— Report P&O CW 2019–2020 Task 4-5 —

Department of Computer Science – KU Leuven August 18, 2020


| Team 03 | |
|---|---|
| <Bram Vanbilsen> | 27h |
| <Adam El M'Rabet> | 21h |
| <Liam Volckerick> | 23h |
| <Maarten Pyck> | 20h |
| <Sebastiaan Wouters> | 22h |
| <Pieter-Jan Van den Broecke> | 20h |


Demo: https://penocw.cs.kotnet.kuleuven.be:8003

# 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. The first challenges we faced when starting this combined task, were optimising and bettering what we previously had coded. This will be explained further into detail in section 3.3. Making our code easier to use as well as creating an interface for the user to use for a simple way of interfering with the application. The challenge that remains ahead of us, is linking the orientation algorithm together with the corner/screen detection algorithm and the triangulation algorithm. In order to successfully compute each slave identity with its corresponding screen and coordinates, we made following premises:

1. Screens have to display at full brightness.
2. Screens can overlap each other, but preferably not corners & no reflections of light on the corners. If in some case, the corners are covered, we will continue to work with the smallest visible quadrilateral made from the 3 other corners. (it will still find 4 corners from the convex hull)
3. Each screen is a rectangle shaped monitor, thus no circular monitors. It needs corners.
4. Every master needs access to a webcam/camera that is at least 480p quality.
5. Orientation will only be calculated in 2D space, we will work on 3D towards the next assignments.
6. The master needs human intelligence and decision making to provide the algorithm with the best choice of color to work with during the corner detection.

# 2 Design schematics

## 2.1 User Interface

When running our application on a screen which has a width of at least 820 pixels, given it would be a desktop computer or an android tablet, then the camera gets called automatically

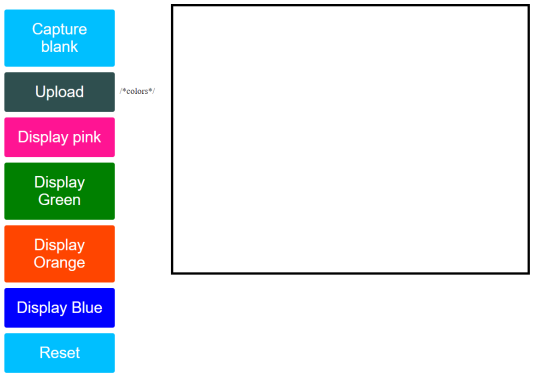



Figure 1: User Interface at initialisation

by the website itself. We display multiple buttons that provide a simple user interface, of which its behind the scenes implementation is defined in section 3.2. When running the application, the first user connected to the server will become the 'Master'. All other connections proceeding this initial connection will become slaves. Once all slaves are connected, the master gets to choose what color will be displayed upon each slave. Note: The color can only be picked in advance of capturing any photos of the slaves! We assume that the master can decide which color would fit best in its slave setup (human interaction). To start, we take a picture of the initial setup with all screens turned on at full brightness, displaying the slave HTML-page. When pressing the initial 'Capture Blank', a picture will be held in the client side for later use in the corner detection algorithm(see 3.3). After the initial capture, we can proceed to display the colors on the last slave connected. We iterate from last connection towards the first connection. This we do by pressing 'Next Slave' and then press 'Capture'. Each iteration, the found corners will be displayed upon the canvas underneath all buttons. At any point in time, we can press the 'reset' button which in turn resets all taken pictures. All slaves and master will remain the same. We can then try another color and see what that output would give us. We also provided a debug mode which can be turned on by going to the web browser's console and writing following command:

```
Window.client.DEBUG = true
```

Its initialised on false and can be turned off by rewriting the command and replacing 'true' by 'false'. It does not reset when pressing the reset button.

To handle the debug mode, we proceed to run the application as usual, whenever pressing the capture button after having pressed next slave, we have to press on enter to see what algorithm 3.3 detects at each given moment. From color detection to providing the convex hull and then the final corners for our slave. As of now, we get stuck at the final step in debugging, which is displaying the final result after displaying the convex hull of the slave screen. We can exit this problem by resetting and turning debug mode off in the console.

# 3 Algorithms

We've reduced the complexity of the whole algorithm by refactoring the entire corner detection algorithm (see section 2). We got rid of the openCV-library that we were using

earlier, as do we have further developed the orientation algorithm. 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). To determine the orientation of a screen we use the coordinates from the four corners and a picture of the screen. We expect that the screen is divided into four colors. Using the coordinates we calculate a point in every area of color. For these for points we compare the color that is found with the color that is expected when the screen is oriented normally. Out of this comparison we can determine if the screen is rotated clockwise, counterclockwise or fully flipped. The exact angle at which this orientation occurs is calculated with the upper edge of the screen.

## 3.2 Slave Flow Handler

In this typescript file we attempted at creating a workflow for handling the correct and fluent manner. Every function within this module computes in constant time $\mathcal{O}(1)$. At first glance, this module keeps a hold of all slaves in an array an then later manipulates and computes all necessary information to make sure our algorithm works.

### 3.2.1 Initialisation

At initialisation we press the start button, which is called *CaptureBlank* on the UI. A deep copy gets made of the entire array with Slave IDs, such that we don't remove or add slaves to the original array located on server side. The current slave ID becomes the last element in the array, by always calling a 'pop' function on the array. Afterwards we draw the image underneath the buttons in the canvas and store the image in our class constant *blancoCanvas*.

### 3.2.2 Take Picture Of Colored Screen

A screenshot of the colored slave screen gets taken from the stream and displayed underneath the buttons. We will call the function $findScreen$, algorithm 3.3. Pressing the capture button again will display the corners of that given slave screen.

### 3.2.3 Show Next Slave

In this function we show the color on the next slave. We do this by displaying a white background on the slave that was lastly used, if there is any. Our current slave ID will display the color held by our client. The client gets updated to the color that the master chooses to use. This color is standard set to pink if no color is chosen.

## 3.3 Screen Detection Algorithm

Last week, we were advised to take multiple pictures (instead of one) to get more accurate data. A modified algorithm was created to take advantage of these multiple pictures.

### 3.3.1 Taking a blank picture

The first picture the user takes is an image in which all the slaves have a white screen color. This will be used to compare to images with colored screens. This allows us to rule out areas in which the screen is not visible easier.

### 3.3.2 Taking a colored screen picture & compare

Here, we display a color $c$ on a slave of which the screen is not yet detected and take a picture of it. It is crucial that this picture is roughly taken from the same position as the blank picture. Next, we loop over each pixel $p_c$ of the colored image. For each $p_c$, we take the pixel $p_b$ from the blank image with the exact same coordinates. If the colors of $p_c$ and $p_b$ are roughly the same, it is safe to assume that $p_c$ (and $p_b$) will not be part of the screen we are looking for. If $p_c$ and $p_b$ have different colors, we check whether $p_c$ has a similar color to $c$. If this is the case, we add it to a list of potential pixels $screen\_pixels$ that make up the screen.

### 3.3.3 Filtering noise

Now we attempt to filter out the potential noise which can occur when the camera was not completely steady or when the environment changed for some reason between the blank and colored picture. We do this in similar fashion to last week. But instead of looping over all pixels, we now only loop over $screen\_pixels$ found in the previous step. For each of these pixels, we check its neighbors in a given range. If the set of neighbors we check is not mostly of color $c$, we discard the pixel. We are currently still looking at different thresholds for the range and the colored neighbor amount, but we've been getting positive results with a range of 20px and 20% similarly colored neighbors. We took a threshold of 20% because a corner should not have more than 25% similarly colored pixels because it only shares one quadrant with the screen.

### 3.3.4 Finding corners

Again, we do this in similar fashion then we did in our previous implementation. We make use of the convex hull algorithm to get all the points which might be corners of the slave's screen. We then connect all these points and search for the two longest connections. If these two connections have points closer to each other than a threshold (we decided on 50px) then we discard the smallest of the two. We then search again for the two longest connections. We do this until we find two connections which have corners further apart than 50px. These corners will be the corners of our screen.

### 3.3.5 Repeat

The previous three steps will be repeated for each slave. At the moment, the user clicks a button to display the color on the next slave and takes a new picture when the screen lights up. We plan to automate this in the future.

### 3.3.6 Remarks

A downside of the new algorithm is that the user has to make several pictures from the same position. If the camera moves too much, the algorithm will not work. For example, if the user moves the camera left 50px between the blank picture and the colored picture, the corners will be incorrect by about 50px. We attempt to make it as easy as possible for the

user to take steady pictures by displaying the corners of the previous found screens. That way, the user has a reference to how the camera should be positioned (not implemented yet).

Also, we upscale/downscale the resolution of the picture to a resolution that fits 1280x720. Thus, we assume a landscape picture. We noticed that if the resolution of the camera is less than 640x480, the algorithm starts to struggle to correctly find screens.

We did add a visual debug mode to the algorithm which is still in development, but it allows the user to go over each step of the algorithm and visually see the steps explained above.

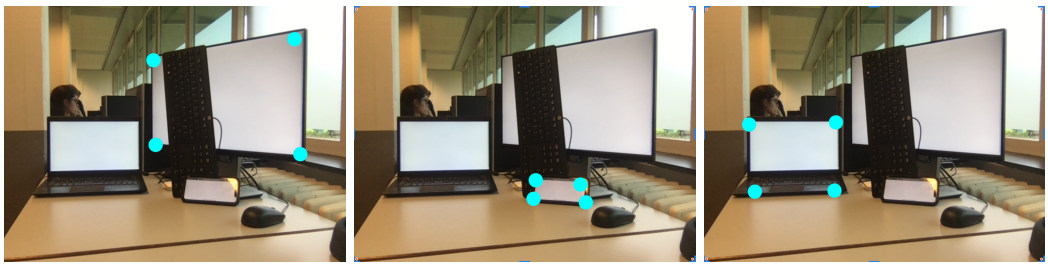
If no display is found for the slave, we should discard the slave gracefully without interference of the user. Currently, an error is thrown. We plan to change this in the future.

### 3.3.7 Performance

We have not performed formal tests yet, so we do not have graphs for this. But, we did time some random test and found that our algorithm takes about 400ms for a screen of +-200x50 pixels. It is evident that bigger screens will take up more pixels, because these will have more pixels to analyze. Finding a screen which took up about 20% of the picture took about 1.5 seconds. We plan on testing this in more detail in later stages, but are content with the current performance and thus prioritized other tasks.

### 3.3.8 Correctness

Our bottleneck in terms of correctness are the users. If they move the camera too much, accuracy will be lost. If the camera is moved to the point that the screens do not overlap any more between both pictures, no result can be found. We will try to prevent this by making it as easy as possible for the user as explained earlier.
Other then that, we have tested our algorithm with a few examples and it seems to work fairly well. Although, errors do occur from time to time as in the following example:



In the last picture, points bellow the screen are found because of the reflection of the screen on the glossy bottom part of the laptop. We are attempting to fix this by trying different combinations of thresholds.
The other screens, even the one which is partly blocked, are found correctly.
We plan on writing a more formal test suite which can be reused for screen detection, orientation and triangulation.

### 3.4 Triangulation

To generate the Delaunay triangulation, we found an implementation[2] of a divide-and-conquer algorithm. This was first presented by Guibas and Stolfi in: *Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams* [1].

The working of the algorithm goes as follows. In every step of the recursion the set of points gets divided along a vertical line. If one makes sure that the two resulting sets are of equal size, this will result in log(n) steps. Next we compute the Delaunay triangulation for each individual set, which we will merge later on along the line which was used for the split. This merging operation can be implemented in $O(n)$ time. Therefor the running time of the complete algorithm is $O(nlog(n))$.

In order to implement this algorithm we must be able to calculate the directional angle between to lines and to find the circumscribed circle of a triangle.

The algorithm works as expected as long as the center points of the screens are not on top of each other. We must ensure that this does not occur in any practical setup. In particular we say that a smaller screen may never lay perfectly centered on top of another larger screen.

The algorithm has as input a list of middle points of the screens and give as output a list of lines. These lines would be necessary to create the Dalaunay triangulation.
Using our algorithm, if we use the same input as in the implementation[2], we get the same result as expected. Another example we tested can been seen in these figures. In figure 2 we give the Dalaunay triangulation. In figure 3 we test if this is indeed a Dalaunay triangulation. We can speak of a Dalaunay triangulation if none of the circumscribed circles(of the triangles) contain another point.

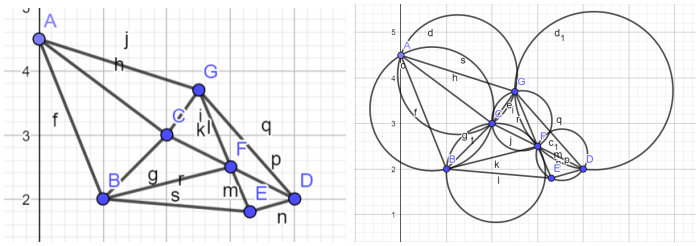



Figure 2: Delaunay Triangulation

In another test we did, we got the following result as seen in figure 3. Here we have an almost degenerated triangle and most certainly isn't a result we're keen on having. We can simply solve this by removing the rightmost line because it still gives a Delaunay triangulation. This has not been implemented yet.

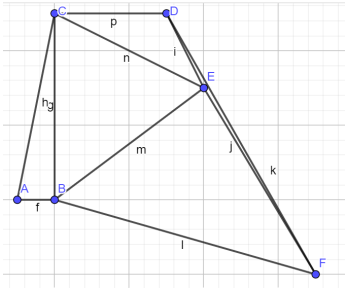



Figure 3: Test for Delaunay Triangulation

## 3.5 Corner labeling and angle computation

Once the four corners of a screen are detected using a uniform color as explained above, this output can be used to calculate the angle by which the screen is rotated (inside the plane of the picture). The benefit of first locating the corners is that it permits to limit the range of pixels to work on to only the range determined by the boundaries of these fours points. Then a second picture can be taken by displaying on that same slave four different colors split over the screen as it is illustrated in figure 4.

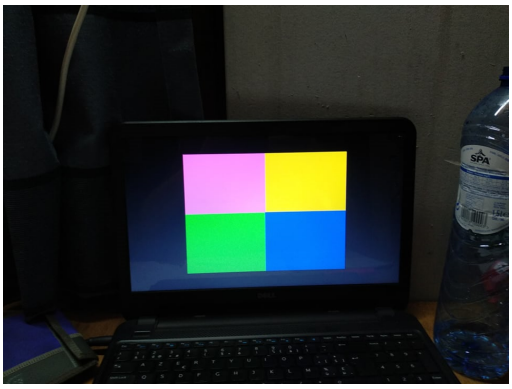


Figure 4: A screen with the basis case for the four colors displayed on it.

### 3.5.1 Computation of the centroids

The first step for determining the orientation consists of the computation of the four centroids of the rectangles determined by every color given coordinates of the four corners. This first requires to determine which of the four given points corresponds to which of the corners of the screen. It is therefore needed to first label the four given points. This can easily be done based on the x and y coordinate of every point. This way the point below right will have the highest x and highest y value (starting with an origin located above left).

Once the four points are labeled, the screen can be split in four parts as illustrated in figure 4. Then the algorithm for detecting whether the slave device is flipped by a certain angle computes the centroid for every rectangle. This is described in algorithm 1.

### 3.5.2 Determining the flip based on the colors

Once the centroids are computed all pixels around each centroid are checked for the color for a given range radius. The color can be one of the four illustrated in figure 4. The more pixels are colored by a certain color, the more certain it is that that rectangle represents the

**Algorithm 1** Computation of the centroid of four points

```
points  #The points representing the four corners of the detected screen
sumX  #Sum of all x components
sumY  #Sum of all y components
for all point in points do
  sumX += point.x
  sumY += point.y
end for
return  (sumX // 4, sumY // 4)
```

rectangle with that detected color. Based on that information it is easy to know whether the slave device is flipped and in which direction.

### 3.5.3 Determining the angle of rotation

Then the angle of rotation in the plane can be determined based on the line determined by the *leftUpper* and *rightUpper* points as it is illustrated in 5. The goal is to calculate the angle of the detected screen using this generated 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).

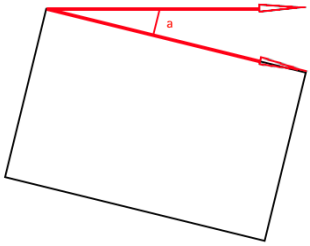



Figure 5: The vectors representing the angle of the screen's orientation

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 image 5.

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

$$\arctan\left(vector2.y - vector2.x\right) - arctan(vector2.y - vector2.x) \tag{1}$$

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.

### 3.5.4 The entire algorithm

Algorithm 2 below describes all what has been described above.

**Algorithm 2** Centroids based computation of rotation angle

```
points #The points representing the four corners of the detected screen
Computing the 4 centroids
    centroids = getCentroidsForPoints(points)
Labeling the points
    sumsArray #every element is sum of xandy
    rightUnder = max(sumsArray)
    leftUpper = min(sumsArray)
    sumsArray.remove(rightUnder)
    sumsArray.remove(leftUpper)
    rightUpper = the point left in array with highest x-value
    leftUnder = the point left in array with lowest x-value
Determining the angle
    points = updated to ordered corner coordinates
    vector1 #Horizontal vector starting from left corner
    vector2 #Vector going from left corner to rightcorner
    radians = getAngleBetweenVectors(vector1, vector2)
    degrees = radiansToDegress(radians)
Determining the Orientation
    centroids #List of centroids
    pixels #Data representing the image
    orientations #List of counters to check which orientation hadhighest probability

    for all centroid in centroids do
      Orientation = compareFoundColorWithExpectedColor(centroid)
      if Orientation == Normal then
            orientation[0] +=1
      end if
      if Orientation == Clockwise then
            orientation[1] +=1
      end if
      if Orientation == Counterclockwise then
            orientation[2] +=1
      end if
      if Orientation == Flipped then
            orientation[3] +=1
      end if
    end for
    index = getIndexWithMaxValue(orientations)
    orientation = getOrientationAssociatedWithIndex(index)
    return orientation
```

# 4 Conclusion and prospects

This week, we worked on the orientation of the slaves' screens in the plane of the camera. At present time there are still a number of restrictions that have to be met in order for the correct result to be attained. Chief amongst them is the fact that the center of all four coloured corner regions may not be covered and must be clearly visible. While the algorithm works, it is still a standalone piece of code. Further integration with the rest of the project will be scheduled for the coming week.

We have also spend time implementing a divide-and-conquer algorithm to compute the Delaunay triangulation. The algorithm works, though further research must be done in regards to its correctness in edge cases. We did not implement a connection to show the triangulation on the slaves' screen. This will be further explored in the coming week, though it will lose some of its relevance as future tasks will require the code to be rewritten to support more functionality.

Finally, we spent extra effort to improve upon our existing screen detection algorithm. We tested the algorithms more extensively in a practical setting with multiple screens and worked on streamlining the process with which users operate the web page.

# 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

## References


[1] Leonidas Guibas and Jorge Stolfi. Primitives for the manipulation of general subdivisions and the computations of voronoi diagrams. *ACM Tmns. Graph*, pages 74–123, 1985.

[2] Samuel Peterson. Computing Constrained Delaunay Triangulations, 1998. http://www.geom.uiuc.edu/~samuelp/del_project.html?fbclid=IwAR3p20UpNpSQSw7Z5k1rq7uPm2TXqnWdZ4o9pXej-g4Z2bZbuR_qyUclBYw.

[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. Delaunay triangulation (Wikipedia), 2019. https://en.wikipedia.org/wiki/Delaunay_triangulation.