

# EyeDraw v3.0

## Programmer's Documentation

### 08/20/04

## Rob Hoselton and Anna Cavender

<b>1. INTRODUCTION</b>	<b>3</b>
<b>2. EYEDRAW SOURCE FILES</b>	<b>3</b>
<b>2.1 eyedata.h</b>	<b>3</b>
2.1(a) GazeDataGetterThread	3
2.1(b) FileDataGetterThread	4
Fig 2.1 Messages Relating to Eye-Movement Data in EyeDraw	5
<b>2.2 EyeDrawDlg.cpp</b>	<b>6</b>
2.2(a) OnEyeMovement(wParam, lParam)	6
2.2(b) OnFixationCompleted(wParam, lParam)	6
2.2(c) OnEyeDwell(wParam, lParam)	6
2.2(d) OnStateChange(wParam, lParam)	7
2.2(e) OnDialogMessage(wParam, lParam)	8
2.2(f) OnCalibrationComplete(wParam, lParam)	8
2.2(e) OnLevelChange(wParam, lParam)	8
<b>2.3 EyeDrawButtons</b>	<b>9</b>
2.3(a) ED_ShapeButton:	9
2.3(b) ED_StampButton:	9
2.3(c) ED_ColorButton:	10
2.3(d) ED_UtilityButton:	10
2.4(e) ED_KeyButton:	10
<b>2.4 PaintDlg.cpp</b>	<b>11</b>
2.4(a) OnEyeMovement(wParam, lParam)	11
2.4(b) OnEyeDwell(wParam, lParam)	11
2.4(c) OnFixationCompleted(wParam, lParam)	11
2.4(d) OnStateChange	12
2.4(e) OnDialogMessage	12
2.4(f) OnPropertyChange	12
2.4(g) OnGridChange	13
2.4(h) OnUndoChange	13
<b>2.5 Shape.cpp</b>	<b>13</b>
2.5(a) setStartingPoint(POINT starting)	13
2.5(b) setEndingPoint(POINT ending)	13
2.5(c) setLocation(POINT loc)	13
2.5(d) setPen(LOGPEN* lpen)	13
2.5(e) setBrush(LOGBRUSH* lbrush)	13
2.5(f) setBitmap(int id)	13
2.5(g) getShapeRect()	14
2.5(h) isPolygonizable()	14
<b>2.6 EyeCursor.cpp</b>	<b>14</b>

<b>3. HELPER FILES</b>	<b>14</b>
3.1 FindFile.cpp (Author: Louka Dlagnekov)	14
3.2 wildcard.cpp (Author: Louka Dlagnekov)	14
3.3 iniFile.cpp (Written by: Adam Clauss, rewritten by: Shane Hill)	14
<b>4. COMMON TASKS</b>	<b>15</b>
4.1 Adding buttons	15
4.2 Drawing the eye cursor	16
4.3 Adding new messages	17
4.4 Adding bitmap resources	17
4.5 Switching to playback mode	17
4.6 Switching to random mode	18
<b>5. OUTSTANDING ISSUES AND KNOWN BUGS</b>	<b>18</b>

## 1. INTRODUCTION

The following document covers the important aspects of the EyeDraw application with respect to the application's code, its files, and functions. It also explains how to do some of the common tasks and procedures for adding new functionality.

All code is written in C++ using Microsoft Visual Studio .NET. This application version was written for the 04/11/05 release of the LC Technologies Eyegaze software.

This documentation assumes a fundamental knowledge of the C++ programming language in a windows environment.

Please refer to the LC Technologies programmer's manual for anything regarding the eye tracker and its programming interface.

## 2. EYEDRAW SOURCE FILES

### 2.1 eyedata.h

eyedata.h contains much of the LC Technologies code to collect eye movement data. All eye-movement data in EyeDraw is collected from one of the two threads within eyedata.h. The GazeDataGetterThread uses the Eyegaze thread provided by LC Technologies to receive gaze samples from the camera. The FileDataGetterThread either extracts eye-movement data from a previously recorded file (PLAYBACK mode) or generates random data on its own (RANDOM mode).

The camera's image of the eye displayed in the upper right of the screen is generated here. It can be turned on or off by setting the variable `eyelImage` to true or false respectively.

#### 2.1(a) GazeDataGetterThread

The purpose of this thread is to collect gaze samples and determine the following states, all of which are returned by the fixation detection function, `DetectFixation()` supplied by LC Technologies.

FIXATING  
FIXATION\_COMPLETED  
MOVING

More on this function and its return values can be found in the LC Technologies programmer's manual. The basic idea is that `FIXATING` is returned during a fixation, `MOVING` is returned during a saccade, and `FIXATION_COMPLETE` is returned the moment

a fixation ends (and a saccade begins). Based on these return values, a user-defined windows message (UWM) is posted to the creator of the thread (in this case EyeDrawDlg) to be processed and to initiate the proper action.

FIXATING determines that there have been a minimum number of samples (min\_fix\_samples) within a given threshold (gaze\_deviation\_thresh\_pix). The gaze point is then smoothed and a UWM\_ED\_MOVE message is posted with the smoothed gaze point. (The main program needs to know that there is still movement even though the eye is fixating. For example, we still want to display movement of the eye cursor). If there have been min\_fix\_samples of gaze points returned as FIXATING and they are all within gaze\_deviation\_thresh\_pix from the previous gaze point, then an ED\_DWELL message is posted.

FIXATION\_COMPLETED causes a UWM\_ED\_FIXATION\_COMPLETED message to be posted and MOVING causes a UWM\_ED\_MOVE message, both of which reset the number of current fixations.

Finally, this thread writes all of the smoothed coordinates and setting changes to a file labeled EDdatayear-month-day-time.dat which is stored in EyeDraw/Data/. This file can be used at a later time to playback the drawing session. More on this playback ability can be found in the following section on the FileDataGetterThread.

In addition, this thread listens for the following messages that can be send to it using a PostThreadMessage:

UWM_ED_GDT	Updates the gaze deviation threshold with the wParam value (first parameter in the message call). This is set by the user in the settings dialog box under Dwell Size.
UWM_ED_EYEIMAGE	Toggles the eyeImage that appears in the upper right. This message results from an F1 keyboard press.
UWM_ED_RECAL	Initiates a re-calibration. This message results from an F12 keyboard press or 90 samples below the screen (user looks at camera for 1500ms).

## 2.1(b) FileDataGetterThread

This thread performs one of two actions: playing back data from a pre-recorded drawing session or generating random data that simulates eye movements. In either case, the thread has the same functionality as GazeDataGetterThread in that it runs the mock gaze points through DetectFixation() and sends the same messages with smoothed data to its creator (EyeDrawDlg).

The following are programmer-defined settings located after the #include statements in eyedata.h:

During a session, GazeDataGetterThread writes all eye data including setting changes to a file created at runtime. These files are stored in EyeDraw/Data/. To playback a previous session, the file name needs to replace *filename* in "CString playbackfile = "EyeDraw/Data/*filename*.dat";" and the line "#define PLAYBACK" needs to be uncommented, both located in eyedata.h. The variable timeBetweenSamples, measured in milliseconds, can also be set to determine the speed of the playback. If set to 100/6, the playback will most closely resemble the actually speed of


the drawing session since the eye tracker captures 60 frames per second; 100/12 would be close to double the speed and 100/3 would be close to half the speed.

If the "#define RANDOM" line is uncommented, FileDataGetterThread generates random gaze points and also uses timeBetweenSamples to determine its speed. This mode can be used for stress-testing purposes.

If neither PLAYBACK nor RANDOM are defined, the program runs normally, receiving data from the camera. Although possible, PLAYBACK and RANDOM should not be defined at the same time. The effect would be a situation where the gaze samples received would alternate between those extracted from a file and those generated at random. Neither would work properly because the random samples would disrupt those from the file and samples from the file would disrupt the randomly generated "fixations".

The diagram below shows the message-sending and message-handling communication scheme of messages that are related to eye movements (listed in 2.1(a) and 2.1(b)). All messages of this type originate in eyedata.h, are handled by EyeDrawDlg.cpp, and are propagated to all of the client applications (the canvas, all of the buttons, and the dialog windows) from there. More on the handling of these messages can be found in the following section on EyeDrawDlg.cpp.

Fig 2.1 Messages Relating to Eye-Movement Data in EyeDraw

An  indicates a message being passed.

eyedata.h	Main Drawing Program (EyeDrawDlg.cpp)	Client Apps. (Buttons, Canvas, etc.)
Sets up data control structure. Calibrates. Initializes the Eyegaze thread (EgInit). Then, loops continuously asking for eye data. After each sample, calls detectFixation(). Posts messages based on the results from that call (as seen below).	When the program starts up, EyeDrawDlg enters the function OnInitDialog(). This function starts the gaze-data-getter thread, which is inside eyedata.h	
Case: MOVING or FIXATING Action: Sends the user-defined windows message UWM_ED_MOVE message.	Maps to the function OnEyeMovement() which moves eyecursor to new location and posts UWM_ED_MOVE to client apps.  Receives a button clicked message, determines where it came from and does the action associated with that button.	Each app. has its own OnEyeMovement() function that this message maps to which moves its own cursor. More on this in Fig. 2.2 Buttons click themselves after enough samples.

<p>Case: min_fix_samples of FIXATINGS occur in a row. Action: Sends a UWM_ED_DWELL message.</p>	<p>Maps to the function OnEyeDwell() which post message UWM_ED_DWELL to canvas.</p>	<p>Canvas enters its own OnEyeDwell() functions and changes state based on red-yellow-green metaphor.</p>
<p>Case: FIXATION_COMPLETE Action: Sends a ED_FIXATION_COMPLETE message. Resets the count for number of fixations in a row that would cause a ED_DWELL.</p>	<p>Maps to the function OnFixationComplete() which posts UWM_ED_FIXATION_COMPLETE to canvas.</p>	<p>Canvas enters its own OnFixationComplete() which essentially "gets out" of a yellow or red if it is in this state.</p>

## 2.2 EyeDrawDlg.cpp

EyeDrawDlg creates all of the components and starts the eye tracking thread. During program execution EyeDrawDlg handles all of the processing of new gaze points and distributes messages to all components that are affected by the new eye gaze data. For every new eye gaze data processed by GazeDataGetterThread (or FileDataGetterThread), one of the following three messages is sent out and mapped to its respective handling function for processing.

UWM\_ED\_MOVE - OnEyeMovement(wParam, lParam)  
 UWM\_ED\_FIXATION\_COMPLETED - OnFixationCompleted(wParam, lParam)  
 UWM\_ED\_DWELL - OnEyeDwell(wParam, lParam)

### 2.2(a) OnEyeMovement(wParam, lParam)

This function extracts the x and y coordinates of the gaze point and redistributes the message to PaintDlg and all of the EyeDrawButtons for further processing. The eye cursor is then erased from its old location, moved to the new location, and redrawn on the screen.

### 2.2(b) OnFixationCompleted(wParam, lParam)

This function redirects this message onto PaintDlg which uses the information to "get out" of a yellow or red. No other processing is needed.

### 2.2(c) OnEyeDwell(wParam, lParam)

This function redirects this message onto PaintDlg which uses the information to progress through the green->yellow->red sequence that starts and stops shapes.

Besides receiving new eye gaze data messages, EyeDrawDlg is the main message processing file. It processes all of the following messages and maps them to their respective handling functions each of which are described below:

UWM\_ED\_STATE\_CHANGE - OnStateChange(wParam, lParam)  
 UWM\_ED\_DLG\_MESSAGE - OnDialogMessage(wParam, lParam)  
 UWM\_ED\_CAL\_COMPLETED - OnCalibrationCompleted(wParam, lParam)  
 UWM\_ED\_LEVEL\_CHANGE - OnLevelChange(wParam, lParam)

## 2.2(d) OnStateChange(wParam, lParam)

EyeDrawDlg handles all of the button clicks (whether they are eye or mouse) from buttons on the main window and processes all of the state changing actions. Once a button has been clicked, a UWM\_ON\_STATE\_CHANGE message is posted to EyeDrawDlg to process the state change that should result from that button. The message UWM\_ON\_STATE\_CHANGE is handled by OnStateChange() in which the new state along with the type of state change are passed through wParam and lParam respectively. All EyeDrawButtons send a UWM\_ON\_STATE\_CHANGE message when clicked so, as a result, OnStateChange() is somewhat of a “hub” of the current window. The type of processing varies depending on the current state and the new state requested by the button click. The table below shows the different types of state changes, what button would cause that change, and the action taken within OnStateChange():

State change	Button that was clicked	Action
SAVE_STATE	Save	Pause the ability to draw on the canvas, and open the modal dialog box to save a drawing.
NEW_STATE	New	Pause the ability to draw on the canvas, open the modal dialog box to save, and if it returns OK (user didn't cancel), send a message to the canvas to clear the screen.
OPEN_STATE	Open	Pause the ability to draw on the canvas. If the current drawing hasn't been saved, open the save dialog box to ask the user if they want to save. Then, open the modal dialog box to open a drawing.
EXIT_STATE	Exit	Pause the ability to draw on the canvas. Check to make sure the program isn't in RANDOM simulation mode, pause the ability to draw on the canvas, and open the exit dialog box. If the dialog box returns OK, then write the current score to a file, and set IsRunning to false which will result in termination of GazeDataGetterThread or FileDataGetterThread.
UNDO_STATE	Undo	Post a message to the canvas to undo the last item drawn.
GRID_STATE	Grid	Post a message to the canvas requesting the grid be toggled.
SETT_STATE	Settings	Open the settings dialog box.
DOT_STATE	Dot On/Dot Off	Toggle the clutch. The clutch is what stops the eye cursor from moving when the Dot On/Dot Off button is clicked. Also, change the caption of that button.
FILL	Any of the shape buttons clicked more than once	Fill occurs when a shape button is clicked more than once resulting in the current shape tool being filled. To the canvas, this is considered a <i>property</i> change versus a <i>state</i> change. So, send the message on to the canvas.
COLOR	Any of the color buttons	Color is also considered a property change, so send the proper message to the canvas. Also, notify all of

		the buttons in case they are interested in the color change. For example, shape button display their shape in the current color and other color buttons display a smaller color square than the one that is selected.
STAMP	Any of the stamp buttons (but not the Stamps button)	Notify the canvas of the change in stamp type as a property change.
LEFT_STATE	The left arrow button	Call the function that rotates the menu at the bottom of the screen to the left.
RIGHT_STATE	The right arrow button	Call the function that rotates the menu at the bottom of the screen to the right.
SHAPE_STATE	Any of the shape buttons	Notify the canvas of the new shape tool selected, notify the buttons so they can choose how to display themselves.
STAMP_STATE	The Stamps button	Open the Stamps dialog box and if it closes with an OK, notify the canvas of the new stamp mode, notify the buttons so they can choose how to display themselves.

#### 2.2(e) OnDialogMessage(wParam, lParam)

EyeDrawDlg also receives messages from dialog box windows when they are open, such as SaveDlg.cpp and OpenDlg.cpp. In these cases, the only object that needs to know about the message (which contains the filename of the file to open or with which to save the current drawing under) is the canvas, and so the message is propagated on.

#### 2.2(f) OnCalibrationComplete(wParam, lParam)

The calibration program that runs at the start of EyeDraw is initiated from the GazeDataGetterThread within eyedata.h. It is spawned as a thread using Calibrate.exe and the GazeDataGetterThread pauses, waiting for its return. At that point, it notifies EyeDrawDlg that the calibration completed successfully by posting a UWM\_ED\_CAL\_COMPLETED. OnCalibrationComplete handles this message by starting the about box (the splash screen seen at the start of the program). When it is created, the about box starts a timer for its own destruction, but also receives messages from EyeDrawDlg about button clicks upon which it would destroy itself if the timer had not yet gone off. See about AboutDlg.cpp for comments on how this is done.

#### 2.2(e) OnLevelChange(wParam, lParam)

There are currently three different drawing levels that a user can be in. Before the LEVEL\_BEGINNER, the user only has access to free eye drawing, after the LEVEL\_BEGINNER, the user has access to everything except for stamps and typing. Finally in LEVEL\_EXPERT the user has access to all aspects of the program. The first level is achieved through 18000 eye movements (5 minutes of drawing), then the expert level is achieved through 300 eye dwell (100 lines or shapes drawn in theory). Since the canvas is doing all of the drawing, it is also keeping track of the current Score and when it hits one of these milestones, the canvas posts an UWM\_ED\_LEVEL\_CHANGE which maps to OnLevelChange(). Here, all of the buttons are placed properly and the current Score is written to file. Because this function handles which buttons are accessible, it is called at the start of the program in OnInitDlg().



## 2.3 EyeDrawButtons

EyeDrawButtons are extended from the MFC CButton class and are owner drawn (meaning that their appearance and their position on the screen is defined by the programmer). All types of EyeDrawButtons are located in EyeDrawButton.cpp. They receive UWM\_ED\_MOVE messages which are handled by OnEyeMovement() in the EyeDrawButton class (other classes inherit this function by default). This function receives the new gaze point and determines if the gaze point falls within its window region by a function called WindowUnderEye(). If the gaze point does fall within its region the button draws its eye cursor and increments the number of gaze samples. After a given number of consecutive samples, the button either is 'pressed' or 'released'. Once the button has been released the clicked() function is called to produce the appropriate action.

EyeDrawButtons can be one of the following based on their functionality: ED\_ShapeButton, ED\_StampButton, ED\_ColorButton, ED\_KeyButton, or ED\_UTILITYButton and each type of button must override the following functions: OnStateChange(), redraw(), and clicked().

OnStateChange() notifies the button that some change to the state of the program has been performed and each button must adapt itself to the new state.

redraw() is the function that defines the appearance of the button on the screen. It is called from the base class's OnPaint() method which is called whenever the button needs to be drawn. Each type of button has a unique visual aspect depending on its type.

clicked() notifies the parent with a UWM\_ED\_STATE\_CHANGE message which contains the type of state change and the new state, both of which depend on the type of button. For example, a click of the rectangle button will result in a type of state change SHAPE\_STATE (because it is a shape button) and a new state of RECTANGLE (which is passed to it during construction).

The positions of the buttons are defined by the window they are in. The buttons on the main window are positioned inside OnInitDialog() in EyeDrawDlg.cpp, the buttons in the Open Dialog are positioned within OnInitDialog() in OpenDlg.cpp and so on. The positioning is based on the current screen coordinates so that the program looks similar in any resolution.

### 2.3(a) ED\_ShapeButton:

An ED\_ShapeButton is a button that determines the type of drawing tool. During construction of an ED\_ShapeButton the type of shape is passed in through the constructor. When the button is clicked, the clicked() function is called and this value is then passed along with SHAPE\_STATE in a UWM\_ED\_STATE\_CHANGE message to notify the parent class.

A shape button is displayed by the redraw() function with the type of shape tool this button represents. The color of the shape depends on which system color has been selected and whether or not this is the system's current tool. The shape is either filled or not filled depending on whether the button has been clicked an odd or even number of times.

All buttons also received UWM\_ED\_STATE\_CHANGE messages from other button clicks which map to OnStateChange() notifying the button of program state changes and information that is used to change how the button is drawn. This function also changes whether the current shape is filled or not when the button is clicked more than once.

### 2.3(b) ED\_StampButton:

An `ED_StampButton` is a button that determines the bitmap to be painted to the screen during stamping mode. The button is constructed by passing the id of the bitmap's resource which the button will hold. (See section 4.4 to add a bitmap resource.)

The `redraw()` function uses the resource id to paint the bitmap to the button and draw a green rectangle around the stamp if the button is selected.

The `clicked()` method uses the resource id as a message parameter in the `UWM_ED_STATE_CHANGE` message. This resource id is then used to paint the bitmap to the screen to and display the eye cursor.

`ED_StampButtons` use the `OnStateChange()` function to know whether the program thinks this button is selected. If there is a change, it will redraw itself to show or erase the rectangle.

### 2.3(c) `ED_ColorButton`:

An `ED_ColorButton` is a button that determines the color of the shapes. The button is constructed by passing the macro variable of the buttons color. These macros are located in `EyeDarwDlg.h`

The `redraw()` function draws a rectangle of the button's color.

The `clicked()` method uses the color as a message parameter in the `UWM_ED_STATE_CHANGE` message.

The `OnStateChange()` function changes the size of the rectangle that is drawn on the button depending on whether or not it is selected.

### 2.3(d) `ED_UtilityButton`:

`ED_UtilityButton`'s are buttons such as Save, Open, New, Undo, Grid, Settings, and Exit. These buttons can either display text as to which function they perform or display a bitmap that shows which function they perform. The constructor is given three parameters; the state with which they will notify the system, and the two strings in which the button will display if the button toggles between states.

The `redraw()` function displays the text on the button or draws a bitmap if `hasBitmap` is true.

The `clicked()` function passes an `ED_STATE_CHANGE` message to the parent with the button's state value.

This button does not use the `OnStateChange()` method but instead has an `OnCaptionChange()` function which handles messages of type `UWM_ED_CAPTION_CHANGE`, toggling the text on the button when it is clicked.

### 2.4(e) `ED_KeyButton`:

`ED_KeyButtons` are the keys in the keyboard in the typing window. They display the letter or string that they represent when clicked, which is provided in the constructor.

The `redraw()` function draws the letter as text on the button and when the button is clicked (while it's down) the button has a green background. This has the effect of a green flash to indicate the button was pressed.

ED\_KeyButtons do not use the OnStateChange() method because they are not effected by changes in program state.

The clicked() function posts an ED\_STATE\_CHANGE to the parent (the typing dialog) with the button's type (KEY) and its value.

All buttons receive bitmaps via a UWM\_ED\_BITMAPCHANGE message that maps to the OnBitmapChange function in the base class (EyeDrawButton).

Often, buttons are added to a vector of buttons so that every button can be notified of a state change. In addition to the vector of buttons, color buttons are also added to a vector color buttons. This vector is used to rotate the palette of buttons located at the bottom of the screen. An example of this occurs in EyeDrawDlg.cpp during a state change message. See the diagram in 2.2(d) for specifics.

## 2.4 PaintDlg.cpp

PaintDlg is the implementation file for our drawing canvas. On initialization, the size and location of the canvas are set along with all of the initial canvas states and the needed device contexts used for the display.

PaintDlg is notified by EyeDrawDlg of all the eye movement data through the following messages and their corresponding handlers:

UWM\_ED\_MOVE - OnEyeMovement(wParam, lParam)

UWM\_ED\_DWELL – OnEyeDwell(wParam, lParam)

UWM\_ED\_FIXATION\_COMPLETED - OnFixationCompleted(wParam, lParam)

### 2.4(a) OnEyeMovement(wParam, lParam)

This function handles the drawing of the eye cursor and the temporary display of shapes during swinging. It also draws a line for each eye movement during the free eye drawing mode.

### 2.4(b) OnEyeDwell(wParam, lParam)

This function handles the progression through the green->yellow->red sequence that sets the starting and ending points of shapes and the permanent drawing to the canvas of stamps and shapes.

### 2.4(c) OnFixationCompleted(wParam, lParam)

This function resets the state along with the cursor color to “just looking” mode. This is where the program “gets out” of a yellow or a red.

PaintDlg also handles messages from EyeDawDlg of state changes affecting the canvas. These messages and their corresponding handlers are as follows:

UWM\_ED\_STATE\_CHANGE - OnStateChange(wParam, lParam)

UWM\_ED\_DLG\_MESSAGE - OnDialogMessage(wParam, lParam)

UWM\_ED\_PROPERTY\_CHANGE - OnPropertyChange(wParam, lParam)

UWM\_ED\_GRID\_CHANGE - OnGridChange(wParam, lParam)

UWM\_ED\_UNDO - OnUndoChange(wParam, lParam)

#### 2.4(d) OnStateChange

This function catches messages from EyeDrawDlg about the changes in program state which are of the type (either SHAPE\_STATE if a shape tool is selected, STAMP\_STATE if stamps are selected, TYPE\_STATE if something has been typed, NEW\_STATE if the user chooses to start a new drawing and erase the current one, and PAUSE\_STATE which is used to disable the ability to draw, such as when dialog boxes are open) and the value (FREE, LINE, CIRCLE, RECTANGLE if the tool is a shape, nothing for the other states).

If the state is SHAPE\_STATE, it then instantiates a new shape tool and this tool becomes the current drawing tool and the eye cursor becomes a small rectangle that changes color on dwells.

If the state is STAMP\_STATE, the current shape becomes a stamp based on the current resource ID (set in OnPropertyChange) and the eye cursor becomes the image of that stamp with a color-changing rectangle encompassing it.

If the state is TYPE\_STATE, the current shape becomes a label with the string that was typed and the eye cursor displays that string with a color-changing rectangle encompassing it.

If the state is PAUSE\_STATE, the eye cursor becomes a rectangle temporarily and because the currentState is set to PAUSE\_STATE, the canvas will not respond to drawing commands.

If the state is NEW\_STATE, all of the shapes and opened image are erased and the grid is replaced if it happened to be on.

#### 2.4(e) OnDialogMessage

This function catches message from EyeDrawDlg that were passed from dialog boxes. The messages contain the type of dialog box the message came from and the new value (eg: filename of file to open or save to). It could do one of two things:

1. If the message is requesting a 'save', then
  - 1a. Remove the grid.
  - 1b. Create a new .png file using the filename given.
  - 1c. Write the drawing data to the file.
  - 1d. Replace the grid.
2. If the message is requesting an 'open', then
  - 2a. Load the drawing from the filename given.
  - 2b. Clear all the shapes.
  - 2c. Draw the new image to the screen and to the device contexts.

#### 2.4(f) OnPropertyChange

Catches message from EyeDrawDlg indicating that a property of a tool should be changed. The message contains the type of property to change and the new value of the change. Examples of property changes are a change in color, fill, the resource ID of a stamp, or the string in a typed label.

#### 2.4(g) OnGridChange

Catches message from EyeDrawDlg requesting the grid be toggled.

#### 2.4(h) OnUndoChange

Catches message from EyeDrawDlg requesting an undo. Undo has different meaning based on the state of the program:

1. If in a 'swing' (IsDrawing is true), just end the swing.
2. Otherwise, remove the last shape that was drawn.
3. If the program is in freeDraw mode, remove the last 60 shapes drawn (or the last 1 second of drawing).

### 2.5 Shape.cpp

Shape.cpp implements the high level shape object. All drawing tools extend shape for the basic functions of all shapes. Every type of shape created gets placed in a vector of shapes. All shape objects inherit the following functions:

#### 2.5(a) setStartingPoint(POINT starting)

All shapes have a point where the bounding rectangle will begin. This point is set by this function during an UWM\_ED\_DWELL message received by the canvas.

#### 2.5(b) setEndingPoint(POINT ending)

Shapes also have an ending point in which the bounding rectangle will end. This point is set by this function during an UWM\_ED\_DWELL message received by the canvas. The ending point must be adjusted as to not include the area of the eye cursor.

#### 2.5(c) setLocation(POINT loc)

Although all shapes have a location, this is only used by stamps and labels. This function is useful because these object get stamped in one place (they don't have a starting or ending point) and because this function offsets the point given in order to center the object to where the user is looking.

#### 2.5(d) setPen(LOGPEN\* lpen)

This function sets the shapes pen color, width, and style (the border of the rectangle or circle, or just the appearance of a line).

#### 2.5(e) setBrush(LOGBRUSH\* lbrush)

This function sets the shapes brush color, width, and style (the filled area of the rectangle or circle, but nothing with respect to a line). So far, EyeDraw's filled shapes always have the same color for the pen and the brush, so they appear to have no border.

#### 2.5(f) setBitmap(int id)

This function changes the bitmap of the stamp.

## 2.5(g) getShapeRect()

This function returns the bounding rectangle of the shape.

## 2.5(h) isPolygonizable()

The only tool that is Polygonizable is the Line tool, but the canvas needs to know this so that rectangles and circle don't respond to the polygon mode of drawing. Other shapes could be polygonizable in the future.

EyeDraw's current set of drawing tools all extend the Shape class so that they can all be included in a Shapes array. Every shape object must implement the Draw() function, which draws the shape to the given device context, and the copy() function, which returns a pointer to a copy of this shape.

The following files contain the classes that extend Shape:

- Line.cpp
- Rectangle.cpp
- Circle.cpp
- Stamp.cpp
- Label.cpp

Stamps and Labels are slightly different than the others because they are constructed with the resource ID of the bitmap or the string of what was typed. Stamps are drawn by loading a bitmap using the current resource id and then calling DrawTransparentBitmap to ensure a transparent background for the bitmap. Note - the resourced bitmap must be 70 x 70 pixels. (See section 4.4 to add a bitmap resource.)

## 2.6 EyeCursor.cpp

EyeCursor.cpp implements the eye cursor and encapsulates the moving, drawing and erasing of the eye cursor during eye movements. The eye cursor is set to be a rectangle during most drawing operations or to the specified stamp or string with encompassing rectangle during stamping by calling the setMode() function. For a description of how to draw the eye cursor, see section 4.2.

## 3. HELPER FILES

### 3.1 FindFile.cpp (Author: Louka Dlagnekov)

FindFile.cpp searches for files and folders with a variety of different options in a given location. This class is used for saving and retrieving drawings.

### 3.2 wildcard.cpp (Author: Louka Dlagnekov)

wildcard.cpp is used by FindFile.cpp.

### 3.3 iniFile.cpp (Written by: Adam Clauss, rewritten by: Shane Hill)

iniFile.cpp is used to store the current EyeDraw settings of fMinFixMs, and whether the eye cursor goes through the yellow stage or not.

## 4. COMMON TASKS

### 4.1 Adding buttons

The following steps are needed to add a new button to EyeDraw:

Step	Action	Meaning
1.	Under resource view open IDD_EYEDRAW_DIALOG. (View->Resource View->EyeDraw.rc->Dialog->Double Click on IDD_EYEDRAW_DIALOG)	The resource view shows the main window with the resources for the buttons on the that window. Since all of our buttons are owner-drawn, their appearance here is not indicative of what they will look like when the program is running.
2.	Add a new button to the resource view. This easiest way is to copy an already existing button.	This creates a resource for your button that will be used during construction and for mapping button clicks.
3.	Open the properties view and set the button's ID and caption. Make sure the Owner Draw property is set to true.	If you cut and pasted from another button, the ID won't be something that makes since. So, change it to reflect the meaning of your button. The caption doesn't really matter as the button will be drawn by the programmer, but it's nice to keep things straight in the resource view.
4.	Open EyeDrawDlg.h.	This is where states, colors, and button click functions for the main window are located.
5.	Add the appropriate state and set its value. For instance, to add the rectangle button the following line was added: const int RECTANGLE = 2; This value will be passed into the constructor of the button in the next steps. If a new color button is added then you will also have to define the color for example, #define RGB_BLACK RGB(0,0,0).	These are the things that define the purpose of the button and they need to be here so that other objects will know how to interpret messages from the button.
6.	Open EyeDrawDlg.cpp.	Assuming your new button will be on the main screen, this is where is it constructed and positioned.
7.	Initialize the button with the appropriate class and constructor. This code is placed before the class definition. For example, the rectangle button was constructed with the line ED_ShapeButton m_rectButton(RECTANGLE);	Check your buttons type constructor for what to pass to it.
8.	In CEyeDrawDlg::DoDataExchange(CDataExchange*pDX) add the line DDX_Control(pDX, button ID, button) replacing button ID with the ID given to the button in the properties view in step 3. For example, the line DDX_Control(pDX, IDC_RECT_BUTTON, m_rectButton) was added for the rectangle button.	This maps the button you created in the resource view to the button you just created by initializing it EyeDrawDlg.cpp. This allows the program to map button clicks and drawing techniques to that button.

9.	Under BEGIN_MESSAGE_MAP(CEyeDrawDlg, CDialog) add the line to map the button click message to the appropriate function. For example, ON_BN_CLICKED(IDC_RECT_BUTTON, OnBnClickedRectButton). Be sure to define this function. See section 2.3 for more on button click messages. This step also requires adding an afx_msg to EyeDrawDlg.h. For example, for a rectangle button, afx_msg void OnBnClickedRectButton(); was added.	This message mapping diverts the flow of control to your specified function whenever that button is clicked with the mouse. The function you define for this can do whatever action makes since for when that button is clicked.
10.	In CEyeDrawDlg::OnInitDialog() set the button window position.	Since all EyeDrawButtons are owner-drawn we have to position them ourselves on the screen. This is done using relative positioning (so that it looks similar in any screen resolution).
11.	Add the button to the buttons vector and if it is a color or stamp button then add it to its appropriate vector as well.	All buttons have to be in the buttons vector in order to receive eye movement data and updates about program state. The colors vector and stamps vector are used to swap out the palette of colors or stamps along the bottom of the screen depending on if the program is in stamp mode or shape mode.

## 4.2 Drawing the eye cursor

Drawing the eye cursor is done in a series of steps. The task is made simpler by initializing a new EyeCursor [eyeCursor = new EyeCursor(p\_pdc,updateRect)] at the start of the program, or when the window is initialized. The constructor takes two arguments which are a pointer to the CDC initializing the object and the ClientRect of the window. For every eye movement data, the previous eye cursor needs to be erased. This is effectively done by storing a memory device context and then displaying what was originally the area under the eye cursor. tempDC is the device context to which the eye cursor will be drawn. Since all of our drawing is first done to a memory device, this area then needs to be invalidated (updated and redrawn) so that the window will be updated. Next the eye cursor is moved to the new location, and then drawn. After it has been drawn to the device context, the area needs to be invalidated once again to show the eye cursor in its new location. The area that needs to be invalidated can be accessed through eyeCursor->getRect(). Here is a simplified order of events that occurs with every new eye movement.

Event	Code	What is does
1. The old eye cursor is erased.	eyeCursor->erase (&tempDC);	Retrieves the memory device context which contains the image that appeared where the eye cursor is now. Draws that image to the tempDC and invalidates, essentially erasing the eye cursor.
2. The eye cursor is moved to the new	eyeCursor->move (eye_x, eye_y);	Changes the x-, y-coordinates stored in the eyeCursor class.



location.		
3 .The new eye cursor is drawn.	eyeCursor->draw (&tempDC, &memoryDC)	The image of the area where the eye cursor will be drawn is saved (for erasing is later) from the memoryDC and the eye cursor is drawn to the tempDC.

### 4.3 Adding new messages

The entire list of user defined messages is located EyeDrawDlg.h. If messages other than the ones previously defined in this header file are going to be added then a new message will need to be defined here. All user-defined windows messages in EyeDraw begin with UWM\_ED\_. Whether a new defined message is added or a previously defined message is going to be used the following steps remain the same.

The line BEGIN\_MESSAGE\_MAP ('your class', 'base class') should be included in every class that will use messages replacing 'your class' and 'base class' with the appropriate values. Within this statement block, add the message map to map the message defined above to its handler function. For instance, to map the message UWM\_ED\_MOVE to the handler function OnEyeMovement, the following line would be added under BEGIN\_MESSAGE\_MAP: ON\_MESSAGE (UWM\_ED\_MOVE, OnEyeMovement). Once this has been accomplished the message handler function needs to be defined within this class.

In order for the message handler function to be defined, the function also needs to be defined in the header file for the same class. The functions will be added to the public section of the header file and will begin with afx\_msg LRESULT followed by the name of your function and its parameters.

### 4.4 Adding bitmap resources

To add new bitmap resource to EyeDraw, under Project->Add Resource select bitmap then either new or import. If you are creating a new bitmap from scratch then select new. If the bitmap has already been created then select import. Your new bitmap and its ID will be located under resource view (View->Resource View->EyeDraw.rc->Bitmap). If you select Properties (View->Properties Window) you will be able to rename the ID of the bitmap. In resource.h your new bitmap will be defined automatically. Note: if this bitmap is to be used as a stamp, the dimensions should be 70 x 70 pixels.

### 4.5 Switching to playback mode

Playback mode refers to the program's ability to playback prerecorded data from a file. To playback a previous drawing session, the file name of the data file created during the drawing session needs to replace filename in "CString playbackfile = "EyeDraw/Data/filename.dat";" and uncomment the line "#define PLAYBACK", both located towards the top of eyedata.h. The data files are stored in EyeDraw/Data. You can choose the speed at which it plays by setting the variable timeBetweenSamples which is measured in milliseconds. To playback at close to the same speed as the original recording, set timeBetweenSamples to 100/6 (since the camera records samples at 60 frames per second). About double the speed would be 100/12 and about half the speed would be 100/3.

## 4.6 Switching to random mode

Random mode refers to the program's ability to create random gaze samples to be used for stress testing purposes. To make the program run in random mode, make sure "#define PLAYBACK" is commented and "#define RANDOM" is uncommented at the top of eyedata.h. You can choose the speed at which it plays by setting the variable timeBetweenSamples at similar settings as described above. Also, you can define randomPlayTime in seconds so that the program only runs for the amount of time you want.

If neither PLAYBACK nor RANDOM are defined, the program runs normally, receiving data from the camera. Although possible, PLAYBACK and RANDOM should not be defined at the same time. The effect would be a situation where the gaze samples received would alternate between those extracted from a file and those generated at random. Neither would work properly because the random samples would disrupt those from the file and samples from the file would disrupt the randomly generated "fixations".

## 5. OUTSTANDING ISSUES AND KNOWN BUGS

- When the program first starts, the GazeDataGetterThread pauses for a short minute to insure that the program has fully initialized before passing eye movement data to it. If the user opens a dialog box (such as Save, Open, Exit, Typing, Stamps...) by clicking on one of these buttons with the mouse during this paused time, the thread will terminate itself (I think) and no eye movement data will ever reach the window. This error is visible because the eye image in the upper right will not appear and when Exit is clicked with the mouse the program does not exit. Note: this error cannot occur with the use of eyes, the mouse must cause it.
- Several other problems occur when the mouse is used: for example, remnants of eye cursors are left over when dialog boxes are opened with the mouse.
- Occasionally, only portions of stamps get stamped down onto the canvas. For example, if the eye cursor is moved away too quickly after a stamp.
- It would be cool if playback mode could somehow happen without having to recompile. Maybe by feeding in the filename of the file to be played-back as a command line argument.
- The current version of EyeDraw (with the eye image and recalibrate abilities) requires version of LC Technologies files that were made after 08/04. Before, we were sending self-contained versions of EyeDraw out to users (they had their own Calibrate and such). But, we would like to incorporate EyeDraw into the Eyegaze menu so that our users would have easier access to it. So there is an outstanding issue of how to distribute EyeDraw. The best idea so far includes implementing an installer which would check the user's version of Eyegaze before installing. If it is too old, the installer would not install and tell the user they need to contact LC for a newer version before installing EyeDraw. Otherwise, the installer would insert itself into the user's Eyegaze menu and put all the right files in the right places. Also, an installer should be easily updatable as new versions of EyeDraw evolve and it would notice when older version of EyeDraw are already installed and take appropriate action.
- A redo function would be nice and not too hard to implement.
- Also, an unclear in the typing would be good because if the clear button is pressed in the text goes away permanently.

- With Norton Personal Firewall is active, the eye image does not appear in EyeDraw, even if configure Norton so that EyeDraw.exe and Calibrate.exe has full access.
- With high processor load, the eye cursor may lag behind eye movements.