1.5em 0pt

— Report P&O CW 2019–2020 X —

Department of Computer Science – KU Leuven August 18, 2020

Team 03	
<Bram Vanbilsen>	90h
<Adam El M'Rabet>	26h
<Liam Volckerick>	60h
<Maarten Pyck>	71,5h
<Sebastiaan Wouters>	36,7h
<Pieter-Jan Van den Broecke>	64h

Demo: https://penocw03.student.cs.kuleuven.be/

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 biggest challenges in getting to where we are now, were the ones in which we had to solve a problem and determine the best possible solution, which also had to be as least error prone as possible. Each individual task came with its own challenges, having to make us revisit all code that had been written prior.

The full application rundown is discussed in 2 and in all of its subsections. From starting a simple chat application that can interact with the server, to making a full on online application that can process screens connected to the server, and manipulate their browsers, is a real accomplishment. The main challenge that lies ahead of us, is further perfectioning of all that has been done so far. We hope that all information underneath clarifies what we have accomplished as well as showcase&explain how we tackled each given problem.

2 Application workflow

The application works as follows starting from the boot up.

1) Client connections with the server(see subsec. 2.1).
2) Close list of connections and start screen detection(see subsec. 2.2).
3) After a screen is detected, orientation calculation is called(see subsec. 2.3).
4) We scale the image to cast(see subsec. 2.4).
5) Cast the images(see subsec. 2.5).
6) Calculate Triangulation(see subsec. 2.6).
7) Get all clients synchronized(see subsec. 2.7).
8) Now all is synchronized, we can either call countdown or animation. The order of these calls don't matter. Countdown(see subsec. 2.7.3) & Animation(see subsec. 2.8).

2.1 Client registration

2.1.1 General idea

In this first section we handle the clients registering themselves with the server and taking up either the master or slave role. This is done by using Socket.io as recommended in the beginning of the project. We also have to ensure that the correct channels to the server are initialized depending on the client's job.

2.1.2 Becoming a client

Using Socket.io we listen on the server for any connections. These come from the registration started on the clients' HTML pages. The server will add every new arrival to its list of connections. There are two different scenarios that can occur at this point:

- There are no other connections. If this is the case this first client automatically becomes the master.
- There are already existing connections. All new clients become slaves and the master is informed of the current list of slaves.

This differentiation between master and slave is communicated back to the client in question through the channel *SharedEventTypes.NotifyOfTypeChange*. Upon receiving this message the other channels and their respective functions are initialized. This is a secure way of working as it ensures that a slave cannot listen on channels meant for the master and vice versa.

2.1.3 Becoming the master

It is important to understand that while we send a client's status through *SharedEvent-Types.NotifyOfTypeChange* this is not in fact how the master is saved on server side. All new connections are pushed to the back of *Array<socketio.Socket>*. The connection at the start of this array is considered to be the master; it is therefore not explicitly saved.
When any changes occur to the connections, the master is automatically informed, so that he continues working with the most up-to-date list.

2.1.4 Ease of use

The fact that the master is the first client to connect is a simple solution, but it is not particularly easy to use or flexible. Early on in the project it necessitated reloading all clients, starting with the desired master, in order to test. Therefore we implemented a number of functions to increase the ease of use of the application:

- When one presses the r-button on the keyboard, the server resets the background colour of all slaves to their initial background colour.
- On the server we can enter *kill_all*. This allows for the server to forcefully disconnect with all clients. Additionally *kill_master* and *kill_slaves* also exist. Their usage is self-explanatory.
- The function *giveUpMaster* allows for the master to give up its status and become a slave. In the default version the second connection in *Array<socketio.Socket>* becomes the master - the current master is removed and added again to the end of the list. It is however possible to enter the id of the specific client one wants to become the master.

While these functions are working, they are not as of yet implemented with a button. To access this functionality, one has to run them through the command line.

2.2 Screen Detection

2.2.1 Input and Idea

This algorithm is used to identify individual screens. It takes two images (img_1 and img_2), one being a picture in which the screen in question has color c_1 and the other one being a picture in which the screen S has color c_2. By comparing these two images, an attempt is made to locate S. After this, filtering steps are applied to reduce the comparison result to just 4 pixels. These are the corners of S (Note that the location, position and visibility of the screens should comply to certain rules and limitations that are explained in the section 4.3 and 4.4.).

2.2.2 Image comparison

In this step, each pixel (x_i, y_i) in img_1 is compared to (x'_i, y'_i) in img_2. If these pixels have similar HSL colors, they are discarded. Two pixels have similar HSL colors if:

- The H values for both pixels are no further apart than 50.
- The S values for both pixels are no further apart than 40.
- The L values for both pixels are no further apart than 40.

The ranges were chosen manually through trial and error to maximize the search range while still making a clear distinction between different colors.
If the colors of (x_i, y_i) and (x'_i, y'_i) are not similar, a filtering step is made to only accept pixels that could be on the edges of S. This is done by checking whether the color of each pixel in img_2 within a radius of 2px from (x'_i, y'_i) is similar to the color c of (x'_i, y'_i). The percentage of similarly colored pixels in this range will be referred to as sr in what follows. It is evident that no more than 50% of these checked pixels should have a similar color to c if (x'_i, y'_i) is on the edge (otherwise, it would be inside of S). But, because cameras are not perfect, we assume (x'_i, y'_i) could be on the edge if $sr \leq 55\%$.
A lower bound check is also in place. Since if $sr < 25\%$, it can't be a pixel on the edge (which include corners, hence the 25%). Again, because cameras are not perfect, the assumption is made that if $sr \geq 10\%$, (x'_i, y'_i) could still be a pixel on the edge.
Therefor, this step from the algorithm essentially accepts all pixels (x'_i, y'_i) in which the following conditions are met:

- The color is different compared to (x_i, y_i)
- $sr \leq 55\%$
- $sr \geq 10\%$

The set of pixels fullfilling these conditions will be referred to as *accepted*.

2.2.3 Filtering to four corners

In most cases, *accepted* will have a lot more than four items in it. To reduce this set to only four corners, the convex hull CH from *accepted* is calculated. In the ideal case, CH will hold just four points. In which case, the corners are found. But again, due to camera imperfections, it is very likely that CH has more than four elements. The reduction from CH to four pixels (the corners) is done as follows:
All possible connections between points in CH are made. Assuming the image comparison step went well (no outliers in *accepted*), the longest connection $diag_1$ holds two of the wanted corners. This is most likely to be a diagonal of S. To find the last two corners, the second longest connection $diag_2$ is found. If $diag_2$ has end points in a range of $50px$ from the endpoints of $diag_1$, we discard $diag_2$ (see 4.4). Now the third largest connection $diag_3$ is found and the end points are again checked to $diag_1$ as above. This process repeats until a connection $diag_i$ is found in which the end points differ enough from $diag_1$.
The corners of S are now the end points of $diag_1$ and $diag_i$.

2.2.4 Notes

- This algorithm detects one screen at a time and will thus be executed for each slave.

- Finding the correct 4 corners relies heavily on the assumptions in 4.3 and 4.4. Mainly the quality of img_1 and img_2 has a big impact. Not just the quality in terms of the amount of pixels is important. Rather, the quality in terms of how little the images differ (except for the part in which the screen is visible) from each other is very important.

2.3 Angle detection

To be able to cut images and display the triangulation it is essential that the angle between the screen and the horizontal axis is known. In order to calculate this angle, the correct edge of the screen is needed. The edge should always be the same for a given screen, no matter the rotation. To find such an edge, the screen is divided into 4 colored areas as seen in figure 1.

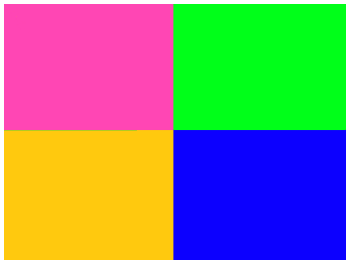

Figure 1: The image displayed on the slave to derive its orientation from

A decision was made to always select the edge which separates the pink and green area. Any another edge could also be chosen, it just needs to be consistently the same.

To find this edge E, the diagonals between the centroids of the screen and each corner is made, this results in 4 diagonals. For each diagonal, the algorithm checks how many pink pixels are on it. The pink area is identified as the area of which the diagonal has the most pink pixels. The corner A of the pink area is now known and is always taken as one of the endpoints of E. From this information, the edge E can be identified:
Name the corners as seen from the image (without any notion of orientation) as follows:

- TL = top left corner
- TR = top right corner
- BL = bottom left corner
- BR = bottom right corner

The second corner, named B, of E can then be found as follows:

- $A == TL \iff B == TR$
- $A == TR \iff B == BR$
- $A == BR \iff B == BL$
- $A == BL \iff B == TL$

With this, the angle of the screen with the horizontal axis is calculated using the $atan2$ function from javascript as follows:

$$Math.atan2(B.y - A.y, B.x - A.x)$$

2.4 Image scaling

The following steps are taken in order to be able to find what each screen must display on the slaves:

1) Scale the image along the x and y axis to fill the bounding box around all screens (call the box BB).

2) Find the largest width $width_{max}$ and largest height $height_{max}$ of all screens.

3) Create a new canvas with the following dimensions:

   - Width: $BB.width + (width_{max}/2)$
   - Height: $BB.height + (height_{max}/2)$

   The extra length $(width_{max}/2)$ and $(height_{max}/2)$ are added because the canvas should be big enough to fit all screens in all rotations. This is illustrated in figure 2.

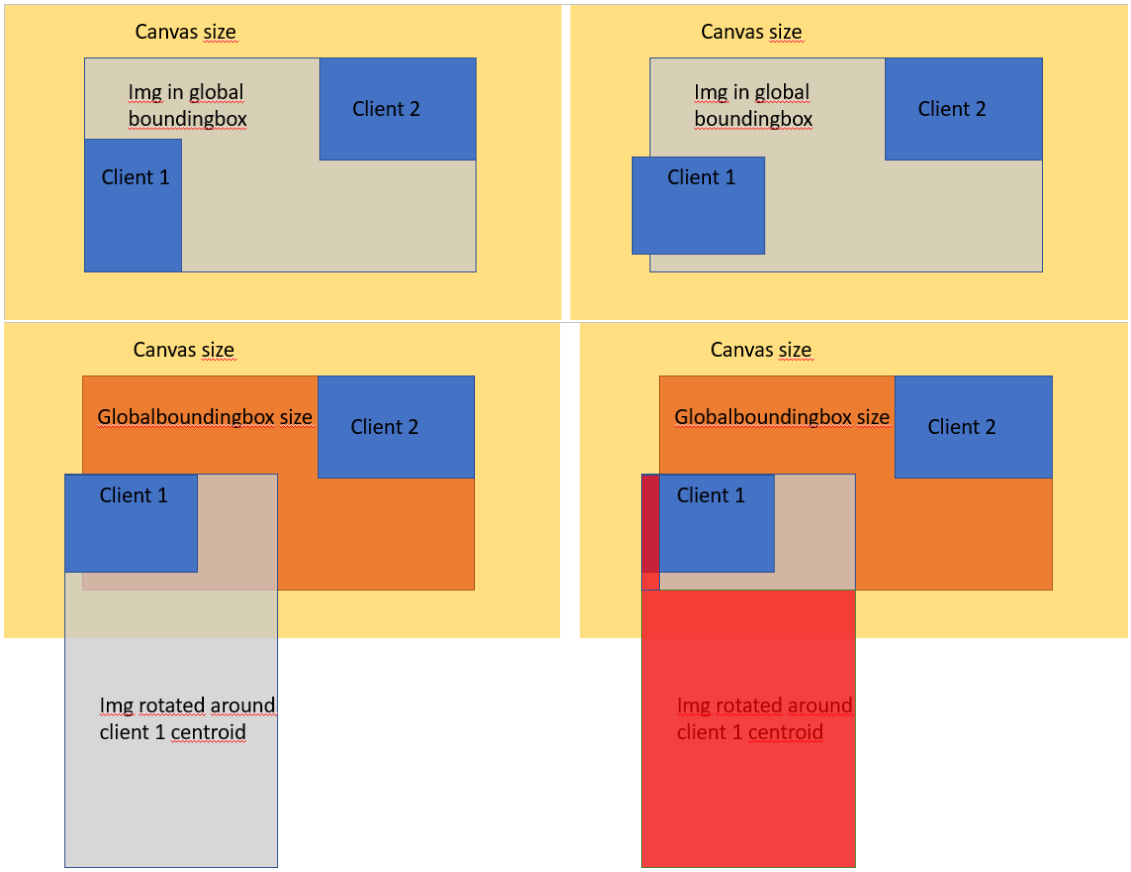

Figure 2: Reason for expanding the canvas rather than using a canvas of size == global bounding box

2.5 Casting the images to Clients

The workflow after re-scaling the image to the global bounding box:

- Rotate client's corners by the negative of the screen's angle around its centroid

- Rotate the image around the client's centroid by the same angle as above
- Mask all of the canvas apart from area defined by the client's rotated corners
- Cut out the non masked canvas area and send this to the slavescreen
- Reset all the points and the image
- Go to the next slave
- Repeat

Here, it becomes more clear why step 3 in section 2.4 is necessary. The screen and image get rotated which may lead to them having coordinates out of the canvas if the canvas is only the size of the bounding box of all screens.

Figure 3: Example of image casting

2.6 Triangulation

2.6.1 General idea

The goal is to connect the centers of all the screens with lines to create triangles without lines intersecting each other. There are a lot of different triangulations, but not every one of these is interesting. A triangulation with an interesting property is the Delaunay triangulation; Every circumscribed circle of a triangle has no other points lying in it.

2.6.2 Triangulation explained

To generate the Delaunay triangulation, we found an implementation[5] 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* [4].

The algorithm works as follows. First of all, the set of points is sorted based on their x-coordinates from small to large. When the x-values are equal, the y-coordinate will be the deciding factor. Once the points are ordered, this ordered set is successively divided into two halves until we are left with subsets containing no more than three points. These subsets may be instantly triangulated as a segment in the case of two points and a triangle in the case of three points. In the case of one point we do nothing.

After this step is completed, all the sets need to be merged back together recursively. This merged set will contain edges from both sides as well as some newly defined edges between the two sets. To maintain the Delaunay properties some original edges may need to be deleted. Newly created edges will not be deleted in a merge, but can be deleted in a next merge.

Finally, we will have only two sets left. These are then merged. The edges that need to be added are calculated and some edges out of the original sets are deleted. This completes the algorithm.

In order to implement this algorithm there must be an option to calculate the directional angle between two lines and the circumscribed circle of a triangle. This is crucial to be able to calculate which new edges need to be added when merging two sets of points.

The algorithm works as expected as long as the center points of the screens are not on top of each other. When this is not the case, the algorithm will always work. if there are two points with the same coordinated the algorithm will fail, because, for the algorithm, it looks like two different points. The algorithm is able to create a line between these points, but if we then calculate the angle between a line starting/ending in this point and the line with the same endpoints it goes wrong. We always get the angle of the line with not the same endpoints and the x axis or zero. This can lead to the algorithm not detecting lines that need to be deleted in the merging step. Also the resulting circumscribed circle will be wrong. 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.

As input the algorithm has a list of centers of the screens and as output it gives a list of lines. These lines represent the connections between the different screens.
Using our algorithm, using the same input as in the implementation[5], we get the same result as expected. Another example we tested can be seen in these figures. In figure 9 we give the Dalaunay triangulation and we test if this is indeed a Dalaunay triangulation.

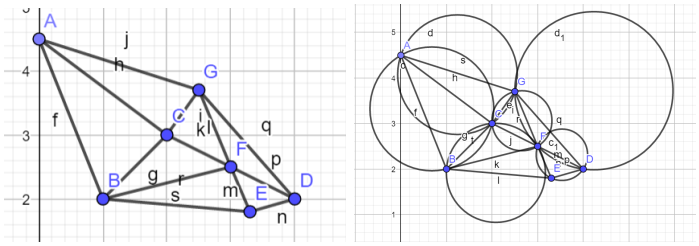

Figure 4: Delaunay Triangulation

In another test, we got the following result as seen in figure 5. Here we have an almost degenerated triangle and this 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 yet been implemented.

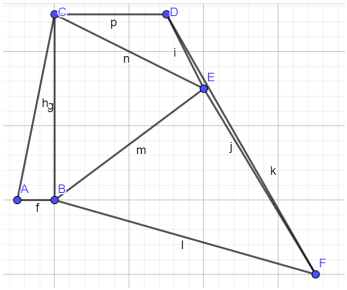

Figure 5: Test for Delaunay Triangulation

We give some more results from tests made with the option for the master to choose random points to calculate a triangulation. We also give the result when adding one more point.

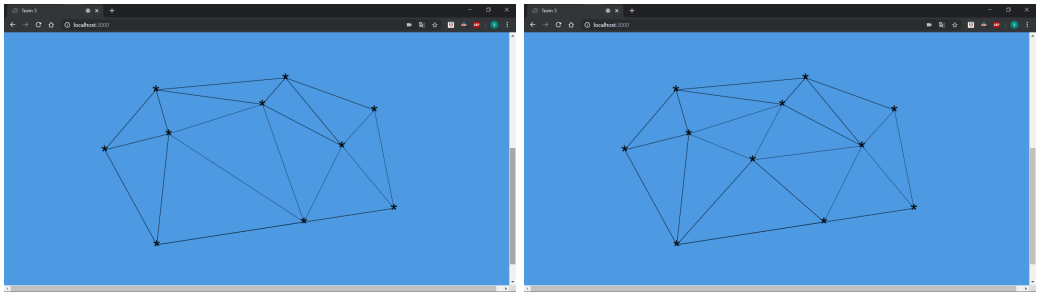

Figure 6: Delaunay Triangulation Test

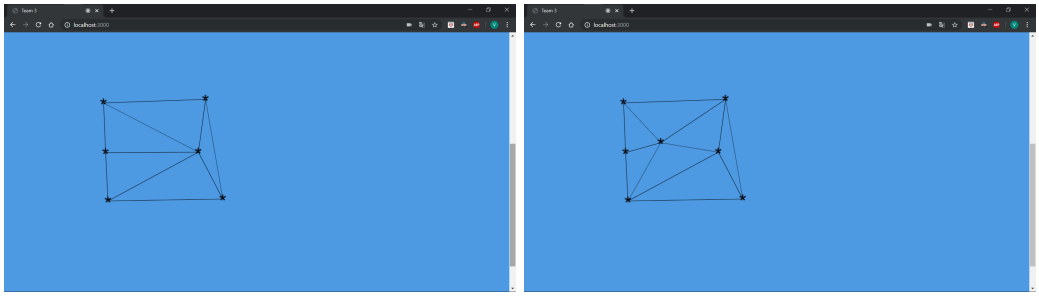

Figure 7: Delaunay Triangulation Test

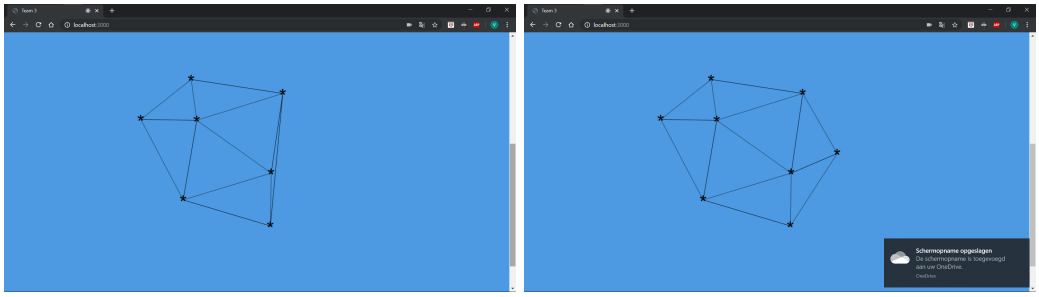

Figure 8: Delaunay Triangulation Test

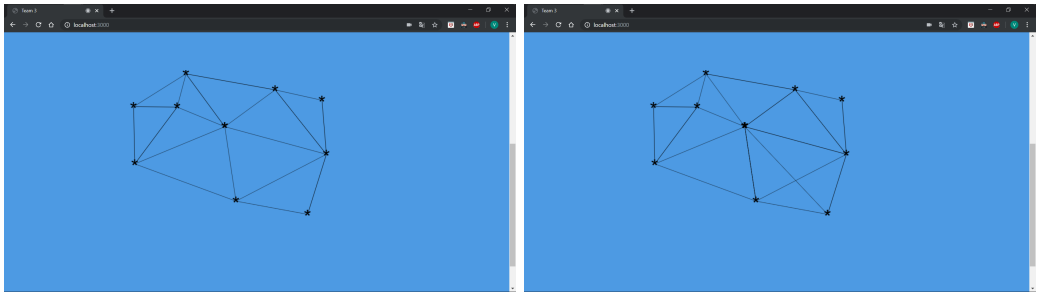

Figure 9: Figure representing the problem that can arise when two points overlap.

2.6.3 Complexity

The algorithm consists of two steps that add to the complexity.

- Sorting all the points based on their x-coordinate: This step takes $O(logn)$ time, where n is the amount of points.
- Merging all the log(n) parts together. This step requires some calculations to be performed. Using some clever tricks, all merges can be done in $O(n)$ time.

The complete algorithm will be performing in $O(nlogn)$ time.

2.7 Synchronization

2.7.1 General idea

Synchronization is the coordination of events to operate multiple systems in unity. If multiple entities are synchronized, it becomes possible for them to perform tasks at the right time with regards to each other. To make this possible, we attempt to eliminate differences on the clock of our different slaves by joining them to a shared time frame.

2.7.2 Synchronizing time Client - Server

When a new slave connects, it calculates the difference between its own time and the server time. That difference is updated every second and then stored as an attribute of the client object. This helps the correctness of the synchronization when the delay of the connection fluctuates. The time difference at the start of a new connection is higher than after a while. To take this into account, at the beginning of a connection some delay is given to a newly connected slave before activating the countdown. The way the difference is computed is formally described as:

$$Date.now() - data.t1 + (Date.now() - data.t0)/2;$$

$Date.now()$ returns the current time of the slave. $t1$ is the time of the server directly sent via a socket emit. $t0$ is the time of the slave itself after it has been sent to the server and received back by the slave. $(Date.now() - data.t0)/2$; is divided by two because it includes going back and forth. While the first part: $Date.now() - data.t1$ computes the difference between a one way emit.

Once the difference in time is computed (which is done every second) it is stored as an attribute of the slave object. As soon as the order for starting the countdown at a given start time is emitted by the server, Each slave uses the lastly stored delay (time difference with server) for computing the start time relative to that slave.

2.7.3 Countdown

When the master wants to start a countdown he emits to the server, via socket.IO, the start time for the countdown (currently set to 10 sec after the emit). That time is still relative to the server. All slaves have ten seconds to compute their own start time using the delay with the server which they already computed (see section 2.7.2). Once the start time is reached, every slave will start in a synchronized way to count from 10 to zero[8]. Once zero is reached two images of the popular Minecraft Creeper figure alternate giving an impression of creeper trying to explode.

Algorithm 1 Countdown

```
count  #The time the timer needs to countdown from
starttime  #The server time when the counter needs to start
starttime ← (newDate()).setSeconds(endtime.getSeconds() + x(count))
Emit start time of server to all slaves
for slave in slaves do
    starttime += lastly computed time difference
    do countdown for 10 seconds
```

2.8 Animation

2.8.1 General idea

At this point we would like to show a moving entity on the screens. This is a task that requires several -if not all- of the sections above. After all, the entity will traverse the triangulation to reach the other screens' middle points and synchronization will be unmissable for the animation to show correctly.

2.8.2 Linking lines to slaves and vice versa

Before we can start the animation we need to calculate the triangulation. This is a class that saves the lines of the triangulation. Of course that is not enough information to make an animation. Thus after the calculation of the triangulation, more information should be added in the triangulation instances and the slave screen instances. In the triangulation, we check and link each line that intersects with a slave and store this information(from the master's perspective). We also keep track of where the slave screen's edges intersect with the line that is chosen to display the animation on and the orientation of the intersection(s) with the use of a string("l" = left, "u" = up, "r" = right, "d" = down). In each slave we

save all the lines that are visible on the screen using the intersection points with the screen edges. All the calculations are made from the master perspective. At last we also link each centroid to the lines of the triangulation that start/end in this point.

2.8.3 Structure

To start the animation, we first choose a random point to start from. Then we choose a random line that has an endpoint in this point. We now use the information that has been saved in the triangulation to get all the slaves that intersect the line. We then emit to each slave the following information: when the slave needs to start an animation on his screen, the duration of the animation, the lines it need to draw(this information is saved in each slave screen, see section 2.8.2) and the line the circle needs to follow. The slaves then sync the start time they received(see section 2.7: Synchronization) and start the animation at that start time. The last slave to finish his animation sends an emit to the server telling that the animation is finished and then we can redo the previous procedure starting in the point where the circle finished. Waiting for the emit of the last slave before starting emitting information is necessary because otherwise there is a chance that the emits made to one slave overlap and this gives us glitchy animations. Waiting till after the last slave notices that he the server is finished makes you sure that all the animation on the screen have ended. We can summarize the procedure as follows.

- choose a random starting point
- choose a random line containing this point
- for each slave containing this slave
  - Emit starting time, duration, lines to draw and line the circle needs to follow
  - the slave start his animation on the synced starting time and if this is the last slave he emits to the server that the animation is finished
- If the server has received an emit from the last slave the procedure restart from step 2 with the endpoint from the previous line.

2.8.4 Sending emits to slaves

When we have a line where we want the circle to move over, we need to send the correct information to all the slaves that intersect this line. The first thing that has to be calculated is the starting time. It is the time the circle needs to move from the starting point to the first intersecting point with the line and the screen edge(from the master perspective). Next we need to calculate the duration, this is how long the circle is visible on the screen(from the master perspective). If we use figure 10 we get the following formulas.

$$startingtime = ||K - I||/speed$$

$$Duration = ||L - K||/speed$$

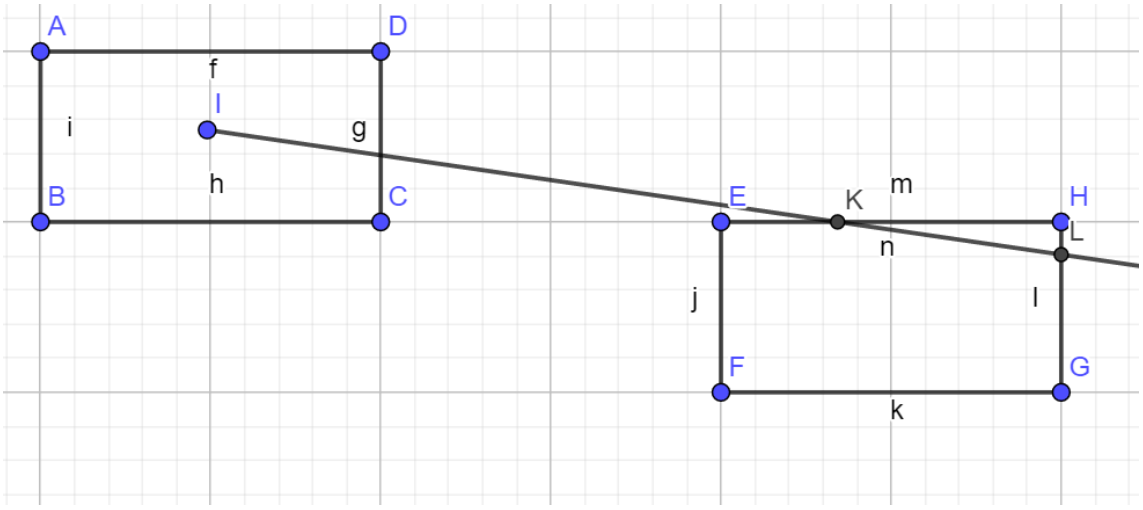

Figure 10

We have chosen the speed to be $0.05 pixels/ms$. If the screen has only one intersection we need to use the centroid as the other point.

Next up are the lines the slave needs to draw and the line the circle needs to follow. The first is saved in the slaves and the second is the line that connects the intersecting points with the line or in case of one intersecting point, the line between the centroid and the intersecting point. Of course we cannot emit these lines like this. We calculate the ratio of the intersecting points and emit this to the slaves. If we use figure 11 we get the following formula to calculate the ratio(this is for the intersecting point in the other edges have an equivalent formula)

$$ratio = ||K - E||/||H - E||$$

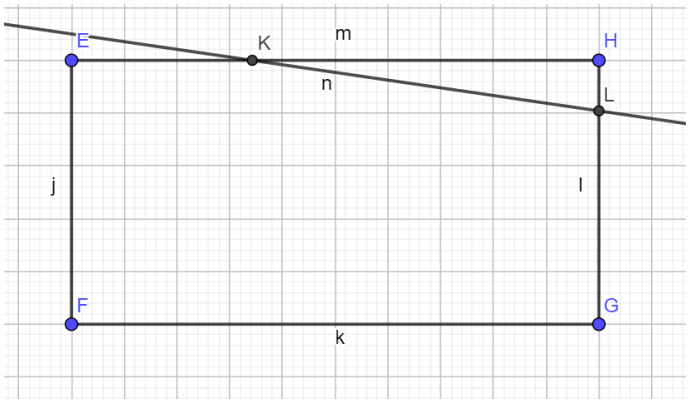

Figure 11

We also emit which edge each intersecting point is located in the form of a string, this information was also saved in the slave as the orientation of the intersection.

2.8.5 Animation on slaves

When the slave has received an emit with the information for the animation, he needs to sync the start time and transform the ratio into points on the edges. Then the slave waits

until the start time using *settimout* and then starts the animation using *setinterval* that finishes on time $starttime + duration$. We have chosen that the animation works on 12,5 frames/second and each frame the slave draws all the lines, it's centroid and the circle. It then hanges the x -and y coordinate of the circle according to the line the circle needs to follow. When the animation is finished and the slave contains the endpoint of the line, meaning the circle stopped in his centroid, he emits to the server that the animation has been finished.

2.8.6 Problems and remarks

After the server receives an emit from the last slave containing the information that the animation has finished, we delay the next movement of the circle with 100 ms. Reason being that due the fact that there is a little delay on the emits and small errors on the sync of times.
If we hadn't done this, then the first slave would begin the animation after the start time, meaning that the circle would disappear before reaching the end.This only happens in the slave where the circle needs to move further on a other line after crossing the centre. Even with this little delay this phenomenon can still occur, but we cannot make the delay higher without changing the smoothness of the transitions in the centres. We fixed this problem by telling the slaves that when the end time has been reached, and the circle is still visible on the screen, that the animation should proceed moving. Of course this means that the duration of the animation will be longer than calculated on the master. The next slave does not know this and starts on the correct time.
If the two slaves lie perfectly next to each other, we see two circles moving, one on each screen. In figure 12, we see this happen. We can now choose to not use the delay but that would make the previous mentioned problem only worse.

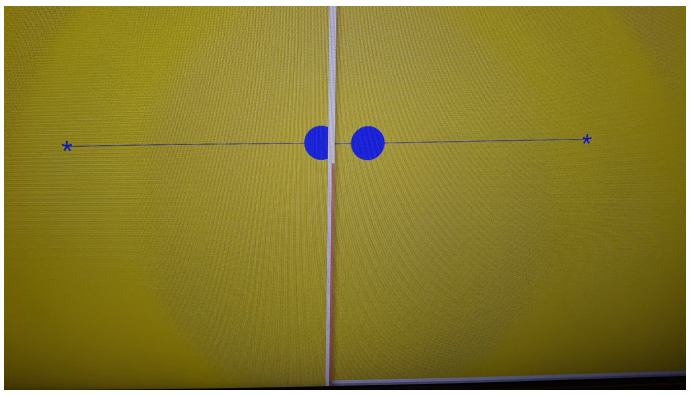

Figure 12

3 Promises

3.1 Detection of slave-screens

If the screens of all connected slaves are recognized, the application will continue to run. Otherwise an error will be thrown. There is no guarantee that a screen shall be detected if the assumptions are not met.

3.2 Angle detection

The angle will be detected if:

- The screen is detected correctly
- The pink area is not covered
- The image is decent in a way that the pink can be detected

3.3 Synchronized countdown

One of the buttons available on the master screen (after the detection process), is a button for starting the countdown. An io-emit will be sent (as described earlier in this report) from the front-end of the master to the server. This will on his turn emit the start-time (which is currently set at 10 seconds after the emit) to all slaves. The start-time will be stored as a relative time according to the delay of every slave. After those 10 seconds, the countdown will start on all devices (whether these devices are detected or not. It is only required for the devices to be connected). The countdown displays a big downwards counting number taking the whole visible screen. Scrolling during the countdown process will cause the displayed number to follow the scrolling and therefore won't be affected by the current "scroll-position" of the screen. The number will in all positional cases be visible on the screen. An example of this is illustrated in figure 13. Furthermore this number will be displayed above whatever is currently displayed on the slave-screen. So if a part of the unicorn is displayed before running the countdown it won't be overwritten by the numbers but the numbers will instead be displayed "above it".

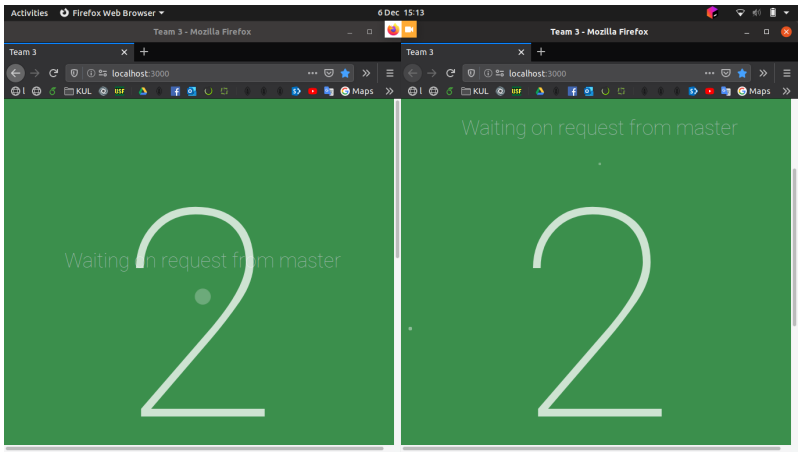

Figure 13: A synchronized two displayed on two slave windows on the same screen running with the localhost server

The countdown does not adapt its display according to the orientation of the slave-screen but it is instead displayed as if the screen is not rotated.
The countdown is synchronized. Only very small differences (less than 1 second) might be noticed in worst case.
At the end of the countdown (i.e. when reaching zero) a small animation displaying two alternating figures representing the Minecraft Creeper[1] character. Sometimes the figure

[1]Creeper is a character from the popular Minecraft video game which explodes when getting too close to the main character controlled by the player of the game

fails to alternate between screens, which have to do with CSS.

3.4 Display of image (unicorn)

Perspective is not implemented, thus we can only work in 2D field. If our setup holds up to all assumptions made in section 4, then our application will work as expected.

4 Assumptions

4.1 Server

- There is no prerequisite to join the server.
  - Reason: All new connections are automatically accepted and given a role.

4.2 Client

- Synchronization sensitive functions should not be called directly after a client connects to the server.
  - Reason:
    - External packages (e.g. jQuery, Bootstrap) can still be loading. Which can slow down other connections (to the server for example).
    - The client and server communicate every second to calculate the emit time. The last ten results are stored to calculate an average from.
- All clients should use a sufficiently modern browser.
  - Reason: The extensive usage of *let* necessitates the support of the browser.

4.3 Analyzed pictures

- A room with light.
  - Reason: If the room is dark and a picture of a colored screen is taken, a glow even around the screen is often observed. This glow will have similar color than our screen and could wrongly be detected by the color detection as part of the screen.
- No lights in the camera view.
  - Reason: Colors of pictures taken by a camera are often not exactly correct. Bright lights often have a shade of light pink which might be detected by the color correction algorithm.
- Pink objects that are not screens can only be in the master image if they are in every picture taken. Otherwise this will cause screens to be detected incorrect.
  - Reason:

        * Imagine that the first picture, with all the blank screens is taken. Afterwards, a pink object enters the scene, and an image with a pink screen is taken. Both the object and the screen will create a difference in color with the original image. If the object is big enough this will cause the screen to not have the right dimensions.

4.4 Screen detection

- Screens have a width and height of at least 50 pixels.
    - Reason: At the end of screen detection, we are left with a set of pixels assumed to be on the edges of the screen. We use the end points of the longest two connections as corners. To assure we essentially do not end up with the same corners twice, we assume that if there are two end points of these connections closer than 50px apart, they are the same end point.
- In the picture, the edge pixels should, in a range of 2 pixels, have at least 10% pixels with a similar color around them and max 55%.
    - Reason:
        * 10%: The pixel is considered an error point (not part of the screen).
        * 55%: The pixel can't be part of the edge because it has neighbors on more than two sides. We take a 5% error margin to allow detection of points close to the edge as well.
- The pixels of the two images used for the image comparison should be as similar to each other as possible, except for the pixels which are part of the screen to search.
    - Reason: These two images will be compared to each other. This is done to eliminate the problem of accepting a lot of error pixels that are not part of the screen when only one image is used with color detection. The less similar the pixels of the both images are, the closer the algorithm gets to the problem of only using one image.
- If the camera is not steady during the process where the different pictures are taken, detection of the screens will work, but the coordinates of the corners will be off by the amount of pixels proportional to how much the camera moved.
    - Reason:
        * For every screen we take a picture where the screen is blank and a picture on which the screen displays bright pink. By comparing these two, the corners can be detected. If the second picture is shifted to the right/left, the algorithm will identify the corners as being shifted as well.
- The resolution of the image needs to be higher than 640x480 for the screens to detected correctly.
    - Reason:

    * The pixels are compared individually to find a difference between them. If this difference is big enough, the screen can be detected. With a resolution that is too low, the pixel data is not as accurate and thus the differences will be incorrect. This will result in the algorithm not/incorrectly recognizing certain screens.

4.5 Angle detection

- For the algorithm to find an angle, it is assumed that the pink quadrant can never be fully covered.

4.6 Triangulation

- The centers of two or more slaveScreens can not have the exact same coordinates. This means that in particular that a smaller screen may never lay perfectly centered on top of another larger screen.
  - Reason: The algorithm will work with two points with the same x -and y coordinate but because it looks like two different points for the algorithm it is able to create a line between them. If we then calculate the angle between a line starting/ending in this point we always get the angle of this line and the x axis. This can lead to the algorithm not detecting lines to be deleted. Also the resulting circumscribed circle will be wrong.

4.7 Image Cut

- No overlap between screens.
  - Reason: The image is cut to fill the screen as found by the screen detection. This exact cut is what will be send to the slave. The slave has no sense of where to align the image to, thus scales it to fit.

5 Conclusion and prospects

At the end we can conclude that while the project is far from flawless, it contains a good base that supports a decent amount of functionality. All the tasks have been completed and the capabilities of the numerous functions and algorithms that needed to be implemented are sufficient.

Connecting all the devices to the server and assigning the master and slave roles works smoothly, although additional functionality to switch between roles could increase usability. Detecting the screens when the connections are set up works as supposed in most cases. If the user addresses for external lightning and brightness of the screens reliability increases greatly. Figuring out the orientation of the displays was a big task, but eventually it was completed. In all the cases where the corners are found and not overlapped by other objects the angle of a screen is calculated properly.

With all this information, our program can then display an image, triangulation and animation. Again, these functions work as expected in normal circumstances. Some minor functionality is not includes. This contains: Content that requires a perspective transformation due to tilted screens is not adjusted, the animation does not adapt to screens with

rotation, The triangulation slightly defers from its expected properties in very uncommon cases.

Overall, given that the four corners of all the displays have been detected, our project is able to perform all the tasks in most circumstances. It is impossible to cover or even be aware of every edge case that can arise, yet we are satisfied with the way these are handled in our current iteration of the software.

A lot of challenges will certainly be presented in the next part of this project. These will not be easy to implement, as were the challenges of the current part, but this is to be expected. Considering the state at which we have arrived with our code, we can say that we have minimized the hurdles that will need to be taken. The base that we have built can be used to extend further upon, without having to drastically adjust current functionality.

6 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. p5 - MIT License: https://github.com/processing/p5.js
4. axios - MIT License: https://github.com/axios/axios
5. tsify - MIT License: https://github.com/TypeStrong/tsify
6. line-intersect - MIT License: https://github.com/psalaets/line-intersect
7. linear-equation-system - MIT License: https://www.npmjs.com/package/linear-equation-system
8. browserify - MIT License: https://github.com/browserify/browserify

References

[1] Karma JS. https://karma-runner.github.io/latest/index.html.

[2] MochaJS. https://mochajs.org/.

[3] calvinfo. Socket ntp javascript. https://github.com/calvinfo/socket-ntp?fbclid=IwAR0RTl2nAEwMgXDWNNPqjqfdj0HrSzmYVw6lvV7ecKa4CM_Q_qAPEbtefHs.

[4] 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.

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

[6] StackOverflow. Right way to get Web Server time and display it on Web Pages , 2018. https://stackoverflow.com/questions/20269657/right-way-to-get-web-server-time-and-display-it-on-web-pages/20270636?fbclid=IwAR2ZS0RwMgq9knkDomX2LWvgexUE70pU_OQG_gDmmQWBM0xqSOxkN-bOKq0#20270636.

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

[8] w3schools.com. How TO - JavaScript Countdown Timer . https://www.w3schools.com/howto/howto_js_countdown.asp?fbclid=IwAR0dWjna-UxChSn9ZywIZRE6KA-kid8NnXtAAgADuDhE2MfW6hVQlEcfci4.

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