Homework 6: Image Processing

Introduction to Computer Graphics and Imaging (Summer 2012), Stanford University
Due **Tuesday**, August 14, 11:59pm

In this homework, you will explore the world of image processing by implementing some basic photo filters. Unlike our previous homeworks, we'll interleave the programming and written parts as you design and implement filters. Before beginning, carefully review the slides from CS 148 lecture 15, which covers the techniques we'll be using for image processing.

Before we process images, we'd like to be able to process one-dimensional arrays. In lecture we discussed an incremental algorithm for convolving a 1D array with the averaging "box filter" that runs in $O(n+k)$ time for a box of width k and a signal of width n.

In our skeleton code, we have a function entitled `iteratedBoxFilter`. This function takes an input 1D array and repeatedly applies the box filter as many times as the user desires. The number of iterations is given by `filterIterations`. The width of your box comes from `filterSize`: Your averages should be taken over a range of width `2*filterSize+1`. That is, when `filterSize` equals 0, you should return the original signal, when it equals 1 you should return a signal where entry $o(i)$ in the output equals the average $\frac{1}{3}(f(i-1)+f(i)+f(i+1))$ for input array f, and so on. You may use whichever boundary conditions you please for averaging beyond the beginning and end of the array.

We're going to use a tricky templating scheme that will simplify later parts of this homework. Notice that `iteratedBoxFilter` is templated over `ArrayType` and `ElementType`; the function `process1dSignal` contains an example of how to call it. You can assume elements of type `ArrayType` have an operator `[]` returning elements of `ElementType` and a method `.size()` containing the width of the array.

Problem 1 (30 points)**.** *Implement the incremental approach to 1D box filter convolution in `iteratedBoxFilter`, and surround it with a loop allowing for iterated box filter application. You may* not *copy over the entire input array into any sort of temporary structure, although you may use space proportional to `filterSize`.*

Notice that you can call our program on 1D signals contained in text files by using command line arguments (call `./hw6` with no arguments to see documentation). An example 1D input is given in `data/impulse.txt`. This file contains all 0's except for a single 1 in the middle; this is the structure of the "impulse function" used to analyze many signal processing methods.

Problem 2 (20 points)**.** *Plot the 1D impulse function with 0, 1, 2, and 3 iterations of the box filter applied. What shape are these functions approaching? Also, plot a "step function"–that is, a function that is 0 for half the entries and then 1 for the rest–with 0, 1, 2, and 3 iterations of box convolution applied. What happens to the smoothness of the step function with each iteration of box convolution?*

Remember from lecture that Gaussian blurs are *separable*. This means that you can take a 2D image, Gaussian-blur its rows as 1D signals and then blur its columns, and the output is the same as having applied a two-dimensional Gaussian to the input.

Figure 1: For problem 4.

Your next job will be to apply this technique to blurring images rather than arrays in the `process2dSignal` function. To make your job easier (this is not the most efficient technique!), we have provided methods that return wrapper classes making individual rows and columns of an image look like 1D arrays. You can get such 1D structures by making calls like the following:

```
ImageRow &signalRow = signal.getRow(i);
ImageCol outSignalCol = outSignal.getCol(j);
```

Notice a slight annoyance with our code: **To function properly you must include the `&` before declaring an `ImageRow`.**

Problem 3 (20 points)**.** *Implement `process2dSignal` to apply our iterated box filter to the rows and then columns of an image. By using calls to `getRow`, `getCol`, and `iteratedBoxFilter`, you should be able to make this function fairly short.*

One interesting effect we can achieve using our 2D blurs is the so-called "tilt-shift" effect, which uses nonuniform blurring to modify the depth-of-field of a photo in a way that makes the subjects look small; an example of the output of the model solution is shown in Figure 1. Read the Wikipedia page to learn about the camera optics that can achieve tilt-shift in real life. For our purposes, however, we notice that an acceptable approximation of tilt-shift is achieved simply by modifying the width of our blur depending on position in the image.

Problem 4 (20 points)**.** *Formulate and implement a tilt-shift style filter in `tiltShift`. You are free to model this effect any way you want; describe your approach in the written component. Do not worry about efficiency, so long as your tilt-shift takes no more than a minute or two to compute.*

Hint: *There are many possible ways to approach this problem. One inefficient but effective strategy is to use `process2dSignal` to blur the image using many different filter sizes and store the result of each of these blurs in a separate image. Then, for each pixel in the output image, select out or interpolate between the different blurred images to choose a blur size as a function of y.*

Finally, we'll let you implement your own image filter:

Problem 5 (10 points plus up to 10 points discretionary extra credit)**.** *Choose an artistic or practical image filter and add functionality to our program to evaluate it. At the least, your filter should involve more than a single per-pixel color space operation. Potential ideas include:*

- *Local or global contrast enhancement*
- *Histogram equalization*
- *Rotation with bilinear interpolation (cutting off the corners is fine)*
- *Edge detection*
- *Minification or magnification*

Feel free to search the internet for potential ideas, although the implementation should be your own. In your writeup, include some images illustrating what your filter can do.