Project Report

A Real-time Eye Gaze Recognition System and Media Player

Deep Learning with Computer Vision

deeplearning.ai

by

Abhishek Tyagi
Roll No.: 42370211217
(Delhi Institute of Tool Engineering)

under the guidance of

Arun Gupta (HoD DITE)

Project title

A Real-time Eye Gaze Recognition System and Media Player

Author

Abhishek Tyagi

Project URL

https://github.com/AbhishekTyagi404/Deep-Learning-Python/tree/master/Gaze_Recognition_Media_Player

Project Video Result

https://youtu.be/DKxA6mvGz5w

Abstract

Face and eye detection are one of the most challenging problems in computer vision area. The goal of this paper is to present a study over the existing literature on face and eye detection and gaze estimation. With the uptrend of systems based on face and eye detection in many different areas of life in recent years, this subject has gained much more attention by academic and industrial area. Many different studies have been performed about face and eye detection. Besides having many challenging problems like, having different lighting conditions, having glasses, facial hair or mustache on face, different orientation pose or occlusion of face, face and eye detection methods performed great progress. In this paper we first categorize face detection models and examine the basic algorithms for face detection. Then we present methods for eye detection and gaze estimation.

Acknowledgement

Perseverance, inspiration and motivation have always played a key role in the success of any venture. At this level of understanding, it is often difficult to understand the wide spectrum of knowledge without proper guidance and advice. Hence, we take the opportunity to express our heartfelt gratitude to my mentor Professor Andrew Ng and Professor S. Majhi director cum principal and Assistant Professor Arun Gupta (HoD) for providing me with the opportunity to carry out the project. I will be obliged to thank Dr. Shalini Sharma for encouraging my ideas and giving me all the possible support.

Index

1. INTRODUCTION

As machines surround our world, interaction between human and machine plays a very important part of our quotidian lives. HCI has become a vast research field and in the past decade, more stress has been given to the user interface since lots of incipient technologies are made available to every human being. From the tangible user interface to touch and now the time is for gesture control. Even the gesture control has become a widespread research field all by itself. And all these researches are focused on one goal, to make user interactions with devices easier and more natural.

Usually, hands are the most used part of the human body to interact with objects naturally. So, a majority of gesture controls are based on hand gestures. Contrarily, if the kineticism of user's eyes are considered then it provides us usual, handy and high-frequency source of input, interaction and search both in conscious and subconscious state of our brain. There can be two major types of gaze tracking. One is estimating the gaze by tracking the head, other is estimating the gaze by tracking the eye movement. In such case tracking the head instead of eye gaze is not so efficient. Tracking the eye rather than the whole head can ensure where the user is looking more accurately. Keeping these concepts in mind we decided to develop a system for gaze-based user interaction.

In this paper, a system is developed to control the media player on PC utilizing eye gaze. The system would sanction the webcam of any PC to track the user in front of it and by tracking both the user's eye blink and the eye gaze it can
detect when to play, pause, move forward or rewind the media. While watching any video on screen we have to look at the screen anyways. Our motivation is, why use hands? Why not use the gaze to control the media player while watching the video? That's where this system comes in which will allow any user to experience a hands-free and more natural experience while watching media on PC.

Available eye center detection methods are: (1) feature- based methods (2) model-based methods and (3) hybrid methods. This research focuses on feature-based approach which tracks image edges, corners and structures from frame to frame. It is pursued in order to localize the center of the eye. This localization is a dynamic and accurate approach to locate and track center of the eye in images with low-resolution and videos taken with a webcam. This paper demonstrates a scheme consisting multiple layers which is ordinarily implemented for feature-based eye center localization. Preprocessing the localized eye center, we find the contour and detect the centroid to track for gaze estimation. Furthermore, the precision and the robustness are evaluated by inspecting transmutations in lighting, contrast, and background.

I. SYSTEM OVERVIEW AND METHODOLOGY

The system takes the feed from the webcam to track the user's eyes and updates the gaze position. Our system is designed to control basic media player functionalities using eye gaze of the user. The workflow starts with capturing live feed from the webcam. Then the eye ROI is being extracted to preprocess it and detect the iris. From the position of the iris center and screen mapping, the user's gaze can be determined. Then actions are performed based on gaze position on the screen and the user's dedicated eye blink. Overview of the workflow can be seen in the flowchart.

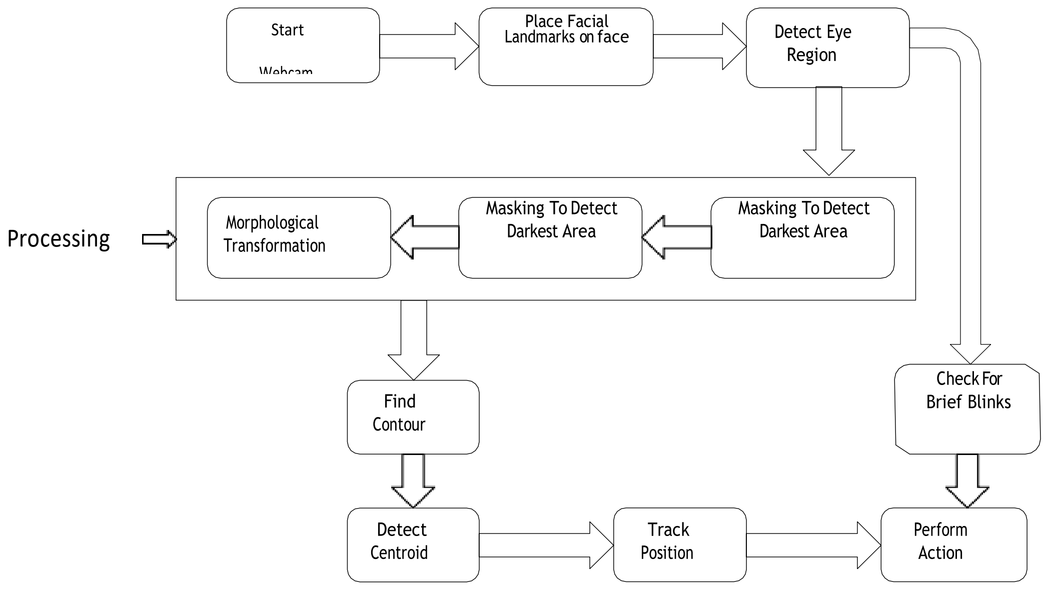

Start
Webcam
Place Facial Landmarks on face
Detect Eye Region
Processing
Morphological Transformation
Masking To Detect Darkest Area
Masking To Detect Darkest Area
Find Contour
Check For Brief Blinks
Detect Centroid
Track Position
Perform Action

Fig. 1. The framework of the developed system.

A. *Extracting the eye region:*

Detecting the eye region of the user from live webcam feed as well as to keep track of delicate blinks is performed. Considering different face and eye detection methods, we decided to use "Facial Landmark Model" for facial feature (in our case Eyes coordinate) extraction.

1) *Facial Landmarks:*

Facial Landmarks are used to pinpoint the significant facial facets in an image. It can localize and pinpoint significant reference points on the face which includes eyebrows, eyes, nose, lips, and jawline. Detecting these landmarks is a subdivision of shape prediction problem. Facial landmarks have many successful and important applications. Some of these applications include face alignment, head pose estimation, face swapping, blink detection etc.

There is two-step process for Facial landmark detection: Step 1: Localize the face in the image.

Step 2: Identify the key facial structures on the face ROI found in step 1.

We utilized the dlib's pre-trained facial landmark model which estimates 68 (x, y) coordinates which is placed on a face to point the major facial structures. Figure 2 will provide a clear idea of the 68 points the model locates from a face. Dlib library encompasses facial landmark detection which is proposed in the paper named “One millisecond face alignment with an ensemble of regression trees” by Kazemi and Sullivan (2014).

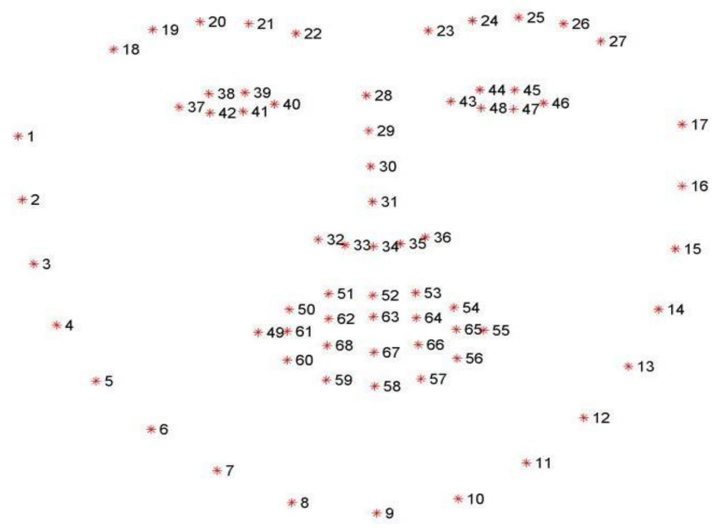


Fig. 2. Visualizing the 68 facial landmark coordinates from the iBUG 300- W dataset.

For the scope of our research, we only need the eyes. So, we took the coordinates from 37-40 and 43-48 respectively for the left and right eye. After that a bounding rectangle around the eye region was drawn. Rest of the work is done on this eye ROI.

B. *Iris Position Estimation:*

After extracting the eye region bounding box, the next step is to detect and track the iris or the darkest part of the eyes so that we can determine which way the eye is rotated or in other words in which direction the user is looking. We performed appropriate image processing on this region to extract the iris and then the iris center.

The webcam feed is converted to grayscale mode to work with single channel color space. After that histogram equalization is done on the frame to adjust and properly distribute the intensities for further processing. This returns the eye region with a better contrast. Then we go for extracting the iris region. As we are in single channel color space the intensities of iris would be considerably lower than the intensities of the white portion of a human eye. Here we decided to use color-based segmentation to extract the dark segment from the eye region. We have figured out a perfect threshold value to generate the image mask containing only the iris. The initial masked image is pretty noisy. So, we have used morphological transformation on the masked image to clear the noise. First, we have dilated the image to wear out the unwanted white regions or outliers. This process not only eliminates the noise but also shrinks the iris portion. So, we erode the image again to enhance the remaining white region which in this case is our iris. Now, this area is not always an ideal circle due to different lighting condition or head orientation. To get the center of this segment we find the contour of it and then use centroid moments to determine the centroid. Centroid coordinate can be calculated as,

$$C_x = M10/M00 \text{ and } C_y = M01/M00 \qquad (1)$$

Here M is the array of moments and (C_x, C_y) is the centroid position. Coordinate of this centroid is convenient approximation of the pupil of the user's eye.

C. *Gaze Estimation:*

Now having the coordinate of user's pupil, our task is to track the position of the centroid and map the position to screen coordinates so that the user's looking direction can be measured. For screen mapping the first coordinate is calibrated for user's eye. The user looks at these 9 points on the calibration window and the system notes down the coordinate of the user's pupil at each position. Then it calculates a ratio of eye movement to on-screen movements from these positions.

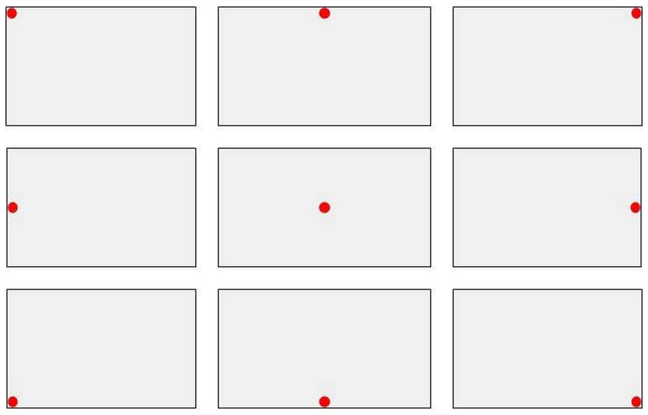
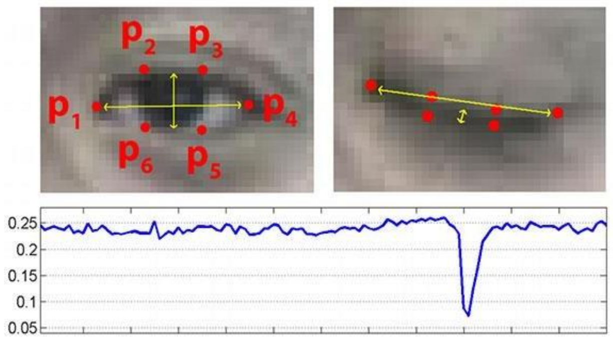


Fig. 3. Calibration Window

A. *Blink Detection:*

Blink is a natural action of eyes for human being. Until now the user's ROI on the screen is known to us. Our next task was to perform different actions based on where the user is looking. We have planned to achieve this using eye blink. All the 6 points achieved from facial landmark is used to calculate Eye Aspect Ratio (EAR). [22] [23] The formula of

EAR is given at equation 2.



EAR = | p2 − p6 | + | p3 − p5 | / 2| p1− p4 | (2)

EAR is calculated from only the distance between 6 landmark points shown in Fig 4 and determines when a user blinks from the ratio.

Fig. 4. 2 pictures on top visualizes eye landmarks while the eye is open and while eye is closed. Bottom picture is a graph which plots eye aspect ratio with respect to time. The sudden drop signifies a blink.

The figure 4 from Soukupová and Čech was considered for better understanding:

Setting up a threshold EAR and threshold number of frames we detected blinks as well as separated intentional blinks from natural ones. Then we programmed specific actions for blink while looking at specific parts of the screen.

• Eye Tracking Algorithm

The major steps in the algorithm are described below

1) *Noise Reduction*

Because the use of the low-cost hardware, the noise in captured video needs to be removed before analyzing the video.

2) *Corneal reflection detection, localization and removal:*

Before analysis, we also need to remove the corneal reflection, which is the brightest region in the image.

3) *Pupil contour detection*

After pupil contour detection, the software will get a set of candidate feature points on the pupil contour.

Input: Eye image with corneal reflection removed, Best guess of pupil center

Output: Set of feature points

Procedure:

Iterate

Stage 1:

a) Follow rays extending from the starting point

b) Calculate intensity derivative at each point

c) If derivative > threshold then

d) Place feature point

e) Halt marching along ray

Stage 2:

f) For each feature point detected in Stage 1

g) March along rays returning towards the start point

h) Calculate intensity derivative at each point

i) If derivative > threshold then

j) Place feature point

k) Halt marching along ray

l) Starting point = geometric center of feature points

m) Until starting point converges

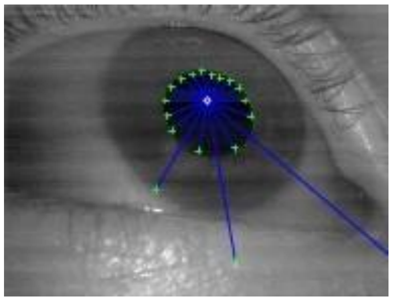


Illustration 4: Stage 1

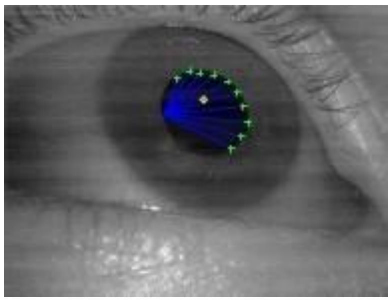


Illustration 5: Stage 2

4) *Ellipse fitting*

After getting a set of candidate feature points, the algorithm needs to find the best fitting ellipse to describe the pupil contour. The Starburst algorithm applies the Random Sample Consensus (RANSAC) to realize the fitting process.

1) *Model-based optimization*

Although the accuracy of the RANSAC fit may be sufficient for many eye tracking applications, the result of ellipse fitting should be improved through a model-based optimization that does not rely on feature detection.

2) *Mapping and calibration*

In order to calculate the point of gaze of the user in the scene image, a mapping between locations in the scene image and an eye-position must be determined. While the user is fixating each scene point
$\vec{s}=[x_s, y_s, 1]^T$, the eye position $\vec{e}=[x_e, y_e, 1]^T$ is measured. We generate the mapping between the two sets of points using a linear homographic mapping. This mapping H is a 3 × 3 matrix. Once this mapping is determined the user's point of gaze in the scene for any frame can be established
as $\vec{s}=H\vec{e}$.

• Starburst algorithm automation

The goal of starburst automation is to exempt from the process of inputting a series of information every time doing the experiment. In another word, put all the required information in a configuration file, and start the tracking experiment every time with only one click.

Step 1:

Modify the Starburst code to remove the pop-up window code. Read in all of the required input information from a configuration file instead of the pop-up window.

Input list:

1) Scene video
2) Eye video
3) Time of synchronization in scene video
4) Time of synchronization in eye video
5) Directory address to save the extracted scene/eye images
6) The number of the starting calibration image
7) The address and number of the ending calibration image
8) Directory address to save the calculated results
9) The number of the first frame to start calculation
10) The number of the last frame to end calculation

Step 2:

Call Python Program with from a batch file, with the appropriate configuration file as an input.

• Observations

Though we ran the off-line analyzer many times, we were unable to get satisfactory tracking.

In most cases, the tracking would work for about 3s after calibration. After that, the tracked gaze point would diverge from the actual gaze point.

In addition, we found that the calibration set-up was cumbersome and prone to error. The calibration process involves finding out synced points in the scene and eye videos; we were unable to come up with a mechanism to make this step deterministic. The accuracy of the eye tracking algorithm, of course, is dependent on getting the calibration to be perfect.

This was one of the observations that led us to explore the real-time option more thoroughly, and to automate to a good extent, the calibration process.

• Real-time eye tracker design

The major part of the project was to develop the appropriate software that can process the video signals coming in from the cameras shown above. As the head-mounted unit is very simple in itself, it falls upon the software to perform things like calibration, compensation of vibrations, and the actual eye tracking itself.

• Language and framework

OpenCV

OpenCV (Open Computer Vision) is an open-source image and video processing framework that has a large number of built-in, optimized routines for computer vision applications. The range of processing algorithms available, along with the fact that it is available free for commercial and student use under the BSD license, makes it an ideal choice as a framework for building the eye tracker.

Coding language

I decided to go with Python as the primary coding language. Python gives the advantages of providing a good object-oriented approach to designing the application, along with the ability to manipulate data at the level of bits and bytes when required. Also, a Python program can be easily ported to other operating systems, if so required, with relatively minimal changes.

The IDE used in development was Anaconda 2019 (Jupyter Notebook).

Queue-based pipeline

Most image processing algorithms are CPU-intensive. Video capture, on the other hand, is an operation that takes a fixed amount of time, and involves activity only when new video frames are available.
Running the video capture and image processing serially would consume more time than that between frames (33ms at 30fps); thus, to ensure full throughput, it is essential to run as many activities in parallel as possible.

In order to facilitate this, we organized the software as a series of *nodes* in a *pipeline.* Each node performs a specific task, *in its own thread.* Thus, a pipeline having several threads will have multiple processing steps happening in parallel. A sample pipe-line is shown below in Illustration 6.

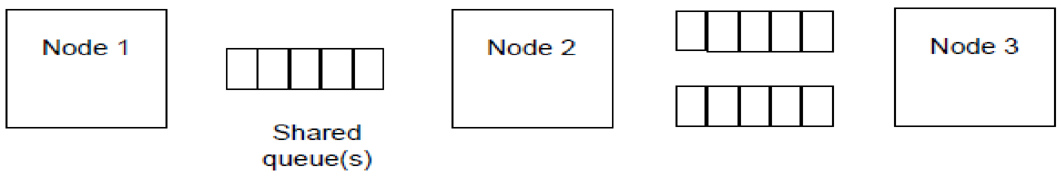


Illustration 6: A model pipeline

The nodes are connected to one other by one or more shared FIFO (first in, first out) queues. These queues are simply place-holders for whatever data that needs to be passed from one node to the next. These queues are implemented using locks and semaphores; this provides a safe and highly optimized way of synchronizing shared data across the producer and consumer threads.

Running the system in terms of parallel threads also allows the system to take advantage of multiple cores, wherever available.

• Algorithm

The approach that we took for eye tracking has two stages:

1. Finding the center of the iris

   This step forms the major part of the algorithm. The algorithm first finds an 'eye window' – a rough estimate of the eye. The second step is to find the exact position of the iris within the eye window, using a Circle Hough transform.

2. Mapping the iris to a point in the video from the scene camera.

   Using pre-determined calibration points, the position of the iris is linearly mapped to a position on the screen.

Eye position detection

The major part of the algorithm concentrates on accurately finding the center of the eye. My algorithm for finding the eye position has two major steps:

1. Find the 'eye window' – a rough position of the eye.

   The algorithm used to detect the eye window has been adapted from the one published by Wu and Zhou [1]. This is described in more detail in Section 5.2.3 .

2. Within the eye window, localize the center of the iris.

   A Circle Hough transform is used to pin-point the center of the iris. This is explained in detail in Section 5.2.6 .

The processing pipeline to accomplish these steps is shown below in Illustration 7.
The processing involved in each stage of this pipeline is described in the next few sections.

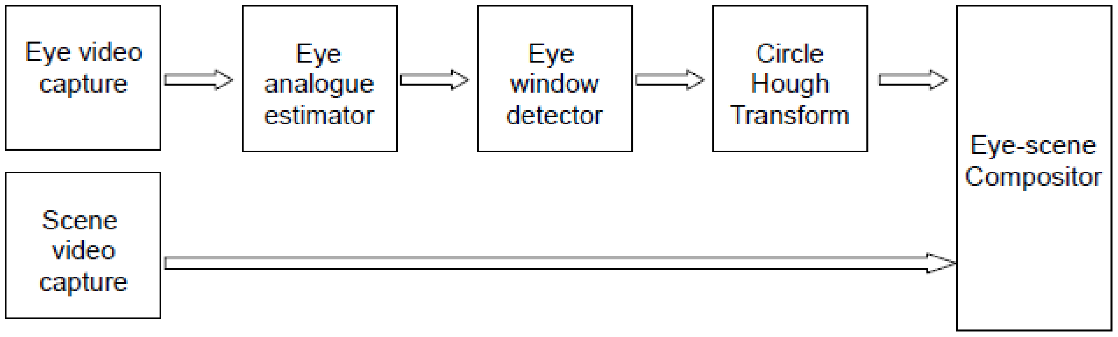


Illustration 7: Eye tracker: video processing pipeline

• Video Capture

The video capture node has minimal processing involved.

Initialization parameters:

(1) The identifier of the camera to be opened

(2) The expected height and width of the eye window.

The processing done in the video capture node is shown below. For clarity, this flow chart shows only one iteration of the process; in the system, this iteration runs until a stop command is received.

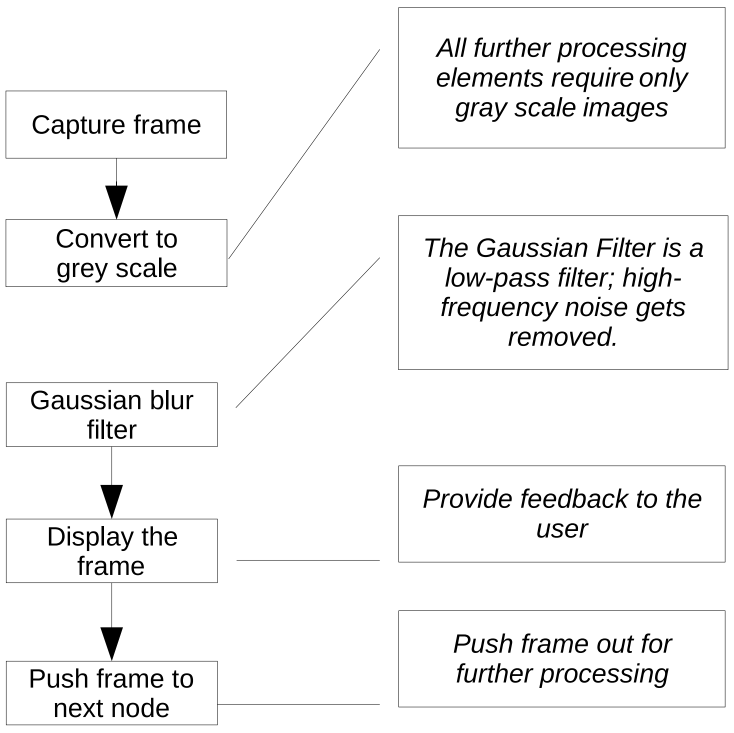

Capture frame
Convert to grey scale
Gaussian blur filter
Display the frame
Push frame to next node
All further processing elements require only gray scale images
The Gaussian Filter is a low-pass filter; high-frequency noise gets removed.
Provide feedback to the user
Push frame out for further processing

Illustration 8: Video capture flowchart

A sample image from the eye camera, after this processing, is shown below.

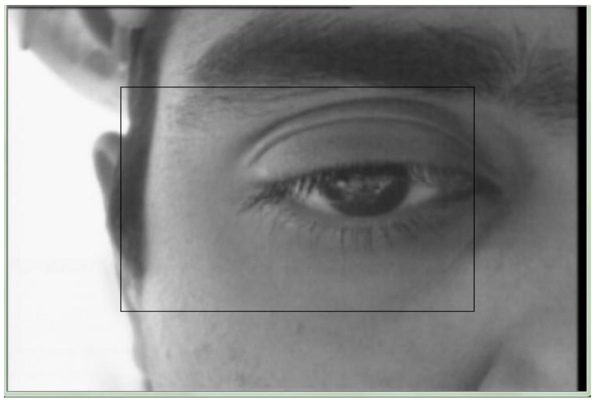


Illustration 9: Eye image

• Eye analogue detection

The second step of the process is to find the rough position of the eye, within the eye image.

Wu and Zhou [1] describe a method to efficiently select face candidates from an image for further processing. The method is based on the model that the pixels corresponding to the eye in a face image are generally darker than the surrounding regions. As a first step, a filter to detect such pixels applied on a gray-scale image, generating eye-analogue pixels. These are reduced to the eye pair by successively applying further models, based on the size, height, width, aspect ratio, and the fact that eyes are expected to occur in pairs.

As the eye-analogue detection is a fairly heavy process, only this step is done in the eye-analogue node; the rest of the processing is carried out in the eye window detector node.

The process is, on the surface, simple.

1. For every pixel in the image, find the *mean* value of the following regions, shown in Illustration 10.

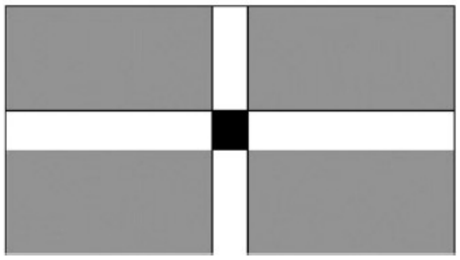


Illustration 10: 'Neighbor' regions

In Illustration 10, if the expected eye size is w x h pixels,

- The width of each gray region is w/2 pixels.
- The height of each gray region is h/2 pixels.
- The white regions are one pixel in height or width.
- The current pixel is never considered in any of these calculations.

2. If the current pixel is *darker* than six of these regions, mark this as an eye-analogue pixel.

The output of this step is a binary image – pixels recognized as eye analogues are marked as white (255), and the rest are marked black (0). A sample image is shown in the appendix, as Illustration 11. It is quite possible for more than one region to be recognized as eye regions in this step – for instance, the eye brows, hair, background objects, noise, etc. may all be recognized.

Illustration 11: Eye analogues

The task of filtering out other objects from the eyes is done in the eye window detector.

• Optimization: Scaling down, cache accesses

The challenge comes in the fact that there are eight sets of means being calculated for every pixel. In the current set-up, the expected eye window size is around 300x200 pixels; this means that there are 300x200 = 60000 values being accessed, *for every pixel in the frame.* Given that the frame size is around 640 x 480 pixels, this leads to an extremely slow process – it takes around 10s to process a single frame!

In order to optimize this process, some steps were taken.

(3) As this step only generates a *rough* idea of where the eye is, the operation may be carried out on a much smaller image without any loss of accuracy. Thus, before running through the eye- analogue operation, the frame is scaled to 210 x 160 pixels. The approximate region can be easily mapped to the original image by scaling it up linearly.

(4) Pixels are traversed row-wise. It can be easily seen that between two pixels, there is a considerable overlap in the regions of comparison. We took advantage of this fact; after the first pixel in a frame, the complete regions are never traversed again – instead, only the differences are calculated, and added or subtracted accordingly. This step, by analysis, can provide a speed- up of 30 times; in practice, the speed-up varied from 10-15 times.

The apparent inaccuracy in the speed-up values above is caused by cache misses. These are caused when reading new pixel values from a column – each pixel access is far away from previous values. To mitigate this, the original image was first transposed; this converts column traversals in the original image to row accesses in the new image. This helps reduce the processing time by a factor of 3 or so.

The above optimization schemes allow the system to run at nearly 26 frames per second – an improvement of about 260 times over the first pass!

• Eye window detection

The eye-window detector receives as inputs the binary image representing eye analogue pixels. This node processes this image to reduce them to a single candidate for the eye.

The steps followed is given in Illustration 13. In brief, this step decomposes the binary image into objects ('blobs') composed of connected pixels. I used the open-source library, CVBlobs, to perform all operations on blobs. The initial set of blobs is filtered using parameters such as the expected height, width and orientation.

The figure below shows the output of this step for the sample images shown previously. As it can be seen, the filter has removed blobs corresponding to the eyebrows and hair, leaving only the eye in place.

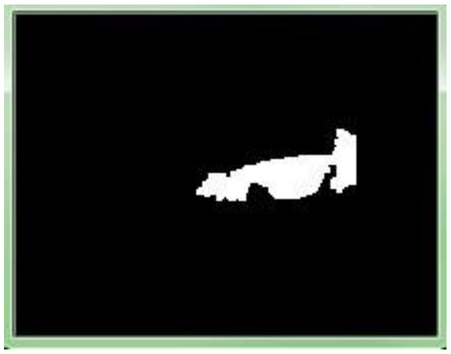


Illustration 12: Single eye window

The output of this node is the coordinates of the center of the eye region. This serves as an approximate estimate of the eye's location. Note that this does not say anything about the position of the eyeball; this merely serves as a reference for further processing.

The coordinates of the eye region are passed on to the next node, the Circle Hough Transform.

• Assumptions

After blobs have been removed based on height, width and orientation, it is possible that the eyebrows, if present in the original image, are also retained. With this assumption, if more than one blobs are present, the lower one is taken as the eye region.

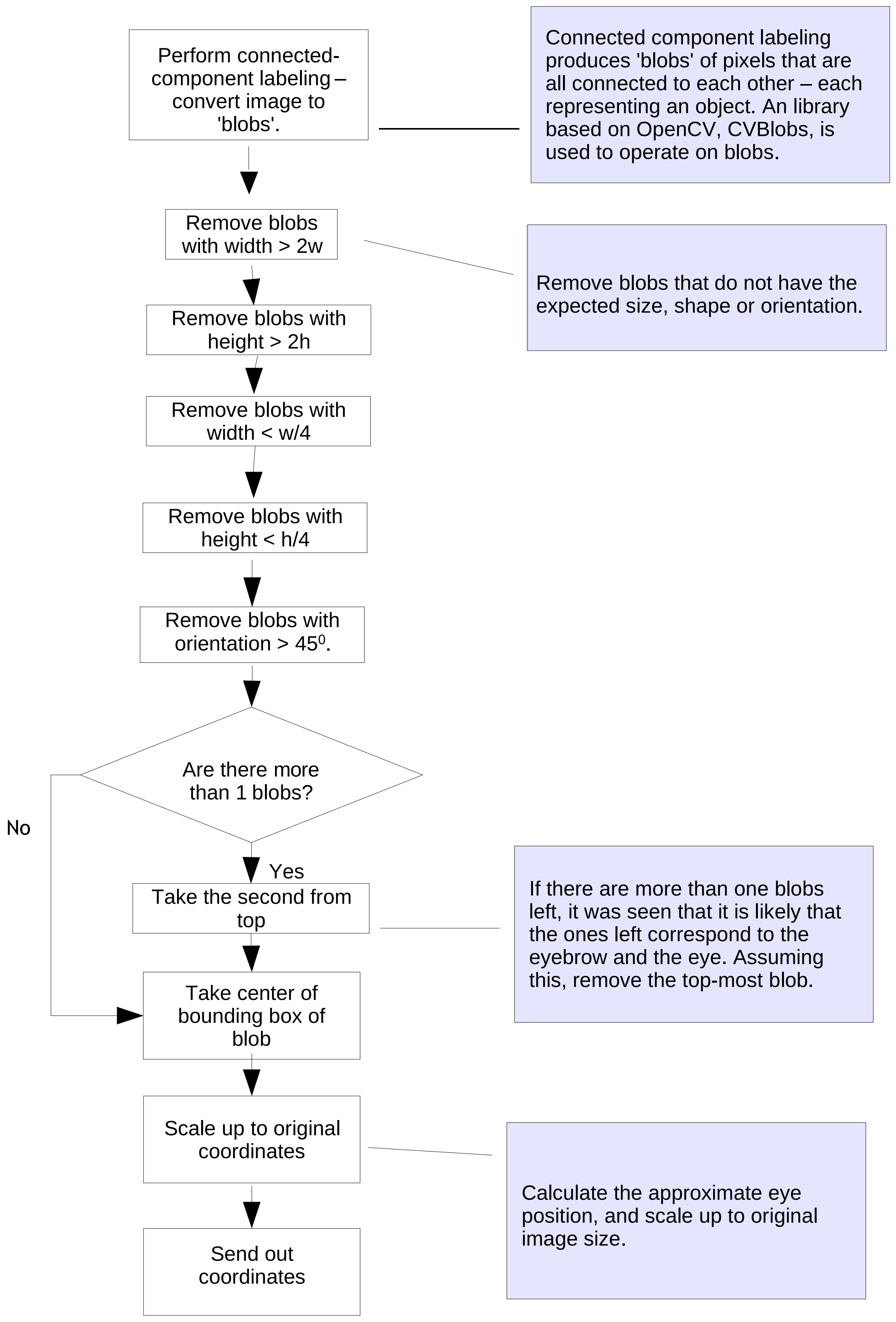

Perform connected-component labeling – convert image to 'blobs'.
Connected component labeling produces 'blobs' of pixels that are all connected to each other – each representing an object. An library based on OpenCV, CVBlobs, is used to operate on blobs.
Remove blobs with width > 2w
Remove blobs that do not have the expected size, shape or orientation.
Remove blobs with height > 2h
Remove blobs with width < w/4
Remove blobs with height < h/4
Remove blobs with orientation > 45⁰.
Are there more than 1 blobs?
No
Yes
Take the second from top
If there are more than one blobs left, it was seen that it is likely that the ones left correspond to the eyebrow and the eye. Assuming this, remove the top-most blob.
Take center of bounding box of blob
Scale up to original coordinates
Calculate the approximate eye position, and scale up to original image size.
Send out coordinates

Illustration 13: Reduction to single eye window

• Pin-pointing the iris: Circle Hough Transform

At this point, the system has a rough idea of where the eye lies within the eye image. To perform accurate eye tracking, it is essential that the center of the iris be located. The tool used to perform this step is the Circle Hough Transform.

A *Hough Transform* is, essentially, a shape detector for edge images. It is widely used to detect straight lines and circles in edge images. Edge images are generated by running an edge detector, such as a Canny or a Sobel operator, over a gray-scale image. Edges correspond to points where there is an abrupt transition from bright to dark intensities.

A Circle Hough Transform detects circles of specified radii in edge images. The operation is as follows:

(5) At every edge point:

A. A tangent is drawn.

B. The perpendicular to the tangent is drawn.

C. We move along the perpendicular, on both sides, to a distance corresponding to the required radius.

D. The points reached each receive a vote.

(6) After all edge pixels have been processed, the pixels with a large number of votes correspond to circles of the required radius!

It is, of course, necessary to take into account factors like:

(7) No edge operator is perfect when there is noise; perfect circles are rarely encountered.

(8) It might be necessary to look at votes cumulatively around a pixel, rather than just at a single pixel.

The greatest advantage of using a Hough transform is that it works even if the shape to be detected is partially hidden. In the current context, the iris, usually close to a circle, is almost always partially hidden by the eyelids. This makes the Hough Transform an attractive method to find the position of the iris.

The actual process is shown in Illustration 14.

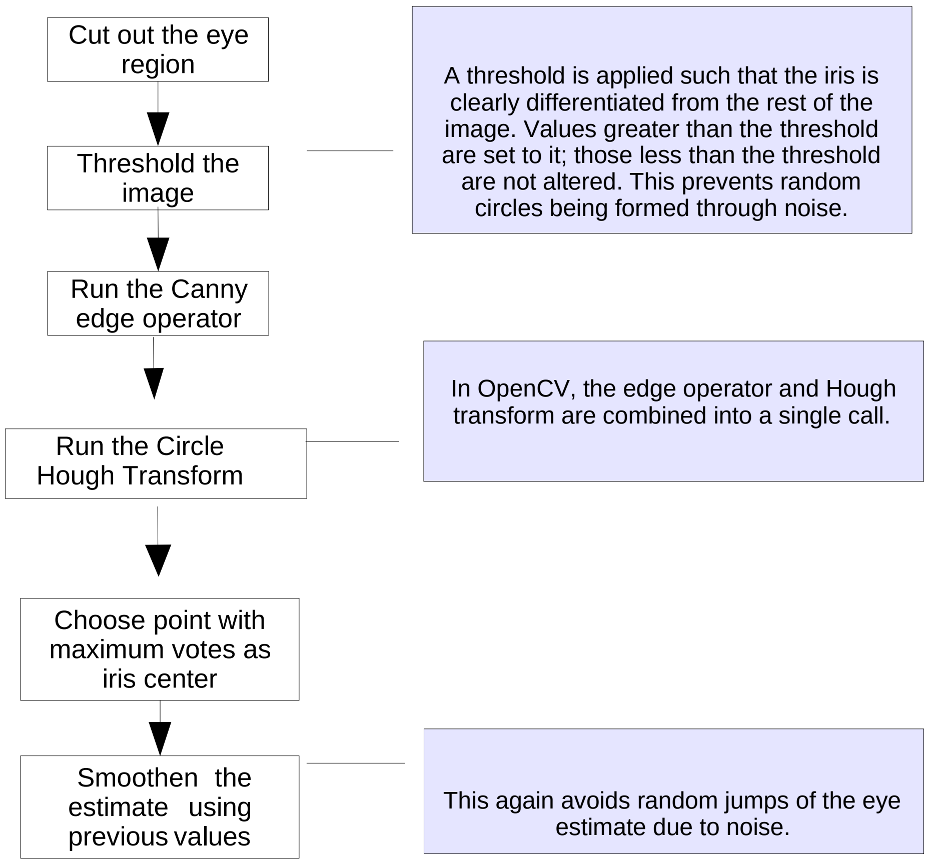

Cut out the eye region
Threshold the image
A threshold is applied such that the iris is clearly differentiated from the rest of the image. Values greater than the threshold are set to it; those less than the threshold are not altered. This prevents random circles being formed through noise.
Run the Canny edge operator
In OpenCV, the edge operator and Hough transform are combined into a single call.
Run the Circle Hough Transform
Choose point with maximum votes as iris center
Smoothen the estimate using previous values
This again avoids random jumps of the eye estimate due to noise.

Illustration 14: Applying circle Hough transform

A sample output, after the threshold step, is shown below.

Illustration 15: Thresholded image

The estimate of the eye position after the Circle Hough transform is shown below.

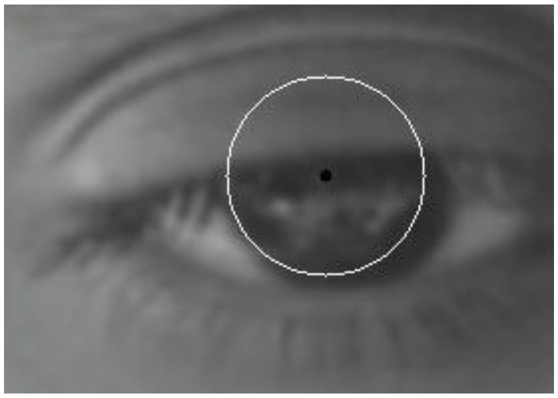


Illustration 16: Estimage of the iris center

• Composition and tracking

The final step in the eye tracking process is to translate the eye point calculated in the previous step into a corresponding point in the scene camera. This can be accomplished through straightforward linear interpolation:

$$s_{new}=(e_{new}-e).\ scale+s$$

where

s_{new} is the gaze point to be calculated

e_{new} is the currently estimated eye-coordinate

scale is the factor by which the eye coordinate has to be scaled up

s is a known gaze point that corresponds to a known eye point e.

Note: s_{new} and s are points on the scene video.

The challenge here is to accurately determine *s,e* and *scale – i.e.,* to calibrate the device.

Once the scene point has been determined, a white bulls-eye is drawn on the scene video to represent the gaze point. A sample image is shown below. If the estimated position is outside the screen, a bull's- eye with only two circles is shown.

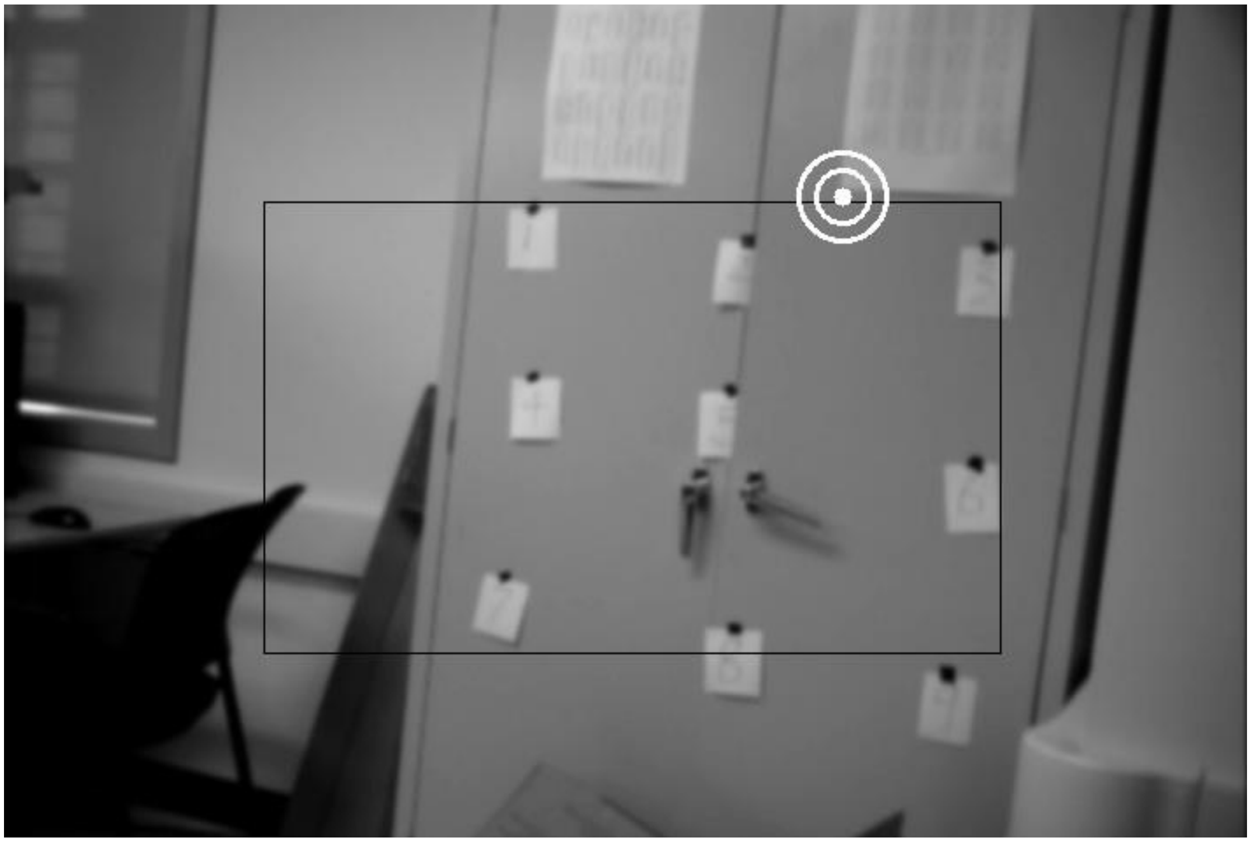


Illustration 17: Estimated gaze point

• Calibration

The *compositor* node operates in one of two modes:

(9) Tracking mode

Tracks the gaze point, as described above.

(10) Calibration mode

Calculates the parameters *s,e* and *scale.*

The calibration proceeds as follows:

(11) The 'known' points on the screen should correspond to the brightest point on the screen. In the calibration mode, this is shown as a black bulls-eye in the 'brightpoint' window. Illustration 18 shows the brightest point in the scene being denoted by a black bulls-eye (close to the right end of the scene).

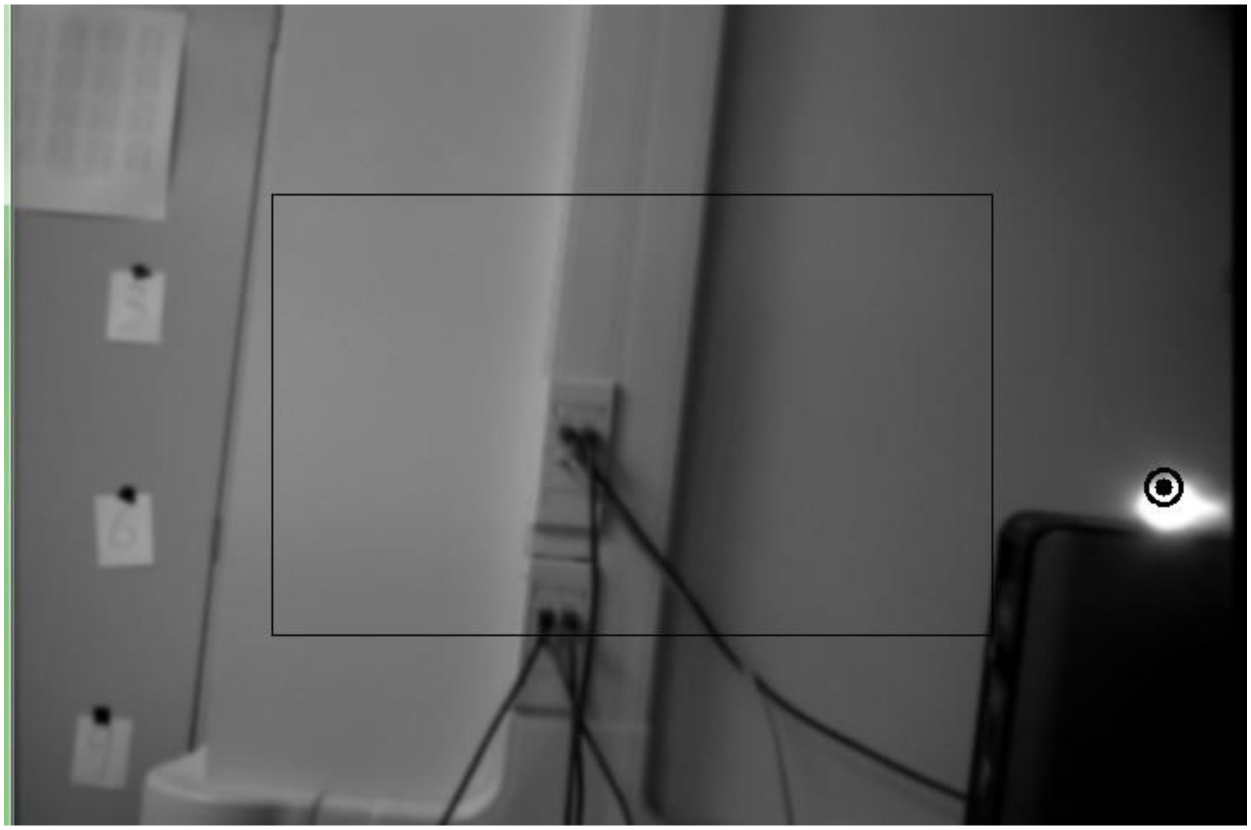


Illustration 18: Brightest point in the screen

(12) It is recommended that a bright object, such as a powerful flashlight, be used. The calibration should ideally happen in a relatively dark environment.

(13) The bright object is positioned in two parts of the scene, *s* & *s'*, ideally in diagonally opposite corners (the second point after the first point has been processed). The user is asked to look at these bright objects (without moving his/head), and the corresponding eye points, *e & e'* are recorded. The software simply finds the brightest point in the screen to find *s* and *s'*.

(14) The scale value is then calculated as:

$$scale=\frac{(s'-s)}{(e'-e)}$$

• Results and observations

ii) Speed

As described in Section 5.2.4 , significant optimization had to be carried out to ensure that the system performs in real-time. One measure of the speed is the throughput – the number of frames being processed per second (expressed as frames-per-second, fps). The system was measured to have a throughput of 26 fps, far exceeding our initial estimates!

Another measure is the latency. This is the time between when a particular input enters the system, and the corresponding output is seen. We have observed that the latency in this particular system is around 3 seconds.

iii) Accuracy

It was seen that when properly calibrated, the eye tracker is able to estimate the gaze position with reasonable accuracy.

To test for accuracy, we adopted the bright-point method we used for calibration. The subject looks at the brightest point on the screen; during testing, we used a flashlight aimed at the camera for this purpose. The system runs a simple maximum operation to determine this point in the scene video. The gaze point estimate is then compared with this brightest point, and the error is determined in terms of pixels.

This experiment, of course, assumes that the subject is constantly staring at the brightest point in front of him, and that the system calculates the same point as the brightest!

A chart of the error calculated over one particular run is shown below.

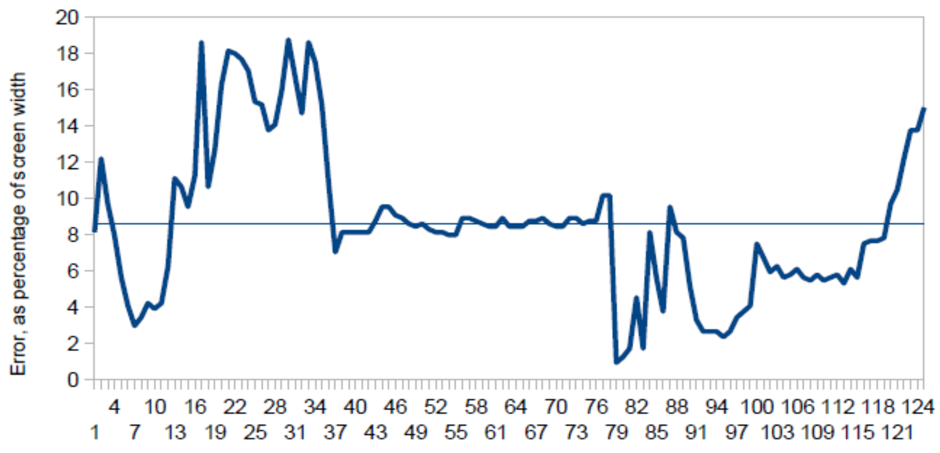


Illustration 19: Error estimates, as a percentage of screen width

It can be seen that the mean error is estimated to be around 8% of the screen width.
A second run is shown below. It can be seen that in this case, the error estimated is higher, around 15%.

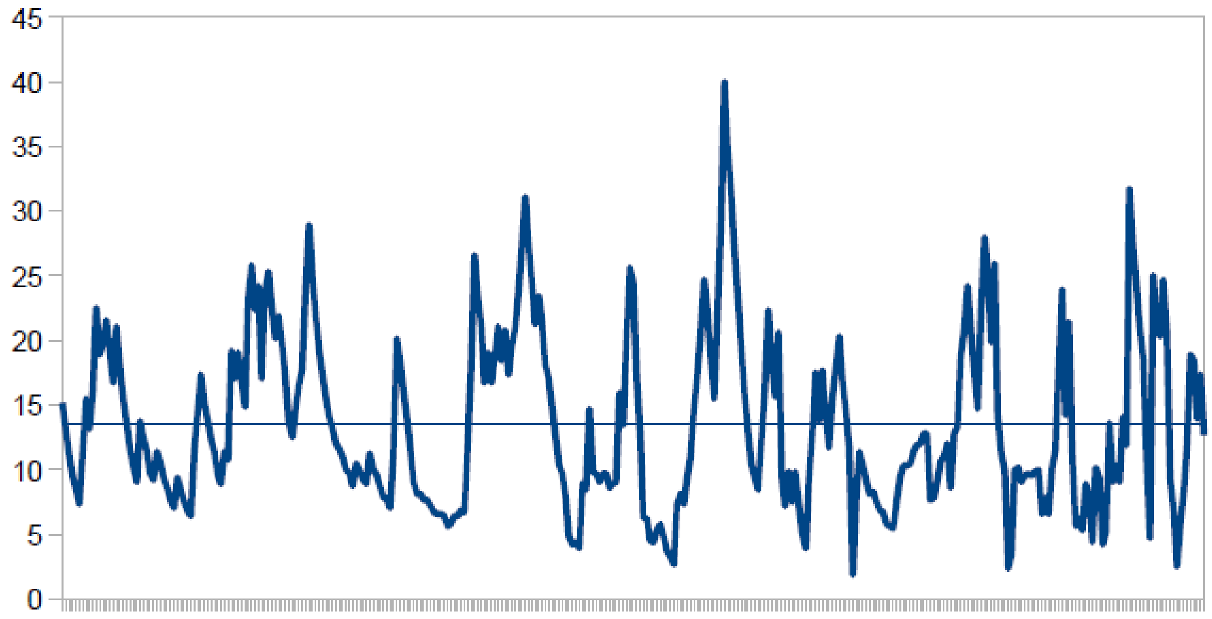


Illustration 20: Error estimates, second run

The accuracy is heavily dependent on several factors.

1. The calibration has to be accurate, in the first place. An incorrect calibration implies that the scale and offset values employed in transposing the eye position to the gaze position are wrong; this implies that the gaze position calculated will be in error.

   Calibration in a lab environment is relatively easy – the lighting can be controlled very easily. However, outdoors, it will be difficult to have a specific bright point. A better mode might be to employ some form of pattern recognition, and use some patterns as the reference points on the scene.

2. The conditions under which the calibration has been performed has to be maintained throughout the tracking session. These conditions include:

   a) The position of the hat must not slip. This will cause a large offset in the eye's position with respect to the camera; the eye may slip out of the region which is being processed.

   b) The position of the scene camera must not change from what it was during calibration.

This is self-explanatory; the scene points recorded will have an offset from the actual ones.

3. The connection from the eye camera to the hat is not rigid. It was intentionally kept flexible, to allow for easy adjustment of the camera. However, this leads to the camera vibrating – and that implies that the calibrated values may not be valid any more.

   We have tried to compensate for this vibration by calculating the offsets and scale using the iris' position with respect to the eye window itself. A better approach would be to calculate the position of the eye corners, and track the eyeball with respect to the corners.

4. Viewing angle of the scene camera

   During testing, it was found that the viewing angle of the scene camera was extremely small – about 40^0 or so. This meant that any time the subject is squinting to one side, the view point would go out of bounds of the scene being recorded.

   The solution, of course, would be to use a wide-angle camera; however, this will have to be balanced against the cost.

• Installation and Documentation -

This is a Python (2 and 3) library that provides a webcam-based eye tracking system. It gives you the exact position of the pupils and the gaze direction, in real time.

• Installation

Clone this project:

git clone **https://github.com/AbhishekTyagi404/Deep-Learning-Python/tree/master/Gaze_Recognition_Media_Player**

Install these dependencies (NumPy, OpenCV, Dlib):

pip install -r requirements.txt

The Dlib library has four primary prerequisites: Boost, Boost.Python, CMake and X11/XQuartx. If you doesn't have them, you can [read this article](https://www.pyimagesearch.com/2017/03/27/how-to-install-dlib/) to know how to easily install them.

Run the demo:

~~~~
```
python example.py
```
## Simple Demo
```python
import cv2
from gaze_tracking import GazeTracking
gaze = GazeTracking()
webcam = cv2.VideoCapture(0)
while True:
    _, frame = webcam.read()
    gaze.refresh(frame)
    new_frame = gaze.annotated_frame()
    text = ""
    if gaze.is_right():
        text = "Looking right"
    elif gaze.is_left():
        text = "Looking left"
    elif gaze.is_center():
        text = "Looking center"
    cv2.putText(new_frame, text, (60, 60), cv2.FONT_HERSHEY_DUPLEX, 2, (255, 0, 0), 2)
    cv2.imshow("Demo", new_frame)
    if cv2.waitKey(1) == 27:
~~~~

```
            break
        ```
```

- ## Documentation

In the following examples, `gaze` refers to an instance of the `GazeTracking` class.

Refresh the frame

python

gaze.refresh(frame)

Pass the frame to analyze (numpy.ndarray). If you want to work with a video stream, you need to put this instruction in a loop, like the example above.

Position of the left pupil

```python

gaze.pupil_left_coords()

```

Returns the coordinates (x,y) of the left pupil.

Position of the right pupil

```python

gaze.pupil_right_coords()

```
```

Returns the coordinates (x,y) of the right pupil.

### Looking to the left

```python
gaze.is_left()
```

Returns `True` if the user is looking to the left.

### Looking to the right

```python
gaze.is_right()
```

Returns `True` if the user is looking to the right.

### Looking at the center

```python
gaze.is_center()
```

Returns `True` if the user is looking at the center.

### Horizontal direction of the gaze

```python
ratio = gaze.horizontal_ratio()
```

Returns a number between 0.0 and 1.0 that indicates the horizontal direction of the gaze. The extreme right is 0.0, the center is 0.5 and the extreme left is 1.0.

### Vertical direction of the gaze

```python
ratio = gaze.vertical_ratio()
```

Returns a number between 0.0 and 1.0 that indicates the vertical direction of the gaze. The extreme top is 0.0, the center is 0.5 and the extreme bottom is 1.0.

### Blinking

```python
gaze.is_blinking()
```

Returns `True` if the user's eyes are closed.

### Webcam frame

```python
frame = gaze.annotated_frame()
```

Returns the main frame with pupils highlighted.

1 Conclusion

Research on using gesture to control OS or media player is already popular. But using eye gaze as multimedia control system hasn't seen much light yet. As researchers are moving forward, it will become a familiar research issue soon enough. The most important part of this research was to detect user's eyes and track them in real time that too using default webcam. There are various challenges like low light, webcam resolution etc.. But with adjustments we could achieve a satisfactory rate of detection. In future we would like to study and analyze more so that the accuracy level increases even in the challenging scenarios. Later we are planning to use advanced machine learning technology that learns and improves performance with use. In our paper, we focused on major features (Play, Pause, Forward, Reverse) as a starter. Considering how things work out hopefully in future we will try to improve our work and take it further by adding more features and stability. Completion of this work will pave the way for us to move forward in the next phase. When all the features are successfully implemented it would be an assurance for us that our eye tracking is properly mapped to the whole screen and then we will take our research even further to control more sophisticated features on windows (like playing games etc.).

2 References

[1] The OpenEyes eye tracking project, at http://thirtysixthspan.com/openEyes/

[2] The OpenCV project, at http://opencv.willowgarage.com/wiki/

[3] The CVBlobls project, at http://opencv.willowgarage.com/wiki/cvBlobsLib

[4] A. Dix, "Human-Computer Interaction," Encyclopedia of Database Systems, pp. 1–6, 2016.

[5] R. J. K. Jacob, Eye Movement-Based Human-Computer Interaction Techniques: Toward Noncommand Interfaces. Wash., D.C.: Human- Computer Interaction Lab. Naval Res. Lab., 1995.

[6] L. E. Sibert and R. J. K. Jacob, "Evaluation of eye gaze interaction," Proceedings of the SIGCHI conference on Human factors in computing systems - CHI 00, 2000.

[7] A. Z. M. E. Chowdhury, T. Hasan, and M. Rahman, "Visual content search using subconscious retinal response," 2017 IEEE Region 10 Humanitarian Technology Conference (R10-HTC), 2017.

[8] F. Timm and E. Barth, "Accurate Eye Centre Localisation By Means Of Gradients," Proceedings of the International Conference on Computer Vision Theory and Applications, 2011.

[9] A. Sharma and P. Abrol, "Eye Gaze Techniques for Human Computer Interaction: A Research Survey," International Journal of Computer Applications, vol. 71, no. 9, pp. 18–25, 2013.