Spectral Colours

Spectral colours are pure unmixed colours, related to the wavelength of the light.

Hue, Saturation & Intensity

We've already met these discussing the eye! Remember the cones & rods discussion

Cones sense Hue (and indirectly saturation)

Rods sense Intensity
**Hue** is the colour we perceive.

It is related to the wavelength of the light (or electromagnetic radiation), that we can see.

For a pure (fully saturated) colour (i.e. light of a single wavelength, it is the "redness", "greenness" or "blueness" of the colour.

**Saturation** is its "pureness".

As we desaturate a colour we add white. As more and more white is added the colour becomes more "washed out" and the hue less distinct.

Desaturating an image is another way of converting a colour image to grey scale.

**Intensity** is the darkness or lightness of a colour.

Older books may refer to it as Lightness or Brightness (and HSI as HSL or HSI as HSB).

It is correctly the perceived luminance of the colour.
For many applications we need to be able to convert our image into an HSI model.

Where we can operate directly on a greyscale image, for example:

- edge detection
- contrast adjustment
- adjusting brightness

Operations such as these (and many others) only affect intensity for a colour image. So:

**Often we need to convert RGB to HSI.**

HSI is a cylindrical model. If you were to make a model one you'd need two dunces hats glued together.

- around the middle would be a rainbow of "pure" spectral colours
- one tip would be black
- the other tip would be white

- a band of different shades of a colour, going from white through pure to black would go along both of the hats.

- If we looked inside the hats, a line through the middle would be shades of grey, from white to black.
We represent hue as a circle going from 0 to 360 degrees (Red, Yellow, Green, Cyan, Blue, and Magenta).

Red 0°

Blue 240°

Green 120°

In colour:
From: http://www2.ncsu.edu/scivis/lessons/colormodels/color_models2.html
Intensity varies between 0 (black) and 1 (white)

From: http://www2.ncsu.edu/scivis/lessons/colormodels/color_models2.html

It is the line through the middle of our hats, when saturation = 0.
If saturation = 0, then:

<table>
<thead>
<tr>
<th>Intensity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.000</td>
<td></td>
</tr>
<tr>
<td>0.150</td>
<td></td>
</tr>
<tr>
<td>0.250</td>
<td></td>
</tr>
<tr>
<td>0.500</td>
<td></td>
</tr>
<tr>
<td>0.750</td>
<td></td>
</tr>
<tr>
<td>0.875</td>
<td></td>
</tr>
<tr>
<td>1.000</td>
<td></td>
</tr>
</tbody>
</table>

Saturation is the radius of the point. It is somewhere between the grey in the centre, of our hats and the rainbow colour band around the middle.

The maximum at any point, is the outside surface of our hats.

From: [http://www2.ncsu.edu/scivis/lessons/colormodels/color_models2.html](http://www2.ncsu.edu/scivis/lessons/colormodels/color_models2.html)

When saturation is 1 we have "pure" colour and are as far from black/white/gray as possible.
If saturation is 0, we have a gray, and hue is undefined (irrelevant).

This is the greyscale table on the previous slide.

If Intensity is 0, we have black and hue is undefined.

If Intensity is 1, we have white and hue is undefined.

If we keep Hue constant, we can vary intensity (or value) and saturation.

If we want to use an HSI model in, for example Adobe Photoshop, we can use a palette picker.
RGB to HSI conversion

Warning Maths ahead , (but don't worry).

We have now discussed two colour models and need to consider how to convert from one to another.
Normalisation

To convert RGB to HSI, we first need to normalise:

For many applications, we need to be able to map (or scale) our colour values onto whatever space is available.

When working on worksheet 2, we wanted to scale a random number to between 0 and 9 inclusive. The random() function however generated a number between 0 and RAND_MAX.

When we want to scale or remap values, normalising first is often useful.

Normalising simply means mapping 0…N on to 0.0…1.0

When working with P{G,P}M files we can take N from the image header, (or be lazy and use 255).

Normalising is easy:

\[
\text{Normalised Value} = \frac{\text{Current Value}}{\text{Current Max}}
\]

But remember: We are converting from integer to floating point!

**RGB to HSI conversion**

Consider:  
\[
\begin{align*}
R &= 30 \\
G &= 226 \\
B &= 217 
\end{align*}
\]
**Step 1:** Normalise your RGB image values

\[ I = \frac{(R+G+B)}{3} \]

**Step 2:**

\[ S = 1 - \frac{R+G+B}{3} \cdot \min(R, G, B) \]

**Step 3:**

\[ H = \cos^{-1} \left[ \frac{1/2[R - G + (R - B)]}{\sqrt{(R - G)^2 + (R - B)(G - B)}} \right] \]

**Step 4:**

If B > G however, H = 360° - H

**NOTE:** The C trigonometry library operates on values in radians. π radians = 180 degrees.

We therefore need to convert to and from radians and degrees. If we wish to normalise hue, then we need to scale by 2π.

So for our example

**Step 1**

R = 30/255 = 0.11765

G = 226/255 = 0.88627

B = 217/255 = 0.85098

**Step 2**

I = (0.11765 + 0.88627 + 0.85098) /3

I = 0.6183
Step 3

\[ S = 1 - ((3/(0.11765 + 0.88627 + 0.85098) * 0.11765) \]
\[ S = 1 - (1.61734 * 0.11765) \]
\[ S = 0.80972 \]

Step 4

\[ H = \cos^{-1} (x/y) \]
\[ x = (0.5*(-0.76862+-0.73333)) \]
\[ x = (0.5 * -1.50195) \]
\[ x = -0.75098 \]
\[ y = \sqrt{(-0.76862)^2 + (-0.73333 * 0.03529)} \]
\[ y = \sqrt{0.59078 + -0.02588} \]
\[ y = 0.75160 \]
\[ H = \cos^{-1}(-0.75098/0.75160) \]
\[ H = \cos^{-1}(-0.99918) \]
\[ H = 177.67482 \text{ degrees.} \]
**HSI to RGB conversion**

To convert HSI to RGB is more complicated:

**Step 1**: Determine the colour sector that (H)ue lies in.

- **RG** if \((0° < H \leq 120°)\)
- **GB** if \((120° < H \leq 240°)\)
- **BR**: if \((240° < H \leq 360°)\)

**Step 2**: Do the calculations specified under the appropriate heading.

**RG**

\[
B = I(1 - S) \left[ 1 + \frac{S \cos(H)}{\cos(60° - H)} \right]
\]

\[
R = 
\]

\[
G = 3i(1 - \frac{R + B}{3i})
\]

**GB**

\[
H = H - 120°
\]

\[
I \left[ 1 + \frac{S \cos(H)}{\cos(60° - H)} \right] G = 
\]

\[
R = I(1 - S)
\]

\[
B = 3i(1 - \frac{R + G}{3i})
\]
\[ BR \]

\[ H = H - 240^\circ \]

\[ G = I(1 - S) \]

\[ R = \frac{3i(1 - G + B)}{3i} \]

\[ B = \frac{S \cos(H)}{\cos(60^\circ - H)} \]

Returning to our example, we have:

\[ H = 177.67482^\circ \]

\[ S = 0.80972 \]

\[ I = 0.6183 \]

**Step 1**

GB since \((120^\circ < 177.67482^\circ \leq 240^\circ)\)

**Step 2**

\[ H = 57.67482^\circ \]

\[ G = 0.6183 \cdot (1 + (0.80972 \cdot \cos(57.67482^\circ)/\cos(60^\circ - 57.67482^\circ))) \]

\[ G = 0.6183 \cdot (1 + (0.43298/0.99918)) \]

\[ G = 0.88623 \text{ (note accumulated rounding errors)} \]

\[ R = 0.6183 \cdot (1 - 0.80972) \]

\[ R = 0.11765 \]

\[ B = 3 \cdot 0.6183 \cdot (1 - (0.11765 + 0.88623 / 3 \cdot 0.6183)) \]

\[ B = 0.85102 \]

We now need to denormalise to get our pixel values:

\[ R_{\text{pixel}} = (0.11765 \cdot 255) = 30.00075 \approx 30 \text{ (rounded to int)} \]
\[ G_{\text{pixel}} = (0.8863 \times 255) = 225.98865 = 226 \text{ (rounded to int)} \]

\[ B_{\text{pixel}} = (0.85102 \times 255) = 217.0101 = 217 \text{ (rounded to int)} \]

We've now got our original colour back!

Whilst the mathematics may look nasty, (although at worst, it is only simple trigonometry), given the equations it is quite straightforward to program this in C.

In C,

```c
malloc(image_size * sizeof(double) * 3)
```

- twice, once for normalised R G B, once for HSI

generate normalised values

apply equations

- 2 modularise
  - 21 write conversion as a function
  - 22 each sub-conversion as a function

- 3 test
  - 31 check your mathematics is correct with known values, e.g. the worked example above.

do whatever we want to with the image

convert back (apply equations)

denormalise

write file
Remember:

The C libraries expect angles in radians $2\pi$ radians = 360°
- $1\pi = 3.141592654$ (very approximately!)

The C header file `<math.h>` is needed to use trigonometry (and other mathematical) functions. You therefore need

```
#include <math.h>
```

The C libraries include `cos()`, `sin()` & `tan()` together with their inverses `acos()`, `asin()`, `atan()`. In other words, $\cos^{-1} 0.70$ would be written as `acos(0.70)`.

You will need to link to the maths libraries (-lm) to use the trigonometrical functions

- 1 If you want to use the workshop build button, you'll need to write a makefile. (tutorial on web site, handout available at CEMS helpdesk). Alternatively,

```
cc -o programe -lm programe.c
```

will work on the command line.
Point operations

Point operations operate on individual pixels. As a result, if we want to save memory, we can write the new values into our existing image as they are calculated.

Point operations are by far the easiest image transforms to code, but do present occasional problems.

Data item size

Often we will need to use an `int` temporary variable to hold results before writing the result into our image. An unsigned char holds values from 0 to 255.

If we wanted to add a constant (see Brightness) then we would need to use an integer to hold the result, (really we should look at using a `LUT` but more on that later!)

E.g. \(243 + 10 = \) ????

Underflow & Overflow

Related to the above, a simple problem with a simple solution.

Clamping

\[
\begin{align*}
\text{if } (x < 0) & \quad x = 0; \\
\text{if } (x > 255) & \quad x = 255;
\end{align*}
\]

If we clamp a value, we simply lock overflowed or underflowed values to the minimum or maximum value that a pixel (or colour value) can take.
**Point Operations: Adjusting Brightness**

Simple procedure:

**Grayscale images:**

- 1 Add constant

**Colour images:**

- 2 Simple (but rubbish) approach
  - 21 Divide constant by 3,
  - 22 Add result to R, G & B separately
- 3 Convert to HSI
- 1 Add constant to I, remembering I is normalised (0…1)
- 2 Convert back to RGB

We need to remember to scale an integer adjustment factor if we are working with colour images.

**Point Operations: Adjusting Contrast**

Again a simple procedure, but this time we require floating point arithmetic for a greyscale image.

**Grayscale Image:**

- 1 Compute product with constant. Almost as easy as brightness, but needs floating point arithmetic.

**Colour images:**

- 1 Convert to HSI
- 2 Compute product of I and constant, remembering I is normalised (0…1)
- 1 Convert back to RGB
Look-Up Tables (LUT's)

A LUT is simply an array to look values up in.

**Why use?**

Many image processing operations involve significant amounts of calculation (or computation).

Our program if written without a LUT would get significantly slower as the image size increased.

Imagine an operation that will take 2ms per pixel to compute.

- A 640 x 480 grayscale image will require 614 seconds.
- A 1280x1024 grayscale image will require 2621 seconds.

Most of the time will be consumed performing the arithmetic we require done. Actually storing values is usually very quick.

The reason that our programs will slow down with the image size is that we are performing our calculations on every pixel.

**But:**

Only 256 possible values can occur (0…255)

For the 1280x1024 image we are performing 1,310,720 calculations!

We could instead create an array of 256 values and use this to store each possible result.

We can then simply look up the value we require

```
output value = lut[input value];
```

```
int lut[8];
```
Consider Normalisation:

```c
void normalise(unsigned char *image,
                double *normalised,
                int image_size)
{
    double lut[256];
    int i;

    for (i=0;i<256;i++)
        lut[i] = ((double) i) / 255.0;
    for (i=0;i<image_size;i++)
        *(normalised + i) = lut[*(image+i)];
}
```
Histograms

Histograms are simply a form of bar chart where frequencies are represented as rectangles.

Histograms are important since they enable us to see the distribution of the pixel values.

Basic idea: chart of intensity

For greyscale: pixel value

For colour: normalised (or de-normalised) intensity

Consider the following colour image:

![Image of a colour image](image)

Convert to HSI, then use intensity (i.e. denormalise) to generate a value between 0 and 255. Alternatively, if we don't wish to return to a colour image, we can simply compute \( R + G + B / 3 \). Saving this as a greyscale image gives us:
We can now count the frequency of each pixel value that occurs.

```c
#include <stdio.h>
#include "ip.h"

main()
{
    unsigned char *image;
    int rows, cols, type, i;
    int hist[256];

    image = read_pnm("floppy.pgm", &rows, &cols, &type);
    for (i=0;i<256;i++)
        hist[i] = 0;
    for (i=0; i < (rows*cols); i++)
        hist[*(image+i)]++;
    for (i=0;i<256; i++)
        printf("%1d %1d\n",i,hist[i]);
}
```

Having counted our frequencies, we can graph these. This is our histogram:

Note the graph is cut to accentuate the distribution.
Many image processing packages let you implement a variety of histogram based techniques on images or their colour components.

Histograms represent intensity. We can compute intensity easily for an RGB image if we wish (normalised or unnormalised), or we can simply sum RGB (but maxval then becomes $3 \times 255$ – too big for unsigned char).

If we are working with colour images we need to ensure any changes to intensity are correctly distributed amongst the RGB components. This means that if we wish to perform an operation which alters intensity in a non-trivial way, we need to perform:

$$RGB \rightarrow HSI - Alter \rightarrow RGB$$

Greyscale images are much simpler to work with!

**Thresholding**

Thresholding is sometimes useful for extracting information from images. We may want to extract an object from the image or perform some operation...
tion which requires a mask. Thresholding is more frequently applied to greyscale images, however we'll continue working with the example floppy disk image used so far.

In order to threshold the image:

- read in file
- select threshold value
- generate bitmap using the following rule.

\[
\text{IF pixel < threshold THEN pixel} = 0 \text{ ELSE pixel} = 1
\]

- save bitmap

If we take a threshold value of 60, then our floppy disk image becomes:
Potential Problems

We need to write bitmaps!
Therefore we need to look at bitwise operators to “stuff” the bytes of the PBM file.

• 1 Each byte represents 8 pixels
• 2 & useful for turning bits OFF
• 3 | useful for turning bits ON

Look at the printing binary example in the C notes.

**Histogram equalisation.**

Spreads Histogram.

Often an image only uses part of the available range.

Histogram equalisation "spreads" the histogram over the full range (Note that the histogram for the floppy disk image is empty from value 212 upwards)

1. Compute histogram (Count intensity values & record in an array)

2. Create a 2nd array, containing the sum of pixels at up to that intensity. Hence cell 0 will contain the same value as cell 0 from step 1. cell 1 will contain the sum of cell 0 & 1, cell 2 the sum of cells 0, 1 & 2 etc..

2. Multiplying by max pixel value/number of pixels will even the distribution
   (if we want to round and have not normalised (i.e. are using integer values 0…255, add 0.5)

3. Use the result from (3) as a LUT to transform original image
Imagine we have a 8 greyscale levels image 10x10 pixels

<table>
<thead>
<tr>
<th>Pixel Value</th>
<th>Histogram (frequency)</th>
<th>Cumulative Frequency (x)</th>
<th>Equalised Value (for LUT) (x * 7/100) + 0.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>(0 + 0.5) = 0.5 = 0</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>8</td>
<td>(0.56 + 0.50) = 1.06 = 1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>9</td>
<td>1.13 = 1</td>
</tr>
<tr>
<td>3</td>
<td>45</td>
<td>54</td>
<td>4.28 = 4</td>
</tr>
<tr>
<td>4</td>
<td>40</td>
<td>94</td>
<td>7.08 = 7</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>100</td>
<td>7.5 = 7</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>100</td>
<td>7.5 = 7</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>100</td>
<td>7.5 = 7</td>
</tr>
</tbody>
</table>

**Contrast stretching.**

Images with poor (low) contrast are either too "light", too "dark" or too "grey". Generally poor contrast images only use a small subset of the available pixel values. Contrast stretching adjusts these images to use the full dynamic range.

In other words the histogram is "bunched up" towards one end, or the middle. For example,
histogram. This is similar to histogram equalisation, but not the same.
Two common approaches:

**basic**

"ends in search"

**Basic.**

1. Compute histogram
2. Find the highest & lowest value pixels in the image.
   \[
   new\_\ pixel = \frac{old\_\ pixel - low}{high - low} \times 255
   \]
   Assuming max\_val = 255:
3. Generate a LUT using the formula from 3.
4. Transform image.

As with most point operations, best implemented using a LUT.

**Ends in search**

More sophisticated.

1. User specifies percentage full black (B) & full white (W).
2. Compute histogram.
3. Find original pixel value that correspond to the full Black percentage. In other words sum frequencies counting up from value 0 until B% of the total number of pixels has been found. This pixel value becomes **low**.
4. Find original pixel value that corresponds to the full White percentage. In other words sum frequencies from 255 downwards until W% of the total pixels have been found. This pixel value is used as **high**.
5. Build LUT. Set output values for **low** or lower to zero, **high** or higher to 255, then proceed with.
   \[
   new\_\ pixel = \frac{old\_\ pixel - low}{high - low} \times 255
   \]
6. Transform image.
**Pseudo Colouring**

Often when we want to see information in a grayscale image adding colour can help.

For these cases we wish to present the image in colour. Here we simply build a LUT to translate graylevel to colour. One of the most common and useful is the "purple headed mountain". (see picture overleaf)

**From: Low, "Computer Vision & Image Processing"**

<table>
<thead>
<tr>
<th>Input Grey Level</th>
<th>Monochrome</th>
<th>Purple-headed mountain</th>
<th>Green</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R  G  B</td>
<td>R  G  B</td>
<td>R  G  B</td>
</tr>
<tr>
<td>0</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>1</td>
<td>3 3 3</td>
<td>0 14 0</td>
<td>0 4 9</td>
</tr>
<tr>
<td>2</td>
<td>5 5 5</td>
<td>0 17 0</td>
<td>0 8 8</td>
</tr>
<tr>
<td>3</td>
<td>7 7 7</td>
<td>0 20 0</td>
<td>0 12 8</td>
</tr>
<tr>
<td>4</td>
<td>9 9 9</td>
<td>0 25 0</td>
<td>0 15 8</td>
</tr>
<tr>
<td>5</td>
<td>11 11 11</td>
<td>19 27 0</td>
<td>0 18 7</td>
</tr>
<tr>
<td>6</td>
<td>14 14 14</td>
<td>23 31 0</td>
<td>0 24 6</td>
</tr>
<tr>
<td>7</td>
<td>16 16 16</td>
<td>35 42 0</td>
<td>0 27 8</td>
</tr>
<tr>
<td>8</td>
<td>21 21 21</td>
<td>49 49 0</td>
<td>2 34 8</td>
</tr>
<tr>
<td>9</td>
<td>27 27 27</td>
<td>53 33 0</td>
<td>17 36 9</td>
</tr>
<tr>
<td>10</td>
<td>31 31 31</td>
<td>63 26 0</td>
<td>22 40 10</td>
</tr>
<tr>
<td>11</td>
<td>37 37 37</td>
<td>54 0 0</td>
<td>34 44 11</td>
</tr>
<tr>
<td>12</td>
<td>43 43 43</td>
<td>63 0 0</td>
<td>46 48 11</td>
</tr>
<tr>
<td>13</td>
<td>50 50 50</td>
<td>52 0 53</td>
<td>57 57 0</td>
</tr>
<tr>
<td>14</td>
<td>58 58 58</td>
<td>54 29 58</td>
<td>63 63 21</td>
</tr>
<tr>
<td>15</td>
<td>63 63 63</td>
<td>63 63 63</td>
<td>63 63 63</td>
</tr>
</tbody>
</table>
Skull coloured with the "purple headed mountain" scheme (from *Low op. cit.*)

**Gamma ($\gamma$)**

Generally Gamma correction is used to compensate for the non-linear response of CRT's CCD's etc to brightness.

Consider the following graphs.

Vertical ($y$) axis is output.
Horizontal ($x$) axis is input.
gamma = 1
A gamma of 1 is the null transform, giving us a straight-line response between input and output:

0 < gamma < 1

An exponential dimming or darkening of the image

1 < gamma < ∞
A logarithmic brightening of the image

**How do we calculate gamma?**

\[ \text{output} = \text{input}^{\frac{1}{\gamma}} \]

Gamma is specified as:
Gamma of 1.0 represents the null transform.

The default gamma for a MS-Windows system is 2.2

Gamma transformation is easily implemented via a LUT

Again, for colour images we could work on intensity only, or on each colour channel, possibly with a different gamma.

Procedure:

1. Normalise pixel value (P) to generate (p)
2. LUT(P) = pow(p, 1.0/gamma) * 255.0
3. Clamp!
4. Apply LUT to image.

**Solarisation**

Often used for "effect", found as a standard filter in many image processing packages (e.g. Adobe Photoshop). Weird and wonderful effects, can apply to individual colour channels.

No practical value, (but fun)!

Define threshold.

If \( x > \) than threshold, output = 255 – x, else output = x.

Implement through a LUT.

What does our transformation curve look like?