Jeff Linahan

Computer Vision Project 1

I chose to program this in the C and C++ programming languages rather than MATLAB due to my familiarity with the field of computer graphics. The code for loading and saving BMP files was downloaded from StackOverflow and uses the Windows API. The heart of the program logic, which I wrote in its entirety, is contained in functions consisting of double for loops which iterate over every pixel. Inside each of these double for loops is the code which computes the final color of the target pixel. For the grayscale and contrast stretch images, this is a fairly straightforward process. Some filter routines require setting up a kernel, this is represented by a matrix of floating point numbers and two more for loops to example every element of the kernel. Most images turned out very similar to what was seen in the project PDF. The Sobel images appear a little different, especially the threshold image. The Gaussian blur is difficult to see unless it is applied repeatedly.

Difficulties included:

* Loading a BMP image in C. I chose BMP due to its simplicity, but there is also no standard graphics interface in the language. I used code snippets online for the task originally written by a user named Andreas Hartl.
* Accommodating for the edges of the images when applying a kernel. If the program tries to apply a kernel outside the image edges, memory out of bounds is read. Alternatively, we can limit ourselves to pixels on the interior of the image by changing the for loop bounds.
* Off by one errors and the fencepost principle. The kernel matrix row and columns are numbered 0 to 3 or 0 to 5. However, when the kernel is applied the center square must be overlaid over the current pixel. Therefore, one must remember to subtract from the current row and column to overlay pixels to the upper left of the current pixel correctly.
* Normalizing the kernel. If this is not done, the color values can overflow and cause artifacts
* Remembering to convert from 0-255 to 0.0-1.0 color systems. If this is not done correctly, the output has artifacts or appears completely black. This can be especially hard to debug due to the sheer number of ways to draw a black screen.
* Clamping color values to the 0.0-1.0 range to prevent artifacts
* Remembering to make a copy for images that used a kernel (grayscale and contrast stretch can be done in place without allocating extra memory). If the input image is also the output image, some of the pixels adjacent to the current one will have been alerted since the program started, causing the current pixel to be incorrect.

Some of these difficulties could have been avoided by abstracting out the concepts of colors (especially bit depth and color space), images, kernels, and filters. I was not able to get the RGB to LAB color converted working due to logarithm of zero errors. If black is used as input the program simply crashes. The fact that my matrix library is of the opposite handedness of the rest of the world's and therefore need to be transposed also did not help.