12-1-2013

Object Detection Using Contrast Enhancement and Dynamic Noise Reduction

Justin Lee Baker
University of Nevada, Las Vegas, bakerj32@unlv.nevada.edu

OBJECT DETECTION USING CONTRAST ENHANCEMENT AND DYNAMIC NOISE REDUCTION

by

Justin L. Baker

Bachelor of Science in Computer Science (B.Sc.)
University of Nevada, Las Vegas
2010

A thesis submitted in partial fulfillment of
the requirements for the

Master of Science – Computer Science

Department of Computer Science
Howard R. Hughes College of Engineering
The Graduate College

University of Nevada, Las Vegas
December 2013

UNLV
UNIVERSITY OF NEVADA LAS VEGAS

THE GRADUATE COLLEGE

We recommend the thesis prepared under our supervision by

Justin L. Baker

entitled

Object Detection Using Contrast Enhancement and Dynamic Noise Reduction

is approved in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science
Department of Computer Science

Evangelos A. Yfantis, Ph.D., Committee Chair

Jan B. Pedersen, Ph.D., Committee Member

John T. Minor, Ph.D., Committee Member

Alexander Paz, Ph.D., Graduate College Representative

Kathryn Hausbeck Korgan, Ph.D., Interim Dean of the Graduate College

December 2013

Abstract

Edge detection is one of the most important steps a computer must perform to gain understanding of an object in a digital image either from disk or from video feed. Edge detection allows for the computer to describe the shape of the objects in an image and create a pixel boundary defining what is considered part of an object, and what is not. Cannys edge detection algorithm is one of the most robust and accurate of these edge detection algorithms. However, as with many algorithms in image processing, there are many cases where the algorithm does not perform as well as an application requires. This can be caused by many problems, many of which are beyond the control of the image analyst because the images were supplied with poor lighting, or from security cameras, or low contrast situations. Even steps like converting the image to grayscale can interfere with detection. In this thesis we will explore improvements to the algorithm by a dynamic system that will select a color channel to help deal with data loss issues and improve the contrast between the object and its background using partial histograms. Then we will use histogram equalization to greatly improve the contrast of the image and explore a progressive implementation of histogram equalization to reduce the noise and get good detection of the objects that an unmodified edge detector would have struggled with.

Acknowledgements

I would like to thank my advisor, Dr. Evangelos Yfantis, for helping me many times along the way. My time at UNLV would have been very different had I not met him and studied under his advice. I appreciate the patience he has shown as I worked through this thesis. I would also like to thank my graduate committee, Dr. Jan Pedersen, Dr. John Minor and Dr. Alexander Paz for finding time in their schedules and support.

I would also like to thank my mother and father as well as my older brothers and younger sister. Without their support and advice, I do not know if I could have completed my Master's degree here at UNLV.

Justin L. Baker

University of Nevada, Las Vegas
December 2013

Table of Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

Computer image processing is a field of study that can solve a large domain of problems. There are many tasks that we perform or hire people to perform by hand often because we are unaware of the technology and the algorithms that exist which could make the tasks easier, or even automate the task entirely. Many of the tasks we have people do by hand are tasks that involve very noisy and difficult to parse data, such as identifying objects in a series of photographs. Often, these photographs must have an algorithmic task performed on them, such as identify and catalog all objects that appear in the photographs, or detect if there is an object of interest in the photograph at all. Another common problem relates to security camera footage. Rather than run through an entire video by hand to find significant events, it would be more useful if the system was able to simply know certain timestamps of interest and even give the information needed to identify the object that caused the event.

Because there is such a wide range of problems that can be solved with image processing, it is difficult to find an algorithm that solves all of the problems you could ever be presented with very well. These images could come from a variety of sources, from different cameras with different lenses or resolutions. Additionally the lighting features of these images could be vastly different. They could be taken by hand by a professional who understands lighting and tries to take the photograph so it is easy to see the features of the object of the photograph. The images could also come from a source like a security camera, or a satellite, which will take the pictures with no regard to providing special lighting conditions. They could also be taken during a dark night, or from a camera that is pointed towards very bright light sources. These types of scenarios require modifications to the standard detector that would often be found in common libraries [XJL06], or at least additional preprocessing of the image [NPKJ08]. Therefore, in order to be skilled with image processing, a good understanding of many image processing techniques is necessary. Very efficient

image processing applications often use techniques that are catered to the specific problem they are trying to solve.

In this thesis, we are going to explore how to detect objects in photographs and digital imagery. We want our algorithm to work on as many types of objects and with as many types of cameras as possible. In order to do this we will need to use advanced edge detection algorithms like Canny's edge detector. Unfortunately, there are situations that Canny's detector alone cannot solve perfectly. So, in addition to our own implementation of Canny edge detector, we will explore ways to process data for the purposes of maximizing contrast while still trying to minimize the number of false edges. If there are false edges introduced, we hope that they are small, so later algorithms can easily identify and remove these false edges. We will look at techniques like color channel selection and histogram equalization. There are downsides to both and we will show that some of these limitations can be resolved. In this thesis we are going to explore modifications to the Canny edge detection algorithm and show how we are able to use it to detect objects. On the other hand, there are major benefits to the systems we suggest, which is why we recommend considering these approaches when trying to detect objects and their edges. Another technique we are going to explore in this thesis is using histogram equalization to correct for poor or extremely strong lighting conditions, often improving situations where increasing the contrast by color channel selection isn't as easily possible. The next chapter will cover the background of the field of image processing, and the techniques we are going to use. The implementation chapter will cover the specific decisions made in this implementation of Canny's edge detector as well as the implementation of the color channel selector and the histogram equalization algorithm. We will also describe our progressive version of histogram equalization, which is a small modification to the existing algorithm but can give better results from better distribution. Finally, in chapter four, we will look at the results of our implementation's decisions and compare them to traditional approaches or other decisions. For example, we will compare the decision to use the luma of the image, as is usually traditional, against other color channels to show that selecting a specific color channel that will maximize contrast enhances the strength of the gradients, allowing Canny's edge detector to more easily select important edges. We will also compare traditional histogram equalization with the progressive histogram equalization algorithm we implemented and compare the results.

Chapter 2

Background

2.1 Additive Color Scheme and RGB Encoding

Before work can be done on analyzing photographs with an edge detector or other image processing program, it is important to understand how images are stored and displayed digitally. Computer monitors and other displays that attach to computers generally consist of a large array of pixels. Pixels are small colored lighting elements that are responsible for lighting themselves to a specific color so they can represent a small part of the image they are responsible for displaying. In order to do this effectively, the pixels obey the additive color model and are actually comprised of three sub-pixels for each of the three different color channels in the additive color model: red, green and blue. When a sub-pixel is completely unpowered, we say the intensity of that pixel is low, and so it displays as black. As the intensity of a specific sub-pixel increases, the color that the sub-pixel is responsible for increases. So if the sub-pixel is for the red component, low intensities will result in dark reds and high intensities will result in a bright red. If the blue sub-pixel gains intensity, the pixel goes from black to dark blue and eventually a bright blue. If red and blue are both at their most intense value for a given pixel, then the visual effect of adding a bright red and a bright blue will be the result, which will yield a magenta color. Color is a human perceptual to seeing various wavelengths in the color light spectrum [PV00]. The additive color model is one of many different ways to represent and organize the concept of color [BB82] but there are other models that could be used that could create the same phenomena. The additive color model, for example, contrasts with subtractive color models that default low intensity values for sub-pixels as white and the colors cyan, yellow and magenta subtract from the white to create black when cyan, yellow and magenta are at maximal intensity. The cyan, yellow and magenta color model is used by most printers, for example, compared to monitors and televisions that use the additive color model [Pit00]. The

$$\begin{bmatrix} (R_{0,0}, G_{0,0}, B_{0,0}) & \cdots & (R_{N-1,0}, G_{N-1,0}, B_{N-1,0}) \\ \vdots & \ddots & \vdots \\ (R_{0,M-1}, G_{0,M-1}, B_{0,M-1}) & \cdots & (R_{N-1,M-1}, G_{N-1,M-1}, B_{N-1,M-1}) \end{bmatrix}$$

Figure 2.1: The basic representation of a digital image.

medium printers display on is usually white paper, so to elicit the same response from human eyes, the printer needs to subtract colors from the white, while monitors fill a dark screen with light of varying colors.

To describe a picture such as a photograph in a digital form, it is easiest to use the same conventions that will be used to display images in monitors, using the additive color model. Each pixel of a digital image will be represented by three bytes. This means that there is a single byte for each of the red, green and blue components of a pixel. This byte, with high values will indicate a high intensity and with low values will indicate a lower intensity. It is actually possible for equipment and digital image formats to represent pixels of the image with more than a three bytes, which means a larger range of intensity values [Pit00], but most digital images, including the ones examined in this thesis, will be using pixels that are defined by three bytes. If a given sub-pixel is set to 0, there is no energy going to the sub-pixel, and the sub-pixel will display as black. If the sub-pixel is 255 then the sub-pixel is at its maximum intensity and will display whatever color it is with full intensity. An individual pixel is therefore represented by three bytes in an ordered tuple (R,G,B) or RGB for short. The values for R, G and B will indicate the intensity for the red, green and blue, respectively.

Often it is also useful to think of an image as a spacial two-dimensional function and the amplitude of that function is the intensity of the image at specific pixels [GW08]. Therefore images are discrete approximations of that function. There are many ways to store a digital image but for this paper we will assume images are stored in a format similar to the bitmap, where each pixel is stored directly as three bytes indicating the intensity for each of the three additive color channels. They, of course, may or may not actually be stored exactly in order of RGB but it is simple to represent the data in this format if provided in a different ordering. All that is important is that each pixel is represented in an RGB encoding. Generally, an image that is NxM in size can be represented as a matrix as shown in Figure 2.1 [Pit00]. Notice that the coordinate system used in the representation of an image is slightly different from the traditional Cartesian coordinate system in that $(0, 0)$ is located at the top left of an image, not the center, and values along the y axis increase positively for an image as you move down the image, rather than progress through negative numbers. For most algorithms, this won't change anything much, but it is important when discussing images, since $(0, 0)$ is not the center of the image.

2.2 Grayscale Image Conversion, Luma

Using the RGB encoding and additive coloring scheme, we can represent $256^3 = 16,777,216$ different colors. Of these colors there are exactly 256 grayscale colors. Grayscale colors are created when the red, green and blue components of the pixel are exactly the same value. For example, the color black is indicated by $(0, 0, 0)$ in the RGB color model and white is $(255, 255, 255)$. For computer image processing, we often want to run algorithms against a single channel of the image. Using the grayscale saves on computation time and simplifies the code of the algorithm, and can even ensure bad side-effects are avoided for some algorithms. The issue comes to conversion of a color image to a grayscale image. Transitioning from 16,777,216 colors to only 256 will cause a large amount of information loss, yet there are many techniques that allow one to convert an image to grayscale and experience minimal loss for many common tasks that image processing programs must accomplish, specifically edge detectors.

Figure 2.2: The fruits image and the color channels of the image. (Top: Color image and red channel. Bottom: Green and blue channel.)

There are many ways to create a grayscale image from a color image. One technique would be to simply use a single color channel, such as red, as the intensity value of the monochromatic image. Since, as was already discussed, the red is one of the primary colors in the RGB encoding scheme, the red values of a pixel are represented by a single 0 through 255 value. If we created an image using only the red components of the RGB encoding, the resulting image would be a grayscale image

where each pixel uses the magnitude of the red channel of the original as the RGB values in the resulting image. Objects that have more red in them will have higher intensity, in general, and objects that lack red will have lower intensities. The same technique can be applied to the other color channels as well.

Figure 2.2 shows a color image and grayscale image representations of each of the color channels of the image. The first image of the figure shows the original color image, and the three grayscale images are the red, green and blue channels. The tomatoes at the bottom of the figure, which appear red in the color image contain large magnitudes of red. When we look at the red channel, those tomatoes are generally a bright color, indicating that the values of the red channel are high at those points. In contrast, the tomatoes appear very dark in the green and blue channels, suggesting that there is very little blue or green used to create the color of the tomatoes. The grapes, while are slightly brighter in the green channel, actually have a lot of red in them as well. This shows that just because an object is green doesn't necessarily mean it will be fully green in an RGB encoded image. In practice, objects of almost any color can have large amounts of the other colors in them as well.

While the technique of using a single channel does create a grayscale image, it doesn't accurately represent the source image. The best way to do this accurately is to use the luma because the luma will generally represent the intensity or lightness of objects of the image better. The luma can be derived from the RGB encoded image and is a measure of the energy an observer perceives [GW08]. The luma, often indicated by the letter "Y", combined with two chroma channels, I and Q create the YIQ color space. The YIQ color space was created by the National Television Systems Committee (NTSC) for transmission of video that could be used for both color and black and white televisions [BB82]. For the purposes of edge detection, the luma is a very reliable grayscale channel that gives an accurate representation of the intensity of the image. The luma is denoted as Y and be calculated as shown below:

$$Y = 0.299 * R + 0.587 * G + 0.114 * B \tag{2.1}$$

When equation 2.1 is applied to all of the pixels of an image, the result is a grayscale image similar to the one shown in figure 2.3.

2.3 Image Histograms

The histogram is a useful statistics and graphical tool that allow us to see how all possible discrete data values in a dataset are distributed. In the context of image processing, an image's histogram

Figure 2.3: The color fruits image and its luma grayscale representation. (Left: Color image. Right: Luma grayscale image.)

is a graphical representation of the distribution of all the pixel intensities of a given channel of the image. This means that the histogram is a function of the intensity and can be defined as:

$$h(I) = \sum n_I \tag{2.2}$$

I is an intensity value in the range $[0, 255]$ and n_I is the number of occurrences of the specific intensity I in that channel[GW08]. The histogram is usually graphically displayed as a simple two dimensional graph with the number of occurrences of the data on the y axis of the graph, and the range of possible data values on the x axis, in increasing order. The height of the curve in the graph over an intensity gives how many times that specific intensity appears in the image.

Since we have defined a pixel in an RGB encoded image to be a three dimensional tuple, in a color image, we actually can generate three different histograms. One histogram is the histogram of the red channel of the image. The second histogram is for the green channel. The final histogram is the blue channel. Additionally, other histograms can be constructed for other grayscale images, such as the luma, since these are derived from the RGB image. Figure 2.3 shows the histogram of the fruits image as well as the histogram of the grayscale version. Note that for the luma grayscale histogram, the red, green and blue histograms are equal. We could represent the data as a single histogram if we chose to.

Simply viewing the histogram of an image can often reveal a lot of information about the contents of the image. Using color histograms, a vision program can identify objects and be used as a way to identify one object as being separate from another object that has a different color histogram. [SB91] Similarly, even in a grayscale image, a histogram with two "hills" at different intensities can often suggest two objects [Ots79]. Using the histogram, we can also find the status of an image,

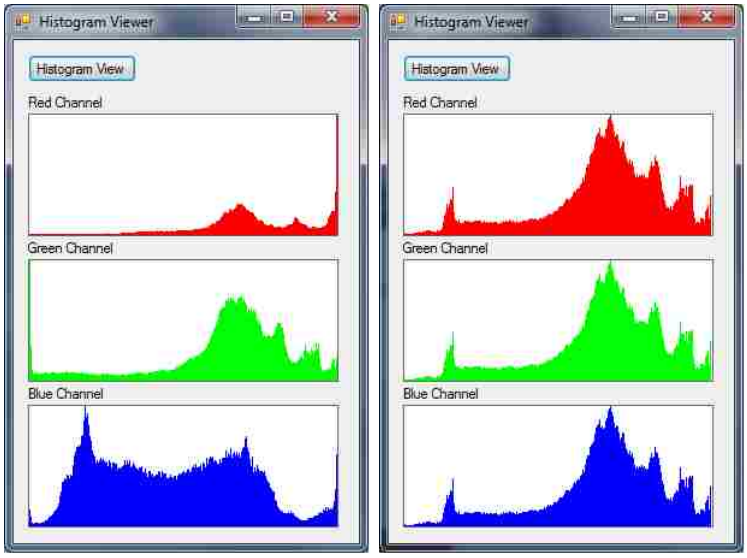

Figure 2.4: Histogram of the fruits image. (Left: Color image. Right: Luma grayscale image.)

such as identifying poor contrast, noise or poor lighting. Images with poor contrast generally tend to clump all of the values together. Images with good contrast will general span the entire $[0, 255]$ possible values for intensity [GW08]. Images with a lot of noise will generally have a jagged or disconnected histogram with gaps and spikes. If we examine the red, green and blue histograms of the dark image, all three histograms will generally be the same and tend towards lower values. Bright images will exhibit the opposite behavior, and the histogram will tend towards the higher end of the histogram. Figure 2.5 shows the histogram of a dark image. The histogram of a very bright image is similar, except the intensities will tend towards the larger values, rather than the smaller values. Images with histograms that are smooth and cover the entire range of values with a good even distribution are generally high contrast and low noise and ideal for using more complex algorithms such as edge detectors.

2.3.1 Histogram Equalization

In a dark image or a light image, the histogram will not generally use the entire range of $[0, 255]$ possible intensity values. Because all of the information of the image is compressed to a small range of intensity values, histograms like the ones in Figure 2.5 show one of the problems with images that are either too dark or too light. Instead of using the entire range of 256 values, these dark images often only use around 30 or 40 of the values [GW08]. Unfortunately, because this is the nature in which the images were taken, there isn't much that can be done to improve the actual entropy of the intensity values. We can, however infer a lot of information even from histograms that are of

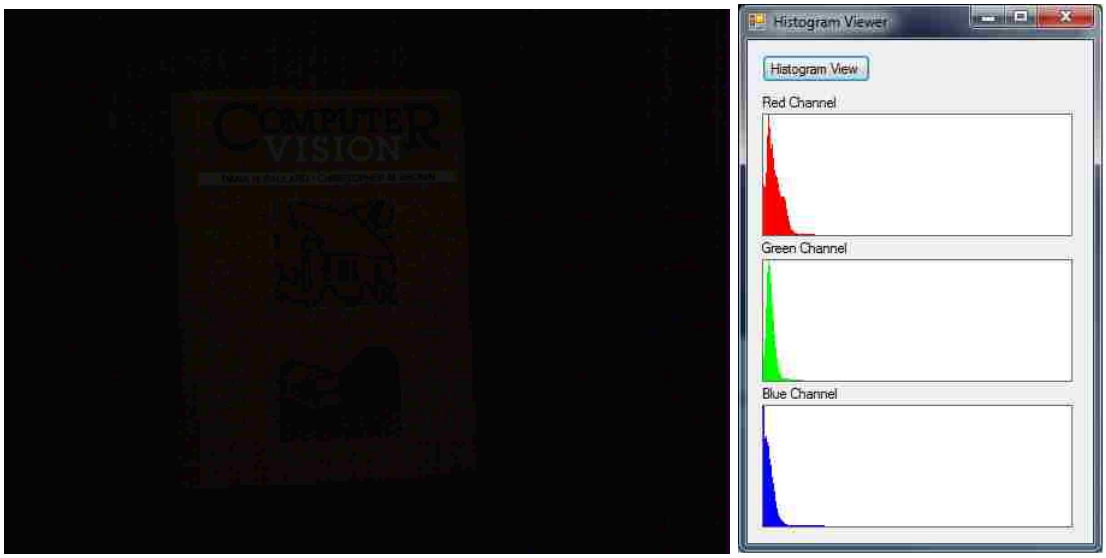

Figure 2.5: A dark image and its histogram.

this form. For example, if we notice that the red and green histograms of a color image are similar and are in the medium to high value range, but the blue histogram is very low in value range, we can conclude that the image is probably of some kind of yellow object.

Additionally, even though we cannot add information to the image that is not there, we are able to more strongly emphasize the information that is in the image already. If we stretch the histogram to cover the entire range of 0 through 255 values, and then map the old values to the, more well spread out new values, we are able to better see the changes in intensity that were otherwise very small and difficult to see. This process is known as histogram equalization. The image's intensity values are reassigned to new values that will better distribute the the data over the full range of values [Rus10]. The algorithm will greatly improve the contrast of images and make an image that otherwise appears to have no detail show the information in a much more clear way as figure 2.6 shows.

From the histogram, we can see that the image must be very noisy, which is indicated by the jaggy behavior of the histogram, but also has good contrast, since it expands the full length of intensity values. To equalize the histogram so we can get an image like the one in figure 2.6, we need to use the cumulative probability density function. The image is created as a mapping of the pixel intensity of the old image to a new intensity value determined by the equalization procedure $N(I)$. If I is the intensity of the pixel $P_{x,y}$ of the original NxM image and we can create the histogram $h(I)$ as defined in equation 2.3, the intensity of a new pixel in the resulting image $P_{x,y}$ is defined recursively as:

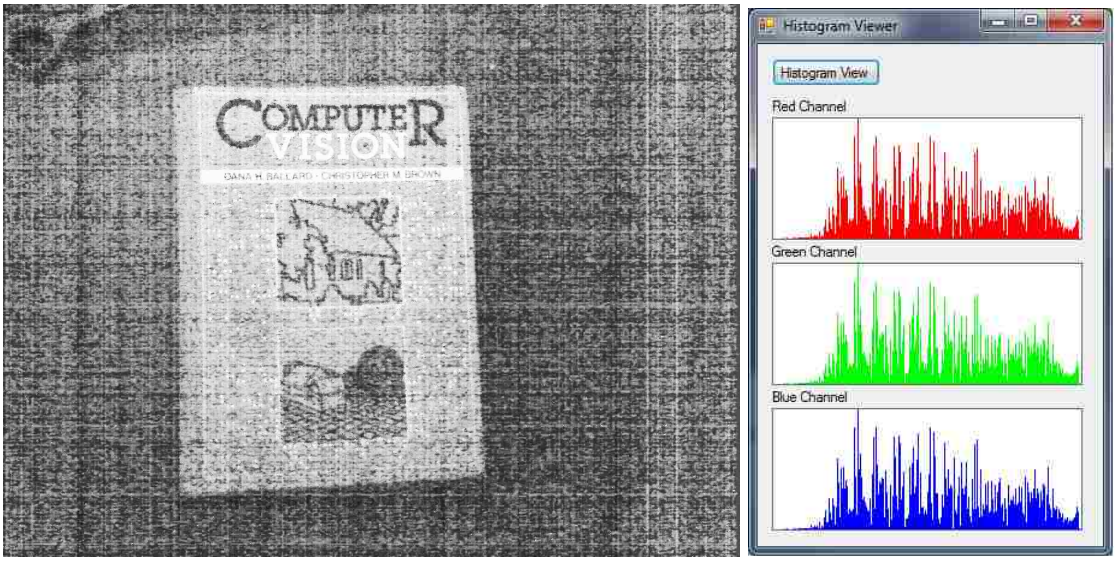

Figure 2.6: The dark image after histogram equalization, and its histogram.

$$P_{x,y} = N(I) = \begin{cases} (h(I)/(N * M)) * 255 & \text{if } I = 0 \\ (h(I)/(N * M)) * 255 + N(I-1) & otherwise. \end{cases} \tag{2.3}$$

The results must be converted back to an integer and clipped based on the RGB encoding constraints afterwards if they exceed the valid values. The results of this operation are shown in figure 2.6. When compared to the darker image in figure 2.5, the new image much easier to see and read, but the cost is that there is a lot of noise introduced. This is because atmospheric noise that would have once blended in has now also been stretched across the entire frequency domain. In order to help control these issues, we can make some modifications. For example, rather than using the entire set of 255 values, it may be better to stretch the histogram only over half of the possible values. Likewise, there are other filters we can use, such as blurring techniques, that will help control some of the problems related to this technique. To use these techniques though, we will have to explore more complicated image processing algorithms.

2.4 Digital Image Processing Techniques

Since images can be represented as a matrix of pixel values created as a function of the intensity values at a given x and y position, we can apply two dimensional linear algebra techniques to the intensities of images. Doing this allows us to modify an image in an easy and mathematical way. For example, if we have two NxM matrices, we can perform operations such as addition and subtraction between the two matrices. With images, if we have two NxM images, A and B, we can create a

new image C that is also NxM in size and define the pixels of the new image C to be defined as shown in equation 2.4 [Pit00].

$$C[x, y] = A[x, y] \pm B[x, y] \tag{2.4}$$

If the image is in color and is represented by RGB encodings, then simply add or subtract the matching color components between A and B. If the image is monochromatic, then you just add or subtract the single intensity value at each pixel location. If, during the process of adding, a value exceeds 255 or is below 0, then you should make that value 255 or 0 respectively. It is important to do this clipping afterwards because the color components of a pixel must fit within a byte in order to be stored and represented in an image. The clipping of these values ensures that the resulting image of the addition operations will be a valid RGB encoded image.

The task of adding and subtracting images has some specialized uses in digital image processing. For example, if image A is a photograph and image B is an image of the same size but contains the tuple (255,255,255) in locations where edges of A are, and all pixels otherwise contain the tuple (0,0,0), then adding A and B will result in an image with the edges of A lying on top of the original source photograph.

Another application of equation 2.4 is in background subtraction. Background subtraction is the practice of taking a photograph from a fixed vantage point and ensuring there are no objects of interest in the photograph. This is very common in video that will have motion or video taken from fixed vantage points [Pic04]. The camera will constantly be taking photos, including ones where the object is not in the frame. This image acts as the background. Later after some time, an object enters the scene and a second photograph is taken. Subtracting the two images, assuming there is no or minimal noise in the image, will result in only the pixels that have the object having non-zero intensity values, while the rest of the values will be subtracted out [BB82].

This algorithm can be useful for automatically supplying the pixels that belong to the object of interest. There are problems, such as shadows that can create edges where they do not exist. Likewise there are many algorithms that can be deployed to counteract against shadows. In general, this thesis will assume that a single photograph is being provided for analysis. We do use background subtraction as a convenience for selecting object histograms later though. Otherwise, manual operation needs to provide input to that step.

2.4.1 Convolution

Convolution is a linear system operation that takes the characteristics of both functions and combines them [BB82]. This is important in image processing because we can can describe an image as a two-

dimension function where the intensity of each pixel is the amplitude of that function. Likewise, we can create another function with aptitudes that carry characteristics we want to "apply" to the image. Convolution will displace the values of the image based on the values of the other function. The convolution of an image f and a convolution mask g is defined by equation 2.5 [BB82].

$$c(y) = f * g = \int_{-\infty}^{\infty} f(x)g(y-x)dx \tag{2.5}$$

Convolution is usually implemented as a four step process. First a convolution mask, sometimes called a convolution matrix or the kernel, is created. Convolution masks are typically small 3x3 or 5x5 matrices, that is applied to each pixel of the image. This often depends on the type of mask being created. The convolution mask is then centered on the first pixel of the image and then slid over the image, pixel by pixel. All values affected by the convolution matrix will be multiplied by the corresponding value in the matrix and their results summed together. This running sum is eventually assigned to the center pixel of the mask. Since the sliding of the mask starts at the top left most pixel, normally, it is necessary to consider what happens at the edges. For example, if a 3x3 convolution matrix is applied to the very first pixel of the image, then the first element of the convolution matrix will be multiplied to a pixel that doesn't exist. There are many techniques for handing these edge cases such as extending the image's edge pixels, or simply wrapping. Lastly each sum must be normalized, since the operations performed may result in values that are above 255 or below 0.

We can use convolution to achieve many effects. We can create masks that will detect edges or perform other pattern matching techniques [BB82]. The convolution algorithm can also be used to perform operations that affect a small neighborhood of pixels. For example, we can effectively use convolution to blur an image. Edge detectors that are created by convolution are limited to only looking in their local neighborhood though, so there are limitations to the algorithm.

Guassian Blur

As an example of convolution, one technique that will be important in this thesis is using the Gaussian blur to reduce the noise in the image. We are able to fill a convolution mask with the kernel's values with the height from the 2 dimensional Gaussian probability density function as defined in equation 2.6. Of course, doing this will result in a mask that consists entirely of fractional values. Normally, due to the efficiency of integer operations compared to floating point operations on the computer, we will want to normalize the matrix to be integer. A simple way to do this is to multiply the values in the matrix until they are whole numbers, round them, and then ensure

that the sum of all the values is equal to one by dividing the entire sum from the convolution by a constant. The Gaussian blur is a low pass filter based on the Gaussian distribution function [Bas02]:

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{\frac{-(x^2+y^2)}{2\sigma^2}} \tag{2.6}$$

Using the density function, we can define a convolution matrix for a given sigma value. The convolution matrix for the gaussian when $\sigma = 1.0$ is shown in 2.7. When the mask is created, the values will be the height of the 2-dimensional Gaussian distribution. We use it to perform the convolution by multiplying the mask values with the intensity values of the image, summing all the products for that pixel together and dividing by the constant. The output pixel will be an average of the neighboring pixels, weighted by the Gaussian distribution.

$$\frac{1}{273}\begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix} \tag{2.7}$$

The Gaussian blur, while a very simple algorithm, is also a very powerful and useful algorithm, specifically among image analysts that perform edge detection [Bas02]. Blurring the image before executing an edge detector can help the edge detector focus on more important edges and reduce the noise of the image. As a low pass filter, the algorithm is convenient since it can just be applied to every single pixel in the image, regardless of the current state that the image is in.

Edge Operators

Another application for convolution is in edge detection. There are many edge detectors that can be created by using convolution to set pixels that satisfy edge conditions equal to higher intensity values than the pixels that do not. These detectors work by sliding a mask over a given pixel's neighborhood and checking if the surrounding pixels define a particular pattern that would suggest that the intensity of the surrounding pixels has changed drastically. These drastic changes suggest that one pixel belongs to one object, while the other pixel belongs to a different object, or the background, so the current pixel is the edge between those two different objects. Creating the separation of object and background is one of the most important obstacles for image processing applications [Bas02] and edge detectors provide one way to accomplish that task.

The Sobel edge detector is a convolution operator, created by Irvin Sobel in 1978, that will estimate the first derivative of the pixel intensity values centered at a given pixel [Sob78]. By finding the rate of change in the intensity of pixel values, we can believe that areas of the image that have a large rate of change in pixel intensity are areas that signify edges of the image. Sobel's convolution mask to detect horizontal edges:

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \tag{2.8}$$

and vertical edges:

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \tag{2.9}$$

must be run separately, and then the results combined. To perform the full algorithm, we generally first convert the image to a grayscale, so we do not operate on all of the channels of the image at once. Afterwards the operator is applied. Sometimes it is desirable to run the entire algorithm against multiple channels [XJL06]. The advantages are that edges that would not have been detected in the luma are sometimes more readily accessible in other color channels. Additionally, other modifications to Sobel's detector can include blurring the image prior to execution. Using the Gaussian blur will reduce the noise that could be detected and mistook as an edge [Bas02]. Edges are defined to occur at the points where there is a large amount of change in the magnitude of the pixel intensities of the gray-scale image. The technique for identifying how large of a change will cause a specific pixel to be an edge is a simple thresholding algorithm that dictates values above the threshold are edges, and the ones below or equal to the threshold are not [SSAaS10].

There are many convolution mask edge detectors, with slight but noticeable differences in their results and reliability. Prewitt's detector, for example, is almost the same as the Sobel algorithm except Prewitt's algorithm doesn't give extra weight to the center of the mask [SSAaS10]. Prewitt's algorithm, like Sobel's detector, also requires detection on both the horizontal and vertical edges separately. The masks detect horizontal edges:

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \tag{2.10}$$

and vertical edges:

$$\begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \tag{2.11}$$

Although both Sobel and Prewitt have separate masks to detect horizontal edges vs vertical edges, that is not always the case for convolution mask edge detectors. The Laplacian operator, for example, uses a single mask that attempts to identify if the current pixel is next to a dark or light edge [SSAaS10]. The Laplacian mask is defined in equation 2.12.

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{2.12}$$

There are limitations to all of these convolution-based techniques. All of the above algorithms are all restricted to looking at their immediate neighborhood for calculating edges. This means that it is common for algorithms that rely exclusively on convolution to have connectivity issues after the algorithm has been completed when presented with images that may have edges that are implied through a more gradual change. These convolution edge detectors could have their masks extended to beyond 3x3 neighborhoods, which would help solve these problems, but introduces other problems with thicker edges on images that should have thin edges [BB82]. Additionally, because edges are determined by the local neighborhood rather than the entire image, the results from these algorithms will generally give more edges, including ones that are not as important when looking at the entire image. In order to improve accuracy of edge detection, a more robust edge detection operator that ensures connectivity and accuracy is required.

2.5 Edge Detection Using Canny Edge Detector

One of the most widely used edge detection operators was created by John Canny in his 1986 paper "A Computational Approach to Edge Detection." [Can86] Canny's edge detection algorithm operates in 5 phases:

1. Noise Reduction

2. Gradient Calculation

3. Non-Maximal Suppression

4. Thresholding

5. Hysteresis

Unlike the other edge detection algorithms discussed so far, Canny's edge detector is not a convolution mask detector, though it does use a convolution mask during the gradient calculations like the other convolution mask algorithms do. This means that the algorithm is not as easily harmed by the downsides of the previous edge detection algorithms. The algorithm usually begins with a grayscale version of a color image. Noise reduction is typically accomplished by a low pass filter such as a Gaussian blur on a specific color channel of the image, typically the grayscale. Once the noise has been reduced, the gradient is calculated from that channel. The gradient is determined by the difference in intensity compared to nearby pixel values, similar to or even just using the Sobel and other edge detection algorithms [Can86]. The larger that difference in intensity is, the larger the gradient at that point will be. During non-maximal suppression, we find the local maxima of the gradients and remove, or suppress, other values. Thresholding will take the results of the suppression and mark pixels into one of 3 categories: definite edges, non-edges, and "maybes" that may be an edge or may not be an edge and the results of the final step will make that distinction. Hysteresis will perform a recursive edge trace routine, starting from definite edges and flipping the "maybe" edges to edges if they are adjacent to an edge. At the end, all unreachable maybe edges are removed and all that remains are the edges of the image. This final step helps ensure connectivity that may have been lost during non-maximal suppression [Can86]. The final two steps make the algorithm very different from the convolution mask algorithms and improve the accuracy of the algorithm at the cost of time to resolve the "maybe" pixels and perform the recursive edge trace algorithm during hysteresis.

Canny's edge detector is a very powerful edge detector and will work in a large number of situations. Despite the advantages the detector gives, there still can be situations where the algorithm cannot perform up to task. As a result, there have been many adaptations of this algorithm that can change the quality of the detection or specialize it for a specific need. Other techniques in the field of computer vision can also be applied to Canny's edge detector to improve or interpret the results from the detector. For example, using multiple color channels, rather than just the traditional grayscale channel in a single lighting condition can give more reliable results [XJL06]. Using multiple color channels for the entire execution of the algorithm allows for the algorithm to adapt better to various lighting conditions and the edges introduced by shadows can be identified and removed and give a more reliable result than the single-channel Canny edge detector [XJL06]. Other techniques include

doing color correction to correct for camera and the exposure, rather than the lighting, of the image. By first doing color correction for the camera's exposure and finding the best contrast for the image, the Canny's edge detector can better adapt to poor cameras and can get better results [NPKJ08].

In this thesis we will be looking at using smarter color channels selection to improve the contrast of the image. By using the histogram of the object and the histogram of the background where the object rests, we can customize the color channel selection to best suit the situation automatically and improve the detection of edges for the object. There are many techniques for improving that we will deploy that will improve the contrast of an image. The second major modification we will be performing is using the histogram equalization algorithm after a color channel has been selected. We will then modify the algorithm to be progressive and gradually modify the threshold rather than do all of the changes at a single point. This will reduce the noise enough to get better detection on objects that would otherwise require manipulating the Canny thresholds or executing additional low pass filters like the Gaussian blur. We hope that these choices not only improve the general results of the image selection when confronted with images that may lose information from the grayscale conversion, but we will be looking at the problem from a security perspective. Objects that were once dark can now be illuminated, and while sometimes these dark images can't easily have edge detectors run against them due to the noise, we want to ensure that all objects, including ones masked in darkness or light can be seen. Likewise, our algorithms will help solve problems pertaining to objects that blend into the background when a grayscale conversion is performed.

Chapter 3

Implementation

3.1 Overview

This implementation of Canny's edge detector assumes there is a human operator that can help guide the algorithm, but the implementation also allows for automated processing. The implementation works as a pipeline, starting with a loaded image and presents several options which can mostly be run in any order the human operator chooses. For example, the color channel selection screen shown in Figure 3.1 lets the operator choose between the luma grayscale, as well as the red, green or blue channels. There is also an automatic selection, which will be discussed shortly. Each time an algorithm is executed, a new tab with the intermediate resulting image is shown. Figure 3.2 shows the program after the program has executed a channel selection and a Gaussian blur. The top of the screen has multiple tabs that when clicked show the intermediate results of each step of the algorithm taking place. This allows for the operator to see all of the intermediate calculations to try to figure out what the best choices are for The bottom tabs are for manual selection or customization of various options that the algorithm can perform. This also allows the user to execute filters multiple times. This is mostly relevant to the Gaussian blur filter.

The automatic feature will run the traditional Canny edge detector steps as discussed in Chapter 2. The algorithm will go through channel selection with the luma, Gaussian blur, gradient calculations, non-maximal suppression, and finally the hysteresis. Some of the specific sub-features, specifically the color channel selection can be done automatically. When this is done, the grayscale isn't necessarily the chosen color channel. Instead, we use our own algorithm for deciding on an optimal color channel for the specific image that tries to maximize the chance that the object will contrast against its background.

This implementation of Canny's edge detector also provides some other minor convenience oper-

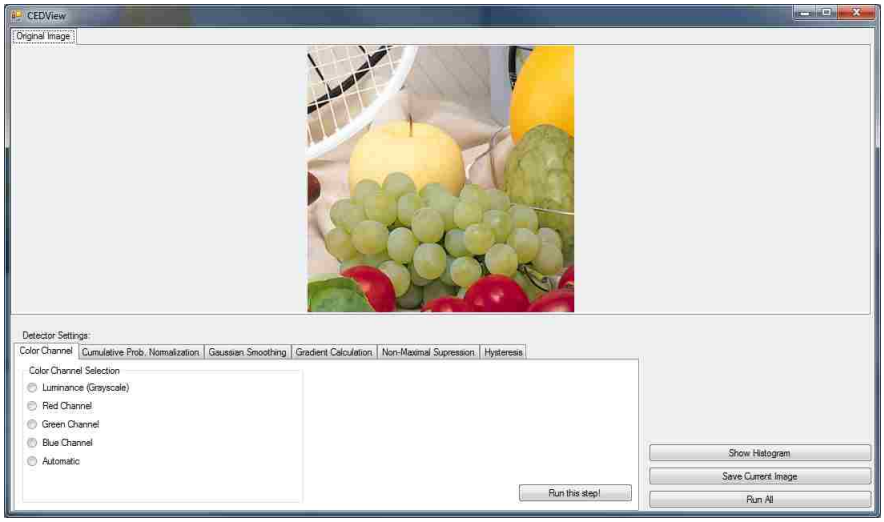

Figure 3.1: Screenshot of the program selecting color channel.

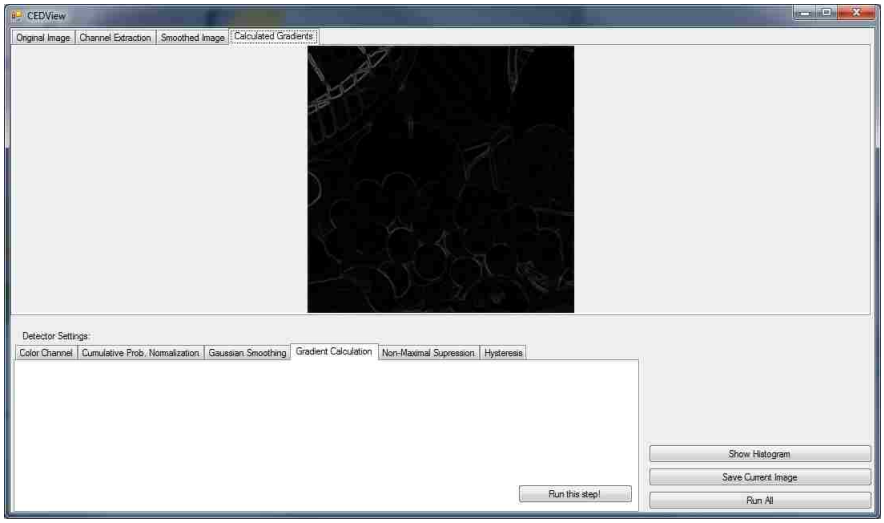

Figure 3.2: Screenshot of the program after executing a channel extraction, Guassian blur and gradient calcuations.

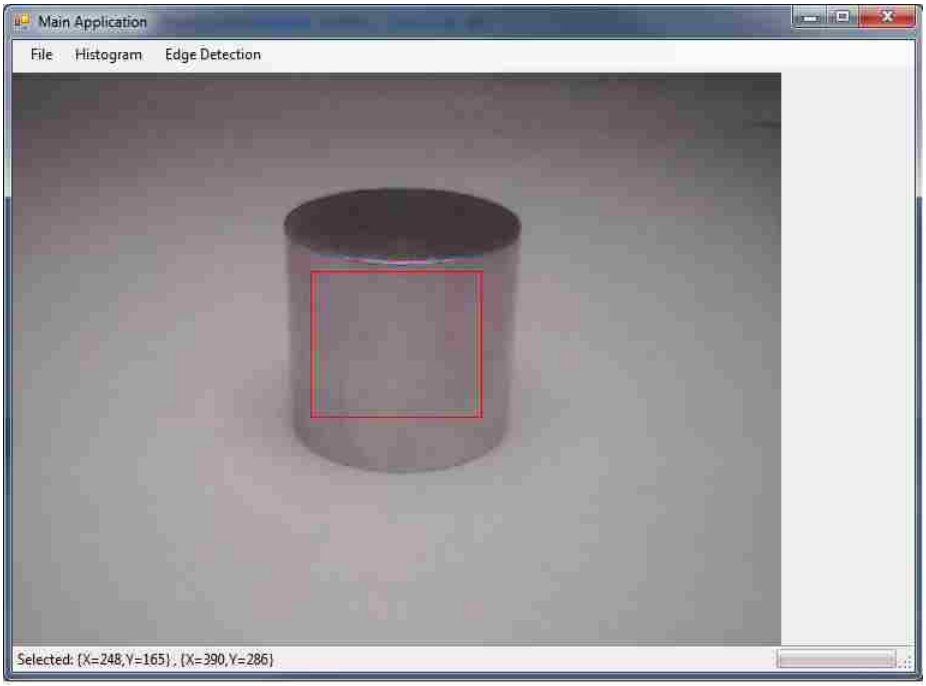

Figure 3.3: Screenshot of selecting a portion of the image to view that part of the image's histogram.

ations such as saving the current image to disk and viewing the histogram of the image or a selection of the image. The selection of a specific area and viewing its histogram, as shown in Figure 3.3, is used by the automatic color channel selector later in the algorithm. The application was written in Microsoft Visual Studio 2010 Professional in the programming language C#. Because the application is quite large, only some of the more important code is provided in the appendix at the end of this thesis. All of the major algorithms will be discussed in this chapter and the specifics behind our implementation of them.

3.2 Color Channel Selection

Normally, when using Canny's edge detector, the image would first be converted into a grayscale image and the algorithm would run against the luma as described in Chapter 2. This works for a large majority of situations and is generally the safest approach to edge detection. But also, as was discussed, as a result of the conversion to grayscale, we have lost a large amount of information. As a result of color information being converted into a set of values in the range of $[0, 255]$ from a much larger range, it is obvious that there are several instances where different RGB encoded pixels will equate to the same grayscale shade. A good way to ensure the loss of information doesn't harm detection, or can even be turned around to aid the detection, is to use a multi-channel edge detector

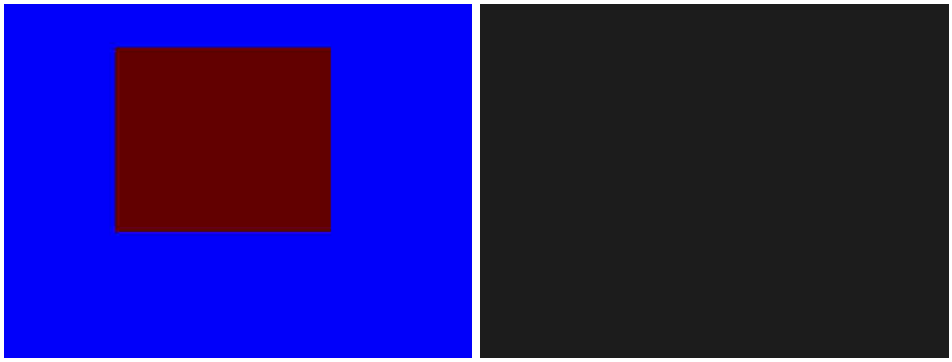

Figure 3.4: Example of an object disappearing when viewed with the luma.

[XJL06] and run the entire algorithm against the color image itself, and it turns out that many of this implementation's algorithms such as blurring were designed to do that in our implementation. We, however opt not to do so, in favor of just using the traditional single-channel approach to gradient calculations, and using extra context to suggest extra information that may be useful for color segmentation. This decision is due to the main algorithm that improves our results, the contrast improvements created by our dynamic histogram equalization algorithm, cannot be implemented in 3 channels without experiencing color distortion issues.

In order to maximize the potential of our detector, we will want to create as much contrast from the object we are interested in detecting from its background or other objects of the scene. If we knew what the object was and what the background was, we wouldn't need to run an edge detector, so the most reliable method has been to use the luma. However, if we can get some additional information about the object, we may be able to use that information to do as good as or better than using the luma. As figure 3.4 shows, the red box in the image is clearly visible, and even simple edge detectors such as Sobel's or Robert's edge detectors should be capable of detecting the edges. The problem is that when the image is converted initially to the luma grayscale, the intensity of the blue color and the intensity of the red box happen to match and therefore, in the grayscale, there are absolutely no objects to detect. If however, we look at the other channels individually as in Figure 3.5, specifically either the red or the blue channels, we will find that the box is still clearly visible and if we had used one of those two channels we would have been able to find the edges correctly.

In general, when performing edge detection, matching the most accurate representation of the light and how humans view objects isn't necessarily the best option. Instead, in the context of the problem of edge detection, we simply want maximal contrast between our object of interest and the background. Using the histograms and general statistics like the mean intensity of each channel of the color image, we can get an idea of what the general color of the object is and the general color of the background.

Figure 3.5: The red box when viewed with the red and blue channels. (Left: Red channel. Right: Blue channel.)

Our implementation of Canny's edge detector allows for the human operator to select any color channel that they believe will be optimal. The easiest solution is to use the luma and rely on other aspects of Canny edge detector to do their work and handle bad properties the image may have. Canny's edge detector with the hysteresis algorithm can be very powerful, and resolve connectivity issues at the end of execution [Can86]. Generally, the good way to choose a channel to operate against is to look at the histogram of the object and the histogram of the background and maximize the difference in average values of the histograms. This is the approach our automatic selection will take. The algorithm for this is outlined in algorithm 1.

First we will need to define a set of pixels as the object pixels and a set of pixels as the background pixels. Our program lets you drag and drop a box to select a rough area of the image as input for this task. From this, we create a histogram of each selection. One of the selections should be over the object you are interested in detecting, while the other is of the background that is close to your object. Another way to supply this input is if you have a separate background image. In this scenario, the background image has its histogram used, and then a background subtraction is performed and the pixels over a threshold of around 16 or so intensity values, after subtraction, is included in the object histogram.

3.3 Histogram Equalization

After the color channel has been selected, we can still further attempt to improve the contrast of the image for the purposes of edge detection by performing histogram equalization. As was discussed in Chapter 2, histogram equalization will stretch the histogram of the image over the entire range of $[0, 255]$ values and will generally improve the contrast of the image as a result. In our implementation of Canny edge detector, we almost always want to perform this operation on images before we do the Gaussian blur, but after we do color channel selection. The algorithm offers several advantages over

Algorithm 1 Automatic Color Channel Selection

```
1: procedure SelectColorChannel(img1, img2)  ▷ Input is two sub images selected by user.
2:     ▷ Create the histograms for both provided images. The histograms actually contain an R, G, and B component.
3:     ▷ Build the histogram for img1
4:     for x ← 0, until x = img1.width, step x ← x + 1 do
5:         for y ← 0, until y = img1.height, step y ← y + 1 do
6:             for c ← 0, until c = 2, step c ← c + 1 do                ▷ This is for each channel
7:                 hist1[c, img1[x, y]] ← hist1[c, img1[x, y]] + 1
8:             end for
9:         end for
10:    end for
11:    ▷ Build the histogram for img2
12:    for x ← 0, until x = img2.width, step x ← x + 1 do
13:        for y ← 0, until y = img2.height, step y ← y + 1 do
14:            for c ← 0, until c = 2, step c ← c + 1 do
15:                hist2[c, img2[x, y]] ← hist2[c, img1[x, y]] + 1
16:            end for
17:        end for
18:    end for
19:    ▷ Find the means for each of the histograms.
20:    rmean1 ← mean(hist1[0])
21:    gmean1 ← mean(hist1[1])
22:    bmean1 ← mean(hist1[2])
23:    rmean2 ← mean(hist2[0])
24:    gmean2 ← mean(hist2[1])
25:    bmean2 ← mean(hist2[2])
26:    ▷ Find the absolute differences between corresponding means.
27:    rdiff ← |rmean1 − rmean2|
28:    gdiff ← |gmean1 − gmean2|
29:    bdiff ← |bmean1 − bmean2|
30:    if rdiff > gdiff & rdiff > bdiff & rdiff > threshold then
31:        return "R"                                    ▷ We should use the red channel
32:    else
33:        if gdiff > bdiff & rdiff > threshold then
34:            return "G"                                ▷ We should use the green channel
35:        else
36:            if bdiff > threshold then
37:                return "B"                            ▷ We should use the blue channel
38:            else
39:                return "L"                            ▷ We should use the luma.
40:            end if
41:        end if
42:    end if
43: end procedure
```

detecting edges without contrast correction. Better contrast in the image often results in having larger gradient values on edges. By improving the magnitude of gradients when the edge detector is finally run, Canny's edge detector thresholds will be more likely to emphasize important edges because they will be closer to the high threshold, or at least be less likely to be destroyed by the lower threshold. Algorithm 2 shows the algorithm for histogram equalization as it appears in our implementation.

Algorithm 2 Histogram Equalization

```
1: procedure EqualizeHistogram(img, max_intensity)   ▷ max_intensity is normally 255, but can be customized.
2:     total ← img.width * img.height
3:     hist ← int[255]
4:     ▷ Build the histogram
5:     for x ← 0, until x = img.width, step x ← x + 1 do
6:         for y ← 0, until y = img.height, step y ← y + 1 do
7:             hist[img[x, y]] ← hist[img[x, y]] + 1
8:         end for
9:     end for
10:    chist ← int[255]
11:    ▷ Build the cumulative probablity histogram
12:    for i ← 0, until i = 256, step i ← i + 1 do
13:        if i = 0 then
14:            chist[i] ← (hist[i]/t) * max_intensity
15:        else
16:            chist[i] ← (hist[i]/t) * max_intensity + chist[i − 1]
17:        end if
18:    end for
19:    ▷ Create a result image based on the old image, but using chist to map intensities
20:    for x ← 0, until x = img.width, step x ← x + 1 do
21:        for y ← 0, until y = img.height, step y ← y + 1 do
22:            result[x, y] ← (int)chist[img[x, y]]
23:        end for
24:    end for
25:    return result
26: end procedure
```

If the image has very good contrast already, there isn't much general harm in running the algorithm since the probability distribution function will simply select the same general intensity values that were already in place before. If the image has poor contrast however, like dark or light images, or even images that sit in the middle of the histogram range [BB82]. This algorithm's main downside, however, is a consequence of its strengths. Histogram equalization reveals objects that are otherwise obscured by dark, light or other low contrast issues, but introduces noise created by amplifying the intensity values of the noisy elements of the image [Sta00].

3.3.1 Progressive Histogram Equalization

In addition to the traditional histogram equalization approach, we implemented a slight but very useful modification to the algorithm. The histogram equalization algorithm will map the entire image's range of intensity values to the range of $[0, 255]$ in a single run. This uses the probability distribution and will move intensity values to a more evenly distributed histogram. This has the downside of allowing for noise from the image to be enhanced along with it [Sta00]. Our solution for this noise is to equalize the histogram of the image in steps. This will make the equalization process more accurate and create a better and more accurate distribution of the values.

To do this, our implementation needs three pieces of information. The first is a maximum intensity. This is the intensity that the histogram will expand to. Normally this is just 255, just like traditional histogram equalization. However, unlike traditional histogram equalization, we will not expand the histogram to that value at first. Instead, we will expand to a new start intensity range. This starting range should idealistically be very close to the range that the existing image already occupies. For example, if an image uses 40 intensity values in its current histogram and the remainder are mostly or entirely empty, then we want our starting intensity for histogram equalization to only go to 41 or 45 in range. Lastly, there is a step variable. The step variable is a small non-zero integer. Typically a good number is around 5. In our implementation, we actually build the resulting image each time so we can display the result to the user. This isn't actually necessary, and probably isn't recommended in a real-time application. Instead, you can simply keep progressively applying the histogram equalization algorithm using the results of the previous round as the input for the next round. Not rebuilding the image would make it more efficient. We prefer to see the algorithm and generate the corresponding images so we can see the progression of the histogram expansion.

The algorithm we use for this modification is listed in algorithm 3. We actually just call on our existing implementation of histogram equalization from algorithm 2 and use the max_intensity variable that already exists to our advantage.

Algorithm 3 Progressive Histogram Equalization

```
1: procedure ProgressiveEqualizeHistogram(max_intensity, start_intensity, step, img)  ▷
   start_intensity should be set to roughly the range of intensity vaules in the initial image.
2:     for x ← start_intensity, until x > max_intensity, step x ← x + step do
3:         result ← EqualizeHistogram(img, x)
4:     end for
5:     return result
6: end procedure
```

The results of progressive histogram equalization compared to traditional histogram equalization are subtile, but when we use edge detectors later, very effective. Figure 3.6 shows the "Lena" picture,

Figure 3.6: Unmodified Lena picture.

Figure 3.7: Lena after performing histogram equalization and progressive histogram equalization. (Left: Traditional histogram equalization. Right: Progressive histogram equalization.)

used by many in the image processing field, before the histogram equalization algorithm has been executed while figure 3.7 shows the lena image after both histogram equalization and progressive histogram equalization. There is a slight difference in the two, specifically around the face, where the progressive algorithm has brightened and improved the contrast further than the traditional algorithm. We will see in the results chapter how the noise amplified by the progressive histogram equalization algorithm is more manageable than the regular algorithm.

3.4 Noise Reduction

Now that the image is in a single channel and the contrast of the image has been increased, we have likely introduced some degree of noise in the operations, specifically with the histogram equalization if the image required a large amount of contrast stretching. Even the progressive variation can introduce some noise, specifically in the case of extremely dark or very low contrast images. Normally, even without running histogram equalization, the Canny edge detector blurs the image to reduce noise anyhow, so it is convenient to do the equalization before properly starting Canny's edge detector. The first phase of Canny's edge detector focuses on minimizing noise by creating a Gaussian blur on the grayscale image. The Gaussian blur is a probability density function that is applied to each pixel at a given x and y position. As discussed in chapter 2 we can use the density function

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{\frac{-(x^2+y^2)}{2\sigma^2}} \tag{3.1}$$

to create a convolution mask that represents a discrete approximation of the Gaussian function [GW08]. In our implementation, we decided to make the $\sigma = 1.0$ static. This is mostly for simplification of the code, since the mask itself can be provided as a constant rather than created at runtime. If additional blur levels are required, the blur can be executed multiple times. Using $\sigma = 1.0$ we can compute the mask as follows:

$$\frac{1}{273}\begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix} \tag{3.2}$$

In order to use the mask, we have to perform the convolution algorithm. Algorithm 4 shows the details of the blur's implementation and the convolution algorithm that it uses. This algorithm was actually implemented to be multiple channel, since it could happen at any point in the im-

age processing pipeline, even though it is recommended that it be used after equalization and/or color channel selection. If there are not other channels in the image, which is implemented as a three dimensional integer array in the application, it will not run the blur on those channels. The implementation of the algorithm itself is a standard convolution using the mask from equation 3.2.

Algorithm 4 Guassian Blur

procedure GUASSIANBLUR(*img*)
    $mask \leftarrow \begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$
    $const \leftarrow 273$
    ▷ Build the histogram
    for $x \leftarrow 0$, until $x = img.width$, step $x \leftarrow x + 1$ **do**
        for $y \leftarrow 0$, until $y = img.height$, step $y \leftarrow y + 1$ **do**
            for $currChannel \leftarrow 0$, until $currChannel = channels$, step $currChannel \leftarrow currChannel + 1$ **do**
                $sum \leftarrow 0$
                for $i \leftarrow 0$, until $i = mask.width$, step $i \leftarrow i + 1$ **do**
                    for $j \leftarrow 0$, until $j = mask.height$, step $j \leftarrow j + 1$ **do**
                        ▷ Indexes image based on current mask index and ensures index is in the image border.
                        $indexi \leftarrow (x - mask.width/2 + i + img.width)\%img.width$
                        $indexj \leftarrow (y - mask.height/2 + j + img.height)\%img.height$
                        $sum \leftarrow sum + img[indexi, indexj, currChannel] * mask[i, j]$
                    end for
                end for
                $sum \leftarrow sum/const$
                $result[x, y, currChannel] \leftarrow min(max(sum, 0), 255)$ ▷ Ensures valid RGB encoding
            end for
        end for
    end for
    return *result*
end procedure

3.5 Gradient Calculations

Once the image is smoothed by the Gaussian, we can begin gradient calculations. For a given pixel at the position x and y we say that the intensity of that pixel is denoted as $I(x, y)$. The gradient for a specific pixel of the image has both a direction and magnitude, and can be represented as a vector. The gradient vector's magnitude, $||\nabla I(x, y)||$, indicates the magnitude of the rate of change in intensity values between pixels while the direction of that vector indicates the direction towards the difference [Can86]. Therefore the magnitude $||\nabla I(x, y)||$ contains a lot of valuable information

regarding the edges of an image, since we can eventually use thresholding to indicate that magnitudes above a certain value are definitely an edge and magnitudes below a certain threshold are definitely not an edge. We calculate each gradient as follows:

$$\frac{dI}{dx} \approx \frac{I(x+1,y) - I(x-1,y)}{2} \tag{3.3}$$

$$\frac{dI}{dy} \approx \frac{I(x,y+1) - I(x,y-1)}{2} \tag{3.4}$$

This finds the gradient for the individual x and y components of the gradient vector respectively. To find the magnitude we simply sum the absolute value of the components [Can86].

$$||\nabla I(x,y)|| = \sqrt{I(x)^2 + I(y)^2} \approx |dI(x)| + |dI(y)| \tag{3.5}$$

That is, the magnitude of the intensity vector for a given pixel is approximately equal to the absolute value of the individual components of the vector added together [Can86]. In our implementation, we simply store this in a three-dimensional array similar to an image, but with the X component, the Y component and the magnitude instead of color channels.

3.5.1 Non-Maximal Suppression

In Canny's original edge detection algorithm, one of the main innovations the algorithm introduced was further processing on the gradient values to reduce the amount of false positives [Can86]. Canny's algorithm does this by going pixel by pixel and finding the direction of the gradients. Generally it is best to create an estimation of the direction of the gradient and round them to multiples of 45 degrees.

First the sign of the gradients is considered. The signs of the gradients will determine which of the four quadrants the gradient is pointing towards. Once the direction is known, the absolute value of the gradients is considered [Can86]. For quadrant 1 of the circle, there are five possible locations where the gradient could lie. It could be directly on the 0°line, which will happen if $\frac{dI}{dy} = 0$. Similarly, if the $\frac{dI}{dx} = 0$, we know the direction of the gradient is on the 90°line. If $|\frac{dI}{dx}| = |\frac{dI}{dy}|$ and we know the direction of the gradient is in the first quadrant, we know the direction of the gradient is on the 45°. The last two possibilities for quadrant 1 is the direction of the gradient falls between the 0°and the 45°, or the direction falls between the 45°and the 90°.

Once we have estimates for the directions of the gradients we now know which set of pixels to compare the current pixel to. If the current pixel has a gradient on the 90°or 275°, for example, we know that we should compare the current pixel to the one directly above and directly below. Similarly, if we know the direction of the gradient is in a 45°or 225°we know that the two pixels we should compare to will be the ones on the corresponding diagonal. As for gradients that fall between one of the major lines, we look at the pixel it points towards. For example, if the gradient is on the 45°line, we would look at pixels $I(x+1, y-1)$ and $I(x-1, y+1)$. If, however, the gradient is in-between the 0°and the 45°lines, we will look at pixels $I(x+2, y-1)$ and $I(x-2, y+1)$ [GW08].

Once we know the two other pixels, we compare the current pixel's gradient magnitude $||\nabla I(x, y)||$ to the other pixel's magnitudes. If the current pixel is not greater than both of its neighbors, then it is suppressed and removed from consideration of being an edge. In implementation, we signify this by setting the pixel's gradient magnitude to 0. If the current pixel's gradient magnitude is maximal, then the value of the gradient is kept as is and must wait for thresholding during hysteresis.

3.5.2 Hystersis

The final step of Canny's edge detector is to resolve discontinuities created during non-maximal suppression and thresholding [Can86]. The results of the non-maximal suppression has resulted in parts of the image that had the greatest potential to be an edge remaining. Not all of the results will be edges, or at least edges we intended. Instead, some may be noise from the background or created by inconsistencies of photographic information. The gradients will also not indicate the edges exactly, and they will contain a large amount of area around the edges. To "trim" off the areas around true edges we can use an image segmentation thresholding technique [Can86].

First we must select two thresholds T_L and T_H for the low and high thresholds, respectively. The problem of threshold selection will be resolved later in the following improvement sections of this chapter, but for now the threshold is selected by a human operator. Regardless, the method of selection for threshold values, we must ensure that $T_L < T_H$.

Since the image is represented in grayscale, the pixels are represented by a single integer ranging from 0 to 255. If $||\nabla I(x, y)|| >= T_H$ then $I(x, y)$ we set $I(x, y) = 255$. This indicates that the pixel will appear white in the final image and is considered to definitely be an edge based on the thresholding selection. If $||\nabla I(x, y)|| <= T_L$ then $I(x, y)$ we set $I(x, y) = 0$. This indicates that the current pixel definitely is not an edge and should not be considered further.

The process has not yet considered the pixels that were neither set as edges or set as definitely not being edges that lie in-between the thresholds. If $T_L < ||\nabla I(x, y)|| < T_H$ then we set $I(x, y) = 128$. This indicates that the pixel is a "maybe" that should be resolved through the recursive edge tracing

step of hysteresis.

The edge tracing algorithm can be implemented in various ways. One of most commonly deployed edge tracing algorithms used with Canny's detector is a small recursive function that scans the image for pixels marked as edges from the hysteresis thresholds. Once a definite edge has been reached, the function looks at all the neighbors of the edge to see if there are any "maybes" directly adjacent to it. If there are it will set that maybe to an edge and begin looking for maybes from that point, recursively. Each time a pixel is visited, it is added to an array. This array will help ensure that the algorithm performs quickly. The result of this execution is an image where all edges are surrounded only by other edges or non-edges. The remaining "maybes" are isolated from the gradients that were selected by the thresholding step because they were not large enough to be considered an edge and also not in the close proximity to another edge. At this point a final scan through the image is run and all maybes are removed and set to non-edges.

Chapter 4

Results

In order to test our algorithm, we needed to find or create suitable images. This thesis has two primary goals. The first goal is to use user input about the object of interest, either through manual box selection of the object, or background subtraction from a given background image, to automatically select a color channel that gives good contrast for edge detection. There are cases where using the luma doesn't work as well as a different color channel, and the algorithm should try to find those situations and create a color channel selection that will minimize those effects and hopefully maximize the contrast of the image. The second objective is to use histogram equalization to improve contrast of images that do not use the entire domain of intensity values. This depends heavily on the Gaussian blur being able to help clean the image from noise that may be introduced in the process. Finally we look at progressive histogram equalization and show that the noise amplified by the progressive algorithm is more manageable without the need for additional blurring.

In order to evaluate each of the techniques, I did them separately. That way, if one of the techniques is not working very well, it will not hinder the performance of the other. I compared the results to running the images through the standard Canny edge detector stack. Typically Canny edge detector first uses the luma grayscale, blurs the image using the Gaussian blur, calculates the gradients of the image, performs non-maximal suppression and finally uses hysteresis to connect the edges and finalize the image. To make the tests easier, when doing a test, we used the same hysteresis thresholds among all the tests to ensure that better thresholding didn't cause the result improvement. The results improvement can only be because of the element we are testing.

Figure 4.1: The base cylinder image.

4.1 Evaluating the Color Channel Selector

The first algorithm we tested was the automatic function of the color channel selector. The theory is that by choosing the channel that maximizes the difference between the background and the object, we get better contrast from the background and foreground objects.

The first set of tests were on images of a small metal cylinder. The source image shown in Figure 4.1 is a mostly grayscale image. Looking at the photograph, we can notice that the cylinder itself tends to have a reddish-hue and the top of the image actually has some of the reddish hue as well. This is likely due to poor lighting conditions, the lens of the camera, or perhaps both. While we could use color correction techniques [NPKJ08] to correct for some of these issues we are going to simply attempt to use a color channel that maximizes the contrast rather than modify the image further for the first test of our edge detector. If we study the histograms of the cylinder and the background as shown in figure 4.2, we can see that indeed the cylinder does have a slight reddish hue to it, but the background's red values are also somewhat high.

The fact that both the background and the object have red hues to them means that the difference between the two isn't as great as it could have been. This property of their histograms means that the differences of the means for that channel are somewhat small, and despite being the most highly represented color is not selected. This is actually a good thing, since it is important to remember that the goal of the color channel selection is to pick a color channel that maximizes contrast, not necessarily be the most heavily represented color channel in the image. Since the background has a lot of red and the object has a lot of red, when the red channel is extracted, details are harder to

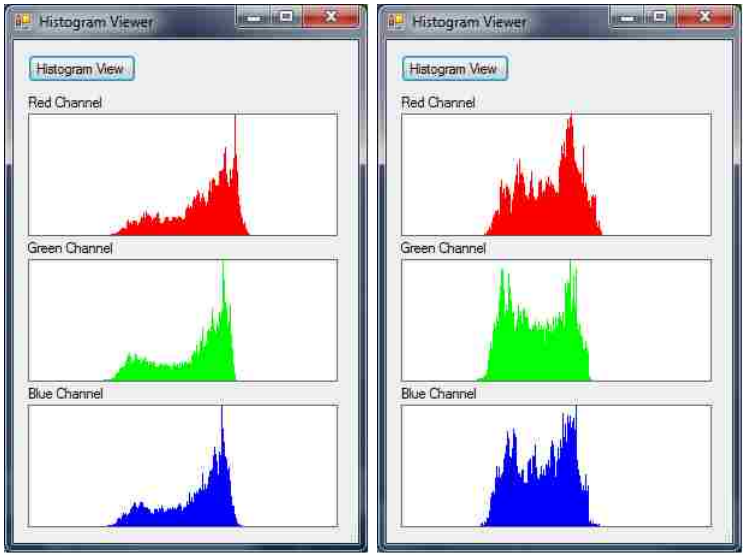

Figure 4.2: Histograms of the background and the cylinder from user selection. (Left: Background. Right: Object.)

Channels	Background Mean	Object Mean	Difference	Selection
Red Channel	148	122	26	Not Selected
Green Channel	145	107	38	Selected
Blue Channel	144	111	33	Not Selected

Table 4.1: Color Channel Selector's logic.

identify. The cylinder's color channels are shown in Figure 4.3.

The color selection algorithm automatically selects the channel that it believes will have the most contrast by looking at the differences between the background's histogram and the object's histogram. The means of the histograms for both the object and the background, as well as the differences considered are shown in the table 4.1. As we stated, despite the red channel having larger values, both the background and the object had large red values, so ultimately the red channel wasn't selected. Because the difference in background and the object was greatest in the green channel, and by a somewhat large number, 38, the green channel was selected as the color channel to execute the algorithm against.

The results of running the remainder of Canny's edge detector on the cylinder image with the hysteresis thresholds T_L and T_H set to 40.0 for the lower threshold and 90.0 for the upper threshold is shown in Figure 4.4. There are better choices for hysteresis thresholds than this selection and we would likely be able to detect the cylinder sides a bit better, but the purpose of this test isn't necessarily to detect the edges of the cylinder, rather, compare the results of choosing the green color channel compared to the traditional selection of the luma channel. If the thresholding isn't perfect, then there is room for a decline or increase in quality of the detection. In the green and the

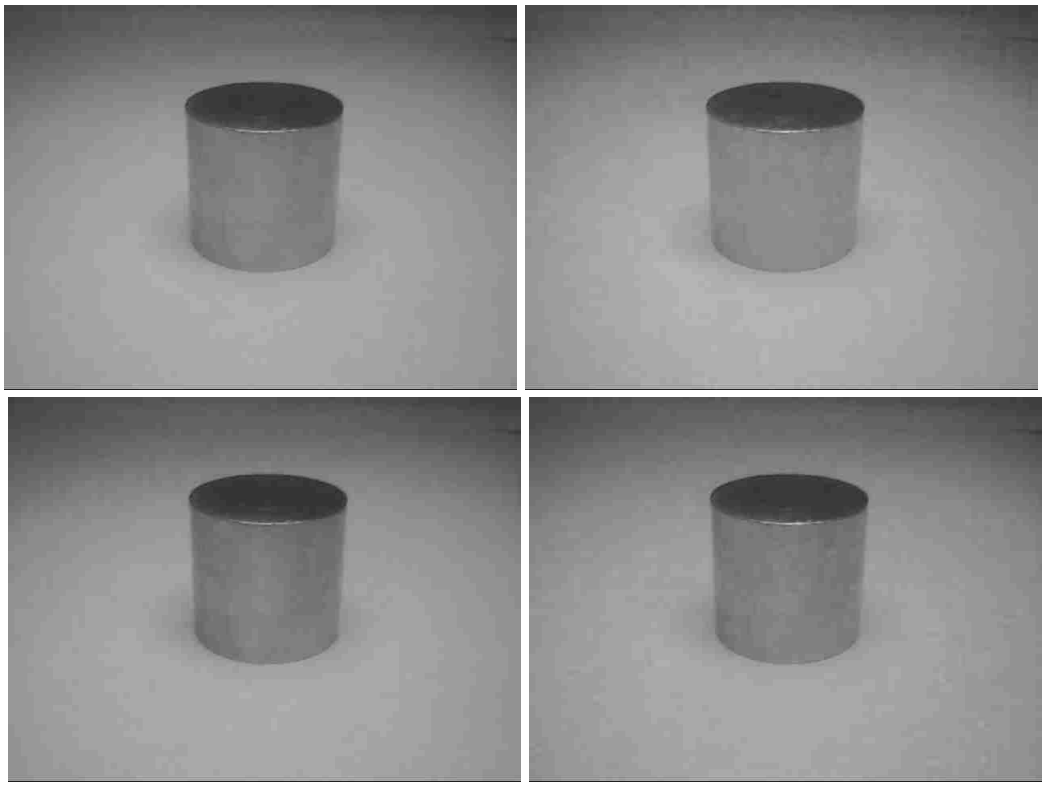

Figure 4.3: Color channels of the cylinder image. (Top: Luma and red channel. Bottom: Green and blue channel.)

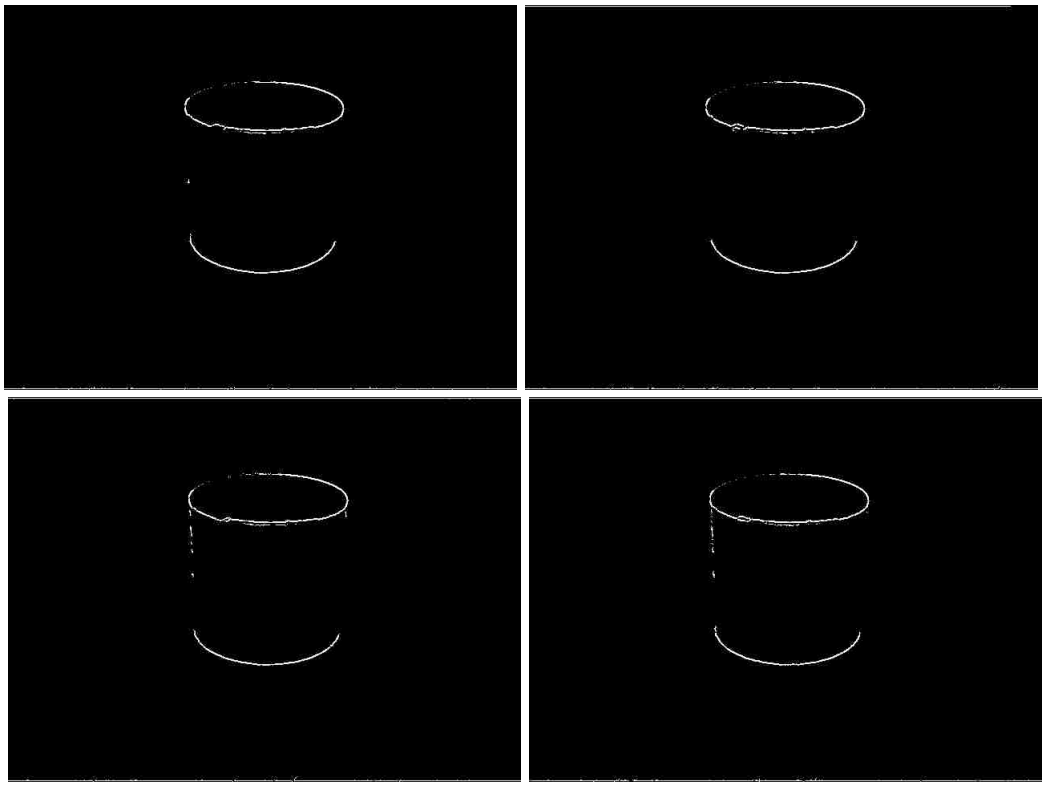

Figure 4.4: Detection results for Thresholds: $T_L = 40$ and $T_H = 90$. (Top: Luma and red channel. Bottom: Green and blue channel.)

blue channels, we can see the left side of the cylinder starting to be outlined by the edge detector while the luma and the red channels do the worst, not indicating any real edge in the locations that the green and blue do. While the difference is subtile, there is a noticeable difference in the output edges of the two algorithms.

4.2 Evaluating Histogram Equalization and Progressive Histogram Equalization

The second major part of this thesis is the histogram equalization algorithm and the progressive histogram equalization algorithm. By applying histogram equalization, we can greatly improve the contrast of the image by allowing the image to use more intensity values. For testing the histogram equalization, I decided to run it against the same cylinder image as we ran the color channel selector against. The main reason the sides do not appear in the other algorithm is that there is very little contrast between the edges of the cylinder and the table it is sitting on. After the Gaussian blur is applied, the little amounts of contrast gained by the color channel selection simply isn't enough to make the edges survive. When non-maximal suppression happens right before hysteresis, some of the gradient values that are needed to ensure hysteresis can make the connections it needs to make fail. If we can create more contrast in the image, we can probably detect the sides of the cylinder without changing the thresholds much or at all.

We decided to run the test with histogram equalization on the luma as well as the green channel. As we saw from the previous section, the green channel gave very good results for this particular image, even better than that of the luma. Because histogram equalization is such a comprehensive algorithm, it is unlikely that the color channel selection will have a substantial role, at least in the case of the cylinder image, on the detection of edges.

The first step will be to extract the color channels for the image. These are shown in Figure 4.3. Using the histogram equalization algorithm, we are able to create images that use the entire range of intensity values. The images in figure 4.5 show what the cylinder looks like after the algorithm has been executed. We run the algorithm on the luma and the green channel. The luma is used because it is the traditional approach. We include the green channel because, as we have seen, we get results that are a little better from the green channel for this image. Doing the green channel as well allows us to see if we can see if this result continues to be true even after applying new procedures such as histogram equalization. Running the rest of Canny's edge detector on the image, using hysteresis thresholds f$T_L = 40$ and $T_H = 90$, we are able to create the edge maps shown in Figure 4.6.

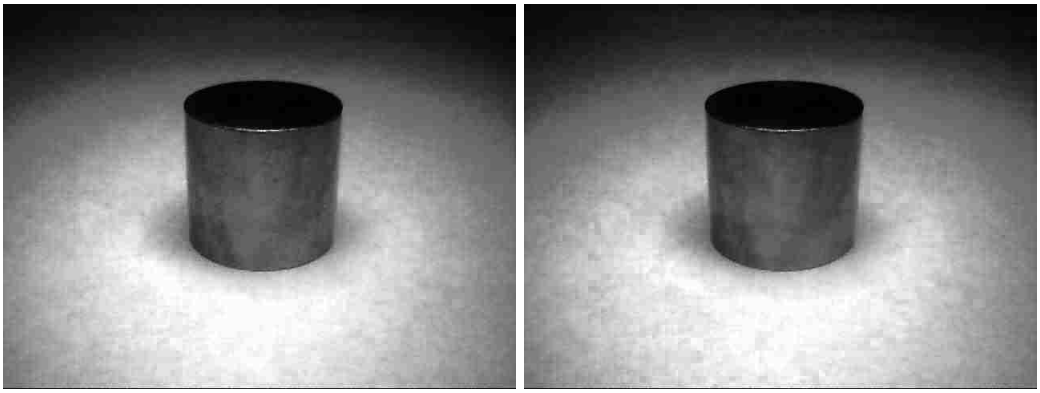

Figure 4.5: The cylinder after histogram equalization. (Left: Luma. Right: Green channel.)

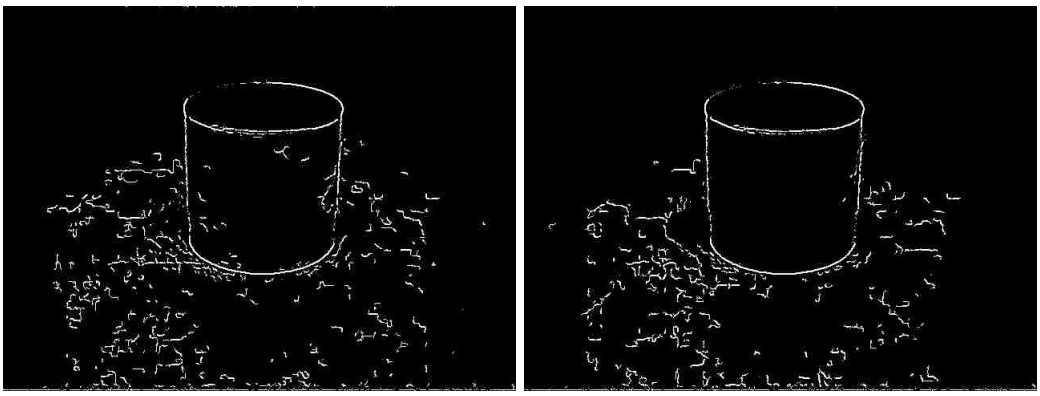

Figure 4.6: Detection results histogram equalization and for thresholds: $T_L = 40$ and $T_H = 90$. (Left: Luma. Right: Green.)

The edges of the object are much more defined when compared to any of the results in figure 4.4, but it should also be obvious that a lot of noise was introduced when the algorithm was executed as well. The area surrounding the cylinder is no longer smooth. These false edges are often created when the gradients created by noise do not fall under the T_L but one of the gradients are high enough to be included by T_H. Some false edges are to be expected and can be dealt with. For example, if there are only a small handful of pixels in a completely disconnected area of the image that are considered edges, there's a good chance they aren't actually edges. Likewise, longer edge lines that don't follow specific geometry we may be expecting are likely not edges for the object either.

When we look at these fake edges though, some of these do not fit the description of acceptable fake edges. Specifically the ones to the left of the cylinder in the luma edge maps. The two horizontal edge lines that appear to be coming from the cylinder could very easily be mistaken for real geometry. The green channel's edge maps also have fake edges that are bad and it would be hard for the program receiving the edges to further do operations on the image as they are currently.

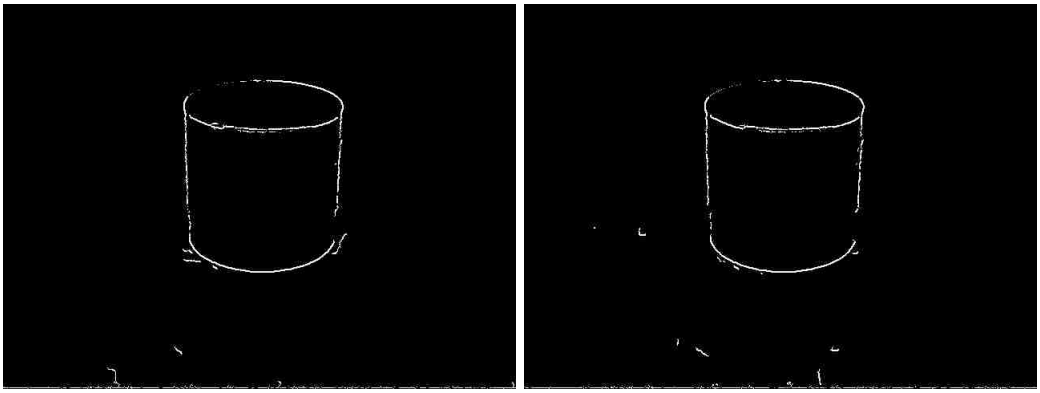

Figure 4.7: Detection results histogram equalization and for thresholds: $T_L = 40$ and $T_H = 95$. (Left: Luma. Right: Green channel)

Clearly, the histogram equalization has defined the edges better by increasing the values of the gradients, but at the cost of adding a lot of noise to the image. There are two tools that Canny's edge detector has to combat the noise introduced by the histogram equalization. The first is to just alter the thresholds during the final step of Canny's edge detector. The low threshold, as explained, is responsible for eliminating weak gradients while the high threshold is responsible for empowering stronger gradients. Gradients equal to or above the high threshold are considered edges while the gradients equal to or below the lower threshold are considered non-edges with the remainder considered "maybes" until the recursive edge trace occurs. By simply increasing the criteria of what it takes to be considered an edge, we can actually eliminate a lot of the false edges. Figure 4.7 shows the result of simply changing the T_H from 90.0 to 95.0.

The other way of eliminating noise is to use the Gaussian blur. While currently the Gaussian is being executed against the image, we are currently only using a single pass of the algorithm with $\sigma = 1.0$. If we run multiple passes of the algorithm, or increase the sigma, we can increase the blurring effect. Of course, this runs the risk of damaging the contrast that was just gained by the equalization, at the same time, it is also a very effective noise handler. We can get an idea of how many times we need to run the Gaussian blur algorithm by looking at the histograms. Figure 4.8 shows the progression of blurring and the effects the blurring process has on the histogram. The first histogram comes directly after the histogram equalization process was executed. Each of the following histograms show one blur level, up to the third level, at the furthest right position of the figure. Each time the blur is performed, the noise of the image and the grainy behavior starts to disappear. This serves as a good visual indicator of how many times the Gaussian needs to be executed to reduce the noise of the image. The result is a smooth histogram that covers the entire $[0, 255]$ range of intensity values.

Figure 4.8: Results of the Guassian on an equalized histogram.

We can now execute the Canny edge detector without adjusting the hysteresis thresholding and we can see that the noise introduced by the histogram equalization is gone. The false edges have been mostly removed, as Figure 4.9 shows, and the overall detection of the cylinder is better than without the histogram equalization for the same threshold values. While this method of using multiple Gaussian blurs after histogram equalization does improve over simply not using histogram equalization at all, for a given threshold, it is likely in general a better solution to adjust the T_H instead, making it unlikely the fake edges out in the background of the image Some of the edges in this result break more than the ones that simply change the thresholding. Changing the thresholding is also more computationally efficient. On the other hand, this solution is much easier for implementation. Choosing good Canny thresholds is a much more difficult task, since it requires modifying two thresholds. You may tweak the high threshold several times but not get better results because the low threshold is too high and removes too many "maybe" edges. Blur levels are an easier thing to quantify because if you still have noise, increase the blur levels until you start harming the detection of the object.

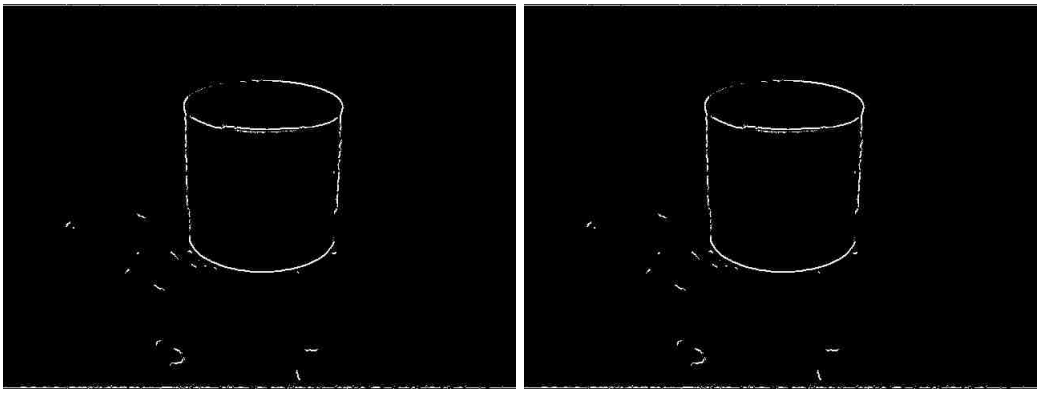

Figure 4.9: Detection results of the equalized image at two different blur levels. (Left: Two blur levels. Right: Three blur levels.)

Finally, another technique we could use is our progressive histogram equalization algorithm. This algorithm will start at an intensity of 40 for the cylinder image and progressively modify the histogram until the histogram covers the maximum 255 range of possible values. The step size is set to 5. The results of using the progressive algorithm is shown in 4.10 for both the luma and the green channels.

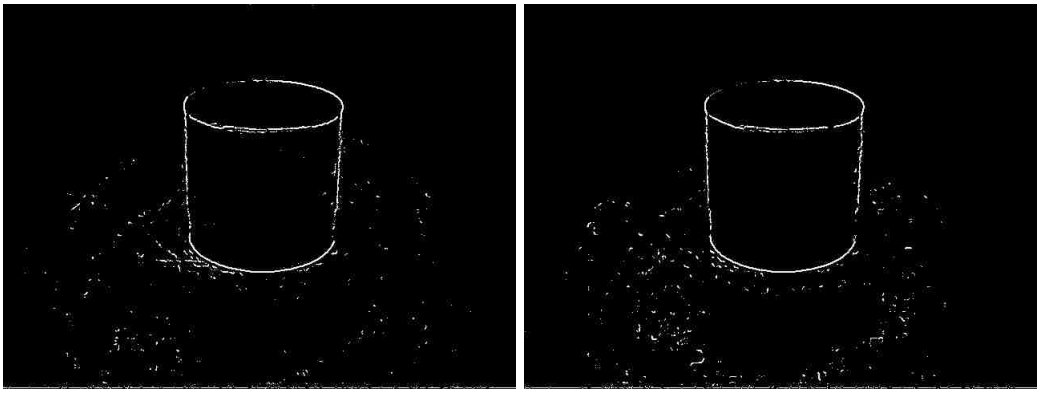

Figure 4.10: Detection results of the progressively equalized image for thresholds: $T_L = 40$ and $T_H = 90$ (Left: Luma. Right: Green channel.)

The progressive histogram equalization algorithm gives much better results than the histogram equalization results from figure 4.6 and the other results from figure 4.4. Even without changing the Canny thresholds or using additional levels of blurring, we can identify the edges of the cylinder clearly. While there are fake edges introduced by the progressive histogram equalization algorithm as well, they are generally smaller and more manageable than the ones introduced by traditional histogram equalization. The long streaks that could have confused a program that used the edge maps as input are generally gone, and most of the fake edges from the noise are small and easy to identify. While doing additional blur levels does give better results, and we could do additional blur levels after progressive histogram equalization as well, we don't really have to in order to identify the object. We also don't really need to modify the Canny thresholds to detect the object better either.

Chapter 5

Conclusion

In this thesis we have demonstrated an edge detector implementation that will automatically select a color channel and perform histogram equalization with the intent to maximize contrast of the object trying to be detected. Using minimal input from the human operator, the program can compare the histograms of what the user has defined as being the general area of pixels for the object, and the general area for pixels for the background. By finding the color channel that has a maximal difference, we are often able to create a stronger contrast which will better represent the edges of the object when the actual detection occurs. Likewise, by using the entire domain of intensities available to the image, histogram equalization greatly improves the overall contrast of the image [GW08]. We can use multiple levels of Gaussian filtering, or modifications of the thresholds to help reduce the noise of the equalization algorithm. Finally, we introduced the progressive histogram equalization algorithm which is able to get good results by better distributing the values of the histogram to begin with, resulting in less and more manageable noise.

Some of these techniques do come at a cost, however. Objects that are primarily one color, but have a small part that is a different color can sometimes be mis-represented in one color channel but would behave normally under the luma grayscale. Likewise, histogram equalization, while it does generally improve the output of the detector by improving the contrast, can also cause harm by introducing noise, and unless sufficient blurring or histogram equalization intensity limit selection is performed, the noise can interfere with the detection of edges. While we are able to resolve the issues in the cylinder examples from our results chapter, this may not be true for other images where increasing the blur levels may cause objects to become harder to detect. The progressive histogram equalization algorithm is generally robust, but there is no guarantee that noise will be reduced by better histogram distribution.

There are many techniques that can further be applied to this implementation of Canny's edge

detector to further improve upon the results. Currently, only the additive color model is being considered in the color channel selection. There are situations where the maximal color channel is very close in values to a second color channel, for example red and blue, and rather than picking red or blue, a better choice may have been magenta. Adding CYMK channels to the selection algorithm could probably help reduce the number of cases where the fallback to the luma is considered the best possible option for the image. Likewise, the algorithm could probably be implemented entirely in color and create a multi-channel Canny edge detector solution. The problem with a multi-channel Canny edge detector is that the progressive and traditional histogram equalization cannot be used against color images in the current implementation. The color channels could experience experience color shift if implemented in full color rather than a single channel. This could be resolved if the image is modeled by a different color space than RGB, specifically one like the hue/saturation/intensity (HSI) model [BK07]. In this scenario, the histogram equalization could be applied to the intensity or the saturation of the image, but the hue can be preserved. The resulting image would then be a full color image with the contrast enhancement benefits of histogram equalization techniques. This would give us the ability to use a full color Canny edge detector implementation and possibly get better results.

Appendix A

Code Listing

```
public void histogramEqualization(int intensity)
        {
            int width = currentImage.GetLength(0);
            int height = currentImage.GetLength(1);
            int channels = currentImage.GetLength(2);
            int[, ,] result = new int[width, height, channels];

            Histogram h = new Histogram(currentImage, intensity);

            // The histogram created a cumulative probability function of the with the current
                image's histogram. We just use it here
            for (int x = 0; x < width; x++)
            {
                for (int y = 0; y < height; y++)
                {
                    for (int currChannel = 0; currChannel < channels; currChannel++)
                    {
                        if (currChannel == RED) {
                            result[x,y, currChannel] = (int)h.rCumulativeProbability[currentImage
                                [x,y,currChannel]];
                        } else if (currChannel == GREEN) {
                            result[x,y, currChannel] = (int)h.gCumulativeProbability[currentImage
                                [x,y,currChannel]];
                        } else if (currChannel == BLUE) {
                            result[x,y, currChannel] = (int)h.bCumulativeProbability[currentImage
                                [x,y,currChannel]];
                        }
                    }
                }
            }

            this.currentImage = result;
        }
```

```
public void calculateCumulativeProbabilityHistograms(int total)
        {
            double t = total;
            for (int i = 0; i < 256; i++)
            {
                if (i == 0)
                {
                    rCumulativeProbability[i] = (rHistogram[i] / t) * max_intensity;
                    gCumulativeProbability[i] = (gHistogram[i] / t) * max_intensity;
                    bCumulativeProbability[i] = (bHistogram[i] / t) * max_intensity;
                }
                else
                {
                    rCumulativeProbability[i] = (rHistogram[i] / t) * max_intensity +
                        rCumulativeProbability[i - 1];
                    gCumulativeProbability[i] = (gHistogram[i] / t) * max_intensity +
                        gCumulativeProbability[i - 1];
                    bCumulativeProbability[i] = (bHistogram[i] / t) * max_intensity +
                        bCumulativeProbability[i - 1];
                }
            }
        }

public void gaussianBlur()
        {

            int width = currentImage.GetLength(0);
            int height = currentImage.GetLength(1);
            int channels = currentImage.GetLength(2);
            int[,,] result = new int[width, height, channels];

            int[,] mask = new int[,] { // mask for gradient
                                       { 1,  4,  7,  4,  1},
                                       { 4, 16, 26, 16,  4},
                                       { 7, 26, 41, 26,  7},
                                       { 4, 16, 26, 16,  4},
                                       { 1,  4,  7,  4,  1}
                                     };

            int sum = 0;

            for (int x = 0; x < width; x++)
            {
                for (int y = 0; y < height; y++)
                {
                    for (int currChannel = 0; currChannel < channels; currChannel++)
                    {
                        sum = 0;
                        // Apply the mask and get the sum
                        for (int i = 0; i < 5; i++)
                        {
```

```
                        for (int j = 0; j < 5; j++)
                        {
                            int temp_x = (x - 2 + i + width) % width;
                            int temp_y = (y - 2 + j + height) % height;
                            sum = sum + (currentImage[temp_x, temp_y, currChannel] * mask[i,
                                j]);
                        }
                    }

                    sum = sum / 273; // divide the sum to get the average
                    result[x, y, currChannel] = Math.Min(Math.Max(sum, 0), 255); // apply the
                         average to the result
                }
            }
        }

        this.currentImage = result;
    }

public void calculateGradients()
    {
        int width = currentImage.GetLength(0);
        int height = currentImage.GetLength(1);
        int[,,] result = new int[width, height, 3];

        for (int x = 0; x < width; x++)
        {
            for (int y = 0; y < height; y++)
            {
                result[x, y, XCOMPONENT] = (currentImage[getIndex(x, -1, width - 1), y,GRAY]
                    - currentImage[getIndex(x, 1, width - 1), y, GRAY]) / 2;
                result[x, y, YCOMPONENT] = (currentImage[x, getIndex(y, 1, height - 1), GRAY]
                     - currentImage[x, getIndex(y, -1, height - 1), GRAY]) / 2;
                result[x, y, MAGNITUDE] = Math.Abs(result[x, y, XCOMPONENT]) + Math.Abs(
                    result[x, y, YCOMPONENT]);
            }
        }

        this.gradientImage = true;
        this.currentImage = result;
    }

public void nonMaximalSuppression()
    {
        int width = currentImage.GetLength(0);
        int height = currentImage.GetLength(1);
        int[,,] result = new int[width, height, 3]; // These aren't color channels. They are
            x comp, y comp and magnitude at this point (requires calc gradients)

        for (int x = 0; x < width; x++)
        {
```

```
for (int y = 0; y < height; y++)
{
    /* we need to determine the x and y values for the pixel we are going to
        compare the pixel against.
     * we do this by looking at the angle of the gradient. Once the angle is
        known, we get 2 sets of xy
     * values. If this pixel is the local maxima for the 3 pixels, then it is an
        edge.
     */

    int x1 = 0, y1 = 0, x2 = 0, y2 = 0; // we use these as indexes into the
        channel values for the comparison

    // if the signs of both are positive, then it is in quadrant 1
    if (currentImage[x, y, 0] >= 0 && currentImage[x, y, 1] >= 0)
    {
        // we look at absolute values now...

        // in quad 1, if y is 0, then the angle is 0 degrees
        if (currentImage[x, y, 1] == 0)
        {
            x1 = getIndex(x, 1, width - 1);
            y1 = getIndex(y, 0, height - 1);
            x2 = getIndex(x, -1, width - 1);
            y2 = getIndex(y, 0, height - 1);
        }
        // if the x is greater than the y (but y != 0), then the angle is between
            0 and 45 degrees
        else if (Math.Abs(currentImage[x, y, 0]) > Math.Abs(currentImage[x, y,
            1]))
        {
            x1 = getIndex(x, 2, width - 1);
            y1 = getIndex(y, -1, height - 1);
            x2 = getIndex(x, -2, width - 1);
            y2 = getIndex(y, 1, height - 1);
        }
        // if they are equal, then the angle is exactly 45 degrees
        else if (Math.Abs(currentImage[x, y, 0]) == Math.Abs(currentImage[x, y,
            1]))
        {
            x1 = getIndex(x, 1, width - 1);
            y1 = getIndex(y, -1, height - 1);
            x2 = getIndex(x, -1, width - 1);
            y2 = getIndex(y, 1, height - 1);
        }
        // if the y is greater than the x (but x != 0), then the angle is between
            45 and 90 degrees
        else if (Math.Abs(currentImage[x, y, 1]) > Math.Abs(currentImage[x, y,
            0]))
        {
            x1 = getIndex(x, 1, width - 1);
```

```
            y1 = getIndex(y, -2, height - 1);
            x2 = getIndex(x, -1, width - 1);
            y2 = getIndex(y, 2, height - 1);
        }
        // else the x was equal to 0, so the angle is 90 degrees
        else
        {
            x1 = getIndex(x, 0, width - 1);
            y1 = getIndex(y, -1, height - 1);
            x2 = getIndex(x, 0, width - 1);
            y2 = getIndex(y, 1, height - 1);
        }
    }
    // if the x is negative, but the y is positive, then it is in quadrant 2
    else if (currentImage[x, y, 0] < 0 && currentImage[x, y, 1] >= 0)
    {
        // we look at absolute values now...

        // in quad 2, if the y is greater than the x, then the angle is between
            90 and 135 degrees
        if (Math.Abs(currentImage[x, y, 1]) > Math.Abs(currentImage[x, y, 0]))
        {
            x1 = getIndex(x, -1, width - 1);
            y1 = getIndex(y, -2, height - 1);
            x2 = getIndex(x, 1, width - 1);
            y2 = getIndex(y, 2, height - 1);
        }
        // if they are equal, then the angle is exactly 135 degrees
        else if (Math.Abs(currentImage[x, y, 0]) == Math.Abs(currentImage[x, y,
            1]))
        {
            x1 = getIndex(x, -1, width - 1);
            y1 = getIndex(y, -1, height - 1);
            x2 = getIndex(x, 1, width - 1);
            y2 = getIndex(y, 1, height - 1);
        }
        // if the x is greater than the y (but y != 0) then the angle is between
            135 and 180 degrees
        else if ((Math.Abs(currentImage[x, y, 0]) > Math.Abs(currentImage[x, y,
            1])) && currentImage[x, y, 1] != 0)
        {
            x1 = getIndex(x, -2, width - 1);
            y1 = getIndex(y, -1, height - 1);
            x2 = getIndex(x, 2, width - 1);
            y2 = getIndex(y, 1, height - 1);
        }
        // else, the y was equal to zero, so the angle is 180 degrees
        else
        {
            x1 = getIndex(x, -1, width - 1);
            y1 = getIndex(y, 0, height - 1);
```

```
            x2 = getIndex(x, 1, width - 1);
            y2 = getIndex(y, 0, height - 1);
        }

    }
    // if the signs of both are negative, then it is in quadrant 3
    else if (currentImage[x, y, 0] < 0 && currentImage[x, y, 1] < 0)
    {
        // we look at absolute values now...

        // in quad 3, if the x is greater than the y, then the angle is between
            180 and 225 degrees
        if (Math.Abs(currentImage[x, y, 0]) > Math.Abs(currentImage[x, y, 1]))
        {
            x1 = getIndex(x, -2, width - 1);
            y1 = getIndex(y, 1, height - 1);
            x2 = getIndex(x, 2, width - 1);
            y2 = getIndex(y, -1, height - 1);
        }
        // if they are equal, then the angle is exactly 225 degrees
        else if (Math.Abs(currentImage[x, y, 0]) == Math.Abs(currentImage[x, y,
            1]))
        {
            x1 = getIndex(x, -1, width - 1);
            y1 = getIndex(y, 1, height - 1);
            x2 = getIndex(x, 1, width - 1);
            y2 = getIndex(y, -1, height - 1);
        }
        // else, the y was greater than the x, so the angle is between 225 and
            275 degrees
        else
        {
            x1 = getIndex(x, -1, width - 1);
            y1 = getIndex(y, 2, height - 1);
            x2 = getIndex(x, 1, width - 1);
            y2 = getIndex(y, -2, height - 1);
        }

    }
    // otherwise, it is in quadrant 4.
    else
    {
        // we look at absolute values now...

        // in quad 4, if the x is equal to 0, then the angle is exactly 275
            degrees
        if (currentImage[x, y, 0] == 0)
        {
            x1 = getIndex(x, 0, width - 1);
            y1 = getIndex(y, -1, height - 1);
            x2 = getIndex(x, 0, width - 1);
```

```
            y2 = getIndex(y, 1, height - 1);
        }
        // if the y is greater than the x (but x != 0) then the angle is between
            275 and 315 degrees
        else if (Math.Abs(currentImage[x, y, 1]) > Math.Abs(currentImage[x, y,
            0]))
        {
            x1 = getIndex(x, -1, width - 1);
            y1 = getIndex(y, 0, height - 1);
            x2 = getIndex(x, 1, width - 1);
            y2 = getIndex(y, 0, height - 1);
        }
        // if they are equal, then the angle is exactly 315 degrees
        else if (Math.Abs(currentImage[x, y, 0]) == Math.Abs(currentImage[x, y,
            1]))
        {
            x1 = getIndex(x, 1, width - 1);
            y1 = getIndex(y, 1, height - 1);
            x2 = getIndex(x, -1, width - 1);
            y2 = getIndex(y, -1, height - 1);
        }
        // else, the x was greater than the y, so the angle is between 315 and 0
        else
        {
            x1 = getIndex(x, 2, width - 1);
            y1 = getIndex(y, 1, height - 1);
            x2 = getIndex(x, -2, width - 1);
            y2 = getIndex(y, -1, height - 1);
        }
    }

    // The indexes for the pixel we are comparing to have been set. If either of
        the two pixels are bigger than the current pixel (x,y), then this is not
         an edge.
    //if (channelValues[x, y] > channelValues[x1, y1] && channelValues[x, y] >
        channelValues[x2, y2] && x1+x2+y1+y2 > 0)
    if ((Math.Abs(currentImage[x, y, 0]) + Math.Abs(currentImage[x, y, 1])) > (
        Math.Abs(currentImage[x1, y1, 0]) + Math.Abs(currentImage[x1, y1, 1]))
        &&
        (Math.Abs(currentImage[x, y, 0]) + Math.Abs(currentImage[x, y, 1])) > (
            Math.Abs(currentImage[x2, y2, 0]) + Math.Abs(currentImage[x2, y2,
            1])) && x1 + x2 + y1 + y2 > 0)
    {
        result[x, y, MAGNITUDE] = Math.Abs(currentImage[x, y, 0]) + Math.Abs(
            currentImage[x, y, 1]);

    }
    else
    {
        result[x, y, MAGNITUDE] = NOTEDGE; // this means that the pixel was less
            than or equal to the neighbor. Suppress it as not being an edge.
```

```
                    }

                } // Ends inner for loop
            } // Ends outer for loop

            this.currentImage = result;
        }

public void hysteresis(double lowThreshold, double highThreshold)
        {
            int width = currentImage.GetLength(0);
            int height = currentImage.GetLength(1);
            int[,,] result = new int[width, height, 1];

            // First we create a histogram of the non-zero pixels
            int[] svHistogram = new int[256];
            // initialize array to zero
            Array.Clear(svHistogram, 0, 256);

            // All non-suppressed pixels will, by default, be 'maybes'
            for (int x = 0; x < width; x++)
            {
                for (int y = 0; y < height; y++)
                {
                    if (currentImage[x, y, MAGNITUDE] != NOTEDGE)
                    {
                        result[x, y, GRAY] = MAYBE;
                    }
                }
            }

            int sum = 0; // sum of total values

            for (int x = 0; x < width; x++)
            {
                for (int y = 0; y < height; y++)
                {
                    // We want to exclude the suppressed values from this calculation
                    if (currentImage[x, y, MAGNITUDE] != NOTEDGE)
                    {
                        svHistogram[currentImage[x, y, MAGNITUDE]]++;
                        sum++;
                    }
                }
            }

            // of the values that survived suppression, the bottom lowThreshold% of values
                 certainly are not edges. We can eliminate them now.
            int highcutoff = (int)(sum * (highThreshold / 100)); // Actual number of high values
                 within highThreshold%
            int lowcutoff = (int)(sum * (lowThreshold / 100)); // Actual number of low values
```

```
    within lowThreshold%
Console.WriteLine("Low Cutoff: " + lowcutoff);
Console.WriteLine("Hig Cutoff: " + highcutoff);
int bottomValues = 0;

int i = 1;
do
{
    bottomValues += svHistogram[i];

    // if the number of values has not reached cutoff, then these are non-edges
    if (bottomValues < lowcutoff)
    {
        for (int x = 0; x < width; x++)
        {
            for (int y = 0; y < height; y++)
            {
                if (currentImage[x, y, MAGNITUDE] == i)
                {
                    result[x, y, GRAY] = NOTEDGE;
                }
            }
        }
    }
    i++;
} while (bottomValues < lowcutoff);

int upperValues = 0;

i = 255; // we are going from top to bottom now
do
{
    upperValues += svHistogram[i];

    // if the number of values has not reached cutoff, then these are edges. Mark
        them with 255
    if (upperValues < highcutoff)
    {
        for (int x = 0; x < width; x++)
        {
            for (int y = 0; y < height; y++)
            {
                if (currentImage[x, y, MAGNITUDE] == i)
                {
                    result[x, y, GRAY] = EDGE;
                }
            }
        }
    }
    i--;
} while (upperValues < highcutoff);
```

```
    // Everything that isn't either 0 or 255 at this point should be a 'maybe' (set to
        128).
    for (int x = 0; x < width; x++)
    {
        for (int y = 0; y < height; y++)
        {
            recursiveEdgeTrace(result, x, y);
        }
    }

    // After recursive edge trace, if we did it, we can eliminate remaining maybes
    for (int x = 0; x < width; x++)
    {
        for (int y = 0; y < height; y++)
        {
            if (result[x, y, GRAY] == MAYBE)
            {
                result[x, y, GRAY] = NOTEDGE;
            }
        }
    }

    this.gradientImage = false;
    this.currentImage = result;
}


/*
 * Recursively trace edges. If the point passed in is an edge (255) then check all the
     neighbors for 'maybes' (128). If a
 * 'maybe' is found switch it to an edge, and recursively trace its edges. Do nothing if
     the neighbors are edges (255)
 * or not edges (0)
 */
private void recursiveEdgeTrace(int[,,] image, int x, int y)
{
    int width = image.GetLength(0) - 1;
    int height = image.GetLength(1) - 1;

    if (image[x, y, GRAY] == EDGE)
    {
        // check all neighbors
        for (int i = -1; i <= 1; i++)
        {
            for (int j = 1; j >= -1; j--)
            {
                if (image[getIndex(x, i, width), getIndex(y, j, height), GRAY] == MAYBE)
                { // if the neighbor of the edge is a maybe
                    image[getIndex(x, i, width), getIndex(y, j, height), GRAY] = EDGE; //
                        then that neighbor is going to be set to an edge
```

```
                    recursiveEdgeTrace(image, getIndex(x, i, width), getIndex(y, j,
                        height)); // and we should do a recursive edge trace on it too
                }
            }
        }
    }
}
```

Bibliography

[Bas02] Mitra Basu. Gaussian-based edge-detection methods-a survey. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 32(3):252–260, 2002.

[BB82] Dana H. Ballard and Christopher M. Brown. *Computer Vision*. Prentice-Hall, Inc, 1982.

[BK07] Nikoletta Bassiou and Constantine Kotropoulos. Color image histogram equalization by absolute discounting back-off. *Computer Vision and Image Understanding*, 107(1):108–122, 2007.

[Can86] John Canny. A Computational Approach to Edge Detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, PAMI-8:679 –698, 1986.

[GW08] Rafael C. Gonzales and Richard E. Woods. *Digital Image Processing Third Edition*. Prentice-Hall, Inc, 2008.

[NPKJ08] Thuy Tuong Nguyen, Xuan Dai Pham, Dongkyun Kim, and Jae Wook Jeon. Automatic exposure compensation for line detection applications. In *Multisensor Fusion and Integration for Intelligent Systems, 2008. MFI 2008. IEEE International Conference on*, pages 68 –73, aug. 2008.

[Ots79] Nobuyuki Otsu. A threshold selection method from gray-level histograms. *IEEE Transactions on Systems, Man and Cybernetics*, 9(1):62–66, 1979.

[Pic04] Massimo Piccardi. Background subtraction techniques: a review. In *Systems, man and cybernetics, 2004 IEEE international conference on*, volume 4, pages 3099–3104. IEEE, 2004.

[Pit00] Ioannis Pitas. *Digital Image Processing Algorithms and Applications*. John Wiley & Sons, Inc, 2000.

[PV00] Konstantinos N. Plataniotis and Anastasios N. Venetsanopoulos. *Color image processing and applications*. Springer, 2000.

[Rus10] John C. Russ. *The Image Processing Handbook*. CRC Press, 2010.

[SB91] Michael J Swain and Dana H Ballard. Color indexing. *International journal of computer vision*, 7(1):11–32, 1991.

[Sob78] Irvin Sobel. Neighborhood coding of binary images for fast contour following and general binary array processing. *Computer Graphics and Image Processing*, 8(1):127–135, 1978.

[SSAaS10] N.V. Kalyankar Salem Saleh Al-amri and Khamitkar S.D. Image segmentation by using edge detection. *International journal on computer science and engineering*, 2(03):804–807, 2010.

[Sta00] J Alex Stark. Adaptive image contrast enhancement using generalizations of histogram equalization. *Image Processing, IEEE Transactions on*, 9(5):889–896, 2000.

[XJL06] Wei Xu, M. Jenkin, and Y. Lesperance. A multi-channel algorithm for edge detection under varying lighting. In *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, volume 2, pages 1885 – 1892, 2006.

Vita

Graduate College
University of Nevada, Las Vegas

Justin L. Baker

Degrees:

Bachelor of Science in Computer Science 2010
University of Nevada Las Vegas

Object Detection Using Contrast Enhancement and Dynamic Noise Reduction

Thesis Examination Committee:

Chairperson, Dr. Evangelos A. Yfantis, Ph.D.
Committee Member, Dr. Jan B. Pedersen, Ph.D.
Committee Member, Dr. John T. Minor, Ph.D.
Graduate Faculty Representative, Alexander Paz Ph.D.