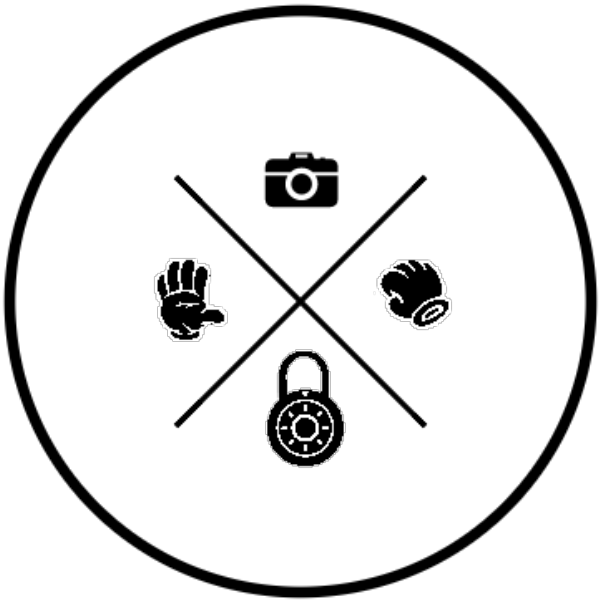
VISUAL COMBINATION LOCK



COMS W4735: Visual Interfaces to Computers

Assignment 1, due: 2/17/2015

Nina Baculinao, uni: nb2406

Introduction

The goal of the Visual Combination Lock is to recognize a sequence of visual images as a password. It builds off the basic assignment of having to identify the combination of a fist in the center of a table and a splayed palm in the corner. It adds one more token to the password, a splayed palm moved to another corner of the image. From there, it builds the vocabulary to include recognition for two, three and four upheld fingers. Finally, it uses this foundation of gesture recognition to build a quasi real time system of password verification of capturing a series of 3 pictures 5 second intervals through a webcam and identifying if the user has placed their hands in the predetermined sequence of locations and poses.

Domain Engineering

Imagery in this project was captured in the first half by an iPhone 6 and the second half in the built-in web camera of a MacBook Air. Lighting was ambient above, directed at the ceiling rather than directly on the photo station. Flash was used during the iPhone session. For both sessions, nonglossy black tshirt cloth was used to provide maximal contrast against the lighter skin of my hand. During the iPhone session, the black cloth was draped over a tall square stool and pictures were taken from a bird's eye view. For the webcam sessions, black tshirts were hung from the bookshelves and covered the back of the couch, in order to best simulate the controlled first environment that my visual recognition algorithms were trained on. In addition, during all photo sessions I wore a black sweater with long sleeves, to cover my wrists and make it easier to detect my hand. The hand was the main subject and intended captive of every single photo, and therefore I made every effort to minimize all distractions in this domain engineering step. At times, I roll my sleeves up, and in the case of the fist placed in the center of the image, this could be the cause of some issues, as we will explore later.

Other hardware specifications of this project are as follows:

- iPhone 6, v8.1.2
- MacBook Air 11 inch running on OSX Yosemite
- Python Standard Library 2.7.9
- OpenCV with a Python binding, v2.4.10.1
- NumPy 1.9.1

In total, I took 28 photos on the iPhone, recycling some for certain sequences and patterns. As for the second half of the project with a webcamera, there were many unsuccessful shots where I would have to cancel the program as a piece of wall or light background would peek into the image and contaminate the controlled environment. My estimate is that I took over 200 photos, as there are 70 diretories containing 3-photo snap sessions, but maybe only 50% of them were used for analysis. Therefore what is included in this report is but a small sample of the test results, successes and failures that were part of parcel of this project.

Photos were stored in JPG format as is standard with the iPhone. When I was writing the program to capture webcam pictures, OpenCV's image capture conveniently included the choice of format. I chose to keep them in JPG format to maintain consistency with the first half of the project and reusability of the code. My algorithms generated a fair amount of processed images, all stored in PNG format, so that if they were in the same folder, I could direct my program to only use JPG as input. By having this JPG/PNG delineation, I was able to let my program handle what went in and out, and simply pass folders as directories for the program to parse for sequences.

Data Reduction

In order to find a way to manipulate the images to get a good binarization of my hand, I delved into OpenCV's helpful tutorials. Data was reduced in a series of steps.

RESIZE: First, I resized the image to 300 x 300 pixels the default 2440 x 2440 pixels the iPhone saves images as was unnecessarily large.

> **ROTATE:** I then used a rotate method and saved over the files rather than save a copy of the image with a different tag like I did in the other steps because half the photos had correct rotation, whereas some were upside down and so on. This way, I could also eliminate rotate from the main method of my visual lock when I moved on to the webcam step.

GRAYSCALE: Next, I converted the color image to grayscale. The reason why I didn't directly use openCV to load the image as a grayscale image is because when my program runs it flashes between every newly generated image so the user can easily see how the data of an image is processed and reduced. In fact, every time I switch an image frame I make it stay for 1000 milliseconds, which is just enough to get a good look before it transitions to the next frame, and fast enough that it's not boring. Vital to that was showing them their original image, albeit resized to 300x300 so it could clearly fit in any computer screen.

> Note: at one point I inverted the colors by subtracting every element of the image matrix by 255 because I thought it made sense to have the hand as a black mass on a white background. This caused problems with contours later, so I removed the inversion step.
>
> Since my domain engineering was so austere, toying with color values to detect skin wasn't necessary so all these methods were pretty simple at this point.

BINARY THRESHOLD: Following these steps, I make the image a binary one, that is, black and white, where black represents background and white represents skin. This was done by finding Otsu's threshold value, andusing median blurring, which I chose over Gaussian blur because median blur is good for small spotty noise, and I noticed that some dog and cat hair along with dust and lint were showing up on the black cloth in my images.

MORPHOLOGICAL CLOSING: After that, I applied morphological closing to close the holes with a small kernel size or 5 to match my small iamge. This was a step I really debated about invoking in the main, because it hardly performed any smoothing for my black and white image, which was already quite smooth because of the strict domain engineering, and later on I realized my hesitations had root as the morphological closing would sometimes close the smaller gaps between less ambitiously splayed fingers, therefore disrupting the finger count of my contour count, which I'm getting to in a bit.

> Another method I toyed with but ended up not including in my main was Canny edge detection. It automatically performed gaussian filter, intesity gradient, suppression, hysteresis thresholding all at once according to the openCV website, but this seemed like it was overlapping with steps I was already taken. Notably, Canny method is used for edge detection, basically creating an outline of the hand instead of a solid fill, and while I toyed with the idea of using running an edgecrawler along the edge Canny produced, this seemed to overlap with the counter reader after the binarization.

CONTOURS and CONVEX HULL: Now this is where the the main analysis happens. Up until this point, it was just simple image manipulation, which could probably be accomplished with a simple library like PIL, which I considered using before finding out there were so many more methods, especially related to real-time processing, in the openCV library. In my contour reader method, I found the contours and convex hull. I actually encountered several errors coding this section, and had a fun time looking at the transformating NumPy matrices these images were made of, trying to figure out the source of the problem. One, I mentioned briefly earlier, was the unexpected values being produced by inverted colors. Another source of trouble was that I was unaware the findContours method in the openCV library would modify the source image. After I addressed these problems and stored the original image in a temp variable and returning it later to its original state after these countour-related transformations, everything worked pretty well.

I can talk about this step by step, although it's pretty visible in the commentary of my code. We begin by storing image (a Numpy matrix, not a picture, as I was often reminded by error warnings) in a temp variable. Then we use findContours on the image to find both the contours and the hiarchy. I store the first array in the contours in cnt. From then on, we can look at cnt and pass it into the convexHull method to get back indices of the contour points that represent the hull and the defects in the image. In this case, the hull is like what it sounds, the part sticking out, the fingers, while the convex "defects" are the in-between webbing of our fingers also occasionally called interdigitalis. After finding the hull and the defects, which unbeknowst to past me had been affecting the image array, I revert back to my original image with the temp variable. After that, I use openCv's draw function to draw the contours, and I also colorize the image so that these drawings can be done in color.

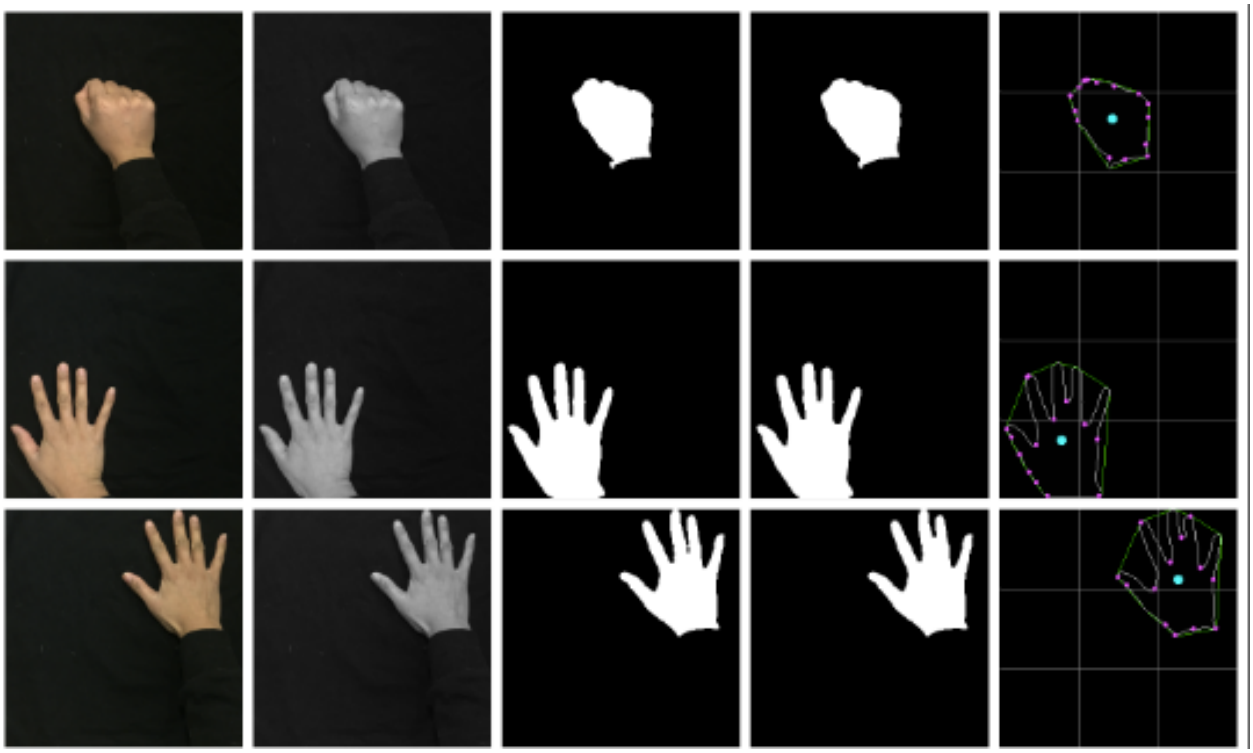
Key to detecting finger gestures is the information stored in the defects.shape[0], which contains four arrays. These arrays include the starting points, end points, and farthest points of the hand shape (which are indices of the contour array rather than pixel values), as well as the approximate distance to the fartests point. These four arrays are respectively called s,e,f, and d. By trial and error, and lots of print statements, I found that 6000 was a good bounding value to set – any distance from the farthest point greater than 6000 could be seen as the distance of a convex defect (interdigitalis) from the hull (tip of the finger).

After that, it's a simple matter to find the center of mass of the hand shape using openCV's moments matrix. Then, calling on the grammar I defined in another method, I was able to classify the hand gestures based on their number of convex defects/interdigitalis, and able to find the location of the hand based on the x and y coordinates of its moments matrix.

ONE MORE POINT: USER FRIENDLINESS: In order to make the process of data reduction and transformations visible to any user or anyone who is curious, I made all the images appear in a small window in an evolving succession as the program tries to make sense of the image and determine whether it follows the password sequence. In addition, I also add visual grid lines to show the different regions that a hand gesture can be located within a square image and how the computer determines whether a centroid belongs to a particular quadrant. While it's immediately obvious to the human eye oftentimes what hand gesture is being thrust up before their eyes, humans aren't always exact with their spatial reasoning so I hoped this ruler guidance would help.

To see the password sequence — look down

And for the data reduction sequence — loook right



Users of the system will just see one image at a time – transforming before their eyes!

Parsing and Performance

GRAMMAR: As I mentioned above in my discussion of convex defects, the number found within the noise of numbers that make up an image array is parsed and interpreted as a series of semantic symbols within the grammar of my program. Here is what each number found represents. When you combine these 6 (+1 unknown) possible gesture symbols with the 7 (+ unknown) possible screen locations, you end up with a decent vocabulary of 42 meaningful symbols that you can set as our 3 token password. So it s a language of not only gesture and pose, but location.

While the program can read all these symbols, it only has 5 possible responses, as they are mutually exclusive and cannot be combined together like the gesture-location syntax.

```
   denial = 'Entered wrong password combination. Access denied.'
    admittance = 'Thank you for entering your password
combination. Access granted.'
    confusion = 'There is an unknown value in your input. Access
denied. Try again.'
    overload = 'There was a surplus of tokens. Access denied.
Try again.'
    underflow = 'There were not enough tokens. Access denied.
Try again.'
```

PARSING: TWO PARTS

IDENTIFYING GESTURE: As was covered in relative detail above, the number of contour defects the program finds is generally 1 more than the number of fingers being held up, since there are n-1 gaps between n fingers. Therefore, after it has counted a number of defects that are a significant distance from the farthest point (in this case 6000) then we have a count that can be used to classify the gesture.

```
def classify(num):
    if num is 0:
        return 'Fist'
    if num is 1:
        return 'Two'
    if num is 2:
        return 'Three'
    if num is 3:
        return 'Four'
    if num is 4:
        return 'Five'
    else:
        return 'Unknown'
```

DETERMINING LOCATION: The location of the gesture was discovered, as mentioned above, by first calculating the centroid of the hand, then comparing that x,y value to certain pre-defined boundary points in the image. In this case, I chose to split the square image into equal sized regions.

v1		v2	
(0,0)	(1,0)	(2,0)	h1
(0,1)	p1 p2 (1,1) p3 p4	(2,1)	h2
(0,2)	(1,2)	(2,2)	

Top-left	Top-Center	Top-right
Unknown	Center	Unknown
Bottom-right	Bottom-Center	Bottom-left

So to find the location, we take the x and y values of the centroid and see if x is between p1 and p2's x –values (center column) and if y is between h1 and h2's y-values – if both conditions are true, that mean it's located in the center, or if we pretended each box is 1 pixel, this would be locted in 1,1. It then checks if it's on top, if so, is left of p1 or right of p2. Same goes for the bottom, if it's in the bottom left corner, or bottom right corner, or if it's in the central column. I left two regions labeled 'unknown' as additional vocabulary for the position to rcognize.

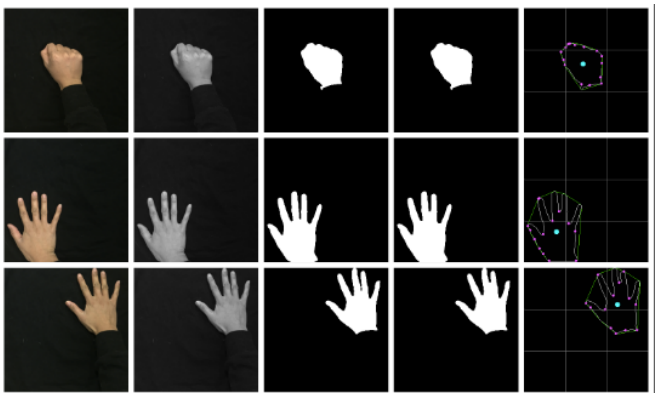
ALTERNATIVE TAGGING: Another way I thought of implementing the grammar would have been to encode the hand-based passwords into a number of number strings. We could create a Python dictionary of keys and values { Fist : 0, Two : 2, Three : 3, Four : 4, Five : 5, Center : 11, Top-left : 00 , Top-center : 10, Top-right: 20, Bottom-left : 02 , Bottom-center : 12, Bottom-right : 22 }

Then the default password could be represented as: 011502520, instead of 'Fist' 'Center' 'Five' 'Bottom-left' 'Five' 'Top-right.' Much shorter and more efficient that the current list of strings
.

Sample Runs with Images Taken by Iphone

Correct password: 1. Fist, Center 2. Five, Bottom-Left 3. Five, Top-right

Success 1/7



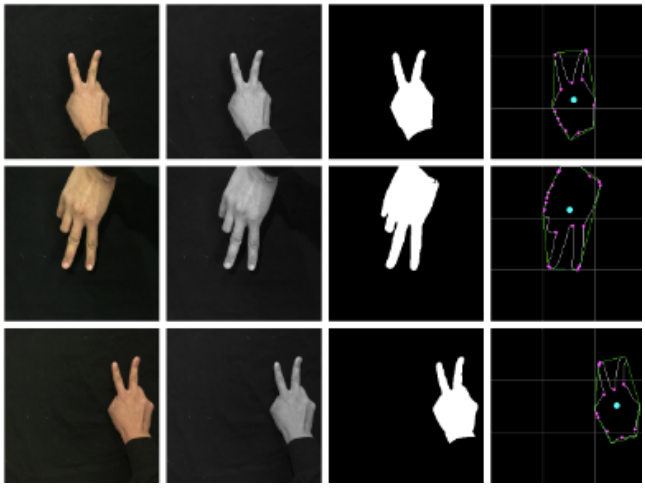
```
Sequence: 1
Expected: Accepted
Result:   Accepted
```

```
→  visual-interfaces git:(master)
✗ python visuallock.py seq1
./seq1/fist_11.JPG >> Fist, Center
./seq1/five_02.JPG >> Five, Bottom-left
./seq1/five_20.JPG >> Five, Top-right
Thank you for entering your password
combination. Access granted.
```

From up to down you can see the password sequence. From left to right you can see the data reduction that takes place, and and how we as humans can interpret the password and how the visual lock program interprets the image and extracts information.

Also from this shrunken viewpoint - the difference between the 3rd and 4th column, respectively the image after undergoing Otsu's binarization and after experiencing morphological closing, is so slight that I have decided to remove the closed images unless they contributed in some way to an unexpected result worth noting.

Successful Rejection 1/7



```
Sequence: 2
Expected: Denied
Result:   Denied
```

```
→  visual-interfaces git:(master) ✗
python visuallock.py seq2
./seq2/two_11.JPG >> Two, Center
./seq2/two_11vflip.JPG >> Two, Top-center
./seq2/two_21.JPG >> Two, Unknown
There is an unknown value in your input.
Access denied. Try again.
```

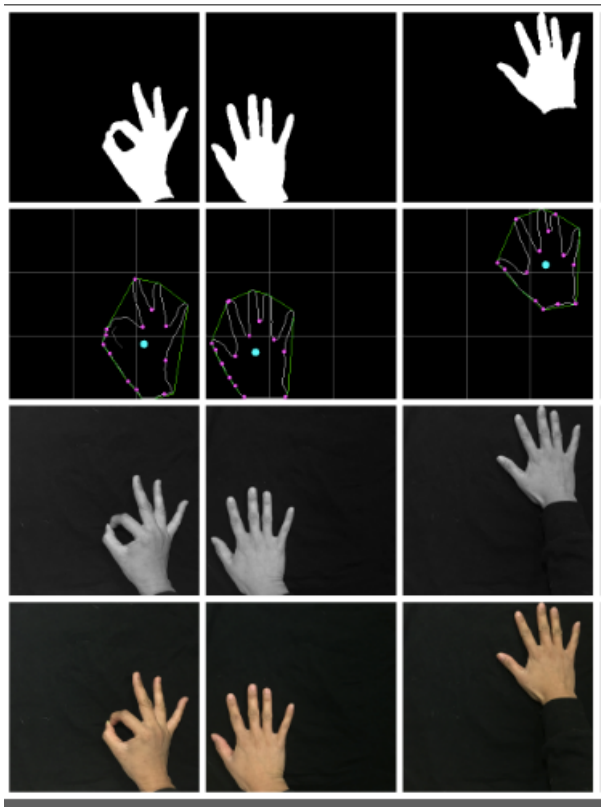
Note that the unknown was not the number, as two fingers were correctly identified for all the poses, including the upside down two. As I mentioned before, I made a decision not to include the middle-left and middle-right sectors of the grid, so the program is correct in identifiying the 2 finger pose as a placed in an unknown location.

CORRECT 3/7

Sequence: 3
Expected: Denied
Result: Denied

```
→  visual-interfaces git:(master) ✗ python
visuallock.py seq3
./seq3/a_fist_02.JPG >> Fist, Bottom-left
./seq3/fist_11.JPG >> Fist, Center
./seq3/two_21.JPG >> Two, Unknown
There is an unknown value in your input. Access denied.
Try again.
```

Unknown locations are properly identified though it's the in between corners/boundaries of quadrants that cause problems as we shall see later.



CORRECT 3.5/7

FOR INCORRECT REASONS 1/4

Sequence: 4
Expected: Denied
Result: Denied

```
→  visual-interfaces git:(master) ✗
python visuallock.py seq4
./seq4/a_ok_22.JPG >> Four, Bottom-right
./seq4/b_five_02.JPG >> Five, Bottom-left
./seq4/c_five_20.JPG >> Five, Top-right
Entered wrong password combination. Access
denied.
```

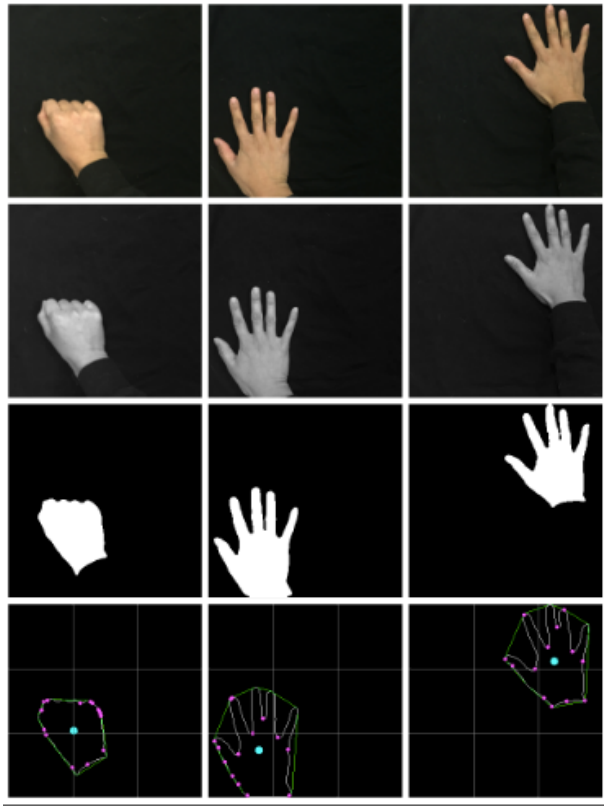
The okay symbol is incorrectly identified as a four, which makes sense when you simply count the number of convex defects. A more robust grammar system would look into the holes and shape of the fingers too.

CORRECT 4.5/7

Sequence: 5
Expected: Denied
Result: Denied

The grammar system correctly understands that semantic and conventions. The password needs 3 tokens only. Too many and access is not given.

```
→  visual-interfaces git:(master) ✗
python visuallock.py seq5
./seq5/c_five_02.JPG >> Five, Bottom-left
./seq5/d_five_20.JPG >> Five, Top-right
./seq5/fist_11.JPG >> Fist, Center
./seq5/two_11.JPG >> Two, Center
There was a surplus of tokens. Access
denied. Try again.
```



Similar to the sequence 5 example, the grammar sysem expects a certain number of tokens before it will even try to parse the password. Too many gestures and it know it's wrong.

CORRECT 7.5/7

FOR INCORRECT REASONS 2/4

```
Sequence: 8
Expected: Rejected
Result:   Rejected
```

Due to failure at the domain engineering step (I was wearing a gray hoodie instead of black) the binarization failed, and the center of mass set improperly. Nonetheless, the program was lucky, as I was trying to trick it anyway with a closed instead of spread palm. So it got the passing start gesture: "fist", even though it was actually a "five" and it got the wrong location "bottle-right" even though it was actually correctly starting at "center."

CORRECT 5.5/7

```
Sequence: 6
Expected: Accepted
Result:   Accepted
```

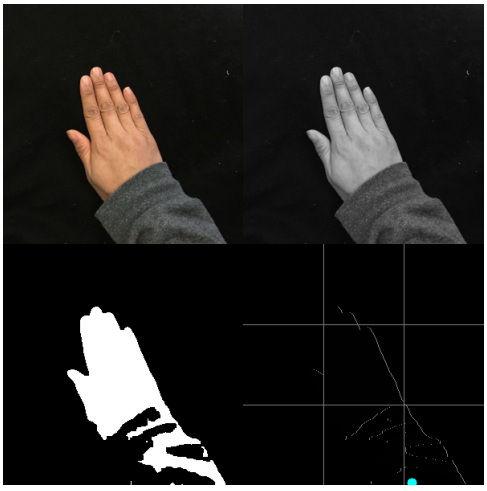
```
→  visual-interfaces git:(master) ✗
python visuallock.py seq6
./seq6/a-fist_1p5x2.JPG >> Fist, Center
./seq6/b_five_02.JPG >> Five, Bottom-left
./seq6/c_five_20.JPG >> Five, Top-right
Thank you for entering your password
combination. Access granted.
```

Correctly interprets a successful password log in, though it's a close call with the close distance between the ring finger and middle finger.

CORRECT 6.5/7

```
Sequence: 7
Expected: Rejected
Result:   Rejected
```

```
→  visual-interfaces git:(master) ✗
python visuallock.py seq7
./seq7/four_01.JPG >> Four, Unknown
There were not enough tokens. Access denied.
Try again.
```



```
→  visual-interfaces git:(master) ✗
python visuallock.py seq8
./seq8/a_image1.JPG >> Fist, Bottom-right
./seq8/b_five_02.JPG >> Five, Bottom-left
./seq8/c_five_20.JPG >> Five, Top-right
Entered wrong password combination. Access
denied.
```

CORRECT 8.5/7

Sequence: 9
Expected: Accepted
Result: Accepted

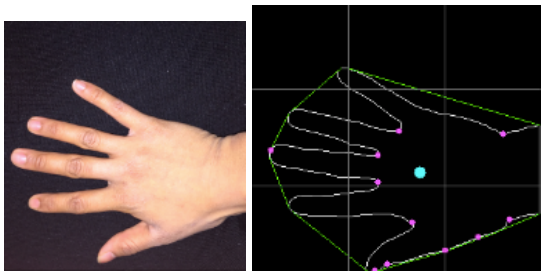
```
→  visual-interfaces git:(master) ✗ python
visuallock.py seq9
./seq9/a_fist_11.JPG >> Fist, Center
./seq9/b_image7.JPG >> Five, Bottom-left
./seq9/c_five_20.JPG >> Five, Top-right
Thank you for entering your password combination.
Access granted.
```

Even though the thumb was partially cut off the program still count the number of dips in between the webbings and identify the right gestures at the wrong locations.

CORRECT 9.5/7

Sequence: 11
Expected: Accepted
Result: Accepted

```
→  visual-interfaces git:(master) ✗ python
visuallock.py seq11
./seq11/a_fist_11.JPG >> Fist, Center
./seq11/b_image6.JPG >> Five, Center
./seq11/c_five_20.JPG >> Five, Top-right
Entered wrong password combination. Access denied
```



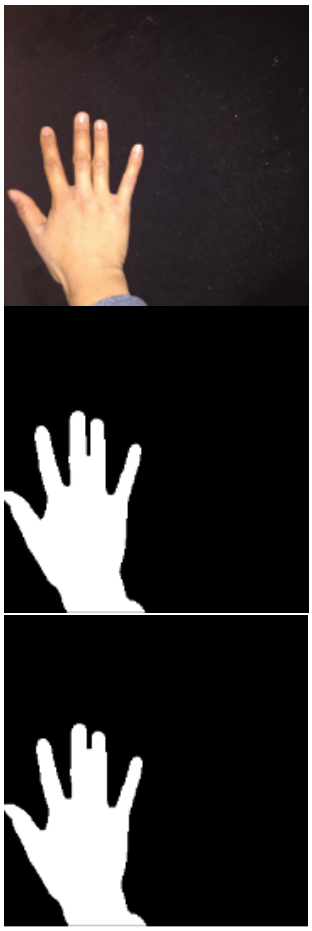
The visual lock interpreted the overly close hand just fine, despite the orientation, distance is relative so all four interstitial points were still located for this five pose and it was rejected for being in the wrong location.

INCORRECT 3/4

Sequence: 10
Expected: Accepted
Result: Accepted

```
→  visual-interfaces git:(master) ✗
python visuallock.py seq10
./seq10/a_fist_11.JPG >> Fist, Center
./seq10/B_image8.JPG >> Four, Bottom-left
./seq10/c_five_20.JPG >> Five, Top-right
Entered wrong password combination. Access
denied.
```

The unintentional culprit of this image was the close proximity of the ring and middle finger. The location was correct for the second token, but the finger count was wrong. As you can see, the morph made the gap even more imperceptible.



The successes and failures count continues…
after the creativity clarification!

Creativity Step

The default definition of this problem is the one given above: two images, one with a fist in the center and one with a "five!" in a corner. I just happened to add another "five" to the password because to me two tokens is not much of a password. For the creativity step, I decided to extend my algorithm and apply it to a quasi real-time gesture recognition interface that assesses whether users have input the correct hand-based password.

Rather than prompt the user pass a directory into the visual lock (which I hoped was already quite simple and better than asking for him/her to pass in individual arguments or look up some kind of numerical code listing values, but still might involve having to transfer data from one's phone and put it into a folder with three tokens) -- I decided to just let users take pictures from their webcam directly, at 5 second intervals with a countdown message in terminal and a small webcam screen open on the side.

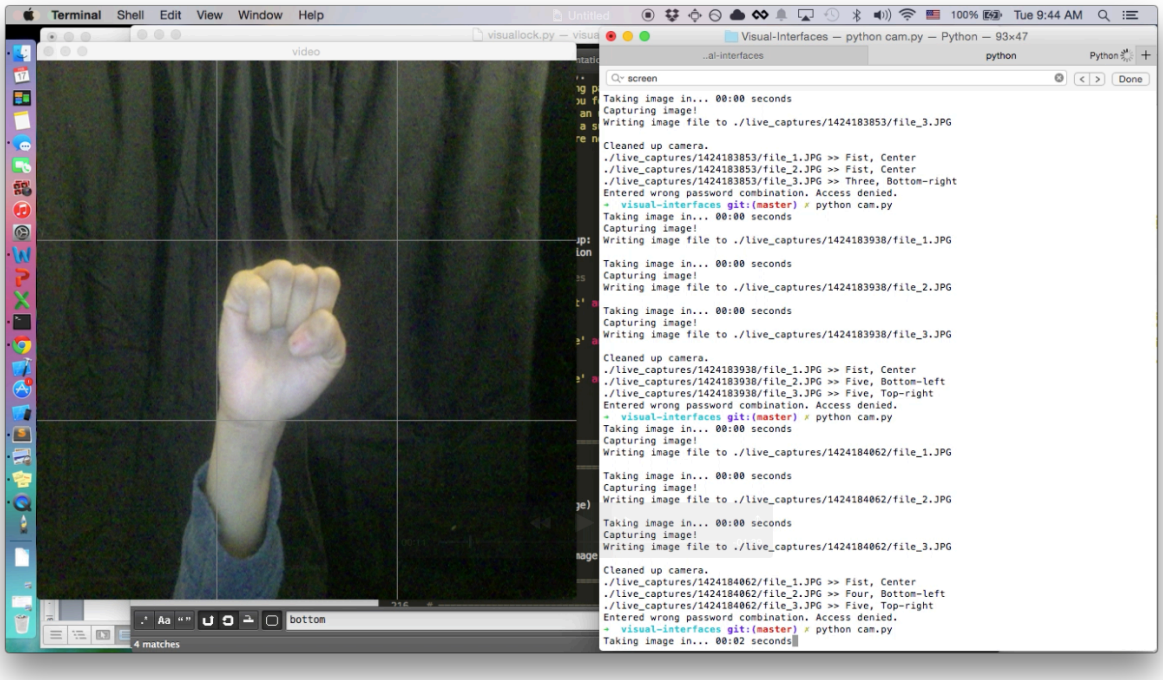
I created a square video screen to show us what the webcam is seeing for that purpose. It should work with any built in webcam or laptop (since the port is automatically set to 0) and it includes grid lines so users can better judge how to include their hands in the right relative locations. I chose a 640 x 640 view for the frame size, because even on my small laptop screen there was enough real estate to have the video and terminal open side by side, which was great because terminal contained all the user interaction messages and also gave users approval or rejection of their input password.

After the three photos in the sequence are captured, the system saves the three images in a unique subdirectory within './live-captures' – unique because it is based on the timestamp that the first image is stored with the other two joining it in the same folder so they are grouped together. Then, visuallock.py is automatically run on that directory, and analyzes those three files.

If I had more time, I would have loved to give users even more power to generate new gestures for the grammar, and to allow dynamic redefinition of the password and so on. I would also try to fix some of the salient problems of my current grammar system and definitions such as the morphology problem, being more attentive to slight cracks between fingers, identifying holes in the hand such as the OK sign, and adding more vocabulary beyond simple finger counting that currently exists in the system, which makes no differentiation between left, right hands, which fingers are extended, whether it's the forehand or back facing us, and so on. I would also have liked to use matplotlib to display images in sequence, and compare to the effect of just flashing through them over one another.

With the introduction of this real time element, came additional complications. I placed similar restrictions on the domain engineering in this portion of the project, requiring that the background of the image be a plain black cloth and the attached arm to the hand be covered by a black sleeve to blend it with the cloth. Generally, photos taken by my iPhone had more clarity and therefore there came an increase in "failures" of the system when these fuzzy webcam photos were generated. These do not even include cases where white walls would peek out of the black cloth, because in those cases half the time the program identified those white patches as skin and would return null when trying to find its contours and convex hull. The relative failures and successes of this system, built on the algorithm and gesture parser of original assignment, are documented below.

Software Execution

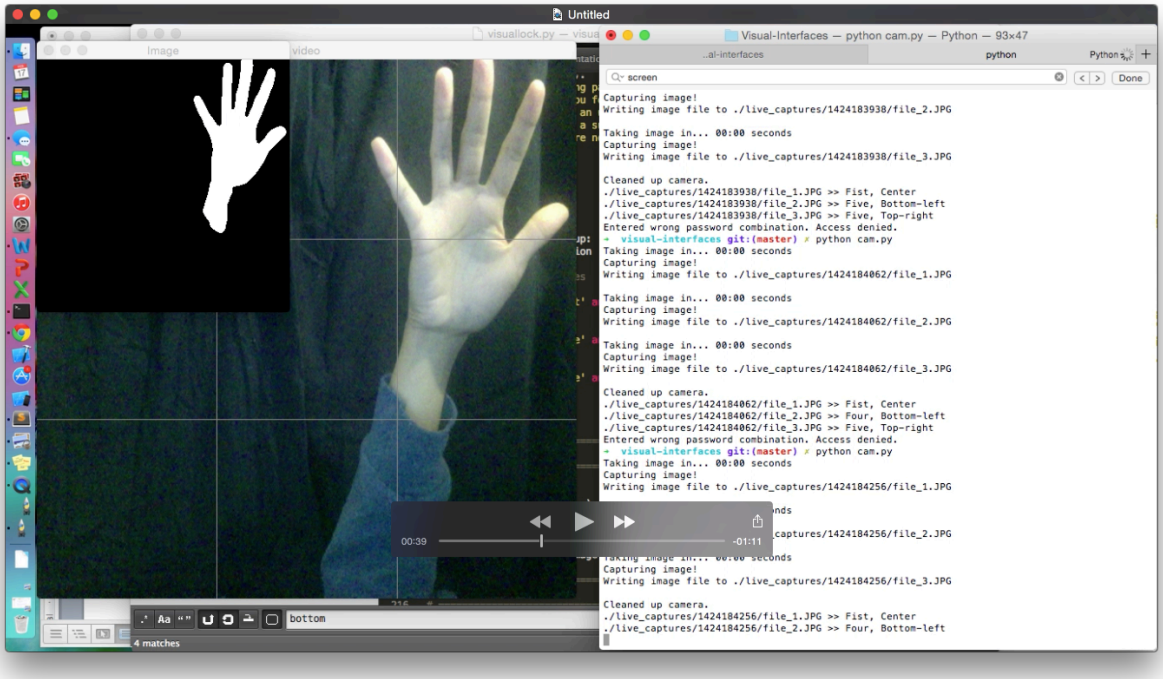


Note the grid lines and square format of the webcam to help users locate themselves within the image. If I had time, slight things I would change is the distortion of the original aspect ratio, and also the mirror flipping of the webcam, which was confusing at times.

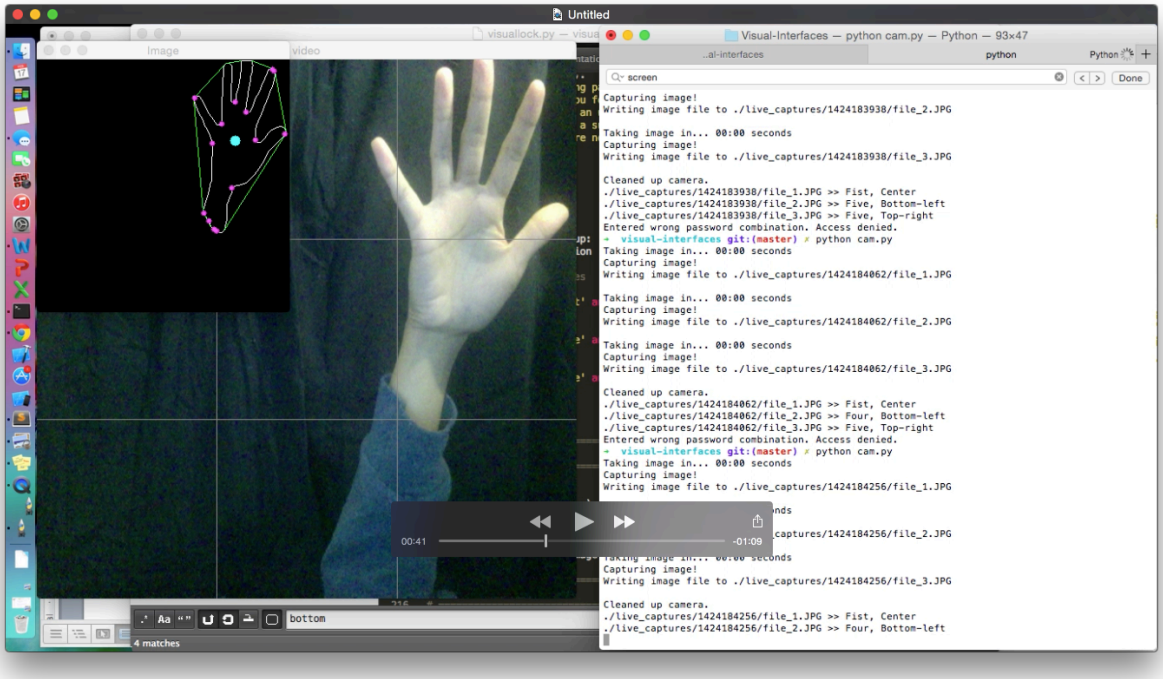
Terminal prints out the following message on the side:

```
Taking image in... 00:05 seconds # the time ticks down in place
Capturing image!                 # with a carriage return
Writing image file to ./live_captures/1424193267/file_1.JPG

Taking image in... 00:00 seconds
Capturing image!
Writing image file to ./live_captures/1424193267/file_2.JPG

Taking image in... 00:00 seconds
Capturing image!
Writing image file to ./live_captures/1424193267/file_3.JPG

Cleaned up camera.

./live_captures/1424193267/file_1.JPG >> # Gesture, location
./live_captures/1424193267/file_2.JPG >> # results follow
./live_captures/1424193267/file_3.JPG >> # here
```

```
# And then a message about your access level
```



After the video was done, the gesture analyzer program in visuallock.py would automatically run on the three newly captured images, checking to see if they followed the password sequence.

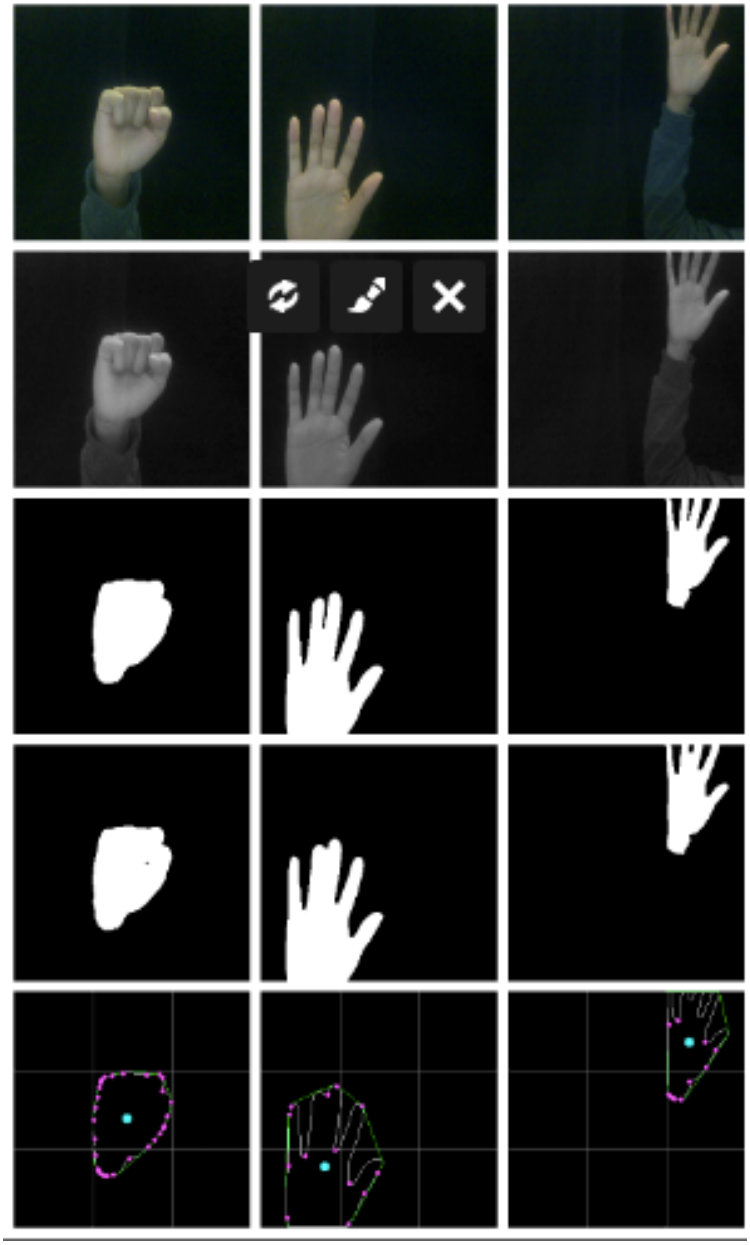


In the top left the contours are labeled and the blue dot shows the relative position of the hand to the nine quadrants in the grid.

Sample Runs with Images taken by WebCam

Correct password: 1. Fist, Center 2. Five, Bottom-Left 3. Five, Top-right

False Negative 4/4

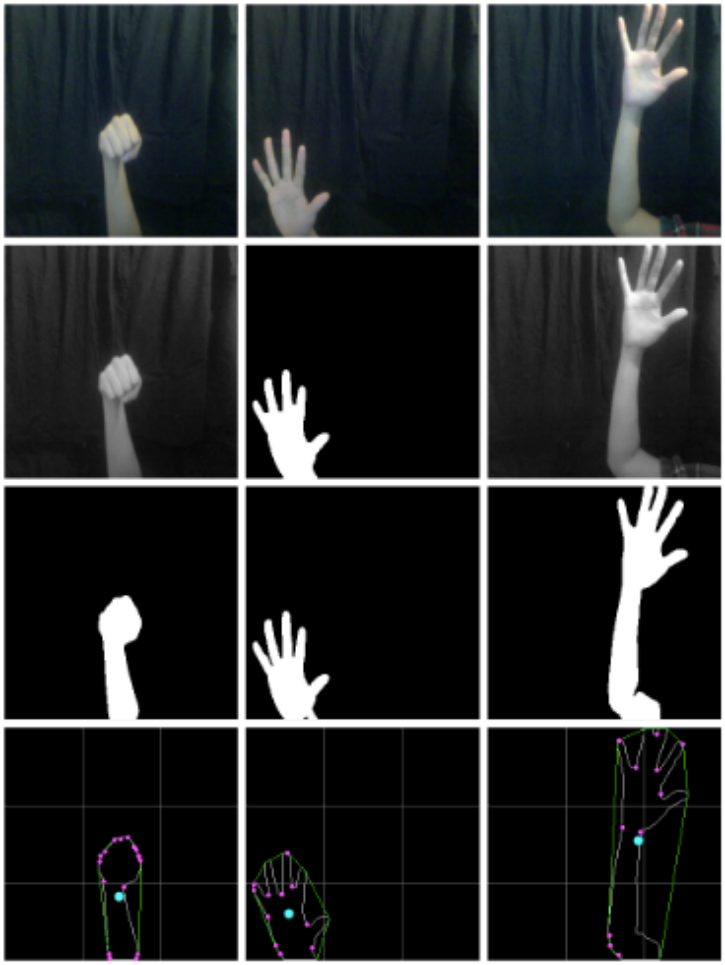


Expect: Accept
Actual: Reject

```
→  visual-interfaces
git:(master) ✗ python cam.py
Taking image in... 00:00 seconds
Capturing image!
Writing image file to
./live_captures/1424193267/file_1.JPG

Taking image in... 00:00 seconds
Capturing image!
Writing image file to
./live_captures/1424193267/file_2.JPG

Taking image in... 00:00 seconds
Capturing image!
Writing image file to
./live_captures/1424193267/file_3.JPG

Cleaned up camera.
./live_captures/1424193267/file_1.JPG
>> Fist, Center
./live_captures/1424193267/file_2.JPG
>> Four, Bottom-left
./live_captures/1424193267/file_3.JPG
>> Four, Center
Entered wrong password combination.
Access denied.
```

Here, the gap between the fingers was too small, and the side effect of morphological closing was such that the gap between the fingers closed and gave the computer the ide athat only four fingers were up on the bottom left. Moreover, the top right fingers were cut off, also skewed the results.

False Negative 5/4

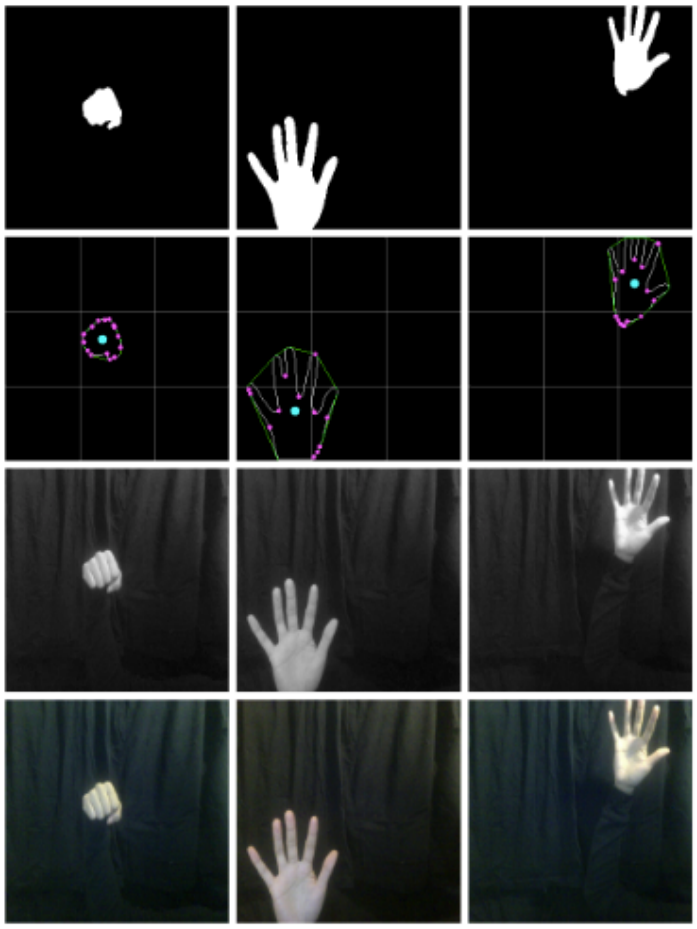


Expect: Accept
Actual: Reject

```
➜  visual-interfaces
git:(master) ✗ python cam.py
live_captures/1424193267
Taking image in... 00:00 seconds
Capturing image!
Writing image file to
./live_captures/1424193355/file_1.JPG

Taking image in... 00:00 seconds
Capturing image!
Writing image file to
./live_captures/1424193355/file_2.JPG

Taking image in... 00:00 seconds
Capturing image!
Writing image file to
./live_captures/1424193355/file_3.JPG

Cleaned up camera.

live_captures/1424193267
./live_captures/1424193267/file_1.JPG
>> Fist, Bottom-center
./live_captures/1424193267/file_2.JPG
>> Five, Bottom-left
./live_captures/1424193267/file_3.JPG
>> Five, Center
Entered wrong password combination.
Access denied.
```

Here, my sleeves were rolled up, so the center of mass was taken somewhere in my arm for the first and third image. The fist, which is meant to be in the center, ends up in center bottom, and last gesture ends ups up categorized in the center when it was really aiming for the top right.

A SUCCESS! 8.5/7

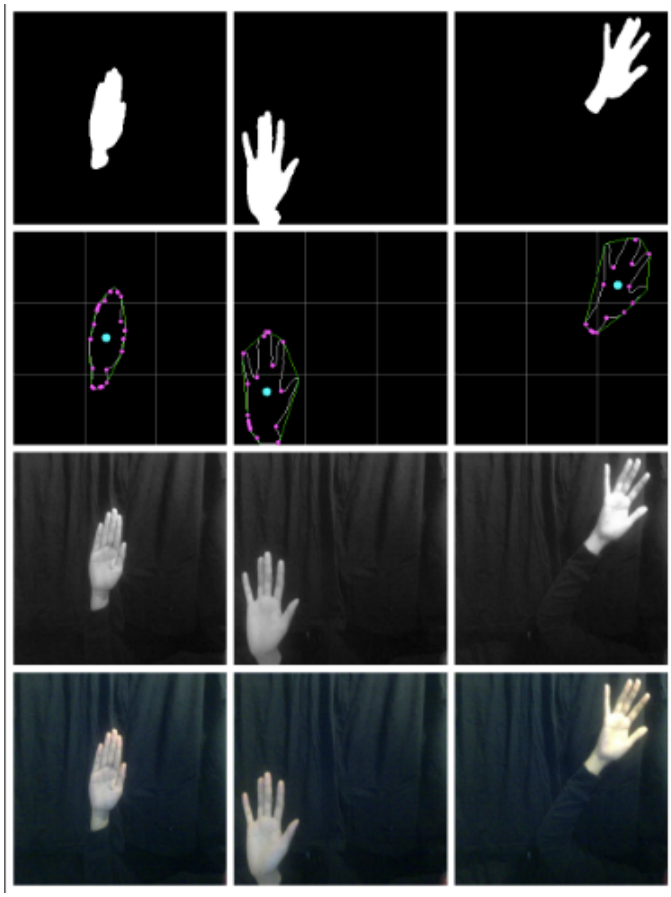


Expect: Accept
Actual: Accept

```
➜  visual-interfaces
git:(master) ✗ python cam.py
Taking image in... 00:00 seconds
Capturing image!
Writing image file to
./live_captures/1424193629/file_1.JPG

Taking image in... 00:00 seconds
Capturing image!
Writing image file to
./live_captures/1424193629/file_2.JPG

Taking image in... 00:00 seconds
Capturing image!
Writing image file to
./live_captures/1424193629/file_3.JPG

Cleaned up camera.
live_captures/1424193629
./live_captures/1424193629/file_1.JPG
>> Fist, Center
./live_captures/1424193629/file_2.JPG
>> Five, Bottom-left
./live_captures/1424193629/file_3.JPG
>> Five, Top-right
Thank you for entering your password
combination. Access granted.
```

Part of the challenge appears to be locating your hand as far as possible from the webcam. What also mad eit difficult was my small 11" monitor, which made my hands feel like they were taking up half the screen most of the time

Success, wrong reasons 9/7? 6/4?

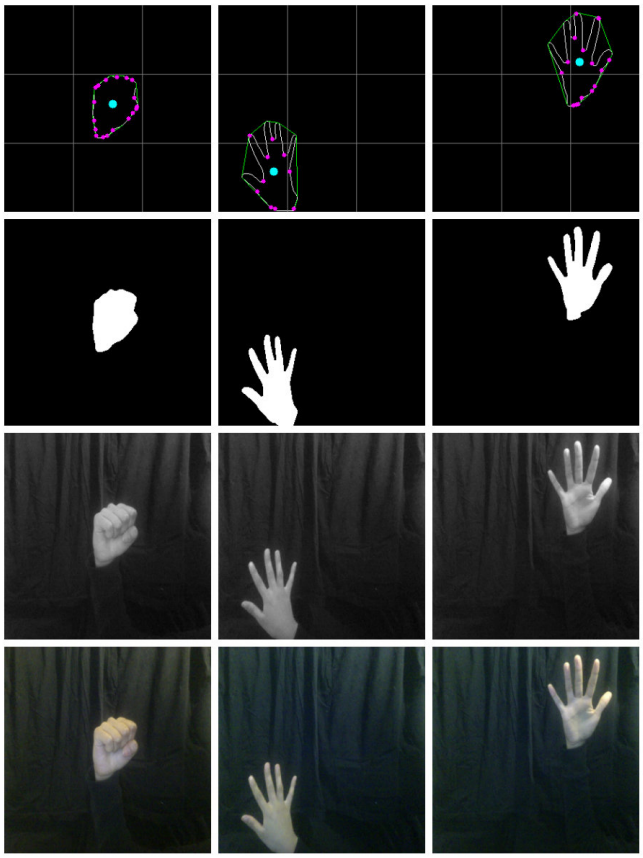


Expect: Reject
Actual: Reject

```
→  visual-interfaces
git:(master) ✗ python cam.py

Taking image in... 00:00 seconds
Capturing image!
Writing image file to
./live_captures/1424193933/file_1.JPG

Taking image in... 00:00 seconds
Capturing image!
Writing image file to
./live_captures/1424193933/file_2.JPG

Taking image in... 00:00 seconds
Capturing image!
Writing image file to
./live_captures/1424193933/file_3.JPG

Cleaned up camera.
./live_captures/1424193933/file_1.JPG
>> Fist, Center
./live_captures/1424193933/file_2.JPG
>> Four, Bottom-left
./live_captures/1424193933/file_3.JPG
>> Four, Top-right
Entered wrong password combination.
Access denied.
```

Here, the camera mis-identified my open palm in the first shot because all my fingers were close together. Something like this would be better classified as a karate chop. In the future, I would like to use more then the convex defects between the fingers to figure out what hand is being held up, but it's been suprisingly successful so far, possibly because of lucky streaks like this. Somehow, even while incorrectly clasifying all three gestures, it reached the correct result. Note that it classified both my fives as wrong, although all five fingers are obviously up to a human eye.

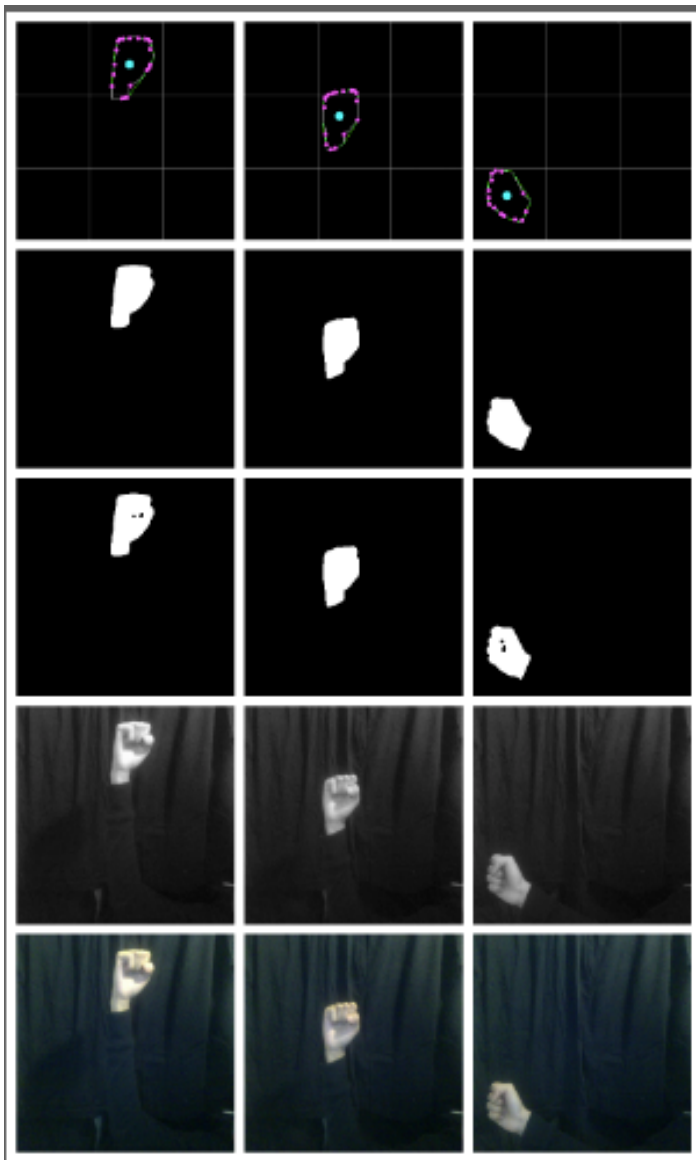
Success 10/7



Expect: Accept
Actual: Accept

```
➜  visual-interfaces
git:(master) ✗ python cam.py
Taking image in... 00:00 seconds
Capturing image!
Writing image file to
./live_captures/1424194129/file_1.JPG

Taking image in... 00:00 seconds
Capturing image!
Writing image file to
./live_captures/1424194129/file_2.JPG

Taking image in... 00:00 seconds
Capturing image!
Writing image file to
./live_captures/1424194129/file_3.JPG

Cleaned up camera.
./live_captures/1424194129/file_1.JPG
>> Fist, Center
./live_captures/1424194129/file_2.JPG
>> Five, Center
./live_captures/1424194129/file_3.JPG
>> Five, Top-right
Thank you for entering your password
combination. Access granted.
```

Success 11/7

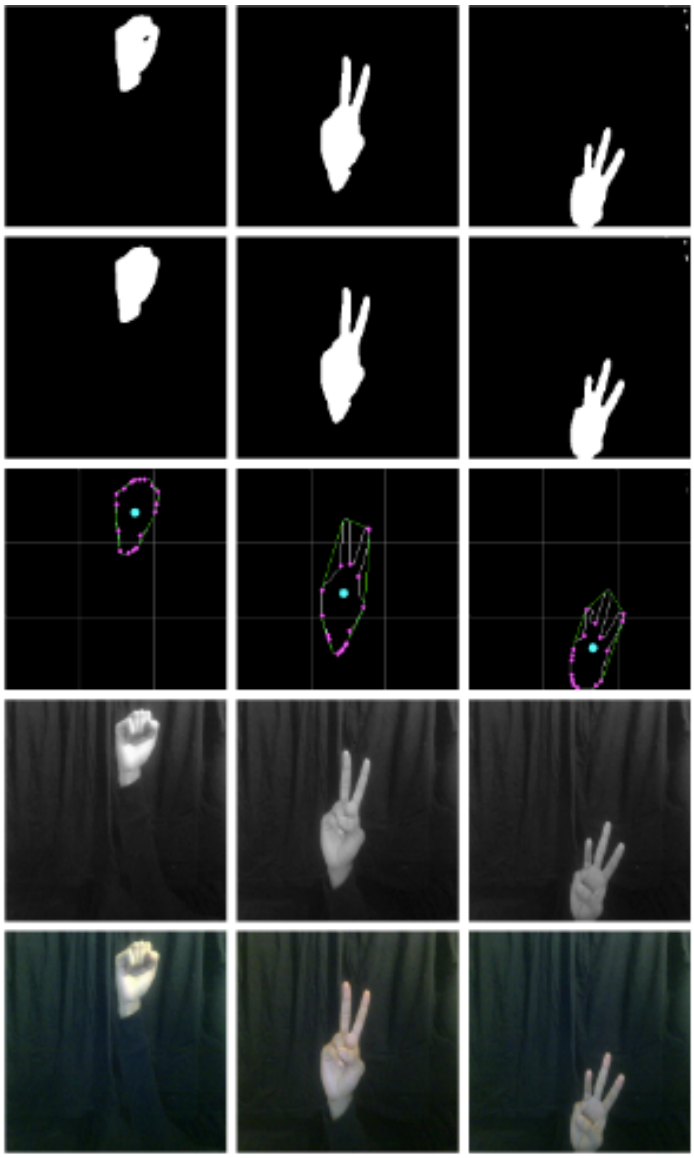


Expect: Reject
Actual: Reject

```
→  visual-interfaces
git:(master) ✗ python cam.py
Taking image in... 00:00 seconds
Capturing image!
Writing image file to
./live_captures/1424194920/file_1.JPG

Taking image in... 00:00 seconds
Capturing image!
Writing image file to
./live_captures/1424194920/file_2.JPG

Taking image in... 00:00 seconds
Capturing image!
Writing image file to
./live_captures/1424194920/file_3.JPG

Cleaned up camera.
./live_captures/1424194920/file_1.JPG
>> Fist, Top-center
./live_captures/1424194920/file_2.JPG
>> Fist, Center
./live_captures/1424194920/file_3.JPG
>> Fist, Bottom-left
Entered wrong password combination.
Access denied.
```

Correctly classifies the location of my fist. Enough said. It's easy to find the center of mass for the fist.

Success! As Easy as 1 2 3 … 12/7

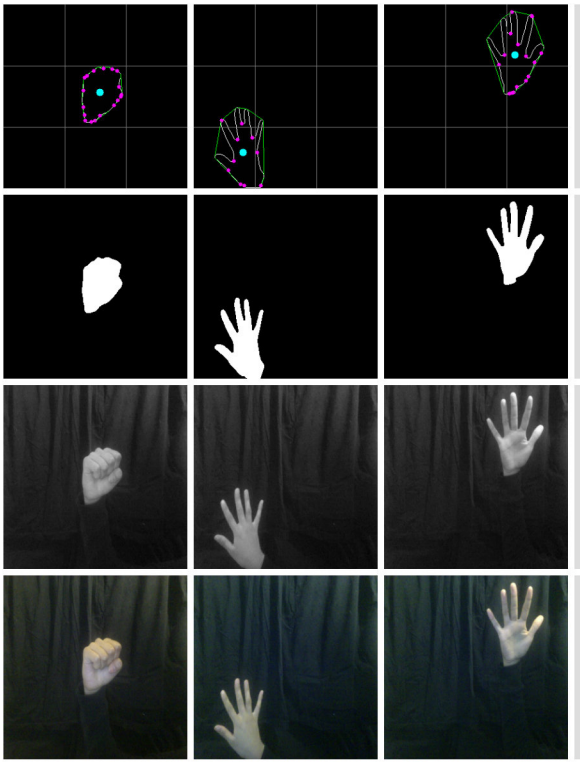


Expected: Reject
Actual: Reject

```
→  visual-interfaces
git:(master) ✗ python cam.py
Taking image in... 00:00 seconds
Capturing image!
Writing image file to
./live_captures/1424195376/file_1.JPG

Taking image in... 00:00 seconds
Capturing image!
Writing image file to
./live_captures/1424195376/file_2.JPG

Taking image in... 00:00 seconds
Capturing image!
Writing image file to
./live_captures/1424195376/file_3.JPG

Cleaned up camera.
./live_captures/1424195376/file_1.JPG
>> Fist, Top-center
./live_captures/1424195376/file_2.JPG
>> Two, Center
./live_captures/1424195376/file_3.JPG
>> Two, Bottom-center
Entered wrong password combination.
Access denied.
```

Gesture recognition accurately identifies number of fingers held up and their relative locations.

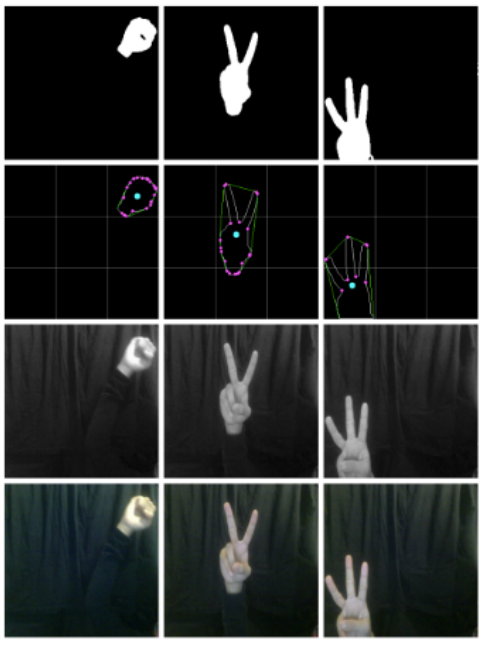
Successful entry 13/7



Expected: Accept
Actual: Accept

```
➜  visual-interfaces git:(master) ✗
python cam.py
Taking image in... 00:00 seconds
Capturing image!
Writing image file to
./live_captures/1424195547/file_1.JPG

Taking image in... 00:00 seconds
Capturing image!
Writing image file to
./live_captures/1424195547/file_2.JPG

Taking image in... 00:00 seconds
Capturing image!
Writing image file to
./live_captures/1424195547/file_3.JPG

Cleaned up camera.
./live_captures/1424195547/file_1.JPG >>
Fist, Center
./live_captures/1424195547/file_2.JPG >>
Five, Bottom-left
./live_captures/1424195547/file_3.JPG >>
Five, Top-right
Thank you for entering your password
combination. Access granted.
```

Successful Rejection 14/7



Expect: Reject
Actual: Reject

```
➜  visual-interfaces git:(master) ✗
python cam.py
Taking image in... 00:00 seconds
Capturing image!
Writing image file to
./live_captures/1424194418/file_1.JPG

Taking image in... 00:00 seconds
Capturing image!
Writing image file to
./live_captures/1424194418/file_2.JPG

Taking image in... 00:00 seconds
Capturing image!
Writing image file to
./live_captures/1424194418/file_3.JPG

Cleaned up camera.
./live_captures/1424194418/file_1.JPG >>
Fist, Top-right
./live_captures/1424194418/file_2.JPG >>
Two, Center
./live_captures/1424194418/file_3.JPG >>
Three, Botto-left
Entered wrong password combination. Access
denied.
```

VISUALLOCK.PY

```
import cv2, os, sys, time
import numpy as np
# import matplotlib.pyplot as plt

# ============================================================
# Data Reduction
# ============================================================

# resize image to 300 x 300 pixels
def resize(image):
    r = 300.0 / image.shape[1] # calculate aspect ratio
    dim = (300, int(image.shape[0] * r))
    image = cv2.resize(image, dim, interpolation =
cv2.INTER_AREA)
    return image

# rotate image 90 degrees counterclockwise
def rotate(image):
    (h, w) = image.shape[:2]
    center = (w / 2, h / 2) # find center
    M = cv2.getRotationMatrix2D(center, 270, 1.0)
    image = cv2.warpAffine(image, M, (w, h))
    show(image, 1000)
    return image

# convert color image to grayscale
def grayscale(image):
    image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    show(image, 1000)
    return image

# convert grayscale image to color (to permit color drawing)
def colorize(image):
    image = cv2.cvtColor(image, cv2.COLOR_GRAY2BGR)
    return image
```

```
# invert image colors
def invert(image):
    image = (255-image)
    show(image, 1000)
    return image

# find otsu's threshold value with median blurring to make image
black and white
def binarize(image):
    blur = cv2.medianBlur(image, 5)
    # better for spotty noise than
cv2.GaussianBlur(image,(5,5),0)
    ret,thresh = cv2.threshold(blur, 0, 255,
cv2.THRESH_BINARY+cv2.THRESH_OTSU)
    image = thresh
    show(image, 1000)
    return image

# apply morphological closing to close holes - removed from main
as it closes gaps
def close(image):
    kernel = np.ones((5,5), np.uint8)
    image = cv2.morphologyEx(image, cv2.MORPH_CLOSE, kernel)
    show(image, 1000)
    return image

# canny edge detection: performs gaussian filter, intensity
gradient, non-max
# suppression, hysteresis thresholding all at once
def edge_finder(image):
    image = cv2.Canny(image,100,200) # params: min, max vals
    show(image, 1000)
    return image

# find contours and convex hull and read that information using
parsing methods
def contour_reader(image):
    temp = image # store here because findContours modifes
source
```

```
    contours, hierarchy =
cv2.findContours(image,cv2.RETR_TREE,cv2.CHAIN_APPROX_SIMPLE)
    cnt = contours[0]

    # check curve for convexity defects and correct it
    # pass in contour points, hull, !returnPoints return indices
    hull = cv2.convexHull(cnt,returnPoints = False)
    defects = cv2.convexityDefects(cnt,hull) # array

    image = temp # revert to original
    cv2.drawContours(image, contours, 0, (255,255,255), 1)
    image = colorize(image) # prep for drawing in color

    # defects returns four arrays: start point, end point,
farthest
    # point (indices of cnt), and approx distance to farthest
point
    interdigitalis = 0 # representing tip of finger to convext
points
    if len(hull) > 3 and len(cnt) > 3 and (defects is not None):
        for i in range(defects.shape[0]):
            s,e,f,d = defects[i,0]
            start = tuple(cnt[s][0])
            end = tuple(cnt[e][0])
            far = tuple(cnt[f][0])
            # print start, end, far
            cv2.line(image,start,end,[0,255,0],1)
            cv2.circle(image,far,3,[255,0,255],-1)
            # print d
            if d > 6000: # set by trial and error
                interdigitalis += 1
    # print 'interdigitalis:', interdigitalis

    # find centroid
    M = cv2.moments(cnt)
    cx = int(M['m10']/M['m00'])
    cy = int(M['m01']/M['m00'])
    centroid = (cx, cy)
    cv2.circle(image, centroid, 6, (255,255,0), -1)
    show(image, 1000)
```

```
    # print 'centroid:', centroid

    gesture = classify(interdigitalis)
    # print 'gesture', gesture

    # calculate size for height as image is a numpy array
    h = len(image)
    location,lst = locate(h,cx,cy)
    # print location

    # draw visual grid based on list coordinates returned by
locate function
    for i in xrange(0, len(lst), 2):
        a = lst[i]
        b = lst[i+1]
        # print a,b
        cv2.line(image,a,b,[128,128,128],1)
    show(image, 1000)

    pair = (gesture, location)
    return image,pair

# ============================================================
# Parsing and Performance
# ============================================================

'''TODO: an alternative form of grammar could have encoded the
hand-based passwords into numbers or bitstrings

Fist : 0, Two : 2, Three : 3, Four : 4, Five : 5,
Center : 11, Top-left : 00 , Top-center : 10, Top-right: 20,
Bottom-left : 02 , Bottom-center : 12, Bottom-right : 22

Then the default password could be represented as: 011502520
instead of 'Fist' 'Center' 'Five' 'Bottom-left' 'Five' 'Top-
right'
'''
def classify(num):
    if num is 0:
        return 'Fist'
```

```
    if num is 1:
        return 'Two'
    if num is 2:
        return 'Three'
    if num is 3:
        return 'Four'
    if num is 4:
        return 'Five'
    else:
        return 'Unknown'

def locate(h, cx, cy):
    w = h # square image
    h1 = int(h/3)
    h2 = h - int(h/3)
    v1 = int(w/3)
    v2 = w - int(w/3)
    lst = [(0,h1), (w,h1), (0,h2), (w,h2), (v1,0), (v1,h),
(v2,0), (v2,h)]

    p1 = (v1,h1) # [0] = x, [1] = y
    p2 = (v2,h1)
    p3 = (v1,h2)
    p4 = (v2,h2)

    if cx > p1[0] and cx < p2[0] and cy < p3[1] and cy > p1[1]:
        return 'Center', lst
    elif cx < p1[0] and cy < p1[1]:
        return 'Top-left', lst
    elif cx > p2[0] and cy < p2[1]:
        return 'Top-right', lst
    elif cx < p3[0] and cy > p3[1]:
        return 'Bottom-left', lst
    elif cx > p4[0] and cy > p4[1]:
        return 'Bottom-right', lst
    elif cx > p1[0] and cx < p2[0] and cy < p2[1]:
        return 'Top-center', lst
    elif cx > p1[0] and cx < p2[0] and cy > p4[1]:
        return 'Bottom-center', lst
    else:
```

```
        return 'Unknown', lst

def authenticate(sequence):
    denial = 'Entered wrong password combination. Access
denied.'
    admittance = 'Thank you for entering your password
combination. Access granted.'
    confusion = 'There is an unknown value in your input. Access
denied. Try again.'
    overload = 'There was a surplus of tokens. Access denied.
Try again.'
    underflow = 'There were not enough tokens. Access denied.
Try again.'

    if len(sequence) > 3:
        return overload
    if len(sequence) < 3:
        return underflow

    for tup in sequence:
        if 'Unknown' in tup:
            return confusion

    # check three sequences
    tup1 = sequence[0]
    tup2 = sequence[1]
    tup3 = sequence[2]
    # print sequence
    condition1 = tup1[0] == 'Fist' and tup1[1] == 'Center'
    condition2 = tup2[0] == 'Five' and tup2[1] == 'Bottom-left'
    condition3 =tup3[0] == 'Five' and tup3[1] == 'Top-right'
    if  condition1 and condition2 and condition3:
        return admittance
    else:
        return denial

# ============================================================
# Helper Functions
# ============================================================
```

```
def save(image, name):
    cv2.imwrite(name, image)

def show(image, wait):
    cv2.waitKey(wait)
    cv2.imshow('Image', image)

# ========================================================
# Main Method
# ========================================================

def main():
    if len(sys.argv) < 2:
        sys.exit("Need to specify a path from which to read
images")

    imageformat=".JPG"
    path = "./" + sys.argv[1]

    combination = []

    # load image sequence
    if os.path.exists(path):
        imfilelist=[os.path.join(path,f) for f in
os.listdir(path) if f.endswith(imageformat)]
        if len(imfilelist) < 1:
          sys.exit ("Need to specify a path containing .JPG
files")
        for el in imfilelist:
            sys.stdout.write(el)
            image = cv2.imread(el, cv2.IMREAD_COLOR) # load
original
            image = resize(image)
            save(image, el[:-4]+'_resized.png')
            image = grayscale(image)
            save(image, el[:-4]+'_grayscale.png')
            image = binarize(image)
            save(image, el[:-4]+'_binarized.png')
            image = close(image)
            save(image, el[:-4]+'_closed.png')
```

```
            image,combo = contour_reader(image)
            save(image, el[:-4]+'_contours.png')
            combination.append(combo)
            print ' >> ' + combo[0] + ', ' + combo[1]

    else:
        sys.exit("The path name does not exist")

    # print combination
    decision = authenticate(combination)
    print decision

    time.sleep(5)

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

CAM.PY

```
import cv2, time, sys, os
import numpy as np

# Camera 0 is the integrated web cam
camera_port = 0
frame_size = 640
timer = 5
file_count = 1
dir_name = "live_captures/"+str(int(time.time()))

# Initialize the camera capture object with the cv2.VideoCapture
class.
# All it needs is the index to a camera port.
camera = cv2.VideoCapture(camera_port)
camera.set(3, frame_size)
camera.set(4, frame_size)

def draw_quadrants(size, frame):
    h1 = int(size/3)
    h2 = size - int(size/3)
    v1 = int(size/3)
    v2 = size - int(size/3)
    lst = [(0,h1), (size,h1), (0,h2), (size,h2), (v1,0),
(v1,size), (v2,0), (v2,size)]
    for i in xrange(0, len(lst), 2):
        a = lst[i]
        b = lst[i+1]
        cv2.line(frame,a,b,[128,128,128],1)

# Captures a single image from the camera and returns it in PIL
format
def get_image():
    # read is the easiest way to get a full image out of a
VideoCapture object.
    retval, im = camera.read()
    return im
```

```
def write_file_to_disk():
    global file_count
    global camera
    global dir_name
    # take the actual image we want to keep
    print('\nCapturing image!')
    camera_capture = get_image()
    imageFile = "./" + dir_name + "/file_"+ str(file_count) +
".JPG"
    print 'Writing image file to', imageFile, '\n'
    # nice feature of the imwrite method is that it will
automatically choose the
    # correct format based on the file extension you provide.
Convenient!
    cv2.imwrite(imageFile, camera_capture)
    file_count += 1

    if file_count > 3:
        camera.release()
        # read the fresh sequence of gestures
        ''' TODO: link these classes properly instead of
hijacking os system '''
        os.system("python gesturereader.py " + dir_name)
    else:
        open_camera_feed()

def open_camera_feed():
    global camera
    global dir_name
    global frame_size

    # create directory
    if not os.path.exists(dir_name):
        os.makedirs(dir_name)

    local_timer = timer
    start_time = int(time.time());

    while (camera.isOpened and local_timer <= timer):
        val, frame = camera.read() # read the frame
```

```
        draw_quadrants(frame_size, frame)
        cv2.imshow('video', frame)

        reverse_timer = timer - local_timer
        mins, secs = divmod(reverse_timer, 60)
        timeformat ='\rTaking image in... {:02d}:{:02d}
seconds'.format(mins, secs)
        sys.stdout.write(timeformat)
        sys.stdout.flush()
        local_timer = int(time.time()) - start_time
    write_file_to_disk()


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