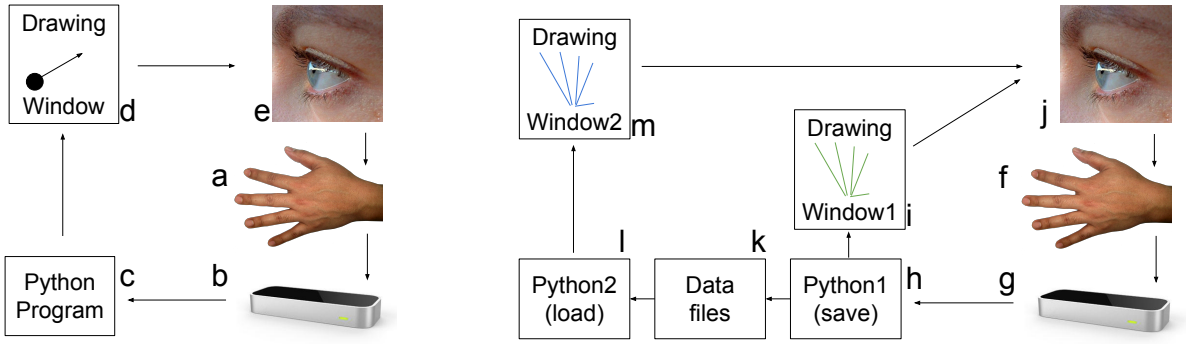

Figure 1: In the first two deliverables, a user gestures (a), the gesture is captured by Leap Motion (b), the data is in turn captured by your Python program from Leap Motion (c), your Python program draws the data to the screen (d) and seen by the user (e), and the interaction continues. In this deliverable, 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).

Creating ASL Educational Software using Leap Motion

Josh Bongard
Department of Computer Science
University of Vermont

Deliverable 3: Recording Gestures

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).

Instructions

1. In the directory where you developed the second deliverable, make a copy of Del02.py and call it Del03.py. This way, you always have a copy of your second deliverable: if you become stuck during this deliverable, you can always recopy Del02.py to Del03.py and start back here.

2. The first thing we are going to do is convert the code in Del03.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 by creating a class called `DELIVERABLE`. To do so, create a new file called Deliverable.py, and place the following stub into it.

   ```
   class DELIVERABLE:
   ```

```
def __init__(self):
    pass
```

Back inside of Del03, create an instance of this class as the new first line of your code, and call `exit()` immediately after it to stop your program. When you run your program now, nothing should happen. Why do you think this is so?

3. We are now going to move the global variables inside of Del03.py into this class, and turn them into instance variables. (If you're not familiar with instance variables, you can read about them here.) Del03.py should currently include the following global variables:

   `controller, pygameWindow, x, y, xMin, xMax, yMin, yMax.`

   Cut these out of Del03, include them in DELIVERABLE's constructor, and place the appropriate prefix in front of them to turn them into instance variables. Run your code now, and you should still see that it does nothing because we are not doing anything with these variables yet. Hint: If you get error messages, you may need to move some import statements from Del03.py into Deliverable.py.

4. Similarly, Del03.py contains several functions. Cut and paste these functions into your new class such that they become its methods. Run your code again: it should still do nothing.

5. The only code inside Del03.py should now be the construction of an instance of DELIVERABLE, the exit statement, and the infinite loop. Cut and paste the entire loop into a new method inside of `DELIVERABLE` called `Run_Forever()`. Delete the exit statement and replace it with a call to Run_Forever. When you run your code now, you may get a number of error messages. These will require you to include `self` as an argument in each of the methods, and add the `self` prefix to any method calls within `DELIVERABLE`. Continue until your code runs as it did in the previous deliverable.

6. We'll perform one more act of refactoring. Cut all of the lines called from within the infinite loop and place them in a new method called `Run_Once`. Call this method from within the infinite loop.

7. Congratulations: you now have a more modular—and thus more easily modifiable—code base.

8. 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.

9. To do so you will need to first store the number of hands in a new instance called

   `self.numberOfHands`. You will do this just before you use that variable in the conditional statement which determines what do next, based on the number of hands detected.

10. We are now going to make some changes to `PYGAME_WINDOW`. But, if we do, we will break your code for the first two deliverables because they rely on `pygameWindow`. We can break this dependency be copying `pygameWindow.py` and renaming it

    `pygameWindow_Del03.py`. You will need a statement at the top of `Deliverable.py` to ensure you import this new version of the class. You can now freely modify

    `pygameWindow_Del03.py` without breaking your original code. Make sure to add, commit, and push this new file to your github repository, as well as any other changes you've made to this point. It is a good to get in the habit of adding, committing and pushing to your repo often during code development.

11. Now, rename pygameWindow's method `Draw_Black_Line` to `Draw_Line`. (Make sure you do so in the correct version of pygameWindow.) Pass a new argument into this method which indicates what color the line should be drawn with. Inside the method, use this argument to draw a line using the specified color. Experiment with drawing the hand with different colors, by passing in different `color` arguments.

12. Now, just before this method is called, use `self.numberOfHands` to draw the line green if one hand is over the device and red if two hands are over it. 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.

13. 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?

14. 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 signs from the American Sign Language (ASL) alphabet. However, you will note that two of the letters require movement rather than a static pose. For this reason, we are going to limit ourselves to recognizing only the digits zero through nine in ASL.

    Familiarize yourself with a few of these digits, 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?

15. To train our system in later deliverables to recognize these digits, we must train our system on actual data captured from people signing these digits. But 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 there is one hand above the device, but there were two hands above it during the previous pass through the infinite loop.

To do this, delete the variable `numberOfHands` and replace it with two new ones:

`previousNumberOfHands` and `currentNumberOfHands`. Initialize them to zero in the method's constructor.

16. Now, just after a frame is captured, store the current number of hands in the appropriate variable. As your code is about to exit an iteration through the infinite loop, store the current number of hands in previousNumberOfHands. During each pass through the infinite loop, we now know how many hands are—and were just—over the device. Run your code now. You may get some error messages if you are still using `numberOfHands`. Please replace it with the appropriate variable at the appropriate places in your code.

17. Now we want add some new functionality just as the secondary hand is leaving the field of view. Create a new function called `Recording_Is_Ending` such that it returns true when there is now one hand over the device, but there were two over it during the previous iteration through the loop. At the end of the `Handle_Frame` method, add this code:

    (a) `if self.Recording_Is_Ending():`

    (b) `    print('recording is ending.')`

18. We are now going to write out the positions of each bone in the hand at exactly the moment in time when line 17(b) 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.

19. You can check whether you have numpy installed for Python2.7 trying to import numpy at the top of Del03.py. If you get an error, the easiest way to install it is with

    `pip install numpy`

    If you had problems with pip in the first deliverable, here are suggestions for other ways to install it.

20. First, let us import NumPy by adding `import numpy as np` at the top of DELIVERABLE.

21. 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 its constructor. 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.

This 3D matrix is represented in numpy as an array. If you're not familiar with numpy arrays, this is a useful introduction.

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

23. So far, we have just created the matrix and initialized each element to zero. Replace line 17b with

    ```
    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.

24. Now replace this line with

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

    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.

25. 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.

26. Finally, replace this line with

    (a) `print(self.gestureData[0,3,3:6])`

    This will print the elements (still currently all zeros) where the x, y, and z coordinates of the tip of your thumb's distal phalanx 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). The '6' is required because numpy interprets this as 'give me the values from stack 3 up to, but not including, stack 6.'

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

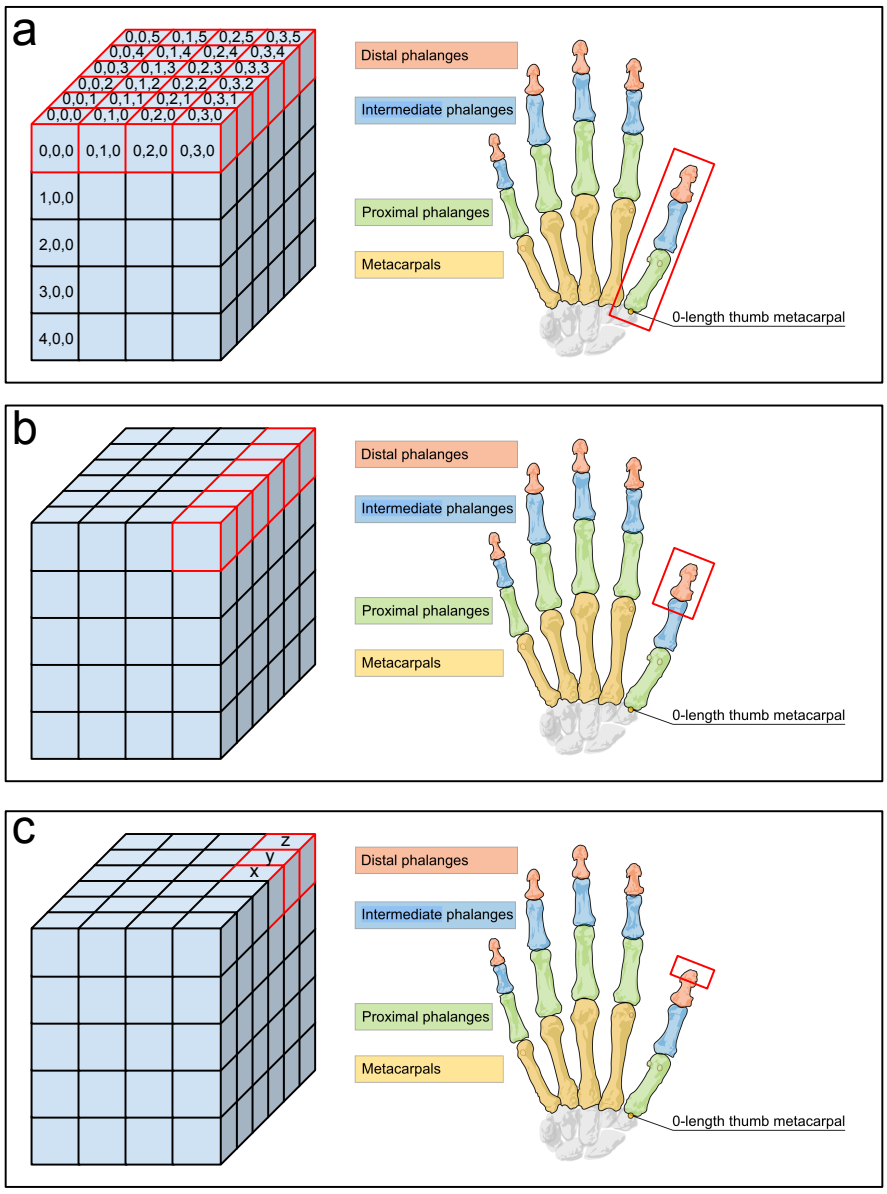

Figure 2: How the 3D matrix gestureData stores data (see step #15). Each row stores data about a specific finger; each column stores data about a specific bone; and each 'stack' (a horizontal sheet of elements) stores information about a particular coordinate of the base (the front three stacks) or tip (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 phalanx (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.

(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*)

You might need to alter `Handle_Finger` and `Handle_Bone` such that, inside of `Handle_Bone`, i and j are both integers, i ranges between zero and four inclusive, and j ranges between zero and three inclusive. You may wish to print i and j at this point in your code to ensure this is indeed occurring.

Also you must store the raw values extracted from `prev_joint` and `next_joint`, before they have been manipulated and scaled by `Handle_Vector_From_Leap`. The reason for wanting to store the raw values is because, eventually, you'll be combining the data you record with data recorded by the other students in the class. Each student may be creating pygame windows of different shapes and sizes, and we do not want those differences reflected in the data.

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

29. Let us make sure that you are capturing all the data you need into the matrix. Replace line 26a 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.

30. We have introduced some code that might slow down your code, because we're capturing gesture data every time through the infinite loop, rather than when the secondary hand leaves the device's field of view. 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 27(a-f) inside of a conditional statement. The conditional 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.

31. 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. Add it to your git repository. We are now going to store all of the data generated by a user in this subdirectory.

32. In your code, immediately after you print the gesture data, add a call to a new, initially empty method called `Save_Gesture()`. Run your program and make sure that it does not do anything differently.

33. Now, whenever you capture data about a gesture, you will save it to a file using another python package called `pickle`. If you're not familiar with it, here's a good introduction. Import `pickle` at the top of DELIVERABLE, and then dump `self.gestureData` from within `Save_Gesture` to a file called `gesture.p` in the `userData` subdirectory. (The p suffix will remind us that this is a pickled file.)

34. 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.p`.

35. When next you push your changes to your git repository, it is up to you whether you wish to add the data files that accumulate in the data subdirectory to the repository or not. For now, it does not matter.

Draw recorded gestures.

36. We are now going to create a separate program that reads in this saved data, and draws the virtual gesture it represents to a Python drawing window. To start, create a new Python program called `Del03b.py`. Make sure it is in the same directory as the rest of your files, and add it to your repository.

37. Inside this new file, import numpy and pickle, load the data in using pickle, store it in a variable called `gestureData`, and print gestureData before the program terminates. Run your code. What do you see? What were you expecting to see?

38. Let us make this Python program object oriented like we did for Del03. Do this by creating a new file called `Reader.py`, create a class called READER inside of it, import it from Del03b.py, create a class instance from it in Del03b called Reader, and, inside of READER's constructor, cut and paste the code in Del03b that reads in the gesture data and prints it. Make sure to add, commit, and push your file (and any other new changes) to your repository. When you run your code now, you should no change compared to the previous step.

39. We are now going to modify DELIVERABLE such that, instead of continuously writing new gesture data to the same file, it will write a new file for each gesture captured. These files will be written out as gesture0.p, gesture1.p, and so on. You can do this by creating a new variable in DELIVERABLE, initially set to zero, which increments every time some gesture data is written to a file. You can then use it to dynamically set the name of the output file to which pickle dumps your data. Run your code now, record a few gestures, and look in the data subdirectory. Are the number of files in there equal to the number of gestures you recorded?

40. Verify that different gestures were saved into these files by loading some of these files into Reader and printing them out.

41. **IMPORTANT**: You are now going to create code that automatically deletes files and directories. Make sure that before you carry out the next instruction that you have added, committed and pushed all of the files that you want to preserve to your git repository. Then, if you make a mistake and your code accidentally deletes your code rather than your data, you can pull your code from your repository back to your computer.

42. Step 39 can lead to a problem however. If you run Del03.py once and you cause it to save four gestures, and then you run it again and cause it to save only three gestures, the data subdirectory will contain four files: the first three from the second time you ran your code, and the fourth file is from the first time you ran your code. This can cause problems later on, as we want the subdirectory to only contain the data you just captured from the last time you ran your code. We can solve this by emptying this subdirectory every time we start up Del03.py. To do so, we will need to import yet another library called `os`. (This also comes preinstalled with Python, so no need for pip.) Create a new method that is called from within DELIVERABLE's constructor. This method should use methods from the `os` library to delete the data subdirectory (and thus all the files within it) and then create it again. You can determine how to do this by Googling

    `python os delete directory`

    and

    `python os create directory.`

    When you have coded this up, carry out the example described at the start of this instruction, and look in the data subdirectory both times. You should see that the number of files equals the number of recorded gestures both times.

43. Now, in READER, we are going to read in the data from each recorded gesture in turn, and print the results. To do so, we first need to know how many gestures have been recorded. Create a new method that is called from READER's constructor that uses the following code

    (a) `path, dirs, files = next(os.walk('userData'))`

    (b) `self.numGestures = len(files)`

    to determine and store the number of recorded gestures.

44. Now create a new READER method called `Print_Gestures`. Have Del03b.py call this method. The method should iterate from zero to `self.numGestures-1`. During each iteration of this loop, it should read in that gesture and print the data corresponding to it to the screen. You can verify that it is not simply reading the first of several files in repeatedly by looking at the values of the printed matrices to see that they differ.

45. Instead of printing gesture data, we will now draw it. We'll do so by again using pygame, and allowing READER to make use of the `PYGAME_WINDOW` class. To get started, give READER a new method called `Draw_Gestures`. Call it from Del03b.py instead of

    `Print_Gestures`. Initially leave it empty. When you run Del03b.py now, nothing should happen.

46. Using the DELIVERABLE class as a guide, create an instance of `PYGAME_WINDOW` in READER's constructor and store it in a variable called self.pygameWindow. When you run your code now, you should see your code create a pygame window and then immediately terminate.

47. Similar to DELIVERABLE's loop (which you can use as a guide), within READER's

    `Draw_Gestures` method, create an infinite loop. Inside the loop, call a new method called `Draw_Each_Gesture_Once`. Inside this new method, copy and paste the for loop from `Print_Gestures`. Instead of loading and printing the data however inside the loop, create a new method called `Draw_Gesture` that takes as an argument the integer indicating the current gesture to be drawn. Call this method from within the for loop. When you run your code now, you should see the for loop iterating forever. This is how we are going to draw the gestures: one after the other until they're all drawn, and then start with the first one again.

48. We will now wipe the screen, draw the current gesture, and reveal it to the user during each call to `Draw_Gesture`. To do so, call PYGAME_WINDOW's `Prepare` method just before the print statement and the `Reveal` method immediately afterward. Since we are not drawing anything yet, this should have no impact on the behavior of your code when you run it.

49. Now, just after the print statment, copy in code from elsewhere in READER that loads in that gesture's data (but does not print it). We have the data right where we need it; now we just have to draw it.

50. To do so we will have to pull subsets of data out of `gestureData`, each subset should correspond to one bone. We'll then draw that bone. So, again using DELIVERABLE as a guide, right after loading the gesture data, create a nested loop. The outer loop should iterate over variable i from zero to four inclusive, and the inner loop should iterate over variable j from zero to three inclusive. Have a look at instruction 27. Look familiar?

51. Inside this nested loop, extract the six coordinates describing the 3D positions of the base and tip of the jth bone in the i finger into a variable called `currentBone`. Fig 2b may be helpful here. Create a print statement immediately after this assignment (still within the nested loop) like this:

    ```
    print(g,i,j,currentBone)
    ```

    what do you expect this to print when you run your code? What do you see?

52. Now extract the x coordinate of the base of the current bone and store it in a variable called `xBaseNotYetScaled`. Do the same for the y coordinate of the base, and the x and y coordinates of the tip. Fig 2c may be helpful here. Print these variables now instead of `currentBone`. Do you see what you expected to see?

53. Copy and paste your function from the previous deliverable that scales coordinates into READER. (You will need to modify it a bit to change it from a function into a method that belongs to READER. If you are not familiar with the difference between a function and method in Python, this tutorial may help.) Call this method with each of the four variables,

in turn, and store in the results in `xBase, yBase, xTip` and `yTip`. This figure might be helpful again. We are not going to dynamically set the minimum and maximum values of the raw coordinate values in this deliverable, so set the minimum values to some very large negative number and the maximum values to some very large positive number. For our purposes, these set numbers do not matter too much, but make sure you set them in constants.py so you can easily find and modify them later if need be.

54. Change the print statement to print these scaled variables instead. Run your code now. You should see that, if they are used for drawing to the pygame window, they should have valid ranges. That is, they should always be greater or equal to zero, the x coordinates should always be less than pygameWindowWidth, and the y coordinates should always be less than pygameWindowDepth.

55. Now we are ready to draw instead of print. Replace the print statement with a call to pygameWindow's `Draw_Line` method. For now, draw all the lines with the same line thickness, and draw them in blue. You should see something like this. In this example, I set my xMin and yMin to -1000, and xMax and yMax to +1000. I recorded two gestures. You'll note that my program alternates draws both gestures, very rapidly, and upside down.

56. Capture a video of your gestures being played back. It is fine if you have more than two gestures, and if the virtual hand is drawn backwards or upside down in a different way. Create a new playlist for this Deliverable, upload this video, and add it to the playlist.

57. First, let us slow down the drawing a bit. Import a new library called `time` into Reader.py, and right after you call `Reveal`, put this line:

    ```
    time.sleep(0.1)
    ```

    This line will pause your code for a tenth of a second before continuing. Play with this number so that you can actually see each gesture before the next one is drawn.

58. Now modify your code so that your recorded gesture is drawn right side up.

59. Also, if you recorded a gesture with your right hand but it looks like it is being drawn as a left hand (or vice versa), modify so your code so that the virtual hand is shown with the correct horizontal orientation.

60. If it is hard to see the details of the gesture, reduce the values stored in `xMin, yMin, xMax` and `yMax`.

61. When you run your code now, you should see something like this.

62. Record a video of your version, upload to YouTube, and append it to your playlist. Again, it does not matter how many gestures you have, just that the grader should be able to see the virtual hand in enough detail to verify it has a correct orientation.

63. Now let us use the reader for what it was designed for: to help you verify that you are recording 'clean' data. Start up your recording software, and record yourself gesturing ASL zero, one and then two. Run your playback software to verify that the device captured these gestures correctly: that is, that the three drawn gestures are clearly the first three ASL digits.

64. Now rename `Del3.py` to `Record.py`, and `Del3b.py` to `Playback.py`. Make sure these renamed files are correctly captured, committed, and pushed to your git repository.

65. 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: show it playing back the first five ASL digits (this of course means you need to record them first.) Upload this video to YouTube, add it to this Deliverable's playlist, and submit your playlist to BlackBoard. You should have three videos in your playlist.