Michael Zhong
mqz2001
May 16th, 2013

Final Project: Swype in the Air

Introduction

Gestural keyboards have become a popular alternative to character-by-character entry on touch screen devices today. The obvious limitation to text entry via a touch screen though is that it is fixed to the device. There are other limitations to using gestural keyboards on a touchscreen. Two particular issues are that first, because there is a visual reference, the set of allowable patterns is limited—that is, there is less room for error in the stroke when trying to outline a word. A second issue is that touchscreens experience the "fat finger" problem, that is that oftentimes one's finger blocks the view of what one is trying to type, making the user more prone to errors.

For this project, I aimed to venture into exploring the field of per-word gesture text entry, but in the air rather than limited to a touchscreen. There have been text-entry systems that have input text in the air, and there certainly have been scores of per-word, gestural, text-entry systems for a touchscreen—however, I believe this is the first time the advantages of per-word gesturing have been combined with text entry in the air. Certainly, with the motion sensing device used—the Leap Motion, which claims to be the most precise finger and hand tracker in the world—it is the first of its kind.

This project aims to address some of the drawbacks of entering text on a touchpad while simultaneously outlining and implementing different methods for overcoming the shortcomings of its own approach, entering text through the air.

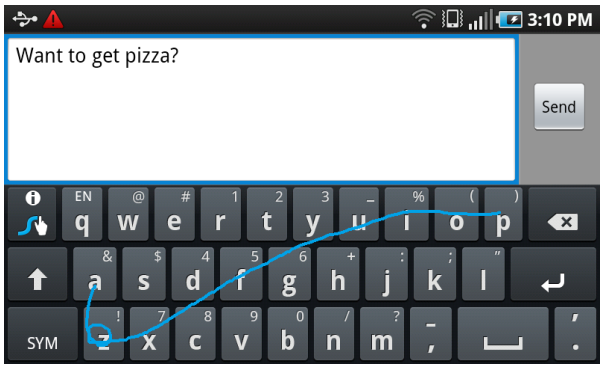

Figure 1: Example of a gestural keyboard (via the commercially available text entry system Swype). Here the word pizza is being spelt out in one continuous motion.

Preprocessing a Dictionary of Words

The first steps for both this project and for gesturing on a touchscreen are identical. Both systems rely on a grammar of potential words to choose from and both associate word frequencies with each word (so for example the word "the" appears more often in the human language than another corresponding three letter word, for example "bee"). Ideally a data source that provides each word in the human dictionary as well as its usage in frequency in recent times would be ideal. Unfortunately, such datasets are only commercially available. For example, wordfrequency.info has a list of the 100,000 most frequently used words and their corresponding frequencies available for 250 dollars. Likewise, word frequency lists for commercial single-stroke gesture keyboards like Swype or SwiftKey are kept strictly confidential. So I improvised.

I took word counts from three sources: the most frequent words from Project Gutenberg which scraped words from a variety of books before 1923, the most popular words used in television from 1990 up until 2006, and the most popular baby names. I assigned rather arbitrary weights to normalize all three sources and the frequencies, then

combined them into one giant list. I found the data was not ideally clean (for example, cd is a fairly popular word as is aaaaa or helloooo). In general though, I found the word lists sufficed. Most of these unclean words came from the television dataset. Removing the television dataset however, would then lead to older words (i.e. words that were more popular in the 1800s and 1900s such as thee or thou), to have an excessive weight. Text prediction strongly overemphasized these older words when the television dataset was taken out.

Preprocessing the Input Gesture Pattern

Once the dictionary was scrapped together, the next step was to generate the patterns and "paths" of both the words in the dictionary and any user-generated paths.

Word in Dictionary Path:

Both Swype and my system assign a pattern to each possible word in the dictionary, which is simply the path drawn to create a certain word. Swype here has two advantages with word implementation, that arguably could be viewed as disadvantages in user convenience. First is that because Swype has a visual output and visual feedback, the user both can and is forced to signal which letter he/she starts at. Second, the path drawn is far more rigid than with a gesture drawn in the air. The reason I argue these two differences can be a disadvantage for Swype is because the user must rely on visual feedback and because the user is fixed to a fairly rigid set of movements, which can be difficult to draw in haste. These differences however require subtle differences in implementation detail for my in-the-air project.

For similarities, both systems assign a pattern to a word then, comparing the user-generated pattern, picks the closest patterns. Swype relies considerably more on the

length of the stroke to get from one letter to another and it also has a reference point on when the stroke begins and ends. My in-the-air alternative does not have these advantages. To accommodate for this issue, I made length a less important parameter and instead focused on the angle (the angles of a gesture I've found, still tend to be fairly reliable even without a visual reference). To do this, I assigned an enormously extra weight to moving vertically and horizontally, so that a user who moved a stroke upwards when the stroke was leaning slightly downwards would be assigned an excessive weight. This distorts the length parameters, but makes the angle pattern recognition better. I assigned fixed positions to each letter in the keyboard, that with exception to adjusting this larger "vertical penalty" mirrored a regular keyboard.

With this adjustment, I assigned paths for each word. Another subtlety is with words where multiple letters are on one edge (so for example, the word "qwerty" is six letters but all fall on the same edge). For these patterns, I defined an arbitrary angle theta between two different letter edges and if the edge angle difference fell within this arbitrary theta, the two paths were combined. Lastly, for words with consecutive letters—for example the "ll" in "hello"—I treated them as one individual letter.

Recording and Transforming the User-Created Path:

To record the user-generated path, I used a Leap Motion controller to record the finger movements. The Leap Motion takes in x, y, and z coordinates. I flattened the path to two-dimensions simply by omitting the z-coordinate. The Leap Motion has a built-in API that can detect and track fingers. The user here holds one finger out and traces the path, then opens a second finger to indicate the end of the path (and the end of the word). Once every datapoint is recorded, the next step is to transform these set of data points

into the pattern associated with each word in the dictionary. To do this requires noting when the user changed direction. My methodology was as follows:

1) Sample every nth data point. If I sample every single data point and look for a change in direction, then the system will be prone to noise. If I sample too few data points, then changes in direction may be missed entirely. Eventually, by trial and error, I found a good number for n to be 14.
2) For every pair of data points now, I compared the direction taken. If the difference in angle was within a given angle (I chose 28 degrees), then it was deemed to be in the same direction. Otherwise, that marked a letter being picked by the user.
3) Given all these vertices which denote a change in direction, then we can now return this list of vertices.

Pattern Recognition Analysis

Given the user-generated path, which is comprised of a set of points, that path is then generated into a set of lengths and angles (stored as edges). The pattern recognition algorithm then takes the difference in angle (and then normalizes by dividing over 180), and takes the normalized length ratio of the larger and smaller lengths, then subtracts by 1. Scores are then generated based on similarity.

One huge optimizer and drawback for this algorithm is that it assumes that the path was drawn correctly—that is, that the number of edges in the path exactly parallel the number of edges in the desired word (that no mistake was made with drawing the path). This is good because it tremendously reduces the search time for finding the right

word. This is bad though because if an extra vertice caused by noise is included in the final path, it is guaranteed not to find the right answer (unlike what Swype does). Certainly, this is a very crude algorithm, but it works decently (with more time I would manually enter in a training set of data and assign parameters using machine learning). Given angle scores and length scores, weights are assigned to both scores (with tremendously more weight being assigned to the angle because lengths can get pretty unreliable without a visual interface). The final weights, after experimentation, were chosen to be angle = 0.75, and length = 0.25. Then, given the final shape weights, another weight accounting for the frequency of a word was applied. The exact frequency-of-word weight algorithm again is fairly arbitrary.

1) Given a word, get its frequency, then take the natural log of that frequency.
2) Normalize by dividing over 19 (a little more than the largest possible ln(frequency).
3) Weight the shape score (arbitrarily assigned 0.75 based on trial and error) and the weight of the frequency score (given a weight of 0.25).

The best seven results are then printed out.

Implementing the "In The Air" Part

I've discussed some of the challenges of implementing in-the-air, per-word gesturing versus touchscreen per-word gesturing, but I'll more succinctly highlight them here.

1) With a touchscreen, it's tremendously trivial to detect the beginning and end of a word. With the Leap Motion, that was considerably more difficult. Every idea I had had its pros and cons. First, I tried utilizing the z-direction of the Leap (that

is, using a tap). However, the Leap could not consistently enough identify the tap. Second, I tried using the other hand. However, as discussed in class, this method is highly taxing, and the Leap's two hand detection was not as ideal. Eventually, I elected to simply raise a second finger to signal the ending of a word, and the beginning of a new word by removing that finger. The con here is that, even for a movement this small it is exhausting. Worse, sometimes when the second finger is being withdrawn, the Leap interprets that finger moving as the path being drawn, which can make results unwieldy (by providing extra, undesired vertices in the final path). Frankly, the final solution I outline here is not satisfactory.

2) Lengths the user generates are far less reliable than with a touchscreen. The visual reference really does make a difference with regards to accurate lengths.
3) With no idea where the path starts by using in-the-air gestures, there are far more permutations. For example, using Swype, when the user starts the path by touching an arbitrary letter (say "g"), an anchor is established. In the air gesturing has no advantage. This results in issues. For example, the word "hello" has a similar path to "can't", "bang", or "jello." Especially for shorter words, because there are so many words with similar paths, this makes it extremely difficult for a more obscure word to have the best score.

Evaluation & Results

For this project, I gathered six friends as part of a quick experiment. After some practice, I gave them the prompt "as quickly and as accurately as possible, enter in the words 'the quick brown fox jumps over the lazy dog on Saturday today'" followed by the prompt "enter in the words 'hello my name is *name* would you like a seat?" I then

compared the results I acquired with a Swype-like interface's lab results, but using a stylus instead (Yatani, Koji, and Khai N. Truong. "An evaluation of stylus-based text entry methods on handheld devices in stationary and mobile settings." Proceedings of the 9th international conference on Human computer interaction with mobile devices and services. ACM, 2007). The stylus-based text entry reported speeds of 25 words a minute, and an error rate of 2%. The Swype reports speeds of 40 words a minute with roughly a 6% error rate. Our results were considerably faster (as expected because the movement has a much larger range of acceptable actions) but considerably less accurate (also expected because without a visual reference, there are many more possible words to skew results). Among the six subjects and myself, we were able to enter the following text at **51 words per minute**, but at the expense of a **28 percent error rate** (an error is when a word isn't one of the top five listed words). The speed is certainly desirable and I'd deem a fantastic performance. With longer scripts, when I tested it by myself, I was able to maintain similar speeds. However the accuracy makes such a project, for now, obsolete.

With more time on this project, I would like to apply machine learning in order to assign better weights to lengths, angles, and word frequencies, which I believe could generate better text-prediction results which could reduce errors to an acceptable rate. Furthermore, cleaner data in the word dictionary/frequencies would make results cleaner, as would a better method to detect the beginning and end point of a word. Overall though, I am pleased with the results.

Output

dyn-160-39-154-47:swype mzhong$ python run.py

Initialized
Connected
Ready for gesturing...
Press Enter to quit...

[(13.929731369018555, 170.67538452148438), (-62.65937042236328, 201.92623901367188), (45.02022171020508, 145.41563415527344), (19.646671295166016, 224.7376251220703)]
1: hello | 83
2: name | 83
3: hang | 83
4: many | 82
5: hand | 82
6: kelly | 80
7: gang | 80
[(11.328566551208496, 226.64590454101562), (29.12833023071289, 139.31857299804688), (-6.042792797088623, 242.02923583984375)]
1: the | 90
2: one | 85
3: eve | 84
4: Eve | 80
5: dvd | 78
6: thee | 78
7: thy | 78
[(-8.115067481994629, 220.7722930908203), (39.69572830200195, 124.73099517822266)]
1: I'll | 91
2: hmm | 87
3: em | 85
4: th | 84
5: hm | 83
6: hmmm | 83
7: ill | 83
[(-8.13471794128418, 216.50123596191406)]
1: I | 96
2: a | 95
3: s | 89
4: j | 88
5: mm | 88
6: t | 88
7: mmm | 87
[(-7.480860233306885, 213.91598510742188), (-7.041054725646973, 225.07432556152344), (14.836867332458496, 194.91644287109375)]
1: seem | 84
2: seen | 83
3: sell | 77
4: sex | 76
5: den | 76
6: dem | 76
7: gym | 76
[(11.864485740661621, 195.10562133789062), (-1.938869595527649, 161.63217163085938)]
1: on | 91
2: iv | 82
3: kn | 81
4: lm | 80
5: ih | 80
6: ob | 80
7: pm | 80

Documentation

populateWordFrequencies.py

```
import parseSources

parseSources.populateWordFrequencies()
```

parseSources.py

```
import urllib2, re

def readWebsite(url):
	site = url
	hdr = {'User-Agent': 'Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.11 (KHTML, like Gecko)
Chrome/23.0.1271.64 Safari/537.11',
	    'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8',
	    'Accept-Charset': 'ISO-8859-1,utf-8;q=0.7,*;q=0.3',
	    'Accept-Encoding': 'none',
	    'Accept-Language': 'en-US,en;q=0.8',
	    'Connection': 'keep-alive'}
	req = urllib2.Request(site, headers=hdr)
	page = urllib2.urlopen(req)
	return page.read()

def storeFrequencyCounts(listOfWords):
	dictionary = {}
	for i in listOfWords:
		if i[0] not in dictionary:
			dictionary[i[0]] = float(i[1])
	return dictionary

def populateWordFrequencies():
	def getWordsFromProjectGutenberg():
		def fetchWebsiteContents():
			wikipediaPageContents = []
			for i in (0,10000, 20000, 30000):
				begin = str(i+1)
				end = str(i+10000)
				url =
"http://en.wiktionary.org/wiki/Wiktionary:Frequency_lists/PG/2006/04/%s-%s" % (begin, end)
				wikipediaPageContents.append(readWebsite(url))
			return wikipediaPageContents
		def parsePGPages(contents):
			wordAndFreqs = []
			searchPattern1 = r"""<a href="/wiki/[a-zA-Z\-']+" title=\S+>([a-zA-Z\-']+)</a> = ([0-
9.]+)"""
			searchPattern2 = r"""<a href="/w/index\S+title=.+>([a-zA-Z\-']+)</a> = ([0-9.]+) """
			#avoid the colon
			#<a href="/wiki/of" title="of">of</a> = 33950064 <a href="/wiki/and"
			#<a href="/w/index.php?title=
			for page in contents:
				wordAndFreqs += re.findall(searchPattern1, page)
				wordAndFreqs += re.findall(searchPattern2, page)
			return wordAndFreqs
		wikipediaPageContents = fetchWebsiteContents()
		wordAndFreqs = parsePGPages(wikipediaPageContents)
```

```
		return storeFrequencyCounts(wordAndFreqs)

	def getWordsFromWiktionaryTV():
		def fetchWebsiteContents():
			wiktionaryPageContents = []
			i = 0
			while True:
				if i < 10000:
					begin = str(i+1)
					end = str(i+1000)
				elif i < 40000:
					begin = str(i+1)
					end = str(i+2000)
				elif i == 40000:
					begin = 40001
					end = 41284
				url = "http://en.wiktionary.org/wiki/Wiktionary:Frequency_lists/TV/2006/%s-
%s" % (begin, end)
				wiktionaryPageContents.append(readWebsite(url))
				if i < 10000:
					i += 1000
				elif i < 40000:
					i += 2000
				else:
					break
			return wiktionaryPageContents
		def parsePages(contents):
			wordAndFreqs = []
			searchPattern = r"""<td><a href=.+>([a-zA-Z\-']+)</a></td>\s<td>([0-9]+)</td>"""
			for page in contents:
				wordAndFreqs += re.findall(searchPattern, page)
			return wordAndFreqs
			#<tr>
			#<td>986</td>
			#<td><a href="/wiki/present" title="present">present</a></td>
			#<td>2313</td>
			#</tr>

		wiktionaryPageContents = fetchWebsiteContents()
		wordAndFreqs = parsePages(wiktionaryPageContents)
		return storeFrequencyCounts(wordAndFreqs)

	def getPopularBabyNames():
		babyNamesFile = open("Popular Baby Names.html", "r")
		searchPattern = r"""<td>([A-Za-z]+)</td>\s*<td>([0-9,]+)</td>"""
		wordAndFreqs = re.findall(searchPattern, babyNamesFile.read())
		for i in xrange(0,len(wordAndFreqs)):
			wordAndFreqs[i] = (wordAndFreqs[i][0],int((wordAndFreqs[i][1]).replace(",", "")))
		return storeFrequencyCounts(wordAndFreqs)
		#<tr align="right">
		#<td>2</td> <td>Mason</td><td>19,396</td>
		#<td>Isabella</td>
		#<td>19,745</td>
		#</tr>

	def combineData(allDicts):
		frequencyOfDicts = []

		#calibrate weights of dictionaries
		for dictionary in allDicts:
			print len(dictionary)
			frequencyOfDicts.append(sum(dictionary.values()))
```

```
                weightOfPG = 1
                weightOfWiktionary = 1
                weightOfBabyNames = .017
                allWordsFrequency = {}

                #populate dictionary with all words
                for i, dictionary in enumerate(allDicts):
                        for word in dictionary:
                                if i == 0: #if the dictionary is the PG one
                                        weight = weightOfPG
                                elif i == 1: #if Wiktionary
                                        weight = weightOfWiktionary
                                elif i == 2: #if baby names
                                        weight = weightOfBabyNames
                                weightedFrequency =
dictionary[word]/float(frequencyOfDicts[i])*weight*1000000000
                                if word in allWordsFrequency:
                                        allWordsFrequency[word] +=
dictionary[word]/float(frequencyOfDicts[i])*weight*1000000000
                                        allWordsFrequency[word] /= 2
                                else:
                                        allWordsFrequency[word] =
dictionary[word]/float(frequencyOfDicts[i])*weight*1000000000
                return allWordsFrequency
        def writeToFile(dictionary):
                fp = open("dictionary.txt", "w")
                for word in sorted(dictionary, key=dictionary.get, reverse=True):
                        if dictionary[word] < .1:
                                break
                        lineToWrite = word + "," + str(int(dictionary[word])) + "\n"
                        fp.write(lineToWrite)
                fp.close()


        dictionariesToCombine = []
        dictionariesToCombine.append(getWordsFromProjectGutenberg())
        dictionariesToCombine.append(getWordsFromWiktionaryTV())
        dictionariesToCombine.append(getPopularBabyNames())
        masterDict = combineData(dictionariesToCombine)
        writeToFile(masterDict)
```

run.py

```
import Leap, sys, Dictionary, PatternRecognizer
from Leap import CircleGesture, KeyTapGesture, ScreenTapGesture, SwipeGesture


class SwypeListener(Leap.Listener):
        def on_init(self, controller):
                self.wordToGesture = Dictionary.wordToGesture()
                self.wordToFrequency = Dictionary.wordToFrequency()
                self.path = []
                print "Initialized"

        def on_connect(self, controller):
                print "Connected"
                print "Ready for gesturing..."

        def on_disconnect(self, controller):
                print "Disconnected"
```

```
	def on_exit(self, controller):
		print "Exited"

	def on_frame(self, controller):
		#print self.path
		frame = controller.frame()
		if frame.hands.empty == False:
			fingers = frame.hands[0].fingers
			if len(fingers) > 1:
				if len(self.path) > 40:
					letterPointsList = PatternRecognizer.getLetterPoints(self.path)
					if len(letterPointsList) > 1:
						letterPointsList =
PatternRecognizer.eraseDuplicatePoints(letterPointsList)
					print letterPointsList
					self.path = []
					bestShapeScores =
PatternRecognizer.compareShapes(letterPointsList, self.wordToGesture)
					results = PatternRecognizer.getBestResults(bestShapeScores,
self.wordToFrequency)
					for i, scoreWord in enumerate(results):
						print str(i+1) + ": " + scoreWord[1] + " | " +
str(int(scoreWord[0]))
			elif len(fingers) == 1:
				tipObject = fingers[0].tip_position
				position = (tipObject.x, tipObject.y) #ignores the z-position
				self.path.append(position)

def main():
	wordToGesture = Dictionary.wordToGesture()

	listener = SwypeListener()
	controller = Leap.Controller()
	controller.add_listener(listener)

  # Keep this process running until Enter is pressed
	print "Press Enter to quit..."
	sys.stdin.readline()

  # Remove the listener when done
	controller.remove_listener(listener)


if __name__ == "__main__":
	main()
```

Dictionary.py

```
#generates the dictionaries word to frequency and word to gesture shape

import string, PatternRecognizer

def wordToFrequency():
	wordToFrequency = {}
	fp = open('dictionary.txt', 'r')
	for line in fp.readlines():
		(word, frequency) = line.split(',')
		wordToFrequency[word] = int(frequency)
	fp.close()
```

```
	return wordToFrequency

def wordToGesture():
	wordToGesture = {}
	fp = open('dictionary.txt', 'r')
	for line in fp.readlines():
		word = line.split(',')[0]
		alteredWord = word.translate(string.maketrans("","",),string.punctuation)
		pattern = PatternRecognizer.getPatternOfWord(alteredWord)
		wordLength = len(pattern)
		if len(pattern) > 1:
			pattern = PatternRecognizer.convertPatternToAngleAndLength(pattern)
		if wordLength in wordToGesture:
			wordToGesture[wordLength].append((word,pattern))
		else:
			wordToGesture[wordLength] = [(word,pattern)]
	return wordToGesture
```

PatternRecognizer.py

```
import Helpers, math, heapq

def getWhereMotionBegins(path):
	THRESHOLD = 2.5
	initial = path[0]
	i = 0
	for point in path:
		if Helpers.getDistance(initial, point) > THRESHOLD:
			break
		i += 1
	return i

def getLetterPoints(path):
	path = path[8:] #trim potential second finger distracting things
	path = path[getWhereMotionBegins(path):]
	path = path[:(len(path)-8)] #trim the end (where second finger comes into play)
	path = path[::14] # pick every nth frame

	letterPoints = [path[0]]
	if len(path) < 3:
		return letterPoints
	oldPoint = path[1]
	oldPath = (oldPoint, path[2])

	for i in xrange(2,len(path)):
		newPoint = path[i]
		newPath = (oldPoint, newPoint)
		if Helpers.isSamePathDirection(oldPath, newPath) == False:
			letterPoints.append(oldPoint)
		oldPath = newPath
		oldPoint = newPoint
	letterPoints.append(path[len(path)-1])
	return letterPoints

def eraseDuplicatePoints(letterPoints):
	THRESHOLD = 26
	listWithoutDuplicates = [letterPoints[0]]
	for i in xrange(1,len(letterPoints)):
		distance = Helpers.getDistance(letterPoints[i-1], letterPoints[i])
		if distance > THRESHOLD:
```

```
                        listWithoutDuplicates.append(letterPoints[i])
        return listWithoutDuplicates


def mapCharToPoint(char, letterPoints):
        ctp = {}
        ctp['q'] = (0,.8)
        ctp['w'] = (.11,.8)
        ctp['e'] = (.22,.8)
        ctp['r'] = (.33,.8)
        ctp['t'] = (.44,.8)
        ctp['y'] = (.55,.8)
        ctp['u'] = (.66,.8)
        ctp['i'] = (.77,.8)
        ctp['o'] = (.88,.8)
        ctp['p'] = (1,.8)
        ctp['a'] = (.05,.4)
        ctp['s'] = (.16,.4)
        ctp['d'] = (.27,.4)
        ctp['f'] = (.38,.4)
        ctp['g'] = (.49,.4)
        ctp['h'] = (.6,.4)
        ctp['j'] = (.71,.4)
        ctp['k'] = (.82,.4)
        ctp['l'] = (.93,.4)
        ctp['z'] = (.1,0)
        ctp['x'] = (.21,0)
        ctp['c'] = (.32,0)
        ctp['v'] = (.43,0)
        ctp['b'] = (.54,0)
        ctp['n'] = (.65,0)
        ctp['m'] = (.76,0)
        desiredPoint = ctp[char]
        if len(letterPoints) > 0:
                if letterPoints[-1] == desiredPoint:
                        letterPoints.pop(-1)
        if len(letterPoints) > 1:
                ANGLE_THRESHOLD = 18
                point1 = letterPoints[-2]
                point2 = letterPoints[-1]
                degree1 = Helpers.getAngle(point1,point2)
                degree2 = Helpers.getAngle(point2, desiredPoint)
                if math.fabs(degree1-degree2) < ANGLE_THRESHOLD:
                        letterPoints.pop(-1)
        return desiredPoint

def getPatternOfWord(word):
        letterPoints = []
        for char in word.lower():
                charPoint = mapCharToPoint(char, letterPoints)
                letterPoints.append(charPoint)
        return letterPoints

def convertPatternToAngleAndLength(letterPoints):
        patternInAngleAndLength = []
        oldPoint = letterPoints[0]
        for i in xrange(1,len(letterPoints)):
                newPoint = letterPoints[i]
                angle = Helpers.getAngle(oldPoint, newPoint)
                length = Helpers.getDistance(oldPoint, newPoint)
                patternInAngleAndLength.append((angle,length))
                oldPoint = newPoint
```

```
        return patternInAngleAndLength

def getAngleError(pattern1, pattern2):
        totalAngleError = 0
        for edge1, edge2 in zip(pattern1, pattern2):
                angle1 = edge1[0]
                angle2 = edge2[0]
                difference1 = math.fabs(angle1-angle2)
                difference2 = min(angle1,angle2)+360-max(angle1,angle2)
                totalAngleError += min(difference1,difference2)/180
        return totalAngleError/len(pattern1)

def getLengthError(pattern1, pattern2):
        totalLengthError = 0
        lengths1 = [x[1] for x in pattern1]
        lengths2 = [x[1] for x in pattern2]
        normalizedLengths1 = Helpers.normalizeLengths(lengths1)
        normalizedLengths2 = Helpers.normalizeLengths(lengths2)
        for length1, length2 in zip(normalizedLengths1, normalizedLengths2):
                totalLengthError += max(length1,length2)/min(length1,length2)-1
        return totalLengthError/len(pattern1)

def comparePatterns(pattern1, pattern2):
        #both patterns are guaranteed to have the same number of edges

        angleError = getAngleError(pattern1,pattern2)
        lengthError = getLengthError(pattern1, pattern2)

        error = 0.8*angleError+.2*lengthError
        return error

def compareShapes(letterPoints, wordToGesture):
        if len(letterPoints) == 1:
                bestScores = []
                for wordPattern in wordToGesture[len(letterPoints)]:
                        bestScores.append((1,wordPattern[0]))
                return bestScores
        patternInAngleAndLength = convertPatternToAngleAndLength(letterPoints)
        nBestScores = []
        for wordPattern in wordToGesture[len(letterPoints)]:
                (word, pattern) = (wordPattern[0],wordPattern[1])
                score = 1 - comparePatterns(patternInAngleAndLength, pattern)
                entry = (score,word)
                if len(nBestScores) < 30:
                        heapq.heappush(nBestScores, entry)
                else:
                        if nBestScores[0] < entry:
                                heapq.heappop(nBestScores)
                                heapq.heappush(nBestScores,entry)
        return nBestScores

def getBestResults(scoresAndWords, wordFrequencies):
        bestResults = []
        for scoreWord in scoresAndWords:
                shapeScore = scoreWord[0]
                freqScore = math.log(wordFrequencies[scoreWord[1]])/19
                totalScore = (0.75*shapeScore+.25*freqScore)*100
                entry = (totalScore,scoreWord[1])
                if len(bestResults) < 7:
                        heapq.heappush(bestResults,entry)
                else:
                        if bestResults[0] < entry:
```

```
                                        heapq.heappop(bestResults)
                                        heapq.heappush(bestResults,entry)
        bestResults = sorted(bestResults, key=lambda scoreWord: scoreWord[0])
        bestResults.reverse()
        return bestResults
```

Helpers.py

```
import math

def getDistance(point1, point2):
        distance = 0
        distance = math.sqrt((point1[0]-point2[0])**2 + (point1[1]-point2[1])**2)
        return distance

def getAngle(point1, point2):
        y = point2[1]-point1[1]
        x = point2[0]-point1[0]
        return math.degrees(math.atan2(y,x))

def normalizeLengths(lengths):
        normalizationConstant = sum(lengths)/len(lengths)
        normalizedLengths = [x/normalizationConstant for x in lengths]
        return normalizedLengths

def isSamePathDirection(path1, path2):
        MIN_MOVEMENT_DIST = 8.6
        ANGLE_CHANGE = 30
        path1Dist = getDistance(path1[0], path1[1])
        path2Dist = getDistance(path2[0], path2[1])
        if path1Dist < MIN_MOVEMENT_DIST or path2Dist < MIN_MOVEMENT_DIST:
                return True

        path1Angle = getAngle(path1[0], path1[1])
        path2Angle = getAngle(path2[0], path2[1])
        if math.fabs(path1Angle-path2Angle) <= ANGLE_CHANGE:
                return True
        else:
                return False
```