University of Southern California – Los Angeles
Ming Hsieh Department of Electrical Engineering
Spring 2010

Eye Catchers: A System for Eye Gaze Detection Using the TI C6416 DSP

By: Ashvin Deodhar {deodhar@usc.edu},
Trevor McGuire {temcguir@usc.edu}, and
Jason Tokayer {tokayer@usc.edu}

For: EE586L – Advanced DSP Laboratory

Date: May 7th, 2010

1. Introduction

Computers can assist disabled individuals in communicating and living an every-day life, especially those with limited ability to speak and move. In this project we will implement gaze tracking for selection of options displayed on a computer screen. Our target user is a person with limited motor skills, and our goal is to provide him with the ability to communicate the desire for some basic necessities such as food and drink.

Several complex algorithms for gaze detection exist but we concentrate on simplification for real-time implementation purposes on the Texas Instruments C6416 fixed-point digital signal processor. We adapt a version of the Starburst algorithm1 for our implementation. When using infrared illumination, the most common feature used for tracking is the pupil-iris boundary. However, when using visible spectrum illumination, the most notable feature that we can track is the iris-sclera boundary. This boundary is commonly referred to as the limbus. At the limbus boundary the luminance gradient is very high and can therefore be detected by simple gradient algorithms. In this project we use visible spectrum illumination and thus concentrate on tracking the limbus contour. Figure 1.1 below shows the notable features of the eye.

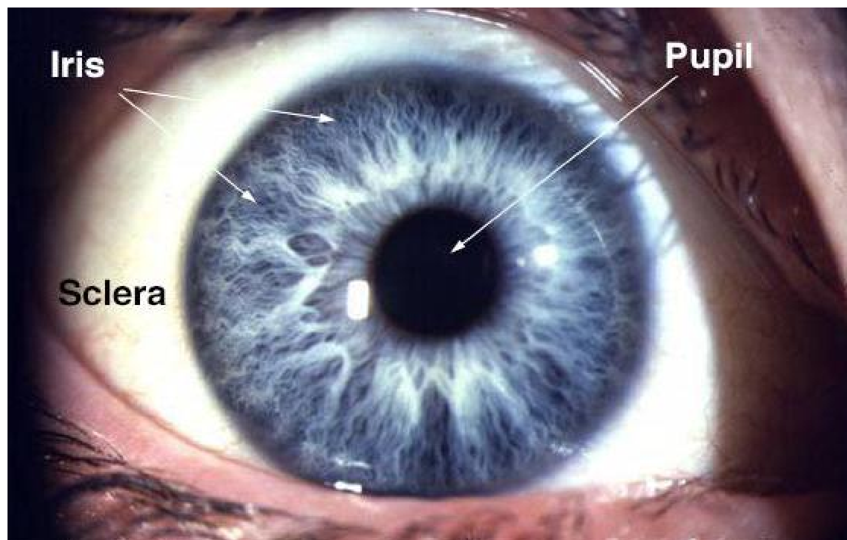


Figure 1.1: Image of the human eye with the three area of interest labeled. The limbus is the boundary between the iris and the sclera. Image From http://webvision.umh.es/webvision/imageswv/pupil.jpeg

Many gaze tracking systems make use of a head-mounted camera. Although these systems allow for user mobility, they tend to be more expensive than stationary cameras. In our implementation we use a stationary, table-mounted camera that is positioned in front of a computer monitor. The user is able to select one of four choices by simply staring at the corners of the computer screen.

Shown below in Figure 1.2 is the program flowchart for our system. Each of the modules will be explained in detail in subsequent sections.

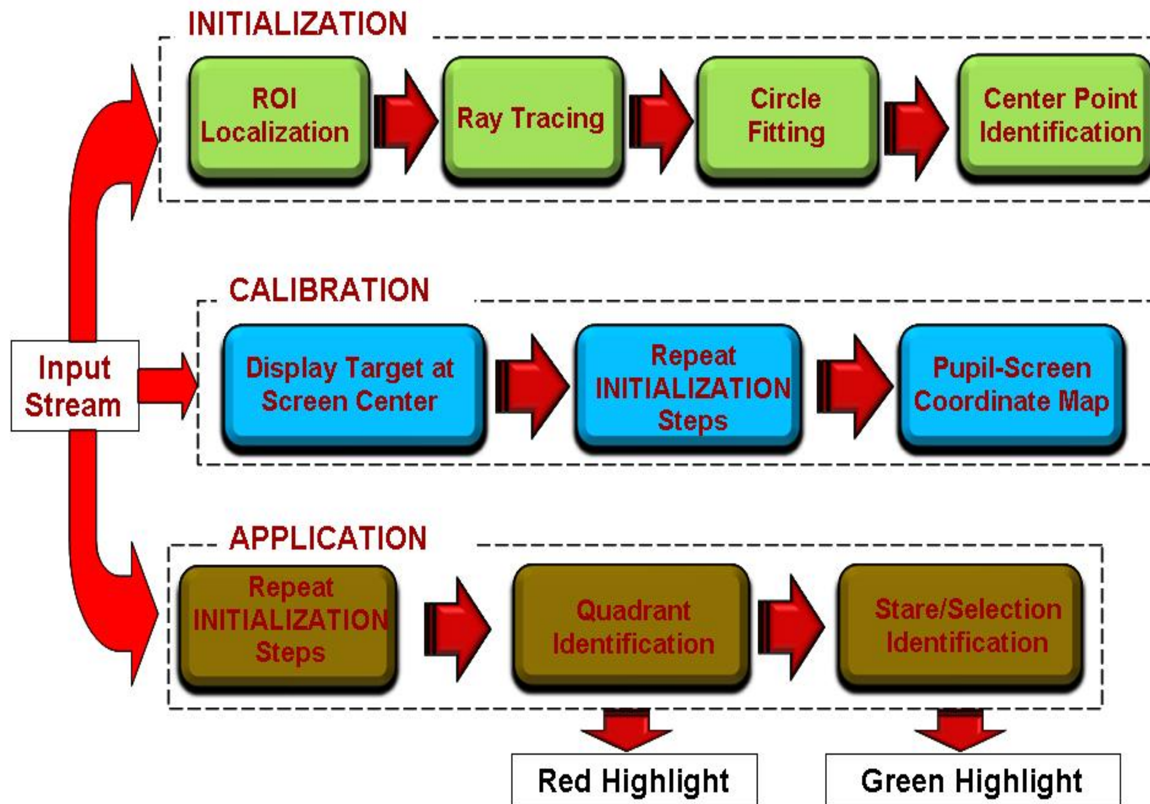


Figure 1.2: Program flowchart showing the three stages of program operation.

2. The Algorithm/Back End

2.1 ROI Localization

Other systems using head-mounted cameras provide a clear, high resolution image of the eye. Such an image is simply not possible with a stationary, table-mounted camera without putting unreasonable constraints on the user not to move. In fact, during testing of our system we observed users will often move relative to the camera even while attempting to stay absolutely still. To accommodate the problem of slight user movement, the camera must have a large field of view (FOV). This creates a new problem, however, as the camera's FOV may contain unwanted features which make tracking the eye difficult. We solve this problem by constraining the search space of the camera's FOV to a much smaller region of interest (ROI) centered around the eye.

We define the boundaries of the ROI with small pieces of colored tape placed on the rim of glasses worn by the user, as shown in Figure 2.1a. The camera captures images in the YUV colorspace, so we

choose to work in this colorspace to remove colorspace conversion overhead. The tape colors represent the four corners of the U-V plane, as shown in Figure 2.1b.

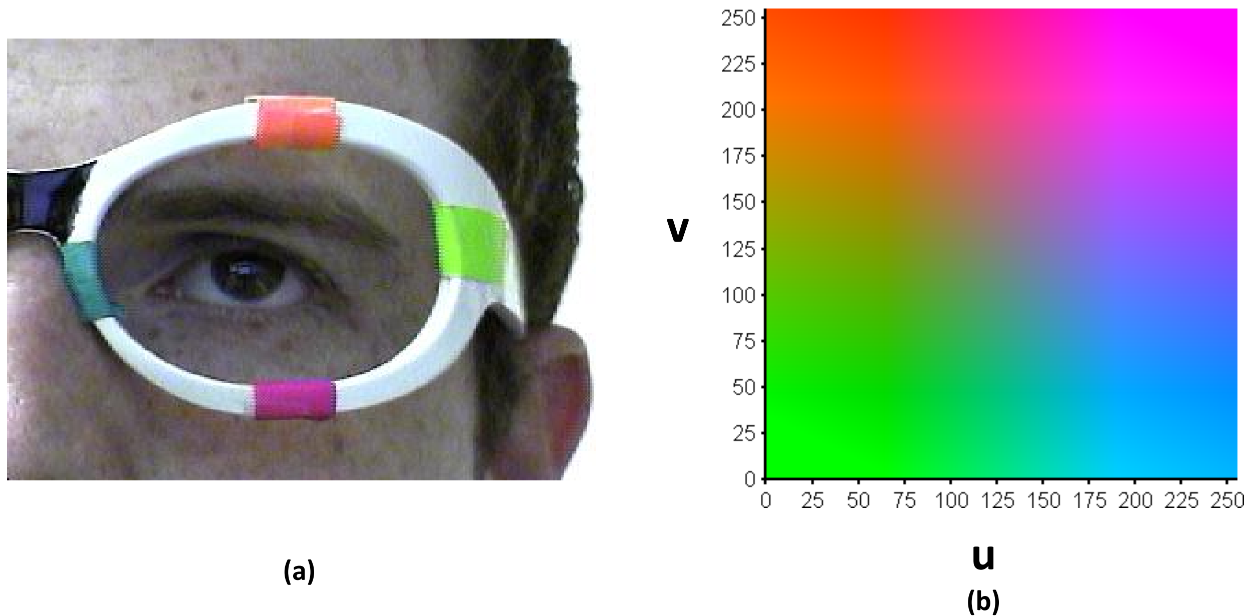


Figure 2.1: (a) Example of glasses with colored tape defining ROI and **(b)** U-V plane (with Y fixed at 127). Note how the four colors of tape represent the four corners of the U-V plane.

Training data consisting of 55 images was captured from the DSK board. Using MATLAB, class labels were applied to each of the four regions of tape for all training images. Figure 2.2a shows the physical distribution of pixels for the four classes and background pixels from the training data. Analysis of these distribution suggests they can be separated with linear boundaries as shown in Figure 2.2b. Linear boundaries require very little computational overhead, and thus are desirable.

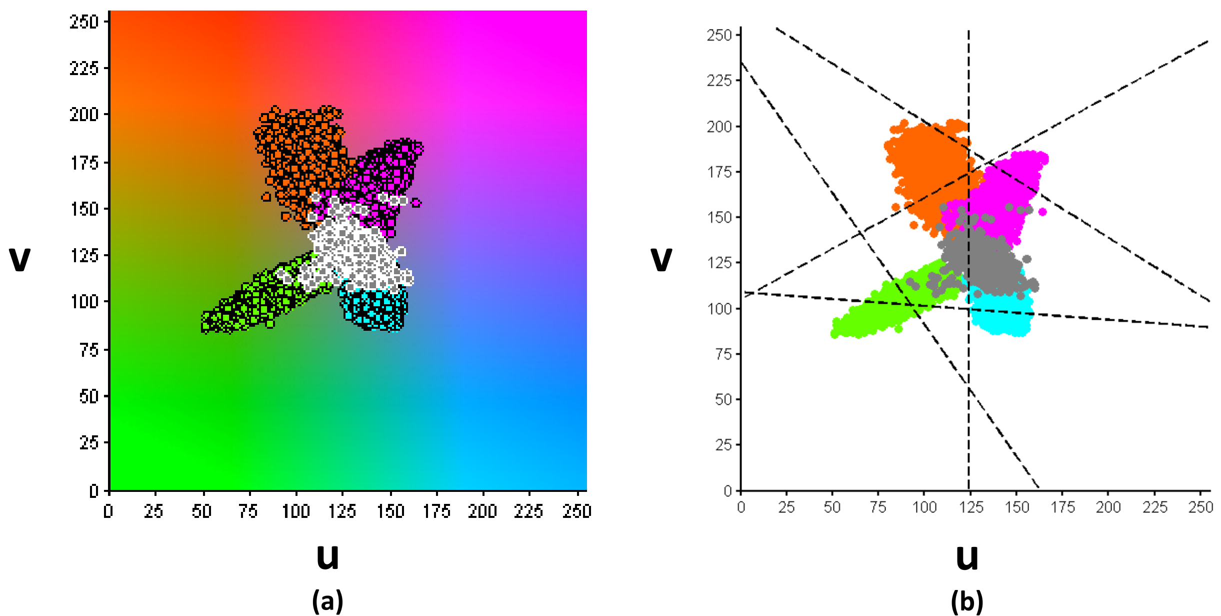


Figure 2.2: (a) Physical distribution of pixels for tape (colored circles) and background (white circles). **(b)** Linear boundaries separating the classes, as determined by LDA.

Fisher's Linear Discriminant, otherwise known simply as Linear Discriminant Analysis (LDA), was used to find the boundaries separating the classes. In the two-dimensional case, LDA attempts to find a vector $\boldsymbol{w}$ which maximally separates two classes when the points in the classes are projected onto $\boldsymbol{w}$. This is accomplished by minimizing the spread of projected points within a class while maximizing the distance between the projected means of each class. The *within-class* scatter matrix, S_W, is calculated from classes i=1,2 with number of points n_i using Equation 2.1. Equation 2.2 shows how the *between-class* scatter matrix, S_B, is calculated.

$$\mathrm{S_W} = \sum_{i=1}^{2} \sum_{n=1}^{n_i} (\boldsymbol{x}_n^i - \boldsymbol{\mu}^i)(\boldsymbol{x}_n^i - \boldsymbol{\mu}^i)^{\mathrm{T}}$$ **(Equation 2.1)**

$$\mathrm{S_B} = \sum_{i=1}^{2} (\boldsymbol{\mu}^i - \boldsymbol{\mu})(\boldsymbol{\mu}^i - \boldsymbol{\mu})^{\mathrm{T}}$$ **(Equation 2.2)**

In the above equations, $\boldsymbol{x}_n^i$ is a point in class i, $\boldsymbol{\mu}^i$ is the sample mean of class i, and $\boldsymbol{\mu}$ is the sample mean over all points from both classes.

The measure of *within-class* scatter in the projection is then give by $\boldsymbol{w}^T S_W \boldsymbol{w}$ and the measure of between class scatter in the projection is given by $\boldsymbol{w}^T S_B \boldsymbol{w}$. We would like to minimize $\boldsymbol{w}^T S_W \boldsymbol{w}$ while maximizing $\boldsymbol{w}^T S_B \boldsymbol{w}$, or equivalently, maximize the ratio $\boldsymbol{w}^T S_B \boldsymbol{w} / \boldsymbol{w}^T S_W \boldsymbol{w}$. The solution to this maximization problem, also known as the generalized Rayleigh quotient[4], is the solution to the generalized eigenvalue problem in Equation 2.3.

$$\mathrm{S_B}\boldsymbol{w} = \lambda \mathrm{S_W} \boldsymbol{w}$$ **(Equation 2.3)**

In the above formulation, only two classes are considered. However, there are actually five classes in our problem (including background pixels). We therefore find boundaries on a case-by-case basis. The first class is the class we are considering and the second class is the combination of all other classes. Once the best projection vector, $\boldsymbol{w}$, is found, the decision boundary is perpendicular to this vector and is chosen to reduce any misclassification of background pixels as class pixels.

Close examination of the decision boundaries in Figure 2.2b shows there will still be misclassification of pixels without the vertical threshold on u. The hard threshold on u is chosen to bisect the U-V plane to eliminate these misclassifications. Figure 2.3 shows how the combination of all decision boundaries dissects the U-V plane.

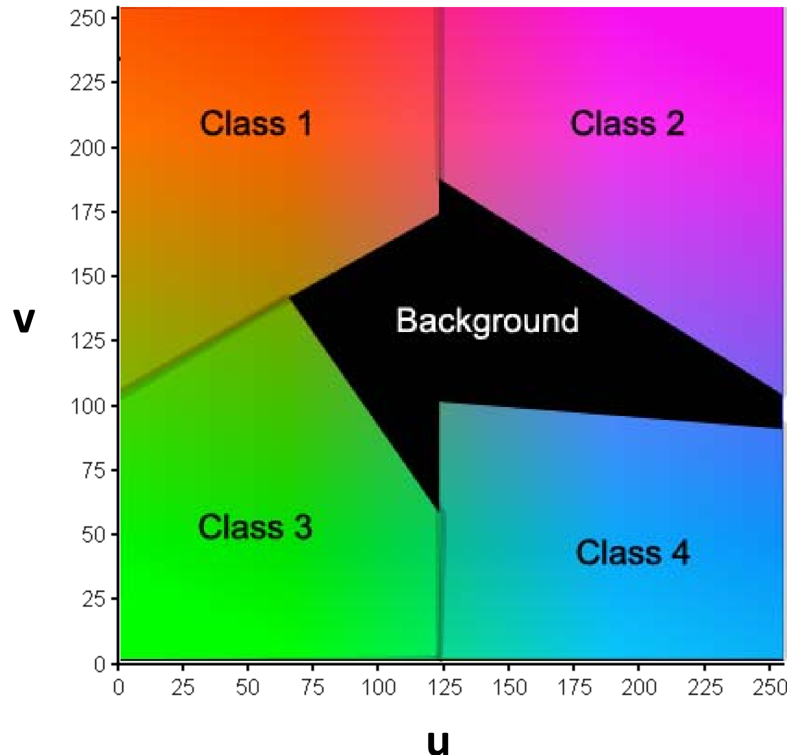


Figure 2.3: Result of combination of all linear decision boundaries from LDA plus hard threshold on u.

During system operation, each pixel being read into the DSK board is assigned a class value. If the class value is not background, the position of the pixel is added to an average of all pixels in that class for the current frame. In this way, average positions of the top, bottom, left and right boundaries of the rim of the glasses are calculated. This gives a much smaller region of interest to search with the ray tracing algorithm.

If a certain class has no pixel values assigned to it after the system has been initialized, the system will turn off. Such a situation occurs when the user moves out of the field of view of the camera or if the user removes the glasses. When this situation occurs, the program counts to 30 frames before exiting. This gives the user some time to move back into the field of view of the camera if they accidentally moved out of it.

2.2 Ray Tracing

In order to begin the ray tracing, which is based on the Starburst algorithm[1], we require an initial point. In the case of the first frame this point is set to the middle of the ROI, and in subsequent frames it is set to the center point found via circle fitting in the previous frame. Beginning at this initial point we trace rays radially outward along predefined paths. Unlike most implementations using ray tracing we utilize a fixed-point processor, and so we cannot make direct use of ray angles with respect to a Cartesian grid centered at the initial point. Therefore, instead of defining rays by angles we define them based on how many pixels we need to move both horizontally and vertically. For example, we predefine

a ray as a pair (1,2), which indicates that the next pixel that should lie on the ray path is 1 pixel to the right and 2 pixels above our current pixel. We define 20 rays per quadrant using pairs such as these and save these pairs in a file. We randomly permute the rays before saving them to a file in order to help with the circle fitting, as will be discussed in that section. At the beginning of program execution these rays are read in and saved in dynamically allocated arrays. It should be noted here that the Starburst algorithm proceeds by tracing rays only in the regions defined by -45° to 45° and 135° to 225°, as the iris-sclera boundary is blocked by the eye lids outside of this region. Yet when the initial point is away from the center of the pupil, many feature points can be found outside of the previously mentioned region. Therefore we did not restrict the ray directions and considered tracing rays over the entire range of 360 degrees.

In order to increase the resolution of ray tracing we include along the ray path defined by (*m,n*) pixels that are not necessarily *m* pixels over and *n* pixels up. Specifically, given that we are in the first quadrant, our initial point is at pixel (x_0,y_0), we are currently at pixel (x_N,y_N) and we are moving along ray path (*m,n*), we define the next point on the ray as,

$$(x_{N+1},\ y_{N+1}) = \left(x_N + 1,\ y_0 + \frac{m}{n} \cdot [x_N - x_0]\right) \qquad \textbf{(Equation 2.4)}$$

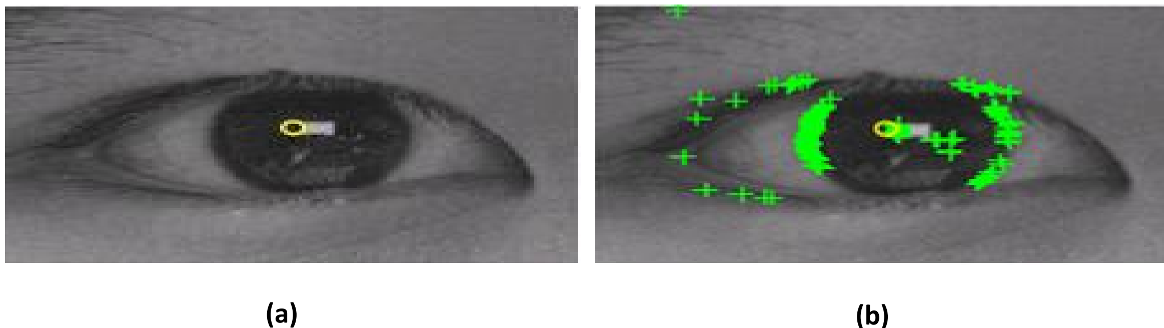
This improves the resolution of the ray tracing as we only move 1 or 2 pixels at each step of a ray trace.

The Starburst algorithm[1] utilizes 1 ray per angle degree, and thus traces 360 distinct rays. Given that we were not using floating point arithmetic or angles for ray paths, we require a method for defining which rays to trace. To do this we use rational fractions between 0 and 1. To keep the resolution high and computation time low we set the maximum denominator in our rational fractions to 6, and use only unique rational fractions (ie 2/3=4/6). This gives 11 unique rays between 1/6 and 5/6 degrees. Then, using Matlab, we find 12 equally spaced points between 1/6 and 5/6, and compute the distance between these 12 points and the 11 rational fractions, and choose the 9 unique rational fractions that are closest to the 12 points. We then mirror these rays over the 45 degree line and add rays at 0 and 45 degrees to obtain 20 rays for the first quadrant. All of these rays are used in the other quadrants with the signs of the individual paths adjusted accordingly. Although we trace significantly fewer rays than are traced in the Starburst Algorithm[1] we are still able to maintain a high level of performance.

While we trace each ray we check the gradient at each pixel along the rays. If the gradient at point N on the ray is greater than 15 (a predefined, empirically determined threshold) then we label point N a candidate feature point. During an earlier iteration of our implementation we labeled this point as a feature point. However, we later realized that we labeled too many feature points that were inside the pupil/iris. This is due to the corneal reflection that is clearly visible using visible spectrum illumination, which can be seen in Figure 2.4a. These false feature points greatly influence our circle fitting, causing an underestimation of circle sizes. Figure 2.4b shows how many false feature points are found on the corneal reflection in a typical frame. To overcome this limitation we add a correction portion of the ray tracing code. Specifically, if a candidate feature point is found, we continue along the ray for 5 more

points and if another candidate feature point is found we ignore both candidate feature points and continue tracing along the ray. This greatly reduces the number of false feature points due to the corneal reflection. Even though we cannot get rid of all of these false feature points, the vast reduction in number reduces their influence on the circle fitting.

As ray tracing and feature point recording proceeds we require a stop condition for ray termination. We stop each ray when either (a) there are 2 feature points found on the ray, (b) when we are outside of the middle 80% of the ROI (either horizontally or vertically) or (c) when we travel more than $2r^2$ from the initial start point. Condition (b) is used in order to avoid feature points too on the glasses and condition (c) is used in order to avoid feature points on the eyebrows. Although conditions (b) and (c) seem to be redundant they are in fact solving 2 different problems. If we only use the radius condition then it is possible that the feature points will slowly drift to the glasses and never come back to the iris. However, if only the ROI condition is used then there will be erroneous circles due to feature points found on the eyebrows. Therefore, both conditions are necessary for our implementation.



(a) (b)

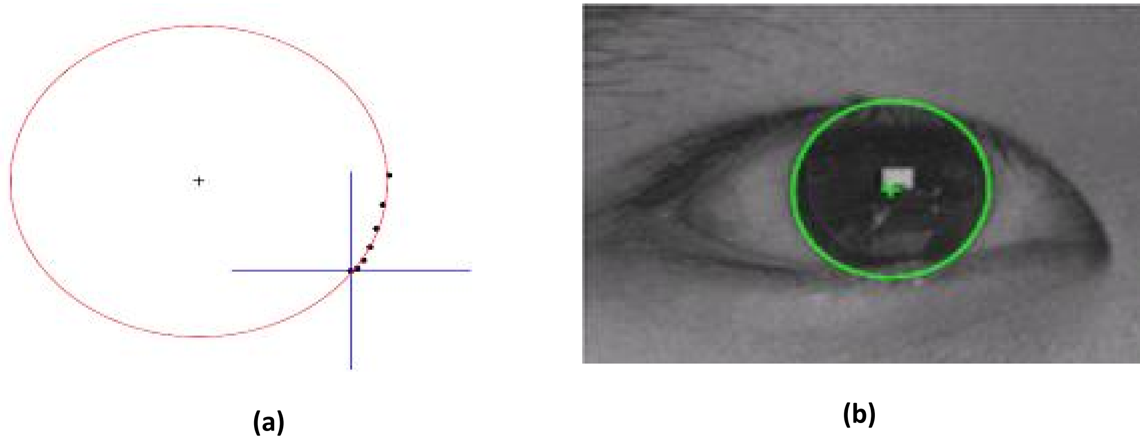
Figure 2.4: (a) Initial point[1]. Notice the corneal reflection (bright spot) located to the immediate right of the initial point. **(b)** Sample feature points[1] are marked with green crosses. Notice that there are a number of false feature points due to the corneal reflection.

Each predefined ray is traced in the first quadrant, followed by the third, second and then fourth quadrants, respectively. This is done in order to maximize the distance between adjacent feature points in order to help with the circle fitting. With a total of 80 rays and a maximum of 2 feature points per ray we can find a maximum of 160 feature points. The actual number of feature points found varies from 10 or less when the initial point is not on the iris to over 110 when the initial point is on the iris.

2.3 Circle Fitting

After finding all of the feature points we use a least-squares circle fitting algorithm[2], which computes the radius and center point of the circle that best fits the data in the mean squared sense. It should be noted that many gaze tracking algorithms use ellipses to fit the iris shape. We use circles for computational simplicity, and do not see a significant degradation of the resulting fit. First, in order to fit any circle we require that at least 20 feature points are found. We use 32 feature points to fit each circle. In order to do a least-squares fit we need to find the horizontal and vertical means and variances of the feature points. Figure 2.5a shows a circle fit to only 5 points. Since we use a fixed-point processor, it is necessary to adapt the algorithm so that divisions are not performed until absolutely necessary.

Also, we do not make use of the math library's square root function and so we simply record the radius squared instead of the radius. For each circle we store the horizontal and vertical center points as well as the radius squared. There can be anywhere from 0-5 circles found for each frame. Although most of the time the circles are similar, in some cases they are significantly different due to false feature points.



(a) **(b)**

Figure 2.5: (a) Sample circle fit with only 5 feature points[2]. **(b)** Example circle fit of the iris[1]. The green cross marks the center point of the circle.

After fitting circles to subsets of the feature points it is necessary to choose the best one. We use the sum squared distance from the circle to all of the feature points as our metric. We exclude any circle that had any point within 10% of the ROI boundary as well as any circle whose radius squared was either greater than 6 times the initial radius squared or less than .25 times the initial radius squared. This helped get rid of erroneous circles that would otherwise have been chosen as the best circle by sum squared distance minimization alone. Figure 2.5b shows an example of a good circle fit to the iris.

There are many frames for which no valid circle is found. This occurs either because the circles are the wrong size, there are less than 20 feature points or the circle is too close to the edges of the ROI. In these cases we increment a no circle counter which is subsequently used to re-initialize the system if its value reaches 30.

As mentioned in the previous section, the rays to be traced were randomly permuted before saving to the file. In the original Starburst algorithm the RANSAC procedure was used to fit ellipses to subsets of the feature points. This algorithm chooses random subsets of the feature points and uses these in circle fitting. By randomly permuting the rays to be drawn, and by tracing rays in the first, third, second and fourth quadrants sequentially, we are able to avoid randomly sampling the feature points since they are already quasi-randomly ordered. Given our implementation it is unlikely that the feature points used for the circle fitting were clustered in the same region. This leads to more robust and faster identification of the best circle.

We then use the center of the best fitting circle as our initial point for ray tracing in the subsequent frame. It should be noted here that this start point has a very high variance due to noise even when the user moves very little. In order to mitigate this effect, we keep a moving average of the initial point and radius squared over 9 frames. This stabilizes the center points and allows for better quadrant identification, which will be discussed in the next section. We found empirically that 9 frames for averaging is high enough for circle stabilization while still remaining low enough so that the circle did not lag too far behind the eye motion.

In order to obtain accurate circle fitting results, we need a good estimate of the eye diameter. To that end, we initialize the eye diameter to one third of the width of the ROI on the first frame for which we find a valid ROI. This value was determined empirically. After the first initialization step, during which the user stares at the center of the screen, we find a stable circle radius squared. At this point we update the eye diameter by looping through all column values 1 thru 240 and find the value that is closest to the radius squared. We do this by looping with index i and stopping when the radius squared is between $i \cdot i$ and $(i+1) \cdot (i+1)$. We then set the eye diameter to two times this value i. This method is robust because it relies on the stable circle that we find during initialization, which is explained in section 3.1.

It should also be noted that we save this eye diameter and the number of columns in the ROI at the end of initialization step 1. Then in every subsequent frame we scale the eye diameter with the number of columns in the current frame's ROI. Specifically, at the end of the initialization step 1 we have *initncols* columns in the ROI and the eye diameter is *initeyediam*. Then, if in any subsequent frames we have *ncols* columns in the ROI we set the eye diameter to (*initeyediam* · *ncols*)/*initncols*. We do this so that when the user moves closer to or farther away from the camera the eye diameter and therefore the search space used in the ray tracing module scales with the ROI size. Also, we choose to base all of this scaling on the number of columns in the ROI rather than the number of rows because the left and right boundaries of the ROI are more stable from frame to frame than are the top and bottom boundaries.

2.4 Center-point Identification/Quadrant Mapping

Getting an accurate reference point is crucial because it will be used later to decide the quadrant in which the user is looking. So during the initialization stage, when the user is looking at the crosshair displayed at the center, we get a gaze point (center of the circle) through ray tracing and circle fitting steps. Here we define a bounding box of 5×5 pixels and as long as the center point of the circle lies inside this box, we increment a counter. The purpose behind the bounding box is to take into account the background noise and user movement during the initialization step. When this counter value reaches a certain level (count of 200 in our case), we are confident of getting an accurate reference point. At this time the initial diameter, *initeyediam*, is estimated as explained in the previous section. The counter is reset to zero if the center of the circle lies outside the bounding box. This reference point is used to determine the quadrant in which the user is looking.

The same process is applied while registering a stare point. Again a bounding box is considered and a counter is incremented as long as the center of the fitted circle lies within the box. If the center of the circle lies outside the bounding box, the counter is reset to zero. When the counter value reaches 15, we treat it as a valid stare point.

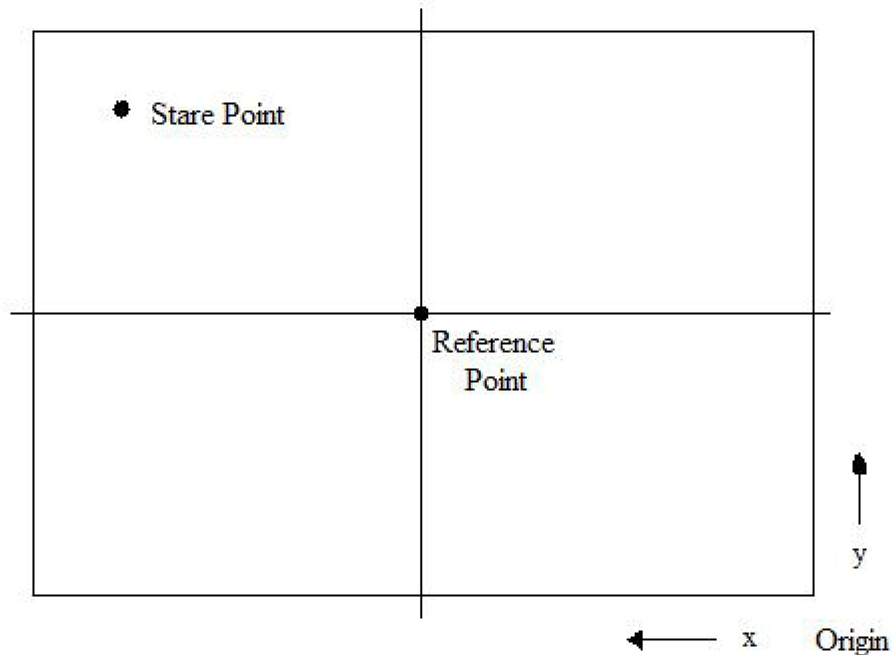


Figure 2.6: Quadrant mapping showing reference point determined during initialization. The user is looking in the second quadrant, as shown by the stare point.

Now at this stage, as of current frame, we have both the reference point as well as the point at which the user is looking at. The next step is to take a decision about the quadrant in which the user is looking. For that we consider the x and y co-ordinates of the stare point and compare them with those of the reference point. Considering the lower right corner of the image frame as the origin (see Figure 2.6), if the x co-ordinate of the stare point is greater than that of reference point, the user is looking either in the second or third quadrant. The y co-ordinate is considered at this step to distinguish between these two. If the y co-ordinate of the stare point is greater than that of the reference point then user is staring into second quadrant, else in the third quadrant.

In order to make the system robust to the user movement, this reference point is continuously updated with respect to the user movement. For that purpose, the original reference point is shifted to the new reference point by taking into account the difference between the original center of ROI and current center of ROI. This difference will give us the actual displacement of the user, which when added to the original reference point will generate the new updated reference point. So at every frame, the decision is made based on the latest updated reference point.

3. Front End Application

The front end application provides a menu to the user in order to make a selection. User just needs to look at the menu to select it. Current front end application has 4 menu boxes displayed in four quadrants. So when user looks at one particular box, it turns red, thus giving feedback to user about the stare point. If the user continues to look at the menu for a longer time – 2 seconds, the menu item turns green signifying its selection. Using front end application is subdivided into 3 major steps. The following section explains these steps. Step-by-step instructions for running the system are included in the readme file packaged with the code.

3.1 Using the Front End

This is the first important step which helps initialize the whole system. During this step, the user is presented with a crosshair at the center of the computer screen as seen in Figure 3.1. User needs to look at this cross hair till the end of initialization. The end of this stage is marked by displaying the menu boxes and removal of the center crosshair. The user should not move during this step as we are trying to get an accurate reference point and a stable radius for iris.

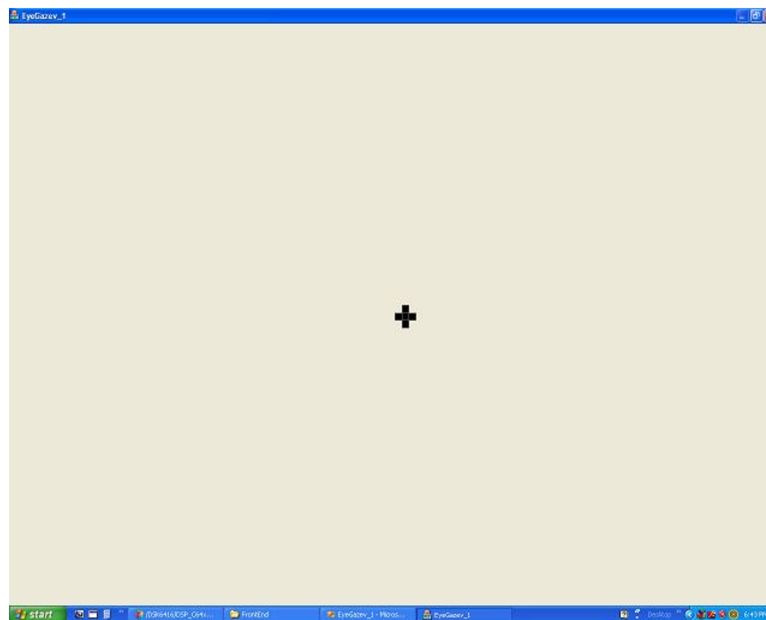


Figure 3.1: Crosshair shown in the center of the screen during initialization.

After initialization, four menu boxes appear on the screen. At this point, the system is calibrated and thus will follow the user eye gaze. The system will provide a feedback to the user about the stare point by highlighting the menu item by red color. The user can select a menu item by staring at it for a certain period of time: 2 seconds in our application. The menu item turns green upon continuous staring. Figure 3.2 shows an example of the upper left menu item selected.

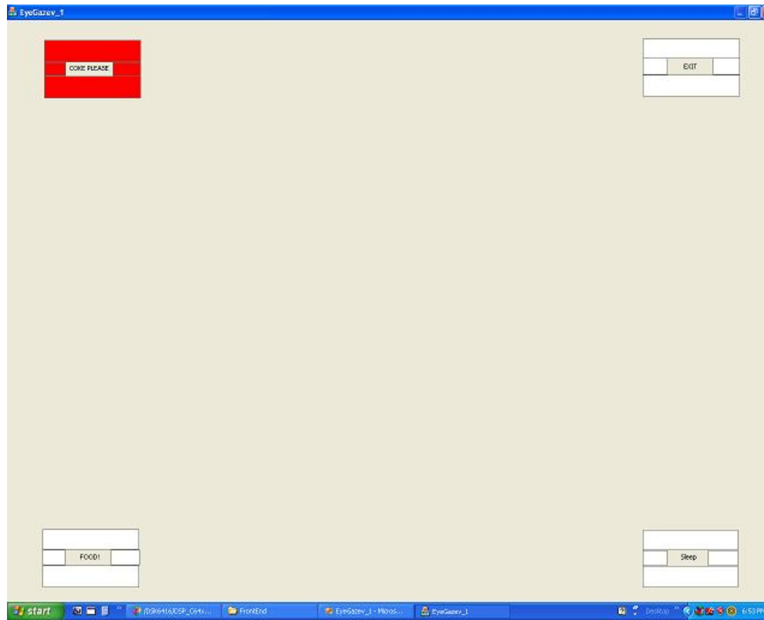


Figure 3.2: Upper left menu item selected. Selection will turn green if stared at for 2 or more seconds.

The application provides a means to quit the system by looking at the top right menu item, as in Figure 3.3. Upon its selection, the front end application exits. The program running on the DSK also breaks out and halts.

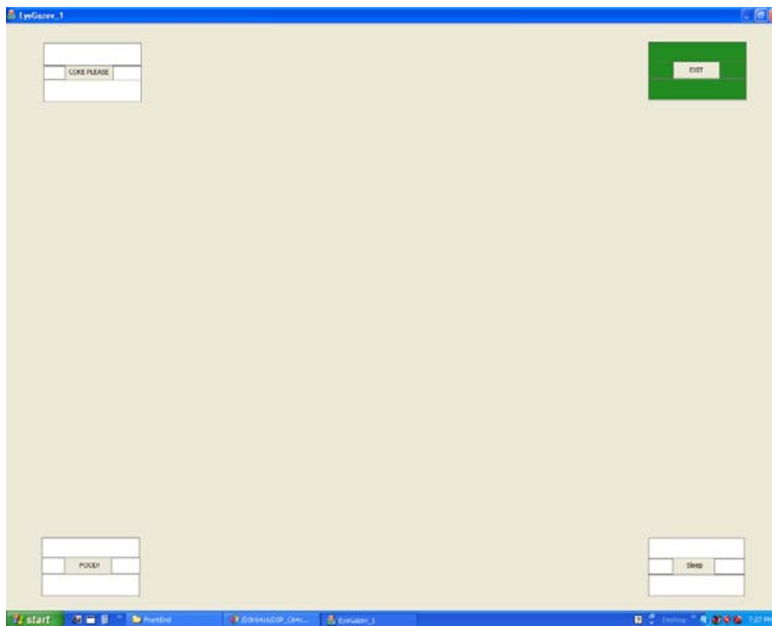


Figure 3.3: Exit selected. This causes the program to exit and halt.

The system knows when to exit by incrementing a counter when user looks in that particular quadrant. So when the user selects an exit menu, the counter has already incremented to a predefined level which denotes the time required for the selection. One more way to exit the system is to take off the glasses or move out of the small screen provided for the user. If the ROI is not found for 30 consecutive frames (empirical value), that is treated as the system exit point.

The feature of re-initialization is provided in order to avoid large user movement. When user moves a lot, the system goes back to the initialization phase with a cross hair at the center of the screen. The threshold values used are 50 pixels for horizontal movement while 30 pixels for vertical movement. These values are found out empirically. The value for vertical shift is lower as the vertical distance is lower than the horizontal distance. So when user moves more than 50 pixels in horizontal direction or 30 pixels in the vertical direction, the system re-calibrates or re-initializes again.

3.2 Visual C++ Application Description

The front end application is developed using Visual C++ and Microsoft Foundation Class library (MFC) technology. It runs on the PC and as a result needs to communicate with DSK using Real Time Data Exchange (RTDX) technology.

There are two main tasks to be performed by this application:

1.) Highlight the menu items on the screen depending upon the information sent by the DSK to this application.

2.) Continuously check for any incoming data sent by DSK. This data signifies the decision taken by DSP processor about where the user is looking at.

As a result the functionality of the front end can be described as – Check the data sent by DSK and depending upon this information, highlight the menu item on the screen. Hence, there are 2 threads running in the application. The first thread continuously polls the channel for the incoming data and updates a variable with this data. The second thread checks this variable continuously and depending upon its value, highlights menu item on the screen. A problem can occur when first thread is writing a data into the variable and at the same time the second thread is reading the value of that variable. This will cause erroneous behavior and in order to avoid that, mutex is used which forbids simultaneous access of the variable and avoids any unexpected results.

Once a decision is made by the back end as to which quadrant the user is looking at, this information must be conveyed to the front end so the selection can be shown on the screen. This is achieved by sending unique integers corresponding to every quadrant. This integer is read by the polling thread of the front end and stored in a variable. The second thread reads the value of this variable and paints the corresponding menu item with red color palette. Now if the user keeps looking at the same place for a longer duration, the front end will receive the corresponding integers for a longer duration and it changes the color palette from red to green selecting the menu item. Table 2.1 shows the integers sent and how they are interpreted.

Table 2.1: Interpretation of integers sent to front end application from back end.

Integer	Meaning
1	Initialization Stage
6	First quadrant selection
7	Second quadrant selection
8	Forth quadrant selection
9	Third quadrant selection
10	Re-initialization

It can be seen from the above table that in order to have a cross hair in the middle of the screen, the board needs to send '1' while numbers ranging between 6 to 9 denote a quadrant selection. Value '10' signifies the system re-initialization. These numbers are sent using the RTDX technology, explained in the next section.

3.3 RTDX technology for DSK- PC communication

In order to send these unique integers from board to PC, we need some robust mechanism with the help of which we can transfer the data. RTDX – Real Time Data Exchange mechanism facilitates this exchange of data. It uses the RTDX software library to transfer the data between host and target application. The advantage of using RTDX is that it achieves this transfer without interfering with the target application. The target application just needs to make function calls to the RTDX libraries API to pass the data to or from it. Consider Figure 3.4 for explanation of how the data is transferred.

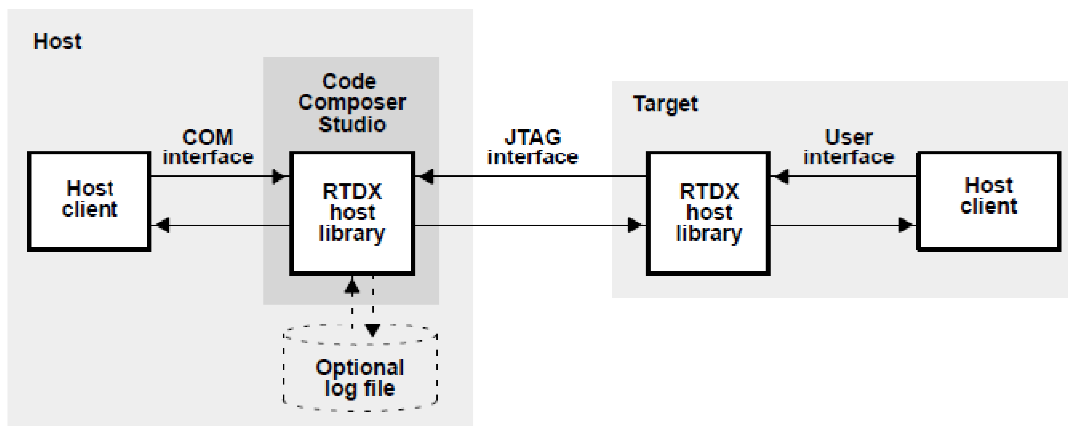


Figure 3.4: RTDX Block Diagram[3]

A channel is required to transfer the data between host and target. Hence, the first step is to open a channel for the communication. The sending party can then write the data onto this channel while receiving party can read this data. In our application, we need only unidirectional communication with the DSK sending the decision integers to the PC.

When a data needs to be transmitted, it gets stored in the target buffer defined in RTDX target library. JTAG interface is used to send this data to the host. RTDX host library receives this data from JTAG interface and stores it in either a log file or a memory buffer. In our application in order to minimize the delay, the option of memory buffer is used instead of log file.

The advantage here is when the processor wants to write onto the channel, it can just raise an interrupt and thus can write the data. On the other hand, the front end application needs to poll the channel continuously. Hence, the computational overhead is not at all on the DSK board and hence, all the crucial computations can be done in real time without any delay. We have used DSP/BIOS along with RTDX as it provides certain advantages.

DSP/BIOS is a scalable real time kernel which facilitates scheduling and synchronization. We can make use of hardware interrupts, software interrupts, tasks, etc which are provided with the DSP/BIOS. The usage of DSP/BIOS with RTDX causes the automatic handling of:

1.) Allocation and location of *.rtdx_text* and *.rtdx_data* sections. These are the buffers where the data is stored temporarily before sending it from host to target. DSP/BIOS automatically allocates these sections into the available memory area.

2.) Insert appropriate ISRs required for RTDX in interrupt vector table. The configuration file in DSP/BIOS reserves some interrupts for RTDX. This avoids the manual configuration of the interrupt vectors.

So in order to write the data onto the channel, an output channel is created, interrupts are enabled and *RTDX_write* command is used. On the other hand, in the front end application, an object of the class *IrtdxExp* is created and a channel is enabled along with specifying the processor, the C6416 DSK. The *ReadI2* method is used iteratively in order to read the data on the channel.

4. Design Difficulties and Future Direction

We faced a problem while mapping the screen coordinates to the iris center coordinates. Initially we planned to have finer tracking by placing initialization points at the center as well as at the four corners. We found out that when the user shifts his/her gaze from left corner to right corner, the center of the iris moves by 20 pixels in horizontal direction. On the other hand, when the user looks from top corner to bottom corner, the typical movement of the iris center is 12 pixels. As these values are very small, if the user moves by the slightest of the amount, these five reference points make no sense going further. If these values were larger, then the user movement might not be such a large problem.

One solution to this problem was to shift the reference points with the ROI. So even if the user moves after initialization, we still have correct reference points to make the decision. But the bigger problem was the non-linear mapping between the user movement and the corresponding compensation required for iris center movement. For that we require the distance as well as angle between the camera and the user – values not easily available. So due to this problem coupled with the small

distances, we decided to trade-off between fine tracking resolution and system robustness. As a result the tracking was made limited to four quadrant distinction with added robustness for user movement.

Another problem was the latency or delay for the communication between DSK and PC. The typical delay was found out to be 11 frames or around 300ms. But the delay is not noticeable as it is included in the time required to highlight a menu item. For example instead of asking user to stare for half a second, we can always ask him/her to stare for 1 second considering the delay. In that sense, the user will not feel if there is any delay.

Many applications exist for our system once the problem of limited tracking resolution is resolved. The application currently demonstrated by our system is its use as an accessible human-to-computer interface for the physically disabled. However, behavioral analysis is another area that could make use of our system. Car manufacturers may be interested in knowing where drivers look when driving. Marketing departments may like to know what advertisements catch a users eye and which they ignore. Similarly, psychological studies may benefit by monitoring how those with hyperactive disorders, such as Attention Deficit Disorder, use a computer.

A few changes need to be made before our system can be considered viable for sale. The first of these changes is solving the problems of limited tracking resolution and movement compensation, as stated before. In the future, our system might also benefit from being adapted to use other, cheaper cameras. If this is possible, the DSP chip and camera could be integrated into a small, cheap package that could be widely distributed. Finally, our system is considered “intrusive” because the user must wear glasses. More sophisticated methods of finding the ROI would allow us to dispose of the glasses, and thus make our system non-intrusive. These are the main changes that should be considered for future work on our system.

5. References

1. Li and Parkhurst, "Open-Source Software for Real-Time Visible-Spectrum Eye Tracking", *The 2nd Conference on Communication by Gaze Interaction,* Sept. 4-5, 2006

2. Bullock, R. “Least-Squares Circle Fit”, from http://www.dtcenter.org/met/users/docs/write_ups/circle_fit.pdf

3. Harish Thampi S/ Jagan Govindarajan, “DSP/BIOS, RTDX and Host-Target communications” Texas Instruments application report SPRA895

4. Balakrishnama and Ganapathiraju, “Linear Discriminant Analysis – A Brief Tutorial”, from http://lcv.stat.fsu.edu/research/geometrical_representations_of_faces/PAPERS/lda_theory.pdf