MIT EECS 6.815/6.865: Assignment 1:

Basic Image Processing

Due Wednesday February 18 at 9pm

1 Summary

- New Image Class Methods and Gamma
- Colorspaces
- Spanish Castle Illusion
- White Balance

2 New Image Class Methods and Gamma

We have added operators to the Image class that allow you to add, subtract, multiply and divide images element-wise in the same way as built-in types like `float`. That is, you can now do things like

```
Image im1(640,480,3), im2(640,480,3);
float a = 2.0, b = -1.0, c = 0.0;
Image out1 = im1 + im2;
Image out2 = im1 - b;
Image out3 = a * im2;
Image out3 = im1/c; // This will throw a DivideByZeroException();
```

You can inspect the code in `Image.cpp`. When an operator is used with two images, they must be of the same size or a `MismatchedDimensionsException()` will be thrown.

We also added an `InvalidArgument` exception that you can throw using `throw InvalidArgument();` if you want to handle arguments that you think are not valid. Since this isn't a software engineering class, we won't test you on whether you handled invalid input correctly. If you do chose to use this exception, make sure the input is actually invalid.

2.1 Gamma

As mentioned in class, the intensity values in a typical image aren't proportional to the radiance in the original scene. They are *gamma*-encoded. That is, the radiance of the scene x is not stored, but instead x^γ (with $\gamma \approx 1/2.2$) is stored to reduce the effect of quantization on the darker tones. For some algorithms, such as simulating a different exposure time, we want to work in linear space.

2a) Implement `Image changeGamma(const Image & im , const double & old_gamma, const double & new_gamma)` in `a1.cpp`. You should assume the image's intensities are stored as $x^{\gamma_{\text{old}}}$ and you want to convert them to $x^{\gamma_{\text{new}}}$. You can do this using the built-in function `pow`.

2b) Implement `Image exposure(const Image & im, const float & factor)` in `a1.cpp`. This function should simulate changing the camera's exposure. That is, the intensity values should be modified by a multiplicative `factor` after the images are gamma-decoded ($\gamma = 1$). Assume the input image is encoded with $\gamma = 1/2.2$ and return an image that is also encoded with $\gamma = 1/2.2$.

3 Colorspaces

In this section, you will implement several functions related to changing an image from RGB colorspace to other colorspaces.

3a) Implement the function `color2gray` in `a1.cpp`, which performs a weighted average across color channels of an input image `im` and outputs a grayscale image. The weights are in the length 3 vector `weights`. The returned image should be a one color channel three dimensional image, not a two dimensional image.

3.1 Luminance-Chrominance

When we convert a color image to grayscale using the `color2gray` function, we get the *luminance* of the image, but lose the color information or *chrominance* (kr,kg,kb). You can compute this chrominance by dividing the input image by the luminance. Once the luminance and color information have been separated, you can modify them separately to produce interesting effects.

3b) Implement the function `lumiChromi` in `a1.cpp`. This function should return a vector of two images, a luminance image and a chrominance image. The luminance image should be the first element in the vector and it can be computed using `color2gray` with the default weights.

3c) Implement the function `brightnessContrastLumi` in `a1.cpp`, in which brightness and contrast of only the luminance of the image should be modified. Decompose the image into luminance and chrominance and then modify the luminance. Recombine the modified luminance with the chrominance by multiplying to produce the output image.

3.2 YUV

Another representation of an image that separates overall brightness and color is the YUV colorspace. An RGB image can be converted to and from a YUV image using the matrix multiplications

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.147 & -0.289 & 0.436 \\ 0.615 & -0.515 & -0.100 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix},$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1.14 \\ 1 & -0.395 & -0.581 \\ 1 & 2.032 & 0 \end{bmatrix} \begin{bmatrix} Y \\ U \\ V \end{bmatrix}.$$

3d) Implement the functions `rgb2yuv` and `yuv2rgb`, which convert images from one colorspace to the other.

3e) Implement the function `saturate(const Image & im, cont float & factor`, which multiplies the U and V channels of an image by a multiplicative `factor`. The input and returned image should be in RGB colorspace.

In YUV space, the elements of the image won't necessarily be in the range 0 to 1. If you try to write the image, the image write function will assume the input is an RGB image and it will round values outside of the range to the endpoints. Keep this in mind when testing and debugging your functions. Up to rounding errors, the functions `rgb2yuv` and `yuv2rgb` will be inverses of each other.

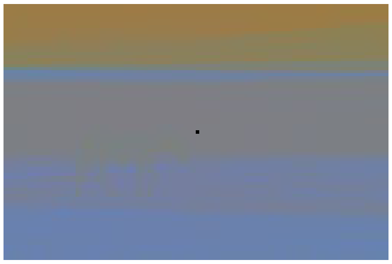
3.3 Discussion

The chrominance-luminance and the YUV conversions perform similar operations: they decouple an images intensity from its color. There are, however, important differences. YUV is obtained by a purely linear transformation, whereas our chrominance-luminance decomposition requires a division. Furthermore, the latter is overcomplete (we now need 4 numbers), while YUV only requires 3. YUV does a better job of organizing color along opponents and the notion of a negative is more perceptually meaningful. On the other hand, the separation between intensity and color is not as good as with the ratio computation used for luminance-chrominance. As a result, modifying Y without updating U and V changes not only the luminance but also the apparent saturation of the color. In contrast, because the luminance-chrominance decomposition relies on ratios, it preserves colors better when luminance is modified. This is because the human visual system tends be sensitive to ratios of color channels, and it discounts constant multiplicative factors. The color elicited by r, g, b, is the same as the color impression due to kr, kg, kb, only the brightness/luminance is changed. This makes sense because we want objects to appear to have the same color regardless of the amount of light that falls upon them.

4 Spanish Castle Illusion

You can use the colorspace functions you implemented to implement the Spanish castle illusion, which you can read more about at http://www.johnsadowski.com/big_spanish_castle.php.

Given an input image, you should create two images. The first image has a constant luminance (Y) and negative chrominance (U and V), and the second image is a black-and-white version of the original, i.e. both U and V should be zero. In the first image, set the luminance to be 0.5. To help people focus on the same location, add a black dot in the middle of both images. If image has dimensions $w \times h$, make sure that the black dot is at the 0-indexed location $\texttt{floor}(w/2), \texttt{floor}(h/2)$.



> 4a) Implement the function `spanish`, which takes an input image and returns a pair of images that can be used within the Spanish castle illusion. Make sure the gray scale image is the second element in the returned `vector`.

You can try out your function on the included `castle_small.png` and `zebra.png` or your own images.

5 White Balance

You will implement a function to white balance an image using the gray world assumption, in which the mean color of a scene is assumed to be gray. Specifically, you want to white balance an input image by multiplying each channel of the image by a factor, so that the mean value of each of the three channels of the output image is the same.

> 5a) Implement the function `grayworld` in `a1.cpp`, which automatically white balances the input image by using the gray world assumption. Make the average gray value of the output image equal to the average value of the green channel of the input image.

You can try out your function on the include `flower.png` image.

6.865 Only

6b) For what kind of images do you think white balancing with the gray world assumption will not produce a good result? Add your answer to a `README.txt` file in the asst directory.

6 Submission

Turn in your files to the online submission system (link is on Stellar) and make sure all your files are in the `asst` directory under the root of the zip file. If your code compiles on the submission system, it is organized correctly. The submission system will run code in your main function, but we will not use this code for grading.

In the submission system, there will be a form in which you should answer the following questions:

- How long did the assignment take?
- Potential issues with your solution and explanation of partial completion (for partial credit)
- Any extra credit you may have implemented
- Collaboration acknowledgment (you must write your own code)
- What was most unclear/difficult?
- What was most exciting?