QUICK COLOUR AVERAGING

Several of the algorithms on image scaling that I published, both in Dr. Dobb's Journal and on-line, depend on a function that calculates the average of two colours. Although this is a key function, the articles on image scaling left the averaging as an exercise for the reader. This paper fills that gap. (In fact, a summary of this paper was proposed as a sidebar for the article "Image Scaling with Bresenham" in Dr. Dobb's Journal of May 2002, but it was not printed.)

Calculating an average of two colours is easy in concept: all three channels of the source colours can be averaged independently. In the interest of performance, however, this paper discusses how to calculate an average of two 3-channel values *without* unpacking the source colours and repacking the destination pixel. In the interest of speed, the routines presented here drop a bit of quality, as the source colours are not reverse gamma-corrected before computing the average — and the resulting colour is not re-corrected for gamma. If the source colours are widely apart, the resulting average colour is too dark.

Averaging pixel values

Throughout the descriptions of the scaling algorithms presented in the earlier articles, there was an implicit assumption that calculation of an unweighted average of two colours is easy to calculate, regardless of the pixel format. This article presents calculation methods for a few pixel formats.

I start with discussing the pixel formats individually, using C/C++ macros for example implementations. The paper ends with a C++ listing with overloaded in-line functions that implement all the discussed pixel formats in a single file.

Gray scale pixels are the easiest. The code snippet below averages two grey scale pixels that are passed in as parameters **a** and **b**.

Averaging grey scale pixels

```
typedef unsigned char PIXEL;

#define AVERAGE(a, b)   (PIXEL)( ((a) + (b)) >> 1 )
```

24-bit RGB pixels can be thought of as three independent channels, where you process each channel individually. With this thought, you would call the grey scale averaging macro on the red, green and blue channels of a 24-bit colour pixel individually. There is a quicker way, however, which is based on the following observation:

$a + b = (a \wedge b) + ((a \,\&\, b) \ll 1)$

The average of a and b is "(a + b) / 2", which we can re-write as:

$(a + b) / 2 = ((a \wedge b) \gg 1) + (a \,\&\, b)$

An interesting property of the right hand of the above equation is that the addition will not overflow.

Assuming that the three RGB channels are packed in one 32-bit integer where the fourth byte of the integer is zeroed out, you can take the average of two of such integers in a single step. For example, the lay-out for a packed pixel is:

$A_7A_6A_5A_4A_3A_2A_1A_0$ - $R_7R_6R_5R_4R_3R_2R_1R_0$ - $G_7G_6G_5G_4G_3G_2G_1G_0$ - $B_7B_6B_5B_4B_3B_2B_1B_0$

I have named the fourth (zeroed out) channel **A** (for "auxiliary"). As written above, the XOR/AND algorithm to calculate averages will not cause overflows between the colour channels, but the "shift-right" instruction may cause an *underflow*: the low bit of, for example, the green channel may "flow into" the high bit of the blue channel. Fortunately, this is easy to correct by masking of the low bits of the three channels before the "shift right" instruction.

Averaging 24-bit RGB pixels

```
typedef unsigned long PIXEL;

#define AVERAGE(a, b)   ( ((((a) ^ (b)) & 0xfffefefeL) >> 1) + ((a) & (b)) )
```

HiColor modes with a bit resolution of 15 or 16-bits are quite popular; they give a nice compromise between number of colours, low memory (and memory bandwidth) requirements for bitmapped graphics and ease of use. To average two 16-bit HiColor pixels, you can use the same trick as the one for 24-bit pixels, but using a different mask (the mask is needed to clear any "underflow" bits).

Averaging 16-bit HiColor pixels (565 format)

```
typedef unsigned short PIXEL;

#define AVERAGE(a, b)   ( ((((a) ^ (b)) & 0xf7deU) >> 1) + ((a) & (b)) )
```

Palette-indexed, 256-colour pixels are harder to average. The quickest averaging method that I have come up with is a table lookup for each pair of possible palette indices. As the lookup table must be precomputed, which is a lengthy process, this is only suitable to scale multiple images based on a single palette. Fortunately, this is mostly the case.

The lookup table is 64 KiB in size: 256 rows × 256 columns where each element is the palette entry that best approximates the average colour of the two palette entries that select the row and column.

Due to symmetry in the lookup table, it suffices to calculate 33,024 (half of 64 k, plus 256 for the diagonal) average colours.

The lookup table is too large for the primary cache (L1 cache) on the CPU, and therefore the bottleneck in this routine may well be the memory fetch subsystem on the CPU.

Averaging palette-indexed pixels

```
typedef unsigned char PIXEL;
PIXEL average_table[65536];     /* precomputed */
#define AVERAGE(a, b)   average_table[((a) << 8) + (b)];
typedef __s_RGB {
  unsigned char r, g, b;
} RGB;
void MakeAverageTable(PIXEL *table, RGB *palette)
{
  int x, y;
  RGB m;
  for (y = 0; y < 256; y++) {
    for (x = y; x < 256; x++) {
      m.r = (unsigned char)( ((int)palette[x].r + (int)palette[y].r) / 2);
      m.g = (unsigned char)( ((int)palette[x].g + (int)palette[y].g) / 2);
```

```
    m.b = (unsigned char)( ((int)palette[x].b + (int)palette[y].b) / 2);
    table[(y << 8) + x] = table[(x << 8) + y] = PaletteLookup(palette, m);
  } /* for */
 } /* for */
}
```

The critical function inside **MakeAverageTable()** that is not discussed here, is **PaletteLookup()**. This function should return the palette index that comes closest to the given colour. Diverse methods exist to quantize a colour to a palette, for example, bit interleaving for RGB colours makes it suitable to sort a palette on the interleaved values and to find a closest match byBINARY search. An alternative is to create a 3-dimensional lookup table in an intelligent way; see for example the article by Leonardo Zayas Vila. For best quality, I propose a weighted Euclidean colour metric.

A side remark is that I specified the pixel format in the C++ listing below as a **signed char**. I did this to make function overloading work; the **unsigned char** type was already reserved to average grey scale pixels.

On the subject of overloading, note that in-line functions may carry non-functional overhead and that they may not be as fast as preprocessor macros; see the article by Fomitchev for details.

This is particularly important because the **average()** function appears in the inner loop of many of the image scaling algorithms.

Summary

Calculating the average colour of two pixels is indeed easy, even if we forego the obvious implementation of unpacking the source colours in their channels and repacking the averaged channels back into a colour.

Like many articles and books on the subject of image resampling and interpolating between source colours, this article ignored the requirement of converting the source pixels from non-linear RGB to linear RGB before the averaging (reverse gamma correction) and to convert the result back to non-linear RGB (gamma correction). At least, this paper mentions that it is, in fact, required.

Source: http://www.compuphase.com/graphic/scale3.htm