Issues in the Design and Development of Experimental Software for Use With an Eye Tracking System

Tim Halverson
Department of Computer and Information Science
University of Oregon
Eugene, Oregon
(541) 346-1124
thalvers@cs.uoregon.edu

Anthony Hornof
Department of Computer and Information Science
University of Oregon
Eugene, Oregon
(541) 346-1372
hornof@cs.uoregon.edu

ABSTRACT

The data collection software that ships with eye tracking systems must often be extended or integrated with additional software developed by the experimenter to present the appropriate visual stimuli to participants in an experiment, and to record the participants' responses (other than eye movements). Though this programming task is routinely undertaken, there has been little or no attempt to record the details of this software development process to facilitate the re-use of code or design patterns by other experimenters. This paper attempts to provide such guidance. Specifically, the paper discusses the details involved in converting the LC Technologies Eyegaze System's Windows-based API (application programming interface) to a Macintosh-based API. This conversion allows the eye tracker to collect eye movement data while the participant in the experiment interacts with the Macintosh, and allows the experimental software running on the Macintosh to have instant access to eye movement data. The paper discusses numerous issues that will arise when designing, developing and testing software for use with an eye tracking system.

Keywords

Experimental design, Eye tracking, eye-tracking software, software design, software development

1. INTRODUCTION

Eye-tracking technology has progressed to the point that "off-the-shelf" units are quickly deployable and come with software that can be readily modified by an experienced programmer to meet specific data collection needs. However, while literature exists on the methods used by eye-tracking devices, there is little guidance available on the advantages and disadvantages of various eye-tracking equipment and their configurations. In addition, there is a shortage of literature discussing experimental software design for eye-tracking studies.

This paper will add to such discussions in three ways. First, we briefly describe and assess the Eyegaze system from LC Technologies. Second, we present a reference for porting Eyegaze client source code, along with a brief description of a port of the Eyegaze client software to the Macintosh OS. Finally, we offer some proposals for the design of eye-tracking experimental software design. Though the Eyegaze System is emphasized, we expect the findings to be generally useful to the human-computer interaction (HCI) and eye-tracking communities.

Copyright © 2001 by the University of Oregon. All rights reserved. Department of CIS Technical Report 02-02.

2. THE LC TECHNOLOGIES EYEGAZE SYSTEM: DESCRIPTION AND ASSESSMENT

The Eyegaze System from LC Technologies (http://www.eyegaze.com) is straightforward to setup "off-the-shelf", and offers many convenient features. Here we specifically discuss its use in a client-server configuration. In the client-server configuration, the Eyegaze System is the "server". The system consists of a MS-Windows computer, monitor, camera, eye-position video monitor, and various image processing and gaze fixation software. A second "client" computer is used to actually display the visual stimuli used in the experiment. As shown in figure 1, the video camera that monitors the eye movement is connected to the Eyegaze computer. The client and Eyegaze system communicate with each other during the course of an eye tracking session. Additional details on this configuration and other aspects of the Eyegaze system may be found in [7].

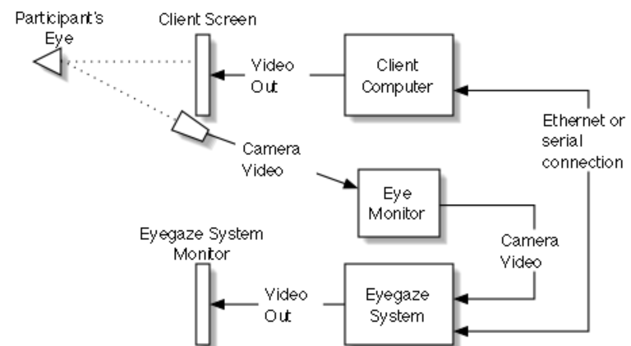

Figure 1. The Eyegaze System in client-server configuration. (Derived from [7].)

2.1 Eyegaze System Description

The Eyegaze System is a video-based eye tracking device for use in real-time gaze analysis and recording of gaze data. This analysis is done with a specialized video camera, mounted under a computer screen, which shines an infrared light on the

eye and captures an image of the eye 60 times a second. Specialized video-processing software uses the infrared images of the eye to determine the coordinates on the screen at which the eye is aimed or point of regard (POR). This is done using the pupil center corneal reflection method [11], which uses the vector between the center of the pupil and the corneal reflection to determine the POR. In addition, the video-processing software determines the diameter of the pupil and the location of the eyeball center.

In order to determine the POR accurately, a calibration must be performed. The default calibration presents nine points that must each be fixated on in turn. A client application may initiate this calibration to take place on the client screen (see "Eyegaze Code" section below).

The client-server configuration gives the added benefits of gathering the data from and using the data on a computer platform other than Windows 2000, which is the Eyegaze System's native platform, and allowing an operator to remotely monitor the operation of the eye tracking without interfering with the participant. Ethernet or serial connections can be used to for communication between the Eyegaze System and a client. Ethernet may be preferred as it is faster and platform-independent packet protocols are well-established. In the client-server configuration, the eye position data can also be sent to the client for the same purposes.

2.2 "Off-the-shelf" Use

We found the Eyegaze System to be straightforward to configure and utilize. The hardware was only marginally more difficult to configure than a normal PC. Utilizing the softwarethat ships with it, calibration and data gathering were also found to be uncomplicated. Our experience thus far is that preparation for gaze data collection takes two to five minutes. The calibration takes as little as 10 seconds and as long as two minutes, depending on how well the participants can be tracked. The majority of calibrations take between 15 to 20 seconds, near the average calibration time claimed by LC Technologies [7].

Perhaps the greatest advantage of the Eyegaze software is that it not only comes with a full-featured application program interface (API) for developing client applications under Windows, but it also comes with a majority of the source code. In this way, the Eyegaze System is especially customizable and allows porting of client code to other platforms besides Microsoft Windows. The shortcoming of the Eyegaze System's software is its lack of "off-the-shelf" utility is its software. It is a disadvantage in that the software is not as elaborate as some packages that run with other eye-tracking equipment (e.g. SMI and ASL). For example, the Eyegaze System does not come with features such as video playback or statistical analysis. The next section discusses one way in which the availability of the Eyegaze source code can be an advantage.

3. PORTING AND CUSTOMIZATION OF THE EYEGAZE CODE

The Eyegaze System ships with a majority of its source code allowing an experienced programmer to customize aspects of its operation and to convert or "port" the client code for use under other operating systems (e.g. Macintosh, Linux). While other eye-tracking systems (e.g. SMI) may come with a software development kit (SDK), no other system from a major eye-tracking manufacturer offers the source code. The following is a brief reference for converting the source code for use with a client, and then a brief description of a port to the Macintosh OS.

3.1 A Reference to Porting the Eyegaze Code

To develop a client application, only a subset of the source code shipped with the Eyegaze System is required. Therefore, a programmer unfamiliar with the Eyegaze source code would have to study the code extensively before finding the proper code to port. This section should aid in finding those bits of code that are required.

3.1.1 Maintaining a Consistent API

Every attempt should be made to maintain an API consistent with LC Technologies API. Maintaining a consistent Eyegaze API between multiple operating systems and various versions of the code on the same operating system will potentially benefit many users of the Eyegaze System. The familiar API of another version will aid any programmer already familiar with one version of the Eyegaze code. A consistent API will likely result in decreased development time when using the new Eyegaze code and will increase the amount of code-reuse, and help to maintain a common understanding of program functionality when discussing issues with LC Technologies staff and other programmers using the system.

Certain minor changes to the API may be unavoidable. For example, the function `EgCalibrate` ("Eg" stands for Eyegaze) takes as an argument a data type of `HWND`. `HWND` is a Windows specific data structure for a window handle (pointer to a pointer). `HWND` will most likely need to be changed to another data type for non-Windows implementations, or possibly removed altogether in an object-oriented implementation to promote encapsulation and to maintain data abstraction. Although certain changes may be needed, programmers are encouraged to maintain the basic function names and roles of the API functions that are declared in `EgWin.h` and also includes `QueryEyegazeSystem` in `Suprt2PC.h`.

3.1.2 Eyegaze Source Files

What follows is a brief summary of the Eyegaze source code files, written in the C programming language, which contain the code that must be modified in order to use the Eyegaze on a non-Windows systems. All of t

`EgWin.h`: Eyegaze Windows header. This file contains the function prototypes for the Eyegaze API, structs used to contain key control data and gaze data. In addition, this file contains constant definitions for messages between the client and server. These messages are detailed latter in this paper.

`EgWin.c`: Eyegaze Windows implementation. This file contains definitions of the API functions. Each function calls a utility function contained in one of the files listed below.

`LctCalLib.h`: LC Technologies Calibration Library header. Only the function definition for `LctCalibrate` is required from this file.

`LctColor.h`: LC Tehcnologies color header. This file contains the constant definitions for the values the server may send to represent colors (RGB represented in hexidecimal).

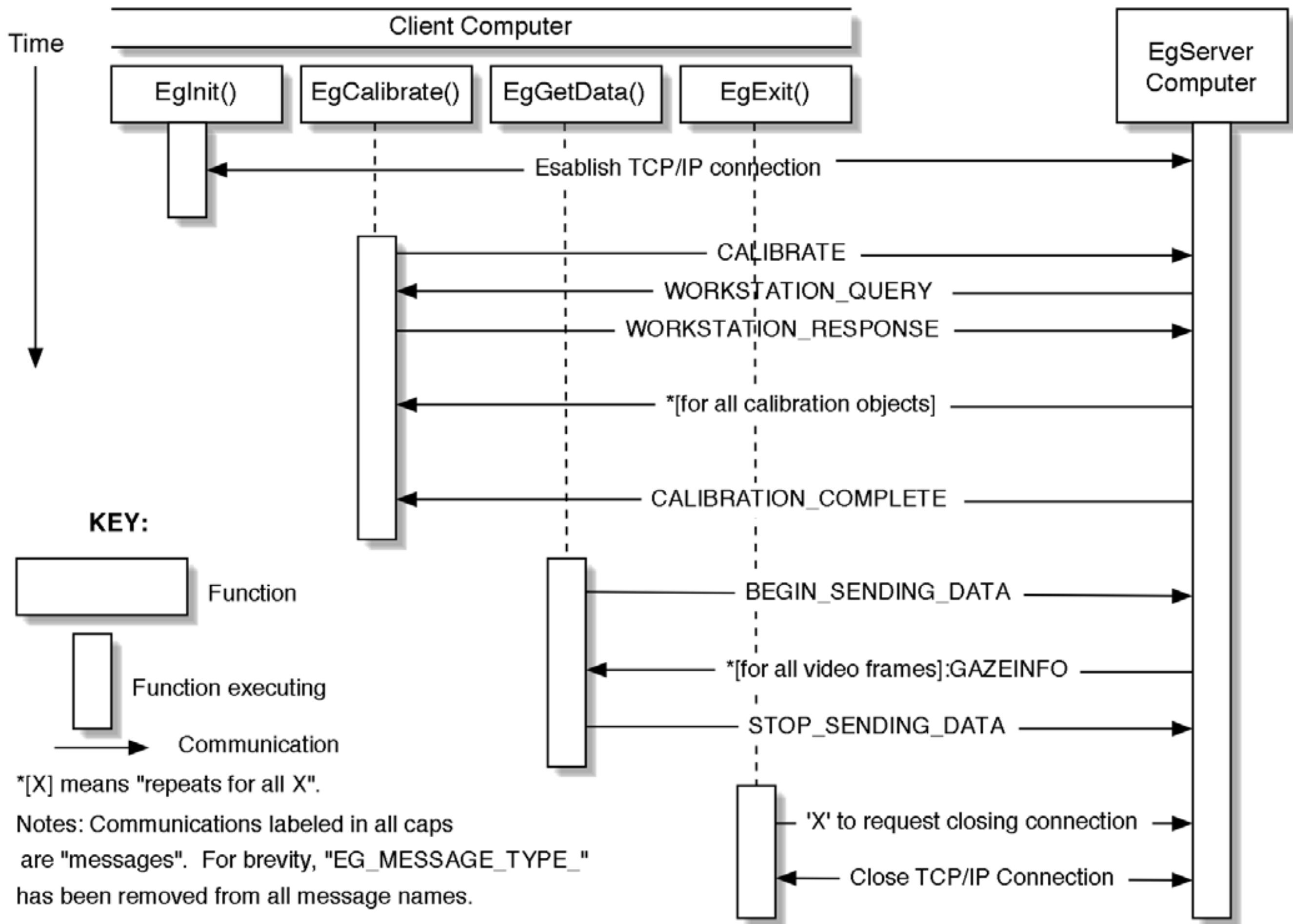

Figure 2. A typical sequence of messages passed between an Eyegaze client and server.

`LctFont.h`: LC Technologies font header. This file need not be ported unless fonts will be handled in the same manner as the Window implementation.

`LctSupt.h`: LC Tehcnologies support functions header. Declarations for utility functions defined in `WinSupt.c` and `Winser.c`.

`LctTypDef.h`: LC Technologies type definition header. Definitions of many data types used throughout the source code.

`Suprt2PC.c`: Support functions for PCs. The functionality of the all functions defined in this file are required, but the exact structure need not be maintained.

`Suprt2PC.h`: Support functions for PCs header. Not all of the functions declared in this file are required. Those concerned with mouse position and clicking may be ignored.

`WinSer.c`: Windows serial communication functions. The functionality of the many of the functions contained in this file is needed. Those that are not required are `passiveTCP` and `passivesock`. In addition, any code in `lct_socket_open` concerned with the constant `SOCKET_SERVER` is not required for a client.

`WinSupt.c`: Windows support functions implementation. The functionality of all of the functions in this file is required. However, the way in which these are implemented will most likely differ in a non-Windows environment.

3.1.3 Messages

Communication between the server and client is implemented using “messages”. Understanding the message constants defined in `EgWin.h` is the key to deciphering the Eyegaze code and finding the code that must be ported. The client uses these constants to request actions from the server, and the server uses them to initiate actions on the client. Figure 2 shows a typical progression of these messages.

3.1.3.1 Sending messages

All messages are sent from the client to the server through a common utility function, `LctSendMessage`. The purpose of this function is to determine the length of the message and create a buffer array of characters representing the following: the message length, the message, and additional data that must sometimes be sent with the message. The utility function `lct_send_buffer` sends this array to the server.

3.1.3.2 Receiving messages

There is no utility function for receiving messages from the server. The process required to receive messages is as follows: The first three bytes of each packet received from the server are read individually. The first three bytes indicate the length of the remaining message. This length is then used by the utility function `lct_socket_read_buffer` to write the remaining bytes of the incoming packet to a buffer array of characters. The first character in this array is the message type. Message types are defined below. If the message carries other information with it (e.g. eye position data), the remainder of the array contains this information.

3.1.3.3 Message definitions

Listed below are descriptions of messages the Eyegaze client code must utilize. These messages are grouped by functionality. Calibration messages are those messages used only during a calibration. Communication messages are those messages that may be used any time after a calibration. In the interest of brevity, certain conventions have been used in the descriptions. First, the message declarations are shown with "`EG_MESSAGE_TYPE_`" removed from the beginning of each (e.g. `EG_MESSAGE_TYPE_CALIBRATE` is listed as `CALIBRATE`). Each message is defined as either a message sent from the server to the client, and the specific client function receiving the message is indicated (such as Server->`client_function_name`), or as a message sent from the client to the server, and the specific client function sending the message is specified (e.g. client_function()->Server).

<u>Calibration Messages</u>

`CALIBRATE`: `EgCalibrate`->Server. The client requests a calibration.

`WORKSTATION_QUERY`: Server->`EgCalibrate`. The server acknowledges a calibration request.

`WORKSTATION_RESPONSE`: `EgCalibrate`->Server. The client responds to the workstation query. The dimensions of the client screen (in millimeters and pixels) and characteristics of the calibration window (dimensions and offset in pixels) are sent with this message.

`CLEAR_SCREEN`: Server->`EgCalibrate`. The server instructs the client to clear the calibration window.

`SET_COLOR`: Server->`EgCalibrate`. The server instructs the client to set a variable to a color predefined in `LctColor.h`. This color is used for objects in the calibration window. The hexadecimal number representing the color is sent.

`SET_DIAMETER`: Server->`EgCalibrate`. The server instructs the client to set diameter of the crosshairs or the radius of a circle that will displayed in the calibration window for the participant to fixate during calibration.

`DRAW_CIRCLE`: Server->`EgCalibrate`. The server instructs the client to draw a circle in the calibration window at a specified location.

`DRAW_CROSS`: Server->`EgCalibrate`. The server instructs the client to draw crosshairs in the calibration window at a specified location.

`DISPLAY_TEXT`: Server->`EgCalibrate`. The server instructs the client to draw text in the calibration window at a specified location. The text would include feedback such as "Gaze is not consitent."

`CALIBRATION_COMPLETE`: Server->`EgCalibrate`. The server notifies the client that the calibration was successful. Gaze data should not be requested until a calibration complete message has been received for a participant.

`CALIBRATION_ABORTED`: Server->`EgCalibrate`. The server notifies the client that the calibration was aborted.

<u>Communication messages</u>

`BEGIN_SENDING_DATA`: `EgGetData`->Server. The client requests that the server start sending gaze data.

`GAZEINFO`: Server->`EgGetData`. The server indicates that gaze data is available. One frame of available gaze data is sent.

`STOP_SENDING_DATA`: `EgGetData`->Server. The client requests that the server stop sending gaze data. It is possible that one more frame of data will be sent following this message.

`FILE_OPEN`: `EgLogFileOpen`->Server. The client requests the opening of a file for data collection on the server. The name of the file is sent. In addition, an 'a' is sent to append to an existing file, or 'w' to request a new file.

`FILE_WRITE_HEADER`: `EgLogWriteColumnHeader`->Server. The client requests the writing of column headers to the file opened with the file open message.

`FILE_APPEND_TEXT`: `EgLogAppendText`->Server. The client to append text to the file opened with the file open message. The text to append is sent.

`FILE_START_RECORDING`: `EgLogStart`->Server. The client requests the recording of gaze data to the file opened with the file open message.

`FILE_MARK_EVENT`: `EgLogMark`->Server. The client requests an increment of the server event counter.

`FILE_STOP_RECORDING`: `EgLogStop`->Server. The client requests the stop of gaze data recording to the file opened with the file open message.

`FILE_CLOSE`: `EgLogFileClose`->Server. The client requests the closing of the file opened with the file open message.

`CLOSE_AND_RECYCLE`: The client requests a disconnection. The server closes the IP connection and recycles to wait for another connection. No current function in the original or ported API uses this message.

3.2 Eyegaze Client Code on the Mac

This is a description of Eyegaze code recently ported to the Macintosh OS. A serious effort was made to keep the Macintosh Eyegaze API consistent with the Windows Eyegaze API. However, some changes to the API were required, some because the target platform was Macintosh. As well, where as the Windows source code was written in a functional style using the C language, we chose to write the Macintosh code in an object-oriented style using C++.

The following description of our Macintosh Eyegaze API emphasizes the few places that its methods differ from the original Windows Eyegaze API. The Mac Eyegaze API includes the following class and methods:

`EgMac` – This is the main class, and the only class which must be instantiated in a client. The constructor now takes part of the original `EgInit's` functionality. `EgInit's` `stEgControl` struct argument is copied internally so that those variables set by the `stEgControl` argument can only be accessed through member functions.

`init` – This is largely equivalent to the original `EgInit`. It establishes a TCP/IP connection with the Eyegaze System. But note that the `stEgControl` struct is now sent to the constructor rather than init. The "Eg" was removed from this and subsequent methods since the Eg is implied because these are methods of the `EgMac` class.

`calibrate` – The functionally is identical to the original `Calibrate`.

`startData` and `stopData` – Instructs the Eyegaze System to start and stop sending gaze data. These subsume part of `EgGetData's` functionality.

`getData` – Returns a `boolean` value indicating the presence of unprocessed gaze data. The gaze data is copied to the `stEgData` struct reference used as the argument to this function. This subsumes part of `EgGetData's` original functionality.

`logFileOpen`, `logFileClose`, `logStart`, `logStop`, `logWriteColumnHead`, and `logMark` – Equivalent to the Windows Eyegaze API functions of similar name.

The following EgMac class methods extend the original API.

`isConnected` – Returns a `boolean` value indicating whether a TCP/IP connection has been established.

`isCalibrated` – Returns a `boolean` value indicating whether a successful calibration has taken place.

`isRecievingData` - Returns a `boolean` value indicating whether gaze data is being sent by the Eyegaze System.

`isLogOpen` - Returns a `boolean` value indicating whether the Eyegaze System has opened a log for gaze data recording.

`isLogging` - Returns a `boolean` value indicating whether the Eyegaze System is currently logging gaze data.

`getBufferOverFlow` – Returns a `boolean` value indicating whether gaze data was lost because the ring buffer allocated for gaze data overflowed.

4. PROPOSALS FOR THE DESIGN OF EYE-TRACKING EXPERIMENTAL SOFTWARE

While porting the Eyegaze code to the Macintosh OS, interesting issues arose that may have a wider application in HCI eye-tracking research. The motivation for porting the Eyegaze code to the Macintosh was to collect fixation data for a visual search study that replicates a previous experiment conducted using a Macintosh [6]. With the goal of reducing opportunities for errors in the eye-tracking component of an experiment, we identified three issues pertaining to the design and application of eye-tracking software. These issues will be discussed, and solutions to some of the problems will be proposed.

4.1 The Motivating Study

The context in which these issues arose was a study on visual search and mouse pointing in two-dimensional visual hierarchies [6]. In this experiment, participants were precued with a word (or pseudo-word) and then asked to find the word in a layout such as that shown in Figure 3. The factors that were varied included the number of items that appeared in the layout (10, 20, or 30) and whether the layout included group labels to guide the participant to the target. Labels were *X1X*, *X2X*, etc.

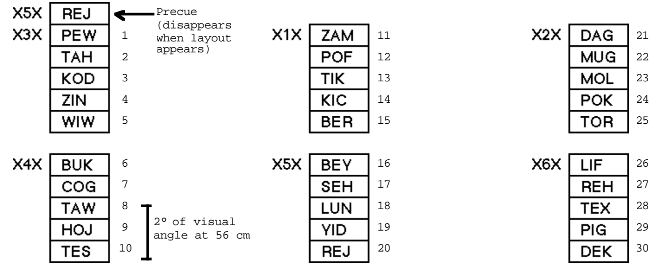

Figure 3. A screen layout with thirty items and with group labels, from the visual search task that the PI will attempt to model in EPIC. In this trial, the target *REJ* appears in the group labeled *X5X*. The number to the right of each screen object is the number assigned to that position. The precue is shown where it would have disappeared when the layout appeared. (The smaller text and numbers did not actually appear during the experiment.)

The experiment was designed to separate visual search time from target selection time by imposing a *point-completion deadline*. Once the participant started to move the mouse from the precue position, he or she had a limited amount of time to click on the target before the trial is interrupted. Search time was thus be measured from when the layout appears to when the participant starts moving the mouse to the target.

The observed visual search times demonstrate that a labeled visual hierarchy can be searched significantly faster than an unlabeled visual hierarchy. Details in the search time data, such as a more pronounced number-of-groups effect for unlabeled layouts, also suggest that people adopt fundamentally different search strategies for labeled versus unlabeled visual hierarchies. More details are available in [6].

It remains to be determined what were the actual visual search strategies used by participants in the experiment. The same experiment will be conducted again, but with eye movements recorded by an eye tracker. It is anticipated that this additional dependent variable will prove invaluable for identifying the actual search strategies employed. A number of issues will be addressed in the modification of the experiment to accommodate eye-tracking, one of which is recalibration of the eye-tracker during an experimental session.

4.2 Objective Recalibration

A possible issue with eye-tracking equipment, as with most instrumentation, is that the initial calibration may become invalid during an experiment. In reviewing the literature on HCI eye-tracking experiments, we have found that most articles do not mention the possibility of a degraded

calibration (e.g. [4]). Some studies use subjective measures to determine the need for a recalibration (e.g. [3]), but, in order to reduce experimenter bias, the need for recalibrating an eye-tracker during an experiment should be determined in an objective manner.

To satisfy the need for an unbiased recalibration determination, a coding scheme can be implemented to help ensure the eye-tracker's calibration remains accurate with an automated, predetermined test. This scheme requires three components. The first is a forced-fixation location (FFL). The second is an implementation of a fixation detection algorithm. The third is an eye-tracking system that allows real-time gaze data analysis.

The definition used here for a FFL is a periodically displayed element on which a participant must fixate in order to continue the experiment. In order to force the participant to look at the object before continuing, the participant may be required to click on the object, or manipulate it in some other way that would require the eyes to fixate the object. As an alternative, the object may contain some detailed information required to continue the experiment successfully. An example of an FFL is the precue object in the experiment discussed, [6]. The precue is the initial object shown in each trial. A participant must fixate and click on the precue to reveal the target and distracter objects, and to know the target object to click on during the trial. The frequency at which a FFL appears is at the discretion of the designer since the need to check the calibration will vary with the experiment and the eye-tracking equipment used.

The implementation and use of an FFL to verify calibration requires a fixation detection algorithm that can accept gaze data from the eye-tracker and generate fixations in real-time. An accurate algorithm that runs in linear time, such as a Hidden Markov Model or Dispersion Threshold algorithm, is best [10]. See [10] for a description of each of these algorithms. We suspect that most eye-trackers can transmit real-time gaze data to a custom application.

We wrote an automated recalibration scheme that runs in linear time, and requires only seven lines of code (see Table 1 for the pseudocode). After the FFL is presented, the scheme proceeds as follows: Continuously monitor for the FFL fixated condition. While the FFL has not been fixated, check for fixations. If a fixation is found and is within range of the FFL a recalibration is not needed. If a fixation is not found within range of the FFL by the time the FFL needs to be fixated, a recalibration should be initiated. Two key issues for this method are how to define "in range" of the FFL and a fixation before a recalibration is required, and when to interrupt the experiment for a possible recalibration.

4.2.1 Identifying fixation in range of the FFL

The maximum distance allowed from the center of the FFL to the fixation coordinates determined by a fixation detection algorithm is a significant issue for the reliability of the automated recalibration scheme. Three issues that are immediately evident are the size of the FFL and noise in the eye-tracker data. Therefore, a beginning guideline for defining "in range" may be the visual angle subtended by the FFL plus the maximum angular gaze error of the eye-tracker.

4.2.2 When to Interrupt for Recalibration

The time at which a recalibration is initiated may negatively impact an experiment. If the experiment software needs to be interrupted for a recalibration of the eye-tracker, this should be done at a point in the experiment at which the participant would not be poised to execute a time-pressured or memory-intensive response. Otherwise, the software could disrupt any motor program preparation or memorization that has been done thus far, and influence the participant's decision to do such preparation in the future. Thus, the software would interfere with the very human performance that the experiment was designed to capture in the first place. Instead, the software should wait for an appropriate time in the experiment when the participant is not poised to execute a critical task.

However, waiting for such a time to initiate a recalibration may result in the loss of data, because the need for a recalibration would indicate that the data about to be collected may not be valid. From this point of view, a recalibration should be initiated as soon as possible. Therefore, the experiment designers must balance the loss of data with continuity of the participant's experience.

```
Present the FFL
While (the FFL is not satisfied)
{
  Fixation Algorithm (gaze data)
  If (fixation)
  {
    If (distance from FFL to fixation < preset maximum)
    {
      Return and do not calibrate
    }
  }
}
Recalibrate
```

Table 1. Pseudocode for an automated recalibration scheme.

4.3 Gaze and Mouse Data Disparity

Some HCI experiments collect both gaze and mouse movement data, which are integrated and analyzed together (such as [1], [3], and [5]). Single computer video-based eye-tracking systems generate a delay between the time the image of the eye is captured and the time at which the eye position data is reported.

As illustrated in Figure 4, the time to capture, transfer, and process the camera image can produce a significant time disparity between eye position and mouse movement data. As an example, the availability of gaze data on the 60 Hz version of the Eyegaze System from LC Technologies is delayed by approximately 8.33ms by the shutter period of the camera, 16.67ms to transfer the image to the frame grabber, and 6.67ms by the video processing software, [7]. The result is a delay of 31.67ms from the time a picture of the eye is taken to the time the data from that picture is available. This delay, if not accounted for, can cause a significant error in experiments that investigate the relationship between eye position and other data, such as mouse movements.

An additional concern with respect to the eye-position data delay may be that a client-server configuration may increase the time delay of eye position data that is integrated in real-time with other data on a client computer. Two factors may increase the time delay if not handled properly. The first factor is inherent network lag. This can be avoided by properly configuring the client-server connection. A viable network configuration is directly connected 100BaseT Ethernet. This configuration supplies ample bandwidth

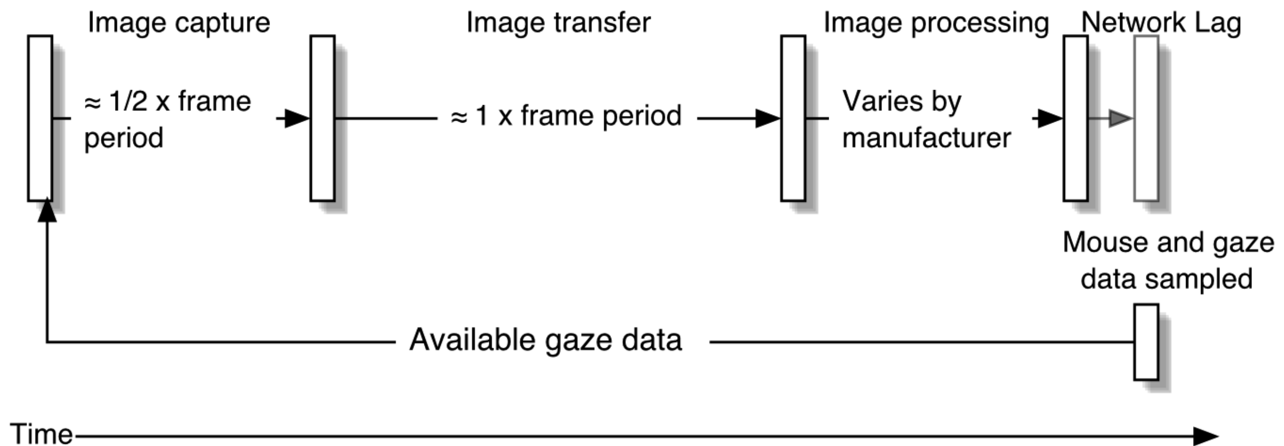

Figure 4. Approximate time disparity between image capture and reported data. The proportion of each delay is not intended to be representative of any one eye-tracker. Each eye-tracker has unique properties. The only consistent delays are the 'image capture' delay of approximately 1/2 the frame period and the 'image transfer' delay of approximately a full frame period. On the Eyegaze System, a frame period is 16.7ms. The network lag for a 100BaseT direct connection between client and server adds an minor delay in relationship to the other delays.

(100Mb/sec or 12.5 million characters per second) and a low latency relative to the delay induced by the eye-tracking system (approximately 0.11ms or 0.35% of the eye-tracker delay)[1]. In addition to the configuration, the programmatic style of the code used by the client application to receive the gaze data will affect the delay. In short, an efficient, responsive method should be used to retrieve the data from the network, such as asynchronous notification or a dedicated thread. However, this topic is beyond the scope of this paper. Consult the target operating systems network software development kit (SDK) for means to implement data retrieval. These two factors, network configuration and programmatic method, if handled properly, will not induce a significant delay in the availability of the eye position data.

Since the delay caused by the eye-tracker, and possibly by the network, causes a disparity between eye position and mouse movement data, a solution is needed to quantify this disparity. We are in the process of developing a software utility for this purpose. The utility will work as follows: Automatically move the cursor across the screen at some constant velocity. Have a participant fixate the cursor object and maintain this fixation with a series of smooth pursuit eye movements [9]. Sample both the position of the screen object and gaze position using the same functions or methods as used in the experimental software. Write this data to a file, preferably in a format that is easily imported into a spreadsheet or statistics package. Calculate the mean distance between the cursor position and gaze position. Assuming the user in this test is actually smooth-tracking the cursor, the calculated mean distance will be the sum of two errors, the error of the eye-tracker and the disparity caused by the timing mentioned above. This quantification can then be used to verify the algorithms and data connection speed, and can be reported along with the experimental data in an eye-tracking experiment.

4.4 Use of a Chinrest

The use of a chinrest may aid in maintaining a consistent setup during a lengthy experiment and obviate the need of a head tracker. If a participant needs to take a break at some point during the experiment, the initial placement of the chinrest will aid in placing the participant back in the proper position without much need for adjusting the eye-tracking equipment. Depending on the tolerances of the eye-tracker, the use of a chinrest may eliminate the need for a recalibration. However, the use of a chinrest could create or exacerbate two potential problems, mouse movement errors and participant discomfort.

However, negative effects of the chinrest must also be considered. Meegan and Tipper [8] specifically do not use a chinrest in their eye tracking study because they believe that it would interfere with the arm movements, and cite Biguer, Jeannerod, and Prablanc [2] as demonstrating that a chinrest will interact as such. Looking at the data reported in Biguer et al. [2], however, it becomes evident that the chinrest only interfered when making eye movements to targets greater than 30° from the center of view. If a chinrest is used in experiments with a standard sized computer monitor (15-19in) at a reasonable distance of 20-30in (50.8-76.2cm), mouse use should not be impacted. We do, however, share Meegan and Tipper's concerns that the chinrest may interfere with the "naturalistic reaching conditions" and ecological validity of the experiment.

There is also cause for concern with the design of conventional chinrests. Many conventional chinrests require the participant to lean over the edge of a table as shown in Picture 1. We believe this may interfere with a participant's comfort and their ability to move the mouse effectively. Therefore, it may be advisable to modify the chinrest's mounting mechanism so that it is angled away from the table as shown in Picture 2.

[1] This is the average time for one-way delivery of data using the 'ping' utility. The test used 1,000 packets of 108 bytes each. The configurations used were a Power G4 (733MHz) running OS X (10.1) and an Intel 867MHz running Windows 2000.

Picture 1. Many chinrests require the participant to lean forward in a position that is potentially uncomfortable and may interfere with normal mouse movements. Shown here is the HeadSpot chinrest.

Picture 2. We expect that slight modifications to chinrests, obviating the need for the participant to lean over the table, can resolve the problems shown in Picture 1.

5. CONCLUSION

As eye-tracking technology become more readily available, inexpensive, easy to use, extendable and expandable, and applied to new problems and new domains, this community will need more discussion on the specifics of these devices and their integration in experimental design. To that end, we have offered an assessment of the Eyegaze System by LC Technologies, and given a general reference for converting the Eyegaze client code for use on other operating systems. We then presented a specific conversion of the Eyegaze client code to the Macintosh OS. We concluded by raising experimental design issues, specifically software design, for HCI eye-tracking experiments. We expect that the design issues discussed should be relevant and useful to programmers and researchers working with other eye-tracking systems.

The discussions of experimental design issues relating to objective recalibration determination, disparity between eye position and mouse movement data, and use of chinrests in an eye-tracking experiment suggest some basic guidelines for lessening potential errors in an HCI eye-tracking experiment. Further study in these areas may be needed to establish firm guidelines. The experimental software described here is currently in the final stages of development, and a formal experiment with paid participants will be conducted in the fall of 2001. We expect that this empirical data will help to resolve and clarify many of the deign issues discussed herein.

6. ACKNOWLEDGMENTS

The authors would like to thank Dixon Cleveland and Peter Norloff of LC Technologies for their assistance with the work discussed in this paper. This work was supported by the Office of Naval Research through Grant N00014-01-10548 to the University of Oregon, Anthony J. Hornof, principal investigator.

7. REFERENCES

[1] Aaltonen, A., Hyrskykari, A., & Räihä, K.-J. (1998). 101 spots, or how do users read menus? *Proceedings of CHI 98,* New York: ACM, 132-139.

[2] Biguer, B., Jeannerod, M., & Prablanc, C. (1985). The role of position of gaze in movement accuracy. In M. I. Posner & O. S. Marin (Eds.), *Attention and Performance XI: Mechanisms of Attention*. Hillsday, NJ: Erlbaum, 407-424.

[3] Byrne, M. D., Anderson, J. R., Douglass, S., & Matessa, M. (1999). Eye tracking the visual serach of click-down menus. *ACM CHI 99,* ACM, 402-409.

[4] Crowe, E. C., & Narayanan, N. H. (2000). Comparing interfaces based on what users watch and do. *ETRA Symposium 2000,* ACM, 29-36.

[5] Gray, W. D., & Fu, W.-T. (2001). Ignoring perfect knowledge in-the-world for imperfect knowledge in-the-head: Implications of rational analysis for interface design. *Proceedings of ACM CHI 2001: Conference on Human Factors in Computing Systems,* New York: ACM.

[6] Hornof, A. J. (in press). Visual search and mouse pointing in labeled versus unlabeled two-dimensional visual hierarchies. *ACM Transactions on Computer-Human Interaction*.

[7] LC Technologies. (2001). EyeGaze Development System: Development Manual. Fairfax, VA: LC Technologies, Inc.

[8] Meegan, D. V., & Tipper, S. P. (1999). Visual search and target-directed action. *Journal of Experimental Psychology: Human Perception and Performance*, 25(5), 1347-1362.

[9] Rosenbaum, D. A. (1991). *Human Motor Control*. New York: Academic Press.

[10] Salvucci, D. D., & Goldberg, J. H. (2000). Identifying fixation and saccades in eye-tracking protocols. *ETRA Symposium 2000,* ACM, 71-78.

[11] Young, L. R., & Sheena, D. (1975). Survey of eye movement recording methods. *Behavior Research Methods and Instrumentation*, 7(5), 397-429.