MIT EECS 6.815/6.865: Assignment 1:

Basic Image Processing

Due Wednesday September 18 at 9pm

1 Summary

- The Image Class
- Brightness and Contrast
- More Image Class Methods
- Colorspaces
- Spanish Castle Illusion
- White Balance

2 The Image Class

Throughout this course we will be using the Image class introduced at the end of the last pset. The Image class specification is given in `Image.h`. Images are three dimensional arrays (width × height × color channel) that store pixels as a vector of floats called `image_data`. In memory, the pixels in the image are stored sequentially in row-major order in three adjacent color planes. That is, the distance (or *stride*) between adjacent values in the same row in `image_data` is equal to 1. The stride between adjacent values in the same column is equal to the width of the image and the stride between the different color channels at the same pixel is width×height. You can use the method `int stride(int dimension)` to compute these values.

`image_data` is a C++ vector, which is essentially an array which manages its own memory (as seen in pset0). You can access elements of it using brackets. For instance, `image_data[0]` returns the front element in the vector. More information about vectors can be found at http://www.cplusplus.com/reference/vector/vector/.

For images that correspond to pictures, the floating points in `image_data` will be between 0 and 1. There is nothing guaranteeing that the values of `image_data` stay in this range and there may be times when you want to use the Image class to store intermediate data that doesn't correspond to pictures, in which case the range is not meaningful. When the image is written to a file, it will assume the data lies in this range and will clip values outside of the range to one of the endpoints.

You can use the Image `write` method to write images to `.png` files. For example:

```
my_im.write("./my_image.png");
```

Then, you can view them in your favorite image viewer. This may be useful for debugging. Alternatively, if you don't feel like providing a filename, you can use `my_im.debug_write()` to write an image to an automatically named file. This might make debugging easier.

2.1   Pixel Accessors and Setters

You are going to implement the accessor and setter operators for pixel values in the image. We adopt the convention that the elements are accessed via the () operator. This is contrary to C++ convention, but will allow us to match the syntax of Halide (http://halide-lang.org/), which we will use at the end of the semester to write fast image processing code. That is, the pixel at location (x, y) in the third color channel of `my_im` is accessed via

```
my_im(x,y,2);
```

Note that there are three channels, but they are index using 0-based-index, that is the channels are accessed as $[0, 1, 2]$.

Implement three accessors with 0-indexing and bounds checking to make sure the input is valid. If the input is not valid, throw an exception using the command `throw OutOfBoundsException();`.

In `Image.cpp` implement:

1.a `number_of_elements()` : returns the number of elements in the image. An RGB (3 color channels) image of 100×100 pixels has 30000 elements.

1.b `my_im(x)` : returns the value of `image_data` at location x as long as x is less than ($<$) the total number of elements and at least ($>=$) 0. You need to implement the functions
`const float & operator()(int x) const`
`float & operator()(int x)`
with identical code. Use `number_of_elements()` for bounds checking.

1.c `my_im(x,y,z)` : returns the value at location (x, y) in the z-th color channel. You need to implement the functions
`const float & operator()(int x, int y, int z) const`
`float & operator()(int x, int y, int z)`
with identical code. Use `width()`, `height()` and `channels()` for bounds checking.

1.d `my_im(x,y)` : returns the value at location (x, y) in the 0-th color channel. You need to implement the functions

---

2

```
        const float & operator()(int x, int y) const
        float & operator()(int x, int y)
```

2.2   Creating Basic Images/Shapes

It will be useful for testing and debugging to have basic images and shapes in our image class. Currently when the Image class constructor is called it generates an all zeros image (when no filename is specified).

In `Image.cpp` implement:

2.a `set_color(float r, float g, float b)` : which sets all the pixels of the image to the specified values. That is, all pixels in the first channel should get the value `r`, pixels in the second channel should get values `g`, etc. Note that pixels should only be assigned if the channels are valid. I.e., if the number of channels is equal to 1, ignore the `g` and `b` values.

2.b `create_rectangle(int xstart, int ystart, int xend, int yend, float r, float g, float b)` : which sets the pixels inside the rectangle [xstart, ystart] → [xend, ystart] → [xend, yend] → [xstart, yend] (inclusive) to the specified color. If the rectangle coordinates are not valid, throw an out of bound exception (`throw OutOfBoundsException()`). As above, only set the color of valid image channels.

Note that both of these functions have default arguments http://en.cppreference.com/w/cpp/language/default_arguments (see their declarations), so that you can call them with a subset of the channels if appropriate.

And now for a treat, in `a1.cpp` implement:

2.c `Image create_special()` which returns an image where the following operations have been performed:

- Create an image of size $290 \times 150 \times 3$.
- Set all pixels to white (1.0, 1.0, 1.0)
- Draw rectangles: with coordinates (zero-based index) and colors:

Start (x,y)	End (x,y)	Color (r,g,b)
(0, 0)	(31, 149)	(0.64, 0.12, 0.20)
(52, 0)	(83, 102)	(0.64, 0.12, 0.20)
(103, 0)	(134, 149)	(0.64, 0.12, 0.20)
(155, 0)	(186, 30)	(0.64, 0.12, 0.20)
(155,48)	(186, 149)	(0.55, 0.55, 0.55)
(207, 0)	(289, 30)	(0.64, 0.12, 0.20)
(207,48)	(238, 149)	(0.64, 0.12, 0.20)

3

2.d Run the function `Image create_special()` and save the output image. What does the function create? (Answer in the submission form).

6.865 Only:

In `Image.cpp` implement:

2.e `create_line(int xstart, int ystart, int xend, int yend, float r, float g, float b)` : which sets the pixels of the line segment [xstart, ystart] → [xend, yend] to the specified color. There are more than one way to draw a line. We will accept the solution as long as it is not too far off.

3 Brightness and Contrast

Now for the fun part. Once we have these accessors, we can perform simple operations like increasing the brightness or contrast of an image.

3.a Implement the function `brightness` in `a1.cpp`, which multiplies the pixels in an image by the value `factor`. Make sure to create a new image and return that rather than modifying the input image. This is good practice in case you want to use the input again.

3.b Implement the function `contrast`, which increases the contrast of an image around a specified `midpoint` by the value `factor`. That is, you should apply the following function to every pixel's value

$$I_{out} = \texttt{factor} \times (I_{in} - \texttt{midpoint}) + \texttt{midpoint}.$$

You can test your functions on the included `Input/Boston_underexposed.png` and `Input/Boston_low_contrast.png`. The input images have low brightness or low contrast - you can use your `brightness` or `contrast` function to increase the brightness and the contrast.

4 More Image Class Methods

We have added operators for 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.

5 Colorspaces

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

> 4 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 two dimensional image with one color channel (instead of three color channels).

5.1 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.

> 5.a 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.

> 5.b 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.

We can think of the luminance as the "light intensity", and the chrominance as the "color information". Therefore, if the luminance of a specific pixel is 0, then a good assumption is that the pixel has no color (as far as images are concerned).

5.2 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

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

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

6.a Implement the functions `rgb2yuv` and `yuv2rgb`, which convert images from one colorspace to the other.

6.b Implement the function `saturate(const Image & im, 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.

5.3 Discussion

The chrominance-luminance and the YUV conversions perform similar operations: they decouple an image's "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.

5.4 Gamma-Encoding

An important element in digital cameras is the sensor. The sensor's main task is to record the amount of photons that make it through the camera's lens. Storing all of this information can be burdensome as it requires a lot of space. As a compromise, cameras directly quantize the "photon counts" into some specified number of bins (typically to 256 values (8-bits)).

We can examine a simple example of quantization, where we have 2-bits or 4 values. Say we take a photo, and the "photon count" in each of the pixels are: 260, 100, 513, 120, 12, and 42. Since we only have 2 bits, we can only store 1 of 4 possible values in each pixel. One way that we can quantize the image data is using uniform quantization, where we put the values into 4 equal-sized bins: $\{0 : 0 - 250, 1 : 250 - 500, 2 : 501 - 750, 3 : 750 - 1000\}$. That would give us quantized values of: 1, 0, 2, 0, 0, and 0, which we can store.

Uniform (also known as linear) quantization is an easy solution for the space problem, as we now need much fewer bits to store light information. However, humans perceive light in a non-linear manner (where mostly multiplicative effects or ratios of intensities matter). As a result uniform quantization is typically replaced with a gamma encoding quantization, where the stored values are $y = x^{1/\gamma}$. In this expression x is the light intensity and γ represents a power controlling the transformation (typically a value around 2). This encoding adds more details in part of the color range that humans are more sensitive to and reduce it where it matters less.

On high-end digital cameras, it is possible to store 'raw images'. These images represent minimally-processed sensor data. They are intended to provide as closely as possible the sensed light intensity and color in the scene. We have provided on of such an image in `./Input/skies_and_trees.png`. When this image is displayed by a device that assumes the data is gamma encoded, it might appear a bit dark and low contrast.

7.a Implement the function `Image gamma_code(const Image &im, float gamma)` in `a1.cpp` which adjusts the values of the input image by the specified gamma. Hint: a good self-check here is to write the output of this function when given our raw image as input and a reasonable guess for gamma. Did the colors improve?

7.b Implement the function `Image quantize(const Image &im, int bit)` in `a1.cpp` which uniformly quantizes the pixel values (in the range [0,1]) to `bit` bits (the output should also be in the range [0,1]). Hint: consider scaling up the input values by the necessary amount, rounding to the nearest integer and scaling back to [0,1]. Note: that this is 'fake' quantization as we have not saved any space (still using the same number of `floats`) in the range [0,1]; however, the number of unique values should be reduced to $\leq 2^{\texttt{bits}}$.

7.c Put it all together, implement `vector<Image> gamma_test(const Image &im, int bits, float gamma)` in `a1.cpp` :

- Create `im1` by quantizing the input image then gamma encoding.
- Create `im2` by gamma encoding the input image then quantizing.
- return im1 and im2 as a vector

7.d Try the function `gamma_test` on `./Input/skies_and_trees.png` with 4 bits, and gamma 1.8. Take a look at generated images. What are the perceptual difference between the images? Which image looks better? (Answer in the submission form).

6 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 its chrominance are the opposite of the input's chrominance (-U and -V). The second image is a black-and-white version of the original, i.e. both U and V should be uniformly 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 $\texttt{floor}(w/2), \texttt{floor}(h/2)$.
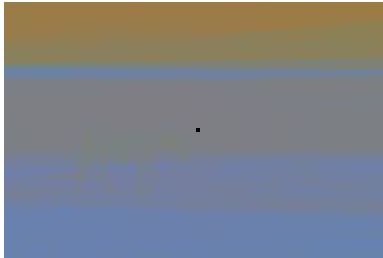
8 Implement the function `spanish`, which takes an input image and returns a pair of images that can be used within the Spanish castle illusion. Make sure the gray scale image is the second element in the returned `vector`.

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

7 White Balance

You will implement a function to white balance an image. We will be 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 average value of each of the three channels of the output image is the same. It may be helpful to write a function that computes the average value of each of the three channels.

> 9 Implement the function `grayworld` in `a1.cpp`, which automatically white balances the input image by using the gray world assumption. Make the average value of each of the three channels 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

> 10 For what kind of images do you think white balancing with the gray world assumption will not produce a good result? (Answer in the submission form).

8 Submission

Turn in your files to the online submission system 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. The submission system should also show you the image your code writes to the `./Output` directory

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

- How long did the assignment take? (in minutes)

- Potential issues with your solution and explanation of partial completion (for partial credit)

- Any extra credit you may have implemented and their function signatures if applicable

- Collaboration acknowledgment (you must write your own code)

- What was most unclear/difficult?

- What was most exciting?