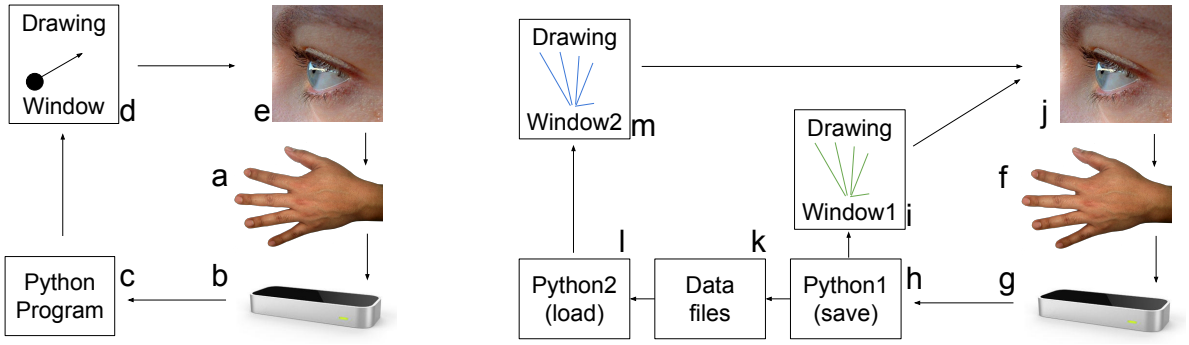

Figure 1: In the first two deliverables, a user gestures (a); the gesture is captured by Leap Motion (b); the data is in turn captured by your Python program from Leap Motion (c); your Python program draws the data to the screen (d); which is seen by the user (e); and the interaction continues. In this deliverable, you will maintain the basic feedback loop (f,g,h,i,j), but will add some functionality that enables the user to record gesture data using their other hand. The recorded gesture data is saved out to a file (k). The data can be read in by a second Python program that you will now write (l), which draws the recorded data in a a second drawing window (m). Demostration video here.

CS228: Human Computer Interaction

Deliverable 3

Due: Monday, September 22, 11:59pm

Description

In this deliverable, you will expand your Python code to allow the user to save gestures to a file. You will then create a second Python program that can load in those gestures and draw them (see Fig. 1). This video shows how your two Python programs should behave when you complete this deliverable successfully.

Instructions

1. In the directory where you developed the second deliverable, make a copy of Del2.py and call it Del3.py. This way, you have always have a copy of your second deliverable: if you become stuck during this deliverable, you can always recopy Del2.py to Del3.py and start back here.

2. The first thing we are going to do is convert the code in Del2.py into object-oriented code. This will make our code much more modular, and easier to expand as we proceed. We are going to do this to create a class called Deliverable. That class will include all of the variables and functions you created for the last deliverable. At the moment, check that

Del2.py contains four global variables (controller, lines, fig and ax), a few lines that initialize Leap Motion and the drawing window, and then one big infinite loop:

(a) (*lines that import various Python libraries*)
(b) controller = Leap.Controller()
(c) lines = []
(d) matplotlib.interactive(True)
(e) fig = plt.figure(...)
(f) ax = fig.add_subplot(...)
(g) ax.set_xlim(...)
(h) ax.set_zlim(...)
(i) ax.set_ylim(...)
(j) ax.view_init(...)
(k) plt.draw()
(l) while (True):
(m) ...

3. Convert your code to the following:

(a) class Deliverable:
(b) def __init__(self):
(c) self.controller = Leap.Controller()
(d) self.lines = []
(e) matplotlib.interactive(True)
(f) ...
(g) def RunForever(self):
(h) while (True):
(i) ...
(j) deliverable = Deliverable()
(k) deliverable.RunForever()

This defines a class called Deliverable (a), and creates two functions that belong to that class: an initialization function (b-f) and a function that runs forever when called (g-i). Line 3(j) creates an instance of that class (an object called deliverable) and implicitely calls the initialization function. Then, the RunForever function belonging to that object is called (line 3(k)). The lines within __init__ create four variables belonging to this class: self.controller, self.lines, self.fig and self.ax. The 'self.' prefix indicates that these are not normal variables, but variables belonging this class. Wherever you refer to these variables inside __init__ or

RunForever (or within the additional functions you will be adding to this class), be sure to use the 'self.' prefix. Any local variables that you make use of only within a function (such as the 'hand' variable inside RunForever) do not need the prefix. Run and debug your code until it runs and behaves exactly like Del2.py. This will tell you that you've successfully made your code object oriented.

4. Let us now further modularize your code: put all of the lines of code that are called each time through the infinite loop (line 3(i)) and put them inside a third function, RunOnce. Replace these lines in RunForever with a call to RunOnce:

```
(a) class Deliverable:
(b)         ...
(c)         def RunOnce(self):
(d)                 ...
(e)         def RunForever(self):
(f)                 while (True):
(g)                         self.RunOnce()
(h) ...
```

   Debug until your program behaves as before.

5. Let's modularize further. Create a function HandleHands that handles the case when a hand is detected above the device. Another function HandleFinger handles each finger in that hand, and a third function (HandleBone) that handles each bone in that finger:

```
(a) def HandleBone(self,i,j):
(b)         ...
(c)         self.lines.append(...)
(d) def HandleFinger(self,i):
(e)         self.finger = self.hand.fingers[i]
(f)         for j in range(0,4):
(g)                 self.HandleBone(i,j)
(h) def HandleHands(self):
(i)         self.hand = self.frame.hands[0]
(j)         for i in range(0,5):
(k)                 self.HandleFinger(i)
(l)         plt.draw()
(m)         (delete the drawn lines)
(n) def RunOnce(self):
```

(o) self.frame = self.controller.frame()

(p) if (*there is a hand above the device*):

(q) self.HandleHands()

Note that all of the lines are added to the drawing window in HandleBone (line 5(c)). Once they have all been added, they are drawn to the window in HandleHands (line 5(l)). Immediately afterward, the lines are deleted using the code introduced as lines 17(a)-(e) in the previous deliverable. Debug until your program behaves as before.

6. We are now going to add functionality that will enable the user to record gesture data at will. They will do this using their other hand: when she waves her secondary hand into and then out of the device's field of view, a 'snapshot' of the primary hand's current state will be captured. (For a right-handed person, their primary hand is their right hand and their secondary hand is their left hand. The situation is reversed for a left-handed person.) Let us start by coloring the drawn hand differently to signal these different functions to the user: draw the hand using green lines when only a single hand is detected, and draw the hand using red lines when two hands are detected. To do so you will need to first store the number of hands in the Deliverable object

(a) def HandleHands(self):

(b) self.numberOfHands = (*extract this from self.frame*)

(c) self.hand = self.frame.hands[0]

7. Then, create a conditional when you draw a line representing a bone

(a) def HandleBone(self,j):

(b) ...

(c) if (self.numberOfHands == 1):

(d) self.lines.append(*draw a green line*)

(e) else:

(f) self.lines.append(*draw a red line*)

Test your code. The green drawn hand should turn red when you wave your secondary hand into the device's field of view, and it should turn green again when you wave it back out.

8. We are not recording any data yet, but practice 'recording' several times with your program. You will note that if you wave your hand in too close to your gesturing hand, the device will have a hard time distinguishing between your two hands and the gesture will be corrupted. Try learning to wave your hand just inside the device's field of view so that you do not disrupt the gesture you are making with your primary hand. You will notice that you quickly start to build up a **mental model** of where that invisible boundary is. What visual feedback are you getting that enables you to build up this mental model? Note that eventually you are going to try out your software on naive users: how will they learn about this potential pitfall? How will they learn to avoid it?

9. To keep things simple, we are going to assume for the rest of this project that we are only going to capture static gestures from the device: that is, we are only going to capture one frame of data that indicates the current position of the hand, not how that hand changes position over several frames. In later deliverables we are going to expand our code to recognize individual letters from the American Sign Language (ASL) alphabet. You will notice that two of those letters—*j* and *z*—require movement, but the other 24 letters are static. So, eventually, your code will learn to recognize 24 of the 26 ASL letters. Familiarize yourself with a few of these letters, and try gesturing them above your device: does the device interpret the gesture correctly? Do you need to shake or rotate your hand to get it to do so?

10. So, when the user indicates she wishes to 'record' a gesture, which frame do we capture? We are going to capture the frame immediately after the second hand leaves the device's field of view: that is, right at the end of the recording period. This gives the user time to adjust her gesture during the recording period before 'snapping a picture' of the gesture. How can we identify this point of time in our code? We can do so by finding the point at which the current number of hands above the device is one, but there were two hands above the device during the previous pass through the infinite loop. To do so we need to add two additional variables to the class

    (a) class Deliverable:
    (b) def __init__(self):
    (c) previousNumberOfHands = 0
    (d) currentNumberOfHands = 0
    (e) ...

11. Then, update these variables during each pass through the infinite loop: Replace line 6(b) with

    (a) self.previousNumberOfHands = self.currentNumberOfHands
    (b) self.currentNumberOfHands = (*extract this from self.frame*)

    Replace line 7(c) with

    (a) if (self.currentNumberOfHands == 1):

    Finally, change the conditional in the HandleBone function to refer to currentNumberOfHands rather than numberOfHands. Test your code. You should see no visible difference in code behavior compared to the previous step.

12. We will now print a message to the screen whenever the secondary hand leaves the device's field of view:

    (a) def RecordingIsEnding(self):
    (b) return (self.previousNumberOfHands==2) & (self.currentNumberOfHands==1)

(c) def HandleHands(self):

(d) ...

(e) if (self.RecordingIsEnding()):

(f) print 'recording is ending.'

(Note that lines 12(f-g) should be placed at the end of the HandleHands function.)

13. We are now going to write out the positions of each bone in the hand at exactly the moment in time when line 12(f) is reached. However, we face a challenge: the positions of the bones are lost every time we exit the HandleBone function. To overcome this, we are going to store all of the bone positions in a data structure, and then write that data structure to a file. The data structure we are going to use is a three dimensional matrix. The Python library Numpy, or 'numerical Python', allows for easy manipulation of arrays and matrices in Python. If you are using Canopy Express, Numpy was already installed for you. If you used something other than Canopy Express, download and install Numpy now.

14. First, let us import NumPy by adding this line

    (a) import numpy as np

    at the top of our program.

15. Now let us add a 3D matrix as a new variable to our class Deliverable by placing this line

    (a) self.gestureData = np.zeros((5,4,6),dtype='f')

    in the initialization function. This line indicates that we want a 3D matrix with five rows (one for each finger); four columns (one for each bone in each finger); and six 'stacks'. The first three stacks store the x, y, and z coordinates of the base of the bone and the second three stacks store the x, y, and z coordinates of the tip of the bone. Fig. 2 illustrates this data structure.

16. To help you think about this matrix, tick off those elements of the box in Fig. 2 that correspond to the x, y, and z coordinates of the *base* of the distal phalange in the index finger.

17. So far, we have just created the matrix and initialized each element to zero. Replace line 12(f) with

    (a) print self.gestureData[0,0,0]

    This will print out the value stored in the front stack of the leftmost column of the topmost row. When you run your code, you should see a single value printed each time you remove your secondary hand from the device's field of view.

18. Now replace this line with

    (a) print self.gestureData[0,:,:]

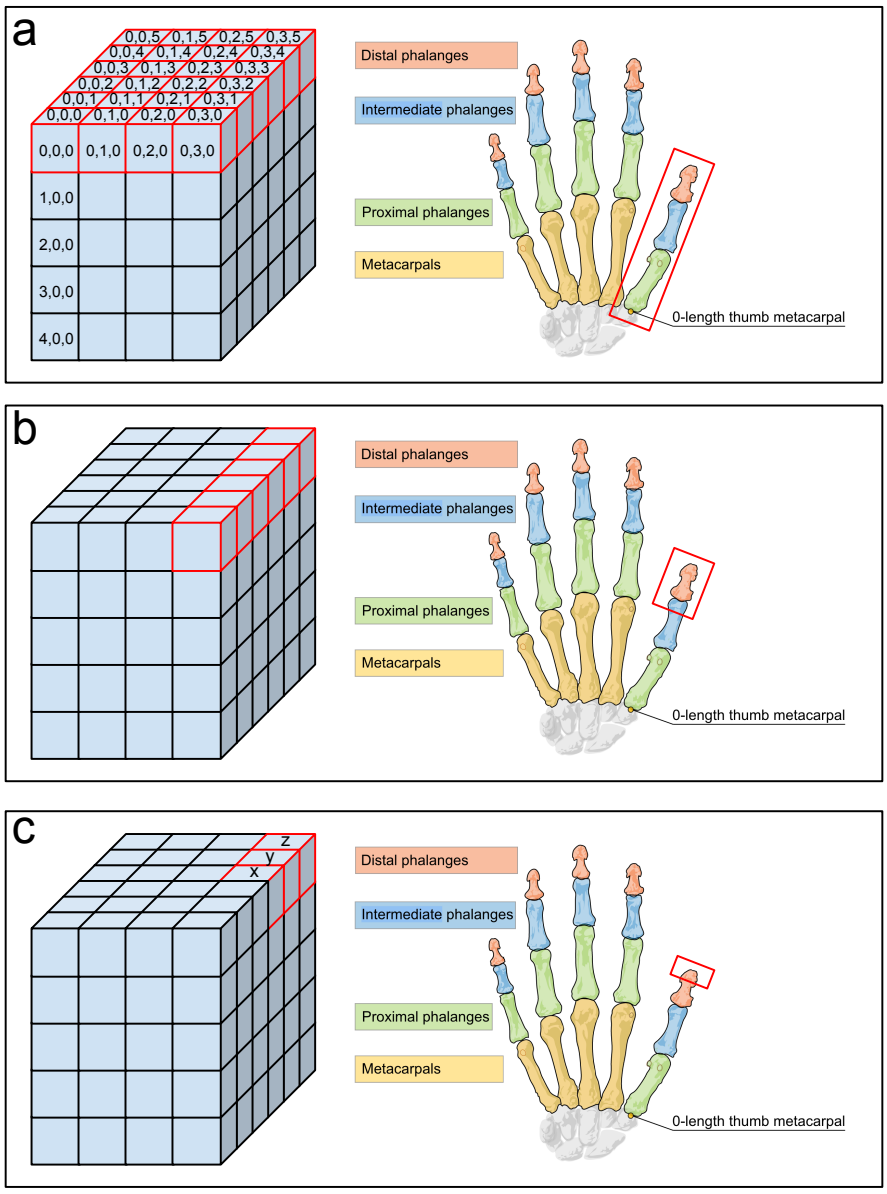

Figure 2: How the 3D matrix gestureData stores data (see step #15). Each row stores data about a specific finger; each column stores data about a specific bone; and each 'stack' (a horizontal sheet of elements) stores information about a particular coordinate of the base (the front three stacks) or tip (the back three stacks) of a bone. For example, (a) indicates that the first row (i.e. the top row) stores information about the first 'finger', which is the thumb. Within that row, the fourth column (i.e. the rightmost column) stores information about the fourth bone in the thumb, which is the distal phalange (b). The fourth, fifth, and six stack in this row and column encode the x, y, and z coordinate of that bone's tip (c). The first, second, and third stack encode the x, y, and z coordinate of the bone's base (now shown). The triplets inside each element indicate how to reference the number stored in it.

The colon indicates that you are referencing the entire range of values for that dimension of the matrix. In this example, you are asking Python to print each column and each stack of the first (i.e. the zeroth) row of the matrix. This corresponds to the red elements in Fig. 2a. Note how the coordinates of the elements in this topmost row range over their entire range for the second and third dimensions, but all have a value of zero for their first dimension. When you run your code now, you should see that it prints out an entire two-dimensional matrix (the top row of the 3D matrix) every time your secondary hand leaves the device's field of view.

19. Replace this line with a new line that prints out the vector composed of the red elements in Fig. 2b. Now when you run your code you should see a vector of six numbers printed out each time your secondary hand leaves the device's field of view.

20. Finally, replace this line with

    (a) print self.gestureData[0,3,3:5]

    This will print the elements (still currently all zeros) where the x, y, and z coordinates of the tip of your thumb's distal phalange will be stored. This line references the first row (i.e. row 0), the fourth column (i.e. column 3), and the fourth through the sixth stacks (stacks 3 through 5).

21. Now, finally, it's time for us to store data from the device into this matrix. In HandleBone, just after you've added a line to the drawing window (line 7(f)), store data about the current gesture into the matrix:

    (a) self.gestureData[i,j,0] = (*x coordinate of the base of bone j in finger i*)

    (b) self.gestureData[i,j,1] = (*y coordinate of the base of bone j in finger i*)

    (c) self.gestureData[i,j,2] = (*z coordinate of the base of bone j in finger i*)

    (d) self.gestureData[i,j,3] = (*x coordinate of the tip of bone j in finger i*)

    (e) self.gestureData[i,j,4] = (*y coordinate of the tip of bone j in finger i*)

    (f) self.gestureData[i,j,5] = (*z coordinate of the tip of bone j in finger i*)

    Your code should now print out the actual x, y, and z coordinates of the tip of your thumb's distal phalange (on your primary hand) when your secondary hand left the device's field of view.

22. Let us make sure that you are capturing all the data you need into the matrix. Replace line 20(a) with

    (a) print self.gestureData

    This will print out the contents of the entire matrix. You should see, when you run your code, that all of these values are now non-zero.

23. We have introduced some code that might slow down your code, so let's make it a bit more efficient. Clearly we only need to store a gesture in the matrix whenever recording stops. So, place lines 21(a-f) inside of an if statement. The if condition should evaluate to true—and the data stored in the matrix—only when the secondary hand leaves the device's field of view. Hint: Use one of the functions you have recently created.

24. Now we are going to save this matrix in a file whenever recording stops. Create a subdirectory called userData inside of the directory from which you have been running your Python program. We are going to store all of the data generated by a user in this subdirectory.

25. Inside HandleHands, replace line 12(e-f) with

    (a) if (self.RecordingIsEnding()):
    (b) print self.gestureData[:,:,:]
    (c) self.SaveGesture()

26. Define a new function above HandleHands as follows:

    (a) def SaveGesture(self):
    (b) pass

    (The 'pass' command just tells program execution to pass by and continue.) Run your program and make sure that it behaves the same as it did when you finished step #23.

27. Replace line 26(b) with the following:

    (a) fileName = 'userData/gesture.dat'
    (b) f = open(fileName,'w')
    (c) np.save(f,self.gestureData)
    (d) f.close()

    Line (a) defines the name of the file; (b) opens the file and indicates that it will be [w]ritten to; (c) uses NumPy's save command (np.save) to save the matrix gestureData to file f; and (d) closes the file. Run your program, wave your secondary hand in and out of frame, and then look inside the userData subdirectory. You should see a file in there called gesture.dat.

28. Now create a new Python program called `Del3b.py`. Make sure it is in the same directory as the rest of your files. Put just one line in it: `print 1`. Run Del3b.py, and make sure that it prints '1'.

29. Now replace `print 1` with the following lines:

    (a) import numpy as np
    (b) fileName = 'userData/gesture.dat'
    (c) f = open(fileName,'r')

(d) gestureData = np.load(f)

(e) f.close()

(f) print gestureData

This program will now load in the gesture data your other program just saved, and will print it out before closing. Run it; it should print out a 3D matrix with all non-zero values.

30. Let us make this Python program object oriented like we did in step #2:

(a) import numpy as np

(b) class Reader:

(c) def __init__(self):

(d) fileName = 'userData/gesture.dat'

(e) f = open(fileName,'r')

(f) self.gestureData = np.load(f)

(g) f.close()

(h) def PrintData(self):

(i) print self.gestureData

(j) reader = Reader()

(k) reader.PrintData()

The program `Del3b.py` now houses a class called `Reader`. Reader has an initialization function which reads in a 3D matrix from a file (lines (c-g)), and another function that prints out the loaded matrix (lines (h-i)).

31. Now, go back to Del3.py, and add a new variable `numberOfGesturesSaved` to the class. Initialize it in __init__ to zero. Run Del3.py again. Its behavior should not change.

32. Now add the following lines to the `SaveGesture` function in `Del3.py`:

(a) self.numberOfGesturesSaved = self.numberOfGesturesSaved + 1

(b) (*lines 27(a-d)*)

(c) fileName = 'userData/numOfGestures.dat'

(d) f = open(fileName,'w')

(e) f.write(str(self.numberOfGesturesSaved))

(f) f.close()

Note how line 32(e) differs from line 27(c): here, we are writing out a single number to a file, rather than a matrix. We first convert the number to a string (str()). This allows you to open this file and see the number written inside. Run `Del3.py`, record a few gestures, stop the program, and open `numOfGestures.dat`: the number there should match the number of gestures you just recorded.

33. Now modify your `SaveGesture` function so that it writes out a separate file for each recorded gesture. The first recorded gesture should be written to `gesture0.dat`, the next recorded gesture to `gesture1.dat`, and so on.

34. Now modify `Del3b.py`: add a variable `self.numberOfGesturesSaved` to the class `Reader`. Debug until it runs as before.

35. Modify the __init__ function in `Del3b.py` to read in only the number of gestures from numOfGestures.dat (and not any gesture data) and store it in the new variable. Make sure to convert the string loaded from the file into an integer using int(...). Modify line 30(i) to print out the number of gestures, rather than gesture data. When you run `Del3b.py` now, it should print out the number of gestures you recorded with `Del3.py`.

36. Now modify the `PrintData` function—and add a new function `PrintGesture`—so that this function prints out the data corresponding to each recorded gesture:

    (a) def PrintGesture(self,i):
    (b) gestureData = (load the data in `gesturei.dat` into a numpy matrix)
    (c) print gestureData
    (d) def PrintData(self):
    (e) for i in range(0,self.numberOfGesturesSaved):
    (f) self.PrintGesture(i)

    Note that now, the gesture data is just saved to the local variable `gestureData`. That data is lost when the program exits `PrintGesture`.

37. Now modify `Del3b.py` so that it draws each gesture rather than prints it out. To do so, first import the matplotlib libraries at the top of your program.

38. Initialize matplotlib to interactive mode and create the `self.lines`, `self.fig`, and `self.ax` variables in __init__ as you did in `Del3.py`.

39. Create a function called `RunForever`, include an infinite loop in it, call `PrintData` from within that loop, and call RunForever on line 30(k) rather than PrintData.

40. Replace line 36(c) with the following:

    (a) for i in range(0,5):
    (b) for j in range(0,4):
    (c) xBase = gestureData[i,j,0]
    (d) ...
    (e) print xBase, yBase, zBase, xTip, yTip, zTip

    Run your code, and you should now see it continously print out the positions of the base and tip of each bone, for each gesture.

41. Replace line 40(e) with:

    (a) self.lines.append(self.ax.plot([-xBase,-xTip],[zBase,zTip],[yBase,yTip],'b'))

    This will draw the lines in [b]lue to help your user (later on) distinguish between green ('play'), red ('record') and blue ('play back').

42. After line 41(a), call `plt.draw()` and include the code that deletes all of the just-drawn lines (lines 17(a-e) in Deliverable 2). Make sure that these lines are only called when the two nested loops have completed. When you run your code now, you should see the gestures you recorded play back, one after the other, in very rapid succession.

43. Let's slow things down a bit. Import the library `time` at the top of your code, and then call `time.sleep(0.5)` immediately after you have deleted all of the drawn lines. When you run your code now, you should see each gesture shown for half a second.

44. Now rename `Del3.py` to `Record.py`, and `Del3b.py` to `Playback.py`.

45. Since you need both hands to record gestures, and since none of us has three arms, you are not required to video your interaction with `Record.py` with a smartphone. Instead, capture some video of just `Playback.py` in action; upload this single video to YouTube; and submit the URL of this video to BlackBoard.