ABSTRACT

A METHOD OF TRACKING EYE MOTION FOR HUMAN-COMPUTER INTERFACE

by
Martin W. Maranski

It would be of value to create another device for interfacing with a computer besides the commonly used handheld devices; one option is to determine where the user is looking on the screen. The major obstacle to overcome in this task is to be able to reliably track the motion of the iris and pupil in the eye. A method to do this is revealed here, using a variation of the Hough transform to detect circular patterns in a close shot of the eye. The algorithm was developed in Matlab and analyzed over a series of 20 sets of 6 frames each; this analysis reveals that it is capable of producing 88.33 percent accuracy under normal conditions. The other obstacles to be overcome are also specified, and possible solutions outlined, with suggestions as to hardware implementation and applications for the handicapped and for a convenient hands-free interface method.

A METHOD OF TRACKING EYE MOTION FOR HUMAN-COMPUTER INTERFACE

By
Martin W. Maranski

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Engineering

Department of Electrical and Computer Engineering

May 2000

APPROVAL PAGE

A METHOD OF TRACKING EYE MOTION FOR HUMAN-COMPUTER INTERFACE

Martin W. Maranski

[illegible]

Dr. Raashid Malik, Graduate Advisor — Date
Visiting Professor of Electrical and Computer Engineering, NJIT

Dr. Yun-Qing Shi, Committee Member — Date
Associate Professor of Electrical and Computer Engineering, NJIT

[illegible]

Dr. Frank Shih, Committee Member — Date
Professor of Computer and Information Science, NJIT

BIOGRAPHICAL SKETCH

Author:	Martin W. Maranski
Degree:	Master of Science in Computer Engineering
Date:	May 2000

Undergraduate and Graduate Education:

- Master of Science in Computer Engineering,
  New Jersey Institute of Technology, Newark, NJ, 2000
- Bachelor of Science in Computer Engineering,
  New Jersey Institute of Technology, Newark, NJ, 1999

Major:	Computer Engineering

To my family for getting me here,

and to my friends for pushing me beyond

ACKNOWLEDGMENT

I would like to express my appreciation to Dr. Raashid Malik, my research supervisor, who helped me find the right track in this project, as well as assisting me along my path in my degree in other ways. Special thanks are also given to Dr. Yun-Qing Shi and Dr. Frank Shih for participating in my committee.

I would also like to give thanks to Ms. Brenda Walker and Dean Ronald S. Kane for their help in the final days of this endeavor, and to my fellow graduate students in the Computer Engineering field.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Output and input govern the world of computers. In general, output devices include printers, speakers, and monitors; the input device takes many forms, but the majority of them involve manually moving one item relative to another. There are other methods, of course, such as voice recognition; but these, while useful, do not accomplish quite the same goal as moving a mouse or joystick. In order to move the pointer across the screen, something else has to move across something else, whether it be the mouse moving across the pad or the stylus moving across the tablet. But what about the iris of the eye moving across its surface? What about a where-you-look-is-where-you-point interface? In order to accomplish this goal, the input device must have the means to track the motion of the eye. This goal is worth pursuing, not only for the purposes of creating a convenient way to interface with the computer, but more importantly to give more computer access to the handicapped or injured who cannot operate any of the current interface devices.

1.1 Objective

The purpose of this thesis is to discover a reliable method of tracking the motion of the iris in a sequence of frames showing an eye. It is understood that tracking the iris is only one step in the attainment of the goal stated above, but it is the key step. Specific information will be given regarding a method to track the iris and its effectiveness, and more general information will be provided regarding steps leading to and following this.

1.1.1 Approach

In order to accomplish the ultimate goal of transforming eye motion into motion of a pointer on a computer screen, there are a number of things that must be done. First, the position of the eye must be captured with a camera or other imaging device. Second, the image must be manipulated with various filters to help the process of analyzing it. Third, the information about the eye's position must be extracted from the still images and analyzed. This is the step that will receive the most attention in the sections to follow. Fourth, the position of the pointer must be inferred from the data gathered in the third step. And fifth, this process must be performed fast enough to make this a useful pointing device. As this is more of a proof of concept than the invention of a new pointing device, this fifth part is left open to the assumption that the processes described can be implemented in specialized hardware for the purposes of increasing the processing speed. Some possible methods for implementing it will be discussed, however. Each of the other steps will be described, along with simplifying assumptions and potential problems, in the chapters that follow.

1.2 Background Information

There has been a good deal of research into image enhancement, manipulation, analysis, and interpretation already. The methods used here are not new in themselves; their application to this problem, however, has not turned up in a cursory literature search. New developments are made constantly in improving the accuracy of the feature extraction methods used here [13]. Also, though all the testing done for these experiments was done in Matlab for ease of use, there has been extensive use of other languages for these purposes; functions to perform some of the basic manipulations described here can be found written in C, which in an actual implementation might be more easily assembled for use on which ever hardware platform would be used [11]. More complicated functions could be easily written or derived from basic ones already provided.

The practice of using a passive preprocessing stage followed by an active feature extraction stage is a tried and true method of image processing [1]. Many other methods of performing each step exist; the arrangement described here was chosen primarily for their simplicity and direct applicability to the specified problem. As an example, the edge detection methods used here could have been replaced by others using the frequency domain rather than the gradient method; however, these were not used because of the additional steps and processing needed to switch back and forth between the two domains.

Another approach that could have been explored is the use of artificial intelligence technology to analyze the images. Bayesian or neural networks could be trained to recognize certain features over time, but this is a much more complicated process than that described here [9]. An approach using a hybrid of both the transform methods and artificial intelligence technology is suggested in a later section, using the transforms to gather the data and a neural net to interpret it.

In the area of hardware implementation, there has been much work in massively parallel processing and DSPs which could be used for an efficient implementation of this technology. One area which draws some attention, however, is the possibility of using optical processors to perform the manipulations. These provide the capability of both performing the Hough transform and housing an artificial neural network to interpret the results [3]. Exciting new technologies with potential for use in this area are being developed on a daily basis, so it is possible that this technology could be seen at any time.

CHAPTER 2

OBTAINING THE IMAGE

The first step towards analyzing an image or frame is to acquire it. In general, there are not many options here; a camera and a capture board are needed, or a camera that can interface directly with the computer. But specifically, there are many options; see [4] for a detailed analysis. Setting aside the technical details, there are other questions. Should the camera work in the visible spectrum? Using ultraviolet or infrared might yield better results.

2.1 Camera and Capture Board

In an ideal situation, a high-resolution capture device would be used, dynamically positioned in such a way as to continually monitor one or both eyes alone, perhaps even another spectrum besides the visible light range. For the purposes of this experiment, however, the capture device used is a simple CCD color camera with a maximum resolution of 640x480 pixels, mounted near a monitor. Focus and zoom are manual, with a minimum focal length somewhat inferior to the true requirements of this problem. On the computer end of the input, a DBS DFG/LC1 video capture card was selected. This device offers all the features required by the application, particularly an analog input for the camera and a relatively simple IDE programming interface and libraries provided by the manufacturer. It has a number of additional features, such as DirectX buffering and overlay capabilities, which were not used but could possibly be of use in future incarnations of this project. For ease of testing purposes, the slower, memory-direct capture method was used for more direct importation to Matlab. A summary description of the capture board, along with relevant functions from the provided libraries is included in Appendix A. Procedures written in C++ and Matlab to interface with the camera are presented with other custom functions in Appendix B. These procedures are not complicated; they merely make use of the provided libraries to prepare and shut down the camera, as well as capture images into memory. With the function descriptions of Appendix A and the code for the custom procedures the methods for doing so should be self-explanatory.

2.1.1 Finding the Eyes

In the scope of this investigation it was assumed that the position of the camera relative to the target's head would remain constant. This is not an unreasonable assumption as two very likely applications of this technology would involve the target being stationary due to injury or wearing a lightweight headset containing the camera. For this reason the general location of the eyes in the captured picture was ignored. In the cases presented, the image was manually cropped to a reasonable area both for improved speed and for focus reasons; the manual focus on the camera would not allow it to be placed any closer or zoom in any further towards the target. Cropping was not purposely done in such a way as to throw off the algorithm's results one way or another. For the single test image used the cropping was done to best display the eye; each subsequent set of frames used for statistical analysis was cut to a specified size, from coordinates visually selected from the first frame in the set.

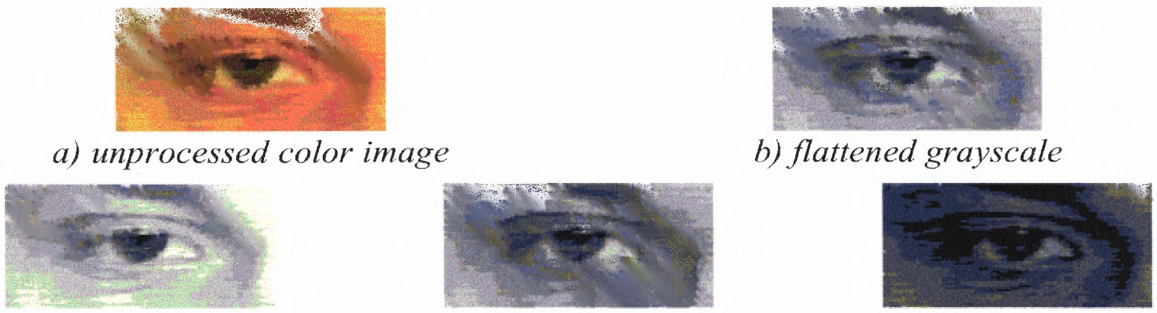
CHAPTER 3

IMAGE PREPROCESSING

Once the image of the eye has been captured and ported to Matlab for processing, the true work of the algorithm begins. There are a number of different processing techniques that can be applied to the image prior to attempting the feature extraction essential to the algorithm. Several of these are examined here and discussed with regards to their relative values to the test images to which the algorithm was applied. While this stage of the algorithm is not particularly processor-intensive, it is nevertheless important.

3.1 Color filtering

The first available method of simplifying the image for feature extraction is the use of color filtering. This is perhaps the simplest method, but can possibly yield useful results. Figure 3.1 shows the effects of color filtering on the first test image.



a) unprocessed color image *b) flattened grayscale*

c) red layer only *d) green layer only* *e) blue layer only*

Figure 3.1: Color layers in the image.

Color filtering is achieved by simply separating the color planes in Matlab's RGB image format, which the capture function produces. Removing hue and saturation values from the image information using the Matlab function `rgb2gray` flattens the three layers to get the gray-scale version of the image. As can be seen from the figure, some color layers give better definition than others; an attempt was made using this to locate the general area to be processed by looking for the whites of the eye, but was unsuccessful.

3.2 Edge Detection

The first major step towards feature extraction is the use of edge detection to eliminate useless information from the image. Unfortunately, some algorithms eliminate too much while others leave behind too much. Therefore, selecting the correct edge detection algorithm is essential to producing good final results.

Most simple edge-detectors work by applying a small filtering matrix to a gray-level image, which is meant to approximate the 2-dimensional derivative, or gradient, of the image [2]. In general, this is the way they are derived. First, we want to approximate the magnitude of the gradient of the picture, or

$$|\nabla \mathbf{f}| = \left[\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2\right]^{\frac{1}{2}} \tag{3.1}$$

where f(x,y) is the 2D image function. The simplest way to do this is by just taking the difference between pixels in the surrounding areas, as in figure 3.2a, hence the use of a small filter. Using that assumption, the above equation can be simplified to the much more manageable

$$\nabla f \approx \left[(z_5 - z_8)^2 + (z_5 - z_6)^2\right]^{\frac{1}{2}} \tag{3.2}$$

But since the filter is to be applied to each pixel in the image, having to constantly take the square and square root of values in the image is not acceptable if the processing is to be reasonably fast. Therefore, another approximation can be made.

$$\nabla f \approx |z_5 - z_8| + |z_5 - z_6| \tag{3.3}$$

This is now much more reasonable in terms of computational speed. This is not the only way of doing things, of course; the next section deals with three standard sets of operators that are often used as approximations to the gradient.

3.2.1 Roberts, Prewitt, and Sobel Operators

The Roberts operators are a set of similar 2x2 filters that, when used in conjunction, can produce fairly good results. The only difference between the above equations and those for the Roberts operators is that the operators use a cross-gradient, finding $z_5 - z_9$ and $z_6 - z_8$ instead [8].

However, even-sized filters are awkward; it is much easier to use an odd-sized filter so that it has a center pixel which is being affected. A 3x3 extension to the above approximation to the gradient, centered on z_5, is

$$\nabla f \approx |(z_7 + z_8 + z_9) - (z_1 + z_2 + z_3)| + |(z_3 + z_6 + z_9) - (z_1 + z_4 + z_7)| \tag{3.4}$$

z_1	z_2	z_3
z_4	z_5	z_6
z_7	z_8	z_9

a) 3x3 region of the image to be processed...

1	0
0	-1

0	1
-1	0

b) 2x2 Roberts operator

-1	-1	-1
0	0	0
1	1	1

-1	0	1
-1	0	1
-1	0	1

c) Prewitt operator, for horizontal and vertical edges

-1	-2	-1
0	0	0
1	2	1

-1	0	1
-2	0	2
-1	0	1

d) Sobel operator, for horizontal and vertical edges

Figure 3.2: Different types of edge detection filters [6].

These are known as the Prewitt operators, and the filters can be seen in figure 3.2c. The Sobel operators are similar to the Prewitt ones, differing in that they try to approximate a slightly different pattern, placing more emphasis on 4-connected neighbors [6].

As a test of effectiveness, these sets were applied to the test image shown above. The results are shown in figure 3.3.

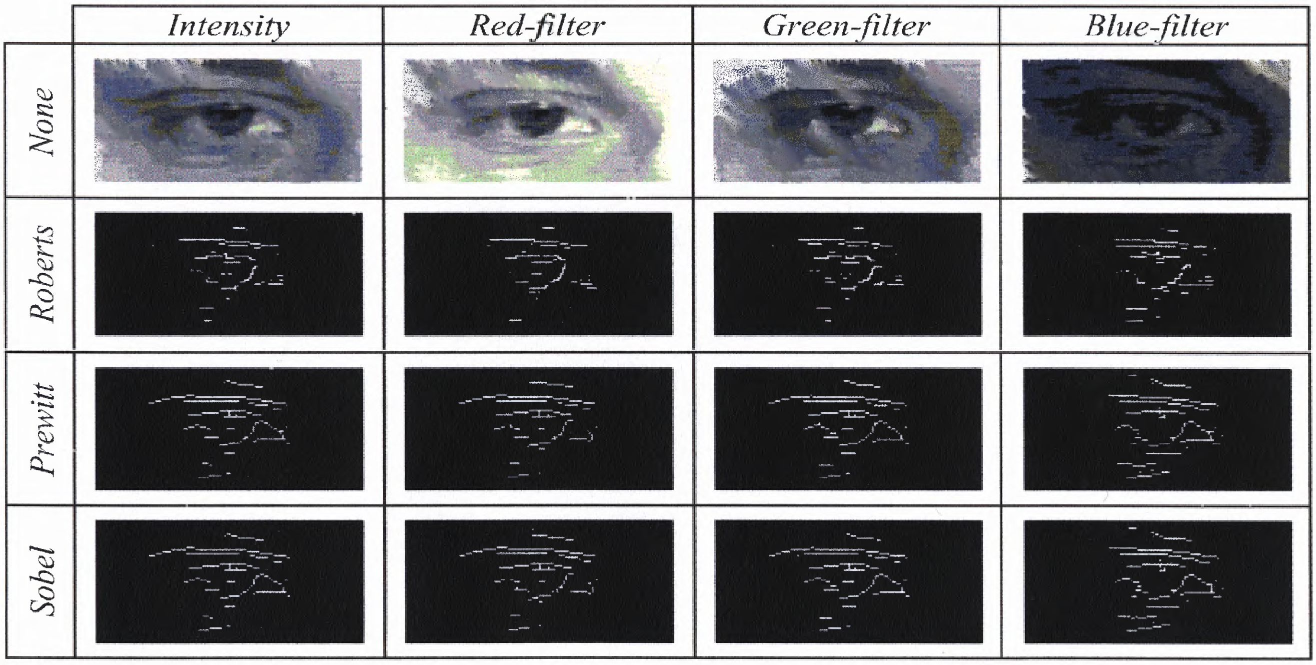


Figure 3.3: Various edge-detection algorithms applied to the test image.

3.2.2 Use of Vertical Filters

It can be fairly easily seen that the resulting binary images are a bit too confused for any practical use. Luckily, there are alternatives. Rather than applying the full operators, it is possible to apply only one from each set, as shown in figure 3.4. The reason for the improved appearance is that most of the clutter in the above images is from various horizontal edges. By using only half of the Sobel or Prewitt operators, only horizontal or vertical edges are detected. This results in a pair of images for each set of operators, one with the horizontal clutter and little useful information, and the other showing quite clearly the outline of the iris and eyelid, which are useful to further steps.

The blue-filtered version is obviously too noisy to be of any real value, but the other three show promise. Also, in general, the vertical Prewitt operator seems to yield better results than the vertical Sobel operator. Therefore, future operations will be limited, for the most part, to using only the vertical Prewitt operator with the green layer of the image.

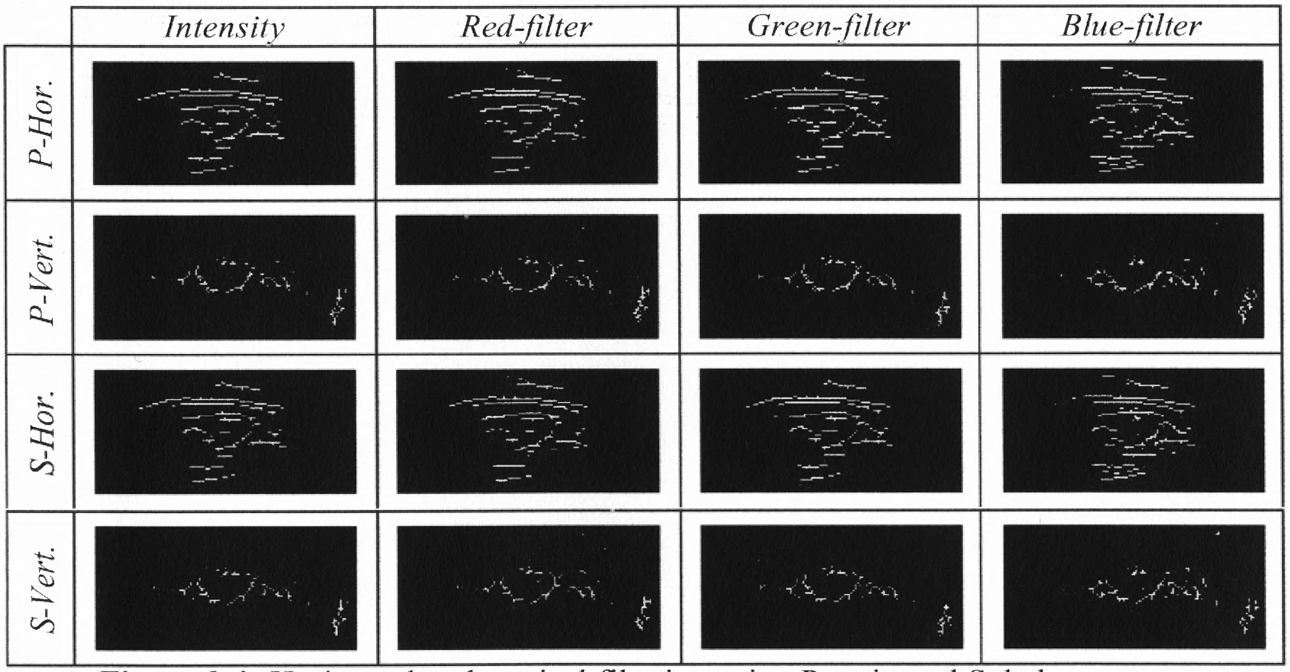


Figure 3.4: Horizontal and vertical filtering using Prewitt and Sobel operators.

3.3 Other Manipulations

There are other things that can be done to further enhance the visibility of the contours in these images, such as dilation and thinning to yield a skeletonized version of the above images. While these manipulations are of some use at times, in this particular situation the increase in quality of the output information does not merit the time spent on the operations. Also, some of the enhancements might possibly even hinder the use of the

Hough transform during feature extraction due to the thinning not resulting in circular forms. Though some testing was done in this area, it was discarded for that reason.

CHAPTER 4

FEATURE EXTRACTION

The key to properly following the motion of the eye is to identify certain features of the eye and make use of their coordinates in the image space. This is, of course, the most complicated part of the process. The image is no longer being merely filtered to make certain information more prominent, but rather being actively searched for patterns that fit the specified requirements; this is, of course, the next logical step after edge detection [12]. This process is by nature time-consuming, and is the part of the algorithm for which hardware acceleration is most needed. There are a number of different methods that can be used for feature extraction. The first method considered was object labeling, which searches through a binary image for connected regions and assigns a label to each. This would be more useful in a stricter environment, but in this situation there are too many irregularities in the image. Lighting problems, especially, contribute to the loss of information about separate objects. For this reason, a more geometric approach was taken, the use of the Hough transform.

4.1 The Hough Transform

The Hough transform is a relatively easy concept to grasp. It starts by taking the geometrical equation of the feature to be found and then inverts it so that x and y are constants in the equation. In this way an inverse function space can be found, and intersections in this space reveal the constants associated with the equation of the features discovered in the original image. A generalized version of this could hypothetically be the way in which the brain interprets what it sees [10].

4.1.1 The Slope-Intercept Model

Perhaps the easiest form, the slope-intercept Hough transform is a transformation from the (x,y) coordinate system to an (a,b) system. Basically, given that the standard slope-intercept equation of a line is given by:

$$y = a \cdot x + b \qquad (4.1)$$

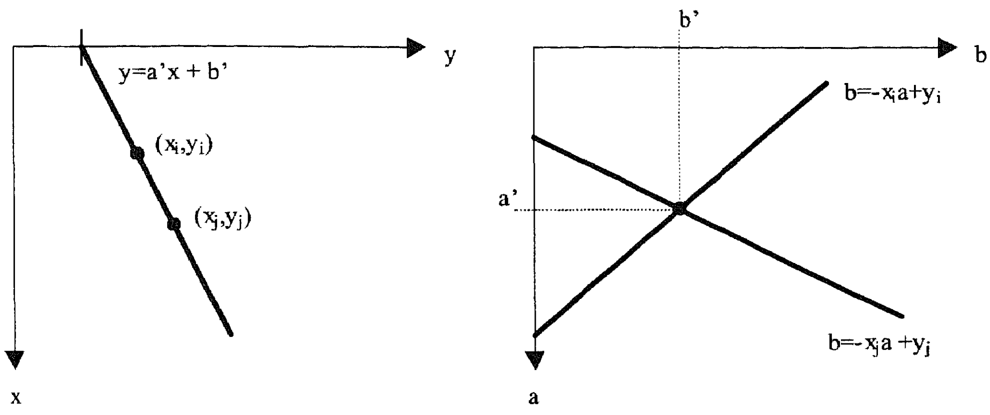
the basic Hough transform is just (4.1) rewritten as

$$b = -x \cdot a + y \qquad (4.2)$$

In this way, the Hough transform creates a second image, an array of accumulator cells, in which each point on the original image becomes a line. The lines are added to the transform one at a time, and anytime there is an overlap the value of that cell is incremented. By looking at the values of the cells of the transform we can determine through which points the most lines cross; the a and b values of these points are the slopes and intercepts of the lines detected by the transforms. This approach is shown in figure 4.1 [6].

4.1.2 The Normal Model

Unfortunately, it is not quite that simple. Both the slope and intercept of the line approach infinity as the line approaches the vertical. The way this is commonly solved is by using the normal representation of a line, which gives the shortest distance from the



a) image space *b) parameter space*

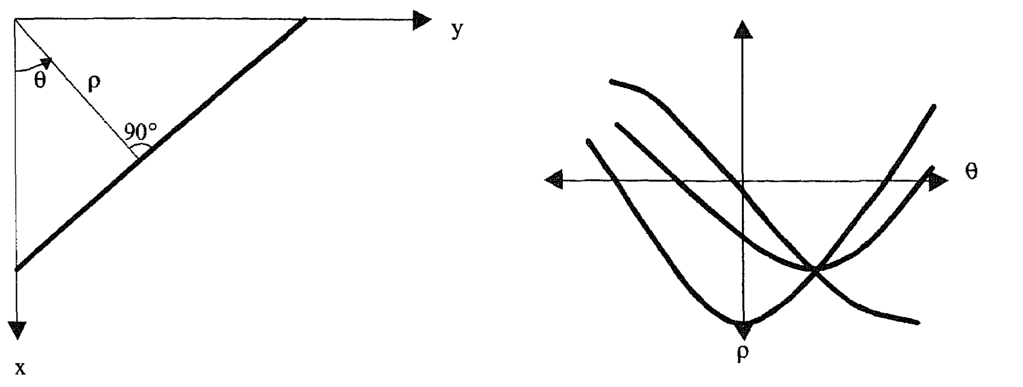
Figure 4.1: xy plane and parameter space for Hough transform.

origin to the line and the angle at which this distance is measured. The standard normal representation is given in (4.3).

$$x \cdot \cos\theta + y \cdot \sin\theta = \rho \qquad (4.3)$$

This maintains the advantage of only needing a two-dimensional Hough transform, but does not tend towards infinity at any time. By simply stepping through values of θ from $-\pi$ to π a number of sinusoidal curves can be derived in the $\rho\theta$ plane. This can be seen roughly in figure 4.2b. Once again, the accumulator cell with the highest value indicates the appropriate ρ and θ for the detected line. If multiple lines are to be found, a threshold can be established in the Hough transform to allow any feature with more than a certain number of points comprising its shape to be detected [14].

Due to the digital nature of the transform, some allowance must be given for rounding errors resulting in cells not reaching as high a value as they should, and therefore a feature not being detected. There are several ways to prevent this; one is to simply apply



a) image space *b) parameter space from 3 points*
Figure 4.2: The normal model Hough transform.

an averaging blur filter on the transform as if it were an image. Another method when rounding the transform's values to fit them into cells is to round both up and down to cover a width of two cells with each point.

This shows that the Hough transform is useful in detecting lines in an image, a very useful function when dealing with assembling mechanical parts, following lines on a road, or other applications which often provide many hard, straight edges. However, in order to track the motion of the human eye, a more organic shape is needed.

4.2 The Circular Hough Transform

The most logical geometric figure to look for in the eye is a circle. Both the iris and pupil are roughly circular and shift when the eye swivels. Also, a circle is typically specified by (4.4), where a and b are the center of the circle and r is the radius.

$$(x-a)^2+(y-b)^2=r^2 \qquad (4.4)$$

The fact that the equation of a circle includes the center is important, as the center of the iris or pupil is what the feature extraction step is meant to find.

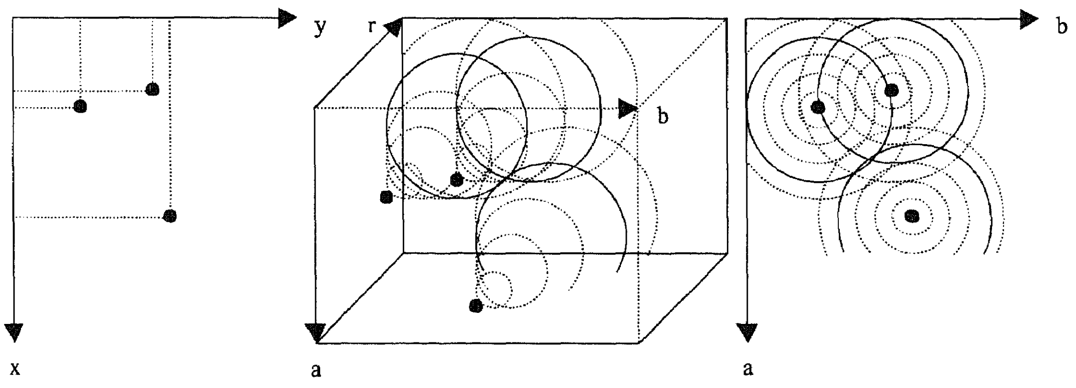
The first step in creating a circular Hough transform is, as in previous cases, to invert the equation to make x and y constants. This is fairly easy with the equation of a circle, as the (x-a) and (y-b) terms are being squared so sign doesn't matter. The resulting equation is shown in (4.5).

$$(a-x)^2+(b-y)^2=r^2 \qquad (4.5)$$

It is easy to see from this equation that the Hough transform will result in intersecting circles, but this is not an easy equation to use in an algorithm. There are three variables, for one thing; in order to use this equation we would need to step through each variable one at a time to fill the three-dimensional array for the transform. Instead, using the fact that the transform itself takes the form of circles, it is best to alter the equation [7]:

$$\begin{aligned} a &= x + r\cdot\cos\theta \\ b &= y + r\cdot\sin\theta \\ \theta &= 0..2\pi \end{aligned} \qquad (4.6)$$

Using θ as a parameter, we can step through it from 0 to 2π, and through r from 0 to its maximum. The resulting transform will be a three-dimensional array containing cones,



a) image *b) 3D view of Hough transform* *a) collapsed view*
Figure 4.3: Circular Hough transform in 2D and 3D. Note intersection point.

which at the r=0 plane each originate with its point on one of the initial pixels from the image. The intersections of the cones show where circles can be placed that touch the various points.

As a test of this theory, a short routine was written in Matlab to discover the accuracy of the transform. Code for the final test routine can be found in Appendix B. The results were good. For two points and a threshold of 2, a line of possible circle centers bisected the line between the two points. Three points with a threshold of 3 yields a good match. For four points, however, rounding problems were encountered. These were solved by rounding both up and down in the transform space, as is seen in the code provided in Appendix B. After this, the transform functioned properly. Results from the test routine can be seen in figure 4.4.

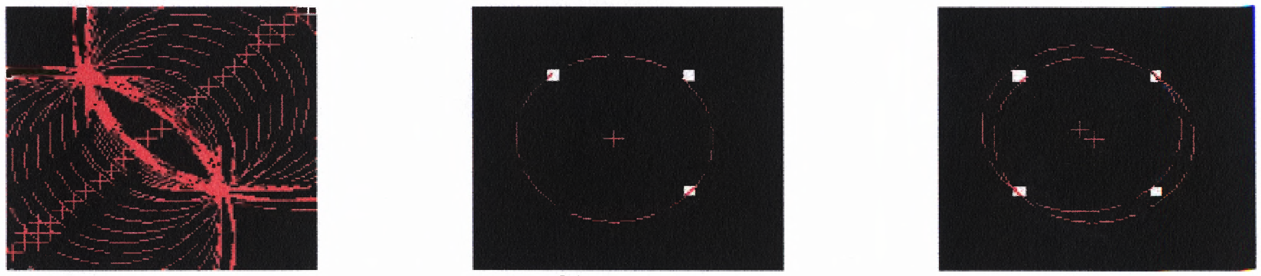
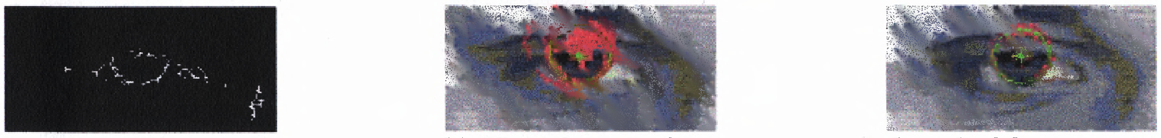

a) 2-point *b) 3-point* *c) 4-point (modified code)*

Figure 4.4: Results of Hough transform test on 2, 3, and 4 points.

Of course, the fact that the Hough transform for a circle is a three-dimensional causes some difficulties with the speed of the process; an even more accurate version could be used which would detect ellipses rather than circles, but this would require an array with an even higher dimensionality [13]. There are, however, a number of methods to decrease the processing time of the circular function defined here, which will be covered in a later section.

4.3 Application to the Eye Tracking Problem

These tests show that the Hough transform is fully capable of fitting a circle to pixels in an image. But in order to analyze properly the motion of the human eye, the algorithm needed to be tested on a more complex image than just a few pixels. To test it further, the transform was run on the same test image of an eye used previously. The results looked promising; although with a threshold of 20 multiple circles were found, the majority of them were clustered around the location of the iris. Figure 4.5b shows this clustering; the green circle is the average of those found. Next, the range of radii over which the procedure searched was reduced in an attempt to cut down on unnecessary data. By reducing the range from a 0-20 pixel radius down to a 10-20 pixel radius, the processing time is reduced and potential erroneous data is removed. The results of this are not shown in the figure, as there is no real improvement in clustering. This is because the maximum radius was not reduced and circles smaller than $10/\pi$ cannot be detected with a threshold of 20 even if all points on the circle are shown. Another method of improving the output of the algorithm is to raise the threshold to the maximum encountered in the transform. This was done, and gives a definite improvement over the previous tests. The results of this are shown in figure 4.5c.


a) edges detected *b) Hough transform* *c) threshold at maximum*
Figure 4.5: Test image processed with Hough transform

But this is not enough to claim to successfully track the motion of the eye. The next step in the development of the method was to capture a sequence of images with the eye in different positions to see if the algorithm could successfully report the position of the iris with a reasonable margin of error. The results are good given certain conditions. If the camera angle is correct and the range of motion required by the eye is not too great, the method will deliver results most of the time. However, if the angle is off, or if too much of the iris is obscured by the eyelid, the results vary wildly. Test results yielded a fairly good accuracy if the camera was aligned nearly perpendicular to the eye. Figure 4.6 shows an excellent test set, with the camera located approximately two feet from the target, offset by about half a foot. Both right and left eyes are included for comparison of accuracy at different viewing angles; note that the algorithm can also be confused by extraneous contours from the background and other sources.

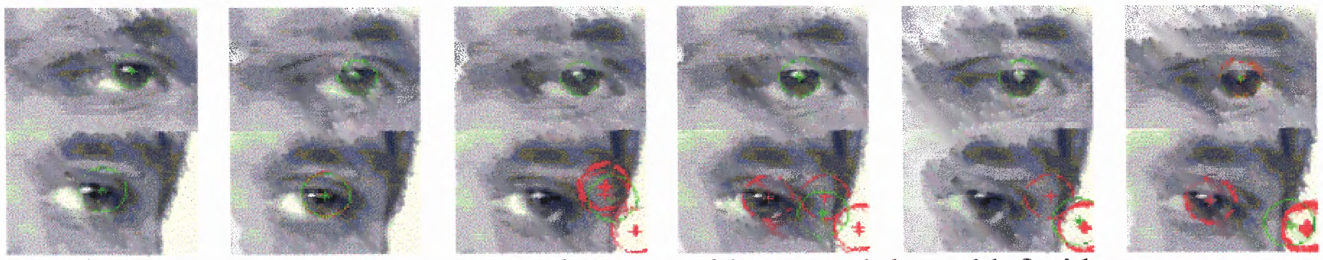


Figure 4.6: Sequence of processed images, right and left sides

4.3.1 Refinements and Optimizations

The method was improved over the test algorithm in a number of ways, some relating to time, some to accuracy, and some to both. Several of these were already discussed. First, as mentioned previously, the range of the radius sought by the algorithm need not be so vague. By assuming that the radius of the iris is not going to change drastically from frame to frame, it is only needed to search a wide area for the first of a sequence of frames, then narrow the search space to a narrow band centered at the previous radius. This cuts down the chance that some other circular pattern will emerge during the sequence, since the probability that a random feature would be the same size as the iris is much less than that it falls within a range from 20 to 0. It also reduces the processing time needed, since the depth of the Hough transform can be reduced so drastically. Memory usage is not reduced, however, since the first in a series of frames still needs to be processed by the complete transform.

Second, only the maximum value of the Hough transform is needed; if this yields more than one answer, then an average of these produces a reasonable response. This method was already discussed in the previous section.

Third, the angle between the camera's line of sight and the normal to the plane of the eye should be as small as possible. Since the human face is not flat, the camera does not need to be sitting inside the monitor for this to work; next to it seems to work best; above it works also, but results in the eyelashes obscuring the iris in some cases.

Fourth, the lighting conditions cannot be such as to obscure details. The test sets were generated under an overhead fluorescent light with a 60W desk lamp illuminating the camera's field of view. These conditions were chosen arbitrarily for two reasons; it is not far off from what can normally be found at a computer desk, and it was what was readily available.

4.3.2 Statistical Analysis

A statistical analysis of a sample set of 120 images processed yielded 106 hits, 6 misses, and 8 cases where the guess was close, but off. This results in an average true accuracy of 88.3%. The data comprising this statistical figure was accumulated using the optimizations described in the previous section. The code used in the tests is found in Appendix B. Here there are two versions of the Hough transform function besides the routine to generate a test pattern and measure it. The first function is the initial frame function, which searches a radius space of 10 to 20 pixels. Additional calls to the Hough transform used the second function, which searches a much narrower band of radius values. The data was collected in sets of six images, each 140x100 pixels, captured under identical lighting and camera conditions. The only major change from frame to frame was the direction of the target's attention, which varied either randomly around the computer screen or following the borders of the monitor. The criteria for success were that visually, the iris in the image shared the majority of its visible circumference and its center with that plotted by the algorithm. If the plot did not trace the outlines of at least 50% of the visible iris, or if the center was skewed, but still close, or if the average was off because of background information, the image was counted as a close miss. If the plot was completely off, having latched on to some other feature, the image was counted as a complete miss. Table 4.1 gives a more detailed breakdown of each of the twenty sets of six images each. The average was calculated as:

$$Av = \frac{Hits}{Hits + Misses + Close} = \frac{106}{120} = 88.333\% \qquad (4.7)$$

Set	Hit	Close	Miss
1	6	0	0
2	4	1	1
3	6	0	0
4	6	0	0
5	4	1	1
6	6	0	0
7	6	0	0
8	6	0	0
9	6	0	0
10	4	1	1
11	6	0	0
12	4	0	2
13	6	0	0
14	6	0	0
15	6	0	0
16	5	1	0
17	6	0	0
18	4	1	1
19	6	0	0
20	3	3	0
Total	106	8	6

Table 4.1: Details of Statistical Analysis

CHAPTER 5

POSTPROCESSING AND BEYOND

After the coordinates of the center of the iris have been successfully computed, the work of the Hough transform is done. But in order to develop useful information from this data, more computations have to be made. And these computations are useless if they cannot be implemented in hardware. This section deals with the nature, if not the specifics, of the calculations to be made, as well as a look ahead at the nature of the devices that could utilize this data.

5.1 Spatial Transforms

The first items to consider is are the post-processing calculations. These could be as simple as a matrix transformation or as complicated as a mathematical model of an eyeball, depending on the required accuracy. In any case, other features on the face may need to be located, much in the same way as the iris, merely to keep track of the background for positioning. The corners of the eye are good points to search for, but any feature that remains constant relative to the face will do for these purposes. In the simplest case this information can be used to construct a simple spatial transformation matrix based on a few calibration points. If the image space is considered to be two-dimensional then all that needs to be done is to find the two corners of the eye and make a few calculations. Observe the diagram in figure 5.1. It is assumed that the points labeled A, B, and C have been found using feature extraction methods. D has been computed by taking the x-coordinate of C and the y-coordinate of B. A set of polar coordinates in the image space can be extrapolated by computing the angle ABD and the distance AB and BC. These coordinates, once converted to rectangular space, give the image-space coordinates of the pupil. If several calibration points are taken, it would be no trouble to interpolate between screen-space and image-space, using a rubber-sheet transform to align the image-space coordinates with those of the screen-space [15].

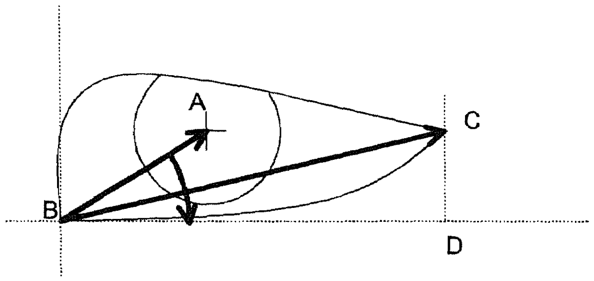


Figure 5.1: Simplified diagram of eye for image-space conversion

This simple assumption may work for some applications, but if more accuracy is required, it becomes necessary to develop a better method of conversion. One way of doing so would be to construct a mathematical model of the movement of a point on a sphere and map the data from the Hough transform on to this. Another method would be to train a neural network to perform the conversions; there have been a number of books and papers written about the subject of using neural networks in image processing. For more information, consultation of [3] or [9] would be beneficial.

5.2 Speculation on Hardware Implementations

The basic requirement of a hardware implementation of this algorithm would be the ability to perform the Hough transform and spatial conversions quickly. An array of fast DSPs could be programmed to do the job quickly and efficiently. If the camera is fixed in relation to the computer and the user is immobile then the number crunching ability of the DSP array is probably all that is required. But in order to make a more complicated unit with a small camera mounted in a headset or glasses, more information is needed. In addition to the data from the camera, at least three local GPS-style positioning devices would be required, as well as at least three transmitters. In this fashion, the camera and DSP array could calculate where the eye is looking relative to a frame of reference defined by triangulation from the positioning devices.

To make the device more error free at the cost of more processing, two cameras could be used, one trained on each eye. In this way if one should show erroneous data the other could correct it. If the two cameras are mounted in a headset, only one set of three positioning devices needs to be included, as both cameras could be working in the same coordinate system.

CHAPTER 6

CONCLUSIONS

This method of tracking the motion of the iris is very promising, and represents a major step in the process of creating a hands-free pointing device. It does pose several problems, however. Although an accuracy of 88.3% is not bad, it is not nearly as accurate as hand-held devices. Using an elliptical Hough transform rather than the circular one demonstrated in this document could increase the accuracy of this figure, but this would come at the cost of even more memory and processing power. Other options might be to use a camera with a higher resolution, or one capable of focusing at a much shorter distance. Using images taken in the infrared or ultraviolet ranges might also help, though this has not been tested. In addition to this, there are other obstacles to be overcome before such a device could be produced, mainly in the areas of locating the eye and in performing spatial transformations to correlate the position of the iris to the desired position of the pointer on the screen. Nevertheless, the method described in this document overcomes the primary obstacle to realizing this goal.

APPENDIX A

DBS DFG/LC1 VIDEO CAPTURE BOARD INFORMATION

Note: these are excerpts from [5], available at http://www.dbs-imaging.com

The DFG/LC1 and DFG/LC2 are great value 32 bit frame grabbers for industrial, medical and multimedia applications in monochrome and color. Video sources with S-VHS (YC) and CVBS outputs (composite video) are supported. The data transmission to the PC or to the VGA board is carried out via a DMA transfer in burst mode and therefore places only a very small load (if any) on the CPU. The images are available in 24-bit true color quality.

In order to help you use the DFG/LC1 and DFG/LC2 in your own programs with Windows 95/NT, a SDK (software development kit) is delivered with the product.

A.1 Features of the DFG/LC1

- Multi standard support (CCIR, PAL, NTSC, RS170, SECAM)
- Support for all VGA resolutions
- Color conversion from YC (YUV) to RGB in real-time
- RGB 24 bit and RGB 32 bit true color support
- Full video resolution of 768x576 at 50Hz (PAL/CCIR) and 640x480 at 60Hz (NTSC/RS170)
- Non destructive overlay support with DirectDraw
- Support for color manipulation as well as image manipulation
- Real-time hardware scaling

- EEPROM to which data may be written
- Programmable interface for Windows NT and Windows 95 (DLL)
- Save images in BMP format
- Direct data transfer to VGA card and the PC's system memory
- PCI bus master DMA interface

A.2 Possible image output modes

A.2.1 Bitmap mode (BMP/DIB)

Once the DFGDEMO demonstration program has been started, as described in the last chapter, the bitmap mode is activated. The image from the DFG/LC1 or DFG/LC2 is stored in the main memory of the PC. The live video display has to be programmed by the user. This should be done by using the CPU to generate a bitmap and then copying it to the VGA board. This mode has the lowest refresh rate. The great advantage of this mode is that it is compatible with all VGA cards and it is possible to access the image data in the memory. Overlay functions have to be programmed by yourself. As Windows takes over the control of the image display, the image can be completely or partly overlapped by many other windows and dialog boxes.

A.2.2 DirectDraw back buffer mode

In this mode, the image data is written to the non-visible area of the VGA card. The requirements for this are i) the DirectDraw driver, ii) sufficient memory on the VGA card and iii) back buffer support from the VGA cards' manufacturer. If a back buffer is not available on the VGA card, it will automatically be stored on the PC. Exactly the same goes for overlay data. Under ideal circumstances there would be image memory as well as overlay memory in the back buffer of the VGA card. Then the VGA chip does a bit block transfer of both memory areas to the visible area of the VGA board during the VSYNC period. The result is a live image display with overlaid text and graphics. With Windows 95 and Windows 98 this mode runs with almost all VGA cards. With Windows NT the background memory is not made available by the VGA card. The *Revolution* graphics card from *Number Nine* allows the background memory to be accessed. The image refresh frequency and the load on the CPU are dependant upon the color depth setting and upon the location of the back buffer (i.e. either the main memory of the PC or the image memory of the VGA card.) With the *Revolution* and Windows NT it is possible to achieve a near real-time image display with an overlay.

A.2.3 DirectDraw primary surface mode

The video image is digitized straight in the visible area of the VGA card. In this mode the highest image refresh rate and fast live image display are possible. It is not possible to overlay images in this mode, as the image data is immediately overwritten by the next image. Unlike many other frame grabber cards, dialog boxes and other windows can be placed over the live image without affecting it. To achieve this, clipping lists are transferred to the board on a second DMA channel and those areas of the video image are left out while displaying.

A.2.4 DirectDraw overlay surface mode

In this mode, a live image with an output speed of the primary surface mode and at the same time display of the overlay can be achieved. One further advantage is that the VGA cards which offer these hardware properties are cheaply available. The video image is digitized to a non-visible area of the VGA board. This area always has to be on the VGA board. By defining a key color and drawing with that color to the output window, the video image will be displayed only in those parts where the key color is used. When the window is filled with the key color, the video image is displayed completely. Graphics and text elements which use a different color will not be destroyed (non-destructive overlay). The fading is done by the VGA chip and requires hardly any CPU cycles. This mode is not supported by all VGA chips and only in 8, 15, and 16 bit mode. The following VGA chips are known to support this mode:

- S3 Virge VX (86C988)
- S3 Virge DX (86C375)
- S3 Virge GX
- S3 Virge GX2
- 3D Rage Pro

The best text and graphics overlay is achieved with the following video mode:

```
Mode = IS_SET_DM_DIRECTDRAW | IS_SET_DM_ALLOW_OVERLAY
```

If the video image should be scaled to the size of a window, the following can be used:

```
Mode = IS_SET_DM_DIRECTDRAW | IS_SET_DM_ALLOW_OVERLAY |
IS_SET_DM_ALLOW_SCALING | IS_SET_DM_ALLOW_FIELDSKIP
```

So as to prevent the interlace effect with moving images, and extra mode can be activated when scaling. This is achieved with:

```
Mode = IS_SET_DM_DIRECTDRAW | IS_SET_DM_ALLOW_OVERLAY |
IS_SET_DM_ALLOW_SCALING
```

A.3 Color and save formats

Each of the color formats which the DFG/LC2 supports uses a different format to save images, as indicated in the following table:

Format DWORD		Pixel Data [31:0]			
		Byte 3 [31:24]	Byte 2 [23:16]	Byte 1 [15:8]	Byte 0 [7:0]
RGB32	dw0	Dummy	R	G	B
RGB24	dw0	B1	R0	G0	B0
	dw1	G2	B2	R1	G1
	dw2	R3	G3	B3	R2
RGB16	dw0	{R1[15:11], G1[10:5], B1[4:0]}*		{R0[15:11], G0[10:5], B0[4:0]}*	
RGB15	dw0	{0, R1[15:11], G1[10:5], B1[4:0]}*		{0, R0[15:11], G0[10:5], B0[4:0]}*	
YUV422	dw0	Cr0	Y1	Cb0	Y0
	dw1	Cr2	Y3	Cb2	Y2
YUV411	dw0	Y1	Cr0	Y0	Cb0
	dw1	Y3	Cr4	Y2	Cb4
	dw2	Y7	Y6	Y5	Y4
Y8	dw0	Y3	Y2	Y1	Y0

* With the file formats RGB16 and RGB15, from the internal 8 bit R,G and B colors, the upper bits, instead of using all of the available bits, are used.

Table A.1: Color formats of DFG/LC1/2 Video Capture card

A.4 Description of driver library

The driver for the DFG/LC2 has been written with the Microsoft Visual C/C+ compiler. For developers who work with Borland C/C++ a definition file (falconbc.def) is copied to the \develop\bc directory when the driver is installed. This definition file has to be added to your Borland project. For VisualBasic developers, an example project file is available in the \develop\vb directory. Likewise for Delphi users there is a file with a choice of function definitions in the \develop\delphi directory.

A.4.1 BOARDINFO data structure of the EEPROM

Using the *is_GetBoardInfo()* function the data which has been written to the DFG/LC1 can be read out. The data structure is build up as follows:

char	SetNo[12]	Card's serial number
char	ID[20]	"DFG/LC1 – DBS GmbH"
char	Version[10]	"V1.00" or later version
char	Date[12]	"11.11.1999" system date of end test
char	Reserved[10]	reserved

Table A.2: Data structure of DFG/LC1 EEPROM

A.4.2 Function list DFG/LC2

Initialization and termination

`is_InitDFG/LC1`	Hardware and image memory initialization
`is_ExitDFG/LC1`	Closes the card and de-allocates memory

Image acquisition and memory management

`is_AllocImageMem`	Allocates image memory
`is_CaptureVideo`	Acquires live video
`is_CopyImageMem`	Copies image to memory as defined by programmer
`is_FreeImageMem`	Frees allocated image memory
`is_FreezeVideo`	Acquires image and writes to destination address (a.k.a. Snap)
`is_GetActiveImageMem`	Returns number of active image memory
`is_GetImageMem`	Returns pointer to image memory
`is_GetImageMemPitch`	Returns line offset (n) to (n+1)
`is_HasVideoStarted`	Has the image acquisition started?
`is_InquireImageMem`	Returns image memory's properties
`is_IsVideoFinish`	Has the image acquisition finished?
`is_SetImageMem`	Makes image memory active
`is_StopLiveVideo`	Stops live video

Double and multiple buffering

`is_AddToSequence`	Records image memory in sequence list
`is_ClearSequence`	Deletes sequence list
`is_GetActSeqBuf`	Determines the image memory which is currently being used for the sequence.

is_LockSeqBuf	Protects image memory of the sequence from being overwritten.
is_UnlockSeqBuf	Allows image memory of the sequence to be overwritten.

Selection of operating mode and return of properties

is_BoardStatus	Gets event counter and counter value
is_GetBoardType	Gets type of board (DFG/LC1 or DFG/LC2)
is_GetColorDepth	Gets current color mode from VGA card
is_GetCurrentField	Returns currently active field (odd or even)
is_GetError	Calls error message
is_GetOsVersion	Calls operating system type
is_GetVsyncCount	Output of VSYNC counter
is_SetAGC	Turns on or off automatic gain control
is_SetBrightness	Sets brightness
is_SetCaptureMode	Sets acquisition mode
is_SetColorMode	Selects color mode
is_SetContrast	Sets contrast
is_SetDisplayMode	Selects image display mode
is_SetErrorReport	Activates or deactivates error output
is_SetGamma	Activates or deactivates gamma correction
is_SetHorFilter	Sets horizontal interpolation filter
is_SetHue	Sets hue value
is_SetHwnd	Resets handle to output window
is_SetImagePos	Sets image position within image window
is_SetImageSize	Sets the size of the image
is_SetSaturation	Sets color saturation
is_SetScaler	Toggles scaler on and off; sets scaling ratio
is_SetSyncLevel	Sets sync level for critical video sources (VCR)
is_SetVertFilter	Sets vertical interpolation filter
is_SetVideoInput	Selection of video input signal
is_SetVideoMode	Selection of video standard
is_SetVideoSize	Sets the digital area of interest on the image

Reading from and writing to EEPROM

is_WriteEEPROM	Writes own data to EEPROM
is_ReadEEPROM	Reads own data from EEPROM
is_GetBoardInfo	Reads preprogrammed manufacturer's information from board

Saving and loading images

is_SaveImage	Saves video image as a bitmap (BMP)
is_SaveImageMem	Saves image memory as a bitmap (BMP)
is_LoadImage	Loads a bitmap (BMP) image

Image output

`is_UpdateDisplay`	Displays refresh with DirectDraw
`is_ShowColorBars`	Displays color bars from pattern generator
`is_SetUpdateMode`	Sets the mode for the display refresh

Supplementary DirectDraw functions

`is_EnableDDOverlay`	Toggles DirectDraw overlay memory
`is_DisableDDOverlay`	Switches off DirectDraw overlay memory
`is_ShowDDOverlay`	Displays overlay memory
`is_HideDDOverlay`	Hides overlay memory
`is_GetDDOverlayMem`	Retrieves address of overlay memory
`is_GetDDOvlSurface`	Returns pointer to DirectDraw surface
`is_GetDC`	Retrieves the device context handle's overlay memory
`is_ReleaseDC`	Releases the device context handle's overlay memory
`is_SetDDUpdateTime`	Update cycle of image output with DirectDraw
`is_LockDDOverlayMem`	Enables access to overlay memory
`is_UnlockDDOverlayMem`	Disables access to overlay memory
`is_LockDDMem`	Enables VGA card to access the back buffer
`is_UnlockDDMem`	Disables VGA card to access the back buffer
`is_SetKeyColor`	Sets the keying color for the overlay display.
`is_OvlSurfaceOffWhileMove`	Switches off overlay surface when the window is moved
`is_PrepareStealVideo`	Initializes image stealing
`is_StealVideo`	Steals image from a DirectDraw live mode

Event Handling (nur WinNT 4.0)

`is_InitEvent`	Sets up event handler
`is_ExitEvent`	Exits event handler
`is_EnableEvent`	Enable object event
`is_DisableEvent`	Disable object event

A.5 Description of functions

To aid the integration of the DFG/LC1 and DFG/LC2 into your own programs, the functions from the driver library, which are shipped with the grabber, are described in this chapter:

Function name: **<name of function>**
Syntax: Function prototype from the header file falcon.h
Description: Function description with cross reference to affected functions
Parameters: Description of the function parameters with range of values
Return value: Description and range of return value. If function returns IS_NO_SUCCESS (-1), the error can be called with the function *is_GetError()*.

Function name: **is_AllocImageMem**
Syntax: `INT is_AllocImageMem(HIDS hf, INT width, INT height, INT bitspixel, char** ppcImgMem, INT* pid)`
Description: *is_AllocImageMem()* allocates image memory for an image with width, *width* and height, *height* and color depth *bitspixel*. Memory size is at least:
size =[width * ((bitspixel + 1) / 8) + adjust] * height (for *adjust* see below)
Line increments are calculated with:

> line = width * [(bitspixel + 1) / 8]
> lineinc = line + adjust.
> adjust = 0 when *line* without *rest* is devisable by 4
> adjust = 4 - rest(line / 4) when *line* without *rest* is not devisable by 4

The line increment can be read with the *is_GetImgMemPitch()* function. The start address in the image memory is returned with *ppcImgMe*m. *pid* contains an identification number of the allocated memory. A newly activated memory location is not directly activated.
In other words, images are not directly digitized to this new memory location. Before this can happen, the new memory location has to be activated with *is_SetImageMem()*.
After *is_SetImageMem()* a *is_SetImageSize()* must follow so that the image conditions can be transferred to the newly activated memory location. The returned pointer has to be saved and may not be used, as it is required for all further *ImageMem* functions. The freeing of the memory is achieved with *is_FreeImageMem()*. In the DirectDraw modes, the allocation of an image memory is not required!
Parameters:

hf	Frame gabber handle
width	Width of image
height	Height of image
bitspixel	Width of image
ppcImgMem	Contains pointer to start of memory location
pid	Contains the ID for this memory location

Return value: IS_SUCCESS, IS_NO_SUCCESS

Function name: **is_ExitBoard**
Syntax: `INT is_ExitBoard (HFALC hf)`
Description: *is_ExitBoard()* cancels the active device handle *hf* and deallocates the data structures and memory areas currently associated with the DFG/LC2. The image memory which has been allocated by the user and which has not been released, is released with *is_ExitBoard.*
Parameters:

hf	Frame grabber handle

Return value: IS_SUCCESS, IS_NO_SUCCESS

Function name: **is_FreeImageMem**
Syntax: `INT is_FreeImageMem(HFALC hf, char* pcImgMem, INT id)`
Description: is_FreeImageMem() deallocates previously allocated image memory. For *pcImgMem* one of the pointers from *is_AllocImgMem()* has to be used. All other pointers lead to an error message! The repeated handing over of the same pointers also leads to an error message!
Parameters:

hf	Frame grabber handle
pcImgMem	Pointer to the beginning of the memory
id	ID for this memory

Return value: IS_SUCCESS, IS_NO_SUCCESS

Function name: **is_FreezeVideo**
Syntax: `INT is_FreezeVideo(HFALC hf, INT Wait)`
Description: *is_FreezeVideo()* digitizes an image and transfers it to the active image memory. In DirectDraw mode the image is digitized in the DirectDraw buffer (either on the VGA card or in a back buffer). When working with ring buffering, image acquisition only takes place in the first image memory of the sequence.
Parameters:

hf	Frame grabber handle	
Wait	IS_DONT_WAIT	The function returns straight away
	IS_WAIT	The function waits until an image is grabbed
	10 < Wait < 32768:	wait time in 10 ms steps (Wait = 100>1 sec) A maximum of 328,68 seconds can be waited. This is approx. 5 mins. and 20 secs). For 1 < Wait < 10 Wait becomes equal to 10.

Return value: S_SUCCESS, IS_NO_SUCCESS or IS_TIMED_OUT if a time out takes place.

Function name: **is_InitBoard**
Syntax: `INT is_InitBoard (HFALC phf, HWND hWnd)`
Description: *is_InitBoard()* opens the driver and establishes contact to the hardware. If the hardware is successfully initialized, this function allocates a handle to the frame grabber. All of the following functions require this as their first parameter. If DirectDraw is not used for image output, *hWnd* can be set to *NUL*L. If several DFG/LC2 or DFG/LC1 frame grabbers are installed in one computer, each one is assigned its own, unique handle.
Parameters:

phf	Pointer to the frame grabber handle
hWnd	Handle to the window in which the image is to be displayed

Return value: IS_SUCCESS, error code (see header file)

Function name: **is_SetCaptureMode**
Syntax: `is_SetCaptureMode (HFALC hf, INT Mode)`
Description: *is_SetCaptureMode()* sets the required digitizing mode. The acquisition of even or uneven fields and totally interlaced video frames is supported. If the acquisition

of half images is required and at the same time these images are to be displayed under one another, the IS_SET_CM_NONINTERLACED paramater can be used.

Parameters:

hf		Frame grabber handle	
Mode	1	IS_SET_CM_ODD	Acquires odd half images
	2	IS_SET_CM_EVEN	Acquires even half images
	4	IS_SET_CM_FRAME	Acquires interlaced video images
	8	IS_SET_CM_NONINTERLACED	Acquires non-interlaced video images
0x8000		IS_GET_CAPTURE_MODE	Retrieval of current setting

Return value: Current setting when called with IS_GET_CAPTURE_MODE else IS_SUCCESS, IS_NO_SUCCESS.

Function name: **is_SetColorMode**

Syntax: `INT is_SetColorMode (HFALC hf, INT Mode)`

Description: is_SetColorMode() sets the required color mode with which the image data is to be saved or displayed by the VGA board. For the first case it is important that, depending upon the color mode which is used, the allocated image memory is large enough. A 24 bit color image requires three times as much memory as an 8 bit monochrome image. When accessing the image data, it is important to know how the memory is arranged in each of the memory modes. Incorrect interpretation of the memory contents leads to incorrect results. With the direct transfer to the VGA card's image memory, it is important to ensure that the display settings correspond to those of the DFG/LC1's color mode. Under certain circumstances the images can be displayed with either the incorrect colors or they can become so distorted that it is impossible to recognize what is actually being displayed.

Please note: The higher the pixel depth of the transferred images, the more load on the PCI bus. It is always a good idea to select gray scale mode for monochrome images and display the image at 256 colors.

Parameters:

hf		Frame grabber handle	
Mode	0	IS_SET_CM_RGB32	32 bit true color mode; R-G-B-Dummy
	1	IS_SET_CM_RGB24	24 bit true color mode; R-G-B
	2	IS_SET_CM_RGB16	Hi color mode; 5 R - 6 G - 5 B
	3	IS_SET_CM_RGB15	Hi color mode; 5 R - 5 G - 5 B
	4	IS_SET_CM_YUV422	YUV4:2:2 (no display possible with VGA board)
	5	IS_SET_CM_YUV411	YUV4:1:1 (no display possible with VGA board)
	6	IS_SET_CM_Y8	8 bit monochrome image
	7	IS_SET_CM_	RGB8 8 bit color image; 2 R - 3 G - 2 B
0x8000		IS_GET_COLOR_MODE	Retrieval of current setting

Return value: Current setting when called with IS_GET_CAPTURE_MODE, else IS_SUCCESS, IS_NO_SUCCESS.

Function name: **is_SetDisplayMode**

Syntax: `INT is_SetDisplayMode (HIDS hf, INT Mode)`
Description: *is_SetDisplayMode()* defines the way in which images are displayed on the screen. For real live video plus overlay, the DirectDraw overlay surface mode has been introduced. The availability of this mode depends upon the type of VGA card used. Only certain VGA controllers support this mode. The best known are the S3 Virge VX (limitations apply) S3 Virge DX/GX/GX2 and the 3D RagePro Chip. This overlay mode can be used with these VGA chips in 8, 15 and 16 bit VGA mode. True color (24 bit) is not supported. In 8 bit VGA mode, the color mode has to be set to RGB15! The VGA card should have at least 4MB on-board, as the overlay mode requires memory up to the size of the current VGA resolution. For example: VGA with 1024x768x16 = 1.5 MB -> OverlayBuffer with up to 1.5 MB required: VGA card with 4MB recommended
Example: `is_SetDisplayMode (hf, Mode);`
Bitmap mode (digitized in system memory):
    Mode=IS_SET_DM_DIB
DirectDraw back buffer mode (Overlay possible. See *is_EnableDDOverlay*)
    Mode=IS_SET_DM_DIRECTDRAW
Allows memory storage in system memory:
    Mode=IS_SET_DM_DIRECTDRAW |
    IS_SET_DM_ALLOW_SYSMEM
DirectDraw primary surface modus (no overlay possible):
    Mode=IS_SET_DM_DIRECTDRAW |
    IS_SET_DM_ALLOW_PRIMARY
DirectDraw overlay surface mode (best live overlay):
    Mode=IS_SET_DM_DIRECTDRAW |
    IS_SET_DM_ALLOW_OVERLAY
Allows automatic scaling to the size of window:
    Mode=IS_SET_DM_DIRECTDRAW |
    IS_SET_DM_ALLOW_OVERLAY |
    IS_SET_DM_ALLOW_SCALING
Avoid interlaced effect at y<288:
    Mode=IS_SET_DM_DIRECTDRAW |
    IS_SET_DM_ALLOW_OVERLAY |
    IS_SET_DM_ALLOW_SCALING |
    IS_SET_DM_ALLOW_FIELDSKIP
All other mode combinations are not valid!
Parameters:

hf		Frame grabber handle	
Basic mode			
Mode	1	IS_SET_DM_DIB	Acquire image in image memory
	2	IS_SET_DM_DIRECTDRAW	DirectDraw mode (back buffer modus)
DirectDraw back buffer extension			
0x40		IS_SET_DM_ALLOW_SYSMEM	DirectDraw Buffer can be set in PC, if the VGA memory is not large enough.
DirectDraw primary surface extension			

0x80	IS_SET_DM_ALLOW_PRIMARY	Primary Surface mode (no overlay possible) DirectDraw back buffer extension
0x100	IS_SET_DM_ALLOW_OVERLAY	Overlay surface mode
0x200	IS_SET_DM_ALLOW_SCALING	Real time scaling in overlay mode. Field display when ysize < PAL/2 (288)
0x400	IS_SET_DM_ALLOW_FIELDSKIP	

Return mode

0x8000	IS_GET_DISPLAY_MODE	Returns current settings

Return value: IS_SUCCESS, IS_NO_SUCCESS or current setting with IS_GET_DISPLAY_MODE

Function name: **is_SetImageSize**
Syntax: `INT is_SetImageSize (HFALC hf, INT x, INT y)`
Description: *is_SetImageSize()* determines the image size when used with the setting from *is_SetImagePos()*.
Parameters:

hf	Frame grabber handle
x	Width of t he image: range 0..768 IS_GET_IMAGE_SIZE_X IS_GET_IMAGE_SIZE_Y
y	Height of image: range 0..576

Return value: When used with IS_GET_IMAGE_SIZE_X and IS_GET_IMAGE_SIZE_Y the current settings are read, when not IS_SUCCESS or IS_NO_SUCCESS.

Function name: **is_SetVideoMode**
Syntax: `INT is_SetVideoMode (HFALC hf, INT Mode)`
Description: *is_SetVideoMode()* selects the video standard for the connected video source. Color and monochrome signal standards with 50 Hz and 60 Hz are supported. With the IS_SET_VM_AUTO parameter, the video standard of the connected video source can automatically be set. All the other required settings can be carried out with the driver.
Parameters:

hf	Frame grabber handle		
Mode	0	IS_SET_VM_PAL	PAL standard color 50 Hz
	1	IS_SET_VM_NTSC	NTSC standard color 60 Hz
	2	IS_SET_VM_SECAM	SECAM standard 50 Hz
	3	IS_SET_VM_AUTO	Automatic video standard detection
0x8000		IS_GET_VIDEO_MODE	Retrieval of current setting

Return value: With IS_GET_VIDEO_MODE, the current settings are read, else IS_SUCCESS or IS_NO_SUCCESS.

APPENDIX B

SOURCE CODE SAMPLES

B.1 CapInit.cpp: Initialization Code for Capture Board

```
// These files are required by Matlab and by the Capture
// card libraries.  Falcon.h is the header for the card.
#include <afxwin.h>
#include <afxext.h>
#include "mex.h"
#include "falcon.h"

// Make the function useful for different camera sizes.
#define WIDTH 640
#define HEIGHT 480

// These variables are required by the card libraries,
// and are simply easier to use as global variables.
HFALC board;
char* image;
int id;

// The initialization function.
void initialize()
{
   int error;
   error = is_InitBoard(&board, NULL);
   error = is_SetVideoMode(board, IS_SET_VM_NTSC);
   error = is_SetCaptureMode(board, IS_SET_CM_FRAME);
   error = is_SetColorMode(board, IS_SET_CM_RGB32);
   error = is_SetDisplayMode(board, IS_SET_DM_DIB);
   error = is_AllocImageMem(board, WIDTH, HEIGHT, 32,
                            &image, &id);
   error = is_SetImageMem(board, image, id);
   error = is_SetImageSize(board, WIDTH, HEIGHT);
}

// The format of this function is specified by Matlab.
void mexFunction( int nlhs, mxArray *plhs[], int nrhs,
                  const mxArray *prhs[] )
{
   int *caphandle, *capmem, *capnum;

   if (nrhs)
   {
        mexErrMsgTxt("No inputs required");
    }

    else if (nlhs != 3)
    {
        mexErrMsgTxt("Wrong number of outputs");
    }

// Call to the function defined above.
    initialize();

// Convert values returned to something useable by Matlab
    const int dims[] = {1};
    plhs[0] = mxCreateNumericArray(1, dims, mxUINT32_CLASS,
                                    mxREAL);
    caphandle = (int*)mxGetPr(plhs[0]);
    caphandle[0] = (int)board;
    plhs[1] = mxCreateNumericArray(1, dims, mxUINT32_CLASS,
                                    mxREAL);
    capmem = (int*)mxGetPr(plhs[1]);
    capmem[0] = (int)image;
    plhs[2] = mxCreateNumericArray(1, dims, mxUINT32_CLASS,
                                    mxREAL);
    capnum = (int*)mxGetPr(plhs[2]);
    capnum[0] = (int)id;

    return;
}
```

B.2 CapSnap.cpp: Activation Code for Capture Board

```
// These files are required by Matlab and by the Capture
// card libraries.  Falcon.h is the header for the card.
#include <afxwin.h>
#include <afxext.h>
#include "mex.h"
#include "falcon.h"

// Make the function useful for different camera sizes.
#define WIDTH 640
#define HEIGHT 480

// These variables are required by the card libraries,
// and are simply easier to use as global variables.
HFALC board;
char* image;
int id;

// The capture function.  FreezeVideo must be run twice
// because of the interlacing used during capture.
void capture()
{
   int error;
   error = is_FreezeVideo(board, IS_WAIT);
   error = is_FreezeVideo(board, IS_WAIT);
}

// The format of this function is specified by Matlab.
void mexFunction( int nlhs, mxArray *plhs[], int nrhs,
                  const mxArray *prhs[] )
{
   char *img;
   int *temp;

   if (nrhs != 3)
   {
      mexErrMsgTxt("Wrong number of inputs");
   }
   else if (nlhs != 1)
   {
      mexErrMsgTxt("Wrong number of outputs");
   }

// Convert values passed from Matlab into values useful
// to the capture card.
   const int dims[] = {HEIGHT, WIDTH, 3};
   plhs[0] = mxCreateNumericArray(3, dims, mxUINT8_CLASS,
                                           mxREAL);
   img = (char*)mxGetPr(plhs[0]);
   temp = (int*)mxGetData(prhs[0]);
   board = (HFALC)temp[0];
   temp = (int*)mxGetData(prhs[1]);
   image = (char*)temp[0];
   temp = (int*)mxGetData(prhs[2]);
   id = (int)temp[0];

// Call the capture function defined above.
   capture();

// Rearrange the pixels of the image from the format
// delivered by the board to that needed by Matlab.
   for (int color=0; color<3; color++)
   {
      for (int column=0; column<WIDTH; column++)
      {
         for (int row=0; row<HEIGHT; row++)
         {
            img[WIDTH*HEIGHT*color + HEIGHT*column + row] =
               image[WIDTH*row*4 + 4*column + 2-color];

         }
      }
   }
   return;
}
```

B.3 CapQuit.cpp: Deactivation Code for Capture Board

```
// These files are required by Matlab and by the Capture
// card libraries.  Falcon.h is the header for the card.
#include <afxwin.h>
#include <afxext.h>
#include "mex.h"
#include "falcon.h"

// These variables are required by the card libraries,
// and are simply easier to use as global variables.
HFALC board;
char* image;
int id;

// The shutdown function
void shutdown()
{
   int error;
   error = is_FreeImageMem(board, image, id);
   error = is_ExitBoard(board);
}

// The format of this function is specified by Matlab.
void mexFunction( int nlhs, mxArray *plhs[], int nrhs,
const mxArray *prhs[] )
{
   char *img;
   int *temp;

   if (nrhs != 3)
   {
      mexErrMsgTxt("Wrong number of inputs");
   }
   else if (nlhs)
   {
      mexErrMsgTxt("No outputs required");
   }

// Convert values passed from Matlab into values useful
// to the capture card.
   temp = (int*)mxGetData(prhs[0]);
   board = (HFALC)temp[0];
   temp = (int*)mxGetData(prhs[1]);
   image = (char*)temp[0];
   temp = (int*)mxGetData(prhs[2]);
   id = (int)temp[0];
// Call the shutdown function defined above.
   shutdown();

   return;
}
```

B.4 HoughTest.m: Test routine for Circular Hough Transform.

```
% Set up test matrix (25x25, 4 hot points)
A = zeros(25,25);
A(7,7) = 1;
A(18,18) = 1;
A(7,18) = 1;
A(18,7) = 1;

% only look for enough circles to catch all points
thresh = 4;

% initial variable set up.
C = [];
X = 25;
Y = 25;
R = 25;
B = zeros(X,Y);
H = zeros(X,Y,R);
a = 0;
b = 0;

% Loop through image to create transform.
% Commented section is part removed in later versions
% and replaced by following section to cover
% rounding errors.
for i = 1:X
   for j = 1:Y
      if A(i,j) > 0
         for r = 1:R
            T = zeros(X,Y);
            for theta = 0:.01:2*pi

%                 a = round(i + r*cos(theta));
%                 b = round(j + r*sin(theta));
%                 if (a>=1) & (a<=X) & (b>=1) & (b<=Y)
%                    T(a,b) = 1;
%                 end

               a = ceil(i + r*cos(theta));
               b = ceil(j + r*sin(theta));
               if (a>=1) & (a<=X) & (b>=1) & (b<=Y)
                  T(a,b) = 1;
               end
```

```
              a = a-1;
              b = b-1;
              if (a>=1) & (a<=X) & (b>=1) & (b<=Y)
                 T(a,b) = 1;
              end
           end
           H(:,:,r) =  H(:,:,r) + T(:,:,1);
        end
     end
  end
end

% Loop through transform to acquire data on cells
% beyond the threshold.
for i = 1:X
   for j = 1:Y
      for r = 1:R
         if H(i,j,r) >= thresh
            C = [C; i, j, r, H(i,j,r)];
         end
      end
   end
end

% Display the data on screen and in the image.
C
figure;
imshow(A);
hold on;
t1 = 0:0.01:2*pi;
plot(C(:,2), C(:,1), 'r+');
t2 = size(C);
for i = 1:t2(1)
   plot(C(i,2)+C(i,3)*sin(t1),
        C(i,1)+C(i,3)*cos(t1), 'r--');
end
```

B.5 Hough2.m: Circular Hough Transform function

```
function [xc, yc, rl] = hough(A,AA)

% Set up initial min and max values for R.
Rn = 10;
Rx = 20;

% Get the X and Y dimensions of the image
temp = size(A);
X = temp(1);
Y = temp(2);

% Set up other variables.
C = [];
B = zeros(X,Y);
H = zeros(X,Y,(Rx - Rn));
a = 0;
b = 0;

% Loop through the image and create the Hough transform
array.
for i = 1:X
   for j = 1:Y
      if A(i,j) > 0
         for r = Rn+1:Rx
            T = zeros(X,Y);
            for theta = 0:.01:2*pi
               a = round(i + r*cos(theta));
               b = round(j + r*sin(theta));
               if (a >= 1) & (a <= X) & (b >= 1) & (b <= Y)
                  T(a,b) = 1;
               end
            end
            H(:,:,(r-Rn)) = H(:,:,(r-Rn)) + T(:,:,1);
         end
      end
   end
end

% Set the threshold to be the maximum value of the
transform.
thresh = max(max(max(H)));
```

```
% Loop through the transform to find data on circles
encountered.
for i = 1:X
   for j = 1:Y
      for r = Rn+1:Rx
         if H(i,j,(r-Rn)) >= thresh
            C = [C; i, j, r, H(i,j,(r-Rn))];
         end
      end
   end
end

% Compute average circle for final results
temp = [(sum(C(:,1).*C(:,4))/sum(C(:,4))),
   (sum(C(:,2).*C(:,4))/sum(C(:,4))),
   (sum(C(:,3).*C(:,4))/sum(C(:,4)))]

% Display the circle(s) overlaid on the grayscale image.
figure;
imshow(AA);
hold on;
t1 = 0:0.01:2*pi;
plot(C(:,2), C(:,1), 'r+');
t2 = size(C);
for i = 1:t2(1)
   plot(C(i,2)+C(i,3)*sin(t1),C(i,1)+C(i,3)*cos(t1),'r--');
end
plot(temp(2), temp(1), 'g*');
plot(temp(2)+temp(3)*sin(t1),temp(1)+temp(3)*cos(t1),'g-');

% Return a the average center point and radius.
C
xc = temp(1);
yc = temp(2);
r1 = temp(3);
```

B.6 Hough3.m: Hough Transform with initial radius input

```
function [xc, yc, rl] = hough(A,AA,Rpass)

% this is only difference between hough3.m and hough2.m
Rn = floor(Rpass-1);
Rx = ceil(Rpass+1);

temp = size(A);
X = temp(1);
Y = temp(2);

C = [];
B = zeros(X,Y);
H = zeros(X,Y,(Rx - Rn));
a = 0;
b = 0;

for i = 1:X
   for j = 1:Y
      if A(i,j) > 0
         for r = Rn+1:Rx
            T = zeros(X,Y);
            for theta = 0:.01:2*pi
               a = round(i + r*cos(theta));
               b = round(j + r*sin(theta));
               if (a >= 1) & (a <= X) & (b >= 1) & (b <= Y)
                  T(a,b) = 1;
               end
            end
            H(:,:,(r-Rn)) = H(:,:,(r-Rn)) + T(:,:,1);
         end
      end
   end
end

thresh = max(max(max(H)));
for i = 1:X
   for j = 1:Y
      for r = Rn+1:Rx
         if H(i,j,(r-Rn)) >= thresh
            C = [C; i, j, r, H(i,j,(r-Rn))];
         end
      end
   end
end
```

```
temp = [(sum(C(:,1).*C(:,4))/sum(C(:,4))),
    (sum(C(:,2).*C(:,4))/sum(C(:,4))),
    (sum(C(:,3).*C(:,4))/sum(C(:,4)))]

figure;
imshow(AA);
hold on;
t1 = 0:0.01:2*pi;
plot(C(:,2), C(:,1), 'r+');
t2 = size(C);

for i = 1:t2(1)
   plot(C(i,2)+C(i,3)*sin(t1),C(i,1)+C(i,3)*cos(t1),'r--');
end
plot(temp(2), temp(1), 'g*');
plot(temp(2)+temp(3)*sin(t1),temp(1)+temp(3)*cos(t1),'g-');


C
xc = temp(1);
yc = temp(2);
r1 = temp(3);
```

REFERENCES

1. G. Baxes, *Digital Image Processing: Principles and Applications*, John Wiley & Sons, New York, NY, 1994.

2. S. Bow, *Pattern Recognition and Image Preprocessing*, Marcel Dekker, New York, NY, 1992.

3. D. Casasent, "Optics and Neural Nets", in *Neural Networks for Vision and Image Processing*, MIT Press, Cambridge, MA, 1992.

4. K. Castleman, *Digital Image Processing*, Prentice Hall, Englewood Cliffs, NJ, 1979.

5. DBS, *DFG/LC1+2 Frame Grabber User Manual,* DBS Digitale Bildverarbeitung und Systementwicklung GmbH, Bremen, Germany, 1999.

6. R. Gonzales and R. Woods, *Digital Image Processing*, Addison-Wesley, New York, NY, 1993.

7. R. Haralick and L. Shapiro, *Computer and Robot Vision, Vol. 1*, Addison-Wesley, Reading, MA, 1992.

8. B. Jähne, *Digital Image Processing: Concepts, Algorithms, and Scientific Applications*, Springer-Verlag, Berlin, Germany, 1997.

9. J. Lampinen, J. Laaksonen, and E. Oja, "Pattern Recognition," in *Image Processing and Pattern Recognition*, Academic Press, San Diego, CA, 1998.

10. M. Levine, *Vision in Man and Machine*, McGraw-Hill, New York, NY, 1985.

11. C. Lindley, *Practical Image Processing in C*, John Wiley & Sons, New York, NY, 1991.

12. R. Nevatia, "Image Segmentation" in *Handbook of Pattern Recognition and Image Processing*, Academic Press, Orlando, FL, 1986.

13. C. Olson, "Constrained Hough Transforms for Curve Detection," *Computer Vision and Image Understanding*, vol. 73, no. 3, pp. 329-345, Academic Press, Orlando, FL, 1999.

14. J. Ramesh, R. Kasturi, and B. Schunck, *Machine Vision*, McGraw-Hill, New York, NY, 1995.

15. S. Umbaugh, *Computer Vision and Image Processing*, Prentice Hall PTR, Upper Saddle River, NJ, 1998.