Chapter 6
Accessing Gaze Data During Recording

Contents

The eye-tracking applications discussed in Chaps. 4 and 5 involve "passive recording" only. We record eye movements while the participant is performing a task, but we do not use the eye movement data in any way during the task. In many research scenarios, however, it is necessary to access real-time gaze data to drive the presentation of the stimuli. For instance, in a moving window task, the observer can view a small screen region around the current gaze position. In a saccadic adaptation task, the target is shifted immediately following saccade onset. This chapter will focus on the retrieval of eye movement data during recording.

Samples and Events

Two types of data are available from the EyeLink Host PC during recording—samples and events. A detailed discussion of these two types of data is presented in Chap. 8. For now, it is sufficient to know that samples are the "raw" eye movement data and comprise the x, y gaze position and pupil size. They are provided every 1 or 2 ms, depending on the sampling rate. Events are fixations, saccades, and blinks detected from the sample data.

The parsing of sample data into fixations and saccades can be a complicated topic, and different research fields have different conventions and approaches. The algorithm used by the EyeLink trackers is velocity-based and is applied by the Host

Z. Wang, *Eye-Tracking with Python and Pylink*,
https://doi.org/10.1007/978-3-030-82635-2_6

PC software during recording. A saccade is detected if the eye velocity or acceleration exceeds predefined thresholds, and the same saccade terminates when both velocity and acceleration fall below thresholds. The onset of a new fixation follows the termination of the leading saccade; the start of a saccade is preceded by the end of the preceding fixation (see Fig. 6.1). You can find further details of the saccade parsing algorithm in the EyeLink user manuals.

The high sampling rate of the EyeLink trackers allows the identification of eye events during recording, and the tracker can make these events available over the link. Sample data can be accessed in a near real-time fashion (see Fig. 6.2), with a delay in the one- to two-sample range (depending on the sampling rate and filter setting). On the other hand, eye events require some time to identify and cannot be accessed in a real-time manner. For instance, for detecting saccade onset, the algorithm includes a brief verification period to ensure velocity or acceleration is indeed above the thresholds. Consequently, event data is typically delayed by about 20–40 ms over the link, depending on the tracker configuration.

The EyeLink trackers allow flexible control of the eye movement data. When you call *startRecording()*, it is mandatory to specify four parameters to let the tracker know whether sample and event data should be saved to file and made available over the link.

<file_samples> -- 1, writes samples to EDF; 0, disables sample recording
<file_events> -- 1, writes events to EDF; 0, disables event recording

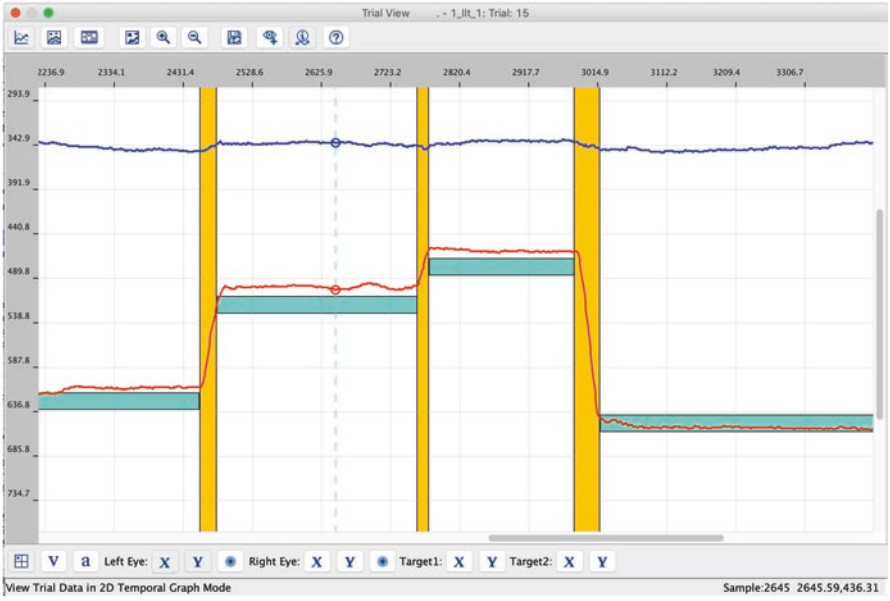

Fig. 6.1 The EyeLink tracker detects eye events based on instantaneous velocity and acceleration. In this temporal graph, the X, Y gaze traces are plotted in blue and red. Blue boxes represent fixations, and vertical stripes represent saccades

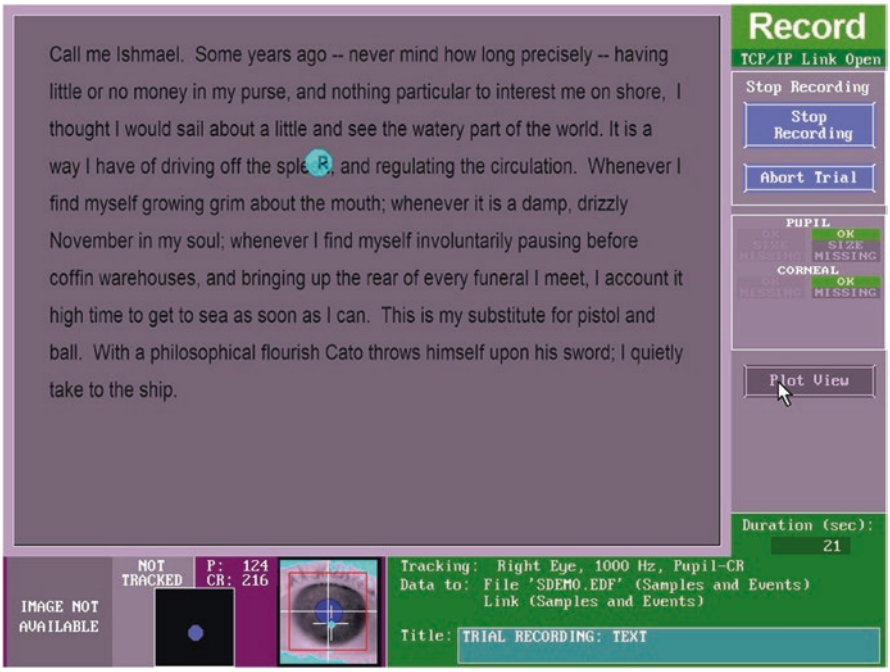

Fig. 6.2 A screenshot of the EyeLink Host PC showing that real-time gaze data is used to drive a gaze cursor. The eye movement data is also accessible from another device through an Ethernet link to the tracker

<link_samples> -- 1, sends samples through link; 0, disables link sample
<link_events> -- 1, sends events through link; 0, disables link event

The following command sets the tracker to record both events and samples in the EDF data file and make both events and samples available over the link. Switching the *<link_samples>* and *<link_events>* parameters to 0 will disable sample and event access over the link during recording.

```
tk.startRecording(1, 1, 1, 1)
```

Accessing Sample Data

Once we have started recording and made sure that the eye movement data will be available over the link, we can use various commands to retrieve samples or events during testing. The retrieval of samples is helpful in gaze-contingent tasks, for instance, the classic moving window task by McConkie and Rayner (1975; see Rayner, 2014, for a review). Fast online retrieval of gaze samples is also necessary

for boundary-crossing tasks, where we manipulate the stimuli when the gaze crosses an invisible boundary on the screen.

Commands for Accessing Samples

We use the *getNewestSample()* function to retrieve the latest sample. This function will return the latest *sample*, which has lots of properties that we can extract. For instance, the following code snippet determines whether the newest sample comes from the left eye, right eye, or both eyes, and it also returns the resolution data (i.e., how many pixels correspond to 1 degree of visual angle at the current gaze position) and the timestamp of the sample.

```
smp = tk.getNewestSample() # retrieve the latest sample
is_left = smp.isLeftSample() # left eye sample
is_right = smp.isRightSample() # right eye sample
is_bino = smp.isBinocular() # binocular recording is on
res = smp.getPPD() # 1 deg = how many pixels at the current gaze position
time_stamp = smp.getTime() # timestamp on the tracker
```

To get the current gaze position, we call *getLeftEye()* or *getRightEye()* to get the actual *SampleData* for the left and right eye, respectively. In the following code snippet, if the right eye data is available, we retrieve the gaze position from the right eye data stream; otherwise, we extract data from the left eye data stream.

```
# Poll the latest samples
dt = smp.getNewestSample()
if smp is not None:
    # Grab gaze, HREF, raw, & pupil size data
    if smp.isRightSample():
        gaze = smp.getRightEye().getGaze()
        href = smp.getRightEye().getHREF()
        raw = smp.getRightEye().getRawPupil()
        pupil = smp.getRightEye().getPupilSize()
    elif smp.isLeftSample():
        gaze = smp.getLeftEye().getGaze()
        href = smp.getLeftEye().getHREF()
        raw = smp.getLeftEye().getRawPupil()
        pupil = smp.getLeftEye().getPupilSize()
```

In addition to gaze position (in screen pixel coordinates), the samples may also contain HREF or RAW data, depending on the tracker configuration. HREF is head-referenced data, i.e., eye rotations relative to the head; it can be helpful in physiological studies examining the actual movement of the eye. RAW data is the raw data from the camera sensor; it is useful in studies where researchers would like to calibrate the tracker with their calibration routines. For instance, one can adjust signal gain and offset to map the RAW signal range to a known screen region. The HREF data can be retrieved with *getHREF()*, RAW data can be retrieved with *getRawPupil()*, and pupil size data can be retrieved with *getPupilSize()*.

Below is a short script to illustrate how to retrieve the various types of sample data, including gaze, HREF, RAW, and pupil size data.

```
#!/usr/bin/env python3
#
# Filename: sample_retrieval.py
# Author: Zhiguo Wang
# Date: 2/6/2021
#
# Description:
# A short script illustrating online retrieval of sample data

import pylink



# Connect to the tracker
tk = pylink.EyeLink('100.1.1.1')

# Open an EDF data file on the Host PC
tk.openDataFile('smp_test.edf')

# Put the tracker in offline mode before we change tracking parameters
tk.setOfflineMode()

# Set sample rate to 1000 Hz
tk.sendCommand('sample_rate 1000')

# Make gaze, HREF, and raw (PUPIL) data available over the link
sample_flag = 'LEFT,RIGHT,GAZE,GAZERES,PUPIL,HREF,AREA,STATUS,INPUT'
tk.sendCommand(f'link_sample_data = {sample_flag}')

# Open an SDL window for calibration
pylink.openGraphics()
```

```
# Set up the camera and calibrate the tracker
tk.doTrackerSetup()

# Put tracker in idle/offline mode before we start recording
tk.setOfflineMode()

# Start recording
error = tk.startRecording(1, 1, 1, 1)

# Wait for a moment
pylink.msecDelay(100)

# Open a plain text file to store the retrieved sample data
text_file = open('sample_data.csv', 'w')

# Current tracker time
t_start = tk.trackerTime()
smp_time = -1
while True:
    # Break after 5 seconds have elapsed
    if tk.trackerTime() - t_start > 5000:
        break

    # Poll the latest samples
    smp = tk.getNewestSample()
    if smp is not None:
        # Grab gaze, HREF, raw, & pupil size data
        if smp.isRightSample():
            gaze = smp.getRightEye().getGaze()
            href = smp.getRightEye().getHREF()
            raw = smp.getRightEye().getRawPupil()
            pupil = smp.getRightEye().getPupilSize()
        elif smp.isLeftSample():
            gaze = smp.getLeftEye().getGaze()
            href = smp.getLeftEye().getHREF()
            raw = smp.getLeftEye().getRawPupil()
            pupil = smp.getLeftEye().getPupilSize()

        timestamp = smp.getTime()

        # Save gaze, HREF, raw, & pupil data to the plain text
        # file, if the sample is new
        if timestamp > smp_time:
            smp_data = map(str, [timestamp, gaze, href, raw, pupil])
            text_file.write('\t'.join(smp_data) + '\n')
```

```
            smp_time = timestamp

# Stop recording
tk.stopRecording()

# Close the plain text file
text_file.close()
# Put the tracker to the offline mode
tk.setOfflineMode()
# Close the EDF data file on the Host
tk.closeDataFile()

# Download the EDF data file from Host
tk.receiveDataFile('smp_test.edf', 'smp_test.edf')

# Close the link to the tracker
tk.close()

# Close the window
pylink.closeGraphics()
```

In the script, we first connect to the tracker, open a calibration window, calibrate the tracker, and close the calibration window. We then send two commands to the tracker to ensure the tracker operates at 1000 Hz and, most importantly, make the HREF and raw PUPIL data available over the link. The second command (*link_sample_data*) controls what types of sample data can be accessed over the link during recording. There is no need to delve into the details here; please see the EyeLink user manuals for more information about the sample data flags (e.g., GAZE, HREF, PUPIL) that can be included in this command.

```
# Set sample rate to 1000 Hz
tk.sendCommand('sample_rate 1000')

# Make gaze, HREF, and raw (PUPIL) data available over the link
sample_flag = 'LEFT,RIGHT,GAZE,GAZERES,PUPIL,HREF,AREA,STATUS,INPUT'
tk.sendCommand(f'link_sample_data = {sample_flag}')
```

For an illustration, we open a plain text file to store the sample data we accessed during recording. Note that, before we enter the *while* loop, *trackerTime()* is called to get the current time on the EyeLink Host PC; in the *while* loop, we repeatedly call this function to check if 5 seconds have elapsed. If so, we break out the loop.

```
# Open a plain text file to store the retrieved sample data
text_file = open('sample_data.csv', 'w')

# Current tracker time
t_start = tk.trackerTime()

while True:
    # Break after 5 seconds have elapsed
    if tk.trackerTime() - t_start > 5000:
        break
```

Both sample and event data are timestamped according to the clock on the EyeLink Host PC. The time returned by *trackerTime*() is the same as that recorded in the EDF file. You can verify this by sending the time returned by *trackerTime*() as a message to the tracker.

Back to the script, a *while* loop is used to constantly check if a new sample is available. One way to tell if a sample is new is to compare its timestamp against the previous sample. If a sample is new, we save it to the text file.

```
        # Save gaze, HREF, raw, & pupil data to the plain text
        # file, if the sample is new
        if timestamp > smp_time:
            smp_data = map(str, [timestamp, gaze, href, raw, pupil])
            text_file.write('\t'.join(smp_data) + '\n')
            smp_time = timestamp
```

In this example, we log the samples into a plain text file, just for an illustration. It would not be sensible to log samples in this way in an actual experiment, as it creates a considerable overhead, and the data is already stored in the EDF file. Please see below for an excerpt from the output text file. The columns are sample timestamp, gaze position, HREF x/y, raw data, and pupil size (area or diameter in arbitrary units).

```
3577909.0 (872.0999755859375, 392.70001220703125) (1631.0, 7100.0)
(-1924.0, -3891.0) 366.0
3577910.0 (872.2000122070312, 393.3999938964844) (1632.0, 7093.0)
(-1924.0, -3889.0) 365.0
3577911.0 (872.2999877929688, 393.79998779296875) (1632.0, 7090.0)
(-1924.0, -3888.0) 365.0
3577912.0 (872.2000122070312, 397.20001220703125) (1631.0, 7060.0)
(-1924.0, -3876.0) 365.0
3577913.0 (872.0999755859375, 397.29998779296875) (1630.0, 7059.0)
(-1925.0, -3875.0) 365.0
3577914.0 (870.7000122070312, 397.3999938964844) (1619.0, 7059.0)
(-1930.0, -3875.0) 362.0
3577915.0 (870.7000122070312, 397.3999938964844) (1619.0, 7058.0)
(-1930.0, -3875.0) 362.0
```

An Example in PsychoPy: Gaze-Contingent Window

The following example script implements a simple gaze-contingent window task, taking advantage of the *Aperture* feature of PsychoPy. We use an *aperture* to reveal a rectangular region centered at the current gaze position (see Fig. 6.3).

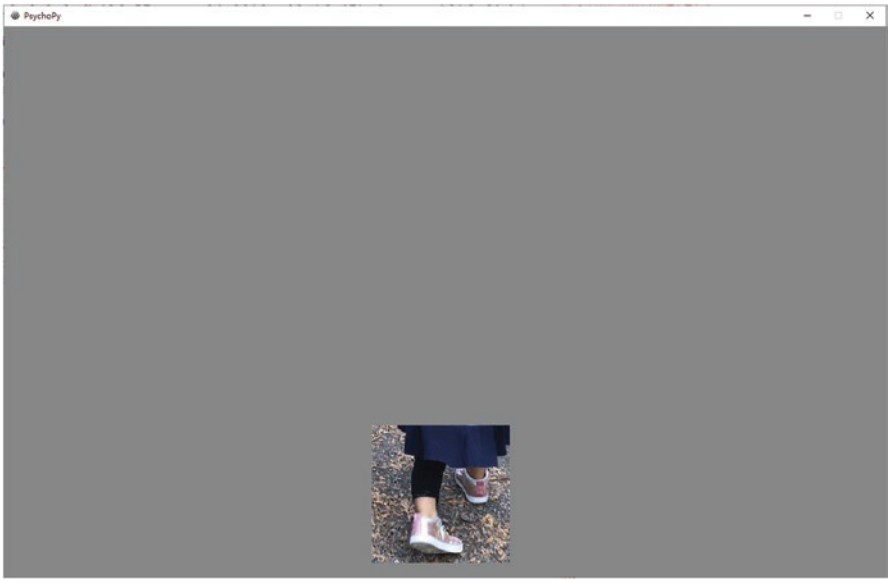

Fig. 6.3 A simple gaze-contingent window task implemented in PsychoPy

```
#!/usr/bin/env python3
#
# Filename: gaze_contingent_window.py
# Author: Zhiguo Wang
# Date: 2/6/2021
#
# Description:
# A gaze-contingent window task implemented in PsychoPy

import pylink
from psychopy import visual, core, event, monitors
from EyeLinkCoreGraphicsPsychoPy import EyeLinkCoreGraphicsPsychoPy

# Connect to the tracker
tk = pylink.EyeLink('100.1.1.1')

# Open an EDF data file
tk.openDataFile('psychopy.edf')

# Put the tracker in offline mode before we change tracking parameters
tk.setOfflineMode()

# Make all types of sample data available over the link
sample_flags = 'LEFT,RIGHT,GAZE,GAZERES,PUPIL,HREF,AREA,STATUS,INPUT'
tk.sendCommand(f'link_sample_data  = {sample_flags}')

# Screen resolution
SCN_W, SCN_H = (1280, 800)

# Open a PsyhocPy window with the "allowStencil" option
win = visual.Window((SCN_W, SCN_H), fullscr=False,
                    units='pix', allowStencil=True)

# Pass the display pixel coordinates (left, top, right, bottom) to the tracker
coords = f"screen_pixel_coords = 0 0 {SCN_W - 1} {SCN_H - 1}"
tk.sendCommand(coords)

# Request Pylink to use the custom EyeLinkCoreGraphicsPsychoPy library
# to draw calibration graphics (target, camera image, etc.)
genv = EyeLinkCoreGraphicsPsychoPy(tk, win)
pylink.openGraphicsEx(genv)

# Calibrate the tracker
calib_msg = visual.TextStim(win, text='Press ENTER to calibrate')
calib_msg.draw()
```

```
win.flip()
tk.doTrackerSetup()

# Set up an aperture and use it as a gaze-contingent window
gaze_window = visual.Aperture(win, shape='square', size=200)
gaze_window.enabled = True

# Load a background image to fill up the screen
img = visual.ImageStim(win, image='woods.jpg', size=(SCN_W, SCN_H))

# Put tracker in Offline mode before we start recording
tk.setOfflineMode()

# Start recording
tk.startRecording(1, 1, 1, 1)

# Cache some samples
pylink.msecDelay(100)

# show the image indefinitely until a key is pressed
gaze_pos = (-32768, -32768)
while not event.getKeys():
    # Check for new samples
    smp = tk.getNewestSample()
    if smp is not None:
        if smp.isRightSample():
            gaze_x, gaze_y = smp.getRightEye().getGaze()
        elif smp.isLeftSample():
            gaze_x, gaze_y = smp.getLeftEye().getGaze()

        # Draw the background image
        img.draw()

        # Update the window with the current gaze position
        gaze_window.pos = (gaze_x - SCN_W/2.0, SCN_H/2.0 - gaze_y)
        win.flip()

# Stop recording
tk.stopRecording()
# Put the tracker to offline mode
tk.setOfflineMode()
# Close the EDF data file on the Host
tk.closeDataFile()

# Download the EDF data file from Host
tk.receiveDataFile('psychopy.edf', 'psychopy.edf')
```

```
# Close the link to the tracker
tk.close()

# Close the graphics
win.close()
core.quit()
```

The code relevant to online sample retrieval is much like that in the previous example. The only thing new here is that we use the retrieved gaze data to update the aperture position. Note that, for the coordinates of the retrieved gaze data, the origin is the top-left corner of the screen. Some conversion is needed because the origin of the screen coordinates used in PsychoPy is the screen center.

```
# show the image indefinitely until a key is pressed
gaze_pos = (-32768, -32768)
while not event.getKeys():
    # Check for new samples
    smp = tk.getNewestSample()
    if smp is not None:
        if smp.isRightSample():
            gaze_x, gaze_y = smp.getRightEye().getGaze()
        elif smp.isLeftSample():
            gaze_x, gaze_y = smp.getLeftEye().getGaze()

        # Draw the background image
        img.draw()

        # Update the window with the current gaze position
        gaze_window.pos = (gaze_x - SCN_W/2.0, SCN_H/2.0 - gaze_y)
        win.flip()
```

Accessing Eye Events

As noted earlier, sample data is parsed online by the Host PC during recording to detect eye events, e.g., the onset and offset of saccades, fixations, and blinks. Depending on the tracker settings, both events and sample data can be saved in the EDF data files and made available over the link during testing. It is possible to convert the EDF data file into plain text files (see Chap. 8) to examine the various eye events parsed by the tracker. In the example data lines shown below, we see two pairs of events: SSACC (start of saccade) and ESACC (end of saccade) and SFIX (start of fixation) and EFIX (end of fixation). The letter “L” (or “R”) following the

event label indicates the data comes from the left or right eye. Note that for a "start" event, the timestamp is the only data recorded. For an "end" event, summary data is available, e.g., in the case of the EFIX event, the duration of fixation (2368 ms), the average x and y gaze position of the samples within the fixation (755, 447), and the average pupil size during the fixation (5161).

```
SSACC L 659818
ESACC L 659818 659962 148 749.0 413.0 752.0 437.0 0.35 571
SFIX L 659966
EFIX L 659966 662330 2368 755.0 447.0 5161
```

Eye events are useful in a number of research scenarios. For example, a saccade end event can be used to determine the response accuracy in a pro-/anti-saccade task and then conditionally administer subsequent trials. Online fixation events can be used to monitor whether participant fixed the proper region of the screen before showing the critical display or develop eye-typing applications that generate a key-press or mouse click events depending on where and how long participant looks on the screen.

Commands for Accessing Events

The Pylink library uses predefined constants to represent the types of eye events that can be retrieved online. For instance, if you enter "pylink.STARTBLINK" (without quotes) in the Python shell (after importing Pylink), the shell will return the corresponding constant "3." The Pylink constants for all eye events are listed in Table 6.1. Note that sample data is treated as a special type of "event"—SAMPLE_TYPE, represented by a constant (200).

```
>>> import pylink
>>> pylink.STARTBLINK
3
```

When talking about eye events, people are generally referring to fixations and saccades. However, as you can see from Table 6.1, these are not the only types of events you can retrieve during recording. For example, the Host PC also flags the start and end of blinks. The onset of blinks is flagged when the Host PC can no longer derive a valid X, Y location for gaze. Blink events are "wrapped" in (blink) saccades, i.e., the sequence of events is SSACC, SBLINK, EBLINK, ESACC. The

reason for this is that as the eyelids descend over the pupil, from the camera's perspective, the pupil center is rapidly changing (shifting downward as the eyelid descends and obscures the upper part). The tracker will see a fast downward "movement" of the eyeball and report a saccade start event. When the eyelids reopen, the tracker sees the pupil center quickly stabilizes. The velocity of the perceived "movement" falls below the velocity threshold, leading to the detection of a saccade end event.

In the previous section, we illustrated how to read the latest sample as it arrives, with *getNewestSample()*. The EyeLink API also allows users to access data (sample and event) buffered in a queue in the same order they arrived. Buffering helps prevent data loss if the script fails to read the arrived data immediately; however, accessing data from a queue will introduce a delay if the queue contains significant amounts of data or cause data loss if the queue is not cleared quickly enough. To access event data in the queue, we first call *getNextData()* to check if there is an event there; this command will return a code representing an event (see Table 6.1). If this command returns 0, there is no data (sample or event) in the queue; otherwise, we call *getFloatData()* to grab the data out of the queue. The *getNextData*() and *getFloatData*() function pairs should be called as frequently as possible to empty the queue (e.g., within a *while* loop), so the event we retrieved is the latest one.

As noted, the access of eye events during testing is not real-time. When an eye event (e.g., STARTSACC and ENDFIX) is available over the link, it would have occurred some 20–40 ms ago. The short script below shows how to examine the event retrieval latency. We continuously examine the start and end events for fixations and saccades in a *while* loop. If an event is detected, we log a message that

Table 6.1 Pylink constants for various eye events

Variables	Constants	Description
STARTBLINK	3	Start of blink (pupil disappears)
ENDBLINK	4	End of blink (pupil reappears)
STARTSACC	5	Start of saccade
ENDSACC	6	End of saccade
STARTFIX	7	Start of fixation
ENDFIX	8	End of fixation
FIXUPDATE	9	Update fixation summary data during a fixation
MESSAGEEVENT	24	Message received by tracker
BUTTONEVENT	25	Button press
INPUTEVENT	28	Change of pin status on the Host parallel port
SAMPLE_TYPE	200	Sample data

contains the timestamp of the event (based on the Host PC clock). This message will have a timestamp. By comparing this message timestamp with the event timestamp, we can determine the delay between the event happening and it becoming available over the link.

```
#!/usr/bin/env python3
#
# Filename: event_retrieval.py
# Author: Zhiguo Wang
# Date: 5/26/2021
#
# Description:
# A short script illustrating online retrieval of eye events

import pylink


# Connect to the tracker
tk = pylink.EyeLink('100.1.1.1')

# Open an EDF data file on the Host PC
tk.openDataFile('ev_test.edf')

# Put the tracker in offline mode before we change tracking parameters
tk.setOfflineMode()

# Set sample rate to 1000 Hz
tk.sendCommand('sample_rate 1000')

# Make all types of event data are available over the link
event_flgs = 'LEFT,RIGHT,FIXATION,FIXUPDATE,SACCADE,BLINK,BUTTON,INPUT'
tk.sendCommand(f'link_event_filter = {event_flgs}')

# Open an SDL window for calibration
pylink.openGraphics()

# Set up the camera and calibrate the tracker
tk.doTrackerSetup()

# Put tracker in idle/offline mode before we start recording
tk.setOfflineMode()
```

```
# Start recording
tk.startRecording(1, 1, 1, 1)

# Wait for the block start event to arrive, give a warning
# if no event or sample is available
block_start = tk.waitForBlockStart(100, 1, 1)
if block_start == 0:
    print("ERROR: No link data received!")

# Check eye availability; 0-left, 1-right, 2-binocular
# read data from the right eye if tracking in binocular mode
eye_to_read = tk.eyeAvailable()
if eye_to_read == 2:
    eye_to_read = 1

# Get the current tracker time
t_start = tk.trackerTime()
while True:
    # Break after 5 seconds have elapsed
    if tk.trackerTime() - t_start > 5000:
        break

    # Retrieve the oldest event in the buffer
    dt = tk.getNextData()
    if dt in [pylink.STARTSACC, pylink.ENDSACC,
              pylink.STARTFIX, pylink.ENDFIX]:
        ev = tk.getFloatData()
        # Look for right eye events only; 0-left, 1-right
        if ev.getEye() == eye_to_read:
            # Send a message to the tracker when an event is
            # received over the link; include the timestamp
            # in the message to examine the link delay
            if dt == pylink.STARTSACC:
                tk.sendMessage(f'STARTSACC {ev.getTime()}')
            if dt == pylink.ENDSACC:
                tk.sendMessage(f'ENDSACC {ev.getTime()}')
            if dt == pylink.STARTFIX:
                tk.sendMessage(f'STARTFIX {ev.getTime()}')
            if dt == pylink.ENDFIX:
                tk.sendMessage(f'ENDFIX {ev.getTime()}')

# Stop recording
tk.stopRecording()

# Close the EDF data file on the Host
tk.closeDataFile()
```

```
# Download the EDF data file from Host
tk.receiveDataFile('ev_test.edf', 'ev_test.edf')

# Close the link to the tracker
tk.close()

# Close the window
pylink.closeGraphics()
```

As with sample data, we send over a command (*link_event_filter*) to control what types of event data can be accessed over the link during recording. More information about the event data flags (e.g., FIXATION, SACCADE, BLINK) that can be included in this command are available in the EyeLink user manuals.

```
# Make all types of event data are available over the link
event_flgs = 'LEFT,RIGHT,FIXATION,FIXUPDATE,SACCADE,BLINK,BUTTON,INPUT'
tk.sendCommand(f'link_event_filter = {event_flgs}')
```

A special "block start" event is passed over the link once the recording starts. This event contains information on what data will be available during recording, and we use *waitForBlockStart*() to look for this event in the link queue. The three arguments passed to this function specify how long to wait for the block start event and whether samples, events, or both types of data would be available during recording. This function returns 0 if no data is available over the link.

```
# Wait for the block start event to arrive, give a warning
# if no event or sample is available
block_start = tk.waitForBlockStart(100, 1, 1)
if block_start == 0:
    print("ERROR: No link data received!")
```

Once the block start event is received, we can use *eyeAvailable*() to check if data is available from the left eye, right eye, or both eyes. We need this piece of information to ensure the event data we retrieved is from the eye of interest to us (e.g., during binocular tracking).

```
# Check eye availability; 0-left, 1-right, 2-binocular
# read data from the right eye if tracking in binocular mode
eye_to_read = tk.eyeAvailable()
if eye_to_read == 2:
    eye_to_read = 1
```

At the end of the script, we download the EDF data file from the Host PC with *receiveDataFile*(). We then use the EDF converter provided by SR Research to convert the EDF data file into a plain text file (see Chapter 8 for details). We configure the EDF converter to extract only eye events and messages, ignoring the samples for simplicity.

In the file excerpt shown below, the first data line starts with the keyword "EFIX," marking the end of a fixation. This data line also contains some summary info about the same fixation, i.e., the start time, end time, duration, average gaze position, and average pupil size. This fixation ended at 5725698 ms, followed by a saccade onset event ("SSACC") at 5725699 ms. The third data line contains a message written by our script, running on the Display PC. It starts with the keyword "MSG," and its timestamp is 5725717 ms. This message was logged when our script received the EFIX event over the link. This EFIX event occurred at 5725698 ms; it is the same fixation summarized in the first data line. So, the link delay in retrieving the EFIX event is 5725717 – 5725698 = 19 ms. Using the script described above, the average link delay for saccade onset events (SSACC) is about 18 ms in my testing setup. The link delay for saccade offset events (ESACC) is about 35 ms, and that for fixation onset events (SFIX) is about 34 ms. These delays limit the utility of event data for gaze-contingent tasks. If you want gaze-contingents to happen as soon as possible, use sample data rather than event data.

```
EFIX R 5725568 5725698 131 560.5 294.0 335
SSACC R 5725699
MSG 5725717 ENDFIX 5725698.0
MSG 5725717 STARTSACC 5725699.0
ESACC R 5725699 5725753 55 556.1 291.1 122.6 129.5 13.10 451
SFIX R 5725754
MSG 5725788 ENDSACC 5725753.0
MSG 5725788 STARTFIX 5725754.0
```

You may want to retrieve more information from an eye event, in addition to its timestamp. The code snippet below shows how to retrieve the onset and offset time, amplitude, peak velocity, and start and end positions from an ESACC event.

```
# Current tracker time
t_start = tk.trackerTime()
while True:
    # Break after 5 seconds have elapsed
    if tk.trackerTime() - t_start > 5000:
        break

    # Retrieve the oldest event in the buffer
    dt = tk.getNextData()
    if dt == pylink.ENDSACC:
        ev = tk.getFloatData()
        # Look for right eye events only; 0-left, 1-right
        if ev.getEye() == eye_to_read:
            print('ENDSACC Event: \n',
                  'Amplitude', ev.getAmplitude(), '\n',
                  'Angle', ev.getAngle(), '\n',
                  'AverageVelocity', ev.getAverageVelocity(), '\n',
                  'PeakVelocity', ev.getPeakVelocity(), '\n',
                  'StartTime', ev.getStartTime(), '\n',
                  'StartGaze', ev.getStartGaze(), '\n',
                  'StartHREF', ev.getStartHREF(), '\n',
                  'StartPPD', ev.getStartPPD(), '\n',
                  'StartVelocity', ev.getStartVelocity(), '\n',
                  'EndTime', ev.getEndTime(), '\n',
                  'EndGaze', ev.getEndGaze(), '\n',
                  'EndHREF', ev.getEndHREF(), '\n',
                  'EndPPD', ev.getEndPPD(), '\n',
                  'EndVelocity', ev.getEndVelocity(), '\n',
                  'Eye', ev.getEye(), '\n',
                  'Time', ev.getTime(), '\n',
                  'Type', ev.getType(), '\n')
```

The output from the above code snippet is listed below. The saccade amplitude and the various velocity measures are reported in degrees of visual angles. The *getStartPPD*() and *getEndPPD*() functions return an estimate of how many pixels correspond to 1 degree of visual angle at the start and end of the saccade event. These measures all require a correct screen configuration, either stored on the Host PC or specified in the experimental script (see Chap. 4).

The Pylink commands that can be used to retrieve information from eye events are summarized in Table 6.2. For clarity, we group these commands based on their functionality:

```
ENDSACC Event:
 Amplitude (4.537407467464733, 0.5766124065721234)
 Angle -7.242339197477229
 AverageVelocity 102.0999984741211
 PeakVelocity 175.3000030517578
 StartTime 52627.0
 StartGaze (417.79998779296875, 1739.0999755859375)
 StartHREF (-1450.0, -2441.0)
 StartPPD (48.0, 49.70000076293945)
 StartVelocity 5.800000190734863
 EndTime 52671.0
 EndGaze (633.0999755859375, 1767.699951171875)
 EndHREF (-236.0, -2669.0)
 EndPPD (48.5, 49.20032176294065)
 EndVelocity 37.599998474121094
 Eye 1
 Time 52671.0
 Type 6
```

1. Commands available to all events. In this group of commands, *getTime()* returns the time point at which an event occurred (based on the Host PC clock), *getEye()* returns the eye being tracked (0-left, 1-right, 2-binocular), *getType()* returns the event type (i.e., the constants listed in Table 6.1), and *getStatus()* returns the errors or warnings related to an event if there were any.
2. These commands return the start and end time of an event.
3. These commands return eye position data in GAZE coordinates.
4. These commands return eye position data in HREF coordinates.
5. These commands return the resolution data (PPD, pixels per degree), i.e., how many screen pixels correspond to 1 degree of visual angle.
6. These commands return velocity data.
7. These commands return pupil size data.
8. Commands that are available to the ENDSACC event only; they return saccade amplitude and direction.

One thing worth mentioning here is that the fixation update (FIXUPDATE) event reports the same set of data as the fixation end event (ENDFIX), at predefined intervals (50 ms by default) following the onset of a fixation. In this way, it is possible to access the states of an (ongoing) fixation (e.g., the average gaze position) before it ends. This feature can be useful in HCI applications, e.g., using gaze to drive the mouse cursor.

Table 6.2 Pylink commands for retrieving event data. Here we use "X" to mark the properties available to each eye event

		STARTSACC	ENDSACC	STARTFIX	ENDFIX	STARTBLINK	ENDBLINK	FIXUPDATE
1	getTime()	X	X	X	X	X	X	X
	getEye()	X	X	X	X	X	X	X
	getType()	X	X	X	X	X	X	X
	getStatus()	X	X	X	X	X	X	X
2	getStartTime()	X	X	X	X	X	X	X
	getEndTime()		X		X		X	X
3	getStartGaze()	X	X	X	X			X
	getEndGaze()		X		X			X
	getAverageGaze()				X			X
4	getStartHREF()	X	X	X	X			X
	getEndHREF()		X		X			X
	getAverageHREF()				X			X
5	getStartPPD()	X	X	X	X			X
	getEndPPD()		X		X			X
6	getStartVelocity()	X	X	X	X			X
	getEndVelocity()		X		X			X
	getAverageVelocity()		X		X			X
	getPeakVelocity()		X		X			X
7	getStartPupilSize()			X	X			X
	getEndPupilSize()				X			X
	getAveragePupilSize()				X			X
8	getAmplitude()		X					
	getAngle()		X					

An Example in PsychoPy: Gaze Trigger

While the link delays limit the utility of event data for rapid gaze-contingent changes, the online parsed eye events can still be very useful in many research scenarios. For example, the ENDSACC event can be used to calculate saccadic response time in a visually guided saccade task within the task itself. In other words, the gaze data need not to be analyzed in a separate step; the script itself can calculate the key metric of saccade latency. The FIXUPDATE event, on the other hand, can be used to set up a "gaze trigger," so the task moves to the next step only if the participant has gazed at a given screen position for a certain amount of time (see the example script below).

```
#!/usr/bin/env python3
#
# Filename: gaze_trigger.py
# Author: Zhiguo Wang
# Date: 5/26/2021
#
# Description:
# A gaze trigger implemented in PsychoPy

import pylink
from psychopy import visual, core, event, monitors
from EyeLinkCoreGraphicsPsychoPy import EyeLinkCoreGraphicsPsychoPy
from math import hypot

# Connect to the tracker
tk = pylink.EyeLink('100.1.1.1')

# Open an EDF data file on the Host PC
tk.openDataFile('psychopy.edf')

# Put the tracker in offline mode before we change tracking parameters
tk.setOfflineMode()

# Make all types of eye events available over the link, especially the
# FIXUPDATE event, which reports the current status of a fixation at
# predefined intervals (default = 50 ms)
event_flags = 'LEFT,RIGHT,FIXATION,FIXUPDATE,SACCADE,BLINK,BUTTON,INPUT'
tk.sendCommand(f'link_event_filter = {event_flags}')

# Screen resolution
SCN_W, SCN_H = (1280, 800)

# Open a PsyhocPy window
win = visual.Window((SCN_W, SCN_H), fullscr=False, units='pix')
```

```
# Pass the display pixel coordinates (left, top, right, bottom) to the tracker
coords = f"screen_pixel_coords = 0 0 {SCN_W - 1} {SCN_H - 1}"
tk.sendCommand(coords)

# Request Pylink to use the custom EyeLinkCoreGraphicsPsychoPy library
# to draw calibration graphics (target, camera image, etc.)
genv = EyeLinkCoreGraphicsPsychoPy(tk, win)
pylink.openGraphicsEx(genv)

# Calibrate the tracker
calib_msg = visual.TextStim(win, text='Press ENTER twice to calibrate')
calib_msg.draw()
win.flip()
tk.doTrackerSetup()

# Run 3 trials in a for-loop
# in each trial, first show a fixation dot, wait for the participant
# to gaze at the fixation dot, then present an image for 2 secs
for i in range(3):
    # Prepare the fixation dot in memory
    fix = visual.GratingStim(win, tex='None', mask='circle', size=30.0)

    # Load the image
    img = visual.ImageStim(win, image='woods.jpg', size=(SCN_W, SCN_H))

    # Put tracker in Offline mode before we start recording
    tk.setOfflineMode()

    # Start recording
    tk.startRecording(1, 1, 1, 1)

    # Wait for the block start event to arrive, give a warning
    # if no event or sample is available
    block_start = tk.waitForBlockStart(100, 1, 1)
    if block_start == 0:
        print("ERROR: No link data received!")

    # Check eye availability; 0-left, 1-right, 2-binocular
    # read data from the right eye if tracking in binocular mode
    eye_to_read = tk.eyeAvailable()
    if eye_to_read == 2:
        eye_to_read = 1

    # Show the fixation dot
    fix.draw()
    win.flip()
```

```
    # Gaze trigger
    # wait for gaze on the fixation dot (for a minimum of 300 ms)
    fix_dot_x, fix_dot_y = (SCN_W/2.0, SCN_H/2.0)
    triggered = False
    fixation_start_time = -32768
    while not triggered:
        # Check if any new events are available
        dt = tk.getNextData()
        if dt == pylink.FIXUPDATE:
            ev = tk.getFloatData()
            if ev.getEye() == eye_to_read:
                # 1 deg = ? pixels in the current fixation
                ppd_x, ppd_y = ev.getEndPPD()

                # Get the gaze error
                gaze_x, gaze_y = ev.getAverageGaze()
                gaze_error = hypot((gaze_x - fix_dot_x)/ppd_x,
                                   (gaze_y - fix_dot_y)/ppd_y)

                if gaze_error < 1.5:
                    # Update fixation_start_time, following the first
                    # FIXUPDATE event
                    if fixation_start_time < 0:
                        fixation_start_time = ev.getStartTime()
                    else:
                        # Break if the gaze is on the fixation dot
                        # for > 300 ms
                        if (ev.getEndTime() - fixation_start_time) >= 300:
                            triggered = True
                else:
                    fixation_start_time = -32768

    # Show the image for 2 secs
    img.draw()
    win.flip()
    core.wait(2.0)

    # Clear the screen
    win.color = (0, 0, 0)
    win.flip()
    core.wait(0.5)

    # Stop recording
    tk.stopRecording()

# Close the EDF data file on the Host
tk.closeDataFile()
```

```
# Download the EDF data file from Host
tk.receiveDataFile('psychopy.edf', 'psychopy.edf')

# Close the link to the tracker
tk.close()

# Close the graphics
win.close()
core.quit()
```

This script is similar to the gaze-contingent window task presented in the previous section. One thing worth mentioning here is the use of the FIXUPDATE event. This special eye event allows users to access information about the currently ongoing fixation before it ends. Following the onset of a fixation, the tracker reports information about the ongoing fixation at predefined intervals (50 ms by default) by making a new FIXUPDATE event available over the link. The FIXUPDATE event returns the same set of data as the FIXEND event. In the example script, we use the *getAverageGaze*() command to retrieve the average gaze position during the fixation update interval. We then check if the average gaze position is close to the fixation dot, taking advantage of the resolution data that can be accessed via the FIXUPDATE event. We break out the while loop if gaze is close to the fixation dot (< 1.5 degrees) for more than 300 ms.

```
                # 1 deg = ? pixels in the current fixation
                ppd_x, ppd_y = ev.getEndPPD()

                # Get the gaze error
                gaze_x, gaze_y = ev.getAverageGaze()
                gaze_error = hypot((gaze_x - fix_dot_x)/ppd_x,
                                   (gaze_y - fix_dot_y)/ppd_y)
                if gaze_error < 1.5:
                    # Update fixation_start_time, following the first
                    # FIXUPDATE event
                    if fixation_start_time < 0:
                        fixation_start_time = ev.getStartTime()
                    else:
                        # Break if the gaze is on the fixation dot
                        # for > 300 ms
                        if (ev.getEndTime() - fixation_start_time) >= 300:
                            triggered = True
                else:
                    fixation_start_time = -32768
```

To briefly recap, the EyeLink tracker images the eye(s) up to 2000 times per second, and each eye image is analyzed to estimate the eye (or gaze) position. In addition to the raw eye position data (samples), the tracker can also detect and record eye events during recording. This chapter illustrates how to access the sample and event data over the Ethernet link during recording. In the following chapters, we will cover some more advanced topics on the Pylink library (Chap. 7) and discuss data analysis and visualization in Python (Chap. 8).