USING VPYTHON FOR PSYCHOPHYSICS

Pearl S. Guterman, Robert S. Allison, Stephen Palmisano[2]
York University, Toronto, ON, Canada
[2]*University of Wollongong, Wollongong, Australia*
pearljam@cse.yorku,ca, allison@cse.yorku.ca, stephenp@uow.edu.au

Abstract

Python is a high-level programming language that is a powerful tool for conducting psychophysical experiments. Like Matlab, it is widely used and supports visual and auditory stimuli and manual user input (e.g., keyboard/mouse). Unlike Matlab, Python is free and open source. Ease of use and intuitive syntax also make Python an attractive option. This paper looks at the use of a Python module called "Visual Python" (VPython) for creating dynamic visual stimuli and response screens. A technique for producing optic flow displays to induce visual illusions of self-motion (known as vection) will be presented using two sample programs that produce passive or "active" (i.e., motion determined by user input) self-motion. Find out why so many psychophysicists are migrating to Python.

Computer-generated displays and computer-controlled data collection have become the standard for psychophysical study. Ideally, the programming tool should be easy for experimenters to learn and use with effective results. There are several program options available both commercially and open-source, but Python is quietly becoming the preferred choice of many psychophysicists for coding experiments. It is a freely available programming language with libraries that support dynamic visual and audio stimuli, rich user interaction, and is platform independent (Windows, Mac, and Linux). Python also has an Integrated Development Environment (IDLE); "charmingly" named after the British comedy group "Monty Python's Flying Circus" and a founding cast member, respectively (Lutz, 1996). Of the many Python libraries, VPython (http://www.vpython.org) supports a wide range of 3D and 2D screen imagery, and is easy to implement without the need to draw on additional dependencies. This paper explores the capabilities of VPython in the context of sample programs that can be used for psychophysical study.

Usage

In this section we illustrate the features of VPython for producing dynamic stimuli and user response screens for psychophysical experiments using two sample programs. The first sample program displays a random pattern of radially moving dots to induce vection, and a slider bar for recording users' feeling of self-motion. The motion produced is passive in that it is not interactive. The second sample program displays a virtual environment and allows the user to actively travel within it using a computer keyboard. We also explore a technique for producing jittering and oscillating motion, and how to link these motions to scene events; for instance, causing the scene to jitter when an observer travels across bumpy surfaces. These events become possible with Python after importing a set of modules.

Table 1:

VPython	Vision Egg	PsychoPy	PyEPL
Stopwatch PIL	NumpyPyOpenGl Pygame PIL Setuptools	Numpy PyOpenGL Pygame PIL Scipy Matplotlib Piglet WxPython Setuptools Pywin32 (Windows)	Numpy PyOpenGL Pygame PIL Libsndfile Libsamplerate SWIG Ode ALSA Pyrex PyODE

Dependencies

Using VPython for psychophysical experiments requires few dependencies. At most, the sample programs described here required downloading two additional modules: Stopwatch for timing stimuli and responses, and the Python Imaging Library (PIL) for adding textures. That is, the only required installs are Python, VPython, Stopwatch, and optionally, PIL. Dependencies are imported in the first lines of the code and may be order-dependent. Alternative modules for coding psychophysical experiments are available, such as Vision Egg, PsychoPy, and PyEPL, but have a greater demand for additional modules and this can lead to dependency issues that prevent the primary module from running. Table 1 shows a comparison of dependencies for running a similar program to the sample programs. Of those listed, Vision Egg is best known for efficiently producing more traditional visual stimuli for experimental psychology (Straw, 2008), but assumes that the user has prior object-oriented programming.

Example program 1: Passive vection (star field)

The first example program demonstrates the use of VPython for screen output and keyboard input for psychophysical experiments. Looming dots shown on screen give the effect of travelling through a star field in space. Specifically, the program does the following:

1. It shows a looming pattern, consisting of circular objects, for a given duration
2. It displays a message screen and waits for user input
3. It presents instructions for rating the proceeding stimuli
4. It shows looming dots for a given duration while continuously recording user input (i.e., vection onset and duration total for trial)
5. It displays a slider bar for rating vection strength and waits for user input
6. It shows a blank inter-stimulus screen

Listing 1 shows the looming dots portion of the code in reduced form and it is explained in 5 steps; keyboard events are shown later in the paper. For convenience, section numbers are included for each function. Lines that begin with "#" indicate that it is a comment to be ignored by the Python interpreter.

Make stimuli. Before calling on VPython to draw the scene, we create an empty list called "`circles`" to contain the position of each object in the scene, and specify the total number of objects. Then, we draw a function called "points" to create an instance of a graphic primitive and specify its parameters such as position, size, and colour; the default

```
1. #make circular objects
   circles=[]
   for i in range(n):
          circles+=[points(pos=(0,0,0), size=circleSize, color=color.white,
          size_units="world",visible=False)]

2. for trial in range(numTrials):
       scene.center=(0,0,0)

       #index points
       for i in range(n):
              circles[i].x=uniform(-x,x)
              circles[i].y=uniform(-y,y)
              circles[i].z=uniform(-maxZ,0)

              while ((circles[i].x**2 + circles[i].y**2) <
                     radiusSquared):
                     circles[i].x=uniform(-x,x)
                     circles[i].y=uniform(-y,y)
                     circles[i].visible=True
       circles.sort(key = lambda depth: depth.z)

    3. #start timer
       timerTrial=stopwatch.Timer()
       timeout = timerTrial.elapsed

    4. frame=0
       #move points (main loop)
       while timeout < trialDur:
              rate(60)
              travelled = -frame*speed[j];
              scene.center = (0,0,travelled)
              fixpt.z = scene.center.z – fixptDist

          5. while circles[-1].z > travelled:
                     circles[-1].z = circles[-1].z - maxZ
                     circles.insert(0,circles.pop())

              frame=frame+1
              timeout = timerTrial.elapsed
```

Listing 1: Section of the passive vection program.

shape is a circle, but this can be changed to a square by adding the option, `shape="square."` Note that the objects are not visible at this time (visible=False). For each trial we need to place the stimuli, draw them and collect data. These operations are all placed within a loop that iterates over the number of trials, `numTrials`.

Place stimuli. Section 2 contains code to place the viewpoint in the scene and initialize a volume of dots. To start each trial, the position of the scene camera is reset with the line `scene.center=(0,0,0)`. The dot positions are initialized in a loop that randomizes the x (horizontal), y (vertical), and z (depth) position of the objects within a constrained volume of space. Randomly assigning each object a position within the volume along the x, y and z-axes, and setting these positions in the "circles" list accomplishes this. It also sorts the list of points by depth order for later testing whether we have passed any objects in the scene while moving. Setting visible to TRUE results in the dots being actually rendered on the display.

Time stimuli. The Stopwatch module is used to time events such as the stimulus presentation, interstimulus interval (ISI), and response time. This module, along with the constant `trialDur`, sets the stimuli duration for each trial and is used in another portion of the code to record response times.

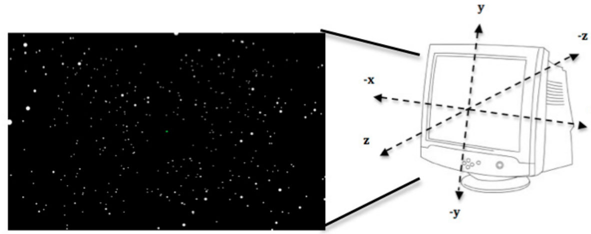

Figure 1: *Screenshot of the looming dots stimuli and VPython screen axes.*

Move stimuli (moving the camera). The main loop produces the apparent motion of the scene, and runs for `trialDur seconds`. The function "`rate()`" specifies the frame rate with which the program updates the screen and is essentially a timer that holds the rendering loop until the next update time. For each frame, the distance that "`travelled`" from the start position is determined by multiplying the frame by speed; for example, if the speed is 2, then the travelled distance would be 2 screen units (i.e., 1 X 2) per frame. As the travelled distance increases on subsequent frames, the virtual camera is moved along the z-axis to produce the sense of forward motion. Also, it was important that the position of the fixation point be maintained relative to the camera, and so the next line updates its position to keep it at the same distance with respect to the camera as the camera translates. To simulate viewpoint jitter, the x or y-axis value can be set to a random value on each frame or smoothly modulated according to a sinusoid or other function.

Update stimuli position. Drawing an infinite number of objects on the screen is not possible. Even if it were possible it would be computationally expensive and would cause a program to lag or even crash. In order to generate a seemingly never-ending field of objects, we can simply redraw objects that pass behind the camera, moving them to the far end of the clipping plane defined by the edge of our volume. The program does this by testing the position of the object that is at the end of the list (i.e., due to sorting this always the nearest object), and redrawing it in the distance when the camera passes it.

Also note that Python has a convenient notation for indexing from the end of a list using negative indices; indexing the list with [-1] reads from the end of the sorted list. If the element is behind the camera we use the line, `circles.insert(0,circles.pop())`, to remove or 'pop' the element from the end of the list and insert it at the beginning of the list maintaining our sorted order. We also change its z-value to move it away to the far end of the volume. This process repeats for the next object at the end of the list, which is now closest to the camera until an element is found that is still in front of the camera. As the list is sorted, once such an element is found we are sure that all elements are in front of the virtual camera. Finally, the frame is advanced and the elapsed trial time checked. Figure 1 shows the resulting looming pattern of objects on the screen.

Example program 2: Active motion (riding a scooter)

The second example program demonstrates the use of the keyboard for simple interactive stimuli for psychophysical experiments and is shown in Listing 2. Pressing the arrow keys moves the camera in the scene, and gives the effect of riding a scooter through a virtual town. Specifically, this function does the following:

1. It "listens" for keyboard input
2. It moves the position of the camera forward or back (up/down key)
3. It rotates the forward direction of the camera (left/right key)
4. It rotates the scooter frame with respect to the camera position
5. It displays the trial runtime on screen

Listing 2: Section of the active vection program.

```
1. if scene.kb.keys:
       s = scene.kb.getkey()

   2. if s == 'up':
          scene.center.x=scene.center.x + 3*scene.forward.x
          scene.center.y=scene.center.y + 3*scene.forward.y
          scene.center.z=scene.center.z + 3*scene.forward.z

      elif s == 'down':
          scene.center.x=scene.center.x - 3*scene.forward.x
          scene.center.y=scene.center.y - 3*scene.forward.y
          scene.center.z=scene.center.z - 3*scene.forward.z

   3. elif s == 'left':
         scene.forward=scene.forward.rotate(angle=(degrot*pi/180),axis=(0,1,0))
         ScFr.rotate(angle=(degrot*pi/180),axis=(0,1,0), origin= ScFr.pos)

      elif s == 'right':
          scene.forward=scene.forward.rotate(angle=(degrot*pi/-180),axis=(0,1,0))
          ScFr.rotate(angle=(-degrot*pi/180),axis=(0,1,0), origin= ScFr.pos)

      ScFr.pos.x = scene.center.x + scooterDist*scene.forward.x
      ScFr.pos.y = scene.center.y + scooterDist*scene.forward.y
      ScFr.pos.z = scene.center.z + scooterDist*scene.forward.z
```

Keyboard interaction. The function `scene.kb.keys` listens for keyboard input, and if a key is pressed, `scene.kb.getkey()` registers the value of the keyboard. In this program, the accepted input keys are the up, down, left, and right key. Using the mouse works the same way but with the functions `scene.mouse.events()` and `scene.mouse.getevent()`.

Moving forward and back. Camera motion involves moving the scene and forward direction of the camera. In this program we move the camera 3 units from its current position each time an up/down key is pressed. The same key press method can be used to collect user input for timed tasks by using "`elif len(s)==1`" to register a press of the "return" button, or to move a slider on a slider bar for a magnitude estimation task.

Rotation. Turning in the virtual town involves changing the forward direction of the camera. Using the function `scene.forward.rotate`, rotates the forward direction of the camera for a given angle and axis of rotation. Because the scooter handles must remain in the same relative position to the camera, they too are rotated. The object "handles" is composed of several objects in a frame called `ScFr`, it is turned using `ScFr.rotate`, and an origin for the rotation is also specified. The next section maintains the distance of the scooter to the camera as it is rotating, by positioning the scooter frame at the camera location and adding the forward direction of the camera; here, the frame object is offset from the camera, so its distance to the camera must scale the camera direction.

Modeling natural motion. Visual motion in a VPython program can easily be manipulated to suit the needs of the programmer. This can be done by adding camera jitter or oscillation as mentioned earlier, or other motion characteristics, to the x, y, or, z-axis position of the camera. More sophisticated programs could implement physical constraints and interactions. In the scooter code, a portion of road can cause a bumpy camera motion when travelling over it, by specifying that the camera y position follow the road contour when the scooter traverses that portion of the road; this is done using a simple `if/then` statement. In the program, a jittering motion occurs when the user travels across speed bumps on the road.

Figure 2: Screenshot of the virtual scooter program with a timer and speed bumps.

Real-time screen output. Text, such as instructions, user feedback, and elapsed time, can be shown on screen using the "label" function. In the scooter program, displaying a timer involves continually deleting and redrawing the elapsed time on screen. Since the label function reads strings, the time is converted from a number into a string using the function `str()`. Figure 2 shows a screenshot of the scooter program; the timer is visible in the top-left corner of the figure, and the road bumps are straight ahead of the scooter frame in blue.

Limitations

While VPython makes it is easy to produce dynamic visual content, this is not to say it is not without limitations or that it cannot be improved. Importing from visual.controls enables the use of interfaces such as buttons and sliders, but these controls can only be added to a special controls window that is separate from the main window. This can be an inconvenience if the programmer wishes to display both the controls and stimuli in the main window. However, it is possible to present both windows simultaneously by placing them side-by-side. Additionally, such interfaces can be created with VPython's objects tools with relative ease.

The most significant limitations of VPython for visual psychophysics revolve around the ability to precisely and reliably control the timing of visual stimuli and the provision of facilities for display calibration. Overcoming these issues requires dealing with the specifics of particular operating systems and computing platforms. These extensions can be added to a VPython solution or more sophisticated Python-based solutions such as Vision Egg or PsychoPy can be used. However VPython allows for rapid implementation and testing of visual experiments and is ideal for experimental psychologists to learn to implement these displays.

Discussion

Using VPython to create rich stimuli and interfaces for psychophysical experiments is free and simple to learn and use. The example programs shown here demonstrated that it is capable of presenting simple and dynamic 3D stimuli, at little computational cost or demand on a relatively lengthy list of libraries; see http://www.vpython.org for stereoscopic options. As the capability of VPython and other Python modules increases, we expect to see a greater shift of psychophysicists to this versatile and powerful programming platform.

References

Lutz, M. (1996). *Programming Python*, 1st Ed.. O'Reilly Media Inc.
Straw, A. D. (2008). Vision Egg: An Open-Source Library for Realtime Visual Stimulus Generation. *Frontiers in Neuroinformatics*. doi: 10.3389/neuro.11.004.2008.