1.5em 0pt

— Report P&O CW 2019–2020 Task 7 —

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

Team 03	
<Bram Vanbilsen>	0h
<Adam El M'Rabet>	0h
<Liam Volckerick>	16h
<Maarten Pyck>	0h
<Sebastiaan Wouters>	12h
<Pieter-Jan Van den Broecke>	12.5h

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 task were quite hard to wrap our heads around. We had to find a way to get the right image crops per screen, and send the result to each slave, trying to achieve as much image display as possible when casting. Our way of handling this issue thus far, will be discussed in section 2.1. In order to successfully compute each slave identity with its corresponding screen and coordinates, we made following assumptions:

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. 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 Algorithms

We will try to explain all steps needed to be taken to achieve the cat caster for displaying images across multiple screens. As it stands we have yet to implement all methods together to form one cohesive structure.

2.1 General Idea

The way we would like to process the cat caster follows next summation of steps:

a) Create a new canvas containing all coordinates of the slaveScreens. This will be a 480p resolution canvas (same resolution as the master image, thus no error margin in the coordinates of all points).

b) Form the bounding rectangle around all slaves' coordinates to form the least amount of image loss when casting the image to all slaveScreens. See subsection 2.1.2.

c) From item 2, scale this new found canvas to the resolution of our image to display. See subsection 2.1.3.

d) Cut out the content of the image, within the bounding box (2.1.2) of each individual slaveScreen for then to be processed.

e) Perspective transform all image content within the quadrilateral, formed by the slaveScreen corners, to the outer edges of the formed bounding boxes for each slaveScreen.

2.1.1 Initialisation

We copy all corner coordinates of all slaveScreens from the master result into a new object/canvas, after having calculated all corners and all orientations of all slaves.

2.1.2 Bounding Rectangle

We basically take the minimum & maximum X-value, the minimum & maximum Y-value. Hereby we can form the outer left and right corners of the rectangle around the given coordinates list. See following algorithm 1 for the concrete implementation. This algorithm Computes in $O(n)$ time. With n being the amount of points we make the algorithm process. We do have some remarks around this section. This will be discussed in section 2.3.

In following figure 1, we'll try and show how algorithm 1 works. Figure 1 contains the results of algorithm 1 over 2 different point sets.

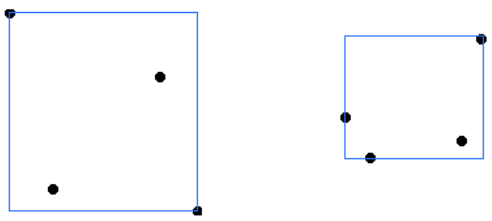

Figure 1: Algorithm 1 showcased

2.1.3 Scaling blank canvas to cat casting image

In order to be able to find what each screen must display on the slaves, we work with three different coordinate systems.

- The first coordinate system is the one which we receive from the algorithm that recognizes the corner points. This system lies along the axis of the original picture; the one taken with the camera from the master.

Algorithm 1 Compute Bounding Box Around Points

```
1: function BoundingBox(points)
2:     points  #The points representing the four corners of the detected screens
3:     tempMinX = 0
4:     tempMaxX = 0
5:     tempMinY = 0
6:     tempMaxY = 0
7:     for point in points do
8:         if point.X ≥ tempMaxX then
9:             tempMaxX ← point.x
10:        if point.X ≤ tempMinX then
11:            tempMinX ← point.x
12:        if point.Y ≥ tempMaxY then
13:            tempMaxY ← point.Y
14:        if point.Y ≤ tempMinY then
15:            point.Y ← tempMinY
16:    LeftUp = point(tempMinX, tempMinY)
17:    RightUp = point(tempMaxX, tempMinY)
18:    LeftDown = point(tempMinX, tempMaxY)
19:    RightDown = point(tempMaxX, tempMaxY)
       return (LeftUp, RightUp, LeftDown, RightDown)
```

- In order to avoid white space around the detected screens, we cut the original image. To do this the smallest bounding box gets computed, the process of which was explained in detail above. For the next step, we only need to translate all points to the center of the system by subtracting (smallest x-coordinate, smallest y-coordinate), as can be seen in Figure 2. This can easily be done in O(n).

- Now we have found coordinates relative to the smallest bounding box, we want to know how to scale them to best fit the image to cast onto the slaves. We cannot allow the image to be stretched, instead the bounding box will be scaled to fit the smallest side of the image, which may result in part of the image being cut. To do this, we compare the width of the image to cast with the width of the bounding box. We do the same for the height. This results in two ratios of which we will be using the smallest. All corner points are now multiplied by this ratio in order to scale them to the same dimension (Algorithm 2). This has time complexity O(n). Figure 3 shows the effect of the scaling.

It is easy to recognize that if the user wants to prevent a significant part of the image to be cut off in the resizing, they have to ensure that the dimensions of the slave screens' setup resembles the dimensions of the image to cast. Other than that this way of resizing should enable the use of a wide variety of images.

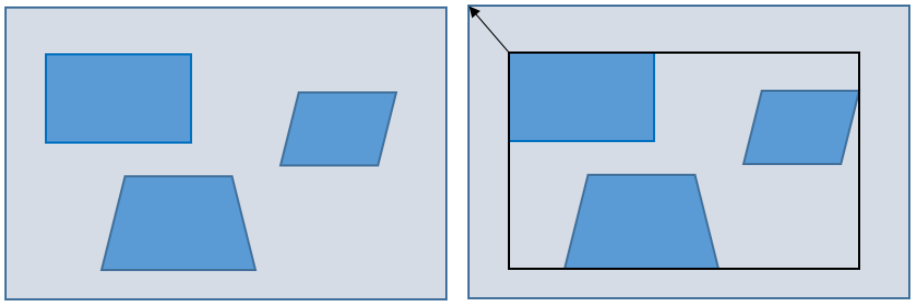

Figure 2: A picture with the recognized screens (left) and the translation of the bounding box (right)

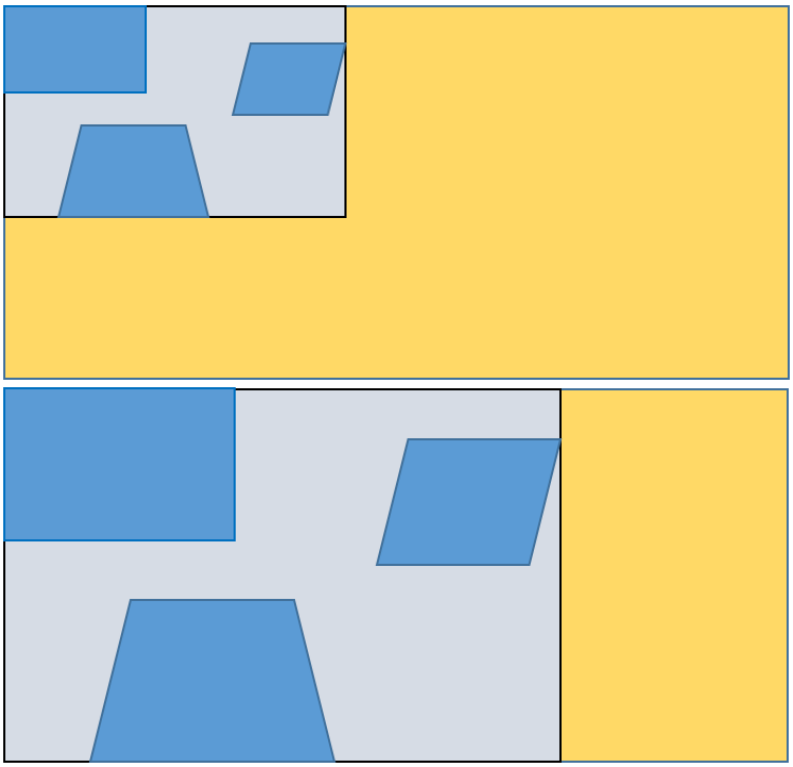

Figure 3: Scaling of the bounding box to the reference image (before and after)

Algorithm 2 Compute Bounding Box Scaled To Reference Image

```
1: function ScaleToImage(width, height, screenWidth, screenHeight, points)
2:     width  # The width of the reference image to scale to.
3:     height  # The height of the reference image to scale to.
4:     screenWidth  # The width of the image to scale.
5:     screenHeight  # The height of the image to scale.
6:     points  # A list with the points on the image to scale.
7:     xRatio = width / screenwidth
8:     yRatio = height / screenHeight
9:     ratio = minimum(xRatio, yRatio)
10:    for point in points do
11:        point = new Point(point.x * ratio, point.y * ratio)
       return points
```

2.2 Perspective Transformation

The corners of the screens, in point of view of the master, don't form a rectangle most of the time, but rather a generic quadrilateral. This is the case when screens are rotated in the plane. In these cases cutting out an image to display using the corner-coordinates wouldn't result in a usable option. The reason for this is that the cutout would be a quadrilateral and the screen is always a rectangle, so this doesn't fit. We assume no round or other shaped screens are used in the setup. To solve this, a perspective transformation from the quadrilateral to the desired rectangle must be performed.

this perspective transformation of the plane is uniquely defined by four source points and four destination points. The transformation is represented by a 3x3 matrix that can be obtained by the following equation:

$$\begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = \begin{bmatrix} h_0 & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & h_{22} \end{bmatrix} \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} \tag{1}$$

where (x1, y1) is the source point and (x2, y2) is the transformed point. If we calculate this equation using four source points and four destination points, the complete Holography Matrix can be derived. Using this matrix, points from the quadrilateral can now be transformed into the rectangle [1].

The way we implement this is described in the following paragraph. The algorithm expects a set of corners, the smallest bounding box of these corners, and the canvas containing the image data in the bounding box. First of all the Homography matrix is computed as explained above, using the four source points and four destination points. Once we have the transformation matrix, we apply it to all the points in the quadrilateral. Resulting in a map between the rectangle and the original screen. Using this map, it is now possible to transform all the pixels. This results in a rectangular canvas that contains the image to be casted to the screen.

Complexity:

The complexity of the algorithm is largely dependent on three parts:

- Calculating Homograph Matrix: This comes down to solving a system of linear equations. This particular case contains a 3x3 matrix, containing nine variables, that needs to be solved. Computing this is done by using Gaussian Elimination, which has a complexity of $O(n^3)$. with n being 3 in our case.
- Iterating over pixels: For every pixel, a transformation must be executed and the color of the pixel must be changed. These two steps are constant in time. So the complexity is completely dependant on the amount of pixels: $O(n * m)$. With n: width and m: height.
- Amount of screens: The two steps explained above need to be executed for every screen independently. So, with n being the amount of screens, the total complexity would be $O(n * (width * height))$. with width and height being the sizes of the screen with the largest dimensions. This is calculated for the worst case, where all screens have dimensions equal to the largest screen. The reason that the complexity of the Gaussian Elimination is not taken in to account, is because compared to the manipulation of the pixels it takes orders of magnitude less time.

2.3 Remarks & Problems

As it stands, we basically find the rectangle that bounds the corners of each quadrilateral by having its top and bottom edges perpendicular to the monitor/screen. This could create quite some big inconveniences when processing the perspective transformation 2.2. Such that we can create a more dynamic and fluent image cast when having done the perspective transformation. Which should give a better result in the end when presenting our Demo. We believe this issue can be resolved by calculating the smallest bounding box which can be in a certain rotation around its given set of points, while also keeping track of the orientation of the slaveScreen. As of now, we have implemented this smallest bounding box in python. This is due to ease of testing code there, you can find the file in our branch under following path $master/public/scripts/image_processing/ImageCasting/sBB.py$.

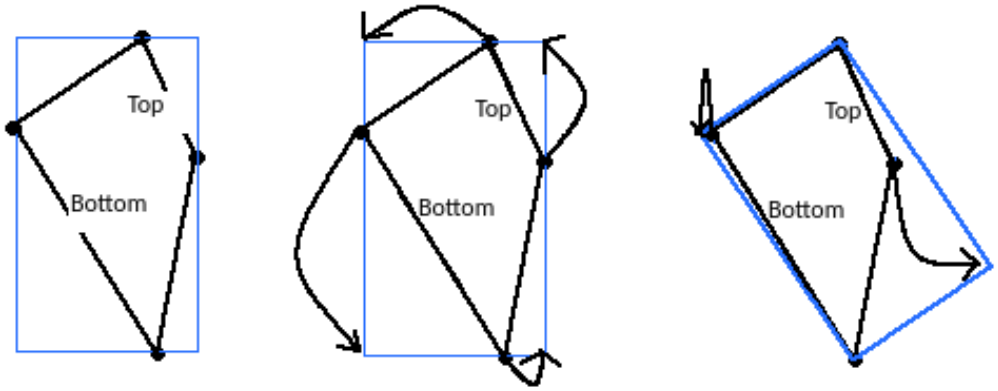

Figure 4: Algorithm 1 problem

3 Conclusion and prospects

This week, we worked on the question on how to handle synchronization between the server and its clients. This was done with the countdown task in mind. We encountered some

initial problems with the timings, but these were eventually fixed. By the end we had a working implementation with limited delay between the slaves.

The second focus this week was the casting of the image to the screens. This was done in two parts: master-side and slave-side. On the master-side we looked into how we can trim the image taken by the master to its basic composition and how to scale it to fit the reference image. This process is not lossless due to the fact that the image cannot be stretched.

On slave-side we tried to find a way to transform a quadrangle into a rectangle. We encountered some setbacks early on, which slowed the process down. We managed to find a working theory, but further work must be put into the implementation and the merging into the wider project.

This last topic will be the most important point of focus for the coming week, as it is a key piece of code that will be necessary to tie the rest of the project together.

4 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] Satya Mallick. Homography examples using opencv. *learnopencv.com*, 2016. https://www.learnopencv.com/homography-examples-using-opencv-python-c/.

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