Gaze Direction Estimation

Matthew Hullstrung
https://github.com/matthullstrung/gaze-estimation

I. Inspiration

Over the summer, I worked on a project involving head pose estimation to track where a person was looking. However, the question constantly came up: what happens if someone's head is facing a different way than their eyes? This sparked my interest in gaze direction estimation. Gaze direction estimation would go one step further than head pose estimation, tracking someone's eyes as well as the direction their head was pointing. This would effectively solve the problem of a face pointing a different direction than the eyes are looking.

II. Eye Tracking Implementation

I understood the concept of head pose estimation, so the main problem at hand was how to track the eyes. Once we track the eyes, we can figure out how to adjust the head pose vector.

A. Blob Detection

The first solution to tracking the eyes was using blob detection. Using a face mesh from the dlib library, we were able to obtain a mesh with 68 landmarks. The dlib face mesh provides facial landmarks around the eyes, which allow us to get a general idea of where the eyes will be. Refer to *Figure 2.1* for a static 2D representation of the dlib face mesh.

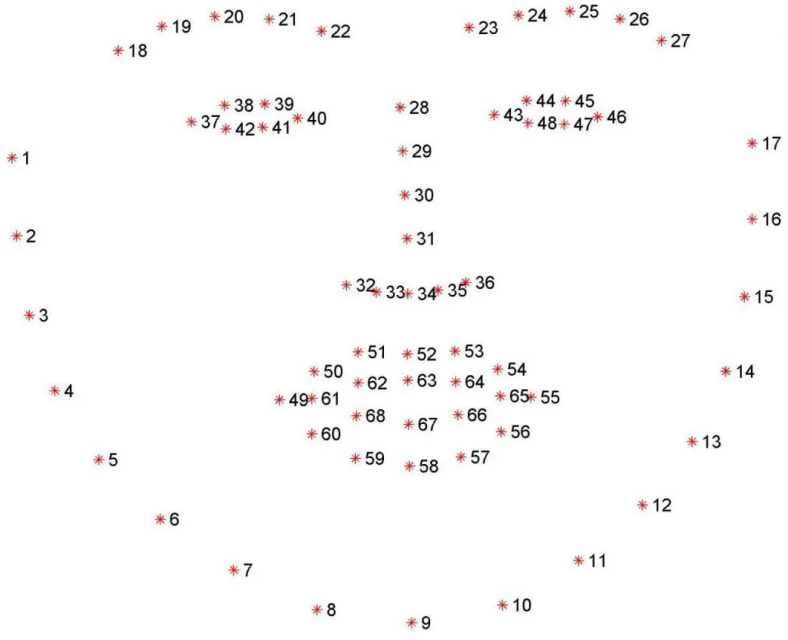

Figure 2.1

As shown in *Figure 2.1*, one can see that the left iris will be located exactly within the six points between [37, 42] and the right iris within points [43, 48]. These points each form a convex polygon for each eye with which we can create a mask using OpenCV's "fillConvexPoly()" function. Thresholding can then be applied on this mask to attempt to separate the iris. To keep the algorithm fairly robust, we can erode and dilate the image then apply a slight blur. The amount of thresholding must be determined by the user via a trackbar, since it will change for every environment. With this, we can then find the iris center using contours and moments. The mass center, or iris center in our case, will be $(\bar{x}, \bar{y}) = (\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}})$ in OpenCV's Moments structure.

a. Results

Tracking the iris center using blob detection worked fairly well, however there were some issues. The tracking was fairly inconsistent and not very robust. Part of this was at the fault of the dlib face mesh. At extreme angles, the dlib face mesh would become very inconsistent in its tracking, and as a result, the blob detection algorithm would break. In an attempt to build the robustness of the algorithm, I decided to switch the face mesh I used to Google's Mediapipe face mesh. The Mediapipe face mesh has hundreds of more points being tracked and has more consistent tracking than dlib.

B. Mediapipe Iris Detection

My initial plan was to completely revamp my previous code by introducing the Mediapipe face mesh for landmark detection. However, in the process of rewriting the code, I realized in the Mediapipe documentation that its face mesh (with some modifications) provided iris detection data. So, I then worked towards shaping the code to implement eye tracking using the iris landmarks provided by Mediapipe's face mesh.

a. Results

The results using Mediapipe's iris detection were fantastic. The algorithm was much more robust, allowing me to turn my head to much further extremes while maintaining consistent tracking than with the blob detection algorithm described above. Mediapipe's only issue with eye tracking was natural to computer vision: when the webcam was not able to see any features of my eyes, the detections would become erratic. This became a natural issue with wearing glasses. Since the camera could not

clearly see my eyes through the reflection of my glasses, tracking would become slightly inconsistent. However, when my eyes were clearly visible by the webcam, the eye tracking was greatly improved compared to the blob detection algorithm.

III. Head Pose Estimation

There are two ingredients to gaze direction estimation: head pose estimation and eye tracking. Without head pose estimation, the eye tracking would provide no value. So, head pose estimation is a crucial part to what we are attempting to achieve.

The main problem we are trying to solve in head pose estimation is a Perspective-n-Point (PNP) problem. Using a Mediapipe face mesh, we learn the 2D locations of several facial landmarks. If we create our own 3D coordinate system using a generic face model, we can get an estimate of the 3D coordinates that these 2D landmarks should correspond to. We will use these correspondences: right and left eye corners, right and left mouth corners, the chin, and the tip of the nose. Instead of having to calibrate the camera, we estimate the intrinsic parameters of the camera. This can be done by assuming the optical center is the center of the image, the focal length is equal to the width of the image, and that there is no radial distortion. A calibrated camera would be ideal, but for ease, this estimation should work fairly well.

With our 3D-2D point correspondences, estimated camera matrix, and distortion coefficients, we can now use OpenCV's "SolvePnP()" function. This function will return a rotation and a translation vector that transform a 3D point in our model coordinate system into the camera's coordinate system.

A. Results

I first used the dlib face mesh to compute the head pose estimation. The results with it were fairly good. However, the estimation would become very inconsistent at extreme angles when moving your head around. This was the same issue I initially had with the eye tracking using the dlib face mesh. To solve the issue, I changed the face mesh to the Mediapipe face mesh. The head pose estimation with the Mediapipe face mesh was very consistent, even at extreme head rotation angles.

IV. Gaze Direction Estimation

With consistent head pose estimation and eye tracking, we have the two ingredients to our gaze direction estimation. So, we must now combine them. Head pose estimation is represented as a rotation and a translation vector that transform a 3D point in our model coordinate system into the camera coordinate system. So, we can

modify the head's rotation vector with a gaze score we can calculate from our eye tracking.

A. Gaze Score

A gaze score can be calculated in both the x and y directions. We do this by creating a bounding box for each eye. The box is defined by the face mesh landmarks at each eye corner to get the width and landmarks above and below the eye to get the box's height. We get this for each eye. A demonstration of the box, where the "**+**" is the iris center is drawn in *Figures 3.1* and *3.2* below:

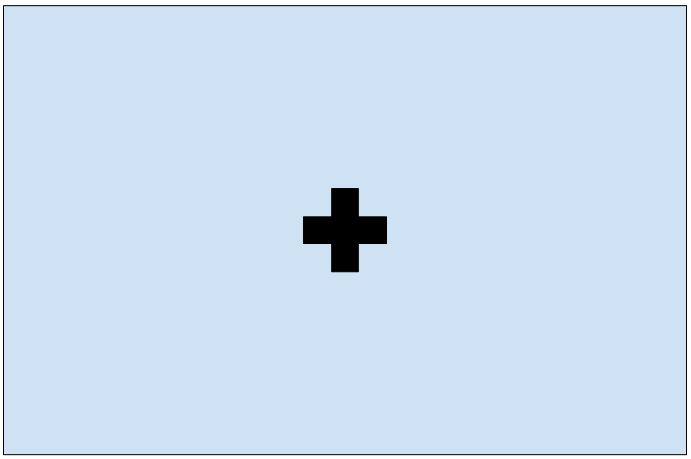

Figure 3.1, Looking straight ahead

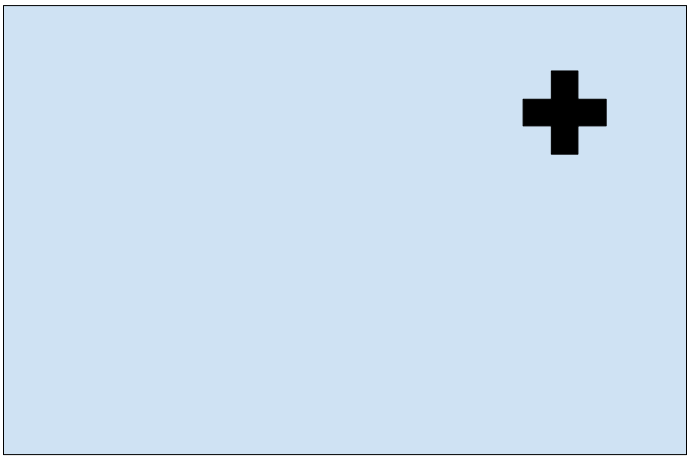

Figure 3.2, Looking up and to the right

With this bounding box, we can compute a ratio of where the iris is in relation to the box. Our gaze score in each direction will then range from 0 to 1, with values closer to 0 being closer to the left (x) or top (y) of the bounding box

and values closer to 1 being the opposite. Scores in each direction at 0.5 denote the iris is at the neutral position, in the center of the bounding box.

To make this gaze score useful, we must subtract both the x and y score by 0.5. This would mean scores at 0 will now denote no movement in the eyes, whereas a score of -0.5 would be looking all the way to the left and a score of 0.5 all the way to the right. Now, we can use this to modify our head pose rotation vector. I add the x-score to the yaw, and the y-score to the pitch of the head rotation vector. The rotation vector is a simplified representation of a rotation matrix. To amplify or reduce the effect the gaze score has on the head pose rotation vector, I created a multiplier that can be updated per session. I found that a multiplier of 4 tends to provide the most natural result.

B. Results

The results turned out to be pretty good. At first, there was a lot of jittering in the estimation. This jitter came from jumpy gaze score calculations. To reduce this jittering, if the score was within a certain threshold of the previous frame, average the scores of the current and last frames. This threshold can be adjusted by the user per session. This works because frames with scores that are close together can be considered close to, if not exactly, the same value. However, when adjusting the head pose rotation vector, tiny changes can go a long way in making it jitter. When the eye is not visible to the camera, we already knew that Mediapipe's iris detection became unstable. But, when implementing the full gaze direction estimation, this instability becomes very problematic. Further changes must be made to account for this instability, which I will describe in *Section IV*.

I recorded a video demonstration of the gaze direction estimation in full swing. It starts with a head pose estimation demonstration, then the following three clips are from two camera angles with two different gaze score multipliers. The second and third clip have a gaze score multiplier of 4, whereas the final clip has a gaze score of 10. The estimation becomes more unstable but more accurate in some scenarios.

VIDEO DEMONSTRATION LINK

https://youtu.be/BFOO-_9tMn4

IV. Further Work

This project is far from complete. There is much more work to be done on the algorithm itself, as well as many paths this project can take as further implementations.

A. Instability Issues

The gaze direction estimation currently has a lot of instability issues, especially when a person's eyes are not clearly visible. This is a common problem in computer vision: lack of information. However currently, Mediapipe tries to make random guesses as to where the iris is, even when the iris is not visible. This causes severe inconsistencies and jittering once the head turns far enough to where a person's eye can no longer be seen by the camera.

A possible solution to this would be keeping the iris landmarks at a default/still location with respect to the rest of the model if an accurate detection is not possible. If only one eye is able to be detected accurately, adjust the other eye's default location with respect to it. This would expand the range of head rotation while maintaining fairly accurate detection, since if one eye is not visible and cannot be accurately detected, it can just pull information from the other eye. In most cases, eye movements are not mutually exclusive. So, one eye can help the other if the other eye cannot be seen or make an accurate detection.

B. Further Implementations

There are several possible further implementations of this project. The most interesting one to me is the idea of using gaze direction estimation to control your mouse cursor. This would provide a valuable solution to people with disabilities preventing them from using a mouse, as well as anyone that would need to control a device without using their hands. To tie it back to the inspiration, another implementation of gaze direction estimation could be to track where someone is looking. This could be useful for tracking attention for a multitude of applications. One example of this would be a safety feature inside of a car to detect if a person has fallen asleep or has stopped looking at the road. Gaze direction estimation is a very powerful tool, and further implementation of my algorithm would allow one to achieve these powerful applications.