Head and Eye Controlled Mac Computer functions

Howon Byun & Joseph Shepley
{hb2458, jls2303}@columbia.edu

Table of contents

Introduction

The majority of computer users use their hands when operating a computer, and as efficiency and accessibility becomes an ever increasing desire in users, methods of meeting those challenges are needed. We propose a system that will use a webcam to track the user's ***eye*** and ***head*** movement which will help them access and manipulate an application to increase productivity and accessibility. This is an excellent example of a visual interface since live video input will be analyzed and visual input will signal decisions. The decisions will be constructed based on a sensible grammar of blinking and head movement. Ultimately, directional signals from head and binary signals from blinking will allow the user to open and close an application and position the application window with increased efficiency over traditional dragging, clicking or keyboard methods. At the very least, we hope to bring greater accessibility to non-traditional computer users such as those individuals who operate who are missing fingers or hands entirely.

Problem Formulation

For decades programmers have tried to incorporate input methods beyond the keyboard and the mouse to provide the user with additional bits of information to control their devices. However this meant that hands became the dominant means through which these tools were manipulated, with other body parts taking a back seat. As such engineers have been motivated to make use of traditionally underutilized body parts to augment the now overloaded hands and fingers. Engineers have proposed solutions ranging from voice command like the Microsoft Cortana,[1] complicated 3D cameras embedded directly into the screen,[2] to even a foot pedal,[3] all to come up with new ways to let users intuitively interact with their computers. Research into this topic is also meant to widen ***accessibility*** to disadvantaged users for whom conventional input methods are not as applicable.

With the recent increase in availability of cheap laptops and webcams that usually accompany them, it is now easier than ever to capture ***visual inputs*** from the end user. Webcams are usually placed in the deadcenter of the laptop's upper frame, thereby providing a wide field of view to capture the user's face well. Webcams are integrated seamlessly into the screen to make it more user friendly than other more intrusive systems such as Leap Motion which requires a third-party device, and external hardware. However not everything is all good and done despite the ubiquitous prevalence of webcams. The system that makes use of this source of input must also be designed intelligently and intuitively. Many try to use hand gesture recognition as it contains a high information density (even detecting finger presence could

[1] https://www.microsoft.com/en-us/windows/cortana
[2] https://www.intel.com/content/www/us/en/architecture-and-technology/realsense-overview.html
[3] https://www.amazon.com/USB-Foot-Switch-Keyboard-Pedal/dp/B008MU0TBU

provide upto 10 bits of information) at ever increasing performance, with some implementations reporting over 90% accuracy on livecam feed. [1] Still, hand detection has inherent problems. First and foremost, proper lighting is crucial as a hand usually resides in a 3D plane. Shadows and darker regions can form that ambiguates user inputs when a single webcam is used. Even greater still, any form of system that requires the user to hover arm/hand in the air is prone to causing fatigue, discouraging users from elongated use. Lastly, a computer user's hands themselves are occupied either controlling a mouse or typing on a keyboard. As a user, taking one's hands away from the keyboard or dragging a mouse and clicking takes time, and and in the event of online research and note taking, opening and closing an application, manipulating browser and application windows, and entering text involves bits of wasted effort due to one's hands being necessary to control these things. Additionally, those with less traditional accessibility related to using the computer would face challenges inherently due to the fact that by default, the computer is made for being controlled with one's hands.

Program Description and Manual

In summary, we present a program that allows a user to control certain computer functions using one's eyes and head. The computer functions that we allow a user to control with their eyes or head are the following: shift window left, shift window right, copy text to clipboard, paste item in clipboard, close an app, expand all applications, switch desktop windows.

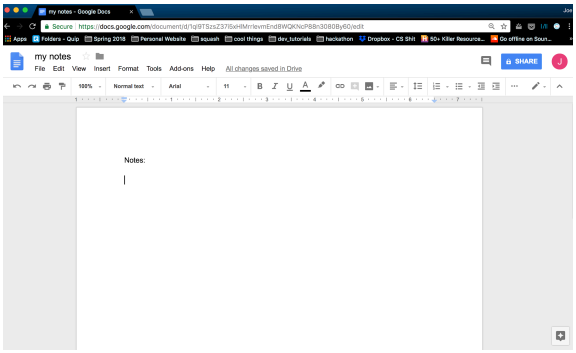
For the purpose of this project we are limiting ourselves to Mac OSX, since it was the only operating system both of our computers ran. Though the programmed macros are designed for this OS, the grammar itself is extensible enough to be port over to other operating systems like Windows and Linux. Unfortunately Mac does not have a native window placer (used for snapping it to the left or right), we are using Spectacle[4] under the hood. Other aforementioned OSes have such features built in natively so it would not be necessary.

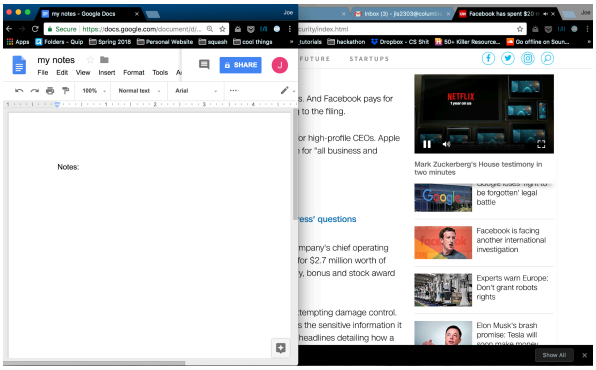
Our program makes heavy use of OpenCV[5], DLib[6], and NumPy[7] for image processing. Each macro is sent to the computer using PyAutoGui[8], which can send various keystrokes and mouse actions to the underlying operating system.

[4] https://www.spectacleapp.com/
[5] https://opencv.org/
[6] http://dlib.net/
[7] http://www.numpy.org/
[8] http://pyautogui.readthedocs.io/en/latest/

Usage Sketch

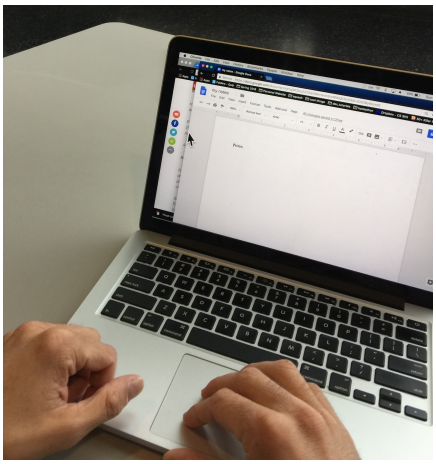
Take for example, the task of adjusting and moving a window to the left during a note taking task. Given the starting note-taking app position (left image), one wants to move it and resize it



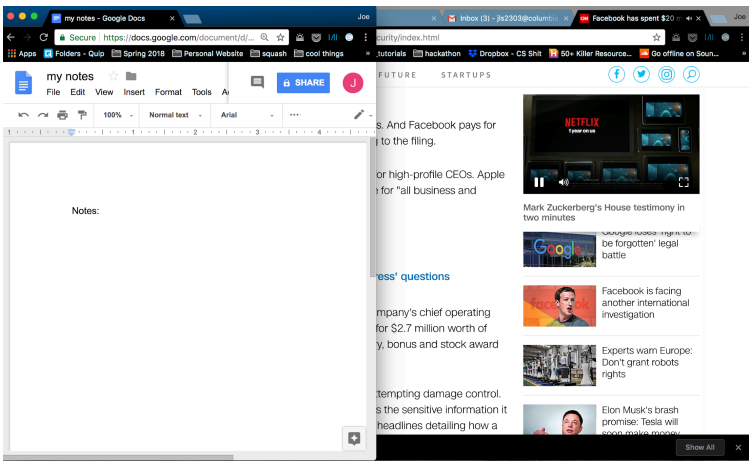


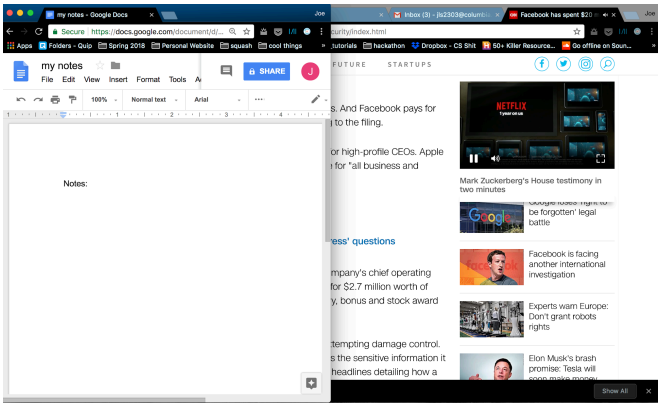
to the left (see the image on the right).

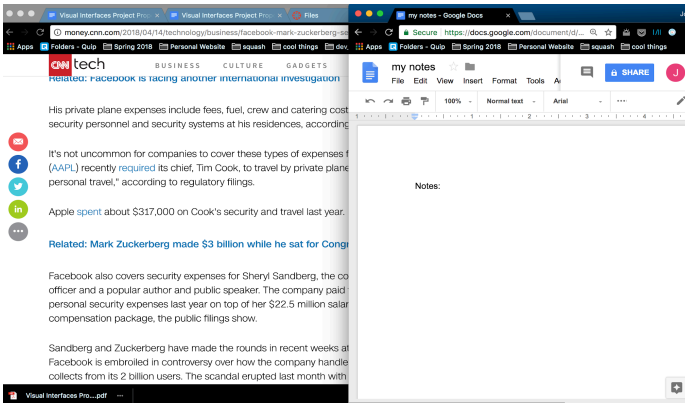
Without our system:



One must manually drag window using mouse and hand like in the pictures below.

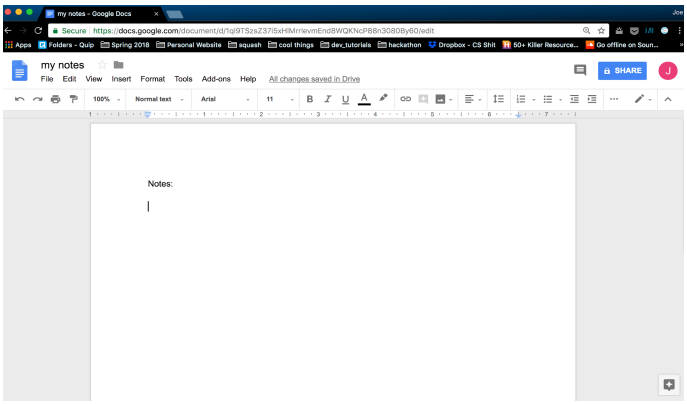






With our system: one can instantly shift window left based on eye signal

Easily shift to the right or back to normal


my notes - Google Docs
Secure https://docs.google.com/document/d/1ql9TSzsZ37i5xHlMrrIevmEnd8WQKNcP88n30B0By60/edit
Apps
Folders - Quip
Spring 2018
Personal Website
squash
cool things
dev_tutorials
hackathon
Dropbox - CS Shit
50+ Killer Resource...
Go offline on Soun...
my notes
File Edit View Insert Format Tools Add-ons Help
All changes saved in Drive
SHARE
100%
Normal text
Arial
11
Notes:

Starting our program

Our program starts by running simple command `./run.sh` in the terminal. Once the program is running the user can control the following commands with their head or eyes.

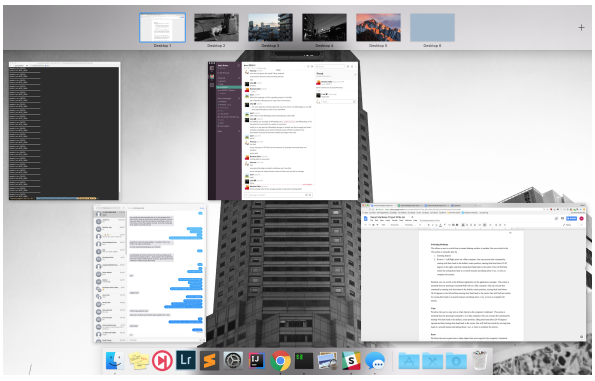
Expose

We allow the user to expand all applications like in the picture below. This action is normally done by pressing f3 on a Mac computer. One can execute this command by bringing their face

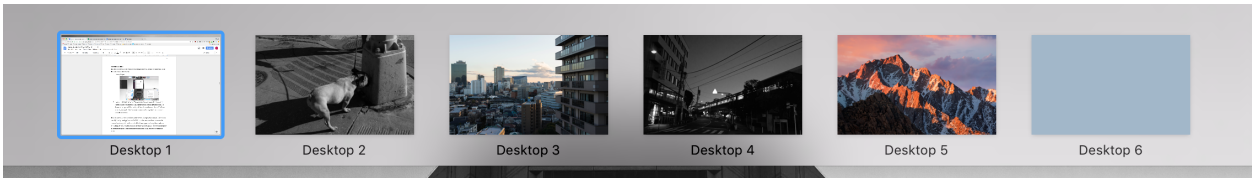
close to the computer in a "zoomed" position.

Switching Desktops

This allows a user to switch from a current desktop window to another. One can switch to the This action is normally done by



1. Enter Expose by scrolling down with four fingers.
2. Move cursor to one of the miniature desktop previews.



3. Click to enter a new desktop.

Using our app one can execute this command by starting with their head in the default, center position, turning their head about 30-45 degrees to the right, and then turning their head back to the center. One will find best results by turning their head in a smooth manner and taking about 1 second or more to either direction to complete the motion.

Copy

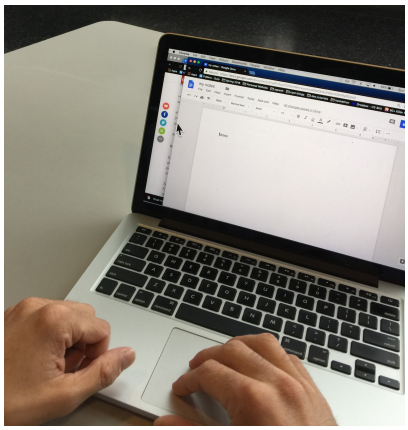
We allow the user to copy text or other objects to the computer's clipboard. This action is normally done by pressing Command-C on a Mac computer. One can execute this command by starting with their head in the default, center position, tilting their head about 20-30 degrees upward and then turning their head back to the center. One will find best results by moving their head in a smooth manner and taking about 1 sec. or more to complete the motion.

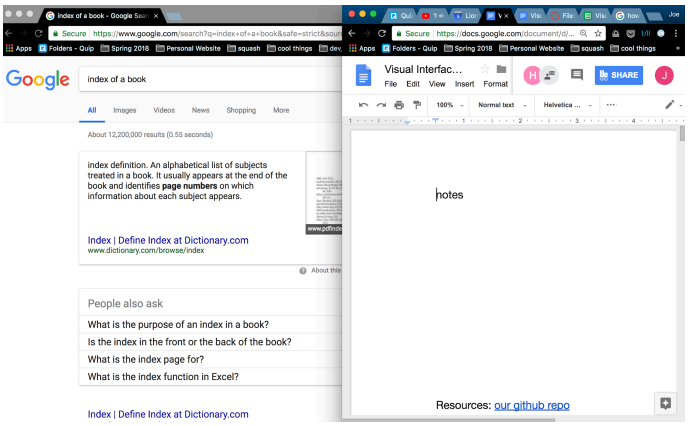
Paste

We allow the user to paste text or other objects that were copied to the computer's clipboard. This action is normally done by pressing Command-V on a Mac computer. One can execute this command by starting with their head in the default, center position, tilting their head about 30-45 degrees downward and then turning their head back to the center. One will find best results by moving their head in a smooth manner and taking about 1 sec. or more to complete the motion.

Snapping Windows

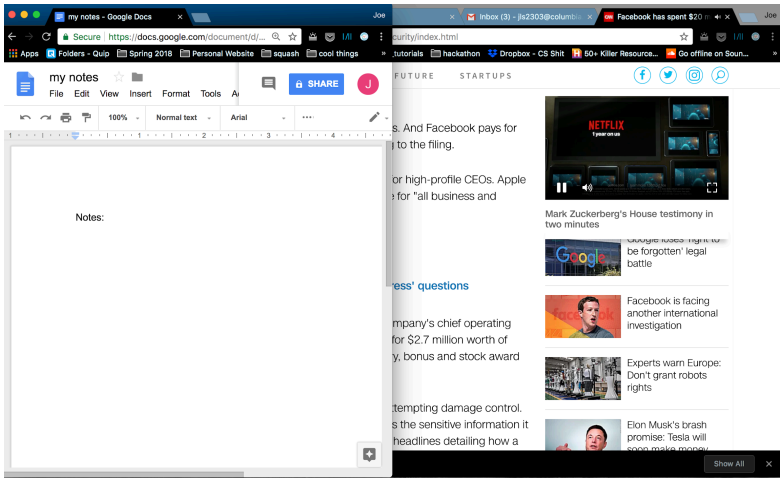
See section under **Example Usage** for even more visuals regarding this specific command. We allow the user to snap windows to the left or right. This action is normally done on a Mac by



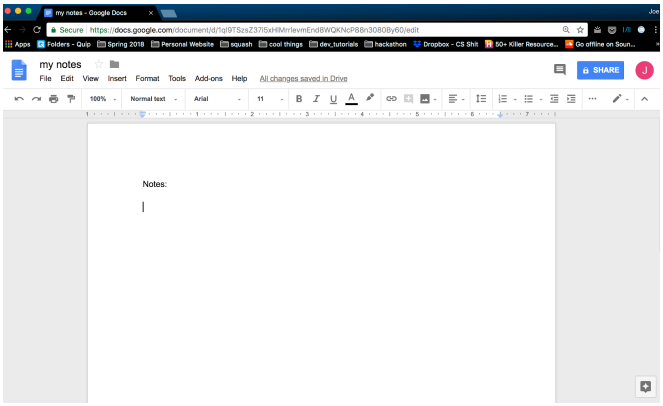


dragging a window to the right like in the picture below.

However, one can instantly “snap” the window to the right like in the picture above using our system by winking their right eye. One will find best results by winking at a moderate speed at about 0.5 sec. for their blink.



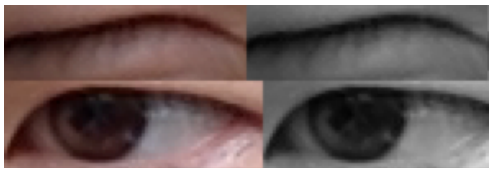
Similarly, one can “snap” the window to the left by winking the left eye in the same way.



Lastly, one can “snap” the window back to full screen by blinking with both eyes once.

Domain Engineering

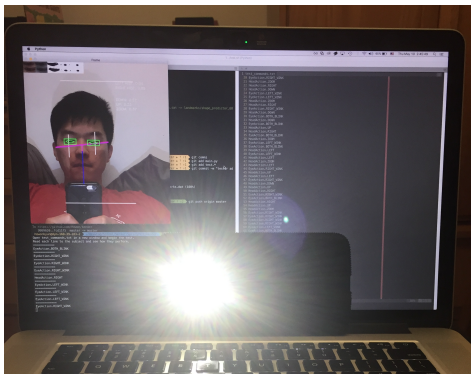
User



An ideal user would be a good winker, meaning that when they wink one eye the other remains relatively still (look above for an example an ideal wink vs. non-ideal wink). He/she should also have a fairly good posture without too much hunch.

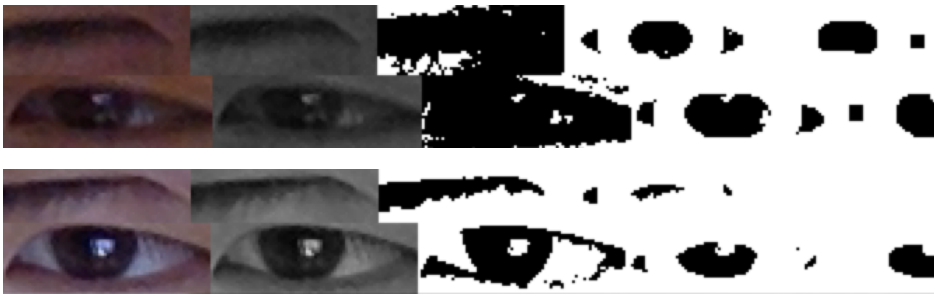
Setup

There are a few steps that we require the user to follow in order to get the best performance out of our application. First off, good lighting is paramount to get the best accuracy. Preferably there is a source of light from underneath the user's eyes as to remove shadow generated by ceiling

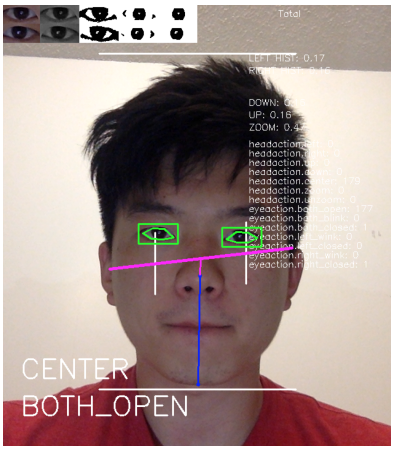


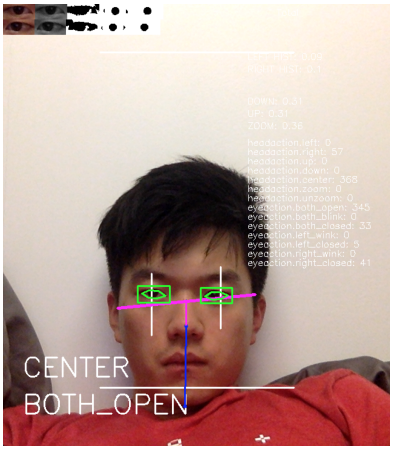
light. For instance we placed a cell phone with its flashlight on when testing our subjects.

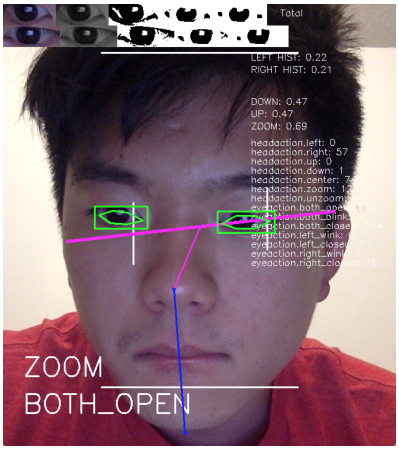
There is a substantial improvement in both accuracy and noise reduction when this method is utilized. Compare above image without a flightlight to the one below with a flashlight. The above screenshot displays far more noise in the sclera region in the opened right eye. For the left, closed eye the eyelash occupies a large volume of the frame. The bottom screenshot, however, has far fewer noise (third frame in the fourth row) and the closed left eye leaves relatively little amount of trace (fifth frame in the third row). This helps our cornea detection algorithm that forms the basis of blink-based grammar our application relies on. Once the lighting condition is established, we also asked the user to exhibit a good posture. A



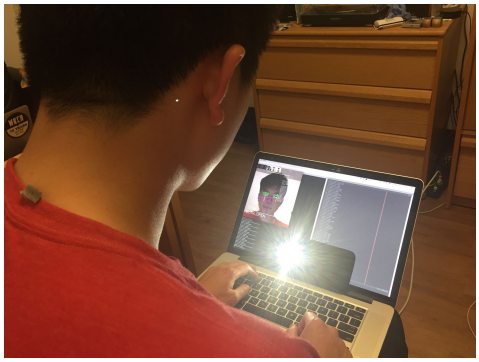
simple way to accomplish this was to display “guidelines” that directed the user’s head to be in a certain position when they were familiarizing themselves with our application. These guidelines are represented as two horizontal lines towards the top and bottom of the frames.





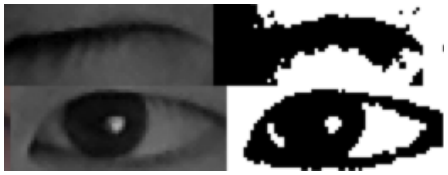


Keeping one's head inside these two lines enforces a crude form of gesture-correction. Slouching back too far dips the head below the bottom line and limits the range of motion. Leaning in too much and having the face be outside the bounds means that the head starts in a "zoomed" state as well as making our eye detection behave poorly for reasons detailed in the succeeding sections. Ideal posture, defined as the relative height of the face to the position of the webcam, is also accomplished by a good positioning




of the camera.

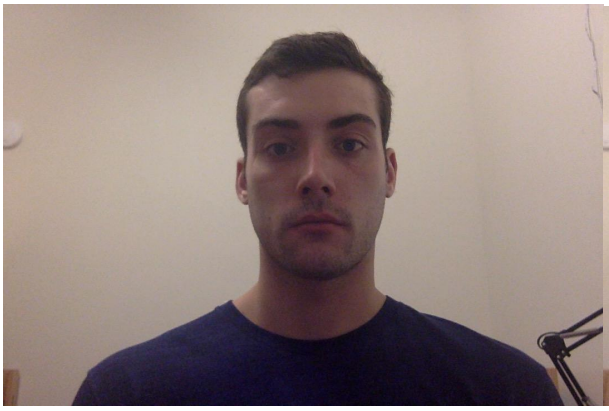
Notice that the third frame of the first row has visibly more noise than the corresponding frame in the third row, even when both of them were captured using the flashlight with the head between the guiding lines.



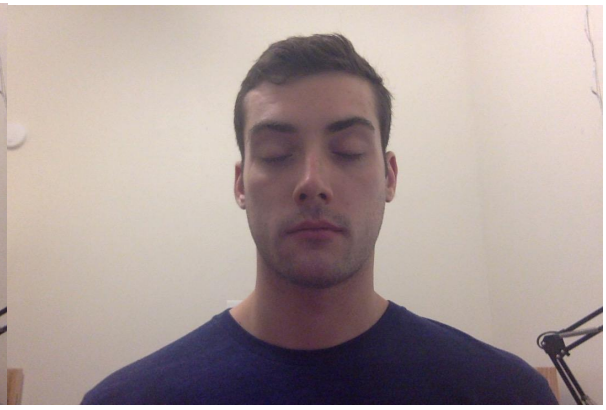

Methods

The core component of our program relies on the ability to detect nine basic actions, and we will explain our methods and algorithms in doing so. The first five have to do with the orientation of the user's head. We detect whether the head is turned ***left*** or ***right*** on its vertical axis (yaw) as well as whether the head is tilted ***up*** or ***down*** (pitch). If the head is in none of these positions, then it is considered to be ***center***. We also detect whether the head is very close to the camera or not and if it is, we say the head is *zoomed*. Additionally, we detect actions related to opening and closing different eyes. We detect whether the right eye has shut (***right_closed***), the left eye has shut (***left_closed***), both eyes are shut (***both_closed***) or both eyes are open (***both_open***). The images below will make it clear to the reader.

enter

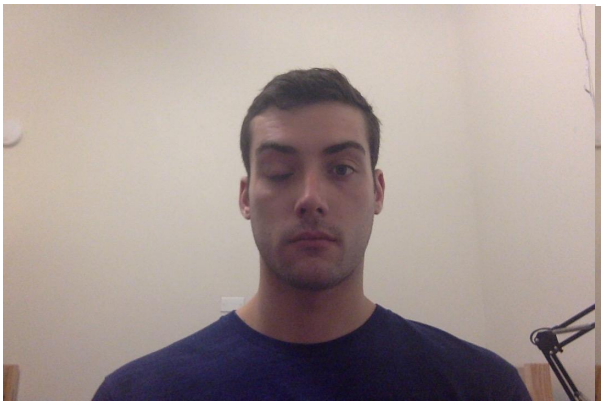


both_closed

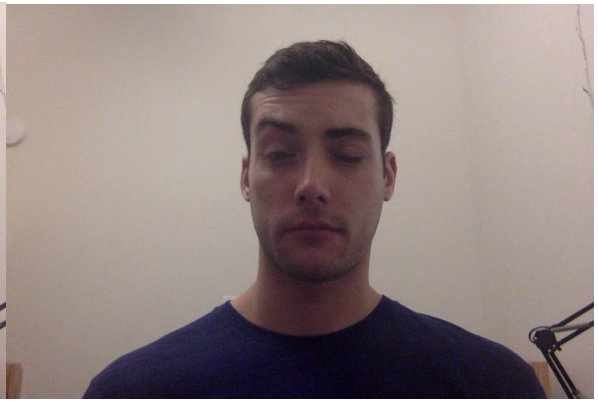


---

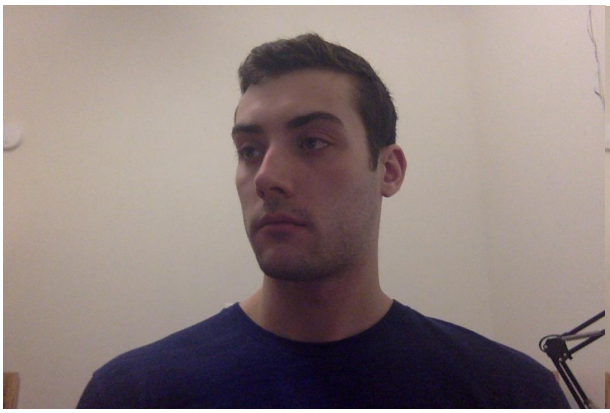
left_closed:



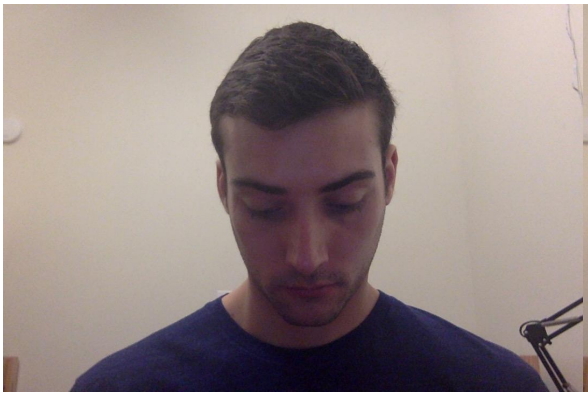
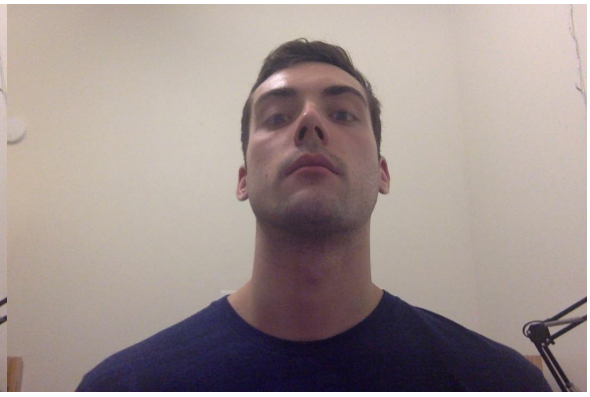
right_closed



left **right**




down **up**

zoomed

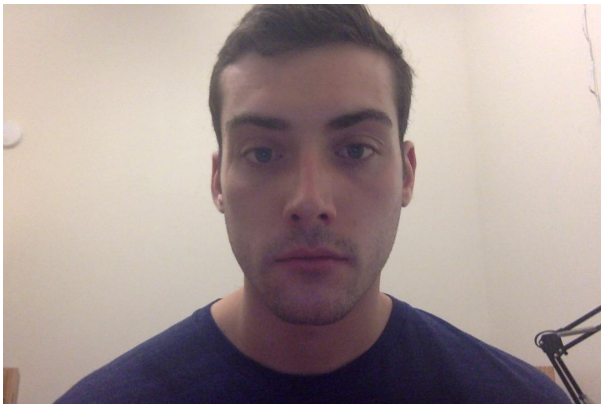
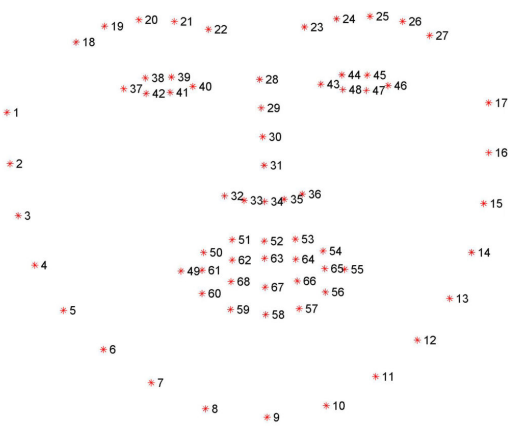
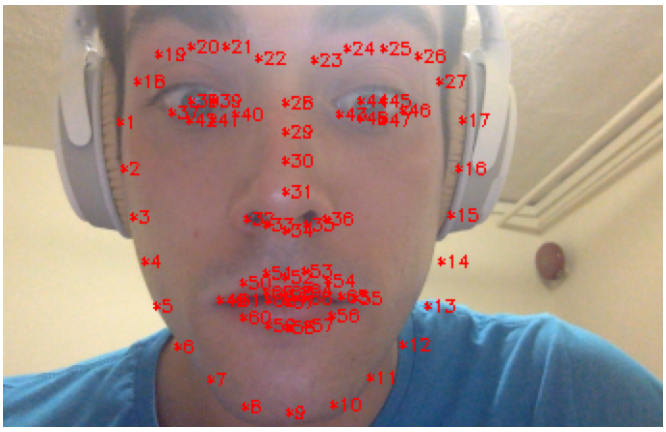


Image Analysis & Visual Processing

Our detection process firsts starts with detecting a head within an image and extracting facial landmarks. We used dlib[9] which is a package containing machine learning algorithms including face detection, and got inspiration from work by Adrian Rosebrok regarding detecting facial landmarks.[10] First we capture an image of of a user, then run **dlib's** face detection algorithm. The algorithm returns an array of 68 points whose indices map to facial landmarks. In the pictures below on can see the index mapping as well as actual results on a real image. The real image





shows the indices and points for indices 36-48 right now i.e. just the eyes.

From the mapping on the left we are able to locate approximate locations of important facial landmarks such as the eyes, tip of the nose, left or right side of the head, and the chin.

We can then compute the center of the head in different ways. One way that worked consistently was getting the midpoint of the ***rightmost*** and ***leftmost*** points on the face (landmarks **1** and **16**).

Once we have knowledge about those points of interest on the face, our action detection algorithms are possible.

Head Detection

The most important features we extract for the purpose of detecting head movements are:

1. Head Center (***head_center***)
2. Chin tip (***chin_tip***)
3. Nose tip (***nose_tip***)
4. Ratio of ***chip_tip*** -> ***nose_tip*** distance to ***nose_tip*** -> ***head_center*** distance (***ch_ratio***).

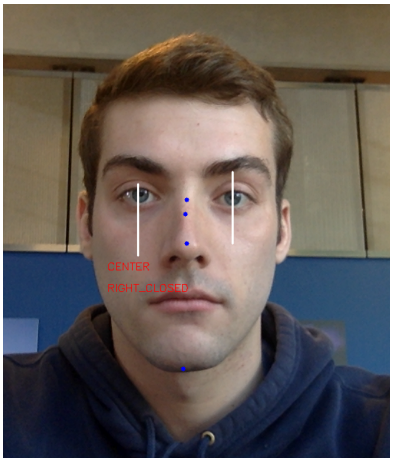
Left-turn and Right-turn Detection

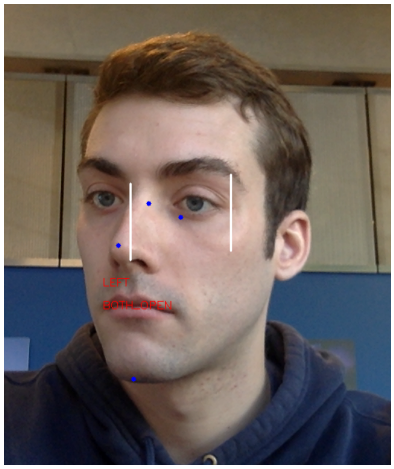
---

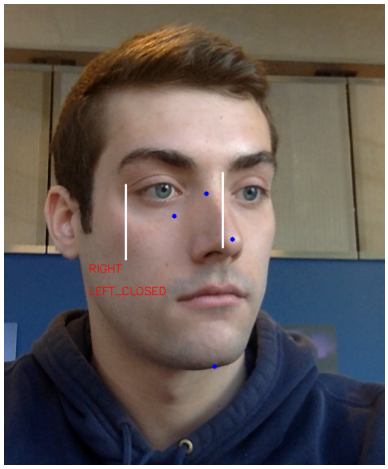
[9] http://dlib.net/

[10] https://www.pyimagesearch.com/2017/04/10/detect-eyes-nose-lips-jaw-dlib-opencv-python/

1. The white line represents the vertical line through the midpoint between the **head_center** and the leftmost point on the head.
2. Check if ***nose_tip*** is left of the line.
3. For right-turn detection, perform steps 1) and 2) but check to see if ***nose_tip*** crosses the



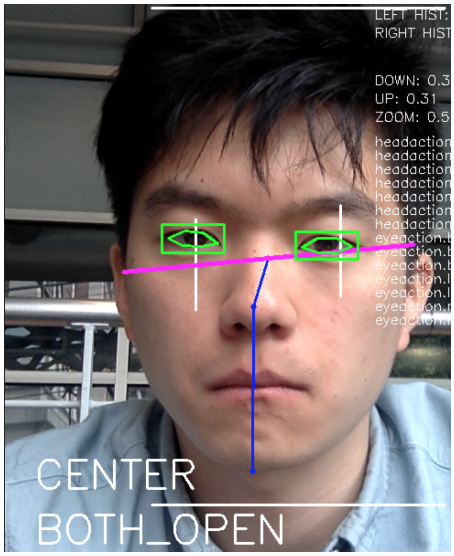


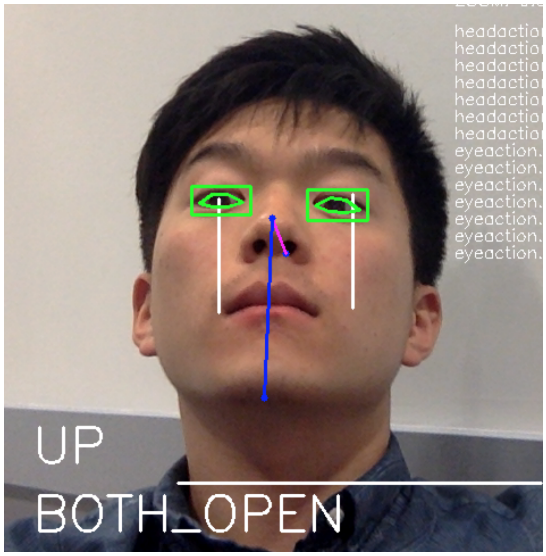


midpoint between the head center and the rightmost point.

Up Detection:

1. Check if the ***nose_tip*** is **above** the ***head_center***.
2. Get the distance between ***head_center*** and ***nose_tip*** (***nh_dist***).
3. Get the distance between ***nose_tip*** and the ***chin_tip*** (***nc_dist***).
4. Take the ratio of ***nh_dist*** to ***nc_dist*** and compare it to a threshold (0.2, ***nd_dist*** now pink).

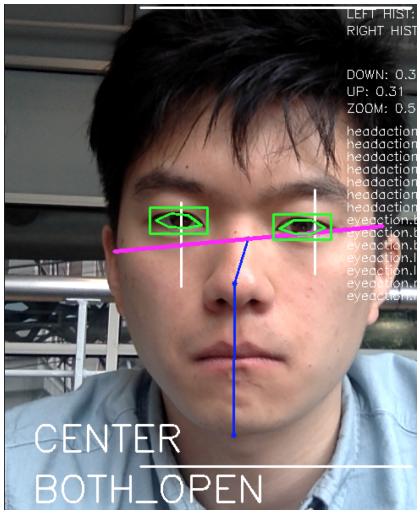


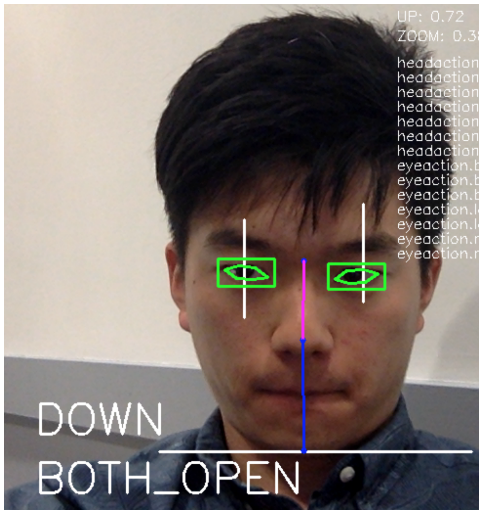


a. If the user is looking **up**, ***nc_dist*** would decrease while ***nc_dist*** would not.

Down Detection

1. Check if the ***nose_tip*** is **below** the ***head_center***.
2. Get the distance between ***head_center*** and ***nose_tip*** (***nh_dist***).
3. Get the distance between ***nose_tip*** and ***chin_tip*** (***nc_dist***).
4. Take the ratio of ***nh_dist*** to ***nc_dist*** and compare it to a threshold (0.5, ***nd_dist*** now pink).



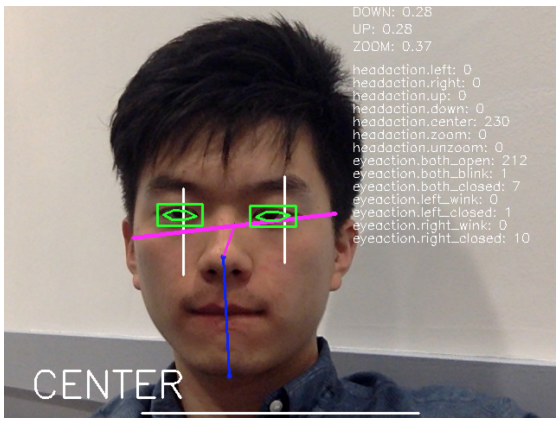


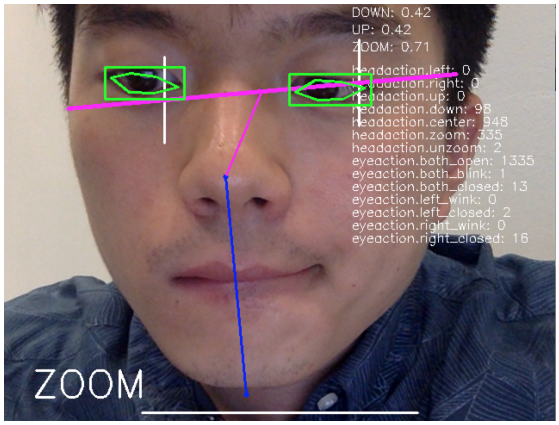
a. If the user is looking **up**, ***nc_dist*** would increase while nh_dist would decrease.

Zoom Detection

1. Get the distance between both ears which is the width of the head (***ear_dist***, pink line).
2. Get the width of the frame (***f_width***).

3. Take the ratio of ***ear_dist*** to ***f_width*** and compare it to a threshold (0.5).





a. ***ear_dist*** increases monotonically as the user leans forward.

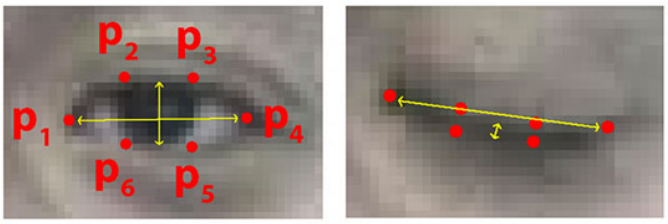
Center Detection

Any head position that does not fit in to the above five states are considered as center.

Eye Detection

Eye Aspect Ratio

Initially we implemented ***eye-aspect-ratio*** based algorithm to detect whether an eye is closed or not. An article on pyimagesearch.com[11] demonstrated a plausible success so we quickly modified



their work to fit our problem domain. The idea behind this algorithm is that when the eye is open, the vertical distance would proportionally increase while the width stays relatively static. By using the ratio between the height of the eye and the width, the openness of the eye could be determined. Though this method did perform admirably on average, it just was not reliable enough for us to continue using it. First and foremost this adapted poorly to the user's distances

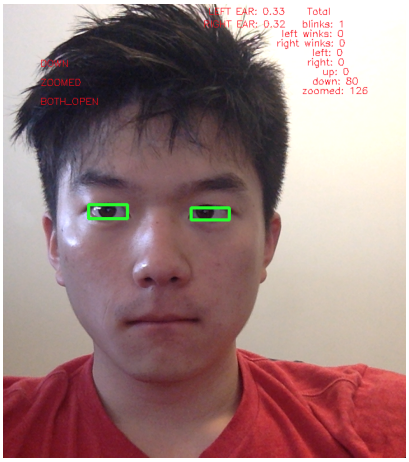
to the camera. Even the slightest increase in distance to the camera (abbreviated as ***DTC*** from here on) leads to a significance drop in accuracy resulting from the enlarged contour that surrounds the eye indices obtained from the landmark-based algorithm. This practically renders

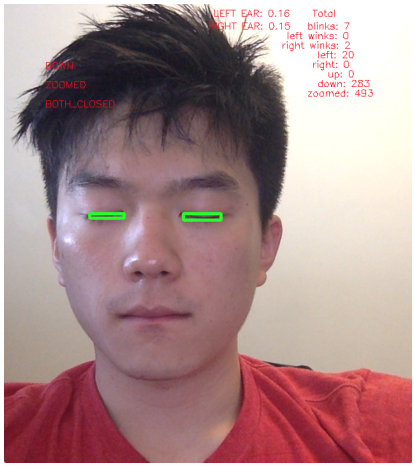
casual winks/blinks, where the user winks without exerting too much force impossible to detect. This method also does not fair against forced winks either. If the user squints too much the bounding hexagon forms at a completely irrelevant location, rendering this method useless. We eventually decided that a live-time system system like ours need to be able to handle such nuanced gestures and began a series of incremental improvements to improve our accuracy.

[11] https://www.pyimagesearch.com/2017/04/24/eye-blink-detection-opencv-python-dlib/

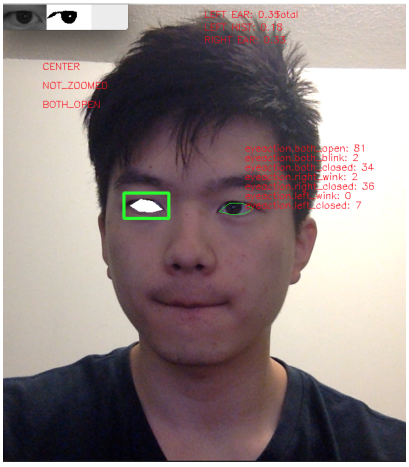
Histogram

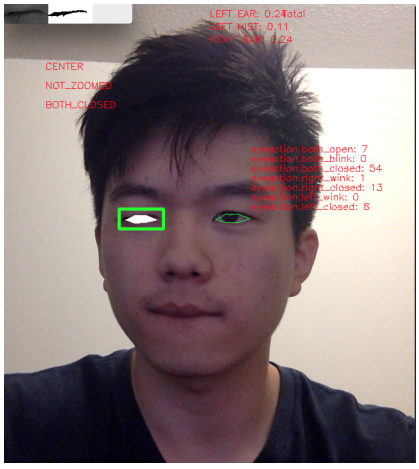
Upon realizing the shortcomings of the EAR-based algorithm to handle various workloads, we





thought perhaps using the histogram of the contents inside the eye might yield a solution. We can

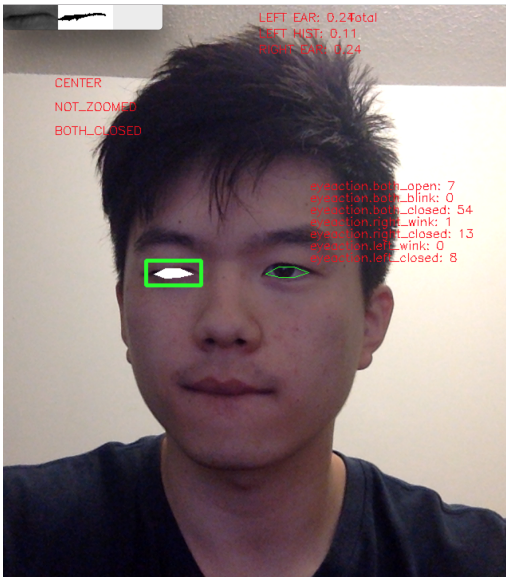


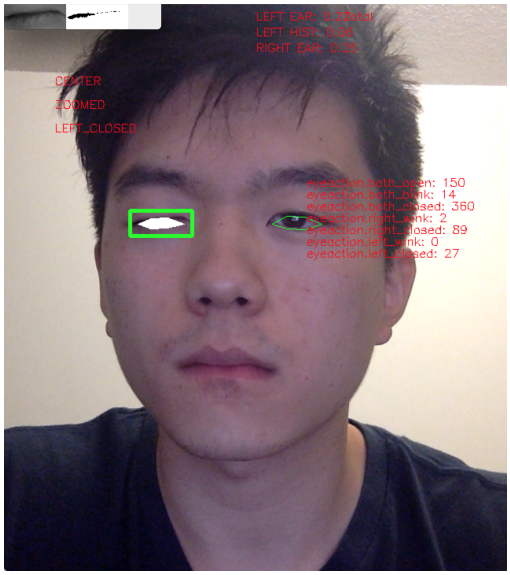


use the bounding hexagons from the previous part and form rounding rectangles to find the minimum-bounding rectangles for both eyes. From this extracted region we can binary the image to find pure white and pure black regions. This approach handled ***DTC*** better than the previous EAR-only method since the bounding box can track the eye as well as not being affected by the squinting action as much. We now simply have to determine the ***ratio*** of ***black pixels*** to the ***eye-bounding box*** to reach a binary decision of "blinks". In practice, blinking of one eye exerts stress on the other and constricts the other eye as well. Our system therefore uses a ***comparison based*** detection where first the absolute percentage of black pixels are determined and then masses of the two eyes are compared. If one is significantly smaller than the other we consider it blinked.

Gaussian Blur + Histogram

To further improve on our initial observation of the effectiveness of histogram-based approach we added a blurring layer to smooth-out figures in the binarized eye-region. This had an effect of





removing residual eyelashes formed when the upper eyelid touches the bottom eyelid. Notice the diminished presence of eyelash on the right image. Blurring masks away some of the short, stray hairs around the main eyelash line, further differentiating blinks from non-blinks.

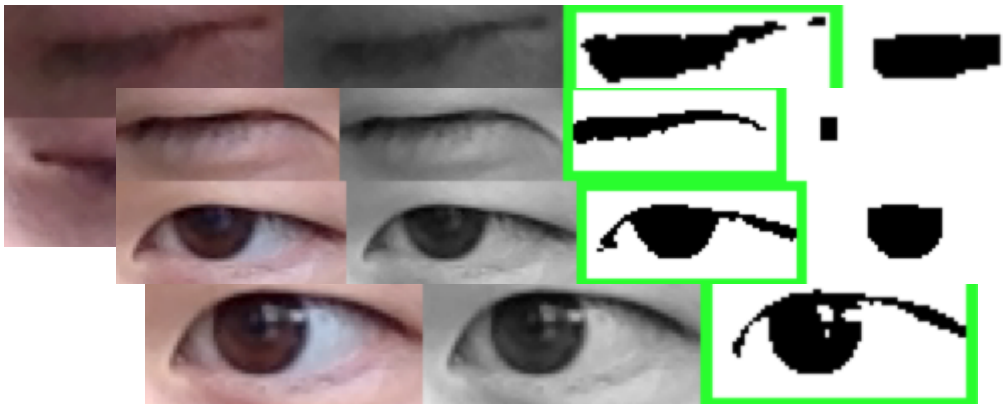
Erosion + Masking + Histogram

At this point we had only calibrated our eye-detection system on Howon. However testing on Joe revealed a few problems that we had not anticipated. First problem we encountered was that of Joe's thick, luscious eyebrows. Howon has relatively thin eyelash that almost disappears he closes his eyelids when Gaussian blur was applied. The algorithm so far however failed to effectively remove Joe's eyelashes. His longer eyelashes form a region almost larger than that of his cornea even when his eyes are closed. This presented a fundamental challenge to the assumption behind our algorithm: that opened eyes would have much more black regions as sclera would be white. Another problem was that of how much we were masking away to binarize eye regions. The threshold we were using, which was calibrated to blacken Howon's cornea out, masked away Joe's ***blue*** iris. This is because Joe has ***blue eyes*** meaning that the

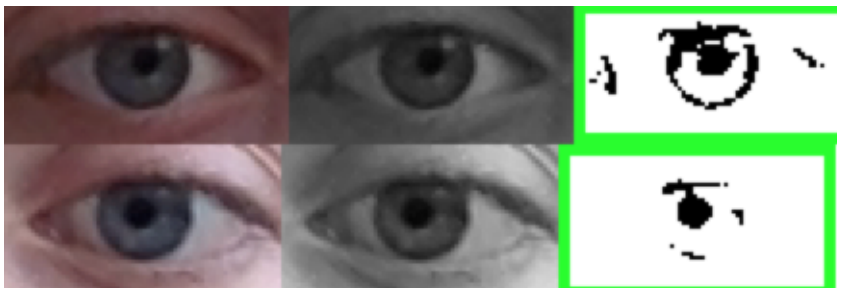
threshold value we had been using, which was set to match Howon's much darker ***brown*** iris,

thereby putting the color above the threshold and rendering them white.

The first problem was solved by performing ***dilation*** and ***erosion***. The former was used to



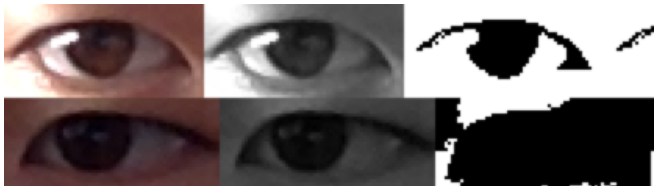
remove the stray hairs around his eyelids, while the latter was used to further emphasize the surviving black regions[12]. Since dilation would have masked the eyelashes out with minimal



effect to the corneal region, following up with erosion seemed like a logical choice to increase contrast, making our system easier to detect open/closed eyes. The example above illustrates how the thick region of eyelash in the third panel in the top row disappears after dilation + erosion was applied, shown in the fourth step.

[12] https://docs.opencv.org/2.4/doc/tutorials/imgproc/erosion_dilatation/erosion_dilatation.html

The second problem-that of iris color-seemed easily fixable by increasing our threshold value so that his blue eyes would now be considered black. However that lead to increased false positives



from skin around the eyes. Increasing the threshold means that more pixels would be considered black. If the user has darker skin tone or the lighting was subpar more of the outside-sclera region would also be considered black, further confusing our algorithm.

This was solved by first forming a bounding ellipse around the eyeball and masking away regions that fall outside the ellipse. This was obtained by using the 6-point polygon obtained using the landmark-based algorithm in the first iteration of this process. Notice how in the bottom row regions outside the eye are basically ignored using this masking technique.

However this alone is not enough to solve the problem. In cases where the eyelash is too thick,

using one mask cannot sufficiently be removed using the previous two steps alone. We therefore added another elliptical mask that basically extracts the middle, cornea from the eye rectangle. This two-step masking technique performed surprisingly well once the user is familiarized with the system. In particular it could detect at near 80% accuracy for both of us. For a system that does not use any machine-learning technique, we think it performs admirably given the lack of domain expertise both of us have.

As far as algorithms we did not consider, here are some and the justification behind our decision.

1. Hough circle detection

   This seemed like a logical choice. We could try to see if there exists a circular region inside the eye (cornea). Unfortunately the top eyelash happens to merge into corneal region in most cases so this circle detection algorithm was finicky if not ineffective.

2. Line fitting

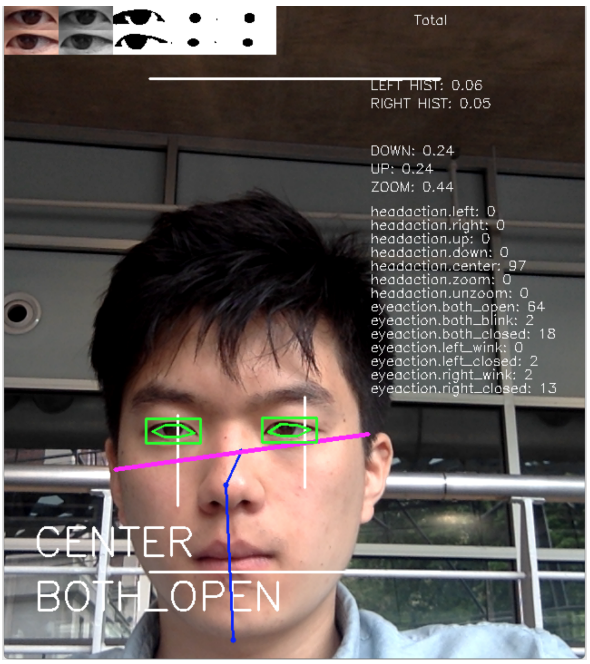
   Like Hough circle detection, one could try to fit a line through the image to see if there exists a line that spans widthwise. However this ran into the problem of the skin region that is otherwise solved using the ellipse method.

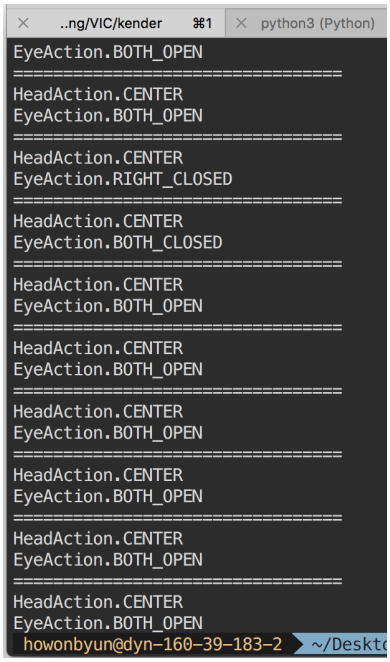
User Interface

Since our application is meant to run in the background, it does not have a formal interface that the user interacts with. With that said, the debugging window could be displayed to show various statistics and detected gestures by passing in `-d` flag. This display includes

1. Various lines denoting head thresholds.

2. Counters for each action.
3. Overlay of various transformations the user's eyes undergo when our system makes a decision on blinks.





The log of actions can also be displayed with a `-l` flag.

Grammar

Once we are able to detect these nine basic actions, we can start building our grammar. Our grammar is composed of

Left_turned = [*center, left, center*]

Right_turned = [*center, left, center*]

Up_nod = [*center, left, center*]

Down_nod = [*center, left, center*]

Zoom = [*center, zoom*]

Unzoom = [*zoom, center*]

Both_blink = [*both_open, both_down, both_open*]

Left_wink = [*both_open, left_down, both_open*]

Right_wink = [*both_open, right_down, both_open*]

We also have "rest states" defined for both head and eye movements.

HEAD_REST: CENTER, **EYE_REST**: BOTH_OPEN

In practice, some of the actions were much harder to be performed independently. For instance, we noticed that our subjects had a tendency to snap back in a downward motion when returning from left/right turns. In some cases people returned while leaning forward, triggering our ***zoom*** action. As such we generalized our grammar to account for such tendencies. Our grammar parsing rules consists of entries that follow the form

```
success = {
    EyeAction.LEFT_CLOSED: ([EyeAction.BOTH_OPEN], EyeAction.LEFT_WINK),
    EyeAction.RIGHT_CLOSED: ([EyeAction.BOTH_OPEN], EyeAction.RIGHT_WINK),
    EyeAction.BOTH_CLOSED: ([EyeAction.BOTH_OPEN], EyeAction.BOTH_BLINK),
    HeadAction.LEFT: ([HeadAction.CENTER, HeadAction.ZOOM], HeadAction.LEFT),
    HeadAction.RIGHT: ([HeadAction.CENTER, HeadAction.ZOOM], HeadAction.RIGHT),
    HeadAction.UP: ([HeadAction.CENTER, HeadAction.ZOOM], HeadAction.UP),
    HeadAction.DOWN: ([HeadAction.CENTER, HeadAction.ZOOM], HeadAction.DOWN),
    HeadAction.CENTER: ([HeadAction.ZOOM], HeadAction.ZOOM),
    HeadAction.ZOOM: ([HeadAction.CENTER], HeadAction.UNZOOM)
}
```

`previous_action: ([current input], triggered action)`

Statements. This allows multiple follow-up actions to be mapped to the same trigger actions. Our finalized system grammar reflects the tendencies we observed when developing this system and works robustly in discerning discrete action states. `previous_action` is set whenever an

action has happened for more than a ***threshold amount of frames***, set differently for eye-based gestures and head-based gestures. This helps alleviate ***midas effect*** from our system. We initially planned on adding a wake-command to our application but upon testing we concluded that it would be too tiring on the user. Instead we picked a threshold for the number of consecutive actions held high enough so that random movements of the eyes and the head would not trigger our system continuously.

Since our system gathers actions from ***both*** eyes and the head, we had to devise a way of picking one action over the other. For the sake of simplicity our program always favors head actions over eye actions. This is based on the fact that our head detection is far more reliable than eye detection and as such it should be trusted over the latter. With that said, if there is any non-rest action registered as a `previous_action`, actions of the same types are preferred since it means the user had expressed an intention to perform this action despite what our might think.

Difficulties and Limitations

Now, we note some of the difficulties and limitations that we experienced with our program. As a reminder, the first step in our action decision process is to detect a user's face. We noticed that there is a limit to how well the initial face detection performs, for example, if the face is tilted upwards, downwards, left or right at an angle greater than 70 degrees, the detector would fail to track the face. This does make sense since dlib's face detection machine learning algorithm was trained on faces that are not tilted to those extremes. Additionally, moving the face too close to the camera that it occupies that majority of the frame will not allow a face to be detected, hence the justification for the bounding lines. For upwards motion, even if the face is detected, around 50 degrees, some facial landmarks will not be accurate such as the nose tip which the algorithm will detect incorrectly. Lastly, there was a variety of difficulties with blink detection and analyzing the user's eye. These difficulties were numerous and the details have been described in previous sections of this report.

Evaluation

We primarily focus on evaluating the accuracy of our detection methods. But we also have a qualitative survey on how people felt about using our program as a whole. Below we described quantitative and qualitative experiments, their results and a discussion on the results. Overall, we note quite decent accuracy and high perceived usefulness for our program.

Quantitative

For the quantitative part, we measured accuracy by having users execute an action some number of times and then the computer recorded a count of the number of times it detected that action. Accuracy is determined by taking the computer's count divided by the number of times the user actually executed that action. For example, if the user's head was at the center, they turned their head to the left, and then turned it back to the center this would be considered "left_turned" in our grammar. If the user completed "left_turned" 10 times but the program only counted 9 "left_turn"s then the accuracy would be 0.9. It is appropriate to measure accuracy the way we did since it is a method that has been used by others doing similar facial action detection. For example others in the opencv community who have attempted blink and wink detection using similar or the same facial landmark packages as us used the exact same method of recording computer's detection count. [13] Furthermore, our system's performance depends a lot on detection accuracy and it makes sense to focus on measuring that aspect as it appears to be the bottleneck as the moment to a successful launch to the general public.

We tested our system on 6 users and for each user we showed them at a high level what the program was analyzing, and allowed the user to try out the program and see the detection methods for about 5 min. Then we conducted our experiments doing 10 trials of detections at a time for each action. We used s version of our program which just focused on detecting actions. In other words for this experiment we omitted the functionality to control the computer since we were just focused on detecting accuracy.

Results

[13] https://www.codesofinterest.com/2017/06/wink-detection-using-dlib-and-opencv.html

action	user 1	user 2	user 3	user 4	user 5	user 6
Left_turned	1	1	0.8	0.9	0.8	1
Right_turned	1	1	0.9	0.9	0.8	1
Up_nod	1	0.6	0.4	0.8	0.7	0.4
Down_nod	1	0.8	0.2	1	0.8	0.7
Zoomed	0.9	1	0.9	1	0.9	1
Both_blink	0.9	0.4	0.6	0.7	0.8	0.6
Left_wink	0.9	0.6	0.3	0.9	0.9	0.9
Right_wink	0.9	0.5	0.5	0.9	1	1

Discussion

We note that on average our left and right turn detection method had the highest accuracies. Based on what was discussed in the Difficulties and Limitations section, this makes sense since left and right turn both had the largest allowable range of motion. Additionally, the method for detecting left and right turn relied on facial landmarks that were some of the most reliable to detect(i.e. nose tip and leftmost and rightmost point on the head).

Up and down nods had varying accuracy but there exists a correlation between users who did well at up nod and did well at down nod. This indicates that some users just understood the motion better than others rather than it being an absolute indication of varying detection accuracy. Based on what was discussed in the Difficulties and Limitation sections, it makes sense that some users would have a bit more trouble with down and up no detection since they both had lower allowable ranges of head movement. Zoom detection did the best overall, and this makes sense because our method was simple yet robust, simply checking the width of the head and comparing it to the width of the frame. The width of the head was calculated based on very stable landmarks (i.e. the leftmost and rightmost points on the head).

Lastly, wink and blink detection had varying accuracy similar to the effect in up and down detection where one can see the users who scored low, scored low for all types of winking and blinking, but the users who scored high, scored high among the different eye detections. This indicates that our wink and blink detection does work quite well, but possibly more so for users who either have certain physical features (i.e. eyelashes to too long) or who have more familiarity / better ability to blink “correctly” in general. User’s who scored low for winking had a tendency to close the open eye more than what would have been ideal.

Overall, our results show quite decent accuracy (in the range of 0.8 or better across all detections). Going forward one way to increase accuracy would be to improve domain engineering, "train" users for a bit longer.

Qualitative

As for our qualitative results, we asked the same 6 users to answer the following questions after they had run our full program as described in the Program Description and Manual section. We asked them to circle how much they disagreed or agreed with the following statements on a Likert scale of 1 to 5. Ultimately, we aim to capture what users feel about our program. The method we chose of measuring that is an appropriate way of gaining that insight since it is a quick and standardized method of understanding sentiment. Likert scales are a commonly used psychometric scale used in questionnaires.

1. This program would be useful to an able-bodied and competent computer user.
    a. Disagree 1 2 3 4 5 Agree
2. This program would be useful to an able-bodied, but lower-skilled computer user.
    a. Disagree 1 2 3 4 5 Agree
3. This program would be useful to someone who has a disability, which prevents them from having full functionality in their hands.
    a. Disagree 1 2 3 4 5 Agree

Results

question	user 1	user 2	user 3	user 4	user 5	user 6
Q1	5	5	3	4	2	4
Q2	3	5	4	5	4	5
Q3	4	5	5	5	4	4

Discussion

One can see that majority of responses were rated 4 or 5 which indicates a general trend of perceived usefulness for our program. As hypothesized, the program was perceived as being most useful for individuals who have a disability which prevents them from having full

functionality in their hands. The program was perceived as somewhat less useful for able-bodied capable users relative to those with a disability, and this makes sense, since by default most people execute the commands in our program with their hands.

Reflection

Overall we learned a great deal about detection using facial landmark, user studies, and what aspects of programs like ours need the most attention. To start, we learned that some tasks are a lot harder than others to detect given facial landmarks. The left, right, up, down, and zoom were quite easy to detect since the facial landmarks worked well for detecting critical points. On the other hand, detecting winks was over tens times as difficult to detect, and the accuracy varied among people. For example, wink detection worked extremely well on Howon, but not so well on me because my eyelashes are long and would create false positives. One may not have thought that different eyelash length could make such a huge difference, but it turns out that it does according to our eye blink algorithm.

The other major reflection on this project is about user tests. It was a challenge explaining different motions to the user, and making them understand what qualified as an appropriate motion or appropriate response from the computer. We needed to spend some initial time explaining the project, as well as our instructions i.e. “please do left turn 10 times.”

Lastly, I learned that detection accuracy might very well be the most important aspect of a project like this. Going into the project we underestimated the chance of poor accuracy and wanted our evaluation to be focused on how much this program could improve speed of certain computer tasks. However, prof. Kender’s intuition was correct in that detection accuracy and user unfamiliarity with the program would likely prevent accurate speed measurements in the way we originally intended. In the end, high detection accuracy turned out to be the most important aspect of our project.

Sources and Citations

[1] A. Chaudhary and J. L. Raheja, “Intelligent Approaches to interact with Machines using Hand Gesture Recognition in Natural way: A Survey,” arXiv:1303.2292 [cs], Mar. 2013.

[2] Y. Song, Y. Luo, and J. Lin, ‘‘Detection of movements of head and mouth to provide computer access for disabled,’’ in Proc. Int. Conf. TAAI, 2011.

[3] T. Soukupova and J. Cech, "Real-Time Eye Blink Detection using Facial Landmarks," *21st computer Vision Workshop*, Rinmske Toplice, Slovenia, 2016.

[4] https://opencv.org/

[5] http://dlib.net/

[6] http://www.numpy.org/

[7] http://pyautogui.readthedocs.io/en/latest/

[8] https://www.intel.com/content/www/us/en/architecture-and-technology/realsense-overview.html

[9] https://www.amazon.com/USB-Foot-Switch-Keyboard-Pedal/dp/B008MU0TBU

[10] https://www.spectacleapp.com/

[11] https://www.pyimagesearch.com/2017/04/10/detect-eyes-nose-lips-jaw-dlib-opencv-python/

[12] https://www.pyimagesearch.com/2017/04/24/eye-blink-detection-opencv-python-dlib/

[13] https://docs.opencv.org/2.4/doc/tutorials/imgproc/erosion_dilatation/erosion_dilatation.html

Demo

1. Brief explanation of our project.
2. Launch our app with macros disabled to show its detection capabilities.
    a. Focus on the top-left corner to show various stages of eye transformation.
    b. Show how the lines move in relation to the head.
3. Run with macros enabled.
    a. Eye-based gestures only.
        i. Snap left, snap right, both eyes.
    b. Head-based gestures only.
        i. Nods up and down, turn left and right.
        ii. Zoom in and out.
    c. Compound gestures utilizing both eyes and head.
4. Let the tests try it with macros enabled.

Code listing and Organization

Github repo: https://github.com/Howon/kender

action.py

```
from
enum
import
Enum
```

```
    _HEAD_FRAMES = 2
    _EYE_FRAMES = 3


    class HeadAction(Enum):
        LEFT = 0
        RIGHT = 1
        UP = 2
        DOWN = 3
        CENTER = 4
        ZOOM = 5
```

```
    UNZOOM = 6


class EyeAction(Enum):
    BOTH_OPEN = 0
    BOTH_CLOSED = 1
    BOTH_BLINK = 2
    LEFT_WINK = 5
    LEFT_CLOSED = 6
    RIGHT_WINK = 3
    RIGHT_CLOSED = 4


HEAD_REST_STATE = HeadAction.CENTER
EYE_REST_STATE = EyeAction.BOTH_OPEN


success = {
    EyeAction.LEFT_CLOSED: ([EyeAction.BOTH_OPEN], EyeAction.LEFT_WINK),
    EyeAction.RIGHT_CLOSED: ([EyeAction.BOTH_OPEN], EyeAction.RIGHT_WINK),
    EyeAction.BOTH_CLOSED: ([EyeAction.BOTH_OPEN], EyeAction.BOTH_BLINK),
    HeadAction.LEFT: ([HeadAction.CENTER, HeadAction.ZOOM], HeadAction.LEFT),
    HeadAction.RIGHT: ([HeadAction.CENTER, HeadAction.ZOOM], HeadAction.RIGHT),
    HeadAction.UP: ([HeadAction.CENTER, HeadAction.ZOOM], HeadAction.UP),
    HeadAction.DOWN: ([HeadAction.CENTER, HeadAction.ZOOM], HeadAction.DOWN),
    HeadAction.CENTER: ([HeadAction.ZOOM], HeadAction.ZOOM),
    HeadAction.ZOOM: ([HeadAction.CENTER], HeadAction.UNZOOM)
}


h_type = type(HEAD_REST_STATE)


class ActionHandler():
    def __init__(self):
        self.__awake = False
        self.__prev_head = HEAD_REST_STATE
        self.__prev_eye = EYE_REST_STATE


        self.__eye_count = 0
        self.__head_count = 0
```

```
    self.__zoomed = False

    return

def __consec(self, prev, action, count):
    if prev != action:
        a_type = type(action)

        if count >= (_HEAD_FRAMES if a_type == h_type else _EYE_FRAMES):
            return True, 0

    return False, count + 1

def __next_state(self, prev, now):
    if prev not in success:
        return False, None

    req, action = success[prev]

    if now not in req:
        return False, None

    if action == HeadAction.UNZOOM:

        print("UNZOOM", self.__zoomed)
        # Don't unzoom if not zoomed.
        if not self.__zoomed:
            return False, None

        self.__zoomed = False
```

```
        if action == HeadAction.ZOOM:
            # Don't zoom twice.
            if self.__zoomed:
                return False, None
            self.__zoomed = True


        return True, action


    def get_next(self, e_action, h_action):
        e_prev, h_prev = self.__prev_eye, self.__prev_head


        if h_prev == HEAD_REST_STATE:
            e_consec, e_count = self.__consec(e_prev, e_action, self.__eye_count)
        else:
            e_consec, e_count = False, 0


        h_consec, h_count = self.__consec(h_prev, h_action, self.__head_count)


        self.__prev_eye = e_action
        self.__prev_head = h_action


        self.__eye_count = e_count
        self.__head_count = h_count


        if e_consec and h_consec or (not e_consec and h_consec):
            self.__eye_count = 0
            return self.__next_state(h_prev, h_action)
        elif e_consec:
            self.__head_count = 0
            return self.__next_state(e_prev, e_action)


        return False, None
```

Capture.py

```
import
datetime
            import cv2
            import dlib
            import math
            import time


            from imutils import face_utils


            from head import Head
            from eyes import Eyes
            from display import *
            from utils import COUNTER_LOG, put_text, resize_frame
            from detection import detect_head, detect_eyes
            from action import ActionHandler, HEAD_REST_STATE


            CALIBRATE = True


            def capture_action(pred_path, cb, debug=False, log=False):
                """Facial detection and real time processing credit goes to:
                https://www.pyimagesearch.com/2017/04/17/real-time-facial-landmark-detection-
            opencv-python-dlib/
                """
                action_handler = ActionHandler()


                detector = dlib.get_frontal_face_detector()
                predictor = dlib.shape_predictor(pred_path)


                camera = cv2.VideoCapture(0)


                while True:
                    _, frame = camera.read()
                    if frame is None:
                        continue
```

```
_, w_original, _ = frame.shape
frame = resize_frame(frame)


h, w, _ = frame.shape


display_bounds(frame)


gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
rects = detector(gray, 0)


for rect in rects:
    shape = predictor(gray, rect)
    shape = face_utils.shape_to_np(shape)


    cur_head = Head(shape, w)
    cur_eyes = Eyes(shape, frame)


    eye_action = detect_eyes(shape, cur_eyes)
    head_action = detect_head(shape, cur_head)


    COUNTER_LOG[eye_action] += 1
    COUNTER_LOG[head_action] += 1


    perform, action = action_handler.get_next(eye_action, head_action)


    if log:
        display_decisions(frame, head_action, eye_action)
        display_counters(frame, COUNTER_LOG)


    if perform:
        COUNTER_LOG[action] += 1
        cb(action)
```

```
                if debug:
                    cur_head.debug(frame)
                    cur_eyes.debug(frame)


                    cv2.imshow("Frame", frame)


            key = cv2.waitKey(1) & 0xFF


            if key == ord("q"):
                break


        cv2.destroyAllWindows()
```

detection.py

```
# import
the
necessary
packages
        import cv2
        import math
        import statistics
        import numpy as np


        from utils import *


        from action import HeadAction, EyeAction
        from scipy.spatial import ConvexHull


        DEBUG_HEAD = True
        DEBUG_EYES = True
```

```
""" Decision Thresholds """
# eventually might want to make these calibrated or dynamic according to face
ratio
# rather than absolute values regarding the frame dimensions
# to make detection more sensitive:


# returns what position the head is in
def detect_head(shape, cur_head):
    head_state = HeadAction.CENTER


    if cur_head.turned_left():
        head_state = HeadAction.LEFT
    elif cur_head.turned_right():
        head_state = HeadAction.RIGHT
    elif cur_head.zoom():
        head_state = HeadAction.ZOOM
    elif cur_head.turned_up():
        head_state = HeadAction.UP
    elif cur_head.turned_down():
        head_state = HeadAction.DOWN


    return head_state


def detect_eyes(shape, cur_eyes):
    eye_action = EyeAction.BOTH_OPEN


    if cur_eyes.leaning():
        return EyeAction.BOTH_OPEN


    if cur_eyes.both_closed():
        return EyeAction.BOTH_CLOSED


    if cur_eyes.left_blink():
        return EyeAction.LEFT_CLOSED
```

```
            if cur_eyes.right_blink():
                return EyeAction.RIGHT_CLOSED


            return EyeAction.BOTH_OPEN
```

display.py

```
import
cv2
        from utils import put_text
        from action import HeadAction, EyeAction


        def display_bounds(frame):
            h, w, _ = frame.shape


            xs = [int(w / 4), int(3 * w / 4)]
            ys = [int(1 * h / 9), int(7.8 * h / 9)]


            for y in ys:
                [left, right] = xs
                cv2.line(frame, (left, y), (right, y), (255, 255, 255), 2)


        def display_decisions(frame, head_action, eye_action):
            h, w, _ = frame.shape
            print("================================")
            align_x, align_y = int(w * 0.05), int(h * 0.8)
            print(head_action)
            print(eye_action)


            put_text(frame, str(head_action)[11:],
               (align_x, align_y + 30), scale=1.5, thickness=2)
            put_text(frame, str(eye_action)[10:],
               (align_x, align_y + 3 * 30), scale=1.5, thickness=2)


        def display_counters(frame, counter_logs):
```

```
            h, w, _ = frame.shape
            align_x = int(w * 0.63)
            align_y = int(h * 0.3)
            for i, (k, v) in enumerate(counter_logs.items()):
                text = "{}: {}".format(k, v).ljust(11).lower()
                put_text(frame, text, (align_x, 15 * (i + 1) + align_y))
```

eyes.py

```
import
cv2
        from utils import *


        _EYE_HIST_THRESH = 0.02
        _EYE_DIFF_THRESH = 1.7


        _EYE_RECT_MODIFIER = 7


        _LEAN_THRESH = 10
        _C_FLOOR = 70
        _ELLIPSE_SCALE = 0.46
        _MASK_THICKNESS = 15
        _DILATE_KERNEL = cv2.getStructuringElement(cv2.MORPH_RECT, (5, 5))


        class Eyes():
            """Eye Status Detection
            For all eye functions the distance between the eyelid and the bottom of
            the eye for each eye is compared to a threshold
            source: http://vision.fe.uni-lj.si/cvww2016/proceedings/papers/05.pdf
            """
            def __init__(self, shape, frame):
                """
                Eye indices:
                        *37 *38              *43 *44
                    *36         *39      *42         *45
                        *41 *40              *47 *46
                """
```

```
        left_eye = [shape_coord(shape, i) for i in range(36, 42)]
        right_eye = [shape_coord(shape, i + 6) for i in range(36, 42)]


        self.__left_eye = shape[36:42]
        self.__right_eye = shape[42:48]


        self.__l_rect, l_disp = self.__find_eye_roi(self.__left_eye, frame)
        self.__r_rect, r_disp = self.__find_eye_roi(self.__right_eye, frame)


        self.__l_disp = l_disp
        self.__r_disp = r_disp
        self.__l_hist = self.__check_hist(self.__l_rect[2])
        self.__r_hist = self.__check_hist(self.__r_rect[2])


        self.__l_closed = self.__is_closed(self.__l_hist, self.__r_hist)
        self.__r_closed = self.__is_closed(self.__r_hist, self.__l_hist)


    def __mask_eyelash(self, eye, ellipse):
        eye_cpy = eye.copy()
        center, size, angle = ellipse


        c = (int(center[0]), int(center[1]))
        axes = (int(2.2 * _ELLIPSE_SCALE * size[0]),
                int(0.7 * _ELLIPSE_SCALE * size[1]))
        angle = int(angle)


        stencil = np.zeros(eye_cpy.shape)
        cv2.ellipse(stencil, ellipse, 255, -1)
        cv2.ellipse(eye_cpy, c, axes, angle, 0, 360, 255, _MASK_THICKNESS)
        cv2.ellipse(eye_cpy, c, axes, angle, 0, 360, 255, _MASK_THICKNESS)


        eye_cpy[stencil == 0] = 255


        return eye_cpy
```

```
def __find_eye_roi(self, eye_points, frame):
    """Finds the ROI for eye action parsing.
    Args:
        eye_points:
               * *
            *       *
               * *
        frame: Current captured frame.
    Returns:
        1. Top left and bottom right coordinates for the rectangle that
           encapsulates the eye coordinates.
            -----------
            |   * *   |
            |*       *|
            |   * *   |
            -----------
        2. Thresholded Eye.
        3. Various stages of the eye transformation stacked horizontally.
    """
    tlx = eye_points[0][0] - _EYE_RECT_MODIFIER
    tly = eye_points[1][1] - _EYE_RECT_MODIFIER


    brx = eye_points[3][0] + _EYE_RECT_MODIFIER
    bry = eye_points[4][1] + _EYE_RECT_MODIFIER


    tl, br = (tlx, tly), (brx, bry)


    rel_eye_points = np.asarray([(x - tlx, y - tly) for x, y in eye_points])


    try:
        eye = frame[tly:bry, tlx:brx].copy()
        h, w,_ = eye.shape


        eye_copy = eye.copy()
```

```
            gray_eye = cv2.cvtColor(eye.copy(), cv2.COLOR_BGR2GRAY)
            _, thresh = cv2.threshold(gray_eye, _C_FLOOR, 255,
                                      cv2.THRESH_BINARY)

            gray_eye_disp = cv2.cvtColor(gray_eye, cv2.COLOR_GRAY2BGR)
            thresh_disp = cv2.cvtColor(thresh, cv2.COLOR_GRAY2BGR)

            ellipse = cv2.fitEllipse(rel_eye_points)

            thresh = self.__mask_eyelash(thresh, ellipse)

            no_eyelash = cv2.cvtColor(thresh.copy(), cv2.COLOR_GRAY2BGR)
            dilation = cv2.dilate(thresh, _DILATE_KERNEL, iterations=1)
            thresh = cv2.erode(dilation, _DILATE_KERNEL, iterations=1)
            cv2.GaussianBlur(thresh, (3, 3), 0)

            eroded = cv2.cvtColor(thresh.copy(), cv2.COLOR_GRAY2BGR)

            disp = np.hstack((eye_copy, gray_eye_disp, thresh_disp,
                              no_eyelash, eroded))

            return (tl, br, thresh), disp
        except:
            pass

        return (tl, br, None), None

    def __check_hist(self, eye):
        if eye is None:
            return 1

        h, w = eye.shape
        return 1 - (len(eye.nonzero()[0]) / (h * w))
```

```
def __is_closed(self, hist, other):
    return _EYE_DIFF_THRESH * hist < other or hist < _EYE_HIST_THRESH


def leaning(self):
    l_tl, l_br, __ = self.__l_rect
    r_tl, r_br, __ = self.__r_rect


    ly = l_br[1] - 0.5 * l_tl[1]
    ry = r_br[1] - 0.5 * r_tl[1]


    return abs(ly - ry) > _LEAN_THRESH


def left_blink(self):
    return self.__l_closed


def right_blink(self):
    return self.__r_closed


def both_closed(self):
    return self.__l_closed and self.__r_closed


def debug(self, frame):
    h, w, _ = frame.shape


    align_x = int(w * 0.63)
    align_y = int(h * 0.1)


    l_tl, l_br = self.__l_rect[0:2]
    r_tl, r_br = self.__r_rect[0:2]


    cv2.rectangle(frame, l_tl, l_br, (0, 255 ,0), 2)
```

```
                cv2.rectangle(frame, r_tl, r_br, (0, 255, 0), 2)


                l_cnt = cv2.convexHull(self.__left_eye)
                r_cnt = cv2.convexHull(self.__right_eye)


                cv2.drawContours(frame, [l_cnt], -1, (0, 255, 0), 2)
                cv2.drawContours(frame, [r_cnt], -1, (0, 255, 0), 2)


                put_text(frame, "LEFT HIST: " + str(round(self.__l_hist, 2)),
                         (align_x, align_y + 20))
                put_text(frame, "RIGHT HIST: " + str(round(self.__r_hist, 2)),
                         (align_x, align_y + 2 * 20))


                l_disp_y = 0
                if self.__l_disp is not None:
                    l_disp_y, l_disp_x, _ = self.__l_disp.shape


                    if l_disp_x < w:
                        frame[0:l_disp_y, 0:l_disp_x] = self.__l_disp


                if self.__r_disp is not None:
                    r_disp_y, r_disp_x, _ = self.__r_disp.shape
                    if l_disp_y >= w:
                        return
                    frame[l_disp_y:(l_disp_y + r_disp_y), 0:r_disp_x] = self.__r_disp
```

head.py

```
import
cv2
        import math
        from utils import *


        UP_THRESH = 0.15      # Decrease to make it more sensitive.
        DOWN_THRESH = 0.5    # Decrease to make it more sensitive.
```

```
ZOOM_THRESH = 0.5    # Decrease to make it more sensitive.


""" Drawing functions """
# draws the threshold line for left turn
def draw_left_line(frame, left_midpoint):
    ptA = ((left_midpoint[0]), left_midpoint[1] + 50)
    ptB = ((left_midpoint[0]), left_midpoint[1] - 50)
    cv2.line(frame, ptA, ptB, (255, 255, 255), 2)


    return frame


# draws the threshold line for right turn
def draw_right_line(frame, right_midpoint):
    ptA = ((right_midpoint[0]), right_midpoint[1] + 50)
    ptB = ((right_midpoint[0]), right_midpoint[1] - 50)
    cv2.line(frame, ptA, ptB, (255, 255, 255), 2)


    return frame


class Head():
    """Head Position Detection
    https://www.pyimagesearch.com/2017/04/17/real-time-facial-landmark-detection-
opencv-python-dlib/
    """


    def __init__(self, shape, width):
        """
        Left Ear Index: 1.
        Right Ear Index: 16.
        Chin Index: 9.
        Nose Index: 30.
        Rightmost point in left eye Index: 39
        Leftmost point in left eye Index: 42
        """


        self.__left_ear = shape_coord(shape, 1)
```

```
        self.__right_ear = shape_coord(shape, 16)

        self.__chin = shape_coord(shape, 8)
        self.__nose_tip = shape_coord(shape, 30)
        self.__head = midpoint(self.__left_ear, self.__right_ear)

        # Used in Down detection
        # Get the vertical distance between the center of the head and the nose tip
        # Get the vertical distance between the center of the head and the chin
        # Take the ratio of those two distances
        nose_head_dist = dist(self.__nose_tip, self.__head)
        nose_chin_dist = dist(self.__nose_tip, self.__chin)

        self.__nc_ratio = round(nose_head_dist / nose_chin_dist, 2)

        # Used in Up detection
        # compure ratio of head width over frame width
        self.__frame_width = width
        self.__head_zoom_ratio = dist(self.__left_ear, self.__right_ear) / width

    def turned_left(self):
        """Detects if the head is turned left.
        Compares the nose point against the midpoint
        of the leftmost head point and head center
        """
        nx, _ = self.__nose_tip
        mx, _ = midpoint(self.__left_ear, self.__head)

        return nx < mx

    def turned_right(self):
        """Detects if the head is turned right.
        Compares the nose point against the midpoint
        of the rightmost head point and head center
        """
        nx, _ = self.__nose_tip
```

```
        mx, _ = midpoint(self.__right_ear, self.__head)

        return nx > mx

    def turned_up(self):
        """Detects if the head is nodding up.
        Checks if nose tip is above center of head
        Checks if head's features have certain ratio implying up
        """
        _, ny = self.__nose_tip
        _, hy = self.__head

        return ny < hy and self.__nc_ratio > UP_THRESH

    def turned_down(self):
        """Detects if the head is nodding up.
        Checks if nose tip is below center of head
        Checks if head's features have certain ratio implying down
        """
        _, hy = self.__head
        _, ny = self.__nose_tip
        _, cy = self.__chin

        return hy < ny and ny < cy and self.__nc_ratio > DOWN_THRESH

    def zoom(self):
        """Determines if the head is zommed in or not.
        Compares one's head width / frame width ratio to threshold
        """
        return self.__head_zoom_ratio > ZOOM_THRESH

    def debug(self, frame):
        h, w, _ = frame.shape
        align_x = int(w * 0.63)
        align_y = int(h * 0.2)
```

```
        # Important features
        cv2.circle(frame, self.__nose_tip, 3, 255, -1)
        cv2.circle(frame, self.__head, 3, 255, -1)
        cv2.circle(frame, self.__chin, 3, 255, -1)


        # Left debugging
        frame = draw_left_line(frame, midpoint(self.__left_ear, self.__head))
        frame = draw_right_line(frame, midpoint(self.__right_ear, self.__head))


        nh_color = (255, 0, 0)


        if self.turned_up() or self.turned_down():
            nh_color = (255, 0, 255)


        cv2.line(frame, self.__head, self.__nose_tip, nh_color, 2)
        cv2.line(frame, self.__nose_tip, self.__chin, (255, 0, 0), 2)


        # Down debugging.
        put_text(frame, "DOWN: " + str(self.__nc_ratio),
                 (align_x, align_y + 20))


        # Up debugging.
        put_text(frame, "UP: " + str(round(self.__nc_ratio, 2)),
                 (align_x, align_y + 40))


        # Zoom debugging.
        put_text(frame, "ZOOM: " + str(round(self.__head_zoom_ratio, 2)),
                 (align_x, align_y + 3 * 20))
        cv2.line(frame, self.__left_ear, self.__right_ear, (255, 0, 255), 3)
```

macro.py

```
import
os
```

```
import json
import time
import pyautogui


from enum import Enum
from action import HeadAction, EyeAction


_SLEEP_DURATION = 0.2


class Macro(Enum):
    MOVE_LEFT = 0
    MOVE_RIGHT = 1
    FULLSCREEN = 2
    EXPOSE = 3
    MINIMIZE = 4
    TAB_FORWARD = 5
    TAB_BACKWARD = 6
    COPY = 7
    PASTE = 8
    NO_ACTION = 9


__ATM = {
    HeadAction.LEFT: Macro.TAB_BACKWARD,
    HeadAction.RIGHT: Macro.TAB_FORWARD,
    HeadAction.UP: Macro.COPY,
    HeadAction.DOWN: Macro.PASTE,
    HeadAction.ZOOM: Macro.EXPOSE,
    HeadAction.UNZOOM: Macro.EXPOSE,
    EyeAction.LEFT_WINK: Macro.MOVE_LEFT,
    EyeAction.RIGHT_WINK: Macro.MOVE_RIGHT,
    EyeAction.BOTH_BLINK: Macro.FULLSCREEN,
}


def translate_action(action):
    return __ATM[action] if action in __ATM else Macro.NO_ACTION


class MacroHandler():
```

```
__macros = {
    Macro.MOVE_LEFT: [],
    Macro.MOVE_RIGHT: [],
    Macro.EXPOSE: [],
    Macro.MINIMIZE: [],
    Macro.TAB_FORWARD: [],
    Macro.TAB_BACKWARD: [],
    Macro.COPY: [],
    Macro.PASTE: [],
}


def __init__(self, config_file):
    """Loads predefined Spectacle commands and populates __macros dictionary.
    """
    with open(os.path.expanduser(config_file)) as json_data:
        commands = json.load(json_data)
        for c in commands:
            name, binding = c['shortcut_name'], c['shortcut_key_binding']
            if binding is not None:
                binding = binding.replace("cmd", "command")
                binding = binding.split('+')


            if name == 'MoveToLeftHalf':
                self.__macros[Macro.MOVE_LEFT] = binding
            elif name == 'MoveToRightHalf':
                self.__macros[Macro.MOVE_RIGHT] = binding
            elif name  == 'MoveToFullscreen':
                self.__macros[Macro.FULLSCREEN] = binding


    self.__macros[Macro.EXPOSE] = ["ctrl", "up"]
    self.__macros[Macro.MINIMIZE] = ["command", "down"]
    self.__macros[Macro.TAB_FORWARD] = ["ctrl", "right"]
    self.__macros[Macro.TAB_BACKWARD] = ["ctrl", "left"]
    self.__macros[Macro.COPY] = ["command", "c"]
    self.__macros[Macro.PASTE] = ["command", "v"]


def execute(self, macro):
    if macro == Macro.NO_ACTION:
        return
```

```
            hotkeys = self.__macros[macro]

            pyautogui.hotkey("ctrl")

            pyautogui.hotkey(*hotkeys)
            time.sleep(_SLEEP_DURATION)
```

main.py

```
import
argparse

        from capture import capture_action
        from macro import MacroHandler, translate_action

        _MACROS = '~/Library/Application Support/Spectacle/Shortcuts.json'

        ap = argparse.ArgumentParser()
        ap.add_argument("-m", "--macros", action="store_true", required=False,
                        help="Enables macros.")

        ap.add_argument("-p", "--shape-predictor", required=True,
                        help="Path to facial landmark predictor")
        ap.add_argument("-d", "--debug", action="store_true", required=False,
                        help="Displays debugging information.")
        ap.add_argument("-l", "--log", action="store_true", required=False,
                        help="Displays logging information.")
        args = vars(ap.parse_args())



        def main():
            pred_path = args["shape_predictor"]
```

```
        enable_macros = args["macros"]
        macro_handler = MacroHandler(_MACROS)


        def trigger_macro(action):
            nonlocal enable_macros


            if enable_macros:
                macro = translate_action(action)
                macro_handler.execute(macro)


        capture_action(pred_path, trigger_macro, args["debug"], args["log"])


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

test.py

```
import
time
        import argparse


        from random import shuffle
        from action import HeadAction, success
        from capture import capture_action


        COMMANDS_LOG = "test_commands.txt"


        ap = argparse.ArgumentParser()
        ap.add_argument("-p", "--shape-predictor", required=True,
                        help="Path to facial landmark predictor")
        ap.add_argument("-d", "--debug", action="store_true", required=False,
                        help="Displays debugging information.")


        args = vars(ap.parse_args())
```

```
def load_test():
    test_actions = [v for _, v in success.values() if v != HeadAction.UNZOOM]


    test_cases = test_actions * 10
    shuffle(test_cases)


    return test_cases


def main():
    global COMMANDS_LOG


    test_cases = load_test()


    with open(COMMANDS_LOG, "w+") as test_commands:
        for t in test_cases:
            test_commands.write(str(t) + "\n")


    print("Open {} in a new window and begin the test.".format(COMMANDS_LOG))
    print("Read each line to the subject and see how they perform.")


    time.sleep(5)


    def test(action):
        print("============\n{}".format(action))


    capture_action(args["shape_predictor"], test)


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

utils.py

```
import
cv2
        import math
        import numpy as np


        from collections import OrderedDict
        from action  import HeadAction, EyeAction


        __DEF_FONT = cv2.FONT_HERSHEY_SIMPLEX


        COUNTER_LOG = OrderedDict({
            HeadAction.LEFT: 0,
            HeadAction.RIGHT: 0,
            HeadAction.UP: 0,
            HeadAction.DOWN: 0,
            HeadAction.CENTER: 0,
            HeadAction.ZOOM: 0,
            HeadAction.UNZOOM: 0,
            EyeAction.BOTH_OPEN: 0,
            EyeAction.BOTH_BLINK: 0,
            EyeAction.BOTH_CLOSED: 0,
            EyeAction.LEFT_WINK: 0,
            EyeAction.LEFT_CLOSED: 0,
            EyeAction.RIGHT_WINK: 0,
            EyeAction.RIGHT_CLOSED: 0
        })


        def midpoint(p1, p2):
            """Midpoint between two points.
            Args:
                p1: Starting point.
                p2: Ending point.
            Returns:
                Midpoint between p1 and p2.
            """
```

```
    return (p1[0] + p2[0]) // 2, (p1[1] + p2[1]) // 2


def dist(p1, p2):
    """Euclidean distance between two points.
    Args:
        p1: Starting point.
        p2: Ending point.
    Returns:
        Euclidean Distance.
    """
    return math.sqrt((p2[0] - p1[0]) ** 2 + (p2[1] - p1[1]) ** 2)


def y_dist(p1, p2):
    """Euclidean distance between two points.
    Args:
        p1: Starting point.
        p2: Ending point.
    Returns:
        Absolute Y Distance.
    """
    return abs(p2[1] - p1[1])


# gets the point at a given index in the shape feature array of points
def shape_coord(shape, i):
    return shape[i][0], shape[i][1]


def put_text(frame, text, loc, scale=0.5, color=(255, 255, 255), thickness=1):
    cv2.putText(frame, text, loc, __DEF_FONT, scale, color, thickness)


def resize_frame(frame):
    h, w, _ = frame.shape


    x = frame.copy()
    right_removed = np.delete(x, range(3 * w // 4, w), axis=1)
    left_removed = np.delete(right_removed, range(0, w//4), axis=1)
```

```
            return cv2.flip(left_removed, 1)
```

run.sh

```
#!/bin/bash
             python3 main.py -p landmarks/shape_predictor_68_face_landmarks.dat $1 $2 $3
```