Lab 4 - Image Processing

Prologue

For the past few labs, we have been working with OpenGL and 3D graphics; however, today we will be momentarily reverting back to 2D Canvas manipulation in order to introduce the next segment of the course: image processing. Throughout this lab, we will be applying different filters to images! This lab will help you prepare for the third project, **Filter**, which will require you to create slightly more complicated filters than the ones we introduce here. Let's get started!

Intro

Digital image processing is a field in which certain operations are performed on digital images in order to enhance or extract useful information from them. It is a type of **signal processing**, in which our inputs are two-dimensional signals, i.e. images! Unlike sound processing, which measures sound signals over one-dimensional time, image processing measures signals over two-dimensional space.

Image processing algorithms cover a wide range of applications, including the conversion of signals from analog sensors to digital images, removing noise artifacts, extracting semantic information, image compression, and image enhancement.

An image can be defined mathematically by the function $f(x,y)$ where x and y correspond to the horizontal and vertical axes, respectively. The value of the function at a certain point provides the color of the pixel defined at that particular set of coordinates. This 2D matrix of pixels represents the image **spatial domain**. This information may be transformed into another domain that you may have heard of in class: the **frequency domain**, which is obtained by applying a **Fourier Transformation** on an image that is currently in the spatial domain. In this lab, we will solely be working in the spatial domain.

Getting Started

The support code is located in */course/cs1230/src/labs/lab04*. Images for testing can be found in */course/cs123/data/image*.

You can install the support code for lab04 by running the below command in your terminal:
`/course/cs1230/bin/cs1230_install lab4`

Note that there is also a demo version of the lab that can be run by inputting */course/cs1230/bin/cs1230_lab04_demo* into your terminal.

Filter Set-Up

The `Canvas2D` class will be responsible for storing and manipulating the 2-dimensional canvas that will be displayed by the application window. This class has a `filterImage()` function that will be called by `MainWindow` whenever the filter button is clicked.

Task 1:

Let's set-up the `filterImage()` function to apply the appropriate effect when the filter button is clicked.

- Declare a filter object of type `std::unique_ptr<Filter>`

Task 2:

Use the information in the global `Settings` object in order to figure out what specific filter will be applied to the image.

- Head over to `Settings.h` to see the enumeration that contains all filter type names.
    - Reminder that enums can be accessed by `EnumName::EnumValue`
- Initialize the appropriate filter with `std::make_unique<className>()`
- Note that `FilterShift` takes in an additional argument, `ShiftDirection`

Tip: Use a switch statement to keep the code clean and readable!

Task 3:

Once the filter object has been initialized, apply the filtering effect.

- Call the `apply()` from the filter object.
- Make sure to send the `Canvas2D` so that the filter object has access to the image data.

Grayscale Filter

The first filter we will apply is a grayscale effect. In image processing, a grayscale image is one in which each pixel stores a single value that represents the amount of light passing through that point. In other words, the pixels only carry **intensity** information. These images are usually gray because each channel is set to the same value. Black (i.e. zero) signifies the weakest light intensity, and white (i.e. 255, since we have 8 bits per channel to store the sampled pixel data) represents the strongest light intensity.

Task 4:

Head over to the `apply()` method of the `FilterGray` class.

- Make a call to the `RGBAToGray()` and store the gray unsigned `char` value that represents the light intensity at `current_pixel`.

Task 5:

We will use the `RGBAToGray()` method in order to calculate the appropriate shade of gray that represents the intensity of the pixel. This function will map the pixel's three RGB values to a single light intensity value.

- Return the correct gray value by computing a weighted sum of the R, G and B components of the incoming pixel.
- Make sure that the image does not keep getting darker as you apply the grayscale filter.
- The method to map RGB values to a the single gray intensity is not unique. For example, the simple **average method** computes the average between the three color channels:

$$Y = \frac{R + G + B}{3}$$

- The **lightness method**, which will desaturate the image, averages the least prominent and most prominent values:

$$Y = \frac{MAX(R, G, B) + MIN(R, G, B)}{2}$$

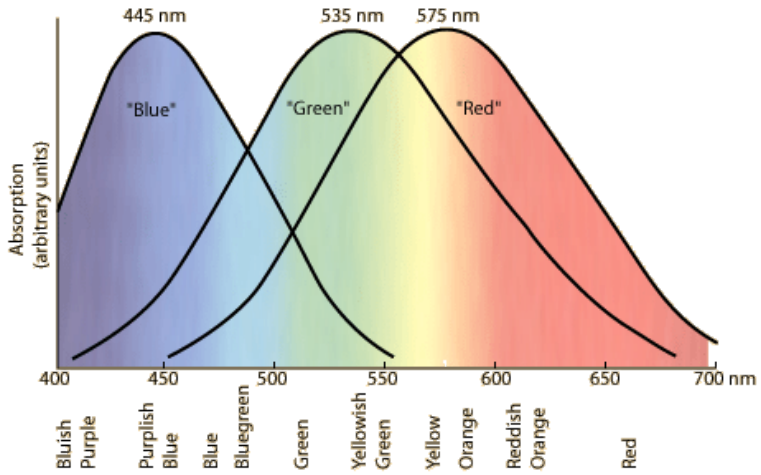
- The **luminosity method** calculates a weighted sum between the three color channels using percentages that account for the human perceptual system (we recommend this!)

$$Y = 0.299 * R + 0.587 * G + 0.114 * B$$

Tip: Use integer math to prevent image darkening.

What does this do?

Cones are the photoreceptor cells of the human visual system. There are three types of cone cells, each sensitive to a different (albeit overlapping) section of the visible light spectrum. The response curve of each cone cell type roughly corresponds to the colors red, green and blue (see image below). We have have a higher density of red and green-responsive cone cells, so we are more sensitive to red and green light. For this reason, the red and green channels are weighed more heavily in the **luminosity method** shown above. Green is weighed most heavily of all the other channels because rods, the other major cell-type in the eye, respond very little to red light as opposed to green and blue light; but this is beyond the scope of this lab!



- The luminosity method is the most sophisticated approach we have covered, and it is the one we recommend for this lab, but feel free to try out each of these methods and compare their results!
    - The average method should produce a more washed-out image compared to the luminosity technique since it doesn't take into account the eye's varying color sensitivities.
    - The lightness method should produce the image with least contrast of all three methods since it desaturates the pixel by setting its color to the midpoint between its maximum and minimum values.

Task 6:
Set the final color of the pixel using the value `RGBAToGray()` returns.

- Update the pixel, which is referenced by the `current_pixel` pointer. Remember to set all three color channels individually by accessing them from `current_pixel` directly!

Invert Filter

The next filter we will implement is image inversion. Also known as the negative effect, this method works by inverting the value of each color channel for a pixel. In order to invert a channel, we subtract its value from the maximum color value, which in this case is 255.

Task 7:
In the `apply()` method of the `FilterInvert` class,

- Invert each color channel of `current_pixel`.

Tip: If you take the inversion of the inverted pixel data, you should end up with the original image! Test this out by simply double-filtering an image with the invert filter.

[Optional] Task 8:
Now we are going to try a different approach using a neat bitwise operator trick! This only works because the size of `CS123Color` is equal to the size of `unsigned int`.

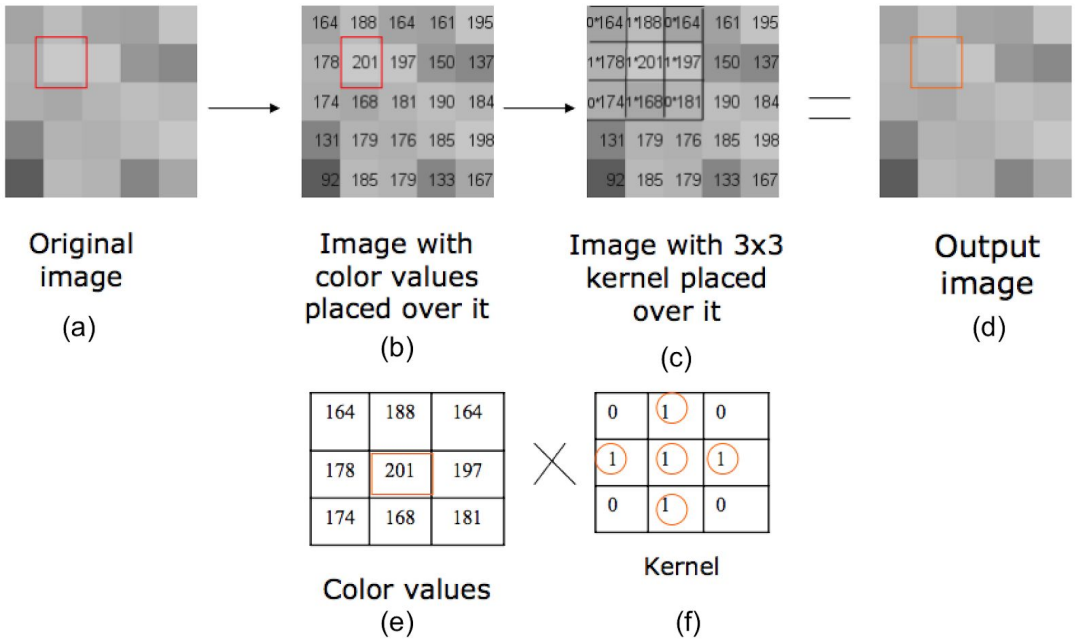
- Obtain a pointer to the canvas data and cast it to an `unsigned int`
- Iterating through the data with the pointer, simply take the bitwise complement of the pixel color.

Convolution

Up until now, we have written filters that deal with each pixel independently, i.e. the final color of a specific pixel in the grayscale and inverse filters did not depend on neighboring pixels. We will now introduce the concept of **convolution**, which allows us to apply operations to a pixel while taking into account information obtained from its neighbors in the spatial domain.

Convolution is one of the most fundamental operations in image processing. It can be used to alter images, as well as extract information from them (for this reason, it is used for dataset analysis in convolutional neural networks). The convolution operation relies on the use of **kernels**, which are two-dimensional matrices of varying dimensions.

Let's look at the example below.[1] For simplicity, let us represent our image data with only one color channel, as shown in (b). We will apply the kernel to the pixel highlighted in red, which contains the intensity value **201**. Our kernel, shown in (f) is a 3x3 matrix. In image (c), we place the kernel over the pixel and multiply each kernel value with the corresponding neighboring pixels that surround our red-highlighted pixel.



To obtain the final value that will be used to update the pixel of interest (i.e. the pixel with intensity **201**), we will add up all of the multiplications in image (c) to obtain a final value of

$$(0 * 164) + (1 * 188) + (0 * 164) + (1 * 178) + (1 * 201) + (197 * 1) + (0 * 174) + (168 * 1) + (1 * 181) = 932$$

Before updating the pixel with this value, we must divide by the sum of the kernel coefficients to preserve overall brightness. The final value will be:

$$932/5 = 186.4$$

As you can probably guess, convolution in two dimensions is very expensive to perform due to the rapid accumulation of multiplication operations when kernel dimensions become large. However, if the kernel is **separable**, then the computation can be reduced to $M + N$

[1] http://web.pdx.edu/~jduh/courses/Archive/geog481w07/Students/Ludwig_ImageConvolution.pdf

multiplications. You will need to implement separable kernels for at least one filter in the upcoming filter assignment. For this lab, we will only cover convolution with 2D kernels.

Tip: Since convolution becomes very inefficient with large images, we recommend you use small to medium-sized data for testing.

Task 9:
We will now fill in the `Convolve2D()` method in the `FilterUtils` namespace. We need to store the new image data in a different `RGBA` array so as to not overwrite the original pixel data during the convolution process.

- Initialize a `result` array to store the new `RGBA` values, using the incoming image `width` and `height`.

Task 10:
In order to iterate through the kernel, we will need its width and height,

- Obtain the dimensions of `kernel` and initialize it at the beginning of the convolve function
    - Since we can get `kernel.size()`, use a math function to determine the dimensions (refer to the tip below). We included `math.h` for you!

Tip: Two dimensional kernels will usually be square matrices since they tend to be symmetric in practice, so you can assume that the width and height are equal in this implementation. You can also assume that the width and height are always odd, since the kernel must be centered around one pixel.

Task 11:
The final pixel color will be the summation of the kernel applied to the current pixel as well as its neighbors.

- Initialize `red_acc`, `green_acc`, and `blue_acc` *float* variables to store the accumulated color channels.
- Recall that `RGBA` stores channel information as integers. The kernel however, is defined by floats. You will need to convert the pixel data to a float before applying the kernel's value to it.

Task 12:
We will now apply the convolution kernel on every pixel as we iterate over the image data.

- Iterate over the kernel by creating a nested for-loop using the kernel dimension calculated in Task 10.
- You will need two values at each iteration of the for-loop: the current value of the kernel and the value of the pixel that corresponds to that kernel element.

Tip: The index of the current kernel element can be obtained by using the current kernel row, current kernel column and kernel width. The index of the current pixel can be obtained

similarly; however, you will need to perform additional steps in order to find the current row and column of the image that pertain to that kernel index.

- Update `red_acc`, `green_acc`, and `blue_acc` with the corresponding pixel value multiplied by the value of the kernel at that for-loop iteration.

Task 13:
You may have noticed an issue when applying kernels with width and height greater than 1: the kernel extends beyond the boundary of the image, where pixel data is not defined. There are various options that you can explore further in the filter project, for now we will simply ignore these pixels.

- Check whether the current neighboring pixel index is outside the image bounds; you can use the `continue` keyword to jump to the next for-loop iteration.

Task 14:
Update the `result` array at the center pixel index.

- Create a `RGBA` variable with the `red_acc`, `green_acc`, and `blue_acc` floats that have been accumulated.
- Use the `REAL2byte()` utility function that has been provided to convert the float back to an integer between *0* and *255*.
- During the convolution process, one must divide the accumulated intensity by the sum of the kernel coefficients in order to preserve overall brightness. We will ignore this for now since our Identity and Shift filters do not risk increasing image brightness; however, you will need to take this into account for your Filter project.

Task 15:
Copy the `result` array to the canvas image data.

- Use `memcpy()` values to transfer the data from the `result` buffer array to the canvas image data.
- The first argument to this function is the destination, while the second argument is the source. The function also requires a third argument, the number of bytes to copy. You will need to obtain this value by using the canvas dimensions, as well as the size of the data type that we use to store the image color intensities.

Tip: The C++ keyword `sizeOf()` returns the size in bytes of a given data type

Identity Filter

The identity filter will perform the convolution using an identity kernel, i.e. a kernel that, once convolution has been performed, results in the original image.

Task 16:
We will need the identity filter to use 2D convolution to filter the canvas data that is sent from `Canvas2D` when the filter button is clicked,

- In the `apply()` method of the `FilterIdentity` class, call `Convolve2D()` from the `FilterUtils` namespace, sending the appropriate canvas information.

Task 17:
The `Convolve2D()` method in `FilterUtils` takes in a kernel that will be used during the convolution process. We will need to initialize it for the identity filter.

- In the `FilterIdentity` initializer list, initialize `m_kernel` to be a vector of floats.
- The size of the vector may be any *odd* number; however, remember that once the kernel size gets too large, 2D convolution becomes very slow.

Shift Filter

The shift filter will shift the image one pixel to the left or right, depending on the value of `m_shiftDir`.

Task 18:
Just as above, we will need the shift filter to use 2D convolution to filter the canvas data that is sent from `Canvas2D` when the filter button is clicked,

- In the `apply()` method of the `FilterShift` class, call `Convolve2D()` from the `FilterUtils` namespace, sending the appropriate canvas information.

Task 19:
Now we will need to initialize `m_kernel` for the shift filter.

- In shift filter initializer list, initialize `m_kernel` to be a vector of floats.
- Use the `m_shiftDir` variable to check whether the filter being used is a shift right or shift left filter.
- Depending on the value of this `m_shiftDir` variable, create the appropriate kernel to shift the image *by one pixel*.

End

Now you are ready to show your program to a TA to get checked off!
Be prepared to answer one or more of the following:

- What do the pixel values in a grayscale image represent?
- Describe the process of convolving an image with a kernel.
- What happens when the kernel exceeds the boundary of an image?
- Provide a kernel to shift an image three pixels downwards.
- Provide a kernel to shift an image five pixels upwards.

Food for thought

We only covered three algorithms for the grayscale filter, this blog-post covers various other options: http://www.tannerhelland.com/3643/grayscale-image-algorithm-vb6/

In this lab, we have only dealt with the spatial domain of images since we have implemented filters that are focused on the spatial location of pixels and their neighbors. If you are interested in the frequency domain, read about frequency filters, such as the high-pass and low-pass filter. Here is a useful tutorial to get you started:
https://homepages.inf.ed.ac.uk/rbf/HIPR2/freqfilt.htm

Convolution in the spatial domain corresponds to multiplication in the frequency domain. This is quite handy since multiplication is faster than convolution. You may consider converting images to the frequency domain when spatial convolution kernels become too large and inefficient!

Another option to deal with large inefficient 2D kernels is to separate them into two 1D kernels! To learn more about convolution with separable kernels, checkout this handout:
https://cs.brown.edu/courses/cs123/handouts/filter/edgedetection.pdf