Chapter 12
Using an Open Source Application Program Interface

Chapter 10 provided details for writing client software to communicate with a table-mounted eye tracker via an Application Programming Interface (API), specifically the Linux Tobii API. That specific API was based on C data structures, which, at the time, were developed on a 32-bit system architecture. The API did not port very well to 64-bit architectures due to byte word alignment problems.

A better approach for development of an eye-tracking communications API would be one which would be independent of architecture (platform) or language (e.g., C), i.e., one that would be platform-agnostic. Such an API is overviewed in this chapter, specifically one developed by Gazepoint, Inc.[1] for their GP eye trackers. What makes this API platform-agnostic is that the communication packets exchanged between eye-tracking client and server are extensible mark-up language XML strings.

Gazepoint's open source API is based on the web-services API model, exchanging XML packets over a TCP/IP communications channel. Both TCP/IP and XML are open standards, currently available on most operating systems and programming languages. Various language libraries are also available to make assembly and parsing of XML fairly straightforward. Data and commands are encoded in plain text XML and transmitted as simple strings. Gazepoint's API is thus not so much an API as it is a protocol. Besides executing commands meant to start calibration or to transmit data, care must be taken to send, receive, parse, and acknowledge the protocol packets.

12.1 API Implementation and XML Format

The Gazepoint API establishes a client-server communication channel. The client application configures selected variables on the server, initiates the data transmission sequence, and then listens for data and/or acknowledgments sent by the server.

[1] http://gazept.com.

A.T. Duchowski, *Eye Tracking Methodology*, DOI 10.1007/978-3-319-57883-5_12

Table 12.1 Gazepoint API XML TAG identifiers

(a) Client	
TAG	Description
GET	Get a data variable or command
SET	Set a data variable or command
(b) Server	
ACK	Acknowledge a successful command
NACK	Acknowledge a failed command
CAL	Calibration result record
REC	Data result record

Data and commands are formatted using XML string fragments called elements. Each element is defined by an empty-element TAG that specifies the element type. An empty-element TAG is of the form <GET…/>, which is shorter than the start/end element format <GET>…</GET>. As of version 2.0 of the API, only six XML tags are required for the open eye-gaze API and they are listed in Table 12.1.

Additional parameters that may be required in a data or command packet are defined by XML name/value attribute pairs, e.g.:

```
<GET ID="ENABLE_SEND_TIME" />
```

where the attribute is ID and the value is ENABLE_SEND_TIME.

To indicate an end of string, the Gazepoint API requires usage of a carriage return (CR) and line feed (LF) pair (\r\n). The CRLF sequence is safe to use as a record delimiter since the XML specification disallows CRLF from appearing within the XML string. The CRLF tuple, acting as an end of text token, is needed due to possibly differing reading and writing speeds of the client and server. The consequence of differing speeds is that XML strings may appear to either system as partial strings if read too quickly or as multiple elements if read too slowly, e.g.:

```
<GET ID="ENABLE_SE
<GET ID="COMPANY_ID" /><GET ID="API_ID" />
```

12.2 Client/Server Communication

To write an eye-tracking program, e.g., either an interaction application such as a game, or a diagnostic type of program like PsychoPy that displays stimuli and collects data, consider the program the client application that connects to the eye tracker, which acts as the server. The client opens a TCP stream, e.g., via a socket, to the server using the server's IP address and an assigned port number (4242 is the

default). Using TCP rather than UDP ensures data is not lost and is received in the correct order.

Applications may be run on local or remote computers and connect to the eye-tracking server by using the appropriate server IP address. For a local connection, typically `127.0.0.1` is used (see example with `Python` excerpts below). According to Gazepoint, multiple eye trackers may run on the same machine by assigning different port numbers to the different servers, e.g., `4242` for the first, `4243` for the next, etc.. A client application may then connect to multiple eye trackers by opening both ports for communication.

12.3 Server Configuration

The Gazepoint eye tracking server operates in one of three modes: configuration, calibration, and data transmission. In configuration mode the server responds to XML accessor or mutator packets (`GET` or `SET` TAG) with appropriate XML (`ACK` or `NACK` TAG) responses. Configuration mode is used to select variables in the requested data stream, get or set other eye tracker variables, and to start or end calibration or data transmission.

An example which turns on or off the tracker's display, e.g., so that the user may or may not see it, is given below:

```
CLIENT SEND: <GET ID="TRACKER_DISPLAY" />
SERVER SEND: <ACK ID="TRACKER_DISPLAY" STATE="1" />

CLIENT SEND: <SET ID="TRACKER_DISPLAY" STATE="1" />
SERVER SEND: <ACK ID="TRACKER_DISPLAY" STATE="1" />
```

where in the first of the two exchanges the client queries the server's display state and in the second it sets it to on. The complete list of configuration variables in the Gazepoint API can be found in the API manual (Gazepoint 2013). Some of the more important ones are listed in Table 12.2.

Note that some of the functionality may not be immediately obvious. For example graceful shutdown of the tracker, i.e., stopping its sending of data is accomplished by sending the following command:

```
CLIENT SEND: <SET ID="ENABLE_SEND_DATA" STATE="0" />
```

Similarly, stopping calibration is accomplished by sending this command:

```
CLIENT SEND: <SET ID="CALIBRATE_SHOW" STATE="0" />
```

Forgetting to send the above may result in the eye tracking server stuck in calibration mode even after it has completed iterating through all calibration points.

Table 12.2 Abridged list of Gazepoint XML packets

XML ID	I/O	Parameters	Type	Description
Configuration Queries and Commands				
ENABLE_SEND_DATA	rw	STATE	bool	start/stop data streaming
ENABLE_SEND_POG_LEFT	rw	STATE	bool	enable left eye gaze data
ENABLE_SEND_POG_RIGHT	rw	STATE	bool	enable right gaze eye data
ENABLE_SEND_POG_FIX	rw	STATE	bool	enable fixation data
SCREEN_SIZE	r	WIDTH	int	screen width (pixels)
SCREEN_SIZE	r	HEIGHT	int	screen height (pixels)
Calibration Display and Control (default point list)				
CALIBRATE_RESET	rw	PTS	int	reset to default point list†
CALIBRATE_SHOW	rw	STATE	bool	show/hide calibration grid
CALIBRATE_START	rw	STATE	bool	start/stop calibration
Calibration Setup Commands (for custom calibration point list)				
CALIBRATE_CLEAR	rw	PTS	int	clear out point list†
CALIBRATE_ADDPOINT	rw	X Y	float	add point with x-, y-coord
CALIBRATE_TIMEOUT	rw	VALUE	int	seconds for stationary dot
CALIBRATE_DELAY	rw	VALUE	float	seconds between dots
CALIBRATE_RESULT_SUMMARY	r	‡	float/int	results
Calibration Data and Results (following completion of each calibration point)				
CALIB_RESULT_PT	r	PT	int	calib. point completed
CALIB_RESULT	r	CALX? CALY?	float	x-, y-coord. for point ?
CALIB_RESULT	r	LX? LY?	float	left eye x, y for point ?
CALIB_RESULT	r	LV?	int	left eye valid for point ?
CALIB_RESULT	r	RX? RY?	float	right eye x, y for point ?
CALIB_RESULT	r	RV?	int	right eye valid for point ?

? = the calibration point number
† result should be 0
‡ results will include AVE_ERROR, VALID_POINTS

12.4 API Extensions

The Gazepoint API outlined here corresponds to version 2.0 of the open source interface, identified by the VER_ID="2.0" variable. Future efforts to improve the API interface will be identified by corresponding API version numbers. Future API extensions may include head mounted, or 3D eye-tracking specific variables (e.g., see

Hennessey and Lawrence (2008)) by expanding the XML command and data dictionary. Excerpts from an interactive Python application are given below.

```
class Gazept:
  def __init__(self,server="127.0.0.1",port=4242):
    self.server = server
    self.port = port
    try:
      self.sock = socket.socket(socket.AF_INET,\
                                socket.SOCK_STREAM)
      self.sock.connect((server,port))
      self.receiving_thread = \
           threading.Thread(target=self.communication_loop)
      self.receiving_thread.start()
    except:
      print "couldn't connect to ",server," on port ",port
      sys.exit()
```

Listing 12.1 Python Gazepoint class with TCP/IP socket.

12.5 Interactive Client Example Using Python

Best and Duchowski (2016) described a user study of a gaze-based Personal Identification Number (PIN) entry interface. The interactive application was written in Python. Some of the code excerpts are given here to provide an example of how the client/server communication is set up.

The Python code for an interactive Gazepoint application is set up as a basic OpenGL/GLUT program so that basic rendering is handled by the OpenGL API and the Graphical User Interface (GUI) is handled by a GLUT window. Before GLUT is allowed to run its main loop, a Gazept object is created and initialized.

Upon initialization, the Gazept object opens the client/server connection and starts a communication thread that listens for data from the server, as shown in Listing 12.1. The communication thread uses the Gazept object's communication_loop method to receive and parse incoming messages. Python's string encode() and decode() methods handle mapping of the incoming bitstream, returning a string for subsequent parsing.

A global state object is used as a shared memory object holding state variables accessible by both the GUI and Gazept communication threads. This allows a mechanism for sharing program state information, e.g., whether the program is idle, calibrating, or running, so that both client and server may remain synchronized.

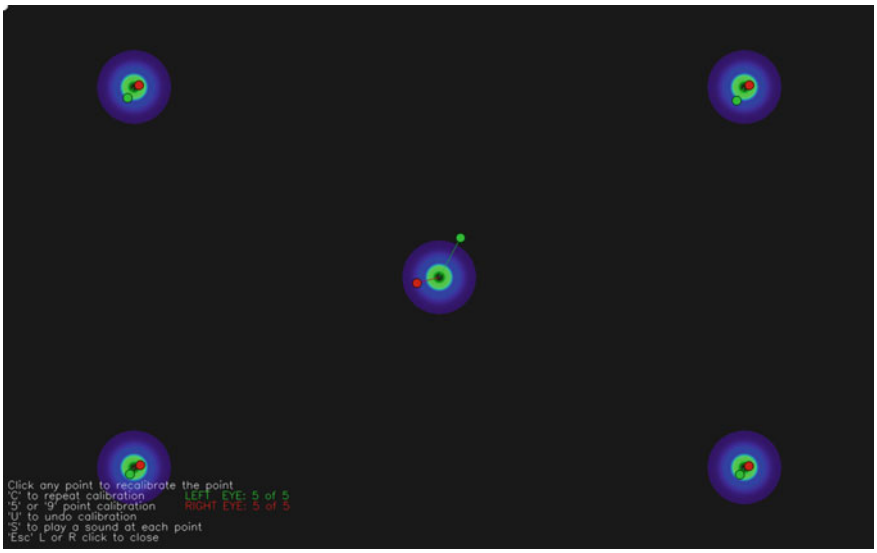

Fig. 12.1 The default Gazepoint calibration screen

12.5.1 Using Gazepoint's Built-in Calibration

If running the eye-tracking client application on the same machine as the eye-tracking server, one can use Gazepoint's built-in default calibration. This usually clears the screen and proceeds to draw calibration points in a pre-defined sequence. An example screenshot of the completion of the default calibration is shown in Fig. 12.1. Setting up this "canned" calibration is straightforward and generally requires issuance of three commands.

First, instruct the tracker to reset calibration to the default points:

```
CLIENT SEND: <SET ID="CALIBRATE_RESET" />
SERVER SEND: <ACK ID="CALIBRATE_RESET" PTS="5" />
```

Second, instruct the tracker to display its own calibration screen:

```
CLIENT SEND: <GET ID="CALIBRATE_SHOW" STATE="1"/>
SERVER SEND: <ACK ID="CALIBRATE_SHOW" STATE="1" />
```

Third, start the calibration:

```
CLIENT SEND: <SET ID="CALIBRATE_START" STATE="1" />
SERVER SEND: <ACK ID="CALIBRATE_START" STATE="1" />
```

The advantage of this canned method is that the eye tracker handles the synchronization and reports the average calibration error (accuracy). The disadvantage is that your client application has no control over how the calibration is drawn, what is displayed at the calibration coordinates, etc.. To wrench control over calibration, the

client application must provide its own animation and it must indicate to the server the precise timing and location of the sequence.

12.5.2 Using Gazepoint's Custom Calibration Capabilities

Designing a custom calibration animation sequence allows complete control over what is portrayed on the screen for the viewer to look at during calibration, even if is still in essence the same sequence of say 5 or 9 points. The difference is that anything one chooses can be displayed as the calibration dots (they might not be dots at all, but say images), and what the dot does between stationary positions can also be made to vary. An example of a nonlinear path between dots is given below.

The trickiest part of custom calibration is specification of the timing of how long each stationary points stays stationary and the time between each pair of them. Custom calibration is more complicated to set up than canned calibration but relies on the same underlying mechanism, defined by a specific number of calibration points, the timing of presentation of each point (for the calibration point to dilate/constrict, for example), and the timing between any two points (to allow animation of a visible dot between the two points).

To set up the basic framework for the custom calibration, first tell the server to clear out all calibration points:

```
CLIENT SEND: <SET ID="CALIBRATE_CLEAR" />
SERVER SEND: <ACK ID="CALIBRATE_CLEAR" PTS="0" />
```

Next, tell the eye tracker where the stationary dots will be drawn:

```
CLIENT SEND: <SET ID="CALIBRATE_ADDPOINT" X="0.5" Y="0.5/>
CLIENT SEND: <SET ID="CALIBRATE_ADDPOINT" X="0.1" Y="0.9/>
CLIENT SEND: <SET ID="CALIBRATE_ADDPOINT" X="0.9" Y="0.9/>
CLIENT SEND: <SET ID="CALIBRATE_ADDPOINT" X="0.9" Y="0.1/>
CLIENT SEND: <SET ID="CALIBRATE_ADDPOINT" X="0.1" Y="0.1/>
SERVER SEND: <ACK ID="CALIBRATE_ADDPOINT" PTS="5" X1="0.500"
Y1="0.500" X2="0.100" Y2="0.900" X3="0.900" Y3="0.900"
X4="0.900" Y4="0.100" X5="0.100" Y5="0.100" />
```

At the end of the calibration, a calibration report can still be obtained just as what would be provided following default calibration, via the following command:

```
CLIENT SEND: <GET ID="CALIBRATE_RESULT_SUMMARY" />
SERVER SEND: <ACK ID="CALIBRATE_RESULT_SUMMARY"
AVE_ERROR="19.43" VALID_POINTS="5"/>
```

Note that more specific information can be obtained even as the calibration proceeds, via the eye tracker's CALIB_RESULT_PT and CALIB_RESULT data packets (see Table 12.2 and the manual (Gazepoint, 2013).

For the duration of the display of each calibration dot, when it is stationary (i.e., when eye tracking camera samples the eye image), specify this duration with this command (e.g., for 2 s):

```
CLIENT SEND: <GET ID="CALIBRATE_TIMEOUT" VALUE="2"/>
SERVER SEND: <ACK ID="CALIBRATE_TIMEOUT" VALUE="2" />
```

then to set the time between calibration dots, use this command (e.g., for 1 second in between dots):

```
CLIENT SEND: <SET ID="CALIBRATE_DELAY" VALUE="1" />
SERVER SEND: <ACK ID="CALIBRATE_DELAY" VALUE="1" />
```

The timing between dots can be used to produce a pleasing animation of the calibration dot between stationary targets. Normally, the basic approach is to linearly interpolate the position of the dot from the last point ($\mathbf{p}_n$) to the next ($\mathbf{p}_{n+1}$) via linear interpolation, e.g.,

$$\mathbf{p}_t = (1-t)\mathbf{p}_n + (t)\mathbf{p}_{n+1}, \quad t \in [0,1] \tag{12.1}$$

where t is the interpolant, normalized over the amount of time available between calibration dots (the `CALIBRATE_DELAY` value). It should be clear that at the beginning of this short animation ($t = 0$), the point will start at $\mathbf{p}_n$ and at the end of the animation it will rest at $\mathbf{p}_{n+1}$. In between, the position is blended linearly between both points such that at the halfway mark ($t = 0.5$) the animation dot will be exactly halfway.

For convenience, the above can equivalently be rewritten in terms of two linear blending functions $h_0(t)$ and $h_1(t)$,

$$\mathbf{p}_t = h_0(t)\mathbf{p}_n + h_1(t)\mathbf{p}_{n+1}, \quad t \in [0,1] \tag{12.2}$$

where $h_0(t) = (1-t)$ and $h_1(t) = t$. Beyond a simple linear trajectory, the movement of the calibration point can be animated along a path modeled as one that might be made by a natural saccade. Such a path can be derived from a force-time function assumed by a symmetric-impulse variability model (Abrams et al. 1989). This model is qualitatively similar to symmetric limb-movement trajectories and describes an acceleration profile that rises to a maximum, returns to zero about halfway through the movement, and then is followed by an almost mirror-image deceleration phase.

A symmetric acceleration function can be modeled by a choice of combination of Hermite blending functions $h_{11}(t)$ and $h_{10}(t)$, so that $\ddot{H}(t) = h_{10}(t) + h_{11}(t)$ where $h_{10}(t) = t^3 - 2t^2 + t$, $h_{11}(t) = t^3 - t^2$, $t \in [0,1]$, and $\ddot{H}(t)$ is acceleration of the calibration point over normalized time interval $t \in [0,1]$. Blending functions $h_{10}(t)$ and $h_{11}(t)$ have two subscripts because there are normally four Hermite blending functions for 2D curves, and for the present purposes only two are used. Integrating acceleration produces velocity, $\dot{H}(t) = \frac{1}{2}t^4 - t^3 + \frac{1}{2}t^2$ which when integrated one more time produces position $H(t) = \frac{1}{10}t^5 - \frac{1}{4}t^4 + \frac{1}{6}t^3$ on the normalized interval $t \in [0,1]$ (see Fig. 12.2).

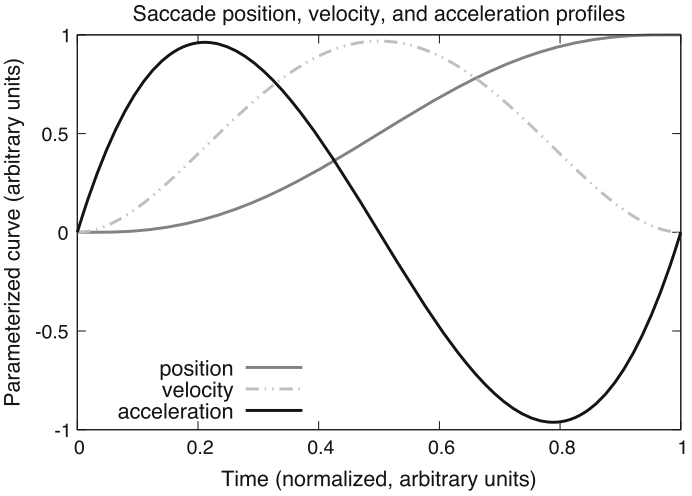

Fig. 12.2 Nonlinear blending function $H(t)$ for position of calibration dot, modeled after a parametric saccade position model derived in turn from an idealized model of saccadic force-time function assumed by Abrams et al.'s (1989) symmetric-impulse variability function: scaled position $60H(t)$, velocity $31\dot{H}(t)$, and acceleration $10\ddot{H}(t)$

```
if clb.movement == 0:
    t = clb.timer_elapsed_ms()/clb.get_moveTime()
    i = clb.get_dot_index()
    if i+1 < clb.get_dot_len():
      v1 = clb.get_dot(i)
      v2 = clb.get_dot(i+1)
      t = (1.0/10.0)*t**5 - (1.0/4.0)*t**4 + (1.0/6.0)*t**3
      t = 60.0 * t
      v = ((1.0 - t)*v1[0] + t*v2[0], \
           (1.0 - t)*v1[1] + t*v2[1])
      clb.set_xy(v)
    if clb.timer_elapsed_ms() > clb.get_moveTime():
      clb.inc_dot_index()
      clb.timer_start()
      # advance to dilation
      clb.movement += 1
```

Listing 12.2 Python code snippet for nonlinear calibration dot movement

Because (12.1) is an equation for position over a normalized time window ($t \in [0, 1]$), blending function $H(t)$ can be inserted into place of t and then t can be stretched (scaled) to any given length $t \in [0, \Delta t]$ to produce movement where the calibration dot accelerates and decelerates slightly at each calibration point. See Listing 12.2 for an example of how the Hermite blending function is used where `clb` is a Python object that stores calibration information including calibration point data, move time, and an elapsed time timer.

12.6 Summary and Further Reading

Some of the rationale for developing an open source API was given in a short paper by Hennessey and Duchowski (2010). Further details of Gazepoint's API can be found in their API manual, e.g., v2.0 as of this writing (Gazepoint, 2013).