Behavior Research Methods
2007, 39 (4), 950-958

PyEPL: A cross-platform experiment-programming library

Aaron S. Geller
University of Pennsylvania, Philadelphia, Pennsylvania

Ian K. Schleifer
Brandeis University, Waltham, Massachusetts

Per B. Sederberg
Princeton University, Princeton, New Jersey

and

Joshua Jacobs and Michael J. Kahana
University of Pennsylvania, Philadelphia, Pennsylvania

PyEPL (the Python Experiment-Programming Library) is a Python library which allows cross-platform and object-oriented coding of behavioral experiments. It provides functions for displaying text and images onscreen, as well as playing and recording sound, and is capable of rendering 3-D virtual environments for spatial-navigation tasks. It is currently tested for Mac OS X and Linux. It interfaces with Activewire USB cards (on Mac OS X) and the parallel port (on Linux) for synchronization of experimental events with physiological recordings. In this article, we first present two sample programs which illustrate core PyEPL features. The examples demonstrate visual stimulus presentation, keyboard input, and simulation and exploration of a simple 3-D environment. We then describe the components and strategies used in implementing PyEPL.

The proliferation of personal computers (PCs) during the past three decades and their ability to control both input and output devices have made them the standard tool for experimentation in the psychological laboratory. Unlike the early days of the PC, when researchers had to write their own low-level code to control experiments, there now exist numerous software tools that enable researchers with relatively modest programming skills to develop sophisticated experiments. Broadly speaking, these tools either take the form of function libraries used in conjunction with scripting languages (e.g., the Psychophysics Toolbox used with the MATLAB scripting language; Brainard, 1997; Pelli, 1997) or they take the form of graphical experiment creation tools that allow users without any formal programming skills to develop experiments (e.g., the E-Prime[1] package sold by Psychology Software Tools, Inc.).

PyEPL (pronounced 'pī-ē-pē-el) is an experiment programming library, written for, and mostly in, the Python programming language. It uses only cross-platform software components, and thus may be used on both Mac OS X and Linux. PyEPL offers several advantages over existing programming libraries and graphical development tools (Table 1). PyEPL adds facilities for recording vocal responses and for the creation of desktop-based virtual-reality experiments to the standard suite of tools commonly available in other packages (e.g., timing, stimulus control, response acquisition, etc.). PyEPL also offers tools for synchronizing behavioral and electrophysiological data. Additional features include screen-refresh synchronization, automated logging of experimental events, and management of participant data hierarchies. Finally, the Python language itself may be considered an additional feature. Python has a very simple syntax, which makes programming minimally daunting for the novice but provides advanced software-engineering features, such as inheritance and polymorphism, for the initiate.

USAGE

Here we present some of the main aspects of experiment coding in PyEPL, by examining two simple experiment programs.

Sample Experiment 1: EasyStroop

The first sample experiment demonstrates the use of PyEPL for simple (screen) output and keyboard input. Specifically, it does the following:

M. J. Kahana, kahana@psych.upenn.edu

Table 1
Features of Interest Provided by Several Experiment Generation Packages and Scripting Languages

Package	Platform	Sound Recording	Sync-Pulsing	VR
E-Prime 1.1	Windows[2]	No	Yes[3]	No
PsyScope X Build 45[4]	Mac OS X[5]	No	Yes	No
SuperLab 4.0	Windows,[2] Mac OS X[5]	Yes[3]	Yes[3]	No
PsyScript[6]	Mac OS 9	Yes	Yes	No
Psychophysics Toolbox	Windows, Mac OS X[5]	No	Yes[3]	No

Note—VR, virtual reality.

1. Shows the instructions to the participant;
2. Presents a textual stimulus on screen; this is a single trial of a forced choice Stroop task, so the participant must choose between two possibilities for the text's color;
3. Waits for the participant's response;
4. Decides whether the response was correct, and gives correct/incorrect feedback with the response time.

We first present the Python code in Listing 1 and explain it. Note at the outset, though, that lines beginning with "#" are comments and are ignored by the Python interpreter. Also note that the line numbers are provided for convenience and are not part of the code.

The `Experiment` class. The first nontrivial line of the script is line 9, which creates an instance of the `Experiment` class. The `Experiment` class manages three functions that pertain to all experiments. Our simple example relies only on the first of these: creating and maintaining a directory structure for each participant. The experiment program is run by invoking the Python interpreter on the experiment script from the command line, and the participant label is passed to PyEPL as a command-line parameter. For example, if the script in Listing 1 were saved in a file called `easyStroop.py`, it could be run from the command line with the command `python easyStroop.py -s sub001`. The `Experiment` object processes the flag `-s sub001` and creates a data directory for participant `sub001`.

The second function served by the `Experiment` class is that of parsing the experiment's configuration file. To facilitate their modification, global experiment variables are typically separated into a separate file called `config.py`. The experimenter sets global variables in `config.py` as a series of Python assignments, as in `numTrials=12`. Upon creation, an `Experiment` instance parses the configuration file and makes the settings available to the experiment program. Maintaining global variables in a separate file has the additional benefit of modularizing the experiment. That is, a specific configuration file can be associated with a specific session for a specific participant. Thus, even if `config.py` should change over time, the settings used for each participant would be known, because the current version of the configuration file is copied into the data directory for each participant for whom the experiment is being run.

The third function served by the `Experiment` class is that of managing experimental state. As described in greater detail below, under Implementation, PyEPL provides functions to facilitate interruption and resumption of experiments. These functions are provided by the `Experiment` class in the methods `saveState` and `restoreState`.

The `Track` classes. On lines 14 and 15, the program instantiates two kinds of `Track` objects. `Track`s, which lie at the center of the PyEPL programming philosophy, entail two concepts. The first is that any experimental quantity that varies over the course of an experiment (i.e., all input and output) should be monitored both comprehensively and without explicit coding by the experimenter. This is a demand for convenient logging of both stimulus presentations (output) and participant responses (input). The second concept entailed in the `Track` philosophy demands that the convenient logging reside inside the stimulus presentation classes (in the case of output) or the response-processing classes (in the case of input).

Thus, when on line 14 the program creates a `VideoTrack` instance, it is doing two important things. First, it sets up the interface by which PyEPL puts images onscreen. That is, it enables a specific mode of stimulus presentation. Second, it prepares PyEPL to log the visual stimulus presentations as they occur. Similarly, when on line 15 the program creates a `KeyTrack` instance, it does two distinct things. It is instructing PyEPL both to *listen for* and *record* any keyboard inputs.

The only *explicit* use of either of our `Track` objects in the program occurs on line 18, where the `VideoTrack` instance is used to clear the screen. However, all screen output and all keyboard input are mediated by the abovementioned `Track` objects.

The `PresentationClock` class. For the class of psychology experiments consisting of a list of stimulus presentations, two kinds of intervals need to be specified. The first is the duration of the stimulus presentation, and the second is the duration of *empty time* between stimuli, or interstimulus interval (ISI). A third kind of interval, during which the program waits for a participant's response, will be considered later, because this duration is not controlled by the experiment program. The `PresentationClock` class is critical for specifying known durations—that is, the first two kinds of intervals.

`PresentationClock`s occur both implicitly and explicitly in several parts of the program. Here we consider their use on lines 37 and 41 of the program, which constitutes the most straightforward application. A `PresentationClock` instance contains the system time and provides the `delay` method to introduce ISIs into a

program. The `delay` call (for example, on line 37) does not actually cause a delay. Instead, it simply increments the time contained by the `PresentationClock` instance; it is thus a simple addition and takes less than 1 msec to execute. Then, when the `PresentationClock` is passed as an argument to a presentation function (as it is on line 41), PyEPL waits to execute the function until the system time is greater than or equal to that contained by the `PresentationClock`. This arrangement greatly improves ISI timing, since it decouples presentation times from program execution latencies. Because ISI delays are executed both as part of the presentation function and in terms of system time stamps (as opposed to offsets relative to prior presentations), they are not affected by the execution time of any code between the call to delay and the call to the presentation function.

Listing 1

```
 1 #!/usr/bin/python
 2
 3 # get access to pyepl objects & functions
 4 from pyepl.locals import *
 5
 6 # create an experiment object:
 7 # parse command-line arguments
 8 # & initialize pyepl subsystems
 9 exp = Experiment()
10
11 # Create a VideoTrack object for interfacing
12 # with monitor, and a KeyTrack object for
13 # interfacing with keyboard
14 vt = VideoTrack("video")
15 kt = KeyTrack("key")
16
17 # reset the display to black
18 vt.clear("black")
19
20 # create a PresentationClock object
21 # for timing
22 clk = PresentationClock()
23
24 # open the instructions file
25 instructions = open("instruct.txt")
26 # show the experiment instructions
27 instruct(instructions.read(), clk = clk)
28
29 # create the stimulus
30 stim = Text("green," color="red")
31
32 # create a ButtonChooser object
33 # to watch for specific keys
34 bc = ButtonChooser(Key("R"), Key("G"))
35
36 # request a 1-second delay
37 clk.delay(1000)
38
39 # delay, then present the test cue,
40 # and process the participant's response
41 ts, b, rt = stim.present(clk=clk,
42                          duration=5000,
43                          bc=bc)
44
45 # time the response
46 response_time = rt[0]-ts[0]
47 # score the response
48 if b==Key("R"):
49 feedback = Text("Correct! %d msec"
50               % response_time)
51 else:
52 feedback = Text("Incorrect! %d msec"
53               % response_time)
54
55 # give feedback for the default duration
56 flashStimulus(feedback, clk=clk)
57
58 # wait for final display to finish
59 clk.wait()
```

Because no delay is introduced before the call to `flashStimulus` on line 56, the use of a `PresentationClock` on that line is not entirely straightforward. In this instance, we take advantage of the fact that presentation functions not only *use* `PresentationClock` for timing but also *update* them in the course of executing. `flashStimulus` increments the `PresentationClock` to reflect the duration of the stimulus presentation (in this case by 1,000 msec, the default duration), but does not block execution for the full duration. Therefore, if the call to `flashStimulus` were the last call before the end of the program, the termination of the program would prematurely cut off the stimulus. We prevent this by calling `clk.wait`, which actually blocks execution until the actual time catches up with the time on the `PresentationClock`.

The use of `clk` on line 27 is similar, in that the `PresentationClock` does not constrain the visual presentation. Here the `instruct` function displays the instructions for the experiment to the participant. These are shown onscreen indefinitely, in a pager environment capable of scrolling backward and forward through the instruction text. Instead of controlling this presentation, the `PresentationClock` is passed as an argument so that it has a fresh time stamp once the program leaves the `instruct` function. Consider the following example: A participant begins the experiment at time t (and therefore the `clk` reads approximately t) and spends 30 sec reading the instructions. Unless the time in `clk` is updated after `instruct` has returned, the call to `delay` will not work at all. This is because it will increment a "stale" time value: that is, the time in `clk` will be incremented from t to $t+1$, but the current time is approximately $t+30$. Because the current time *exceeds* that contained in `clk`, no delay will occur. Thus, even output functions that are not constrained by `PresentationClock`s accept them as arguments and increment them by the duration of the output. Passing a `PresentationClock` into these functions keeps its time stamp fresh—that is, useful for timing control later in the experiment.

The `ButtonChooser` class. Although the `KeyTrack` instance created on line 15 is necessary for processing keyboard input and sufficient for logging of all keystrokes, the keystrokes themselves are not available to the experiment program without further coding. That is, a PyEPL experiment with just a `KeyTrack` will record keystrokes but will not be *aware* of them as they occur in order to respond to them. The program must explicitly expose the keystroke events by creating an object to listen for them. The `ButtonChooser` object created on line 34 does just this. The `ButtonChooser` groups a specific set of keys that may then be monitored as potential stimulus responses. In the example program, the keys "R" and "G" are potential

responses and are grouped in a `ButtonChooser` called `bc`. When `bc` is passed as an argument to the `present` function, as it is on line 43, the stimulus is presented only until one of the keys in `bc` is pressed or until the duration passed as the `duration` argument elapses.

As seen on line 41, the `present` function returns a list of 3 values when used to elicit a keyboard response. The first element of the list is the time of stimulus presentation, the second is the `Key` of the response, and the third is the time of response. This is how an experiment is made interactive with keyboard input in PyEPL.

Experiment output and logs. By default, PyEPL runs in full-screen mode (Figure 1). As mentioned in the section on `Track`s, all screen and keyboard events are automatically logged. Thus, for example, after the experiment has run, the file `key.keylog` (generated automatically by the `KeyTrack`) contains the text found in Listing 2.

The first column of the log contains the time of each event. The second column contains imprecision values for the time stamps in the first column. The motivation for, and implementation of, recording imprecision values is described in the Implementation section, under Timing. The third column codes a key-down event with a "P" (press) and a key-up event with an "R" (release). The fourth column gives the kind of key event.

Listing 2

```
1133749093393   0   B   Logging Begins
1133749104225   1   P   RETURN
1133749104337   0   R   RETURN
1133749106225   0   P   R
1133749106337   1   R   R
1133749107226   0   E   Logging Ends
```

Sample Experiment 2: TinyCab

This program is a stripped-down version of the virtual-navigation task employed by Ekstrom et al. (2003). The program does the following:

1. Simulates a trivial 3-D environment, with a floor, four walls, and sky;
2. Simulates navigation of the environment as the participant drives around it with a joystick;
3. Presents a sprite (a 2-D image always facing the user) at a specific location in the environment;
4. Quits when the participant drives into the sprite.

Again, we present the code (in Listing 3) and then explain it. Because this experiment is substantially more complex than the preceding one, our exposition emphasizes the high-level organizing concepts. We begin with an overview of the program and its underlying strategy.

The programming paradigm differs substantially between Sample Experiments 1 and 2. Whereas Experiment 1 completely specifies a finite sequence of stimuli and their durations, Experiment 2 merely simulates a virtual environment, which is left to the participant to explore. The paradigm therefore shifts from enumerating a stimulus sequence to configuring the virtual environment and the participant's possible interactions with it. With this configuration done, we start PyEPL's `renderLoop` function to allow those interactions to occur—in principle indefinitely.

The organization of our program is as follows:

1. Preliminaries (lines 1–16)
2. Configuration of the environment (lines 20–70)
3. Configuration of the avatar (lines 73–89)
4. Configuration of the `Eye` (lines 92–95)
5. Navigating the environment (lines 99–111)

Preliminaries. The first section of the experiment resembles the beginning of Sample Experiment 1: The purposes of the `Experiment` and `VideoTrack` instances are identical to those in that example. In this experiment, we also instantiate a `VRTrack` (line 13) to gain access to the PyEPL VR application programming interface (API), and a `JoyTrack` (line 16) to enable joystick input.

Environment. Currently, only square environments are supported by PyEPL. To make these environments moderately plausible, we circumscribe these virtual worlds with a visible and impassable barrier—that is, a square wall. Together with the sky (rendered as a large box overhead) and the ground (or floor), these boundaries are the first main category that needs to be configured in our environment.

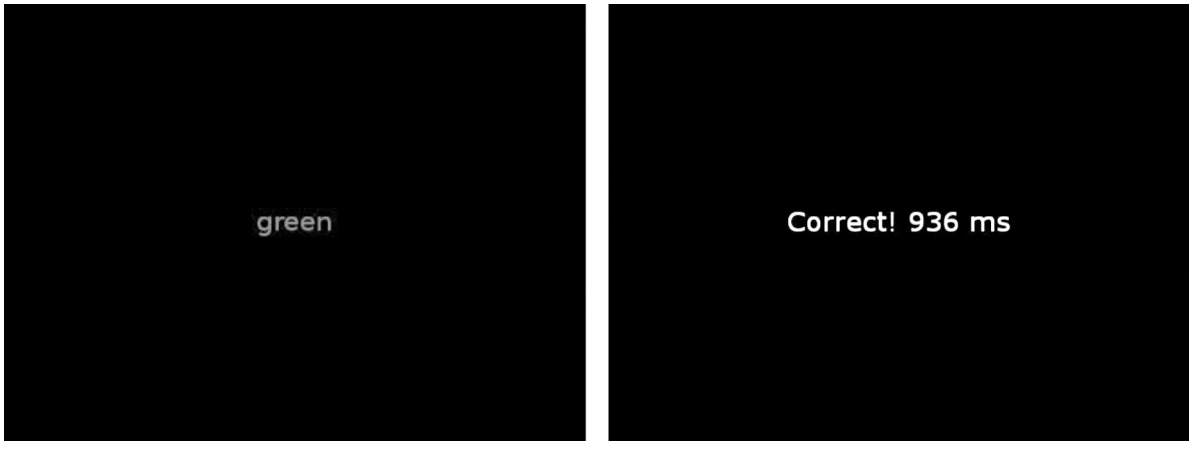

Figure 1. Stimulus and feedback screens from Sample Experiment 1.

Listing 3

```
 1 #!/usr/bin/python
 2
 3 # import PyEPL
 4 from pyepl.locals import *
 5
 6 # create the experiment object
 7 exp = Experiment()
 8
 9 # create the video track
10 video = VideoTrack("video")
11
12 # create the virtual-reality track
13 vr = VRTrack("vr")
14
15 # create the joystick track
16 joystick = JoyTrack("joystick")
17
18 # 'done' remains false until the avatar hits
19 # the sprite
20 done = False
21
22 # add the sky to the environment
23 vr.addSkyBox(Image("sky.png"))
24
25 # add the floor and walls to the environment
26 vr.addFloorBox(x = 0, y = 0, z = 0,
27      xsize = 50.0,
28      ysize = 7.0, zsize = 50.0,
29      floorimage=Image("floor.png"),
30      wallimage=Image("wall.png"))
31
32 # set some gravity
33 vr.setGravity(x = 0.0, y = -0.1, z = 0.0)
34
35 # add an infinite-plane impassible barrier
36 # for the floor
37 vr.addPlaneGeom(a = 0.0, b = 1.0, c = 0.0,
38             d = 0.0, mu = 0.0)
39
40 # add infinite-plane impassible barriers
41 # for the walls (so the avatar
42 # can't travel through them)
43 vr.addPlaneGeom(a = -1.0, b = 0.0,
44             c = 0.0, d = -14.99,
45             mu = 0)
46 vr.addPlaneGeom(a = 1.0, b = 0.0, c = 0.0,
47             d = -14.99, mu = 0)
48 vr.addPlaneGeom(a = 0.0, b = 0.0, c = 1.0,
49             d = -14.99, mu = 0)
50 vr.addPlaneGeom(a = 0.0, b = 0.0, c = -1.0,
51             d = -14.99, mu = 0)
52
53 # add a sprite at 10.0, 0.0, 10.0
54 vr.addSprite(x = 10.0, y = 0.0, z = 10.0,
55      image = Image("sprite.png"),
56      xsize = 3.0, ysize = 5.0)
57
58 # when the avatar hits the sprite,
59 # this function will be called
60 def hitTheSprite():
61     global done
62     done = True
63
64 # add a permeable invisible sphere in the
65 # same place as the sprite
66 vr.addSphereGeom(x = 10.0, y = 0.5,
67             z = 10.0,
68             radius = 2.2,
69             permeable = True,
70             callback = hitTheSprite)
71
72 # create the subject's avatar
73 av = vr.newAvatar("subject_avatar,"
74             eyeheight = 1.0,
75             radius = 0.5)
76
77 # create an axis which will control
78 # the avatar's turn speed
79 turning = JoyAxis(0, 0) * 0.0009
80
81 # create an axis which will control
82 # the avatar's forward speed
83 forward = ScaledAxis(JoyAxis(0, 1),
84             -0.0025, -0.009)
85
86 # set the avatar's forward and
87 # turn speed controls
88 av.setControls(forward_speed = forward,
89      yaw_speed = turning)
90
91 # create the eye object
92 eye = av.newEye("subject_view")
93
94 # set the eye's field of view to 60.0 degrees
95 eye.setFOV(60.0)
96
97 # show the eye view so that it fills
98 # the whole screen
99 video.show(eye, 0, 0)
100
101 # this function will be called by PyEPL as part
102 # of the render loop. When it returns False,
103 # the render loop will finish
104 def checkStillGoing(t):
105     global done
106     global video
107     video.updateScreen()
108     return not done
109
110 # Start the render loop
111 video.renderLoop(checkStillGoing)
```

This is done in two steps. First (on lines 23–30), we describe the barriers' appearance. Then (on lines 37–51) we give them virtual physical properties—specifically, the property of impassability.

The other kind of object requiring configuration in our experiment is the sprite (see Figure 2). In contrast to the boundaries, whose rendering includes both scaling to convey proximity and deforming to convey orientation, the sprite is rendered only to convey distance; its display does not vary with orientation and always faces the `Eye` (concerning which, see below).

The interface for sprite configuration parallels that for the walls. Its visual component is set up first (lines 54–56), followed by its (virtual) physical properties (lines 60–70). The sprite configuration, however, has an added wrinkle, in that the program must *do* something should the participant's avatar (concerning which, see below) collide with it: It must quit. To arrange this, we create a function

to be called just in case of such a collision, and register it with the PyEPL VR module (line 70).

Avatar. The avatar is the participant's virtual body, fixing him or her in a specific location and endowing him or her with specific traits, such as velocity and mass, and potentially an appearance (although not in our example). From a programming point of view, the avatar is a data structure that contains these traits (minimally, the virtual location and velocity) and continually updates their values on the basis of input from the participant. This experiment demonstrates creating an avatar (line 73) and configuring it for joystick control (lines 79–89).

`Eye`. The final component of the VR system that needs to be set up is the `Eye`. The `Eye` is essentially the virtual camera; it picks a specific view of the environment and renders it on screen. To provide the feeling of exploring the virtual environment, the `Eye` in our experiment follows the avatar. That is, the view on screen is updated to reproduce what is viewable at the current position of, and in the current direction of, the participant. An `Eye` need not be fixed with an avatar, however; it is possible, for example, to render the environment as seen from a fixed position, as in a bird's-eye view.

The `Eye` is fixed on the avatar by calling the avatar's `newEye` method on line 92, and its field of view is set on line 95.

Starting the navigation. The final step in the program is to start the navigation. This has two parts: displaying the initial view from the `Eye` (line 99), and calling the `renderLoop` function to start the simulation. On line 111, the `renderLoop` is started, with the function `checkStillGoing` as an argument. On each iteration of the `renderLoop`, PyEPL calls the function that the loop was given (`checkStillGoing` in this case); if the function returns `True`, the simulation continues. In our experiment, `checkStillGoing` inspects the global variable `done`, which records whether the participant's avatar has collided with the sprite.

Logs. Just as in Sample Experiment 1, the `Track` instances used in Experiment 2 provide automated and comprehensive logging. The `JoyTrack` creates a log called `joystick.joylog`, which records the (*x*, *y*) displacement of every joystick manipulation. More importantly, the `VRTrack` creates a log called `vr.vrlog`, which records the virtual 3-D coordinates of the avatar at each iteration of the `renderLoop`, as well as its virtual yaw, pitch, and roll.

Figure 2. Screen shot of Sample Experiment 2, showing sprite (person with hand extended).

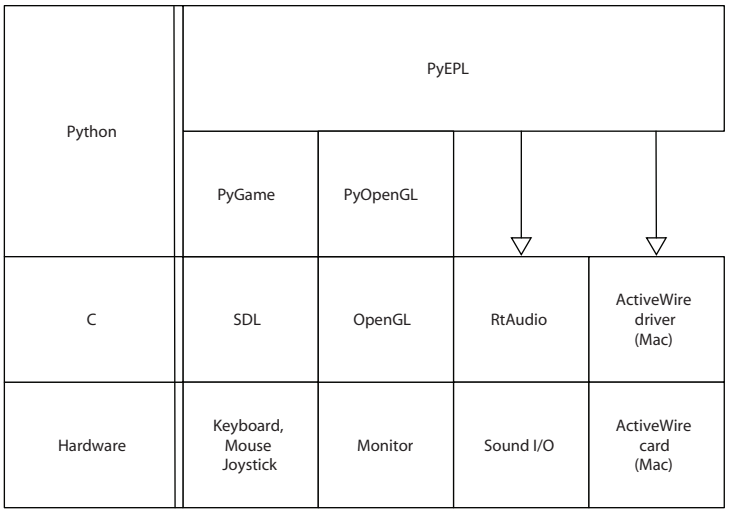

Figure 3. The software infrastructure of PyEPL and its hardware interfaces. The leftmost boxes describe three system layers: Python, C, and hardware. SDL is the Simple DirectMedia Layer, a cross-platform library for hardware management. The rightmost boxes describe the system for sync-pulsing via an ActiveWire card, which is only done on Mac OS X. On Linux, PyEPL accesses the serial port via the kernel.

IMPLEMENTATION

Timing

Timing is perhaps the most basic aspect of any experiment programming library. It is essential for controlling the duration of stimulus presentation, for measuring the response latencies, and for synchronizing behavioral and physiological measurements. Because operating systems on nearly all PCs perform multitasking, accurate timing presents a serious challenge to the experimenter. These problems are exacerbated by an interpreted language such as Python, which can never be as fast as a compiled language.

In PyEPL, we address the timing problem in two ways. The first strategy is to optimize critical code by writing it in C. The second is to provide an estimate of the precision of all measured event times by associating a lag value with each event. We describe both strategies here.

To minimize any extra latency in timing due to code interpretation, critical sections of PyEPL are implemented as compiled code invoked by the interpreter. Two software layers participate in this strategy. First, the main libraries that PyEPL uses to interact with hardware are all C code. These include SDL, the Simple DirectMedia Layer (2005), which PyEPL uses to manage both manual input (keyboard, mouse, and joystick) and video output, and RtAudio (Scavone, 2005), which manages audio input and output (see Figure 3).

The second layer of compiled software is a set of PyEPL functions that frequently interact with the preceding layer. The most important of these is the `pollEvents` function, which monitors manual inputs. PyEPL uses PyGame's (2005) event loop to manage hardware inputs. User inputs such as keystrokes or joystick manipulations are available as PyGame events, and PyEPL continually checks for these and logs them.

The `pollEvents` function also exemplifies the second strategy mentioned above, providing a precision estimate for event times. Using the PyGame event loop is a powerful tool for simplifying experiment code, especially for programs such VR tasks that receive continual user inputs. This technique comes with a potential degradation in timing accuracy, however.[7] The problem is that PyGame's event functions provide only the *kind* of events that have occurred (e.g., "key F depressed") but not the time at which they occurred. Timing events that are registered in PyGame's event loop thus depends on rapid, continual checking for these events.

A full benchmark of a PyEPL system, including measuring the hardware latencies of all input and output devices, along the lines of the BlackBox Toolkit (Plant, Hammond, & Turner, 2004) is beyond the scope of this article. Furthermore, it would obscure the fact that PyEPL is a cross-platform software package, capable of running on diverse hardware and operating systems. What we can present is an overview of the notion of *software timing*, the manner in which PyEPL monitors for imprecision due to its own execution.

PyEPL compensates for the potential inaccuracy in its event loop by recording the time elapsed between `pollEvents` calls, which defines the maximum error in their time stamp. The time stamps thus take the form of ordered pairs (t, ϵ), where t is the time of the previous iteration of the event loop, and ϵ is the time between the previous iteration and the current one. Any events discovered by PyEPL are thus guaranteed to have occurred between t and $t + \epsilon$. For 153,584 key events recorded during a typical experiment on a Mac G5 tower running OS 10.3 with Python 2.3, we observed a mean ϵ of 0.96 msec. By comparison, the MATLAB Psychophysics Toolbox running on the same machine had a mean ϵ of 4.1 msec for keystrokes.

A similar error estimate is provided for output events. In general, output functions in PyEPL do not block; that is, execution proceeds to the program's next line as soon as the particular function has started running. This makes possible a timing precision estimation as follows: Any output operation (such as drawing to the screen or playing a sound) is preceded and followed by checking the system time. These two time stamps again define an interval inside which the output function is known to have run.

Screen Refresh Synchronization

To provide maximum timing precision of visual stimulus onset, PyEPL makes use of double-buffering and of vertical-blanking loop (VBL) synchronization functionality provided by the OpenGL library.[8] Double-buffering allows PyEPL to draw the next screen before it is necessary to show it, and then to simply flip the new screen to the display when it is ready. With OpenGL, the screen flip can be made to wait until the next vertical blank before it executes. This ensures that the entire screen is updated at once and that PyEPL knows when that update has occurred. Consequently, when synchronization to the VBL is activated in PyEPL, the experimenter can know (within 1 msec) when a stimulus is presented to the participant.

When synchronized to the vertical refresh, screen updates are assumed to have substantial delays (on a 60-Hz monitor, up to 16.6 msec). Thus, they constitute an exception to the time-stamping regime described above; instead of returning time stamps both from before and after the screen update has run, only the postupdate time stamp is returned. The time of this stamp is known to be approximately 1 msec after the actual time of the update.

Synchronization Pulsing

Given the widespread interest in the neural substrates of cognition, the desire to conjoin behavioral cognitive tasks with electrophysiological measures is natural. We will take electroencephalographic, or EEG, recordings as an example. In order to precisely characterize the behavioral import of a particular segment of EEG, however, it is necessary to know with maximal precision when, with respect to the behavioral paradigm, a given electrophysiological signal was observed. This requires a mechanism that allows the experimenter to interconvert two kinds of time stamps: those associated with physiological signals on the one hand, and those associated with behavioral events on the other.

PyEPL achieves this synchronization by sending intermittent, brief pulses to the recording equipment throughout the behavioral task. By recording when the pulse was sent (via PyEPL) and observing when the pulse was received (as recorded by the EEG equipment), one can easily compute the mapping to interconvert the two sets of time stamps.

EEG synchronization pulsing (sync-pulsing) in PyEPL is one instance in which cross-platform uniformity is not possible. Most Intel-based Linux systems have parallel ports that can be used for sending transistor–transistor logic (TTL) pulses for synchronization. On Macs, which lack parallel ports, we send sync-pulses from an ActiveWire USB card.[9] Our sync-pulsing routines check which platform they are running on and select the appropriate port.

Virtual Reality (VR)

PyEPL uses another C library, the Open Dynamics Engine (ODE;[10] Smith, 2005), together with PyGame and OpenGL, to create a VR API. PyEPL uses ODE to manage nongraphical aspects of VR. This includes modeling the boundaries of the virtual environment, the locations and trajectories of objects in it, and their interactions. Interactions generally correspond to the action of a force, either mechanical (as in a collision) or the presence of friction or gravity; any or none of these interactions may be included in a given virtual world. PyEPL visualizes this nonvisual environment by maintaining a special virtual object, called an `Eye`, that selects a specific view of the virtual world. In a VR program, PyEPL runs in a loop that computes what the `Eye` currently sees, and uses OpenGL to render this view to the screen. As mentioned above, PyEPL uses PyGame to receive input from hardware controllers, such as a joystick or keyboard.

Sound

PyEPL's sound module is another component implemented in C but exposed to Python. Using the cross-platform RtAudio (Scavone, 2005) library to gain low-level access to the audio devices, we coded two circular buffers, one providing read (record) functionality, the other providing write (play) functionality. This code is exposed to Python with the SWIG (Beazley, 2005) utility.

Sound file I/O is implemented with the libsndfile (Castro Lopo, 2005b) and libsamplerate (Castro Lopo, 2005a) libraries. These enable PyEPL to play sound from all conventional file formats and sampling rates. File output is currently restricted to 44100-Hz WAV format.

State Management

For experiments run in noncontrolled environments, easy interruption and resumption of experiments is critical, and this requires saving to disk (serializing) one or more program variables. For example, if the participant is a hospital patient, the testing session may be interrupted for various nonexperimental exigencies like administration of medication or physiological diagnostics. To cope with these possibilities, convenient stopping and resumption of the experiment is essential. The implications for the experiment program are as follows. In general, experiment programs consist of a loop that iterates through a list of trials, so this state management task entails, at a minimum, saving both the list of trials and the index of the trial to be run. PyEPL uses the Python `pickle` serialization module to record experimental state variables.

CONCLUSIONS

PyEPL makes the coding of feature-rich experiments quite easy. The sample experiments described above can, with a modicum of effort, be scaled up into real experiments. Furthermore, the features made accessible by PyEPL are provided by very few programming utilities that are currently available. Future directions for PyEPL development include enhancing our documentation and sample code base and streamlining the installation of PyEPL and its dependencies.

RESOURCES

PyEPL is available from pyepl.sourceforge.net. This Web site provides instructions for PyEPL installation, documentation, a user forum, and the PyEPL distributions themselves. Currently, installation instructions are available for Mac OS X (both PowerPC and Intel) as well as Linux. PyEPL was downloaded over 400 times in 2006, and its user base continues to grow.

AUTHOR NOTE

We acknowledge support from NIH Grants MH55687, MH61975, and MH62196; NSF (CELEST) Grant SBE-354378; and the Swartz Foundation. We thank Jacob Wiseman for his work on the installation system. The PyEPL project grew out of earlier C/C++ libraries developed by M.J.K., J.J., Daniil Utin, Igor Khazan, Marc Howard, Abraham Schneider, Daniel Rizzuto, Jeremy Caplan, Kelly Addis, Travis Gebhardt, and Benjamin Burack. Correspondence concerning this article may be addressed to M. J. Kahana, University of Pennsylvania, 3401C Walnut Street, Room 302C, Philadelphia, PA 19104 (e-mail: kahana@psych.upenn.edu).

REFERENCES

BAAS, M. (2005). PyODE: Python bindings for the Open Dynamics Engine [Computer software]. Retrieved December 29, 2005, from pyode.sourceforge.net.

BATES, T., & D'OLIVERO, L. (2003). Psyscript: A Macintosh application for scripting experiments. *Behavior Research Methods, Instruments, & Computers*, **4**, 565-576.

BEAZLEY, D. (2005). SWIG: Simplified Wrapper and Interface Generator [Computer software]. Retrieved December 28, 2005, from www.swig.org.

BRAINARD, D. H. (1997). The Psychophysics Toolbox. *Spatial Vision*, **10**, 443-446.

CASTRO LOPO, E. DE (2005a). Libsamplerate [Computer software]. Retrieved December 28, 2005, from www.mega-nerd.com/SRC/.

CASTRO LOPO, E. DE (2005b). Libsndfile [Computer software]. Retrieved December 28, 2005, from www.mega-nerd.com/libsndfile/.

COHEN, J. D., MACWHINNEY, B., FLATT, M., & PROVOST, J. (1993). PsyScope: A new graphic interactive environment for designing psychology experiments. *Behavior Research Methods, Instruments, & Computers*, **25**, 257-271.

EKSTROM, A. D., KAHANA, M. J., CAPLAN, J. B., FIELDS, T. A., ISHAM, E. A., NEWMAN, E. L., ET AL. (2003). Cellular networks underlying human spatial navigation. *Nature*, **425**, 184-187.

KECK, D. (2007). ActiveWire driver & interfaces for OS X [Computer software]. Retrieved January 17, 2007, from sourceforge.net/projects/activewire-osx/.

MACWHINNEY, B., ST. JAMES, J., SCHUNN, C., LI, P., & SCHNEIDER, W. (2001). STEP—A system for teaching experimental psychology using E-Prime. *Behavior Research Methods, Instruments, & Computers*, **33**, 287-296.

PELLI, D. G. (1997). The VideoToolbox software for visual psychophysics: Transforming numbers into movies. *Spatial Vision*, **10**, 437-442.

PLANT, R. R., HAMMOND, N., & TURNER, G. (2004). Self-validating presentation and response timing in cognitive paradigms: How and why? *Behavior Research Methods, Instruments, & Computers*, **36**, 291-303.

PYGAME (2005). [Computer software]. Retrieved December 28, 2005, from www.pygame.org.

SCAVONE, G. P. (2005). RtAudio [Computer software]. Retrieved December 28, 2005, from www.music.mcgill.ca/~gary/rtaudio/.

SIMPLE DIRECTMEDIA LAYER (2005). [Computer software]. Retrieved December 28, 2005, from www.libsdl.org.

SMITH, R. (2005). Open Dynamics Engine [Computer software]. Retrieved December 28, 2005, from www.ode.org.

STRAW, A. (2006). VisionEgg [Computer software]. Retrieved January 25, 2006, from visionegg.org.

NOTES

1. See MacWhinney, St. James, Schunn, Li, and Schneider (2001).
2. Versions: 95 and later.
3. Functionality exists via third-party hardware and/or software.
4. See Cohen, MacWhinney, Flatt, and Provost (1993).
5. Also runs on Mac OS 9 and earlier.
6. See Bates and D'Olivero (2003).
7. See, e.g., www.pygame.org/docs/tut/newbieguide.html, Sec 11.
8. In implementing this, we have drawn on work by Straw (2006).
9. Mac OS X drivers provided by Keck (2007).
10. Accessed via the PyODE module (Baas, 2005).

(Manuscript received January 26, 2006;
revision accepted for publication December 27, 2006.)