Simple Histogram-Based Thresholding
Computer Vision: CS4984 Assignment 1

Alpha Chen

February 22, 2006

1 Assignment

1.1 Task

One of the most basic processing operations for computer vision is to separate an object from the image background. A common approach for an image with just object and background is to assume that there is a clear bi-modal distribution of the intensity values of the image, and to threshold the image (for image value Threshold T, all pixels values $\geq T$ belong to one class, and all pixels $< T$ belong to the other). Write a program to read an image, extract its histogram, manually pick a threshold from the histogram, and threshold the image. Your program will produce a bi-valued (white & black/grey) image that you can print.

Since this assignment is also intended to introduce students to working with images, you are to make two versions of this program: One in Matlab, and the other in a programming language of your choice.

1.2 Deliverables

You are to write and test these programs on images that we will provide on the class web page. You are to turn in your program code and a 5 to 10-page report on what you learned in the project (e.g. programming issues, ease/difficulty in finding the thresholds, how you might automate the process of finding thresholds). You report should include thresholding result images.

2 Implementation

2.1 Core Image

2.1.1 Why?

The first language I chose to write this assignment was Objective-C. The reason for this choice is because I develop on a PowerBook running Tiger, which includes the Core Image technology. Core Image allows the machine's GPU to work on image-related effects, and looks to be a very promising for graphics work in the future. Thus, with a simple task, I chose to use this opportunity to learn more about desktop applications and how to program closer to the system level.

2.1.2 How?

The resulting program heavily uses OS X's included frameworks and APIs. The program itself runs under Cocoa, using the Application Kit framework to build a document-based application. This part of the program utilizes the Model-View-Controller pattern, as many Cocoa apps do. The Model contains the image, performs the filter, and provides the end result.

Core Image comes into play through the Model. The image is stored as a bitmap reference, which is then converted to an object which can be filtered by Core Image. A Image Unit was written in the CIKernel language to perform the filtering. This Image Unit is incorporated into the program through glue code, which "defines the interface, initializes the kernel, and provides and output image from the filter."[1]

2.1.3 Problems

The bulk of the time spent on this portion of the assignment was learning Cocoa. Specifically, the Application Framework took a long time to understand fully enough to use it in the program. Despite the Core Image code being in CIKernel (derived from OpenGL Shading Language) and it being the main logic for the assignment, it was the easiest part of the program.

A small stumbling block involved converting the image from an NSImage (for ease in displaying) to a CIImage (to be used by Core Image). After some research, a clean solution to this problem was found online.

Overall, this program was fairly straightforward, with the frameworks being the most complicated part to understand and code.

2.1.4 Sample Code

```
kernel vec4 thresholdEffect(sampler image, float threshold)
{
        vec4 s;
        vec3 rgb;
        float c;
        s = sample(image, samplerCoord(image));
        rgb = s.rgb;
        c = (rgb[0] + rgb[1] + rgb[2]) / 3.0;
        s.rgb = (c >= (1.0-threshold))
          ? vec3(1.0, 1.0, 1.0)
          : vec3(c, c, c);
        return s;
}
```

[1] From Apple's *Developing With Core Image* (http://developer.apple.com/macosx/coreimage.html)

2.2 RMagick

2.2.1 Why?

ImageMagick is a popular free software suite which can manipulate images. There are interfaces available for many languages, including my language of choice, Ruby. The Ruby interface is called RMagick, and like the interfaces for the other languages, makes it possible to use ImageMagick functions straight from the libraries instead of calling ImageMagick's various command line utilities.

2.2.2 How?

This program was almost trivial to write. Reading the RMagick documentation revealed `quantize`, `color_histogram`, and `white_threshold` methods which were then used to process the image. The image is first quantized to produce a histogram with a certain number of bins, and then displayed so the user can choose a threshold value. After a value is chosen, the image is thresholded and then written to a file.

2.2.3 Example Code

```
image = Magick::ImageList.new(options[:input])

unless options[:threshold]
  puts ''Generating histogram...''

  histogram = image.quantize(HISTOGRAM_BINS, Magick::GRAYColorspace).color_histog
  pixels = histogram.keys.sort_by {|pixel| histogram[pixel]}
  max_color = histogram.values.max
  histogram.each do |k,v|
    histogram[k] = v.to_f / max_color
  end

  pixels.each_with_index do |pixel,i|
    print "%02d: " % [i]
    print "(%0.2f) " % [histogram[pixel]]
    stars = (HISTOGRAM_BINS * histogram[pixel]).to_i
    stars.times { print '*' }
    puts
  end

  options[:threshold] = ask("Choose a threshold value from 0.0 to 1.0: ", Float)
end

image = image.white_threshold(Magick::MaxRGB * options[:threshold])
image.image_type = Magick::GrayscaleType
image.write(options[:output])
```

2.3 Results

Note that the Ruby script uses the RGB-centric 0.0 to represent black and 1.0 to represent white, while the Core Image application reverses this idiom for easier human comprehension.

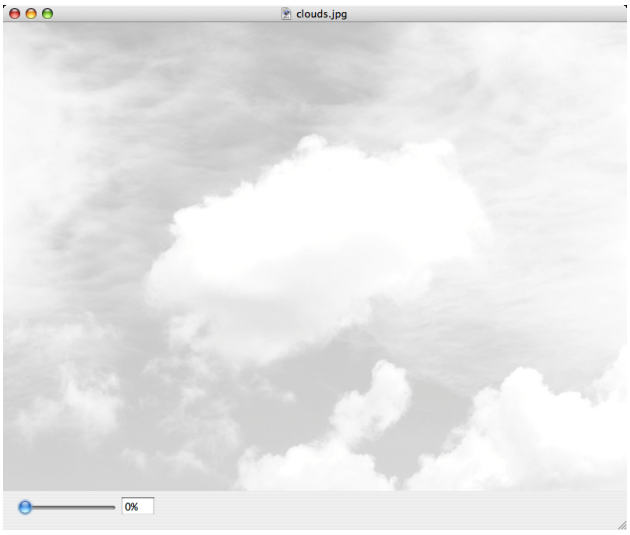

Figure 1: Core Image: loaded image

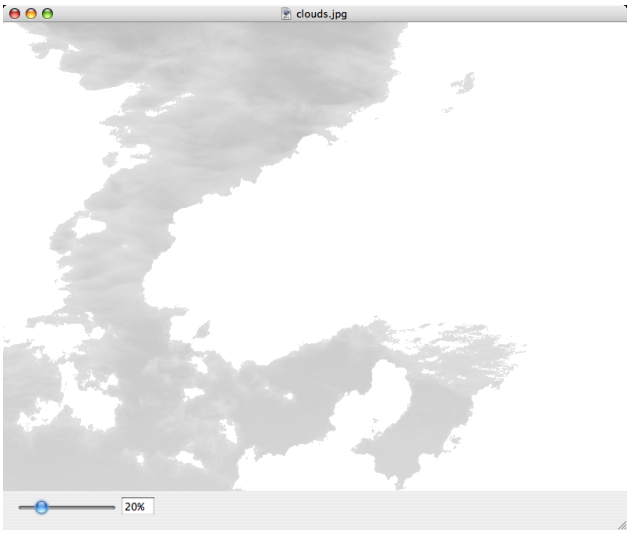

Figure 2: Core Image: filtered image

Figure 5 was created by using `./threshold.rb -t0.5 ../images/shadow.jpg report/shadow.jpg`. By running `./threshold.rb ../images/shadow.jpg report/shadow.jpg`, the interface below is used to choose a threshold:

Figure 3: Core Image: clouds.jpg with a 0.2 threshold

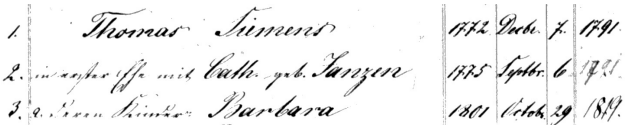

Figure 4: Core Image: handwriting.gif with a 0.37 threshold

```
Generating histogram...
00: (0.01)
01: (0.16) *
02: (0.22) **
03: (0.23) **
04: (0.28) **
05: (0.38) ***
06: (0.42) ****
07: (0.43) ****
08: (0.67) ******
09: (1.00) **********
Choose a threshold value from 0.0 to 1.0: 0.5
```

2.4 Conclusion

Through this assignment, I learned how to use a wide range of technologies to implement a simple thresholding algorithm. Although the thresholding itself was trivial to code, the surrounding application took far more time to develop. I did not implement histogramming for the Core Image application, as I did not have enough time, but it was implemented in the Ruby application.

I found that it was harder to find thresholds with a histogram, mostly because there was not enough detail in my default histogram level to find good thresholds. It was easier to use my OS X application, as I could visually see what threshold values corresponded to what pictures.

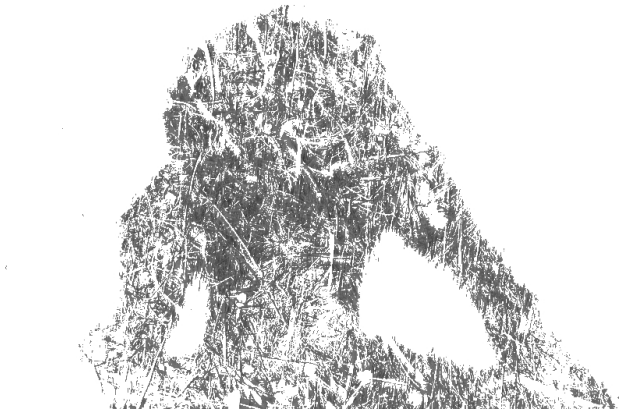

Figure 5: RMagick: shadow.jpg with a 0.5 threshold

Figure 6: RMagick: trees.jpg with a 0.2 threshold

There are several ways that the threshold choice could be automated, but the first one that comes to mind is to use the histogram data. Looking for large differences (slopes) in grayscale densities and local minimums could yield potential threshold points.