CS 465 Program 2: Resample

(revised September 15, 2005)

out: 15 September 2005
due: 27 September 2005

1 Introduction

In this assignment you will implement general filtering and resampling for images. We provide a framework that provides image I/O and a GUI. You will provide the implementation of a general filtering and resampling algorithm and the implementations of several useful filters.

2 Assignment Overview

The main requirement of this assignment boils down to writing a general method of evaluating the convolution of a continuous 2D filtering function with an image, a discretely sampled 2D function. You will then implement a variety of filter functions and will investigate the result of using different functions for filtering and resampling operations. After implementing the brute force version of this algorithm, you will take advantage of the separability of the filter functions to improve the performance of the filtering application. You will see a drastic improvement for large filters.

We stress that that this assignment will be conceptually difficult rather than being difficult to program. Our final versions of the general filtering/resampling methods are collectively less than 150 lines of Java. Whether your implementation is longer or shorter is not part of the assignment, but be aware that working without a careful design and plan can make the problem much more difficult than necessary. We strongly suggest that you work a couple of pencil and paper examples of resampling and filtering (Hint: Think Homework 2) and think carefully before starting to code. If you plan it right you will find that the filtering and resampling operations are almost identical and much of your code can used in both cases.

3 Requirements

1. You must implement 4 methods in FilterOperations.java: `filter()`, `fastFilter()`, `resample()` and `fastResample()`. The filter functions will take an input image, an output image, a filtering function, and some numbers specifying the filter size. The input and output images will be the same size and the methods should set the output image to the convolution of the input image and the filter. The resampling functions will take only an input image, an output image and a filtering function. You must determine the correct filter sizes

from the input and output image sizes and evaluate the convolution of the resampling filter as the output image. In general the input and output images will *not* be the same size and you must correctly calculate locations of the output pixels in the input image. The "fast" versions of each function will implement seperable versions of the algorithms while the regular versions will use brute force methods.

2. You must implement 7 different filter functions: Box, Tent, BSpline, Catmull-Rom Cubic, Mitchell-Netravali Cubic, Gaussian, and Unsharp Mask. Chapter 4 of Shirley describes in detail all of the filters that you will implement for this assignment. The class files for these filtering functions are already defined. For each you will need to implement the `getSupport()` method and the `evaluate(double x)` method. The first method returns the size of the filter's canonical support and the second evaluates the value of the filter at a given location.

4 Framework Overview

In this section, we provide an overview of the code provided and in the last section we give some insight for the overall layout, give some tips, and answer some common questions.

4.1 The GUI

Out of the box, running MainFrame.java opens the assignment GUI application. You can load images and save results using the File menu. The application has two modes and by default the application is in Filter mode. In this mode, you can select a filter type and set its parameters, if it has them (If the function has parameters a separate sub-window will appear automatically when the function is selected. You can enter the parameters in this window in text fields next to an Update button. **The fields in the cooresponding filter object will be set after (and only after) the Update button is clicked.**). There are also fields to set the filter width and height scale. The Filter button will apply the filter and display the result. The Apply button will make the current window contents the source image. Any future filtering and resampling operations will then be performed on that image.

Clicking the Auto-resample Mode check box puts the GUI in Resample mode. The original image will be resampled to fill the current window and will be again resampled from the source every time the window size is changed. The filtering buttons and the width and height input buttons will disappear in this mode. Instead you can check a box to hold the aspect ratio constant during resizing.

Finally clicking the Fast Filtering Mode box will cause the GUI to call the "fast" seperable versions of the filtering and resampling methods.

4.2 `MainFrame` and `ScrollingPanel`

`MainFrame` is the root class of the application. It defines and controls the GUI interface. `ScrollingPanel` defines some of the properties of the image viewing window. There should be no reason to change either of these classes.

4.3 `Color`, `Image` and `ResizableImage`

The `Color` and `Image` classes are the same as in the ray1 assignment. `ResizableImage` is a sub-class of `Image` that supports a resize operation that adaptively allocates new memory during resizing. The adaptive resizing make the auto-resampling mode possible. Otherwise the sheet allocation of the new image would take longer than the filtering in many cases. This method is preferable since once the maximum size image is created no new data is every allocated. Your seperable filtering will also require an intermediate image of arbitrary size. We have already placed one in the `FilterOperations` class for you to use. It is important that you don't allocate other images. If you do, your filtering operations will become very long because they will spend most of the time on image allocation. Be aware that while the method will resize the image, it will not clear previously stored data so you must explicitly clear or load image data after a `resize()` operation.

4.4 `FilterFunction`

`FilterFunction` is the base class of all the filter functions used in this assignment. It has two important methods, `getSupport()` and `evaluate(double x)` that return the filter support size and evaluate the filter function respectively. You will have to override these methods for each of the filter functions that you will write. The general filtering and resampling methods should depend only on the contents of the `FilterFunction` class and any details specific to a filter should be encapsulated within the specific sub-class. `FilterFunction` has additional methods specifically related to creating and updating the options panels for certain filter functions. You should not need to use or alter these methods.

4.5 `BoxFilter`, `TentFilter`, `BSplineFilter`, etc.

These are the subclasses of `FilterFunction` that you need to implement. For each you will only need to override the `getSupport()` and `evaluate(double x)` methods. Chapter 4 of Shirley describes in detail all of the filters that you will implement for this assignment.

4.6 `FilterOperations` and `DefaultFilterOperations`

The `FilterOperations` class contains the stubs of filtering and resampling functions that you must implement. `DefaultFilterOperations` has the same methods but implements point sampling instead of correct filtering and resampling. So that the GUI will do something out of the box, the methods in `FilterOperations` call the methods in `DefaultFilterOperations` by default. Note that point sampling amounts to the identity operation if the source and destination images are the same size so nothing will happen if you click Filter at first. Point sampling is the same as pixel dropping or pixel replication in the downsampling and upsampling cases respectively so initially resampling will work with ugly results.

5 Implementation Details

5.1 Normalization, or, What to do at the edge?

In class, several methods were described to handle the cases where the filter would fall off the edge of image, such as wrapping around to the other side, reflecting across the border, and using the closest value. For this project we want you to renormalize the convolution operation. You will divide the the final convolution value at a point by the sum of the filter weights used for samples under the filter support at that point.

For example in one dimension, the resulting equation for a convolution is:

$$I'[j] = \frac{\sum_i I[i] f(j-i)}{\sum_i f(j-i)}$$

where $I'[j]$ is one output sample at j, I is the original set of samples, f is the filtering function and r is the radius of the filter.

5.2 Separability or Why are all the filter functions 1D?

It should be clear that to filter images you will need 2D filter functions and all of the filtering function code seems to imply only 1D functions. As noted in class, this is because all the filter functions are seperable. For a seperable function $h_2(x, y)$ in 2D:

$$h_2(x, y) = f_1(x) g_1(y)$$

where f and g are 1D functions that describe the filter's behavior separately in each dimension. For this assignment, each of the `FilterFunctions` will be used to construct a separable 2D filter by using the function for both f and g in the above equation.

5.3 Coordinate systems, or, How to I make sense of all this sample stuff?

Perhaps the hardest part of this assignment is defining a consistent coordinate system. This will be vital to correctly finding the samples that fall under a particular filter's support and for calculating correct filter values. We suggest the following: always work in the coordinate system of the source image. A good layout for doing this is to imagine that the source image is a grid with samples taken at the centers of each grid cell. Align the lower left corner of the grid with the point $(-0.5, -0.5)$. If you assume the pixel spacing of the source image is the same as the spacing on the axes, the lowest leftmost sample should fall on the point $(0, 0)$ and all of the other samples should lie on integer coordinates (see Shirley Figure 3.1).

Using this grid concept you must work out how to calculate two different pieces of information. First, determine a method to find the source image samples that lie within a rectangle centered an arbitrary point in the source image. You will not want to assume that this point will have integer coordinates. Second, work out a way of converting points in the output image to points in the input image. A good visualization to use here is to imagine the output image as another grid that has exactly the same boundaries as the input grid but with more or fewer pixels in each direction. Of course the output grid cells will be different sizes than the input grid and they will most likely not

even be square. However, you should be able to use the ratios of the input and output image sizes to determine how big the cells are relative to the input spacing, and from there you should be able to determine the input coordinates for an arbitrary output image point.

5.4 Canonical Sizes, or, How do I deal with different filter sizes?

Another difficulty with the assignment is finding a way to generalize the filtering operations even though the filters have different regions of support. We have adopted the following nomenclature. `FilterFunctions` have a canonical radius. The canonical radius tells you the size of the filter only *relative to other filters*. Thus a box filter has a canonical radius of 0.5 while a Catmull-Rom Cubic Filter has a canonical radius of 2. However, the canonical radius *does not* tell you how big the filter is relative to the image sample spacing. In general you will want to scale the filter up or down (or squish or squash it if the scales are different in the x and y directions). This is the purpose of the `xSupportScale` and `ySupportScale` parameters to the `filter()` function. The "real" radius of the filter is the canonical radius times the support scale in that direction. The `getSupport()` method should return the canonical radius of the filter and the `evaluate(x)` method should assume its input values are relative to the canonical radius. In practice, this last requirement means you will have to scale the real image distance between the filter point and the sample point down by the inverse of the support scale before calling evaluate.

5.5 Sharpening filters, or, Why do Unsharp Masks sharpen images?

One common image processing operation is to sharpen the image. This is often done by subtracting a blurred version of the image from the original. Because the filter contains a blurred image version, filters that do this are generally called Unsharp Masks. Usually Unsharp Masks are discrete rather than continuous filters, but it is convenient for this assignment to use only continuous filter functions. We will define the Unsharp Mask filter as a small tent filter minus a Gaussian filter and you can construct the Unsharp Mask using these two filters and two other pieces of information. First, the tent filter should be able to be squished to a narrow spike and the mask contains a parameter `tentWidth` that defines how wide the tent should be (in canonical units, this might be then scaled by the filter scales). Second, the parameter `alpha` weights how much each filter contributes so if $t(x)$ is the tent filter and $g(x)$ is the Gaussian:

$$\text{unsharp}(x) = (1 + \alpha)t(x) - \alpha g(x)$$

5.6 Rounding, Integer math, and Java, or, Why does Java hate me?

At many points in this assignment, you will have to compute floating point values from integers and vice versa. Java has some quirky aspects that can make it easy to make simple mistakes. It is good to remember that:

- Java uses integer division for integer operands. So, 3 / 7 is 0 to Java but 3.0 / 7 or 3 / 7.0 is 0.4285714.
- The precedence of a cast is higher than the numerical operations. So (int) 1.51 * 2 = 2 because the cast occurs before the multiplication. It is good to always get in the habit of explicitly using parenthesis to define the range of the cast, (int) (1.51 * 2) = 3.

- Java uses truncation to cast values to int. So 4.9999999 and 4.0000001 both cast to the integer 4. It is very common for numerical accuracy to affect the cast value one way or another.

5.7 Final notes

As part of the assignment, you must to determine the correct scaling factor to use when resampling an image. As a rule of thumb, one canonical filter unit should be as large as one pixel spacing in the input image or the output image whatever is larger. This means that in the enlarging or upsampling case where output pixels are smaller than input pixels, one canonical unit will equal one input pixel spacing and in the reducing or downsampling case where output pixels are larger than input pixels, one canonical unit will be as large as one output pixel spacing. Also even though we think of the filter size both in terms of input and output pixel spacing, it will be convenient to consistently measure only according to the input spacing. So in this latter case, you would want to convert the one output pixel space into input pixel units.

Finally, it is entirely possible for the convolution value at a point to be greater than 1 or less than 0. In order to avoid strange artifacts, you will have to clamp the output pixel values to this range just before finally storing them in the image.

6 Submission and FAQ

Submission will be through CMS. A FAQ page will be kept on the course web site detailing any new questions and their answers brought to the attention of the course staff.