Color

All of the images we have seen this far have been black + white, or grayscale, images. But we know that many, if not most these days, images are in full color.

We noted previously that our eyes can detect about 100 levels of gray in an image. But our ability to distinguish colors is much greater - most of us can see thousands of different colors, or more. So while eight-bit pixels suffice for grayscale images, we will want to use more bits for many color images.

RGB

We know that the cones in our eyes come in three varieties, Those most sensitive to red, to green, or to blue. All of the colors we perceive are sensed by these three types of cone cells and then interpreted as color in the brain.

Hence, one plausible way to have a computer generate color information is to use three separate images, one for the red channel, one for green, and one for blue. We refer images displayed this way as RGB images, for obvious reasons.

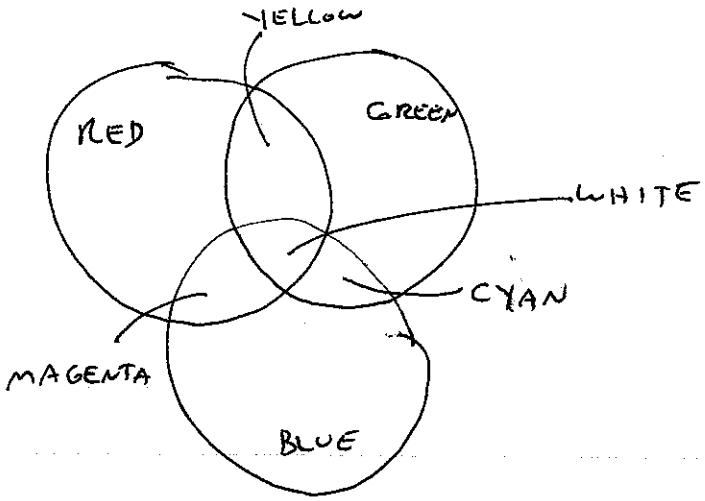
We might implement such ~~a~~ a color image by assigning three bytes to each pixel, one of red, one of green, and one of blue. This gives us 24 bits controlling color, so that we can represent $2^{24} \sim 16 \times 10^6$ different colors.

This method of storage is sometimes called true color, direct color, or 24-bit color.

Hence, by controlling the amount of red, green, and blue for each pixel, we can generate any color we want.

Color spaces

You have probably seen a simple color mixing chart such as this, which shows what you get by turning completely on or off each of the color components:



Here colors are combined using the subtractive process appropriate for light. Red and blue together make magenta, red and green make yellow, and red, green, and blue together make white.

With three separate color values needed to describe a given pixel, we can't easily plot all possible color values in a simple chart. Such a plot requires three axes, and hence must be plotted as a volume such as a cube. With a cube we can't see all of the "inside" colors at once.

So we need three values, or an ordered triple, to represent arbitrary colors.

Whereas for grayscale images we could use a single function $f(x,y)$ to represent brightness at each point in an image, for color data we need three functions:

$f_{red}(x,y)$ for the red intensities
$f_{green}(x,y)$ for the green intensities
$f_{blue}(x,y)$ for the blue intensities

Rather than deal with three separate functions, we could make $f(x,y)$ a vector function instead of a scalar function, with three values (r, g, b) at each point (x, y). This sometimes keeps the notation compact.

HSB

There are other ways to represent colors as well. But they all will require at least three different values, as does RGB because the total space occupied by the colors is three-dimensional.

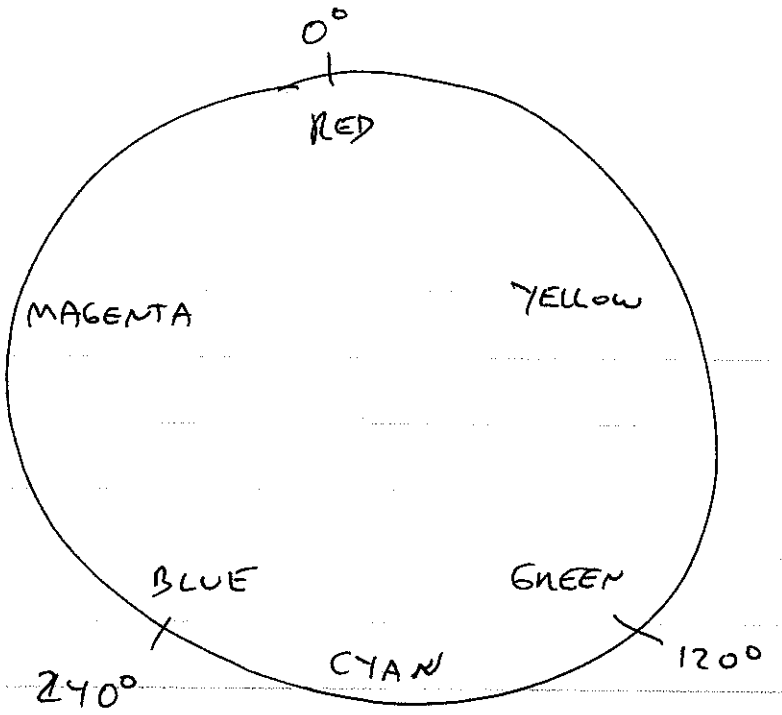
One common system other than RGB is HSB, or Hue-Saturation-Brightness. Sometimes this will be HSV (H-S-value) or HSI (H-S-Intensity), but it is simply another ordered triple that denotes different colors.

The definitions of the components in HSB are:

Hue - A number either from zero to one or from 0° to 360° that represents tint, or "pure" colors of constant brightness with at most two of the three R, G, B values non-zero.

Saturation - From 0 to 100%, it represents the amount of white added to a hue, and hence is the amount of the third color added to the two that form a hue. Any hue at 0% saturation is white (or grey). A 100% saturated color is as pure as it can be, say all red or all yellow.

Hue:



Note that yellow has a hue of 60° in this representation, and so forth.

Brightness: The total intensity of the color.

There is no easy, direct analytic function to convert RGB to HSB, but two algorithms that go back and forth are:

```
// r,g,b values are from 0 to 1
// h = [0,360], s = [0,1], v = [0,1]
//          if s == 0, then h = -1 (undefined)

void RGBtoHSV( float r, float g, float b, float *h, float *s, float *v )

{
      float min, max, delta;

      min = MIN( r, g, b );
      max = MAX( r, g, b );
      *v = max;                         // v

      delta = max - min;

      if( max != 0 )
            *s = delta / max;           // s
      else {
            // r = g = b = 0            // s = 0, v is undefined

            *s = 0;
            *h = -1;
            return;
      }

      if( r == max )
            *h = ( g - b ) / delta;     // between yellow &
magenta
      else if( g == max )
            *h = 2 + ( b - r ) / delta;    // between cyan & yellow

      else
            *h = 4 + ( r - g ) / delta;    // between magenta &
cyan

      *h *= 60;                         // degrees
      if( *h < 0 )
            *h += 360;

}
```

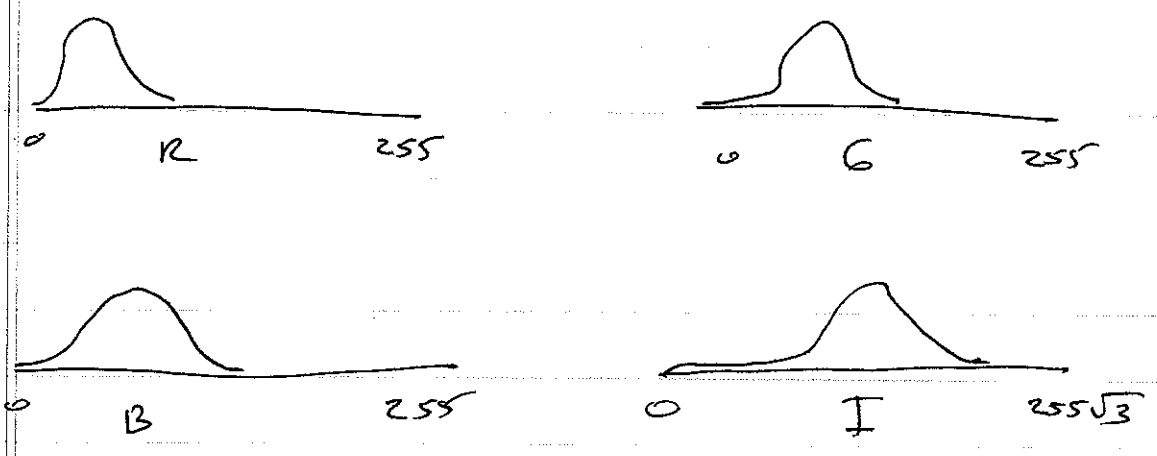
```
void HSVtoRGB( float *r, float *g, float *b, float h, float s, float v )

{
    int i;
    float f, p, q, t;

    if( s == 0 ) {
        // achromatic (grey)
        *r = *g = *b = v;
        return;
    }

    h /= 60;                    // sector 0 to 5
    i = floor( h );
    f = h - i;                  // factorial part of h
    p = v * ( 1 - s );
    q = v * ( 1 - s * f );
    t = v * ( 1 - s * ( 1 - f ) );

    switch( i ) {
        case 0:
            *r = v;
            *g = t;
            *b = p;
            break;
        case 1:
            *r = q;
            *g = v;
            *b = p;
            break;
        case 2
            *r = p;
            *g = v;
            *b = t;
            break;
        case 3:
            *r = p;
            *g = q;
            *b = v;
            break;
        case 4:
            *r = t;
            *g = p;
            *b = v;
            break;
        default:            // case 5:
            *r = v;
            *g = p;
            *b = q;
            break;
    }

}
```

Operations on color images

So, how do we go about optimizing contrast and so forth in a color image, where we now have 24 bits of data to consider? Usually we simply optimize each channel individually, and use the result. More complicated transformations, which we'll skip for now, include such items as principal components analyses.

We can thus simply do the same things we learned for 8-bit images on each channel to optimize pictures. Where it gets tricky is when we want to decide whether to let colors remain constant through the stretches or whether to allow them to change, resulting in false-color images.

Say we have an image in three channels, representing red, green, and blue. Let's generate 4 histograms for the image, one for R, one for G, one for B, and one for total intensity $I = \sqrt{R^2+G^2+B^2}$. Notice that while R, G, & B range from 0 to 255 as usual, I goes from 0 to $255\sqrt{3}$. Say our 4 histograms are:



We recognize these histograms as narrow and perhaps dark, so we can stretch them to get a more pleasant image.

Note: an alternate definition of I is simply R+G+B, in which case the histogram would go from 0–765 (3·255).

Say we wish to contrast stretch this image. We have several choices: Either apply the same scaling to all three channels, preserving the color balance, or scaling each channel individually, which maximizes our use of the color space but does not preserve the colors. This is ok if the image is false-color to begin with and we have no physical meaning in our colors.

If we want to keep colors the same, we derive a single transformation from the Intensity or from one or more of the channels, and apply the same stretch to all three channels.

If we want to maximize the color space, we simply operate on each channel independently and combine the result into a 3-byte (24-bit) quantity for display.

How about HSB Images?

We can do the same kinds of histogram-related stretches on hsb images too. Here if we only want to stretch overall brightness, we can apply a stretch to the B-channel only. If we want to use a particular color range, we can stretch or compress the hue channel. Similar arguments hold for saturation.

Indexed color

While manipulating and storing 24-bit color data is straight-forward, sometimes we want to either store data more compactly or have a more direct way to manipulate colors in an image. We can accomplish either of these by means of indexed color tables, where the pixel value of an image determines a color mapping rather than R G B directly.

This method is implemented using a color table, which is a list in the computer of which R, G, and B values we want to display for each byte value in an image. We have already used one such mapping when we displayed black & white images.

Recall that we have said before that for a b+w image, a pixel value of 0 maps into black, 255 into white, and intermediate values into grays. Because gray values are formed by mixing equal amounts of R, G, and B, we used the following definition:

Pixel value	Red	Green	Blue
0	0	0	0
1	$\frac{1}{255}$	$\frac{1}{255}$	$\frac{1}{255}$
2	$\frac{2}{255}$	$\frac{2}{255}$	$\frac{2}{255}$
3	$\frac{3}{255}$	$\frac{3}{255}$	$\frac{3}{255}$
⋮	⋮	⋮	⋮
255	$\frac{255}{255} = 1$	1	1

The display hardware in the computer reads each pixel value from our image, uses the color table to examine how much red, green, or blue to display at that pixel, and sets the monitor appropriately.

Note that there is no reason to use the above mapping. Say we wanted an image to appear as shades of red only. We might use the following:

Pixel value	R	G	B
0	0	0	0
1	$\frac{1}{255}$	0	0
2	$\frac{2}{255}$	0	0
⋮	⋮	⋮	⋮
255	1	0	0

If we want to invert the image and make it green, then the following works:

Pixel value	R	G	B
0	0	1	0
1	0	$\frac{254}{255}$	0
2	0	$\frac{253}{255}$	0
⋮			
255	0	0	0

We can define any mapping we want.

Indexed color is sometimes called 8-bit color, although some hardware supports 16-bit color as well. The idea is the same except that instead of 256 input levels, we have 16,384 (16 bits).

Manipulating color tables

Once an image is defined and displayed, then we can change the appearance by changing only the 256 entries in the color table, rather than by having to change the entire image.

This is an easy way, say, of choosing between two images quickly. We can do this by loading each image into 4 bits of each pixel with one image in the higher order bits and one in the lower order bits. Then we can simply load a color table that uses just the upper 4 or lower 4 bits, and easily go back and forth. Alternatively, we could make one image use one set of colors and the other use a different set. We often use this technique to compare two images on a screen.

A little thought shows that we can also implement image stretches and other intensity transformations by simply changing the color table.

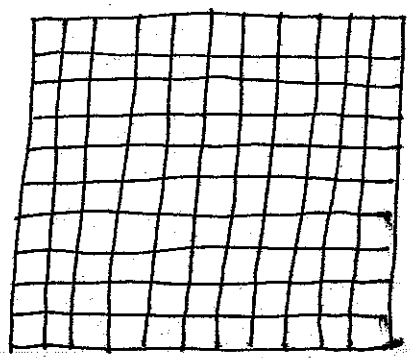
Practical Color Issues

We have seen that we can represent a color image similarly to the way we represent grayscale images, but with an ordered triple (r, g, b) for each pixel rather than a single number. But this implies a tripling of bandwidth or storage volume.

Recall that we noted early on that our eyes were more sensitive to detail in black + white than in color, due to the relative densities of rods and cones in our retinas. This suggests that it is possible to display images with lower color resolution than intensity resolution, and in fact this is how the existing television color standard in the US (NTSC) works.

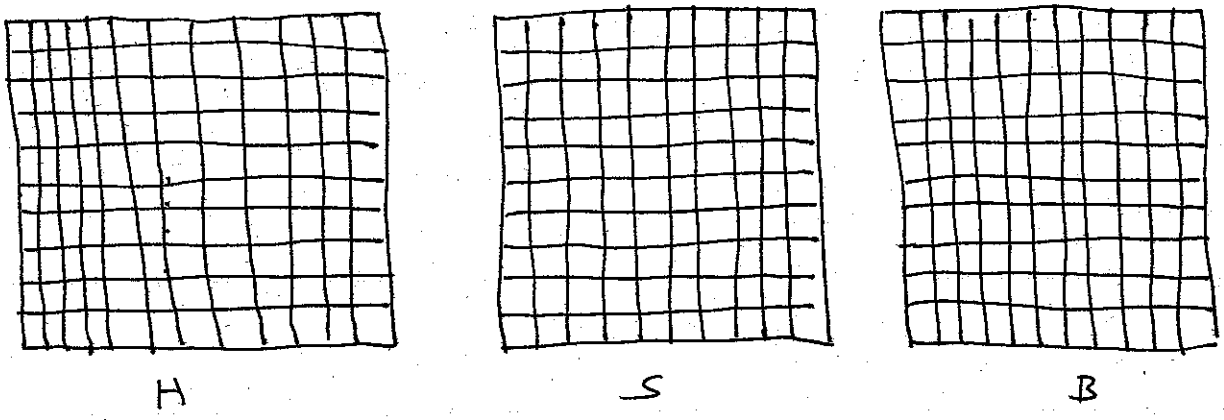
Consider breaking a color image up not into R, G, B channels but rather H, S, B channels. The brightness (B) channel carries all of the intensity information, while the ~~brightness~~ hue and saturation carry the color info. Then we can group together pixels in the hue and saturation channels to form larger pixels, so that the total number of bits is diminished. We retain the full resolution for the brightness.

Suppose this is our original image pixel spacing:

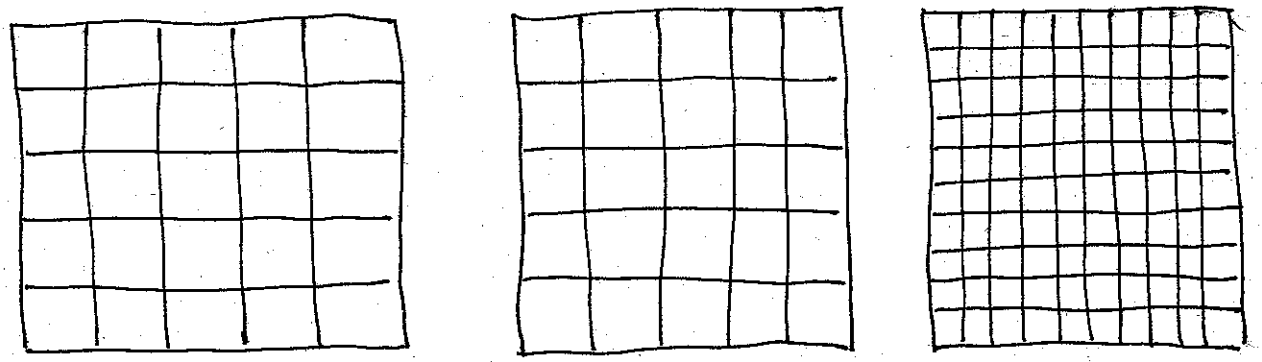


← Original Image RGB pixels

Separating into H - S - B:

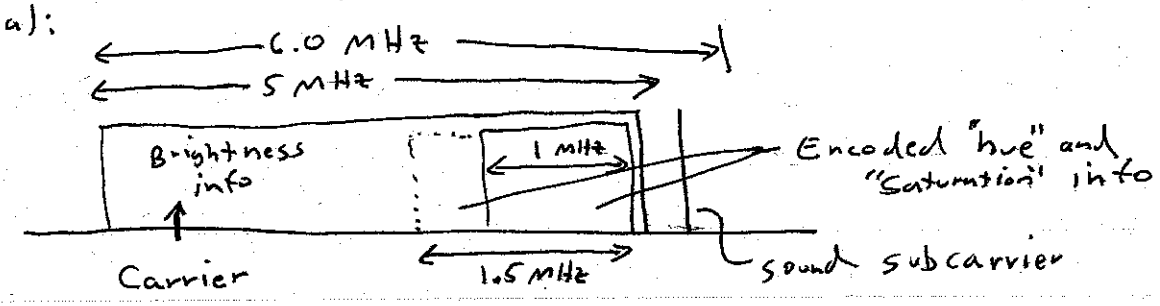


Lower the bit requirement by averaging H & S channels:



Note that there are far fewer pixels in the H & S channels. Transmit or store these reduced data to save space or time, and recombine to RGB when displaying. The brain will reconstruct the image properly.

This trick has been known by TV engineers for decades. Consider the spectrum of a TV broadcast signal:



Although some other tricks are used to encode the analog signal, the brightness is transmitted mainly over a 3.5 MHz part of the spectrum, with color information in a 1.5 MHz region. In fact the brightness data use slightly more than this, with color data on a coherent 3.579 MHz "subcarrier". What we want to remember is that we can send more information about brightness than color to best use the available spectrum.

If you are a circuits type, here is how rgb is encoded in a real NTSC box: