

# **EyeDraw v2.5**

## **Programmer's Documentation**

**07/13/04**

**Anna Cavender and Rob Hoselton**

<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)	6
2.2(e) OnDialogMessage(wParam, lParam)	8
2.2(f) OnCalibrationComplete(wParam, lParam)	8
<b>2.3 EyeDrawButtons</b>	<b>8</b>
2.3(a) ED_ShapeButton:	8
2.3(b) ED_StampButton:	9
2.3(c) ED_ColorButton:	9
2.3(d) ED_UtilityButton:	9
<b>2.4 PaintDlg.cpp</b>	<b>10</b>
2.4(a) OnEyeMovement(wParam, lParam)	10
2.4(b) OnEyeDwell(wParam, lParam)	10
2.4(c) OnFixationCompleted(wParam, lParam)	10
2.4(d) OnStateChange	11
2.4(e) OnDialogMessage	11
2.4(f) OnPropertyChange	11
2.4(g) OnGridChange	11
2.4(h) OnUndoChange	11
<b>2.5 Shape.cpp</b>	<b>11</b>
2.5(a) setStartingPoint(POINT starting)	12
2.5(b) setEndingPoint(POINT ending)	12
2.5(c) setPen(LOGPEN* lpen)	12
2.5(d) setBrush(LOGBRUSH* lbrush)	12
2.5(e) getShapeRect()	12
<b>2.6 EyeCursor.cpp</b>	<b>12</b>
<b>3. HELPER FILES</b>	<b>12</b>
<b>3.1 FindFile.cpp (Author: Louka Dlagnekov)</b>	<b>12</b>
<b>3.2 wildcard.cpp (Author: Louka Dlagnekov)</b>	<b>13</b>

<b>3.3 iniFile.cpp (Written by: Adam Clauss, rewritten by: Shane Hill)</b>	<b>13</b>
<b>4. COMMON TASKS</b>	<b>13</b>
<b>4.1 Adding buttons</b>	<b>13</b>
<b>4.2 Drawing the eye cursor</b>	<b>14</b>
<b>4.3 Adding new messages</b>	<b>15</b>
<b>4.4 Adding bitmap resources</b>	<b>15</b>
<b>4.5 Switching to playback mode</b>	<b>16</b>
<b>4.6 Switching to random mode</b>	<b>16</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 was written for the 11-06-03 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 `eyeImage` 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`.

## 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 actual 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.
Case: min_fix_samples of FIXATINGs occur in a row. Action: Sends a UWM_ED_DWELL message.	Maps to the function OnEyeDwell() which post message UWM_ED_DWELL to canvas.	Canvas enters its own OnEyeDwell() functions and changes state based on red-yellow-green metaphor.
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.	Maps to the function OnFixationComplete() which posts UWM_ED_FIXATION_COMPLETE to canvas.	Canvas enters its own OnFixationComplete() which essentially "gets out" of a yellow or red if it is in this state.

## 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:

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

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

UWM\_ED\_CAL\_COMPLETED - OnCalibrationCompleted(wParam, lParam)

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

EyeDrawDlg handles all of the button clicks (whether they are eye or mouse) from 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 is passed through wParam and lParam respectively. As a side note, all EyeDrawButtons send a UWM\_ON\_STATE\_CHANGE message 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, and open the modal dialog box to open a drawing.
EXIT_STATE	Exit	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.
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 and also toggle the caption on the grid button.
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)	A change in stamp type is the final property change that the canvas can receive through the EyeDrawDlg.
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	Make sure the color buttons appear on the bottom of the screen, make sure the stamp buttons do not appear on the bottom of the screen, 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	Make sure the stamp buttons appear on the bottom of the screen, make sure the color button do not appear on the bottom of the screen, 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 started a timer for its own destruction, but also receives messages from button clicked 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.

## 2.3 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 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.

### 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 how many times the button has been clicked.

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.

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.

This Button does not use the `OnStateChange` function because it is not affected by changes in program state. This could be easily added if a situation arises where it would need to be notified of program changes.

### 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 display text as to 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.

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.

Every button has a message mapped to a specific function (located in the parent class) to handle mouse clicks. These functions perform the same actions as the clicked() functions described above. They pass a UWM\_ED\_STATE\_CHANGE message to the parent with the type of state change and the new state.

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 and stamp buttons are also added to either a vector of stamp buttons or a vector of color buttons. These vectors are used to switch out the palette of buttons located at the bottom of the screen. For example, if the stamp button was selected, the color buttons will all be hidden while the stamp buttons will all be displayed. This is done by looping through both vectors and performing the correct action. This is taken care of 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.

### **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.

PaintDlg also handles messages from EyeDrawDlg 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, or STAMP\_STATE if stamps are selected) and the value (LINE, CIRCLE, RECTANGLE if the tool is a shape, nothing if the tool is a stamp). It then instantiates a new shape tool and this tool becomes the current drawing tool. The eye cursor becomes either a rectangle or the current bitmap encompassed by a rectangle depending on the tool selected.

#### 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 .bmp 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 bitmap 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 or fill.

#### 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.

### 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) 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(d) 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(e) getShapeRect()

This function returns the bounding rectangle of the shape.

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 returned 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

Stamps are slightly different because they are constructed with the resource ID of the bitmap. 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 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	Check your buttons type constructor for

	and constructor. This code is placed before the class definition. For example, the rectangle button was constructed with the line <code>ED_ShapeButton m_rectButton(RECTANGLE);</code>	what to pass to it.
8.	In <code>CEyeDrawDlg::DoDataExchange(CDataExchange* pDX)</code> add the line <code>DDX_Control(pDX, button ID, button)</code> replacing button ID with the ID given to the button in the properties view in step 3. For example, the line <code>DDX_Control(pDX, IDC_RECT_BUTTON, m_rectButton)</code> 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 <code>EyeDrawDlg.cpp</code> . This allows the program to map button clicks and drawing techniques to that button.
9.	Under <code>BEGIN_MESSAGE_MAP(CEyeDrawDlg, CDialog)</code> add the line to map the button click message to the appropriate function. For example, <code>ON_BN_CLICKED(IDC_RECT_BUTTON, OnBnClickedRectButton)</code> . Be sure to define this function. See section 2.3 for more on button click messages. This step also requires adding an <code>afx_msg</code> to <code>EyeDrawDlg.h</code> . For example, for a rectangle button, <code>afx_msg void OnBnClickedRectButton();</code> 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 sense for when that button is clicked.
10.	In <code>CEyeDrawDlg::OnInitDialog()</code> set the button window position.	Since all <code>EyeDrawButtons</code> 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.	<code>eyeCursor-&gt;erase (&amp;tempDC);</code>	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 location.	<code>eyeCursor-&gt;move (eye_x, eye_y);</code>	Changes the x-, y-coordinates stored in the eyeCursor class.
3 .The new eye cursor is drawn.	<code>eyeCursor-&gt;draw (&amp;tempDC, &amp;memoryDC)</code>	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 purposed. 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".