Project 3 Filter
Introduction to Computer Graphics, Fall 2015

1 Introduction

In this assignment you will be exploring the area of image processing. Image processing can be as simple as cropping an image to fit in a certain space, or can be as complicated as removing sinusoidal noise in the image generated as a side effect of the medium through which you obtained it. The realm of image processing covers an infinite space of operations that can be applied to images and the combinations thereof. You will be implementing just a few of the most common image processing algorithms (which, conveniently, happen to be the more useful ones).

Play with the demo by running /course/cs123/bin/cs123.demo. A wide variety of images for testing may be found in /course/cs123/data/image. Using the demo for this project is as simple as selecting a filter, dragging an outline box on the image with the right mouse button (this is assuming you have loaded an image), and clicking “Apply Filter” to apply the current filter to the selection. Clicking outside the box with the right mouse button will cancel the selection. If there is no selection, the filter will be applied to the entire canvas. Several of the filters have their own controls which customize filter-specific parameters. There are also options for extra filters. The TA demo includes a few extra filters.

2 Requirements

You will need to implement the following filters for this assignment:

1. Invert Colors
2. Grayscale
3. Edge Detection/Sobel
4. Blur (triangle or Gaussian)
5. Scaling up and down
6. One additional filter of your own choosing

The first two filters are trivial pixel operations. The next two filters are convolution operations. Scaling will be implemented by sampling. Your additional filter can use any technique you like. You will also need to write and hand in a README_Filter.txt file documenting your design decisions and any known issues/extra credit you have implemented.

2.1 Invert (warm-up)

This filter is pretty straightforward. All it has to do is invert the color of each pixel in the selected area. You should invert each of the red, green, and blue channels separately. Be sure to preserve the alpha channel (don’t invert it).

2.2 Grayscale (warm-up)

This filter sets the red, green, and blue channels of a pixel to the same value. Add together 29.9% of the red value, 58.7% of the green value, and 11.4% of the blue value of each pixel, and set the final intensity to that sum. Again, preserve (don’t modify) the alpha channel.

2.3 Edge Detect

Your edge detection filter will detect areas of discontinuity in an image. Run Edge Detect after converting your image to grayscale by first invoking your grayscale filter. Grayscale images have the same intensity for red, green and blue.

Edge detection is useful in a number of other filters, since it is a type of frequency representation. When you think about it, it makes sense for the edges to have the highest intensity (energy) values. Therefore, we recommend that sharp edges appear brightest and dull edges/backgrounds appear dimmest.

We have not discussed edge detection in class, but one way to efficiently implement his filter is by convolving your image (in the spatial domain) with the Sobel kernels. The Sobel kernels are approximately equivalent to taking the mathematical derivative of the image signal in the spatial domain. All of this math assumes your colors have been normalized to [0,1] – don’t forget to multiply the end result by 255 to get back to [0, 255]!

\[
G_x = \begin{pmatrix}
-1 & 0 & 1 \\
-2 & 0 & 2 \\
-1 & 0 & 1
\end{pmatrix} \ast A; \\
G_y = \begin{pmatrix}
-1 & -2 & -1 \\
0 & 0 & 0 \\
1 & 2 & 1
\end{pmatrix} \ast A
\]

Here, \( \ast \) represents the 2-dimensional convolution operation. At each point in the image, the resulting gradient approximations can be combined to give the gradient magnitude, using

\[
G = \sqrt{G_x^2 + G_y^2}
\]

The Sobel kernels can be thought of as 3 × 3 approximations to first-derivative-of-Gaussian kernels. That is, it is equivalent to first blurring the image using a 3 × 3 approximation to the Gaussian (a triangle), and then calculating first derivatives.
Mathematically, it’s not exactly the derivative, but it is “close enough for government work!” It works because convolution and derivatives are both commutative and associative (a very useful principle to remember, in general):

\[
\frac{\partial}{\partial x} (I \ast G) = I \ast \frac{\partial}{\partial x} G
\]

A 2D filter is separable if it can be expressed as the outer product of two vectors. Notice that the Sobel kernel above can be written as a matrix product of a column vector \( v = [1, 2, 1]^T \) and a row vector \( h = [-1, 0, 1] \). It turns out that the matrix product of a column vector and a row vector is equivalent to the two-dimensional convolution of the two vectors. In other words, \( v \ast h = S \). So you can filter with \( s \) by filtering first with \( v \), and then filtering the result with \( h \).

\[
G_x = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \ast ([ -1 \quad 0 \quad 1 ]) \ast A
\]

\[
G_y = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} \ast ([ 1 \quad 2 \quad 1 ]) \ast A
\]

This separable convolution may be advantageous because it implies fewer arithmetic computations for each image point. You should also take note of the values in the filter function. Because the values are not in \([0,1]\), you can get values of computations for each image point. You should also take note of the values in the kernel.

You must be extremely careful of boundary conditions when writing this filter. Think about what will happen around the perimeter of your image if you do not compensate for the boundaries. What can you do to alleviate this problem?

Also think about how you can speed the blurring process up. Is your blur kernel separable? There is a way (as it is implemented in the demo) to achieve a box blur in \( O(1) \) time per pixel. If you can achieve this, you will be rewarded with extra credit (we’ll accept the fast box blur in lieu of a triangle, pyramid, or Gaussian kernel).

We mentioned that you may opt to implement a Gaussian kernel instead of a triangle or pyramid kernel. Nice implementations of the Gaussian will receive some extra credit. Note that if you want to do a Gaussian blur, it would be nice if the gaussian kernel was near zero at the edge of your kernel. As a general rule of thumb in graphics, this means you set \( \mu \) to zero and \( \sigma \) to radius divided by three.

### 2.5 Scale

You must allow the user to scale the image up as well as down. This filter is one of those which has an obvious solution, which is wrong: Blindly replicating pixels to fill in gaps when scaling up and removing excess pixels when scaling down doesn’t cut it. You will need to find a good, efficient way of creating a resampled image that looks like it is simply larger or smaller, not like it has been put through a cheese grater.

There are faster and cleaner ways to implement scaling. Note that it may be easier to implement horizontal scaling first and then add vertical scaling later.

For extra credit, you may instead implement some more advanced type of resampling, such as bicubic, which is better for scaling images up. We’ll be very happy if you pull this off, so document any achievements here in your readme as usual.

### 2.6 Special Filter

You are required to implement one additional filter of your own choosing. There are some ideas for possible filters listed at the end of this handout. If you want to implement even more filters, you may do so for extra credit.

### 2.7 Marquee Selection

The support code contains a marquee selection function that allows the user to filter only a part of the image. You need to be able to handle the marquee selection; that is, the user should be able to apply the selected filter to any arbitrary region of the image. Replicate the marquee selection behaviors exhibited in the demo, taking care to note how we handle marquee selections for the scale filter. If there is no
marquee selection, you should filter the entire image. Also note that the marquee selection might, for some reason, extend beyond the bounds of the actual image. If that happens, you should only filter the part of the image that actually exists – don’t try to illegally access memory and then blame it on our buggy marquee selection – you’ve been warned!

3 Support Code

The support code for this assignment consists of the same user interface which you have seen in your other assignment. In this project, you’re going to need to do your own design. In the source tree, create a folder called filter. One possible design is to define a base class from which all your filters inherit (maybe call it Filter), and then to make individual classes for each type of filter you implement. In ui/Canvas2D.cpp, there is a method called filterImage which you will need to fill in. This method will be called when the Filter button is pressed in the GUI. When this button is pressed, you should filter the image. You can get the marquee selection region by calling the marqueeStart and marqueeStop functions, which return QPoint values. The marquee selection is not defined (i.e., no selection) when marqueeStart() == marqueeStop().

4 Assertions

Assertions provide a convenient way to check for logically correct operation at various points within a program. The assert function takes a boolean (or usually an expression that evaluates to a boolean) as an argument. If the boolean is true, the program proceeds normally. Else, if the boolean is false, assert prints out a message stating that the assertion failed and the program is aborted.

Debugging this assignment will be a lot easier if asserts are used to check that the current indices being examined fall within an expected range. The assertions will actually fail and report what was logically incorrect; such information is much more helpful than just having seemingly random skewing and banding in your filter’s output. For more information on assertions, type man assert in a shell.

5 Extra Credit and Half-credit

5.1 [Optional] Extra Credit

Remember that half-credit requirements count as extra credit if you are not enrolled in the half-credit course.

There are millions of different effects that you can perform on raster images. Play around with Photoshop, GIMP, or some other image manipulation package to get some ideas, and try your hand at implementing them. Our GUI allows for two additional “wildcard” filters (and of course can be expanded to allow more); take advantage of that. You can also modify the GUI to add more filters if you would like. :-) The filters in the List of Additional Filters section below lists filters that are appropriate for extra credit. We may give more extra credit to harder filters. We’ll automatically consider the hardest filters you choose to implement as extra credit. The easiest filter you implement will be used to satisfy the required extra filter.

Finally, in the interest of pure, vicious, exhilarating, processor-screaming speed there are a lot of tricks you can do. What can you do to make your filters faster?

5.2 Half-Credit Requirements

Remember that half-credit requirements count as extra credit if you are not enrolled in the half-credit course.

1. For Filter you must implement an additional filter that is listed as medium or higher in section 8.1. Please note that seamcarve will not count towards this requirement. If you are choosing to implement a filter not on the list please email cs123TAs@cs.brown.edu to confirm that your chosen filter is of the correct difficulty.

2. For the second requirement you must implement rotation by arbitrary angle. See the demo for an implementation of this, and section 8.3 below for more information.

6 Final Notes

In an effort to make some of the expectations more clear, here’s some supplementary information. Speed is very important in this program. Less so than a working algorithm (so get them working first), but still important.

• You are required to use the raw pixel memory directly. If you find any Qt or built-in functionality, don’t use it.

• Try to make the instructions in your inner loops as clean and distilled as possible. There are a lot of boundary conditions, just as in Brush, so try to come up with good/fast solutions to this issue. For example, look out for repetitive instructions, and needless multiplications.


• Some of the filters you will be implementing use kernels that are linearly separable. In other words, you can sometimes separate a 2D kernel into two 1D kernels. See the appendix about linearly separable kernels at the end of this handout.

• Using 1D arrays instead of 2D arrays will be faster because the data will be contiguous in memory. If you do this, we also recommend making an inline function that converts 2D coordinates to 1D array coordinates.

• When dealing with large amounts of contiguous data in memory (i.e. a row of an image or region of interest), it can be faster to use `std::memcpy` and `std::memset` to copy data from temporarily allocated memory space or fill memory with a specified value, respectively. For more information, see the man pages for each function.

Filter is a hard assignment. This is where you drive off the cliff. Be sure to start early and code carefully. Time invested coding carefully will pay off dividends later, because debugging off-by-one errors can be extremely time consuming. The TAs will not be able to find your off-by-one errors for you!

7 Handing In

To hand in your assignment, run `/course/cs123/bin/cs123_handin filter` from your main project folder (the folder that contains CS123.pro). Your handin will be confirmed by an e-mail receipt.

8 List of Additional Filters

Here is a list of some filters you might want to implement, listed in approximate order of difficulty (easiest first). If you choose to implement more than one of these filters, we’ll count the easiest one for the required credit and the hardest ones for extra credit.

8.1 List of Possible Special Filters

1. **Sharpen (easy):** Enhance the regions of high contrast in the image. This can be accomplished with a simple convolution, which we leave to you to determine.

2. **Median filter (easy):** Instead of convolving a kernel as usual, take the median over a sample area. Filter the red, green, and blue channels separately. This filter has the effect of removing noise from an image, but also has a blur effect.

3. **Seam carving (medium):** It’s not too bad to make this work slowly for most cases but can be tricky to get it fast and robust. Even the less robust solution is pretty awesome. You can make use of the scale parameters when seam carving is selected. Here’s the original paper by Avidan & Shamir: [http://www.shaiavidan.org/papers/imretFinal.pdf](http://www.shaiavidan.org/papers/imretFinal.pdf)

4. **Auto level (medium):** Modifies an image to automatically stretch the dynamic range between full black and white, and can automatically apply a gamma correction. In other words, images that are too dark will be automatically brightened. Images that are too bright will become darkened.

5. **Bilateral smoothing (hard):** Blur the image sensitively, preserving hard edges. This filter has a really neat, photo-enhancing effect. Adobe Photoshop implements a bilateral filter in its surface blur tool. Read *A Gentle Introduction to Bilateral Filtering and its Applications*\(^1\) by S. Paris, P. Kornprobst, J. Tumblin, and F. Durand, and select one of the many variants. You can hard-code the parameters (usually \(\sigma_s\) and \(\sigma_r\)). The naive implementation of the bilateral filter is \(O(n^2)\) per pixel, which is \(O(n^4)\) overall. We’ll be happy with the basic implementation, although the website referenced has some (not that easy to implement) ideas for doing better...

6. **Bicubic resampling (hard):** A more advanced scaling algorithm that results in smoother images.

7. **Ripple: (hard)** Make part of the image ripple as if it were a pond and a stone was dropped in it. Remember the demo’s special brush in the Brush assignment?

8. **Page curl: (hard)** Make it look like the corner of the image has been curled up.

9. **Frequency domain: (hard)** See below.

10. **Rotation by arbitrary angle: (hard)** See below.

Note: The above are just ideas. All the TAs have not implemented all of them and are therefore not able to guarantee support for your extra efforts (though they will try send you in the right direction). Also, the demo may have some extra credit options that really aren’t “filters” in the sense that they incorporate animation and other temporal effects. But the concepts are the same as any filter, and you would get credit for them, so go nuts. If you want to do an additional filter that is not in the list, get your idea approved by a TA first to ensure you will receive credit for implementing it.

\(^1\)Available here: [http://people.csail.mit.edu/sparis/bf_course/](http://people.csail.mit.edu/sparis/bf_course/)
8.2 Frequency Domain

You can choose to implement either a two-dimensional DFT (simpler to code, but slower) or a two-dimensional FFT (harder to code, but faster). Once you implement your DFT, you should do something in the frequency domain, and then perform the inverse transform to convert back to the spatial domain. Here are some ideas of things you can do.

1. **Smoothing (easy):** Truncate high frequencies, keeping the low frequencies. This is called a *Butterworth Low-Pass Filter*.

2. **Edge detection (easy):** If only keeping the low frequencies discards the edges of the image, then attenuating the low frequencies should only keep the edges. This is exactly what we find when we apply the above highpass filter to the image.

3. **Sharpening (easy)**

Your algorithm must transform your image to the frequency domain, apply the filter, and then transform back to the spatial domain. Instead of talking about one dimensional signals that represent changes in amplitude in time, here we are dealing with two dimensional signals which represent intensity variations in space. These signals come in the form of images. The images we will deal with here are digital, and thus have a finite width and height in pixels, which we will assume have a real number value. Because our signals are discrete, we will need an analog of the one dimensional DFT for two dimensional signals. This analog is the following pair of transforms:

\[
F(u, v) = \frac{1}{MN} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y) e^{-2\pi i \left( \frac{ux}{M} + \frac{vy}{N} \right)}
\]

\[
f(u, v) = \frac{1}{MN} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} F(x, y) e^{-2\pi i \left( \frac{ux}{M} + \frac{vy}{N} \right)}
\]

Recall that \( e^{i\theta} = \cos \theta + i \sin \theta \). Thus an MxN image has an MxN set of (complex) Fourier coefficients. To implement this transform, we would like an analog of the FFT, which will let us quickly compute the coefficients of the transform. In fact, we can do better. The two dimensional DFT is separable into two one dimensional DFTs which can be implemented with an FFT algorithm.

\[
F(u, v) = \frac{1}{N} \sum_{x=0}^{M-1} e^{-j2\pi ux/M} \sum_{y=0}^{N-1} f(x, y) e^{-j2\pi vy/N}
\]

You may look online for more information about the Fourier transform, although your implementation must be your own work.

8.3 Rotation

There is also a “Rotation” option built into the GUI. Rotation is a really cool filter to implement, but for the sake of time we are not requiring it. Nonetheless, we encourage you to give it a try:

This filter is another that can be done “wrong”, but in this case it is painfully obvious that it is (you will get ugly, aliased results; unparalleled even by a “bad” scaling routine). However, being good CS123 students, the wrong way should never even enter your mind. The right way also has a few pitfalls. Depending on how you tackle the problem, you may come out with excellent rotation, but it will take a long time (and that is no good). There is a really fast way for performing good rotation, and it just takes a little linear algebra (or a little research through graphics texts) to figure it out. Do your best to come up with a good way of rotating an image.