— Report P&O CW 2019–2020 Task 3 —

Department of Computer Science – KU Leuven **August 18, 2020**

Team 03	
<Bram Vanbilsen>	26h
<Adam El M'Rabet>	19h
<Liam Volckerick>	21h
<Maarten Pyck>	11.5h
<Sebastiaan Wouters>	15.5h
<Pieter-Jan Van den Broecke>	12h

Demo: `https://penocw.cs.kotnet.kuleuven.be:80#/demo-task-44?0xjwkkslam9`

1 Introduction

We are developing an application that can detect screens and their orientation, given an image containing all screens, and link these screens and all information regarding that screen to a slave ID, which in turn is connected to our server. We developed 4 algorithms that work based on each other's output to analyse a given image by using computer vision. The first challenge lied in detection screens that would display a color. By assigning each individual screen a certain color, we could hypothetically differentiate between server slave ID's and their corresponding screens. At the moment of writing this report, we have yet to implement the connection between slave IDs and their corresponding screens on the initial image. Also, we can only work with one color as of now, to detect each screen. Our algorithms will be further discussed in section 3. Following premises were held to make our algorithms functional for now:

1. Screens have to display at full brightness.
2. No screens can overlap each other & no reflections of light on the screens, we have found a possible solution for this. Yet to implement (thoughts will be discussed in section 3.2.7)
3. All screens have to display the same vibrant color.
4. Each screen is a rectangle shaped monitor, thus no circular monitors. It needs corners.

2 Design schematics

2.1 Refactoring

Due to a lot of code being located in the same folder, and both slave and master scripts being displayed by both, we had to refactor our whole system. We set up the client in such way that it's usable by both the master and slaves. The server emits the user type to the client, which then in turn is constantly listening. Within the client, calling functions from within the user's console, will now trigger alerts and cancel the command, if the user-type does not have access to those function calls.
To remove any typo errors we also created ENUMs which transform each string type of

command to a constant variable that can be used by us when writing code. For example each user connection type:

```
export const enum ConnectionType {
        SLAVE = "slave",
        MASTER = "master"
}
```

We also have created ENUM's for every command socket.io can emit. See code underneath:

```
export const enum SharedEventTypes {
        NotifyOfTypeChange = "notify-of-type-change"
}

export const enum SlaveEventTypes {
        ChangeBackground = "change-background",
        DisplayArrowUp = "display-arrow-up",
        DisplayArrowRight = "display-arrow-right"
}

export const enum MasterEventTypes {
        ChangeSlaveBackgrounds = "change-slave-backgrounds",
        SlaveChanges = "notify-master-of-slaves",
        SendArrowsUp = "send-arrows-up",
        SendArrowsRight = "send-arrows-right"
}
```

The client also allows us to get and set each connection made, making it possible to return the list of users within the user-type, connected to the server.

3 Algorithms

At this point, we've come up with 4 algorithms that work together to detect each screen and its orientation. We'll explain this further in all subsections. Our general idea is based upon color detection and the orientation will be calculated given all four corner coordinates of each screen.

3.1 General Idea

We're trying to detect a screen by assigning it an uncommon color, hoping that no interference with that given color occurs in the surroundings when taking the picture. An example color would be one like pink which has the following RGB values: (255, 80, 170). We can determine the orientation by calculating corner coordinates and linear connections between the four corner coordinates. Calculating the angle of each corner could then give us the angle the screen is making with the x-axis, given that the screen is rectangular. If both edges, that are the longest, are in a 0-45 degree angle in comparison to our imaginary comparison x-axis, we maintain top, bottom, left and right as it would be with a standard screen. If the angle is in between 45-90 degrees we do following (idem solution will apply for when the orientation is in the opposite direction (-45 degrees tot -90 degrees)):

- Right becomes Top
- Top becomes Left
- Bottom becomes Right
- Left becomes Bottom

Given that the positive angles result when the monitor rotates counter clockwise, starting from a standard orientation with Top, Bottom, Left and Right as we are used to when looking at a normal monitor.

3.2 Screen Detection Algorithm

This algorithm consists of multiple steps to filter out a single screen with a given $color = C$ in the HSL color space. We start with the following image (see 1).

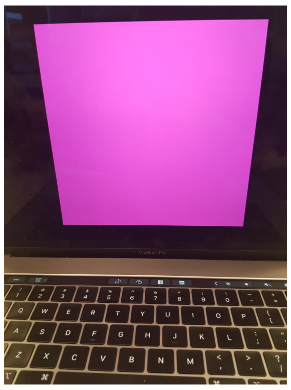


Figure 1: original picture

3.2.1 Removing unwanted colors

In this step, the algorithm removes all colors that are not similar to C. It does so by first looping over each pixel and removing the color if it is not close to C. We call the resulting image $filter$ (see 2).

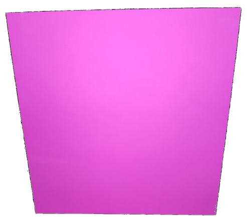


Figure 2: removed unwanted colors

3.2.2 Finding edges

In the following step, we make a copy of *filter* and scale this up by 1px. Next, we use the XOR operation to get the difference between the *filter* image and the scaled up *filter* image. This results in a first image in which the edges are visible (see 3). But, due to imperfections in the pictures, there will still be a lot of seemingly random noise in the result.

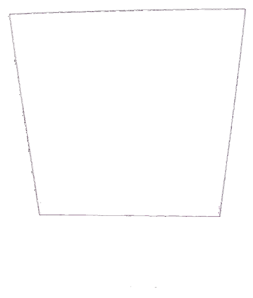


Figure 3: first edge filter

3.2.3 Tracing the edges

We now loop over the image again and store every pixel that still has color. To visualize this better, we also thicken the edges (see 4).

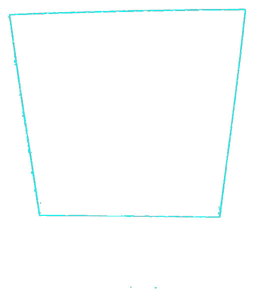


Figure 4: traced edges

3.2.4 Filtering noise (1)

In this first noise filtering step, we loop over each pixel p in the picture. If p has less neighbors with a similar color to C then a set threshold, we remove the pixel (see 5). This will remove a lot of scattered noise because the noise often consists of some isolated and random pixels in the image.

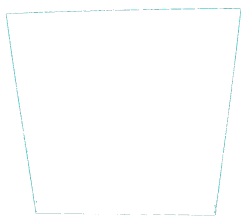


Figure 5: First filter result

The following code is not yet implemented but this is our plan to extend the algorithm

3.2.5 Filtering noise (2)

This step will check the image for the last bit of noise and attempt to remove it. It will do this by separating the resulting image from the previous step in squares of set dimensions (to be decided). Each square counts the amount a of colored pixels inside of it. If a is smaller than a threshold (will depend on dimensions of S), all the color of the pixels inside the square will be removed, nothing happens if a is larger than the threshold. The assumption here is that there should be a lot of pixels in the squares containing edges, but only few or no pixels in the others.

3.2.6 Finding corners

In this final step, we will attempt to reduce the output of the previous step to just 4 pixels. This will be done by connecting all points on the screen and reducing the connections to just the 2 longest connections.

3.2.7 Extra thoughts

What we have implemented has proven to work for multiple cases. Edges tempt to not get filtered out which is of course important for ‘Filtering noise (2)‘. Assuming corners are not covered too much, we are fairly confident that our ‘Finding corners‘ step will succeed. If a small part of the corner is covered, there might be an imperfection because only 4 corners are searched for, but this will not be significant. Also if the screen is split in two (perhaps there is an object in front of it), it should still find the corners because part of the edges still remain.
Lastly, this algorithm should run in $O(N \times M)$ (M being the width and N the height) due to the fact that all steps are simple loops over the image. This being said, multiple steps can and should be combined into one loop, but this is something we are keeping until we have a working implementation.

3.3 Hough Line Transform

One of the techniques that can be particularly useful when processing an image is the detection of shapes. This would simplify the interpretation of the image significantly. However this is not a trivial task. One way of detecting arbitrary shapes is through the identification of lines in an image. This allows the further interpretation of shapes like triangles, rectangles etc. Richard Duda and Peter Hart have come up with an algorithm that does this and called it: 'Generalized Hough Transform'. [4] The way this algorithm works is through a voting procedure using a 2D array. This 2D array (also called accumulator) will be used for detecting high probabilities of the existence of a line at specific locations. The more 'votes' there are for a specific location, the bigger the chance is to have a line at that location.

A line's location can be mathematically described using two different ways. $y = mx + b$ describes a straight line and is entirely represented by the pair (m, b). For the Hough transform, straight lines are rather represented using the Hesse Normal Form:

$$r = x \cos\theta + y \sin\theta \tag{1}$$

By this manner a straight line can be fully represented by the pair (r, θ). Then starting form a specific origin, every line is detected (passing pixel by pixel) by adjusting and incrementing θ and r and therefore generating the Hough Space. Lines are then represented as dots where many curves intersect.
The number of intersections at one point determines its voting. And as described before: the more voting, the more chance that we are dealing with a circle.

Figure 6 represents the result of applying Hough Line Transform (after detecting edges using Canny Edge algorithm) using the opencv.js library.

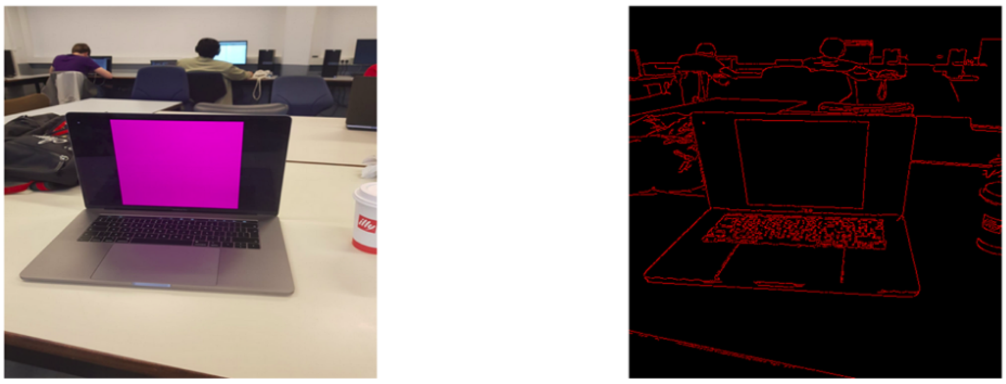


Figure 6: Detecting lines using Hough Line Transform with opencv.js

The library allows the user to edit parameters such that the results become better (or worse) as represented in figure 7.

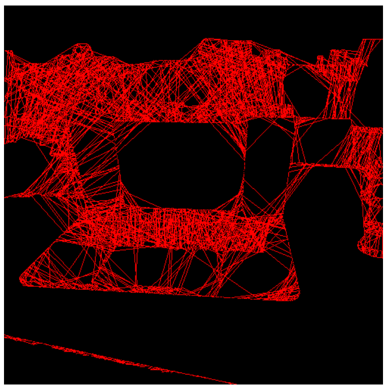


Figure 7: Editing parameters on the Hough Transform has significant effects

The only remaining task is to draw the lines forming a shape of a specific color. The library opencv.js [1] allows us to edit the picture and keep only pixels from a specific color using the *inRange*() method. After applying this "color filtering", applying Hough Transform results in much more precision as is represented in figure 8.

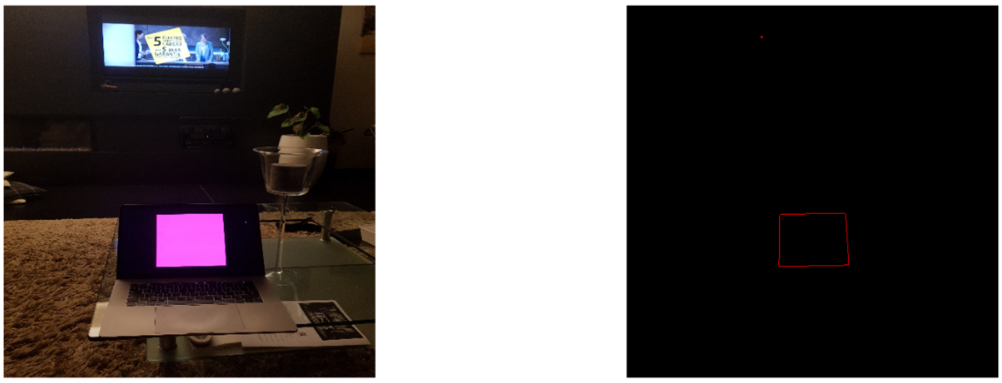


Figure 8: Hough Line Transform on a specific color.

3.4 Harris Corner Detection

After preprocessing the master's uploaded image for the resulting screen edges, we can retrieve the corner coordinates of each screen by running the Harris corner algorithm. This algorithm detects corners by looking for change in gradient. A corner is an intersection of two edges resulting in a high variation in gradient over both directions in the x,y-direction. Thus we will try to find these changes in gradients by iterating over the whole image and detecting any changes in gradient. See image 9 for the visualisation of this function.

Let's assume we have a gray scaled image I, called by the builtin gray scaling function in openCV: `cv.cvtColor(src, cv.BGR2GRAY)`. We're going to sweep over the image with a window $\omega(x, y)$ with according displacements u in the x-axis, and v in the y-axis. By using the Eigenvalue of the displacement we can retrieve the variation of intensity/gradience on a certain location. [2]

$$E(u,v) = \sum_{x,y} w(x,y)[\ I(x+u,y+v) -\ I(x,y)]^2 \tag{2}$$

With

1. $w(x, y)$ is the window at position (x,y)
2. $I(x, y)$ is the intensity at (x,y)
3. $I(x + u, y + v)$ is the intensity at the moved window $(x + u, y + v)$

As we're looking for corners within our windows, we have to look for maximal variation in intensity. Thus we will have to maximize the equation above. The subsection that we mainly have to take into consideration will be the following term:

$$\sum_{x,y} [\ I(x+u,y+v) -\ I(x,y)]^2 \tag{3}$$

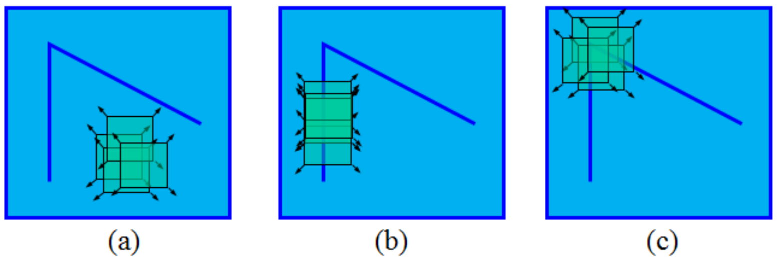


Figure 9: Window variation of intensity detection

Which we proceed to further estimate the result by calculating the Taylor expansion as follows:

$$E(u,v) \approx \sum_{x,y} [\ I(x,y) + u\ I_x + v\ I_y -\ I(x,y)]^2 \tag{4}$$

Further calculations give:

$$E(u,v) \approx \sum_{x,y} [u^2\ I_x^2 + 2uv\ I_x\ I_y + v^2\ I_y^2] \tag{5}$$

Expressing 5 into a matrix form will give:

$$E(u,v) \approx [u,v] \left(\sum_{x,y} w(x,y) \begin{bmatrix} I_x^2 & I_x\ I_y \\ I_x\ I_y & I_y^2 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} \right) \tag{6}$$

We can denote a term into following variable M:

$$M = \sum_{x,y} w(x,y) \begin{bmatrix} I_x^2 & I_x\ I_y \\ I_x\ I_y & I_y^2 \end{bmatrix} \tag{7}$$

So our original equation after substitution will turn out to be:

$$R = det(M) - k(trace(M))^2 \tag{8}$$

with

- det(M)=$\lambda_1\lambda_2$
- trace(M) = $\lambda_1 + \lambda_2$

Our variable R will contain a score. If R is greater than the given threshold it will be considered a corner. At the moment of writing, this threshold was set at 94 for all test examples concerning corner detection.

3.4.1 Harris corner detection results

Given figure 1 and figure 8 its output, we can then continue to run Harris corner detection algorithm to process all four corners. We can notice that the algorithm notices more than 4 corners on the edges of the polygon. This is due to noise and no smooth lines in the polygon, as we preprocess these lines by displaying pixels next to each other with a given width, thus creating imperfect smooth lines as a result. Finding a way to smoothen these lines will fix this issue.

Figure 10: All corners detected given fig 1 its output

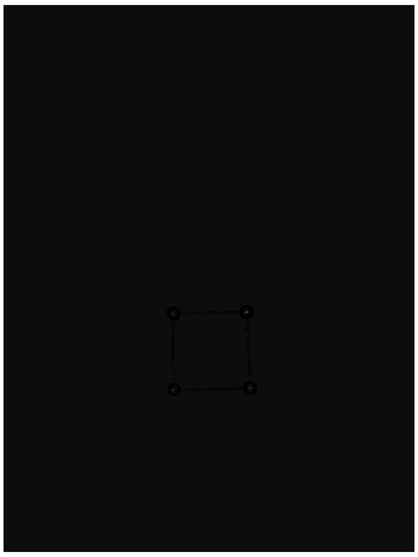


Figure 11: All corners detected given fig 8 its output

3.5 Corner labeling and angle computation

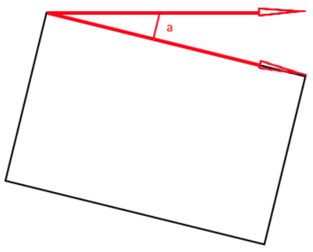
As explained above, the Harris corner detection algorithm returns the four coordinates that belong to the corners. The goal is to calculate the angle of the detected screen using this output. To make this possible, the corners need to be labelled. This means that all of the coordinates get one of these labels: (leftUpper, rightUpper, rightUnder, leftUnder).

First of all the labeling algorithm computes the sum of the x and y-coordinate for all the corners. The one with the highest sum gets assigned as rightUnder and the one with the lowest sum gets assigned as leftUpper. This can be explained as following, our x-axis goes from left to right and the y-axis moves from high to low. In most cases the leftUpper corner is located more or as much to the left and to the upside then any other corner. Accordingly with how we defined our axes this results in the leftUpper corner having the smallest sum.

The same logic applies to the rightUpper corner, but this one has the highest sum.

Only the leftUnder and rightUpper corners remain to be found. To accomplish this, the algorithm compares the x-value and y-value of both and then assigns the right label. This is simple because the leftUnder corner has a smaller x-coordinate and higher y-coordinate then the rightUpper corner.

Once the corners have been labelled, only the leftUpper and rightUpper corners are needed to obtain the angle of the screen. We define the leftUpper corner as the origin, and define two vectors from here as visible in the image underneath.



The angle between these two vectors, for which all the coordinates are given, can be calculated using this simple formula:

$$\arctan\left(vector2.y - vector2.x\right) - arctan(vector2.y - vector2.x) \tag{9}$$

This result is expressed in radians at first, which is then converted to degrees for convenience.

Algorithm complexity and test results: The algorithm to calculate the angle of a detected screen takes four coordinates as input and returns an angle as output. This should run in constant time since the input is of a fixed size and calculations do not depend on the given coordinates. While testing the runtime, some strange results were obtained for the first few executions. These fluctuations in execution time may be caused by the way javascript handles function calls. It seems that when the test is ran enough times, the results are consistent. The execution time stays constant as excepted at around 0.0097 milliseconds.

Some functions were written to test the algorithm with a few different inputs. Special cases that needed to be tested, include situations where screens are rotated a lot or where the corners do not represent a perfect rectangle All these cases returned the expected output without flaws.

4 Conclusion and prospects

This week, we decided to combine task 3 and 4 and work on the (advanced) screen detection algorithm. Although we do not yet isolate the corners of a screen with our algorithms, we are able to detect them while having multiple ideas on how to isolate the corners using the method described in ‘Harris Corner Detection‘ or ‘Screen Detection Algorithm‘. We also worked on a fairly straight forward way to detect the orientation of the screen. We believe this should be possible without the use of some sort of arrow on the screen as described in ‘Corner labeling and angle computation‘.

We have worked a lot with the html5 canvas (and its node equivalent) and with OpenCV to manipulate our images. OpenCV might end up being difficult to use on the server side due to some problems building on the server's machine, but should work fine on the client side. This does not appear to be a problem for us, because client computers/smartphones have become powerful enough to perform the needed calculation without the help of a server, but we would like to test this thoroughly later on.

This week, we will try to perfect our screen detection algorithm as it is a core part of our application. A version that handles most cases should be finished by Monday. A part of the team might also start working on task 6.

5 User licenses

1. node-canvas - MIT License: `https://github.com/Automattic/node-canvas/blob/master/Readme.md`
2. get-pixels - MIT License: `https://github.com/scijs/get-pixels`
3. OpenCV - 3-clause BSD License: `https://opencv.org/license/`

References

[1] openCV. Hough line Transform with opencv, 2019. https://docs.opencv.org/master/d3/de6/tutorial_js_houghlines.html/.

[2] Unknown. Corner Detection: Harris Corner and Shi-Tomasi Corner Detection, 2019. http://www.programmersought.com/article/2622158355/.

[3] user3150201. Using atan2 to find angle between two vectors, 2014. https://stackoverflow.com/questions/21483999/using-atan2-to-find-angle-between-two-vectors/21486462/.

[4] Wikipedia. Hough Transform (Wikipedia), 2019. https://en.wikipedia.org/wiki/Hough_transform.