CS 559: Computer Graphics

Homework 2 Solutions

Question 1:

The $L^*u^*v^*$ space (defined below) is approximately perceptually uniform. Hence, one way to decide whether two pairs of colors in RGB space, say a, b and c, d, are separated by the same perceptual distance is to first convert all the colors into LUV space then compute their relative distances there using a standard distance metric. To get from RGB to XYZ, use the following matrix:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0.73467 & 0.27376 & 0.16658 \\ 0.26533 & 0.71741 & 0.00886 \\ 0.00000 & 0.00883 & 0.82456 \end{bmatrix} \begin{bmatrix} r \\ g \\ b \end{bmatrix} \tag{1}$$

For each of the following pairs of RGB colors, transform them into $L^*u^*v^*$ space. Then, for each pair, compute the distance between the two colors using the standard Euclidean distance function: *[10 points]*

$$d = \sqrt{(L_0^* - L_1^*)^2 + (u_0^* - u_1^*)^2 + (v_0^* - v_1^*)^2}$$

a. (1.0, 0.0, 0.0) and (0.9, 0.0, 0.0) *(58.7, 282.5, 19.4) and (56.2, 270.1, 18.5). Distance = 12.70*

b. (0.0, 1.0, 0.0) and (0.0, 0.9, 0.0) *(88.1, -176.9, 117.5) and (84.5, -169.7, 112.8). Distance = 9.38*

c. (0.0, 0.0, 1.0) and (0.0, 0.0, 0.9) *(8.1, -1.4, -47.5) and (7.3, -1.2, -42.7). Distance = 4.82*

This confirms our expectations for color sensitivity. Two greens appear further away than two blues, meaning that we can more readily distinguish between two greens than between two blues.

According to some sources, the matrix for converting RGB to XYZ was incorrect. It should be:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \begin{bmatrix} r \\ g \\ b \end{bmatrix} \tag{2}$$

This changes the numeric values for the answers, but not the relative differences. It does get rid of the negative (u^*, v^*) values.

Question 2: Consider an image format that uses indexed color (it stores a table of colors and then each pixel is an index into the table). Let n be the number of pixels in the image, let k be the number of bits for the index at each pixel, and assume that the table uses b bits for each color it stores.

a. How many bits are required to store the color table? *[2 points]*

   $2^k b$

b. How many bits are required to store the pixel data? *[2 points]*

   nk

c. How many bits are required in total? (Ignore the requirement to store the width and height or other information.) *[1 point]*

   $2^k b + nk$

d. Consider an image with 500000 pixels (about 800×600). Fix $b = 32$. How many bits are required if $k = 8$? How many are required for $k = 16$? *[2 points]*

   4008192 bits for $k = 8$, or about 0.5MB

   10097152 bits for $k = 16$, or about 1.2MB

e. For the same 500000 pixel image, fix $k = 8$. How many bits are required for $b = 64$? *[1 point]*

   4016384 bits, or still about 0.5MB

f. Is the size of a GIF file more sensitive to the number of colors in the color table, or the number of bits used for each color in the table? *[2 points]*

   The size of the file is more sensitive to the total number of colors, rather than the number of bits for each color in the table. Adding more colors requires adding more bits for every pixel and a bigger color table, while adding more bits to each color only adds to the number of bits required for the color table.

Question 3: Consider a variant of Floyd-Steinberg dithering in which the error at a pixel is distributed to only three of its neighbors:

e	+3/8
+3/8	+1/4

a. What happens if you run this version on an image with constant intensity of 0.5? What are the artifacts? *[2 points]*

This version will produce a checkerboard when run on a constant grey image. In general, it will produce more structured artifacts than the version given in class, and hence is inferior.

b. One possibility is to only distribute error to the pixel ahead of the current one on the same row. Why is this a bad idea? (Hint: Consider an image with a white left edge and a black right edge, with a gray ramp in between.) *[2 points]*

If you only distribute error horizontally, you tend to get vertical stripes in the dithered image. In particular, if you dither an image with vertical grey bands, you will get an image with vertical stripes running from top to bottom. To see why this is the case, consider a single row. The error is pushed from left to right until it goes over the threshold, at which point a pixel is produced. Then it is pushed again, until it goes over the threshold, and so on. Now, when the next row is processed it will go over the threshold at the same pixel locations*, giving vertical stripes as all the rows are processed. Even on general images, the error will accumulate at about the same rate from one row to the next (the rows don't differ by much), still resulting in near-vertical stripes. Humans tend to really notice vertical (or horizontal stripes), so emphasizing them in an image when they are not inherently there is a bad idea.*

c. Another possibility is to only distribute error to the pixels below the current one. Why is this a bad idea? *[1 point]*

If pixel errors are only pushed down, then the dithered image will tend to show horizontal stripes. The same argument as above explains this, and explains why it's a bad idea.