P&O: Final report

August 18, 2020

— Report P&O CW 2019–2020 Final Report —

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

Team 03	
<Bram Vanbilsen>	308,5h
<Adam El M'Rabet>	76,5h
<Liam Volckerick>	119,5h
<Maarten Pyck>	240,5h
<Sebastiaan Wouters>	116,5h
<Pieter-Jan Van den Broecke>	116h

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

Contents

1 Introduction[Liam]

We developed an application that can detect screens and their orientation. Given an image containing all screens, we are able to link these screens and all information regarding that screen to a slave ID, which in turn is connected to our server.

Further on, the application is able to cast an image, an animation and a video using the correct perspective, over all connected and detected screens, relative to the master's position. Both the image casting and video casting functionalities have the ability to be executed in real-time depending on the position of the camera device, which belongs to the master device. The latter meaning that a form of tracking algorithms has been used in order to update the image/video casting in real-time at any given moment, if chosen by the user to do so.

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 error resistant as possible. This often meant choosing between multiple algorithms, each of which had its own flaws. Each individual task came with its own challenges, having to make us revisit all code that had been written prior.

In order to further increase the robustness and performance of our application, we tested some basic concepts of synchronization as well as a revisit of the entire screen detection algorithm by the usage of scientific testings on color detection. Short conclusions and main takeaways can be found in section 2.

Going from a simple chat application that can interact with the server, to making a full online application that can process screens connected to the server, manipulate their browsers in perfect synchronization and have real time tracking algorithms implemented in the casting functionality, is a great accomplishment. We hope that all information underneath clarifies what we have accomplished as well as showcase and explain how we tackled some of the problems that were given to us.

2 Scientific Testing and Iterative Development

A project as complex as this one has to start somewhere. It must be quite clear to the reader that no implementation shall be perfect from the get go. Instead we needed to iterate multiple times over some parts in order to find the best working method.

In this section we will describe the current state of -parts of- the project, compare it against the previous implementation and explain our reasoning for these changes by using some of the conclusions of our scientific experiments. While many changes have occurred -some more noticeable than others- we shall limit ourselves here to the two most important ones: screen detection (2.1) and synchronization (2.2).

2.1 Screen Detection[Pieter Jan]

Screen detection was in all likelihood the most tricky part of the project to get to work reliably and with acceptable performance on mobile devices, therefor big changes were made to improve its accuracy and performance.

2.1.1 Previous Iteration[Pieter Jan](All)

Input and idea The algorithm, used to identify individual screens, 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.

Image comparison 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 HSL colors are considered similar if:

- The H*ue* values for both pixels are no further apart than 50.
- The S*aturation* values for both pixels are not further apart than 40.
- The L*ightness* values for both pixels are not further apart than 40.

These ranges were chosen manually through trial and error to maximize the search range while still making a clear distinction between different colors possible.

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 2 pixels 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). However as cameras are never 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 we make is 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)
- $10\% \leq sr \leq 55\%$

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

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 now 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 corner points, 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 50 pixels from the endpoints of $diag_1$, we discard $diag_2$. Now the third largest connection $diag_3$ is found and the end points are again checked to $diag_1$ as above. This process repeats itself 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 $diag_i$.

2.1.2 Scientific Test[Pieter Jan](Maarten, Sebastiaan, Pieter Jan)

After we worked out the scientific tests pertaining color and screen recognition, we came to two conclusions that we used in improving screen detection in our project.

The first one is that red and pink are colors that are likely to be wrongly detected more often than others due to environmental influences. As we used a shade of pink during the first iteration we decided to change this.

The second conclusion follows in part from the first. Colors, no matter in which shade or form, whether they are represented in RGB, HSL or something else, are difficult to recognize accurately. While our scientific tests showed some decent results, we still had to conclude that there are better and more reliable ways to detect screens than working directly with colors. We instead opted for a more streamlined approach, which makes use of the difference in color between two images.

2.1.3 New Iteration[Pieter Jan](Bram, Maarten)

Our newest method of working starts by iterating over the slaves, ordering each of them in turn to display a cyan color, RGB code (0,150,150), followed by a dark red color, RGB code (150,0,0). The master shall snap a picture after each image that gets displayed. This gives us two different images to compare for each slave screen.

When the master has its two images, it will detect for any **big color differences**. In order to do this we will iterate over all pixels, comparing the color we find in the first image with the color in the second. To make this comparison we convert the RGB values to the LAB color space. In LAB, the 'A' channel shows us the values on the red to green axis, while the 'B' channel shows the values on the blue to yellow axis. The 'L' channel shows no color info at all. Here we can see how light or dark each color is. The LAB color space is designed to closely resemble visual changes with the same amount in its L-A-B channels [3]. Thus if two colors are visually very similar, they will also be very similar in LAB. If the difference, calculated with ΔE^*_{00} [15], exceeds a threshold (40 in our case) we will accept the pixel as being part of the screen we are looking for. As a result of using color changes, the camera must stay approximately stationary.

Next we will consider all found pixels. For each of these pixel its direct neighbors are checked and the pixel is considered noise if it does not have at least five neighbors. These noise pixels are **filtered** out.

Now we must **divide the found pixels** into areas. These areas will include pixels of which the colors were changed and are close enough to each other to be considered connected. To perform this task, we created a sweepline algorithm which will first sort all points by smallest x-value. We iterate over all the sorted pixels and place them into areas. A new area along the x-axis is created when the consecutive points differ in x-value by at least *sweepWidth* positions. We consider one single pixel to be a sufficient gap. Next, the points in these intermediate areas along the x-axis are sorted by y-value. We now partition the points in each of these areas with the *sweepWidth*, thus an area along the x-axis can be split into multiple areas along the y-axis. All points are now partitioned into areas taking into account the x- and y-axis.

At this point at least one area must be found in order to continue the screen detection. If this is not the case, we must discard this screen as a failed attempt. If multiple areas were found, we will continue with the area consisting of the largest number of pixels, as areas consisting of noise will be comparatively smaller than the screen.

The last part that remains is to find the **corners of the area**. We create an image with the same dimensions as the two pictures of the camera. On this image we add all

remaining pixels of the found area as white points on the black default background. Next we apply Gaussian blur [2], an algorithm to reduce noise and finally the Sobel edge filter [11, 13]. These filters combined will return the edge of the found area. We can now use this edge to calculate the four corners:

- The first corner, c_1, can be found by approximating the centroid of the screen by calculating the median of the edge points. The first corner is then the point farthest from this point.
- The second corner, c_2, is the point located farthest from the first corner.
- For the third corner we calculate the centroid of the first two corners. The corner we are looking for is the point located farthest from this centroid. We reject any corners for which the angle between the line c_1c_2 and the line between the centroid and the hypothetical c_3 is too small, namely less than 20°.
- The fourth corner is now the point located farthest from c_3.

These four points are now considered the corner points of the screen we were looking for.

2.2 Synchronization[Adam]

In order to be able to perform a task simultaneously on different but connected devices, a synchronization system is needed. This semester we have been analyzing two ways of setting up such a system. In this section we shortly describe these two methods, discuss the results of scientific tests and finally clarify which one we opted for and why.

2.2.1 Synchronizing local clocks

One method consists of synchronizing the local clock-times on every connected device. This would require the synchronization to be performed by sending a **start time** to the clients that is enough delayed so that even the slowest client [1] would receive the emit before it. In order to verify whether such a method is suited for our purposes we decided to investigate the accuracy of local clock-times on multiple devices and compare the differences between multiple operating systems.

The way we tested this approach is by calculating the average time it takes for each device to reach a specific ntp server and then calculate its own clock offset relative to that server. All tests were executed in a Node environment, not in the browser. This allowed us to use the UDP protocol which prioritizes speed in comparison to TCP (as explained in our previous report) which in turn allows us to use the NTP protocol [10, 12].

Our conclusion from these tests was that the operating system plays a big role in the clock differences. For example our results for Windows showed a much bigger (on average 500 ms bigger) clock offset than with other operating systems. Multiple factors could play a role in these clock drift errors like network conditions, accuracy of the computer's hardware clock or even the amount of CPU and network resources available for the operating system's time service [7]. It is also important to use a reliable NTP server as it may take less time to reach an NTP server, but the NTP server could be very busy. This may result in a

[1] By the slowest client we mean the "listening" client for which an *socket-io* emit sent from the server would take the longest time to reach. This can easily be determined by sending an emit back and forth from the server to all clients and recording the time needed and then dividing this time by two.

difference between the time when the NTP server records its time and when the server actually sends back the packet containing the time. This ‘lag‘ on the server might cause errors. Furthermore, graphs of our test results showed that clock offsets could fluctuate over time for a same device over a range of 35ms as it was the case for MacOS systems (see figure 1).

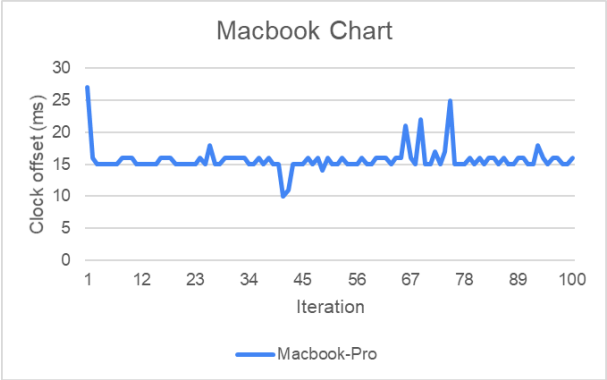


Figure 1: Clock offsets on MacOS systems

All these aspects pushed us towards not considering this option. One could argue that in our Cat Caster program we do not need such a perfect synchronization like in a payment transaction system. But even an offset of 500ms, for example, could lead to a tremendous fail when executing a task simultaneously over multiple devices. Because of that we decided to not synchronize the clocks of the clients and investigated an alternative method which will be described in the next section.

2.2.2 Synchronizing based on delays

Instead of calculating the offsets of the clocks one might calculate the offsets induced by the differences in the current times and the *socket-io* emit latency. Then, instead of sending a start-time the server would send a **delay** before performing the task (like "start the countdown in 10 seconds" instead of "start the countdown at XX:XX:XX"). Naturally these 10 seconds are relative to the server and will, at the arrival of the emit, not be interpreted as 10 seconds for the receiving clients. This makes the entire synchronization process completely independent of the differences in clocks. This was how our program was initially implemented.

We then investigated whether such a method provides better results than the method of synchronizing local clocks. To do so, we implemented an extension to our existing code (that already worked using this method) that records the offsets induced by emit latency every second for a duration of 5 minutes. Due to previous experience we expected the resulting curve to be a down-sloping curve starting with a very high offset due to the big load of initialization process on the single-threaded JavaScript call stack. According to our assumptions this curve would then go down until reaching a stable state close to the ”real” offset because the offset due to emits would be minimized.

The results of these test were stunning. After a first run we got something very unexpected (see figure 2 left). Instead of getting a down-sloping curve we got a fluctuating curve with results between an offset of 0ms and 60ms.

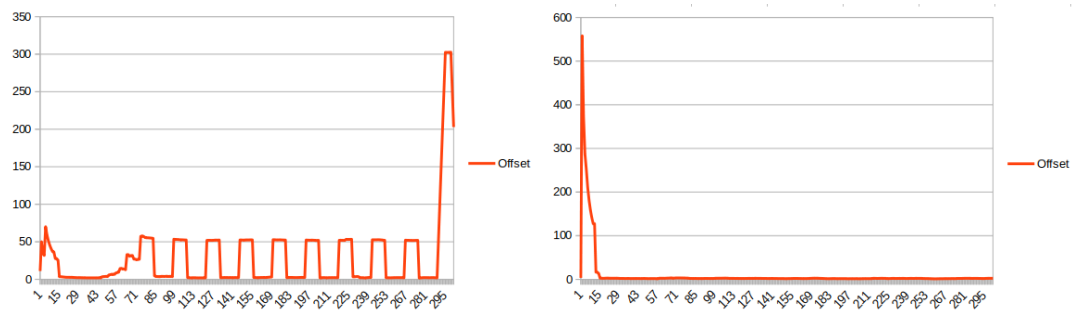


Figure 2: Tests of the delay-based synchronization method using a busy computer (left) and a computer at rest (right)

After we analyzed these results we assumed that the cause for this fluctuation was the fact that the laptop on which the test was run was busy doing some other work. After killing the significant processes that could slow down the communication process we got what we expected (see figure 2 right). After the 14th iteration the offset stabilized around 3ms.

One might argue that this method is not better than the previous one because of the possibility for fluctuation which makes it unpredictable. After all we cannot require the device that would use this synchronization method to be idle and not run other tasks (although we might freeze these tasks during the synchronization process). Still there exists a way to overcome this issue. Instead of calculating a final offset value per slave at creation, we could keep this synchronization ongoing during the whole lifetime of the slave. Then, when a synchronization task would be needed only the last calculated offsets will be taken into consideration. This was not possible for the previous method because this would require manually calculation and saving of the clock offsets of every device before connecting that device to the server (as it is not possible to use UDP and as a result NTP in the browser).

2.2.3 Performance of animations

Besides the reception of a correct start time for synchronizing tasks on multiple but connected devices, it is also important to make sure that the task keeps running in a synchronized way. This is of great importance for animations on web browsers. In the previous iteration we used *setInterval()* for animations. Later on we discovered that there is an alternative function to make JS animations: *requestAnimationFrame()*. In our previous report we describe how this alternative provides better performances due to multiple reasons [1]. We therefore changed our approach in the new iteration.

We then tested browser efficiency by making a simple animation of a ball moving diagonally over the screen and running this animation under a varying artificial workload imposed on the browser by using 'busy-loops' in our code. By doing so, we could investigate how the browser would slow down as a consequence of increasing the loop-iterations. We ran the tests over multiple devices using Chrome, Firefox and Safari. An important result from these tests was that Firefox is more sensible to load than Chrome on the same device (see figure 3). We then discovered a blog-article by Firefox claiming to conserve more RAM than other browsers which explains this behavior [6].

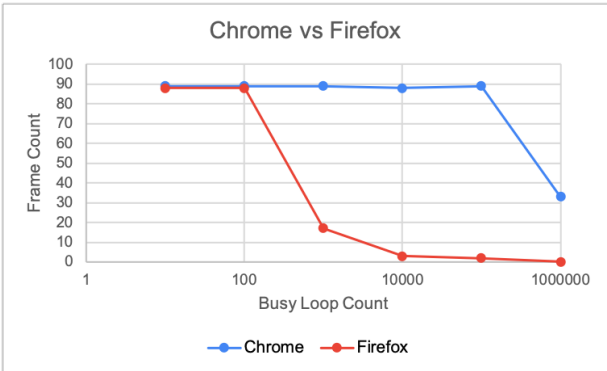


Figure 3: Figure showing the different effect of the busy loops on Firefox and Chrome

2.2.4 Conclusions and changes to the new iteration[Adam](Bram, Adam, Liam)

The synchronization method based on clock offsets did not provide satisfying results. Induced offset might be way too large for guaranteeing a correct working synchronization. The alternative method that used emit latency offsets instead, provided much better results when doing a frequent "re-synchronization" (by for example calculating the time-offsets with the server every second). The same device that caused 500 ms of clock offset using the clock-based method caused a maximum of 60 ms offset with this method, which is a significant improvement. If on top of that we use the average offset over the last few emits, this would drop the offset to a maximum of 30ms.

We therefore conclude that frequent re-synchronization through communication between client and server, by using a custom sntp protocol, is a better method which guarantees synchronization at every moment instead of synchronizing the clocks of every connected device with an ntp server. In this case we did not need to change our implementation because we already used this method in the previous iteration.

On the other hand, a key change that occurred to our implementation was the way animations were synchronized. In the past we made use of $setInterval()$, but due to $requestAnimationFrame()$ providing better performance we decided to use a framework called P5.js which implements $requestAnimationFrame()$ and wraps it in a useful API. This API furthermore allowed us to easily set the wanted frame rate and modify our animation per frame [8, 9]. Another thing we learned from these tests is that in order to use the full potential of browser animations we should avoid using Firefox and always opt for chrome or safari whenever possible.

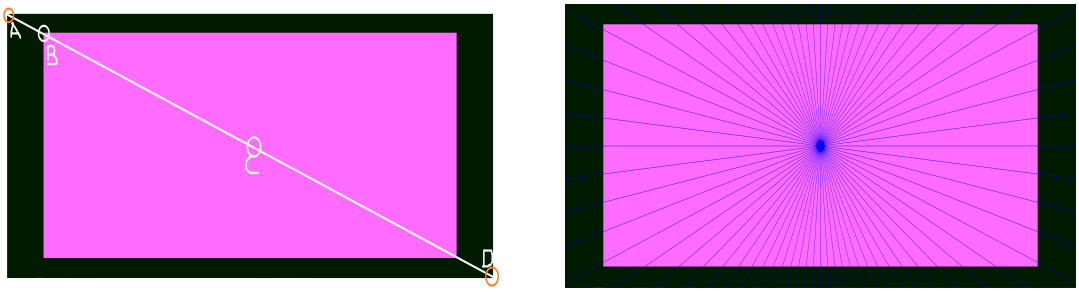
3 New Features

3.1 Screen Tracking[Bram](Bram, Maarten)

We decided to only track one screen and calculate the positions of the other screens using a projective transformation calculated from tracking that single screen.

3.1.1 Tracking a single screen

Appearance To track a single screen (S), we first make it stand out from the other screens by giving it a bright pink colored background with a dark green edge at the borders 6a. The green and pink areas preserve the aspect ratio of the screen.



(a) Appearance of tracking screen

(b) Tracking screen edge line detectors

Figure 4: Background used for detecting a screen and edges

Translating corners to the pink border The mentioned appearance is used because it visually shows a well defined edge between the pink and green area. This edge can be tracked and as a direct result, the corners can be tracked. But, because we are not tracking the actual corners found with screen detection, but rather the corners of the pink area, we first need to translate those actual corners inwards. This is done using the cross ratio [14, 5] which given four points A, B, C, D, (as shown on 6a) gives us a ratio which is preserved even under different perspectives:

$$crossRatio = \frac{\|AC\| \cdot \|BD\|}{\|BC\| \cdot \|AD\|}$$

Because of this cross ratio and our knowledge about the appearance of the tracking screen under no perspective (exactly as it is drawn in code), we can translate the corners found in screen detection to the pink corners.

Detecting edges Based on the previously found corners (either from 3.1.1 or a previous tracking iteration), we calculate the centroid C of the previous location of S. The goal now is to find the edge between the pink and green color. Assuming C is still in the pink area[2], we define multiple lines from C towards the edges of S. Each of these lines is under a different angle as shown in 6b. The algorithm steps along each of these lines until it finds an abrupt color change (the color changes are again calculated in the LAB color space). The points where this happens are stored and considered the edge. Assuming the pink on S is more or less equal through the view-port of the camera, the first abrupt color change from the center along the lines is at an intersection with the green border and thus the edge will be found correctly.

Marking points of interest Next, to find the corners, we define an area of interest for each corner previously found. This is simply done by looking how distant the found edge points are from each previous corner. If the distance between an edge point and a previous corner is smaller than a threshold, the edge point is considered to be a valid candidate for replacing that corner. The threshold is set in such a way that no edge point can be a candidate for more than a single corner. Thus at the end of this step, four lists of points are formed, one for each corner.

[2]This assumption thus prevents the user from moving too much between frames. The amount of movement allowed depends on multiple factors such as the size of S from the perspective of the camera and the speed of the algorithm and by extension the computational power of the device.

Narrowing to four corners Lastly, for each found list, the point that is most distant from C is considered to be the new corner. This results in four new points which represent the new positions of the corners.

3.1.2 Tracking a single screen on different kinds of hardware

Although this algorithm performs fast, mostly due to the fact that it only needs to analyze a small subset of pixels compared to screen detection, the exact time for analyzing and tracking between two frames depends on the hardware of the device running the algorithm. But by using $requestAnimationFrame$ as explained in 2.2, we can ensure that the call stack does not get filled up even on slower devices. Instead the algorithm waits until the device finishes a tracking iteration before it pushes a new one to the stack.

3.1.3 Tracking multiple screens

Once one screen is being tracked, we have a plane that is moved around in 3D space. The transformation of this plane between the original corners and the newly found corners is a projective transformation of which we can calculate the transformation matrix H [4]:

$$\begin{bmatrix} h_0 & h_1 & h_2 \\ h_3 & h_4 & h_5 \\ h_6 & h_7 & h_8 \end{bmatrix}$$

Each new corner position (u, v) of the other screens can then be found using its original corners (x, y) as follows:

$$H \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = k \cdot \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$$

This transformation ensures that the other screens on the same plane as S are moved accordingly along the plane of S. A flaw in this approach is that all points are considered to be on the same plane as S, which might result in inaccurate results if this is not the case.

3.1.4 Visualising a 3D scene[Maarten](Maarten)

Tracking one screen can also be used to render a simple 3D scene. For the scene, one of the simplest 3d objects is used, a cube. The cube is centred around the centre of the tracking screen $(x_s, y_s, 0)$ (the tracking screen, as displayed on the master, is a plane with a z coordinate of zero). To see this 3D scene we use an artificial camera. This artificial camera initially shows a top view of the cube, so this camera stands on coordinate (x_v, y_v, k), with $x_v = y_s$,$y_v = x_s$ and k the distance of this *camera* to the cube. This distance has nothing to do with the real distance of the master device to the tracking screen.

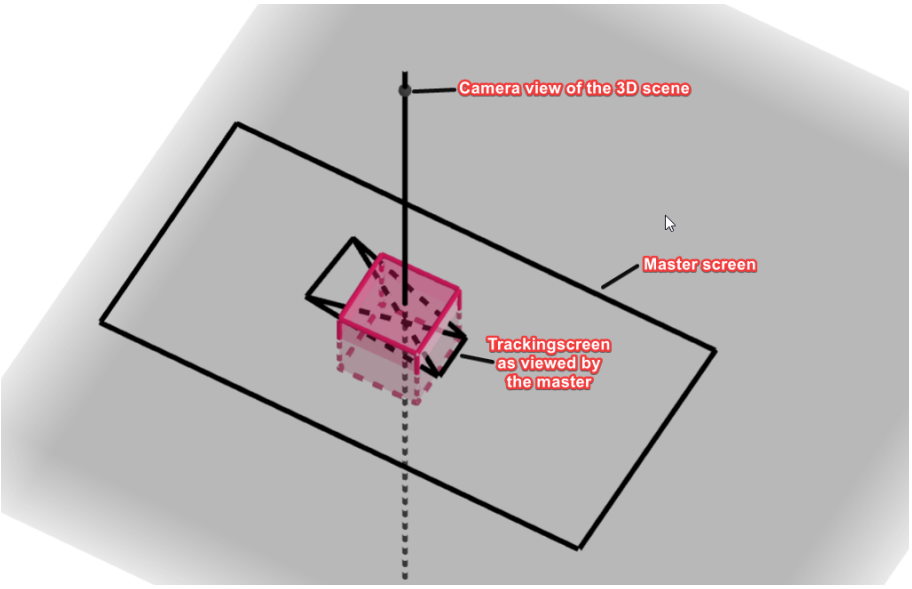


Figure 5: Initial position of the cube and the artificial camera

When the master device is moved and the tracking begins we assume that this distance k will be constant. This assumption is needed because the tracking does not give enough information to track the changes in all the three dimensions. When this assumption is made, the tracking gives enough information of changes in the x and y direction of the artificial camera. Using the transformation matrix H we can calculate the changes in the 2d plane:

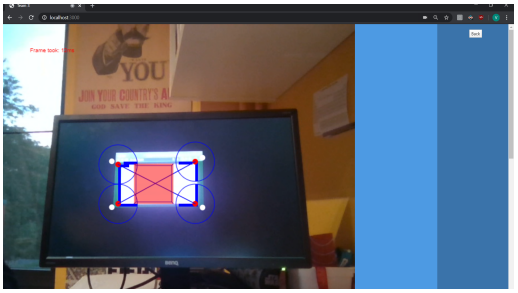
$$H \cdot \begin{bmatrix} x_v \\ y_v \\ 1 \end{bmatrix} = k \cdot \begin{bmatrix} x'_v \\ y'_v \\ 1 \end{bmatrix}$$

This gives us the new position of the camera, (x'_v, y'_v, k). After this, the cube must be projected on a plane which is the master screen. If the "camera" has coordinate (x_v, y_v, k) at the moment of projecting a point with coordinates (x, y, z) to the projected coordinates (X, Y) on the plane we get:

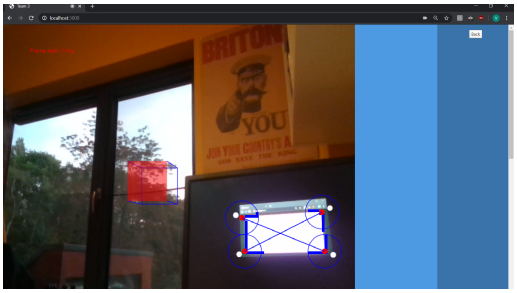
$$X = (x - x_v) * (z - k)/z + x_v$$

$$Y = (y - y_v) * (z - k)/z + y_v$$

After this we simply connect the needed projected points to get the projected cube. All the lines of the cube will be visible on the master. This makes it sometimes difficult to see how the cube looks, therefor we made the plane that is closest to the camera in the initial position red.



(a) Initial position



(b) After movement of master device

Figure 6: After moving with the master device

3.2 Extra: Video[Sebastiaan, Liam](Sebastiaan, Liam, Pieter Jan)

Another extra feature that we added to this iteration of the process is the possibility to play a synchronized video across all the connected slaves. Every slave is instructed to display the correct part of the video so that from the master perspective, it looks as if the video is being displayed on one big screen. This section will explain how this feature was implemented.

3.2.1 Playback of the video

The video is always initiated from the master. After all the slaves are detected, buttons to start, toggle and stop the video playback become available to the master. When the start button is pressed, the video element is moved to the foreground on the slaves. The video content was already loaded, as this gets done the moment a slave connects. This way, you can be sure the loading is finished and won't cause any problems. During playback, the video can be paused, resumed and stopped. Stopping the video will return all the slaves to their default state.

3.2.2 Synchronizing the video

In an early stage of the implementation we had added the video code which allowed to play asynchronous videos across the slaves. As an experiment we wanted to check how big the differences in playback rate were this way. It became clear that after only a couple of seconds, some slaves lagged behind around one hundred milliseconds. If this went on for some more time, the difference became very clear to the human eye. It was very obvious that real time synchronization was needed. We used the methods described in section 2.2 to handle the synchronization.

First of all, we were now able to start the video on all slaves at exactly the same time. The master makes this possible by including a delayed start Time in the emit to start the video. Each slave receives this emit, not essentially at the same time, but because of the synchronization process that has taken place between slaves and master, every slave knows how to perform this action in sync. As mentioned, after a couple of seconds some slaves start to show a visible delay even when starting the video at the same time. As a solution we implemented a continuous synchronization.

To do this, the master sends out an emit every 3 seconds with a delayed starting time that lies 2.5 seconds in the future. The slaves receive this emit and use the info available from the syncing process to send out their current video frame in sync. To explain it in more simple terms: each slave sends out to the master which frame the video has reached

at exactly the same time. The master now establishes a list of all these *timeStamps* linked to the corresponding IDs. The slave that has played the most frames is used as a reference. For every other slave, the difference with the reference slave is calculated. These differences are then sent out to each slave individually. The slave that has progressed the most will receive a difference of 0 milliseconds and thus will not have to update. All the other slaves will receive non-zero offsets, to keep up with the fastest moving slave they will skip ahead the amount of frames calculated in the offset.

These differences sent out by the master will not be delivered at the exact same time on all slaves, as we only used synchronization for slaves to get their current frame all at the same time. After that, the communication between the master and its clients is no longer in sync. This should not be a problem because it's a difference that's being sent out. If this arrives a tenth of a second too late you can be sure that the difference between two playing videos has not yet increased significantly.

3.2.3 Performance of the video

As mentioned above, all the slaves load the videos as soon as they connect to the server. This is better then only loading it when the video is requested since this could cause lag. When the master is switched, the new slave will also automatically load the video. We are currently skipping to the frame of the furthest slave. This skipping happens every three seconds. If a slave is behind about 100 milliseconds a skip will be quite noticeable. To prevent this, a system could be developed to increase the playback rate accordingly instead of skipping abruptly so that the slower slaves could catch up more smoothly. In our current iteration, when the master pauses the video it does not stop the syncing process. As none of the slaves are progressing this means that they all should be at the same frame after a maximum of three seconds. After this, the syncing process could be stopped and initiated again when the video is resumed.

In general we can conclude that the starting, toggling and stopping of the video on all slaves works perfectly. To assure playback happens at the same speed we implemented a synchronization mechanism that's ran every three seconds. This works as intended, however in occasional cases it can be slightly annoying for the viewer that a certain slave is visibly skipping frames. Some non essential syncing is being performed when video playback is paused as well. Everything else is integrated into the project very well and doesn't introduce unwanted problems.

4 Conclusion[Liam, Bram]

Lastly, we would like to point out that even though our final application works quite well in ideal circumstances, it has still plenty of room left for improvements. As of now most of the functionality has been realized by writing the core algorithms ourselves. The main takeaways for color recognition lie in the fact that none of the color spaces (e.g. HSV, HSL and RGB) lead to a robust color detection. None of the prior mentioned color spaces were robust enough to recognize specific colors in a range of different lighting environments. If one would consider making a similar application, we would suggest detecting color differences instead of trying to detect specific colors. The delta function, ΔE^*_{00} [15], defined in the LAB color space is a great tool for this.
We believe that the true solution to computer vision lies in an application which can reason

about data, like humans can. The latter leads to an A.I. implementation that can learn to recognize objects within images as well as provide the simple data needed to further extend our functionalities on this base created by the A.I..

5 Remarks

- We would like to point out that mostly all of the algorithmic parts in this project were written by ourselves. This does not just include algorithms that we came up with, but also widely used algorithms such as Sobel edge detection and circle detection in hough space (which, due to performance reasons, did not make it into the final project). The usage of external packages were only used in the form of saving time on solutions to problems which were out of scope of the assignment, like algorithms for solving matrices in the case of perspective casting.

6 Assumptions

- For screen tracking, the inner area of the tracking screen should be more or less solid pink from the view of the camera. If this is not the case, a big color change might be detected at a position which does not lie on the green border.
- For screen detection, the color of the screen should be more or less solid. If this is not the case, due to for example a bright light reflecting on the screen or a really dark spot on the screen, the color change between the two colors might not be significant enough to be detected.
- For corner detection, the found screen area should not have too many outliers. If the outliers are too large, the angle condition does not hold. The same applies for detecting corners of screens with a lot of perspective.
- For screen tracking, the camera should not move too quickly. As explained, when a new tracking iteration is started, the center of the previous tracking result should still be in the pink area. Similarly, when a new tracking iteration is started, blue circles are displayed on the camera to indicate the search range for each new corner, if the user moves too fast, these ranges might become too small and might not include the new position of the corner.

7 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] Better Performance With requestAnimationFrame. https://dev.opera.com/articles/better-performance-with-requestanimationframe/.

[2] Canny edge detector. https://en.wikipedia.org/wiki/Canny_edge_detector. Accessed: 2020-03-14.

[3] Cielab color space. https://en.wikipedia.org/wiki/CIELAB_color_space. Accessed: 2020-03-05.

[4] Computing css matrix3d transforms. https://franklinta.com/2014/09/08/computing-css-matrix3d-transforms/. Accessed: 2020-05-05.

[5] Cross-ratio. https://en.wikipedia.org/wiki/Cross-ratio. Accessed: 2020-05-05.

[6] Firefox Uses Less Memory Than Chrome, Edge and Safari. https://blog.mozilla.org/firefox/firefox-uses-less-memory-chrome-edge-safari/.

[7] How the Windows Time Service Works. https://docs.microsoft.com/en-us/windows-server/networking/windows-time-service/how-the-windows-time-service-works.

[8] P5.js. https://p5js.org.

[9] p5.js/main.js at 5f37f6784d481219ffb20a1594d28ec86fc7152a · processing/p5.js. https://github.com/processing/p5.js/blob/5f37f6784d481219ffb20a1594d28ec86fc7152a/src/core/main.js#L399.

[10] UDP/Datagram Sockets | Node.js v13.9.0 Documentation. https://nodejs.org/api/dgram.html.

[11] Ashish. Understanding edge detection (sobel operator). https://medium.com/datadriveninvestor/understanding-edge-detection-sobel-operator-2aada303b900. Accessed: 2020-03-13.

[12] Callan Bryant Clément Bourgeois. ntp-client. https://www.npmjs.com/package/ntp-client.

[13] Sean Riley Dr Mike Pound. Finding the edges (sobel operator) - computerphile. https://www.youtube.com/watch?v=uihBwtPIBxM&t=311s. Accessed: 2020-03-13.

[14] Brady Haran Federico Ardila. The cross ratio - numberphile. https://www.youtube.com/watch?v=ffvojZONF_A. Accessed: 2020-05-05.

[15] Zachary Schuessler. Delta e 101. http://zschuessler.github.io/DeltaE/learn/#toc-defining-delta-e. Accessed: 2020-03-05.

8 Appendix

8.1 Reverse Demo Instructions

1. Connect all devices to the server by browsing to the following link: https://penocw03.student.cs.kuleuven.be.
   The device that's connected first will get assigned the master role.
2. Once all devices are connected, the start button on the master can be pressed.
3. A viewfinder will appear, make sure that all the slave displays are fully visible before pressing the red button to start the screen detection. It's important that the master is stationary while executing the detection algorithm.
4. If something goes wrong during the detection, all devices can be refreshed to start over.
5. When the screen detection has finished, a set of intuitive buttons will show up, allowing you to test all of our implemented functionalities.
6. To test Real-time tracking in our application: We suggest firstly either showing an image by displaying master image or the horse. Display video is also possible. Once this step has been established we can press on the tracking button, which will activate real-time tracking and adapt either the image or the video to the new position.
7. To toggle our 3D-scene, simply press the 3D button while tracking.
8. To exit the real-time tracking, simply press the 'Back' button in the top right corner of the master's screen.