A Simple Algorithm for Eye Detection and Cursor Control

Surashree Kulkarni¹, Sagar Gala²
¹, ² Computer Department
K. J. Somaiya College of Engineering, Vidyavihar, Mumbai, India
¹surashreek@me.com, ²sagargala56@gmail.com

Abstract

This paper presents an effective albeit simple technique to perform mouse cursor movement by first detecting the user’s eyes, and then calculating the position on screen at which the user is looking. The idea of our paper is to use a series of steps for image processing, and then use a certain algorithm to convert screen coordinates to world coordinates. For users without spectacles, the formula works quite well.

Keywords: Eye Detection, Image Processing, Connected Component Labelling, Cursor Movement

1. Introduction

The purpose of eye detection in the face lies in the fact that eyes are the most easily detectable features in the face. This is because, after processing a user’s image, the colour of the eyes are found to be distinct and can be pin-pointed much more easily than other features, like say, the nose. Also, the basis of our paper lies in the fact that the pupil position of the user’s eyes changes all the time, and tracking this position is important. Once this position is found in terms of screen coordinates, factoring in the distance from the computer and the head position, one can approximate a close estimate of the point he is looking at on the screen. Note that the words pupil detection and eye detection are used interchangeably and mean one and the same thing in our paper.

In our paper, we have formulated our own algorithm after studying several eye detection methods, two of which are: feature-based matching and template-based matching, both of which are a part of the active infrared approach for eye detection.

1.1. Principle of our method

The algorithm that we have come up with combines several different approaches, using their strengths to zero in on the eye position. A webcam is used to capture the user’s face continuously. The captured image goes through a series of transformations, at the end of which we get a binary image in which the pupil of the eyes are clearly highlighted (being white), the region around the pupils-eyes- are black, while the face is, again, white. One can understand why pupil detection in such an image will be simple.

Once pupil detection is done, we get the coordinates of the pupil in the image. Our algorithm uses a concept of distance of these coordinates from the midpoint of the screen to calculate the actual coordinates of the position on screen the user is looking at. A formula is applied to get these actual coordinates. These are then passed to a function in MATLAB to move the cursor.

2. Proposed Method

2.1. Architecture

The premise of our algorithm is very simple. However, it assumes two things: a) face detection of the image is done b) the eyes in the face are clearly visible, such that the eyelids do not cover the user’s pupils. Once these conditions are met, eye detection becomes easy. The user’s image is captured continuously, with a frequency of 2 seconds (subject to change, of course). This image undergoes a series of image processing steps, all of which are shown clearly in Fig. 1.

The captured image is converted to a grayscale image since it’s much faster to process a grayscale image. The grayscale image then undergoes K-Means clustering after which it is converted to a binary image. This binary image undergoes further transformation, at the end of which we get a rectangular region containing the eyes of the user. Another small algorithm follows to get the exact coordinates of the pupils in these eyes. Once these coordinates are calculated, a formula is applied...
factoring in the distance of the face from the screen, and the average radius of an eye. The output of this is the actual coordinates to which the cursor is moved.

2.2. Detection of eyes’ region

Once the image has been captured and a fast facial recognition method is applied to it, we get an image that has only the user’s face, with the eyes clearly visible. The second part is very important - that the eyes be visible clearly, because if the user blinks, and the eyelids cover the eyes, the algorithm fails and nothing is returned.

So after facial recognition, the image is converted to a grayscale image. This is essentially for two reasons: first, the colours of the skin, hair, eyes, etc. are not important for us. Colour in an image thus becomes plain noise. Secondly, processing a grayscale image is much, much faster in terms of speed than processing an RGB image. An eye-detection algorithm depends on images captured on-the-fly; the processor has to process an image in a time period that leaves no room for lag. An algorithm to move the cursor to the position the user is looking at on the screen is of no use if it presents a delay, because the user will look at several different positions on screen during this delay.

The grayscale image then undergoes K-Means clustering and thresholding [1]. K-Means Clustering divides the image into n clusters of similar intensity. This makes it easier to convert it into a binary image. After the clustering operation, the image undergoes thresholding. This portion is particularly tricky because the threshold value to be set is subjective to a number of factors: the complexion of the user, the lighting conditions in the surrounding area, presence or absence of facial hair, etc. We decided on taking an average of the pixel intensities of all the pixels in face, and using it as the threshold. This approach works only in some cases, so further work is required to develop a perfect method for calculating this threshold. Once the threshold is found, all the pixels below this threshold are set to 255 (white) and the ones above it to 0 (black). We’re basically making the lighter regions of the face white, and the darker regions (pupils, eye brows, etc.) black.

Once we get a binary image that highlights the areas that we want - the pupils of the eyes, that is, we proceed to perform ‘cropping’. This essentially involves treating the image as a 9x9 grid and changing the intensity of pixels in all except the middle block to 0 (black). This is done because, in most cases, the eyes are located in the centre region of the image. Incidentally, if the eyes are not in the centre region, they will also be blackened, and the image becomes useless, and is discarded. This is a small price to pay if compared with the processing time saved by discarding unimportant regions of the face, and focusing only on the eyes.

Once we have a ‘cropped’ image, we come to the most important part of the algorithm. We apply a 5x5 mask to the input image to locate the pixel where the mask matches perfectly. Now, the 5x5 mask we used was as follows:

\[
\begin{pmatrix}
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 255 & 0 & 0 \\
0 & 255 & 255 & 255 & 0 \\
0 & 0 & 255 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
\end{pmatrix}
\]

This mask can be altered in terms of size or pixel values, of course. We decided to use a 5x5 mask because a 3x3 mask was too small to give accurate results, while anything bigger than a 5x5 mask would simply increase the processing time, making the algorithm unfavourable. This 5x5 mask is run over the middle block (of the virtual 9x9 grid) in the image. A pixel match is considered true if there are more than 20 pixel matches (out of a possible 25) around the middle pixel. This is the most time-consuming portion of the algorithm as it involves running a pixel-by-pixel matching process over the image. Once a matching
pixel is found, the algorithm stops and returns the coordinates of this matching pixel. These, in fact, are the coordinates of either the left or the right pupil- one can tell which by simply checking the x-coordinate of the pixel and comparing it with the centre coordinates of the image. The coordinates of the other pupil can be calculated by simply adding or subtracting 75- the average number of pixels between two eyes. It would be much better if the facial recognition algorithm performed in the first step gave an approximate value of the face width. This would help in calculating the distance between two pupils in a better manner. Refer Fig. 2 to see the result of this entire process.

Figure 2.

2.3. Getting actual coordinates
We now come to the last portion of the algorithm- converting the pupil coordinates into actual coordinates. Once the pupil coordinates are obtained, they are subtracted from screen centre coordinates. The following formula is applied to this difference:

\[
\text{newx} = \text{centrex} + [(d+r)/r]\times\text{subx}; \\
\text{newy} = \text{centrey} + [(d+r)/r]\times\text{suby};
\]

where,

- \text{newx} = \text{actual x-coordinate}
- \text{newy} = \text{actual y-coordinate}
- \text{centrex} = \text{x-coordinate for screen centre}
- \text{centrey} = \text{x-coordinate for screen centre}
- \text{d} = \text{distance of user from screen in terms of number of pixels}
- \text{r} = \text{average radius of the human eye in terms of number of pixels}
- \text{subx} = \text{x-coordinate difference between pupil and screen centre coordinates}
- \text{suby} = \text{y-coordinate difference between pupil and screen centre coordinates}

The mouse cursor is then moved to the position [\text{newx},\text{newy}].

3. Experimental Results
In this section, we talk about the empirical results of our algorithm. We used MATLAB for writing and running the code, mostly because MATLAB has several inbuilt functions that help speed up the process.

We ran the code in real time, and found that the algorithm gave a success rate of 75%. The execution time for every frame captured was 0.68 s, which, if not excellent, is pretty good, considering we were using MATLAB and an i3 processor to run the code. The code would run a lot faster on a better processor, or if the programming language was converted to C or C++.

We also experimented with users who wore spectacles, but the results were very unsatisfactory. The eye detection algorithm gave wrong results more often than not. This was because the captured image showed pupil positions that were slightly distorted due to the presence of spectacles. However, this slight distortion turned into a major error in terms of calculation of actual coordinates.

4. Conclusion
Our algorithm is divided into two portions: eye detection and calculation of actual screen coordinates. The image processing algorithm for eye detection is extremely simple and is efficient too. The formula for calculation of screen coordinates places the cursor within proximity of the position the user is looking at, with an error of 2%. The execution time for the entire process is 0.68 s for each frame, which is also quite good.

In terms of limitations of our algorithm, we can say that it does not work very well if there are no ambient lighting conditions in the surrounding environment. Also, the algorithm produces erroneous results for users with spectacles. For future work, it would serve well to focus on these two areas: improving the eye detection algorithm to work in poor to moderate lighting, and factoring in a distortion produced due to spectacles in the final formula that calculates actual screen coordinates.

REFERENCES