Behav Res (2013) 45:1293–1312
DOI 10.3758/s13428-013-0322-5

PandaEPL: A library for programming spatial navigation experiments

Alec Solway · Jonathan F. Miller · Michael J. Kahana

Published online: 3 April 2013

Abstract Recent advances in neuroimaging and neural recording techniques have enabled researchers to make significant progress in understanding the neural mechanisms underlying human spatial navigation. Because these techniques generally require participants to remain stationary, computer-generated virtual environments are used. We introduce PandaEPL, a programming library for the Python language designed to simplify the creation of computer-controlled spatial-navigation experiments. PandaEPL is built on top of Panda3D, a modern open-source game engine. It allows users to construct three-dimensional environments that participants can navigate from a first-person perspective. Sound playback and recording and also joystick support are provided through the use of additional optional libraries. PandaEPL also handles many tasks common to all cognitive experiments, including managing configuration files, logging all internal and participant-generated events, and keeping track of the experiment state. We describe how PandaEPL compares with other software for building spatial-navigation experiments and walk the reader through the process of creating a fully functional experiment.

Keywords Experiment creation · Programming library · Spatial navigation · Virtual reality · Stimulus presentation

Since the discovery of place cells in the 1970s (O'Keefe & Dostrovsky, 1971; O'Keefe & Nadel, 1978), researchers have engaged in a large and sustained effort to understand the neural mechanisms underlying spatial navigation. The widespread availability of noninvasive recording techniques such as scalp electroencephalography (EEG) and functional magnetic resonance imaging, combined with the rise in cognitive research involving neurosurgical patients with implanted electrodes, have made it possible to study these mechanisms in humans (e.g., Alvarez, Biggs, Chen, Pine, & Grillon, 2008; Astur, Taylor, Mamelak, Philpott, & Sutherland, 2002; Cornwell, Johnson, Holroyd, Carver, & Grillon, 2008; Doeller, Barry, & Burgess, 2010; Doeller, King, & Burgess, 2008; Ekstrom, Copara, Isham, Wang, & Yonelinas, 2011; Ekstrom et al., 2003; Gron, Wunderlich, Spitzer, Tomczak, & Riepe, 2000; Hassabis et al., 2009; Iaria, Chen, Guariglia, Ptito, & Petrides, 2007; Jacobs, Kahana, Ekstrom, Mollison, & Fried, 2010; Jacobs, Korolev, et al., 2010; Maguire et al., 1998; Shipman & Astur, 2008; Suthana et al., 2012; van der Ham et al., 2010; Watrous, Fried, & Ekstrom, 2011; Weidemann, Mollison, & Kahana, 2009). Because these techniques require participants to remain stationary throughout the recording period, researchers often use computer-controlled virtual reality (VR) environments.

Many VR experiments, including those listed above, have the same basic structure and requirements. Participants navigate the environment from a first-person perspective, triggering events when they arrive at particular locations. Instructions between runs are presented using 2-D text and images superimposed on the 3-D environment. Input is processed using a computer keyboard or joystick, and every event, either experiment- or participant-generated, is recorded in a text-based log. In some cases, multiple computers are used: for example, one to run the experiment, and another to record neural data. Such setups require that the computers be synchronized through the use of additional hardware.

We have designed a programming library that automates these tasks and at the same time leaves users the flexibility

A. Solway
Princeton University, Princeton, NJ, USA

J. F. Miller
Drexel University, Philadelphia, PA, USA

M. J. Kahana (✉)
University of Pennsylvania, 3401 Walnut St. Suite 303C, Philadelphia, PA 19104, USA
e-mail: kahana@psych.upenn.edu

necessary to implement novel experimental paradigms. Our library, called PandaEPL, is built on top of the open-source Panda3D game engine. Panda3D is a fully fledged modern game engine with support for 3-D graphics; efficient rendering; basic physics; fog, lighting, and particle effects; shaders; flexible camera control; and mouse and keyboard input. Panda3D was developed by the Disney corporation and subsequently released as open-source software.

With PandaEPL, researchers can design rich and complex environments using the 3-D modeling software of their choice, and participants can then navigate these environments from a first-person perspective. User-defined events can be triggered automatically when participants collide with objects in the environment, or more complex contingencies can be created by executing user-defined code before each frame is rendered. Helper functions display text and images in 2-D, allowing users to display instructions and construct custom heads-up displays (e.g., to display a compass, the score, or the time). A number of other features simplify tasks that are common to all cognitive experiments, including configuration and log-file management and computer synchronization. The basic use of the software does not require knowledge of Panda3D or of any other library. However, a rudimentary knowledge of Python is required.

Our presentation here proceeds as follows. First, we describe how PandaEPL compares with existing software for creating spatial-navigation experiments. Next, we discuss how PandaEPL handles precise timing, an important issue for all psychology experiments. We then demonstrate how PandaEPL can be used to build an experiment by leading the reader through a fully functional example. The example is a reduced version of Experiment 2 of Miller, Lazarus, Polyn, and Kahana (in press). Finally, additional details about the library are provided in the Appendix.

Comparison with existing approaches

We know of only two existing general-purpose software packages aimed at the creation of spatial navigation experiments (Ayaz, Allen, Platek, & Onaral, 2008; Geller, Schleifer, Sederberg, Jacobs, & Kahana, 2007).

Geller et al. (2007) reported on a general-purpose Python-based Experiment Programming Library, or PyEPL. PyEPL streamlines many of the common development tasks shared by cognitive experiments by making available a set of Python-based classes and functions. This allows researchers to quickly and easily create new experiments, while at the same time leaving them the flexibility necessary to program novel features. As a whole, PyEPL is aimed at general experiment development. Its "virtual reality module" simplifies the construction of spatial-navigation experiments in particular. We continue to advocate the use of PyEPL for experiments using 2-D stimuli (here we will not reenumerate PyEPL's particular benefits; see Geller et al., 2007); however, PyEPL's VR module is now outdated. Users can build only simple 3-D environments by placing different cubes on a grid and applying textures to them. Irregularly shaped objects can only be presented as 2-D sprites that appear the same from every angle. Missing from PyEPL is the ability to load and render true 3-D models created with modern graphic design packages. This makes the constructed environments both less attractive and more cumbersome to create. Moreover, because rendering is controlled by a basic graphics engine built in-house, even rudimentary environments can be slow to render. PyEPL is officially supported on Mac OS X and Linux.

Ayaz et al. (2008) reported on Maze Suite, a set of standalone tools aimed at the construction of spatial-navigation experiments. It consists of three programs: MazeMaker, a point-and-click interface for constructing new environments; MazeWalker, the graphics and game engine; and MazeViewer, a tool that aids with data analysis. Maze Suite allows users to create and analyze basic experiments without having to learn a programming language, and for simple environments, without having to learn how to use graphic design software. However, the range of experiments that can be constructed and the analyses that can be performed are limited by the built-in capabilities of the point-and-click interface. Because Maze Suite is a set of standalone programs rather than a programming library, its features cannot be extended or used in novel ways by being invoked from a programming language. Maze Suite runs only on Microsoft Windows systems.

The approach that we take is similar to that of PyEPL. PandaEPL makes available a set of Python-based classes and functions that encapsulate many common development tasks, but it is up to the experiment writer to decide how to best deploy these features to construct a particular experiment. There is not a single "pipeline" through which all new experiments are created. The advantages and disadvantages of such an approach are obvious: It provides for a large amount of flexibility in creating new experiments, but it has a steeper learning curve than a specialized point-and-click interface such as the one provided by Maze Suite.

However, unlike PyEPL, which forces users to design all aspects of their experiment programmatically, PandaEPL allows users to construct the most cumbersome part, the 3-D environment itself, using the point-and-click interface in the graphics package of their choice. The 3-D environment can then easily be loaded in, either as a single model or by piecing together several separately constructed models. Moreover, whereas the only 3-D objects supported by PyEPL are cubes to which users apply different textures, PandaEPL supports arbitrarily complex 3-D models created by packages such as 3ds Max (Autodesk, Inc., San Rafael,

CA), AutoCAD (Autodesk, Inc.), Blender (Blender Foundation, Amsterdam, The Netherlands), LightWave 3-D (NewTek, Inc., San Antonio, TX), Maya (Autodesk, Inc.), and others. This allows users to create significantly more realistic and engaging experiments than are allowed by PyEPL. This ability is made possible by using Panda3D under the hood, and herein lies the main difference between PandaEPL and PyEPL's VR module. PandaEPL uses Panda3D to handle all aspects of the 3-D environment, a task that Panda3D was specifically designed for. In addition to supporting a wide range of 3-D graphics formats, Panda3D has an efficient rendering engine and collision detection system, and includes support for fog and lighting effects, custom shaders, and more. Panda3D is open source and continues to be developed by the open-source community. As new features are developed, they are automatically available for use in new experiments. PandaEPL is thus not simply an incremental update to PyEPL's VR module, but instead represents a completely different approach to experiment development. On top of the features made available by Panda3D, PandaEPL provides the necessities common to all cognitive experiments: precise millisecond timing, logging of behavioral data, configuration file management, and synchronization of multiple computers. PandaEPL runs on Microsoft Windows, Mac OS X, and Linux.

A different approach to building spatial navigation experiments involves the use of custom-built software. Often this has involved modifying game engines that have been made open source (e.g., Doeller et al., 2010; Doeller et al., 2008; Hassabis et al., 2009; Iaria et al., 2007; Maguire et al., 1998), although some researchers have opted to create software from scratch (e.g., Gron et al., 2000; van der Ham et al., 2010). There are at least three disadvantages to this approach. First, whether co-opting an existing game engine or writing the software from scratch, this approach requires programming the features that we previously identified as being shared by all or most experiments. Second, once the software is built, it may be more difficult to reuse beyond a single experiment or set of experiments, as compared with a general-purpose programming library. Finally, because the software is used in few experiments, it is not tested as extensively as software used by a broader community of researchers.

Timing

Obtaining precise timing information about both experiment- and participant-generated events is important for all psychology experiments. PandaEPL's approach to timing is similar to that of PyEPL (Geller et al., 2007). Most common consumer operating systems do not have real-time capabilities and cannot provide guarantees about the exact timing of events. However, it is still possible to obtain a reasonable bound on the occurrence of an event by querying the time before and the time after a block of code has executed. Because the operating system may preempt the control of program flow, these values are approximate. However, we can be sure that the estimated start time is less than or equal to the actual start time and that the estimated duration is greater than or equal to the actual duration. Each event is thus guaranteed to occur sometime within the recorded range. On modern computers, this range is often very small for short functions (1–2 ms on the computers that we use in our lab) and provides excellent bounds on the exact timing of an event.[1] Moreover, it is possible to set a threshold on event duration and to discard the few events with larger-than-normal execution latencies. Additional information on timing is provided in the Appendix.

Building an experiment

In order to demonstrate how PandaEPL can be used to create a new experiment, we now lead the reader through the construction of a fully functional example. This example is a reduced version of Experiment 2 of Miller et al. (in press). The complete source code, together with all of the 3-D models and other helper files necessary to run the example, can be downloaded from http://memory.psych.upenn.edu/PandaEPL.

Knowledge of Python is necessary in order to use PandaEPL effectively. In what follows, we assume that the reader has a rudimentary familiarity with the language. However, before we proceed with the example, we will briefly review some of the advanced Python features that PandaEPL makes extensive use of. Python experts can skip the next few paragraphs and proceed directly to the Overview of the Experiment section. First, we review Python's class system. A *class* is a user-defined collection of methods (standard functions declared using Python's `def` mechanism) and variables. An *object* is a particular instance of a class. When we create a new object, we are said to *instantiate* a copy of the class. Each object has a separate copy of the class variables; when a class variable is changed in one object, other copies are unaffected. Objects are instantiated using a class *constructor*, a method that has the same name as the class and, like other methods, takes zero or more arguments.[2] After a copy of the class is

[1] The precision of events that change the display is actually lower and depends on the screen refresh rate and other factors. See the Appendix for more information.

[2] In reality, the constructor has at least one argument: the class that is to be instantiated. However, this detail is unimportant for the present discussion.

constructed, the class's `__init__` method is automatically invoked if it is defined. It receives a copy of the new object in its first argument, followed by the values passed to the class constructor. We previously said that class methods are standard functions declared using Python's `def` mechanism. This is almost true. In addition to having the properties of standard functions, class methods have the special property that their first argument is always the object calling the method. The name of this argument is always `self`. For example:

```
1. class foo:
2.     def __init__(self, a):
3.         print "Object initialized with value: ", a
4.     def start(self):
5.         print "We are going to start the experiment..."
```

Methods are dispatched using *dot notation*. To demonstrate, the following code snippet creates an instance of class `foo`, defined above, and calls the `start` method:

```
1. f = foo(42)
2. f.start()
```

It is also possible to have *static class members*, or variables and methods that belong to the whole class rather than to a specific object. Static members are also accessed using dot notation, by using the class name before the dot. Static members play a particularly important role in PandaEPL. While most classes are defined with the goal of instantiating multiple object copies, some classes are meant to be instantiated only once. Such classes are called *singletons*. For example, the PandaEPL `Experiment` class is a singleton that maintains information about the experiment state. It would not make sense to have multiple `Experiment` objects, because each participant can only run in one experiment at a time. Singleton objects in PandaEPL are accessed by invoking the static `getInstance()` method of the corresponding the class. This will be demonstrated and used throughout the example below.

Finally, PandaEPL also makes extensive use of *callbacks*. These are simply references to standard functions or class methods. Just as it is possible to assign a primitive value, such as the number 5, to a variable, it is also possible to assign it a function or class method. This variable can then be used to call the corresponding function or class method, and just like other variables, it can be passed as an argument to other functions and class methods. Functions can also be created "on the fly." Such functions are called *anonymous*, because they are not associated with any one particular name. They are defined using the keyword `lambda`. For example:

```
1. lambda a, b: a+b
```

defines a function that accepts two arguments and returns their sum. We can assign a function declared in this way to a variable: for example,

```
1. foo = lambda a, b: a+b
```

We can also pass such a function directly as an argument to another function or method.

Overview of the experiment

The experiment consists of a cover task in which the participant plays the role of a delivery person, navigating and delivering packages from one location to another in a small town. A screenshot of one location in the town, as seen by participants, is shown in Fig. 1a. Figure 1b shows a bird's-eye view of the entire town. The town consists of a number of distinct buildings and various props, including trees, bushes, benches, and so on. A subset of the buildings are stores, and participants deliver packages between the stores. A delivery is made by walking up to the appropriate store.

Initializing a new session

All of the commonly used PandaEPL features can be made accessible by loading the `pandaepl.common` module:

```
1. from pandaepl.common import *
```

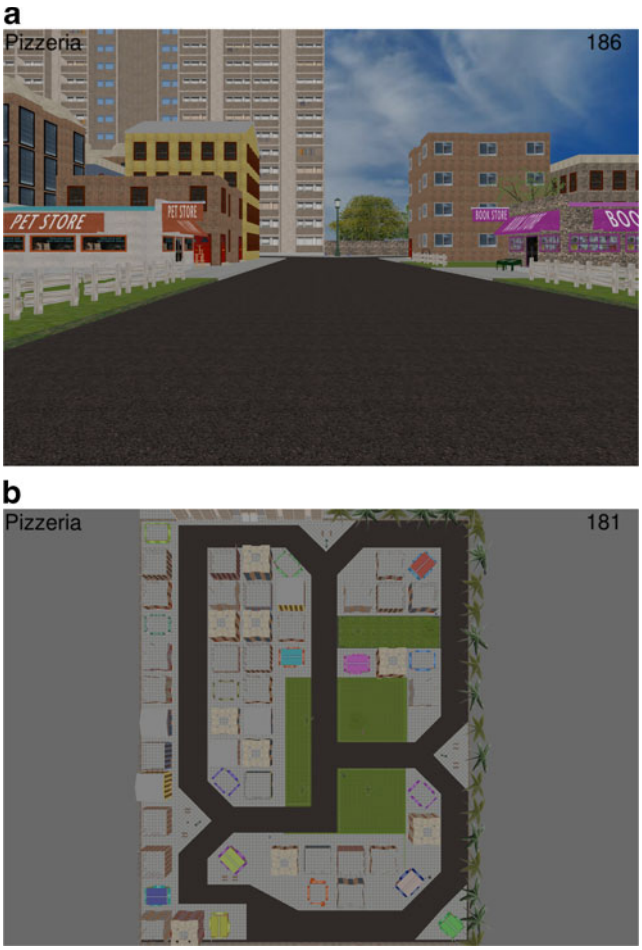

Fig. 1 (**a**) Screenshot of one location in the town, as seen by participants. (**b**) Bird's eye view of the entire town

After loading PandaEPL, the first order of business is usually to set the session number. This is done using the singleton `Experiment` object, which controls the general settings relevant to the session as a whole. The `Experiment` class follows the singleton design pattern and is accessed by the static class method `getInstance()`. Our sample experiment will consist of a single session, which we label as session zero:

```
1. class dboyLite:
2.     def __init__(self):
3.         exp = Experiment.getInstance()
4.         exp.setSessionNum(0)
```

When a session number is set, PandaEPL looks to see whether the session has previously been started by checking for the existence of a session-specific data directory. This directory is expected to reside relative to the current path in `data/[subject_id]/session_[X]`, where `[subject_id]` is specified on the command line (see the section on Command-Line Options in the Appendix and Table 2), and X is the session number given to `setSessionNum`. If this directory exists, PandaEPL will look for previously saved configuration, log, and experiment state files. If it does not exist, the directory is created, and the current global configuration files are copied into it.

We have opted to place the session initialization code into the `__init__` method of a new class that we call `dboyLite`. This class encapsulates the functionality of our new experiment. Until we note otherwise, the next several blocks of code also belong to the `__init__` method of the `dboyLite` class.

Two other singleton classes are of present interest: the `Conf` class, which manages configuration files, and the `Vr` class, which manages aspects of the 3-D environment. We use the `Conf` class to retrieve the configuration dictionary for this session, and we retrieve a reference to the `Vr` class so we can later use it to help set things up:

```
1. config = Conf.getInstance().getConfig()
2. vr = Vr.getInstance()
```

The configuration dictionary is a standard Python dictionary (a set of key–value pairs) that maps configuration variable names to their values. For example, one can reference the configuration variable `linearAcceleration` by writing `config['linearAcceleration']`. PandaEPL reads configuration values from one of two locations. Additional details about the configuration system are provided in the Appendix.

Next, we initialize the *experiment state*. The experiment state is a user-defined dictionary that contains information specific to each participant. It identifies the parts of the experiment that have already been completed, the parts that have yet to be completed, and the general settings that differ between participants (e.g., perhaps in a counterbalanced way). PandaEPL does not use this dictionary directly, but it provides a set of mechanisms to easily save (the `setState` method of the `Experiment` class) and load (the `getState` method of the `Experiment` class) this information for each session. It is up to the experiment designer to make use of the information in a meaningful way. In Listing 1, we initialize the state of the delivery experiment.

The experiment state consists of a combination of static and dynamic values. Line 3 attempts to retrieve

the experiment's current state. If it evaluates to `False`, the experiment state has not yet been set for the current participant and must be initialized anew. Lines 7–11 look in a configuration-specified directory for 3-D models (`.egg` files) of the stores, and lines 15–16 select a random subset of the stores to use for the current participant. Lines 19–28 repeat this process for the other buildings in the town. Lines 31–37 create a random sequence of deliveries, with each delivery linked to a particular store through the store's numerical index in the list built on lines 7–16. The stores, buildings, and deliveries make up the set of static experiment state variables. The experiment also has two dynamic state variables: one for the participant's current delivery assignment, and another for the participant's score. Participants begin the experiment with a predefined score, and the score is then decremented at a regular interval. Points are awarded for each delivery. The five state variables are saved on lines 39–44. Calling `setState` writes the state variables to disk in the appropriate subject- and session-specific directory.

Logging custom events

Every event handled by PandaEPL, such as loading the configuration file(s), loading and placing 3-D models, displaying instructions, moving the participant inside the environment, and so on, is automatically written to a text-based log file. However, almost every new experiment requires for a custom set of events to be logged in addition to those automatically handled by PandaEPL. Two steps are required to achieve this. First, the *event type* must be registered with PandaEPL's log system by calling the `addType` method of the singleton `Log` class. This must occur once for each event type, and is usually done at the beginning of a session. The event type consists of three pieces of information: a name, a list of name–class pairs describing the event's data fields, and a boolean value indicating whether duplicate events should be logged. Our sample experiment makes use of three custom event types: one for arriving at a store, one for receiving a new delivery assignment, and one for updating the score. They are registered with PandaEPL as follows:

```
1. Log.getInstance().addType("ARRIVED", [("STORE", basestring)], False)
2. Log.getInstance().addType("ASSIGNED", [("STORE", basestring)], False)
3. Log.getInstance().addType("SCORE", [("VALUE", int)], True)
```

Each event type has a single field. When the participant arrives at a store, we will save the store's name in the `STORE` field, which we set to be of the built-in Python type `basestring` (line 1). Likewise, when the participant receives a new delivery assignment, we will save the name of the destination in a similar manner (line 2). As these two lines demonstrate, field names may be the same across different event types. In fact, field names are not used by PandaEPL directly and exist solely for the programmer's benefit, to make explicit what is stored in each field.

Both event types have the value `False` associated with the third argument to `addType`. This tells PandaEPL to log duplicate events for these event types (having the value `False` specify this action may seem counterintuitive, but the reader should note that the third argument specifies whether or not to write only new events rather than whether or not to write only duplicate events). This means that, if on two consecutive occasions of each event the `STORE` field has the same value (this may happen, for instance, if the participant bumps into the same store multiple times without first visiting another store), both events will be logged. Of course, each duplicate event will have a unique timestamp. By contrast, line 3 specifies that duplicate `SCORE` events should not be logged. If the score does not change between two or more consecutive `SCORE` events, only the first event will be logged. The second step to logging a custom event is to specify when the event occurs. We will demonstrate how this is done later.

Constructing the heads-up display

The heads-up display consists of objects drawn in "front of" the 3-D environment during navigation. In our example, the heads-up display consists of two text fields. The first, displayed on the right side of the screen, displays the current score. The second, displayed on the left side of the screen, displays the current delivery assignment. The two text fields are created as follows:

```
1. self.score = Text("score", str(config['startingScore']),
2.     config['scorePos'], config['scoreSize'],
3.     config['scoreColor'], config['instructFont'])
4. self.assignment = Text("assignmentHUD", "", config['assignmentPos'],
5.     config['assignmentSize'], config['assignmentColor'],
6.     config['instructFont'])
```

Each field is associated with a corresponding PandaEPL `Text` object, whose constructor accepts a number of arguments describing the text and its appearance. We store and retrieve this information from the configuration file. In addition to parameters controlling the text and its appearance, the constructor requires a unique identifier for each text object. This identifier is used to identify the object in the log.

Although not demonstrated here, PandaEPL has a similar class, `Image`, for managing 2-D images. Furthermore, any 3-D object can also be made part of the heads-up display by passing the value `True` to the object's `setSticksToScreen` method.

Registering tasks to run between frames

After starting the main portion of the experiment, PandaEPL relinquishes control of program flow to Panda3D's main control loop. The loop is responsible for handling certain types of input, rendering each frame, and managing other basic aspects of the environment. In order to run other code, Panda3D allows users to register a set of callbacks to be executed on each loop iteration. PandaEPL registers callbacks to dispatch input events, manage movement, handle collisions, flip the frame buffer, and perform other maintenance tasks. In addition, PandaEPL can manage experiment-specific tasks. Using PandaEPL's task management system rather than interfacing directly with Panda3D ensures that PandaEPL's callbacks are executed in the proper order (both in relation to each other and in relation to experiment-specific code) and simplifies task scheduling.

The PandaEPL `Task` class encapsulates client tasks. Its constructor expects a unique identifier for the task, the callback to be executed, and an optional execution interval given in milliseconds. If the latter is given, the task will execute approximately at the requested interval. Before each frame is rendered, PandaEPL will check whether enough time has elapsed since the task was last run. If the amount of time elapsed equals or exceeds the task's execution interval, the corresponding callback is executed. On modern computers using simple 3-D environments, this check is made very frequently. However, because Panda3D or PandaEPL may be busy doing other things when it comes time to execute the task (or a background process may preempt the entire experiment), there is always some error associated with the timing of tasks. If the experiment is run on a very old computer, or if it is made up of complex 3-D models, the magnitude of the error will be larger because more time is spent rendering each frame. In practice, by being mindful of this issue, we have prevented it from becoming a problem in our experiments. Moreover, because the actual time of each task execution is logged with high precision, it is possible to assess large latencies post-hoc and discard data from problematic intervals.

In order to encourage the participants in our delivery experiment to take the shortest paths possible, we register a task that decrements the score at a regular interval. The task is registered with PandaEPL by passing the `Task` object to the `addTask` method of the singleton `Vr` class:

```
1. vr.addTask(Task("decrementScore",
2.     lambda taskInfo:
3.         self.updateScore(-config['scoreDecrement']),
4.     config['scoreDecrementInterval']))
```

The task consists of an anonymous function that in turn calls the `updateScore` method, passing it the amount by which to decrement the score. This amount is defined in the configuration file. The `updateScore` method (shown in Listing 2): reads the current score from the experiment state, updates the score on the basis of the given value, updates the

heads-up display to show the new score, adds a SCORE event to the video log queue (described below), and saves the new score to the experiment state.

Up until now, we have been incrementally building the __init__ method of the dboyLite class. Our definition of updateScore represents our first departure from this. We will make two more additions to __init__ later below.

Line 9 of Listing 2 demonstrates one way of logging a custom event. This may be done either by directly interfacing with the Log class or by using the specialty video log queue (VLQ) class, as is demonstrated here. The video log queue expects the name of an event type and a list of event-specific values, one for each field registered in the call to addType. Adding an event to the video log queue does not append the event to the log right away. Instead, as is implied by the name of the class, the event is added to a queue. The events in the queue are flushed to the log when the next screen frame is displayed. All events are assigned the same timestamp: the one associated with the display of the new frame. Logging events in this way is sensible in many circumstances. For example, although the code responsible for decrementing the score is executed between frames, the earliest that a participant can begin processing the new score is after the next frame is displayed.

One can assign a specific timestamp to a custom event by using the singleton Log object. Like the writeLine method of the VLQ class, the writeLine method of the Log class expects the name of an event type and a list of event-specific field values. In addition, it takes a 2-tuple (a list of two numbers) indicating the event's estimated start time and duration.

Responding to input events

Panda3D and PandaEPL manage input through the use of event handlers. In addition to logging all input events, PandaEPL provides a single interface to access multiple input devices. Currently, it provides keyboard support through Panda3D and joystick support through PyGame (see the Joystick Support section in the Appendix). The interface makes it easy to add other input devices in the future.

For increased flexibility, input events are handled using a two-step process. First, each valid input is associated with an event name:

```
1.  keyboard = Keyboard.getInstance()
2.  keyboard.bind("exit", ["escape", "q"])
3.  keyboard.bind("toggleDebug", ["escape", "d"])
4.
5.  joystick = Joystick.getInstance()
6.  joystick.bind("toggleDebug", "joy_button0")
```

Because configuration files are standard Python code files, this is usually done in a configuration file. We retrieve the singleton Keyboard object on line 1, and create two keyboard events on lines 2–3 by calling the bind method. The first argument to bind is a unique name for the input event, and the second argument is a list of key names. The event will be triggered when all of the listed keys are pressed simultaneously. The mappings of keys to key names are the same as those used by Panda3D. For details, see the Panda3D manual (www.panda3d.org).

Line 5 retrieves the singleton Joystick object, and line 6 binds the toggleDebug event to the primary joystick trigger. This does not override the association of the toggleDebug event with the keyboard established on line 3. Instead, the event remains bound to both devices and can be triggered by either one. This design pattern allows users to manage key mappings, which may change between platforms and usually appear in the configuration file, separately from event handlers, which are more stable and appear as part of the main experiment code.

After the input events are defined, each event must be associated with an event handler. The following block of code associates the toggleDebug event with an anonymous function that flips the environment in and out of debug mode. In this mode, normal navigation is disengaged and users may move in all three dimensions using the mouse. This feature is useful when building a new experiment.

```
1. vr.inputListen("toggleDebug",
2.       lambda inputEvent:
3.           Vr.getInstance().setDebug(not Vr.getInstance().isDebug()))
```

We place the call to `inputListen` inside the `__init__` method of our `dboyLite` class.

The `exit` event is handled internally by PandaEPL and ends the experiment. PandaEPL also automatically handles the events `moveForward`, `moveBackward`, `turnLeft`, and `turnRight`. By default, they are bound to the keyboard arrow keys and the primary joystick axis (if a joystick is attached).

Loading and managing 3-D models

The 3-D models used in the experiment are loaded by the `loadEnvironment` method, shown in Listing 3. We call `loadEnvironment` from `__init__`:

```
1. self.loadEnvironment()
```

Models are divided into four types: the overall terrain, including scenery and props; the sky; the buildings; and the stores. The buildings and stores appear at a set of prearranged coordinates, with the identity of the building or store at each location chosen pseudorandomly. Lines 9–10 load the terrain model by creating a corresponding PandaEPL `Model` object. The constructor for the `Model` class expects at minimum a unique identifier for the model, the path to the `.egg` or `.bam` file containing the model specification (see the File Formats section in the Appendix), and the model's position in a three-dimensional grid.

Line 13 sets the terrain's collision callback to the static `handleSlideCollision` method of the `MovingObject` class. The callback is executed whenever a suitable object collides with the terrain. In this example (and in most cases), the only object that can trigger collision events is the participant, who is represented in the environment by an invisible sphere that travels along with the camera. More complicated scenarios are supported as well; see the Collision Handling section in the Appendix. The `handleSlideCollision` method prevents a colliding object from intersecting with a target by sliding the source object along the target's surface. Similarly, `handleRepelCollision`, also a static method of the `MovingObject` class, prevents collisions without inducing additional movement. When a source object reaches a target, it is prevented from moving any farther.

The model of the sky is loaded on lines 16–17. Because it appears beyond participants' reach, no collision callback is set. Lines 20–28 load the building models, again making use of `handleSlideCollision`. Finally, lines 31–40 load the store models. Here we use a custom collision callback, the `collideStore` method. The `collideStore` method makes use of three helper methods: `nextDelivery`, `showDeliveryInfo`, and `storeName`. These methods are defined in Listing 4. The `collideStore` method compares the name of the store the participant collided with to the name of the current target. If they match, it updates the participant's score with a call to `updateScore` (line 19), informs the participant of a successful delivery (lines 22–24), and calls the `nextDelivery` method (defined on lines 29–44) to assign the next delivery. Whether or not the store was the intended target, an `ARRIVED` event is logged (line 12), and `handleRepelCollision` is called to keep the participant from moving inside the store (line 27).

Displaying instructions

The PandaEPL `Instructions` class is a wrapper around the `Text` class used to display instructions. Instructions displayed using this class span the full extent of the screen. If the text extends beyond the range of the screen, the `Instructions` class automatically allows participants to scroll through it by overriding the `moveForward` and `moveBackward` input events. A configuration setting alows the user to force participants to look through the entire set of instructions before proceeding. This and other instruction properties, including the foreground and background colors, the font and text size, and the set of command key(s) used to dismiss the instruction screen, are set globally in the configuration file. Table 1 lists all of the properties that can be set.

Lines 22–24 of Listing 4 demonstrate how the `Instructions` class can be used. In addition to a unique identifier and the text to display, the constructor takes an optional argument specifying the function to execute when the instructions are dismissed. We pass in a reference to the `nextDelivery` method, which updates the experiment state to reflect the next delivery.

Table 1 Configuration settings for instruction screens

Variable Name	Description	Example Value
`instructSize`	Size of the text in Panda3D units	0.075
`instructFont`	Path to the font file to use	/home/user/FreeSans.ttf
`instructBgColor`	Background color, a Point4 object specifying RGB and alpha (transparency) values	Point4(0, 0, 0, 1)
`instructFgColor`	Foreground color, also a Point4 object	Point4(1, 1, 1, 1)
`instructMargin`	The amount of space to leave at each edge of the screen, in Panda3D screen units	0.06
`instructSeeAll`	Boolean (true/false) toggle that determines whether all of the instructions must be seen before the participant can proceed. This only affects instructions longer than the height of the screen.	False

The `display` method is called on the newly constructed `Instructions` object to display the associated text (informing the participant of a successful delivery) on the screen.

Starting the experiment

The `start` method, defined below, starts the experiment.

```
1.  def start(self):
2.      self.intro()
3.      Experiment.getInstance().start()
4.
5.  def intro(self):
6.      # Read intro text in from external text file.
7.      fid = open(Conf.getInstance().getConfig()['instructionFile'], 'r')
8.      introText = fid.read()
9.      fid.close()
10.
11.     # Display intro text followed by information about the first delivery.
12.     Instructions("intro", introText, self.showDeliveryInfo).display()
```

It first calls the `intro` method to display the introductory instructions and information about the first delivery, and then relinquishes control of program flow to PandaEPL by calling the `start` method of the singleton `Experiment` object. After performing some initialization of its own, PandaEPL relinquishes control to Panda3D. From that point on, experiment-specific code is executed in one of three ways: in response to collision events (e.g., with the call to `collideStore` when a participant reaches a store), in response to input events (e.g., with the call to the anonymous

function handling the `toggleDebug` event), and when called by the task manager.

The experiment can be started by creating a new `dboyLite` object and calling its `start` method:

```
1. dboyLite().start()
```

Conclusion

We have presented PandaEPL, a Python library for programming spatial-navigation experiments. PandaEPL uses the Panda3D engine to manage tasks related to loading, displaying, and navigating a 3-D environment. Built on top of this are features specific to the development of psychology experiments, including managing configuration files, logging experiment- and participant-generated events with precise timing information, keeping track of the experimental state, and synchronizing multiple computers. PandaEPL provides a unified framework for developing new experiments, saving researchers from having to customize standalone 3-D game engines for this purpose. Unlike Maze Suite (Ayaz et al., 2008), which provides a point-and-click interface for developing simple experiments, PandaEPL is a general-purpose programming library that can be adopted to build complex new paradigms. This parallels the approach taken by PyEPL and its VR module (Geller et al., 2007). However, by using Panda3D for managing aspects of the 3-D environment, PandaEPL automatically gains access to all of the features offered by a modern game engine. This includes true 3-D model support, efficient rendering and collision detection, fog and lighting effects, the use of shaders, and more. In contrast, PyEPL's VR module uses a nonoptimized custom-built rendering engine that can display only textured cubes and 2-D sprites. Future work on PandaEPL will need to make the library more accessible to novice programmers who wish to create simple experiments.

Author note The authors gratefully acknowledge support from National Institutes of Health Grant No. MH61975.

Appendix: Implementation details

In this section, we provide more extensive technical details on several key topics introduced in the example. A few additional features, not discussed in the main text, are also introduced.

Dependencies

The PandaEPL library and all of the files needed to run the example in this paper can be found at http://memory.psych.upenn.edu/PandaEPL. At minimum, PandaEPL requires Python and Panda3D in order to run. The Panda3D website (www.panda3d.org) is an invaluable source of information. The website includes the Panda3D manual, reference documentation covering all available classes and functions, and an active public forum.

Additional libraries are necessary in order to play and record sound (tkSnack and its dependencies), to use a joystick (PyGame and its dependencies), and to synchronize multiple computers (a suitable device driver and Python interface for the requisite hardware). These libraries are optional and are only necessary if the corresponding features are used.

Sound playback and recording

Panda3D supports sound playback through the use of various third-party audio libraries. However, at present, these libraries do not provide cross-platform support for recording sound. Since this is a necessary feature for some experiments, PandaEPL uses the tkSnack library for both playback and recording. More information on tkSnack can be found on its website, www.speech.kth.se/snack/. The interface between PandaEPL and tkSnack is managed by the `SimpleSound` class. The complete source code for the delivery experiment, downloadable from our lab's website, contains a demonstration of how this class may be used. Installing tkSnack is only necessary if the `SimpleSound` class is used. Otherwise, PandaEPL will not look for it.

Joystick support

Panda3D currently does not provide native joystick support, but PandaEPL can process joystick input by interfacing with the PyGame library. Unfortunately, PyGame itself has a number of dependencies and requires several other libraries to be installed. For more information, visit the PyGame website (www.pygame.org). Installing PyGame is only necessary if joystick support is required. PandaEPL will silently check for PyGame, and enable joystick support only if it is available.

The PyGame joystick interface is handled by the `Joystick` class. In principle, it should be easy to modify PandaEPL to use a more self-contained library should one become available in the future.

Synchronizing multiple computers

In experiments where neural data is recorded, it is common practice to display the experiment on one

computer and to record the neural data on another computer. In such circumstances, an additional mechanism is required to align experimental events with the neural data. In our lab, we synchronize the two computers using an ActiveWire USB card. The computer running the experiment uses the ActiveWire card to send a synchronization pulse. The timing of each pulse is saved on both computers, and the two sets of timestamps are used to align the data. The pulses are managed by the `EEG` class in PandaEPL. Although named `EEG` for historical reasons, the same mechanism may be used to synchronize other recording equipment. Synchronization is set to be "on" by default, but may be disabled by supplying the `--no-eeg` command-line option. The user can override the `EEG` class that comes with PandaEPL in order to interface with other synchronization hardware.

Model file formats

We used Autodesk Maya to create the 3-D models and determine the overall layout of the town used in the example. However, users are free to choose from a wide range of other packages for their projects, including the free and open-source Blender software (www.blender.org). The range of supported graphics software is set by the availability of suitable conversion utilities, to convert from each package's native file format to Panda3D's `.egg` format. The Panda3D website provides up-to-date details on supported graphics software and conversion procedures.

The `.egg` format is Panda3D's custom structured text format for describing 3-D models. Texture graphics are stored externally and are referenced by path and filename. The format is cross-platform compatible. However, loading a `.egg` file requires that the markup is parsed each time, resulting in significant overhead. For efficiency, Panda3D also supports a binary file format (`.bam`) whose implementation details may differ between different platforms and different versions of Panda3D. In general, users work with `.egg` files and convert to `.bam` when the experiment is ready to be run. Panda3D provides two methods for converting `.egg` files to `.bam` files: using the command-line `egg2bam` utility, and via a series of Python-based calls. In order to simplify the conversion process for the 3-D models used in the example, we have provided a helper utility to perform all of the necessary conversions in one step. See the README file provided with the package on our website for details.

Collisions

In most experiments, the participant is the only actively moving object in the environment. PandaEPL represents the participant by an invisible sphere whose radius is set using the `avatarRadius` configuration variable. Collisions between this sphere and other objects in the environment trigger collision callbacks set on target objects. Collision callbacks are required to accept a single argument corresponding to a `CollisionInfo` object describing the collision.

PandaEPL also supports more complicated scenarios in which other moving objects may trigger collision events. Such objects must be of the `MovingObject` class or one of its derivatives, and must have their `fromCollision` flag turned on. The singleton `Avatar` object, which represents the participant, is simply a special type of `MovingObject`.

Configuration files

The configuration files used by PandaEPL are regular Python files with Python source code. Usually, they consist of a series of `key = value` assignments, but arbitrarily complex preprocessing may also be included. Whatever variable assignments remain after the configuration file is executed with the Python interpreter are transferred to PandaEPL's configuration dictionary.

PandaEPL requires at least one configuration file. However, it can also process an optional secondary "subject" configuration file. New variables introduced in the subject configuration file are appended to the same configuration dictionary. Duplicate variables that also exist in the primary configuration file are replaced with the values in the subject configuration file. This allows users to create one configuration file for values that are the same across participants, and a set of secondary configuration files for values that vary between participants (or groups of participants).

Configuration files are loaded when the `setSessionNum` method of the singleton `Experiment` object is called. Once PandaEPL has the session number, it will look for both configuration files in the corresponding subject- and session-specific directory. If a primary configuration file is found but a secondary configuration file is not, PandaEPL will load the primary configuration file and will not look for the secondary configuration file in any other place. If a primary configuration file is not found, PandaEPL will look elsewhere for both files. By default, PandaEPL expects both files to be located in the

current working directory (i.e., the directory from which the experiment is launched) and to be named `config.py` and `sconfig.py` for the primary and subject configuration file, respectively. It is possible to override these defaults with the `--config=` and `--sconfig=` command line options. The primary configuration file and, if it exists, the secondary configuration file, are copied into the corresponding subject- and session-specific directory. Future runs of the same subject and session number will load the configuration files from that location. This allows users to continue updating configuration files as the experiment evolves, while retaining the ability to easily run old sessions and reference old configuration values.

Command-line options

In addition to options for overriding the names of the configuration files, PandaEPL supports a number of other command-line options and arguments. These are displayed in Table 2. The only required argument is `-s`, specifying a unique identifier for the current subject. For example, the following command starts the delivery experiment for Subject 0 with synchronization disabled:

```
python dboyLite.py --no-eeg -s0
```

Allowed values for the subject identifier are limited by the restrictions set by the local file system for naming files. All of the data generated by PandaEPL are stored in the directory `data/[subject_id]`. In general, it is a good idea to restrict subject identifiers to consist of alphanumeric characters and underscores.

Logging events

A separate log file is written for each subject and session to `data/[subject_id]/session_[num]/log.txt`. If the same session is restarted, new events are appended to the end of the existing file. Each line corresponds to a single event and is tab-delimited with a variable number of fields. The first three columns are always the same and correspond to, respectively, the estimated start time, the estimated duration, and the name of the event type. As is demonstrated in the example, each event has a corresponding event type

Table 2 Command-line options recognized by PandaEPL

`-s`	Subject identifier (required)
`--config`	Path to the primary configuration file (default: `config.py`)
`--sconfig`	Path to the secondary configuration file (default: `sconfig.py`)
`--no-fs`	Runs the experiment in a window (default: when option is not present, the experiment is run in full-screen mode)
`--resolution`	Screen resolution (default: 800×600)
`--show-fps`	Displays the number of frames rendered per second (default: when option is not present, the number of frames rendered per second is not reported)
`--js-zero-threshold`	Value beyond which a joystick axis is considered "active." Axis values are in the range [0, 1] (default: 0.2)
`--no-eeg`	Do not send synchronization pulses (default: when option is not present, synchronization pulses are sent).

registered with a call to the `addType` method of the singleton `Log` object. The event type consists of a name, a list of name–class pairs describing the data fields, and a flag indicating whether duplicate events should be logged. The third column of the log contains the name of the `event Type`. Each additional column corresponds to a single data field, with the number of additional columns matching the number of fields registered for the specified event type.

Events are written by calling the `writeLine` method of either the singleton `Log` or the singleton video log queue (`VLQ`) object. Before writing the event to the log, the `Log` class checks to make sure that the number of given values matches the number of fields registered for that event type, and that the class of each value matches the class registered for the corresponding field. Attempting to log an event that does not meet these criteria results in a `LogException` exception. When using the video log queue, type checking is performed only when the queue is flushed. The type checking performed by the `Log` class makes logging less error-prone and provides helpful guarantees for log parsers.

It is up to the user to parse and analyze the tab-delimited log files as they see fit. PandaEPL does not provide built-in utilities for data analysis. Tab-delimited files are widely supported and can easily be loaded into packages ranging from complete programming environments like MATLAB (The MathWorks, Inc., Natick, MA) and R (R Development Core Team, 2012), to spreadsheet programs like Excel (Microsoft Corp., Redmond, WA).

Recording the timing of events

Timing information in PandaEPL is based on two values: the time just before an event and the estimated duration of the event. Because the start time is queried before an event is executed, the estimate is guaranteed to be less than or equal to the actual start time. Likewise, the end time is queried after an event is executed, and the duration is guaranteed to be greater than or equal to the actual duration. On modern computers processing simple events, the recorded duration is small (only 1–2 ms). Events are timed using the `timedCall` function. The first argument specifies the function to time, and the remaining arguments are passed to this function unchanged. `timedCall` returns a 2-tuple containing information about the timing of the call and the function's return value, respectively. The timing information is itself a 2-tuple containing the approximate start time and duration of the call.

The same 2-tuple format for specifying the beginning and duration of an event is used by the `writeLine` method of the `Log` class. The `writeLine` method saves this information in the first two columns of the log. PandaEPL uses `timedCall` internally to track the timing of several different events, and users may use it to track custom events of their own. For events for which precise timing is not required, such as instantiating a PandaEPL class or modifying the value of a setting, PandaEPL approximates the time of the event with a single value. Such events are recorded in the log as having zero duration.

Obtaining a precise range of execution times in the manner described above requires that the time obtained at each end of the interval is precise. The `timedCall` function uses the Python `datetime` module to obtain the current system time, and it in turn uses the local implementation of the POSIX standard `gettimeofday` function. Although `gettimeofday` returns time in microseconds on all platforms, its actual precision varies.

Precision also varies between platforms when it comes to the display. The video log queue (`VLQ`), used extensively by PandaEPL itself and recommended for logging most custom event types, assigns to its events the timestamp corresponding to flipping the display buffer. This makes the next frame "visible" on the screen. However, additional latencies associated with displaying the next screen may occur after the requisite function call. Some platforms will, depending on the value of a global setting, wait until the next vertical screen refresh, while others will not. On all platforms, additional latency is associated with actually producing each part of the screen.

Because PandaEPL is a cross-platform library, general guarantees about timing cannot be made. If high-precision timing is required, it is important to carefully assess and test the local environment.

Other features

The underlying Panda3D game engine has a number of other features for creating rich and complex environments. Most users will want to keep their experiments as simple as possible, but the reader should be aware that much more is possible. The best way to explore the available features is to look through the Panda3D website. PandaEPL interfaces with two additional Panda3D features not presented in the example above: lighting and fog effects. PandaEPL's only contribution to these features is to track their use in the experiment log. The complete source code for the delivery experiment, downloadable from our lab's website, provides a simple example of how these effects may be used.

Listing 1

```
# If this is the first time running this subject,
# initialize the experiment state.
if not exp.getState():
    # Load list of available stores.
    # Each store has an associated '.egg' file in the
    # stores directory.
    storeDirList = os.listdir(config['storeDir'])
    stores = []
    for store in storeDirList:
        if store[-4:] == '.egg':
            stores.append(store[:-4])

    # Shuffle store list and reduce to the number of
    # stores in the environment.
    shuffle(stores)
    stores = stores[:config['numStores']]

    # Load list of buildings.
    buildingDirList = os.listdir(config['buildingDir'])
    buildings = []
    for building in buildingDirList:
        if building[-4:] == '.egg':
            buildings.append(building[:-4])

    # Shuffle building list and reduce to the number  of
    # buildings in the environment.
    shuffle(buildings)
    buildings = buildings[:config['numBuildings']]

    # Construct random sequence of deliveries.
    deliveries = []
    while len(deliveries) < config['numDeliveries']:
        nextSequence = range(config['numStores'])
        shuffle(nextSequence)
        if len(deliveries) == 0 or deliveries[-1] != nextSequence[0]:
            deliveries.extend(nextSequence)
    deliveries = deliveries[:config['numDeliveries']]

    # Save experiment parameters.
    exp.setState('stores': stores,
        'buildings': buildings,
        'deliveries': deliveries,
        'currentDelivery': 0,
        'score': config['startingScore'])
```

Listing 2

```
1.  def updateScore(self, deltaScore):
2.      # Get experiment parameters.
3.      state = Experiment.getInstance().getState()
4.
5.      # Update score.
6.      state[‘score’] = max(0, state[‘score’] + deltaScore)
7.
8.      # Log new score.
9.      VLQ.getInstance().writeLine(“SCORE”, [state['score']])
10.
11.     # Update heads-up display.
12.     self.score.setText(str(state[‘score’]))
13.
14.     # Save new score.
15.     Experiment.getInstance().setState(state)
```

Listing 3

```
def loadEnvironment(self):
    # Get configuration dictionary.
    config = Conf.getInstance().getConfig()

    # Get experiment parameters.
    state = Experiment.getInstance().getState()

    # Load terrain.
    self.terrainModel = Model("terrain", config['terrainModel'],
       config['terrainCenter'])

    # When hitting an object that is part of the terrain, slide across it.
    self.terrainModel.setCollisionCallback(MovingObject.handleSlideCollision)

    # Load sky.
    self.skyModel = Model("sky", config['skyModel'])
    self.skyModel.setScale(config['skyScale'])

    # Load buildings.
    self.buildingModels =[]
    for i, building in enumerate(state['buildings']):
       buildingModel = Model(building,
          os.path.join(config['buildingDir'], building + ".bam"),
          Point3(config['buildingLocs'][i][0],
              config['buildingLocs'][i][1],
              config['buildingZ']),
          MovingObject.handleSlideCollision)
       self.buildingModels.append(buildingModel)

    # Load stores.
    self.storeModels =[]
    for i, store in enumerate(state['stores']):
       storeModel = Model(store,
          os.path.join(config['storeDir'], store + ".bam")
          Point3(config['storeLocs'][i][0],
              config['storeLocs'][i][1],
              config['storeZ']),
          self.collideStore)
       storeModel.setH(config['storeLocs'][i][2])
       self.storeModels.append(storeModel)
```

Listing 4

```
1.  def collideStore(self, collisionInfoList):
2.      # Get configuration dictionary.
3.      config = Conf.getInstance().getConfig()
4.
5.      # Get experiment parameters.
6.      state = Experiment.getInstance().getState()
7.
8.      # ID of the store the participant collided with.
9.      store = collisionInfoList[0].getInto().getIdentifier()
10.
11.     # Log collision.
12.     VLQ.getInstance().writeLine("ARRIVED", [store])
13.
14.     # If a delivery is currently assigned, is this the store where it is going ?
15.     if state['currentDelivery'] >= 0 and \
16.        state['currentDelivery'] < len(state['deliveries']) and \
17.        store == state['stores'][state['deliveries'][state['currentDelivery']]]:
18.        # Update score.
19.        self.updateScore(config['deliveryBonus'])
20.
21.        # Inform participant they have made the delivery and assign the next
    one.
22.        Instructions("deliveryMade",
23.           config['deliveryMadeText'] % self.storeName(store),
24.           self.nextDelivery).display()
25.
26.     # Don't let the participant move inside the store.
27.     MovingObject.handleRepelCollision(collisionInfoList)
28.
29.  def nextDelivery(self):
30.     # Get configuration dictionary.
31.     config = Conf.getInstance().getConfig()
32.
33.     # Get experiment parameters.
34.     state = Experiment.getInstance().getState()
35.
36.     # If all deliveries have not been made prior to this one.
37.     if state['currentDelivery'] < len(state['deliveries']):
38.        # Move on to next delivery.
39.        state['currentDelivery'] += 1
40.
```

```
        # Save next delivery.
        Experiment.getInstance().setState(state)

    self.showDeliveryInfo()

def showDeliveryInfo(self):
    # Get configuration dictionary.
    config = Conf.getInstance().getConfig()

    # Get experiment parameters.
    state = Experiment.getInstance().getState()

    # If all deliveries have been made, end the experiment.
    if state['currentDelivery'] == len(state['deliveries']):
      Instructions("experimentComplete", config['experimentCompleteText'],
          Experiment.getInstance().stop).display()
    # Otherwise, display information about the current delivery.
    else:
       store = state['stores'][state['deliveries'][state['currentDelivery']]]
       VLQ.getInstance().writeLine("ASSIGNED", [store])
       self.assignment.setText(self.storeName(store))
       Instructions("assignmentInstruct",
          config['assignmentText'] % self.storeName(store)).display()

def storeName(self, store):
    return store.replace("_", " ")
```

References

Alvarez, R. P., Biggs, A., Chen, G., Pine, D. S., & Grillon, C. (2008). Contextual fear conditioning in humans: Cortical-hippocampal and amygdala contributions. *Journal of Neuroscience, 28,* 6211–6219.

Astur, R., Taylor, L., Mamelak, A., Philpott, L., & Sutherland, R. (2002). Humans with hippocampus damage display severe spatial memory impairments in a virtual Morris water task. *Behavioural Brain Research, 132,* 77–84.

Ayaz, H., Allen, S. L., Platek, S. M., & Onaral, B. (2008). Maze Suite 1.0: A complete set of tools to prepare, present, and analyze navigational and spatial cognitive neuroscience experiments. *Behavior Research Methods, 40,* 353–359. doi:10.3758/BRM.40.2.353

Cornwell, B., Johnson, L., Holroyd, T., Carver, F., & Grillon, C. (2008). Human hippocampal and parahippocampal theta during goal-directed spatial navigation predicts performance on a virtual Morris water maze. *Journal of Neuroscience, 28,* 5983–5990.

Doeller, C. F., Barry, C., & Burgess, N. (2010). Evidence for grid cells in a human memory network. *Nature, 463,* 657–661.

Doeller, C. F., King, J. A., & Burgess, N. (2008). Parallel striatal and hippocampal systems for landmarks and boundaries in spatial memory. *Proceedings of the National Academy of Sciences, 105,* 5915–5920.

Ekstrom, A. D., Copara, M. S., Isham, E. A., Wang, W. C., & Yonelinas, A. P. (2011). Dissociable networks involved in spatial and temporal order source retrieval. *NeuroImage, 56,* 1803–1813. doi:10.1016/j.neuroimage.2011.02.033

Ekstrom, A. D., Kahana, M. J., Caplan, J. B., Fields, T. A., Isham, E. A., Newman, E. L. (2003). Cellular networks underlying human spatial navigation. *Nature, 425,* 184–188. doi:10.1038/nature01964

Geller, A. S., Schleifer, I. K., Sederberg, P. B., Jacobs, J., & Kahana, M. J. (2007). PyEPL: A cross-platform experiment-programming library. *Behavior Research Methods, 39,* 950–958.

Gron, G., Wunderlich, A. P., Spitzer, M., Tomczak, R., & Riepe, M. W. (2000). Brain activation during human navigation: Gender-different neural networks as substrate of performance. *Nature Neuroscience, 3,* 404–408.

Hassabis, D., Chu, C., Rees, G., Weiskopf, N., Molyneux, P., & Maguire, E. (2009). Decoding neuronal ensembles in the human hippocampus. *Current Biology, 19,* 546–554.

Iaria, G., Chen, J., Guariglia, C., Ptito, A., & Petrides, M. (2007). Retrosplenial and hippocampal brain regions in human navigation: Complementary functional contributions to the formation and use of cognitive maps. *European Journal of Neuroscience, 25,* 890–899.

Jacobs, J., Kahana, M. J., Ekstrom, A. D., Mollison, M. V., & Fried, I. (2010a). A sense of direction in human entorhinal cortex. *Proceedings of the National Academy of Sciences, 107,* 6487–6482.

Jacobs, J., Korolev, I., Caplan, J., Ekstrom, A., Litt, B., Baltuch, G., ... Kahana, M. (2010b). Right-lateralized brain oscillations in human spatial navigation. *Journal of Cognitive Neuroscience, 22,* 824–836.

Maguire, E., Burgess, N., Donnett, J. G., Frackowiak, S. J., Frith, C. D., & O'Keefe, J. (1998). Knowing where and getting there: A human navigation network. *Science, 280,* 921–924.

Miller, J. F., Lazarus, E. M., Polyn, S. M., & Kahana, M. J. (in press). Spatial clustering during memory search. *Journal of Experimental Psychology*: *Learning*, *Memory*, *and Cognition*. doi:10.1037/a0029684.

O'Keefe, J., & Dostrovsky, J. (1971). The hippocampus as a spatial map: Preliminary evidence from unit activity in the freely-moving rat. *Brain Research, 34,* 171–175.

O'Keefe, J., & Nadel, L. (1978). *The hippocampus as a cognitive map*. New York: Oxford University Press.

R Development Core Team. (2012). *R: A language and environment for statistical computing*. Vienna: R Foundation for Statistical Computing. Retrieved from www.R-project.org

Shipman, S. L., & Astur, R. S. (2008). Factors affecting the hippocampal BOLD response during spatial memory. *Behavioural Brain Research, 187,* 433–441.

Suthana, N., Haneef, Z., Stern, J., Mukamel, R., Behnke, E., Knowlton, B., & Fried, I. (2012). Memory enhancement and deep-brain stimulation of the entorhinal area. *The New England Journal of Medicine, 366,* 502–510.

van der Ham, I. J. M., van Zandvoort, M. J. E., Meilinger, T., Bosch, S. E., Kant, N., & Postma, A. (2010). Spatial and temporal aspects of navigation in two neurological patients. *NeuroReport, 21,* 685–689.

Watrous, A. J., Fried, I., & Ekstrom, A. D. (2011). Behavioral correlates of human hippocampal delta and theta oscillations during navigation. *Journal of Neurophysiology, 105,* 1747–1755.

Weidemann, C. T., Mollison, M. V., & Kahana, M. J. (2009). Electrophysiological correlates of high-level perception during spatial navigation. *Psychonomic Bulletin & Review, 16,* 313–319. doi:10.3759/PBR.16.3.313