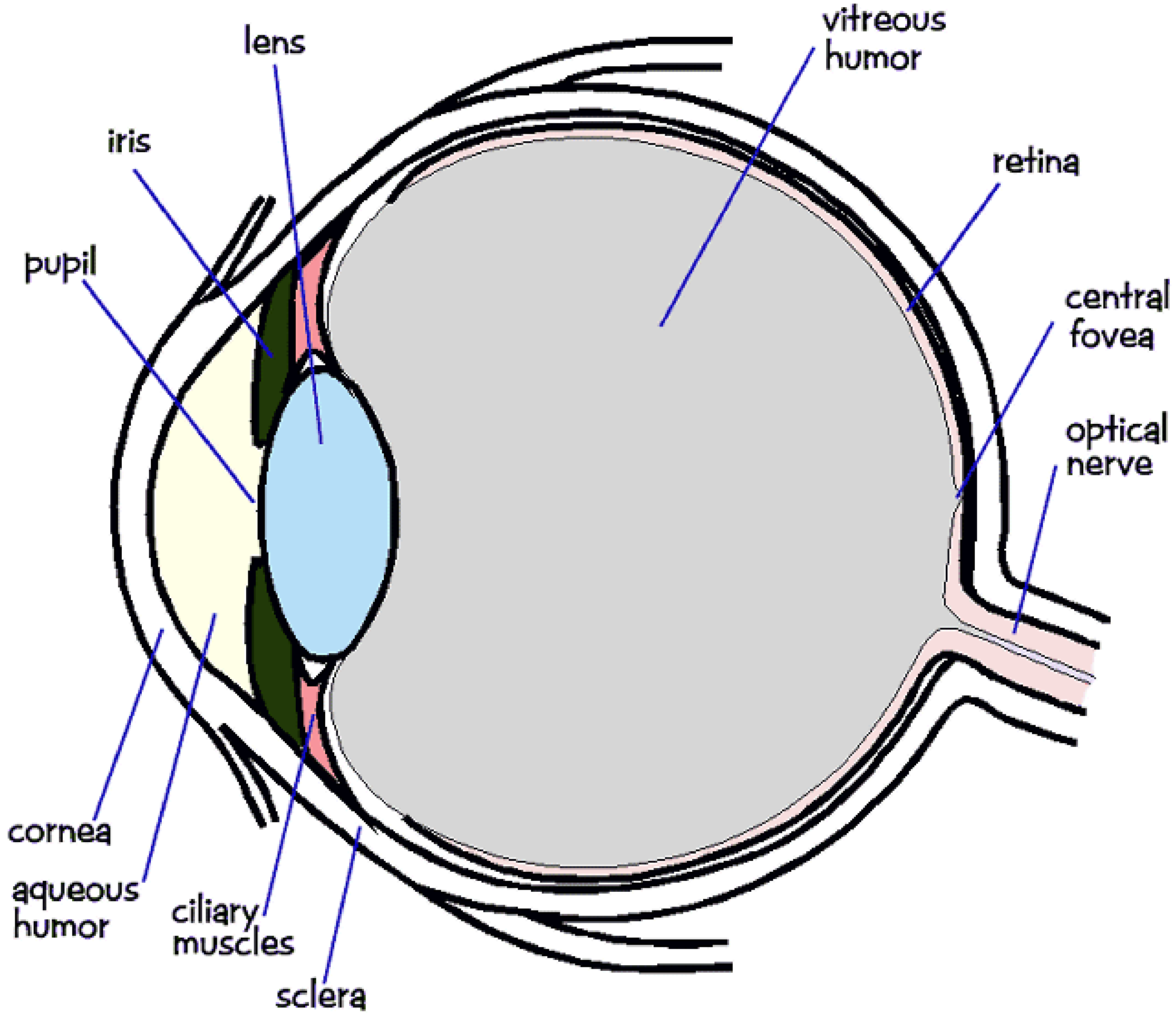
We percieve electromagnetic energy having wavelengths in the range 400-700 nm as *visible light*.



The Eye

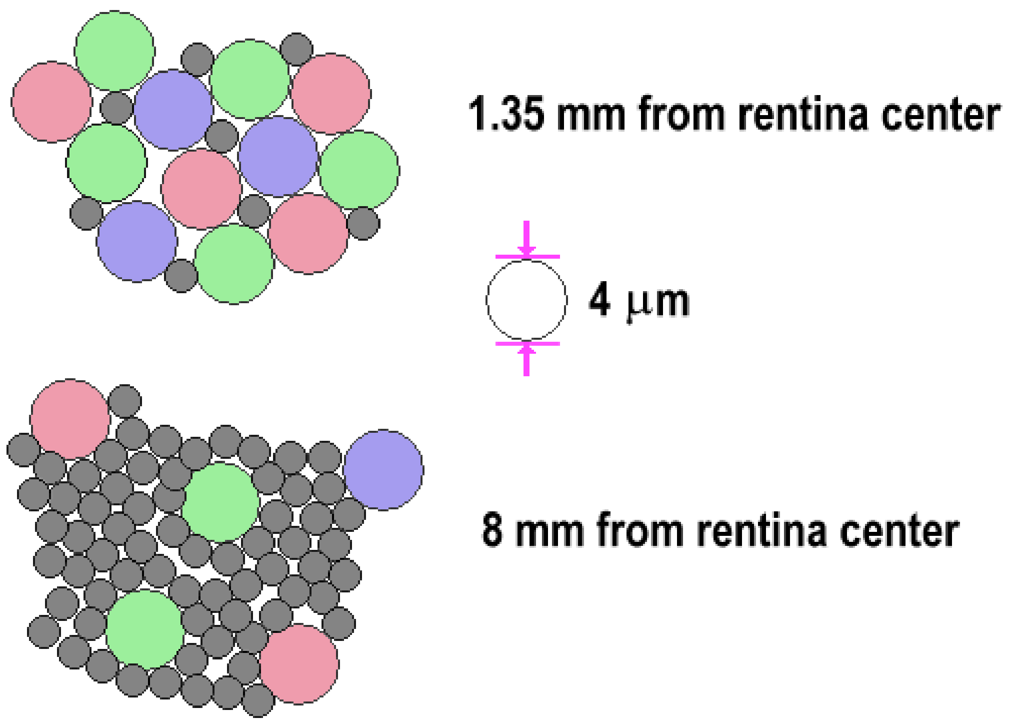
The photosensitive part of the eye is called the *retina*.

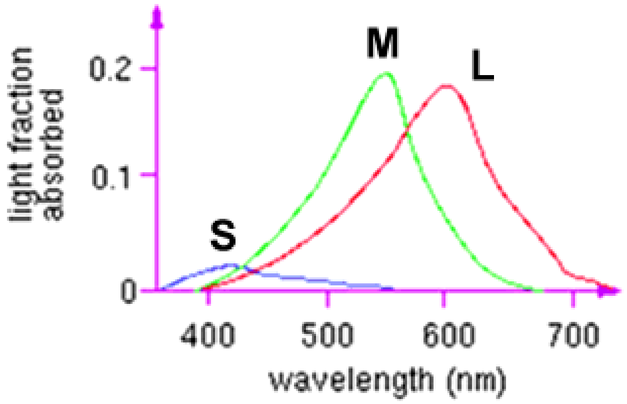
The retina is largely composed of two types of cells, called *rods* and *cones*. Only the cones are responsible for color perception.



The Fovea

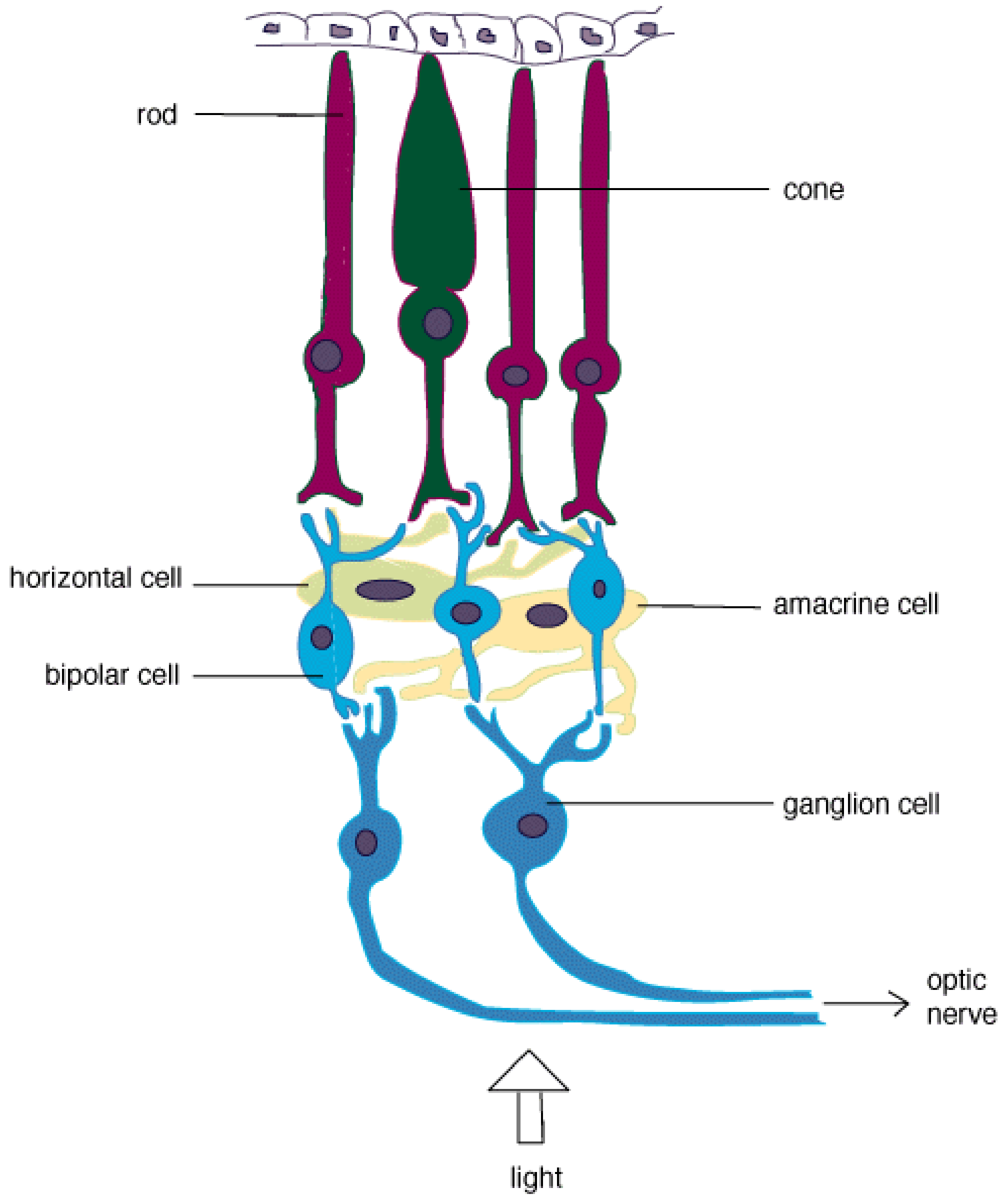
Cones are most densely packed within a region of the eye called the *fovea*.

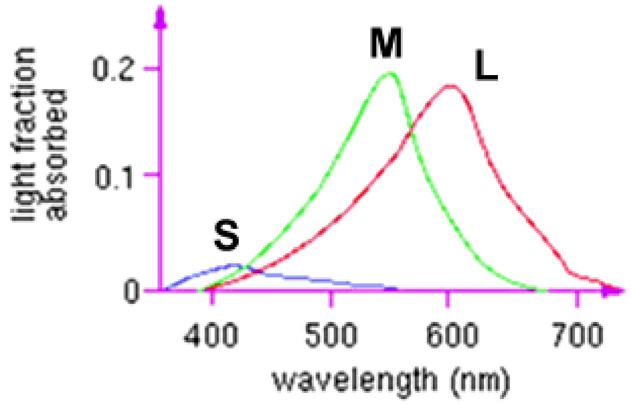




There are three types of cones, referred to as S, M, and L. They are roughly equivalent to blue, green, and red sensors, respectively. Their peak sensitivities are located at approximately 430nm, 560nm, and 610nm for the "average" observer.

The Fovea

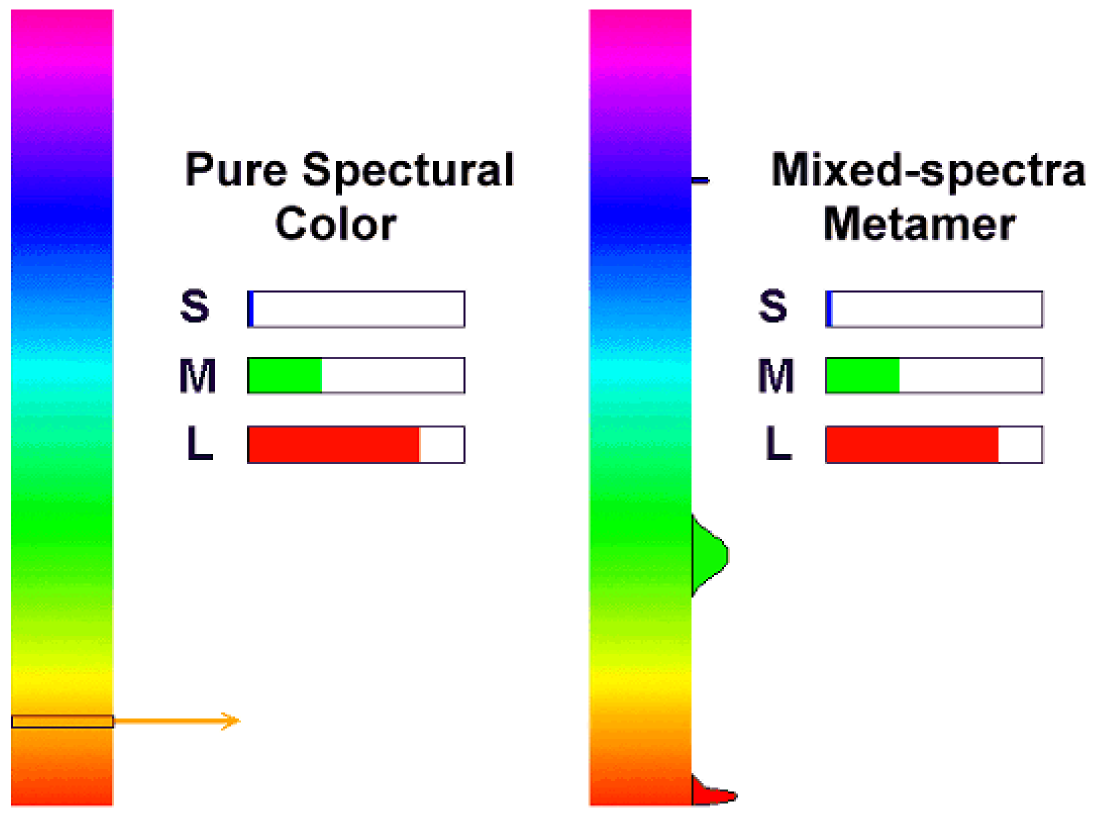




Colorblindness results from a deficiency of one cone type.

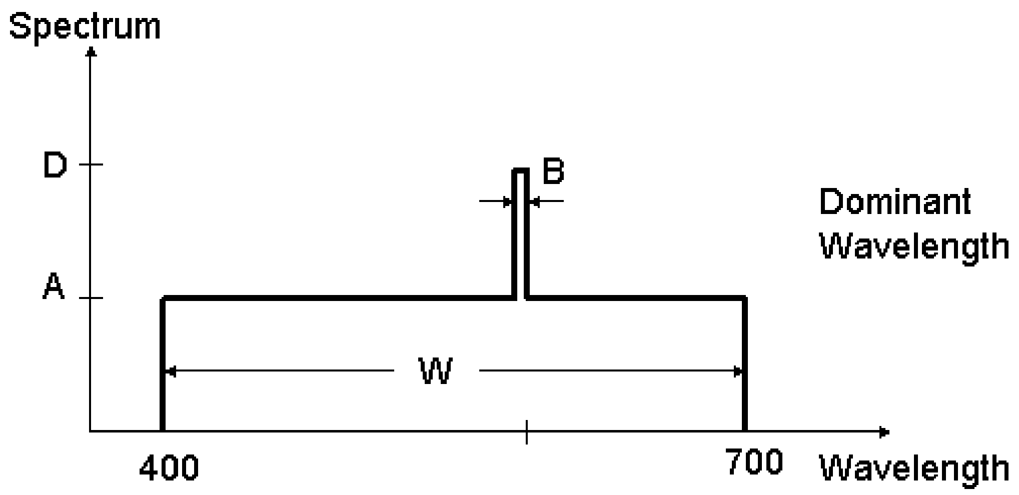
Color Perception

- Different spectra can result in a perceptually identical sensations called *metamers*
- Color perception results from the simultaneous stimulation of 3 cone types (*trichromat*)
- Our perception of color is also affected by surround effects and adaptation



Dominant Wavelength

- Location of **dominant wavelength** specifies the **hue** of the color
- The **luminance** is the total power of the light (area under curve) and is related to **brightness**
- The **saturation** (purity) is percentage of luminance in the dominant wavelength



L = (D – A)B + A W   luminance
S = (D-A)B / L   saturation

Color Algebra

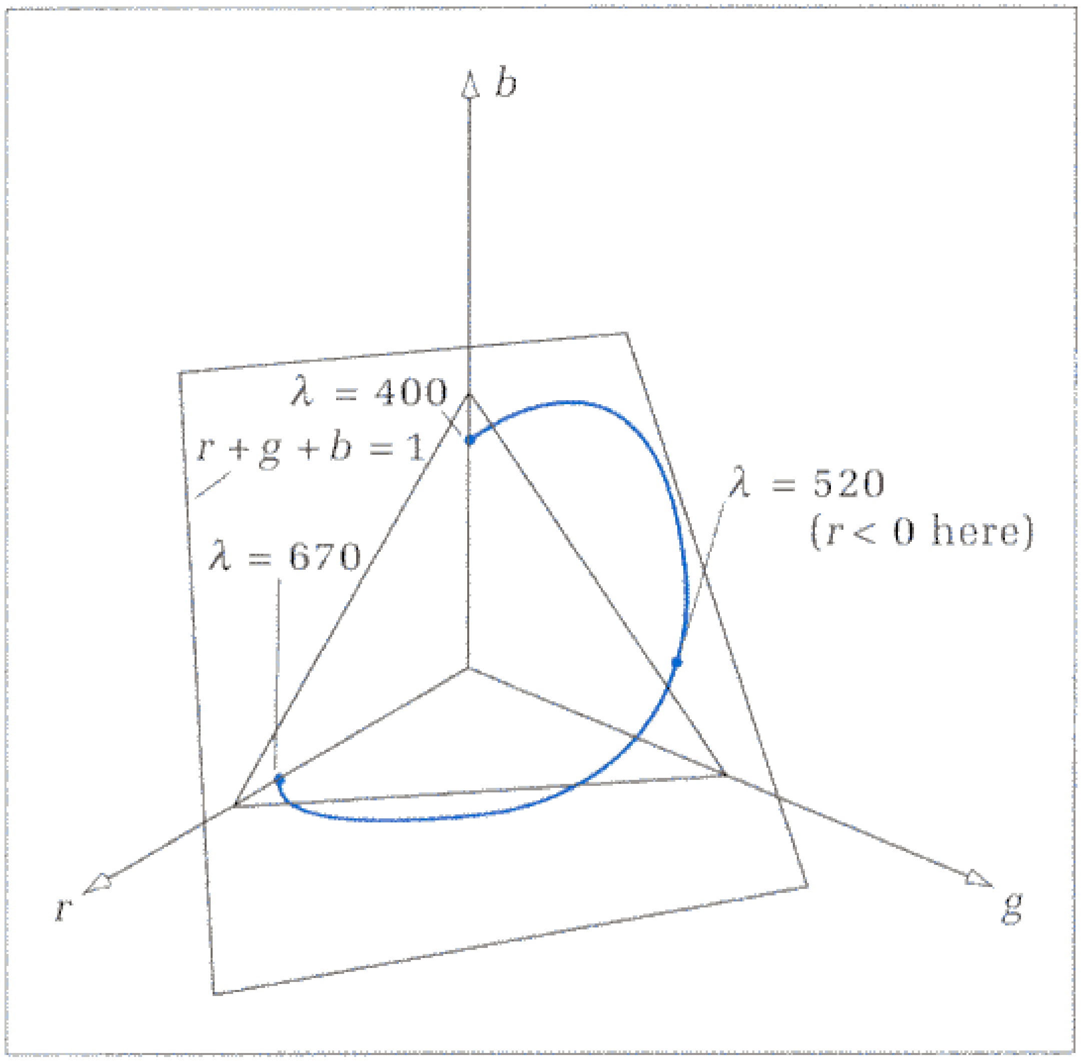
- S = P, means spectrum S and spectrum P are perceived as the same color
- if (S = P) then (N + S = N + P)
- if (S = P) then aS = aP, for scalar a
- It is meaningful to write linear combinations of colors T = aA + bB
- Color percepion is three-dimensional, any color C can be constructed as the superposition of three primaries:

$$C = rR + gG + bB$$

- Focus on "unit brightness" colors, for which r+g+b=1, these lie on a plane in 3D color space

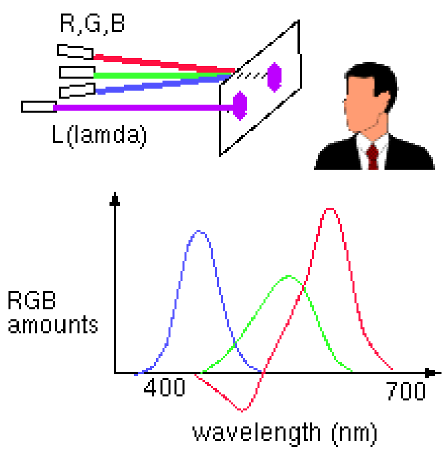
Saturated Color Curve in RGB

- Plot the saturated colors (pure spectral colors) in the r+g+b=1 plane



Color Matching

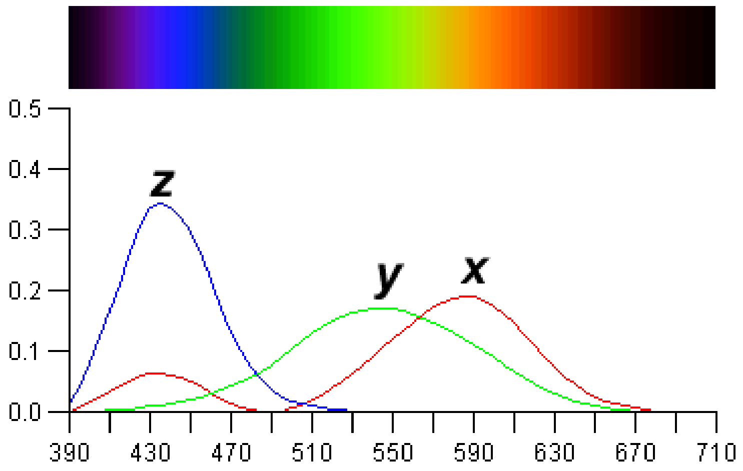
In order to define the perceptual 3D space in a "standard" way, a set of experiments can (and have been) carried by having observers try and match color of a given wavelength, lambda, by mixing three other pure wavelengths, such as R=700nm, G=546nm, and B=436nm in the following example. Note that the phosphours of color TVs and other CRTs do not emit pure red, green, or blue light of a single wavelength, as is the case for this experiment.



The scheme above can tell us what mix of R,G,Bis needed to reproduce the perceptual equivalent of any wavelength. A problem exists, however, because sometimes the red light needs to be added to the target before a match can be achieved. This is shown on the graph by having its intensity, R, take on a negative value.

CIE Color Space

In order to achieve a representation which uses only positive mixing coefficients, the CIE ("Commission Internationale d'Eclairage") defined three new hypothetical light sources, x, y, and z, which yield positive matching curves:



If we are given a spectrum and wish to find the corresponding X, Y, and Z quantities, we can do so by integrating the product of the spectral power and each of the three matching curves over all wavelengths. The weights X,Y,Z form the three-dimensional CIE XYZ space, as shown below.

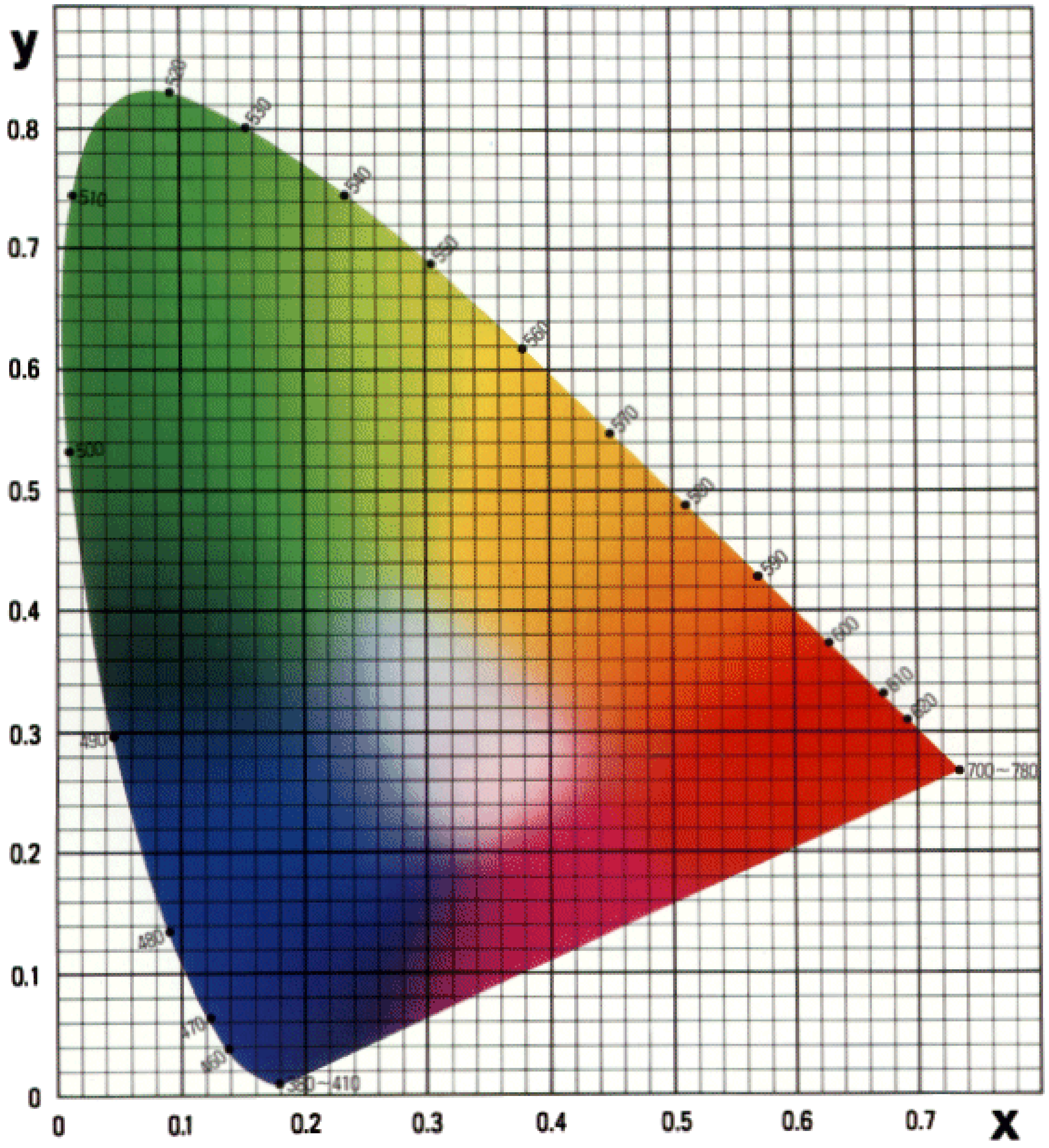
CIE Chromaticity Diagram

Often it is convenient to work in a 2D color space. This is commonly done by projecting the 3D color space onto the plane X+Y+Z=1, yielding a *CIE chromaticity diagram*. The projection is defined as:

$$x = \frac{X}{X+Y+Z} \qquad y = \frac{Y}{X+Y+Z}$$

$$z = \frac{Z}{X+Y+Z} = 1 - x - y$$

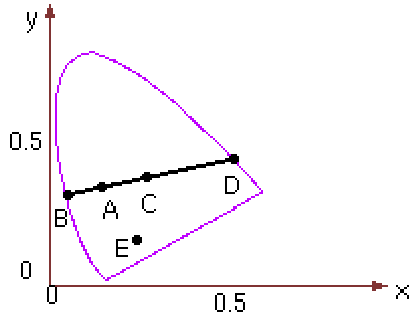
Giving the chromaticity diagram shown on the right.



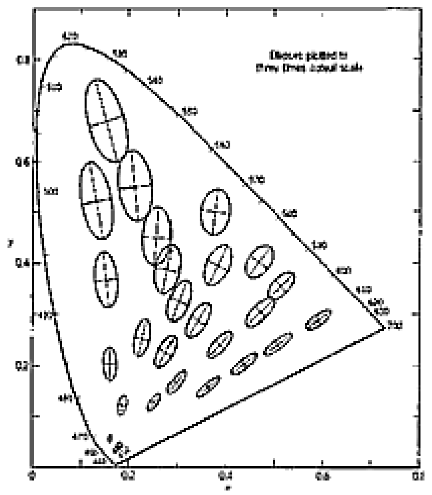
Previous | Lecture 2 | Slide 29 | 6.837 Fall '01 | Next

Definitions:

- Spectrophotometer
- Illuminant C
- Complementary colors



- Dominant wavelength
- Non-spectral colors
- Perceptually uniform color space



A few definitions:

spectrophotometer

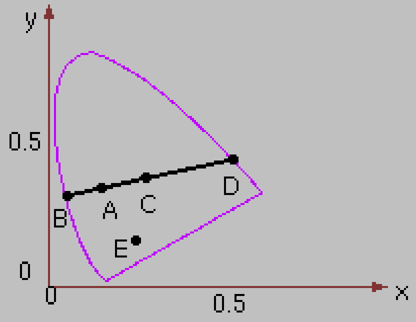
A device to measure the spectral energy distribution. It can therefore also provide the CIE xyz tristimulus values.

illuminant C

A standard for white light that approximates sunlight. It is defined by a color temperature of 6774 K.

complementary colors

colors which can be mixed together to yield white light. For example, colors on segment CD are complementary to the colors on segment CB.



dominant wavelength

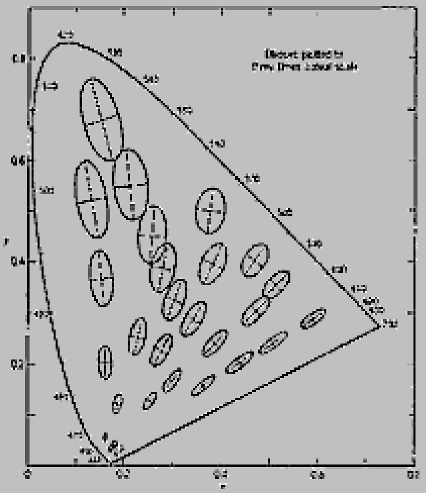
The spectral color which can be mixed with white light in order to reproduce the desired color. color B in the above figure is the dominant wavelength for color A.

non-spectral colors

colors not having a dominant wavelength. For example, color E in the above figure.

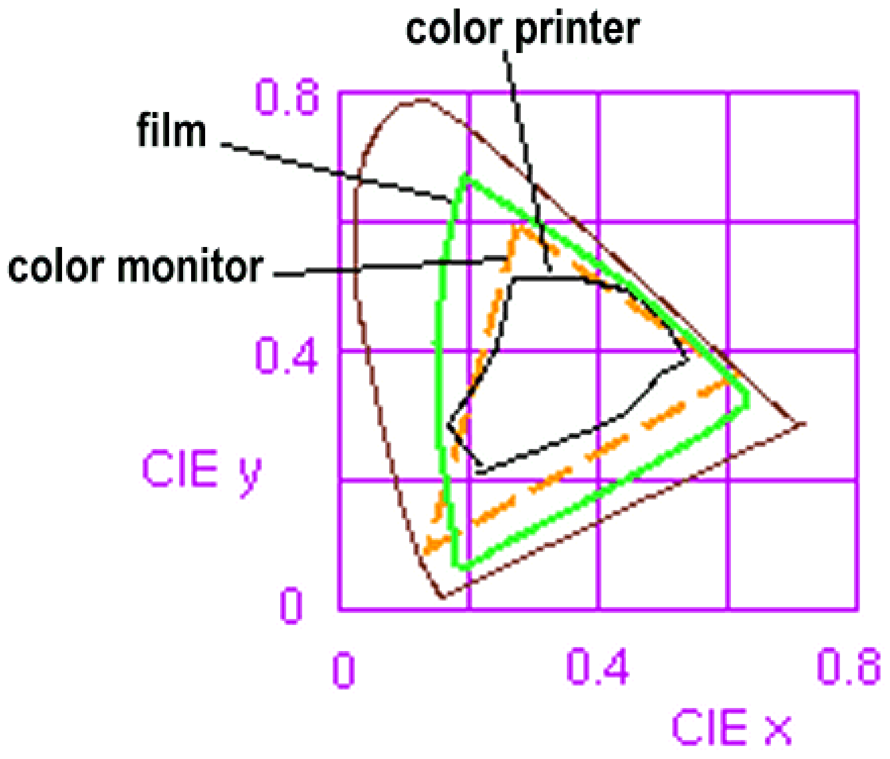
perceptually uniform color space

A color space in which the distance between two colors is always proportional to the perceived distance. The CIE XYZ color space and the CIE chromaticity diagram are *not* perceptually uniform, as the following figure illustrates. The CIE LUV color space is designed with perceptual uniformity in mind.



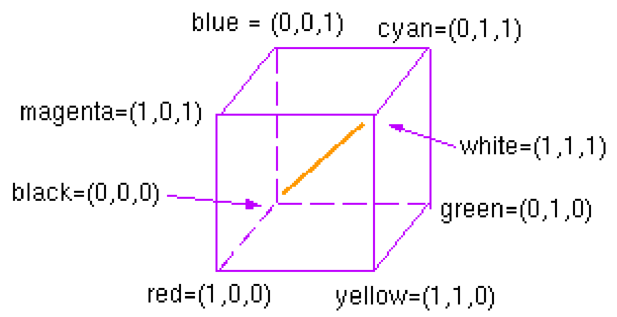
Color Gamuts

The chromaticity diagram can be used to compare the "gamuts" of various possible output devices (i.e., monitors and printers). Note that a color printer cannot reproduce all the colors visible on a color monitor.

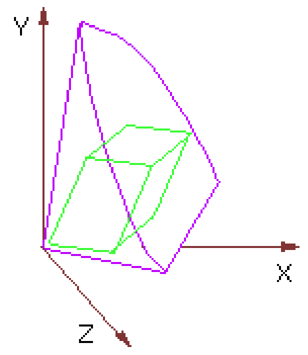


The RGB Color Cube

The additive color model used for computer graphics is represented by the RGB color cube, where R, G, and B represent the colors produced by red, green and blue phosphours, respectively.



The color cube sits within the CIE XYZ color space as follows.



Color Printing

Green paper is green because it reflects green and absorbs other wavelengths. The following table summarizes the properties of the four primary types of printing ink.

dye color	*absorbs*	*reflects*
cyan	red	blue and green
magenta	green	blue and red
yellow	blue	red and green
black	all	none

To produce blue, one would mix cyan and magenta inks, as they both reflect blue while each absorbing one of green and red. Unfortunately, inks also interact in non-linear ways. This makes the process of converting a given monitor color to an equivalent printer color a challenging problem.

Black ink is used to ensure that a high quality black can always be printed, and is often referred to as to K. Printers thus use a CMYK color model.

Color Conversion

To convert from one color gamut to another is a simple procedure. Each phosphour color can be represented by a combination of the CIE XYZ primaries, yielding the following transformation from RGB to CIE XYZ:

$$\begin{bmatrix} R' \\ G' \\ B' \end{bmatrix} = \begin{bmatrix} X_R & X_G & X_B \\ Y_G & Y_G & Y_B \\ Z_B & Z_G & Z_B \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

The transformation $\mathbf{C}_2 = \mathbf{M}_2^{-1}\mathbf{M}_1\mathbf{C}_1$ yields the color on monitor 2 which is equivalent to a given color on monitor 1. Conversion to-and-from printer gamuts is difficult. A first approximation is as follows:

```
C = 1 - R
M = 1 - G
Y = 1 - B
```

The fourth color, K, can be used to replace equal amounts of CMY:

```
K = min(C,M,Y) C' = C - K
               M' = M - K
               Y' = Y - K
```

Other Color Systems

Several other color models also exist. Models such as HSV (hue, saturation, value) and HLS (hue, luminosity, saturation) are designed for intuitive understanding. Using these color models, the user of a paint program would quickly be able to select a desired color.

Example: **NTSC YIQ color space**

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.30 & 0.59 & 0.11 \\ 0.60 & -0.28 & -0.32 \\ 0.21 & -0.52 & 0.31 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

Next Time

Flooding and Other Imaging Topics