אוניברסיטת בן-גוריון בנגב
Ben-Gurion University of the Negev

# Augmented Reality

Final Project-Digital Image Processing 361.1.4751

Daniel Duenias-307983130

Ophir Gruteke-302363049

Rom Schilman-313307753

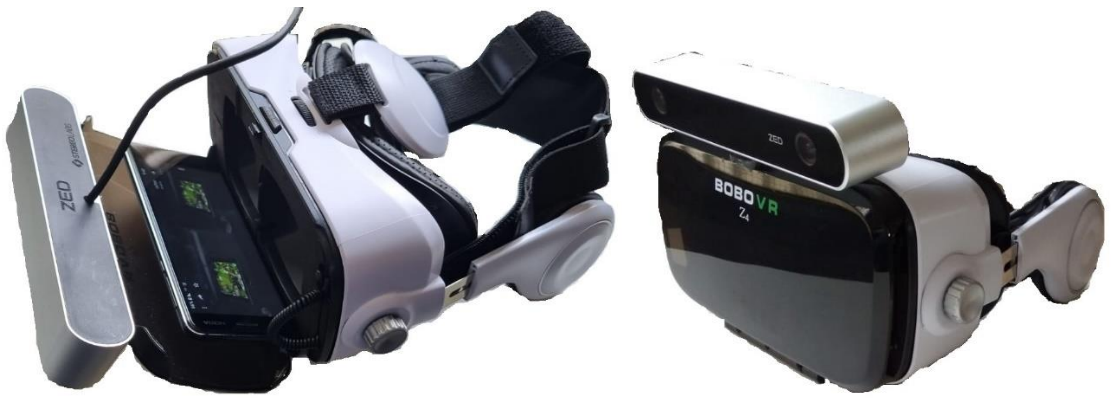
Yossi Bodek-318436268

# Table of contents

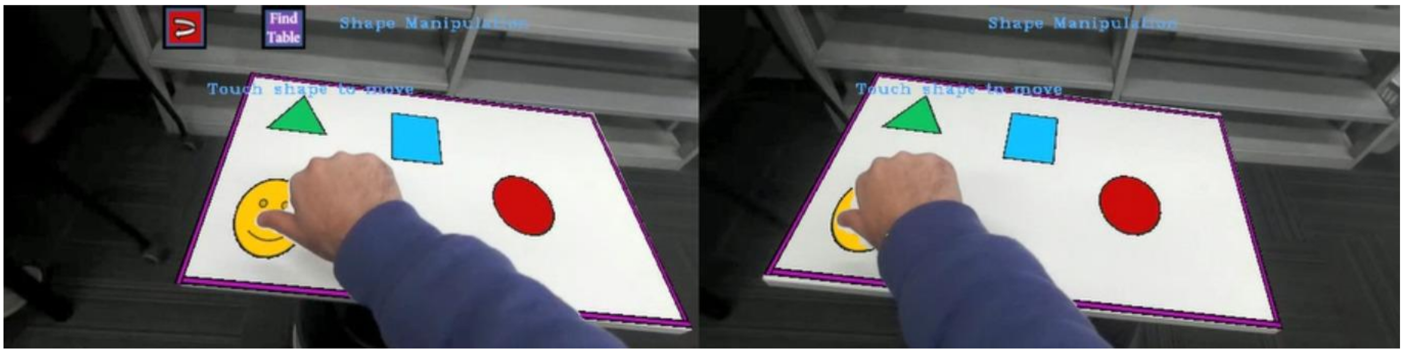
## Abstract

YouTube link for a small demonstration of the project - https://www.youtube.com/watch?v=I7b0lHkV0Ug

Our project is an Augmented Reality (AR) application based on camera with two lenses and virtual reality phone glasses.

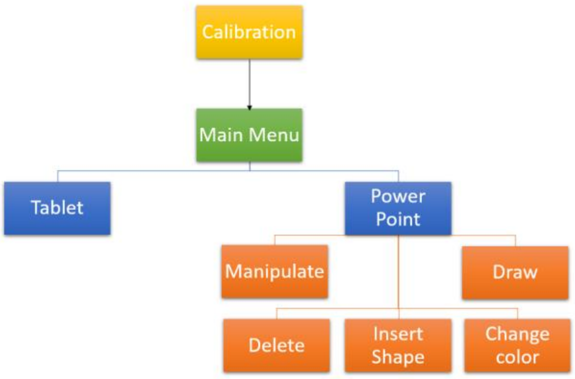


The application detects a table in the user's frame and implants virtual objects on the table according to the user's demand using dynamic camera. The user can use his hands gestures to maneuver between the different menus and states of the application and touch the table, using it as a touch screen, to manipulate the virtual objects on the table.

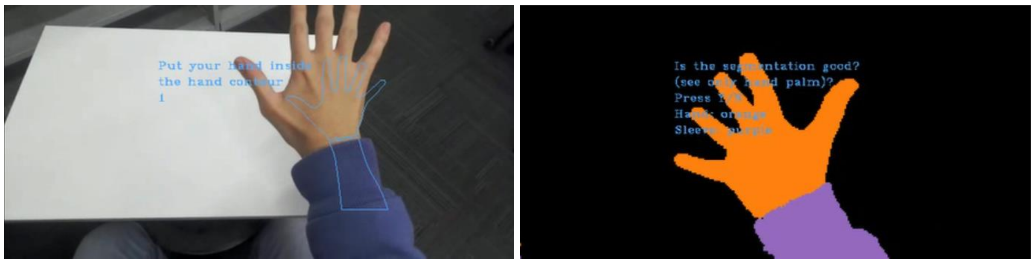


## Project Description- The Application

The application interface is the implementation of the following state machine:



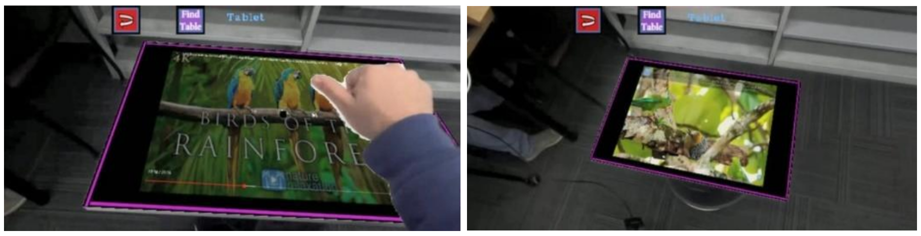
In the calibration the user locates his hand and sleeve in the designated contour, as shown in the image below. The calibration's goal is to train a Gaussian Mixture Model for the segmentation of the hand and the sleeve of the specific user in the specific lighting (detailed explanations in the hands segmentation section)


Put your hand inside the hand contour
1
Is the segmentation good?
Press Y/N

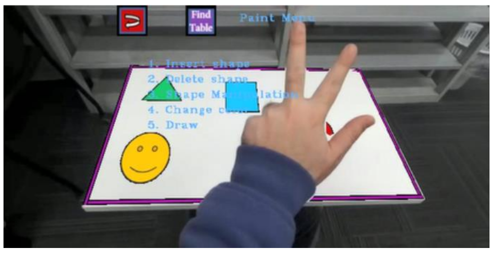
After the calibration the user can move through the application's states. Moving through states is done by rising different number of fingers or by placing the hand on the 'back button' (all is shown to user in the virtual menu). In every state, the user can use the 'find table' Find Table button to renew tracking of a table in the frame.

Main menu – choosing between 'Tablet' and 'Paint' (Power Point) applications.

Tablet – a tablet's screen will appear on the table; the user can use the table as a tablet.



Paint Menu – the menu to the Paint application (Power Point - PP). The application is based on PP software and the table is used as a blank PP page.



Insert Shape – the user can choose a shape to insert using several hands gestures (described in shape classifier section).

Manipulate – the user can touch the shapes that are on the table to move, scale and rotate them.

Delete – each shape the user touches will be deleted.

Draw – the user can use his hand to touch the table and draw on it.

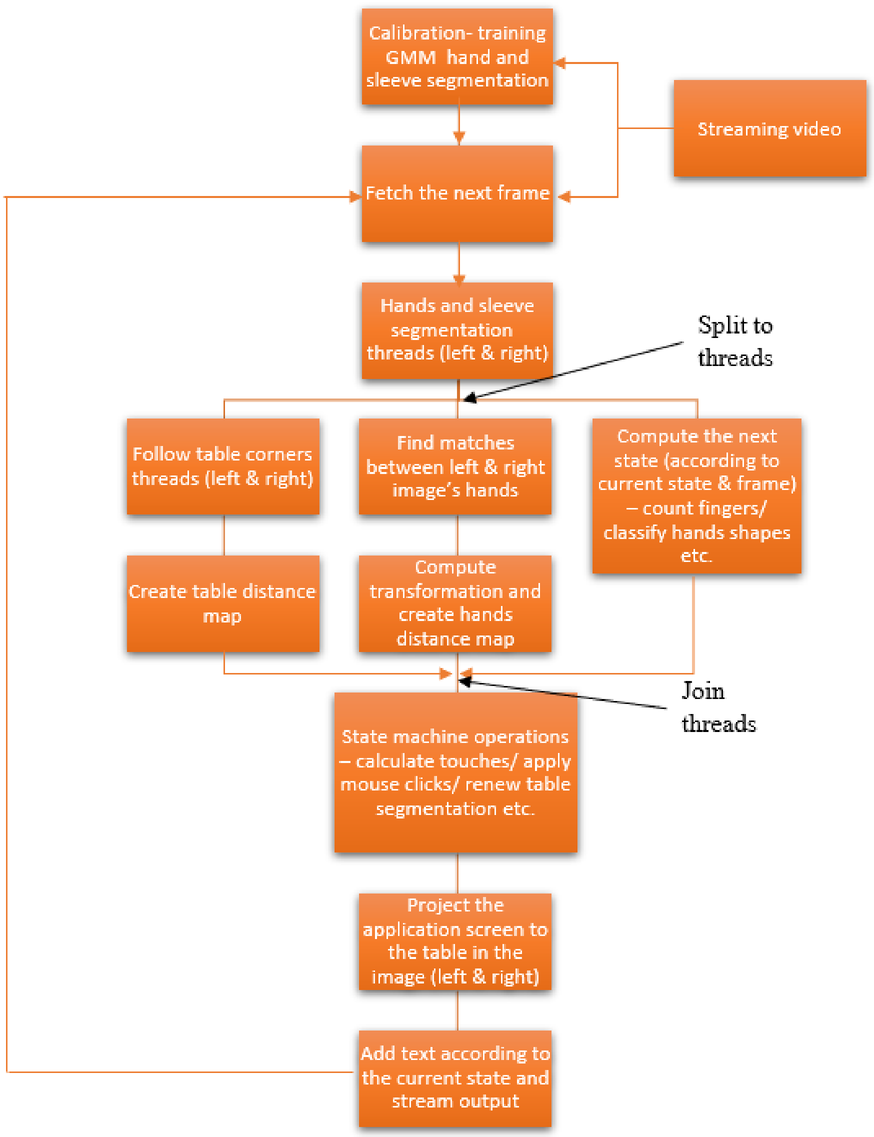
Change Color – the user can change the color of the shapes to insert and the drawing pen with raising different number of fingers.

## System presentation

### General Program Flow

Both the calibration and the main program starts by fetching a frame from the streamer (explained in video streaming section) and then process it. During the processing, there are several independent tasks that can be done simultaneously, therefore, we used threads to get more CPU utility for faster run-time. The main program starts by segmenting the hands and sleeves that are in the frame because the other tasks are based on that segmentation. Then, simultaneously creating the table's distances map, the hand's distances map and computing the application's next state (as explained in the former section). After that and according to the application's state, the relevant operations are done. Then the application's screen is projected to the table plane and the writing, according to the application's state, is added to the frame.

A general overview of the processing flow:



Calibration- training GMM hand and sleeve segmentation
Streaming video
Fetch the next frame
Hands and sleeve segmentation threads (left & right)
Split to threads
Follow table corners threads (left & right)
Find matches between left & right image's hands
Compute the next state (according to current state & frame) – count fingers/ classify hands shapes etc.
Create table distance map
Compute transformation and create hands distance map
Join threads
State machine operations – calculate touches/ apply mouse clicks/ renew table segmentation etc.
Project the application screen to the table in the image (left & right)
Add text according to the current state and stream output

## Hands and sleeves color clustering

[1]- [7]

### General information

This hand segmentation feature is one of the most essential features of the system. Many other features rely on good segmentation of only the hands with the background.
At the project, we tested several ways to segment the hands. At the beginning, we tried hand segmentation based upon Optical flow under the assumption that only the hand moves. The results were fine and little bit noisy, but with the dynamic camera the noise increased severely. Therefor we decided that color-based segmentation was better suitable for us, GMM (Gaussian Mixture Model) was the most accurate considering its run time.

The algorithm, receives a regular three-channel image and output GMM model and a list of labels that belong to the hands and the sleeves.

### Implementation

This feature implementation is part of the calibration at the beginning of the AR application. First, we draw hand and sleeve contours on the output image, so the user puts his hand inside the contours. We give the user countdown on the screen to know how long he needs to keep his hand inside the contours. The countdown begins with the substruction background image (the first image in the calibration flow) and the background and the hand. It required a match of a least 25% of the hand inside the contour.

After capturing the image with a good match between the hand and the contour, we convert the image to LAB color space. We needed a good separation between the hand and the background for good hand segmentation. The main problem could be shadows that can double the hand and create an odd shape. There are two main color spaces HSV and LAB. Those spaces have a unique channel for the lightness, that by removing it, we can overcome such lightning problems. We choose LAB for his ability to distinguish better between colors. It does not have the issue of the Helmholtz–Kohlrausch effect (when the intense saturation of a spectral hue is perceived as part of the color's luminance). In addition, HSV does not match their perceptual analogs to the original image.

The central part of the flow is the Gaussian Mixture Model. After the conversion to LAB, we removed the L channel to overcome the problems that we mentioned before and then trained the GMM model on that image. The GMM EM is a soft clustering algorithm when all the data points are assigned to all clusters with a certain weight (each point is likely to be in a specific cluster). Its inputs in our case is a set of 2d points (A and B channels) and the output is number of gaussians described by their mean, std and the probability of occurrence. The histogram of an image contains a mixture of gaussian distribution that describes each part of the image. Therefore, this algorithm is good for fuzzy image segmentation by separating the Gaussians to N components we choose. The algorithm is an iterative Estimation and Maximization until there is a convergence.

$$N(\mu,\sigma) = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{1}{2\sigma}(x-\mu)^2}$$

$$argmax_{\mu,\sigma} p(Data|\mu,\sigma)$$

The output of the GMM model is a labeled image where each label is the gaussian that the pixel has the best probability to be. In this part, we count the appearance of each label inside the hand and sleeve contours. We choose the most labeled in each contour and assign that label to hand, sleeve or background. The output is two lists of labels related to the hand and the sleeve. We sort good labels to one and bad labels to zero for each segmentation. In the end, we get a masked image with only hands or only sleeves.

### Assumptions and limitations

First and most important, we assumed a uniform light condition because the feature is based on colors. Also, the color of light should be natural and not colored to maintain good diversity and resolution of colors.

Second, in the beginning, we did only a two-component GMM model, and we assumed that the hand was the main object in the image. But then we came across a problem when the hand wasn't the main object, and then the GMM model segment also objects in the background as hands as they had similar colors. Therefore, we had to raise the number of Gaussian mixtures to four. To distinguish between the hand and the sleeve, the user needs to match them to a contour presented on the image. Here we assume that the hands, the sleeves and the table have three different (and far enough) colors – the background can be the same as the sleeve or as the table.

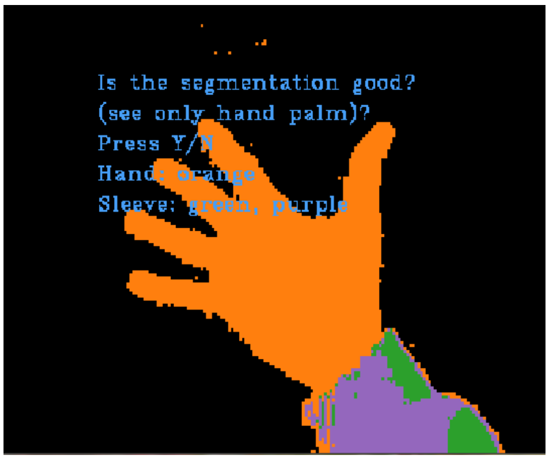
At a later stage, when we started to connect all the components of the AR system, we understood that when following the table corners dynamically, we had a problem with hiding. We solved it (there is an explanation next in the paper) by segmenting the sleeve. Therefore, another limitation is that we need a sleeve to operate the system. The sleeve's color needs to be far enough (in AB color space) from the hand color so that the GMM will distinguish between them well enough.

## Real-time adaptation

To maintain good segmentation, we had to train the GMM model with the best resolution image to get the best diversity of Gaussian mixtures. The best resolution is a trade-off to computational run time. We had to change the H.264 stream protocol parameters to get a good enough image with correction code to get the best quality from the camera to the PC. Another parameter is the computation time of the GMM; more components mean more computation time. We trained with full resolution at the calibration process, but later in the process, we had to resize the image to a smaller size and give up upon resolution for a run time between two frames.

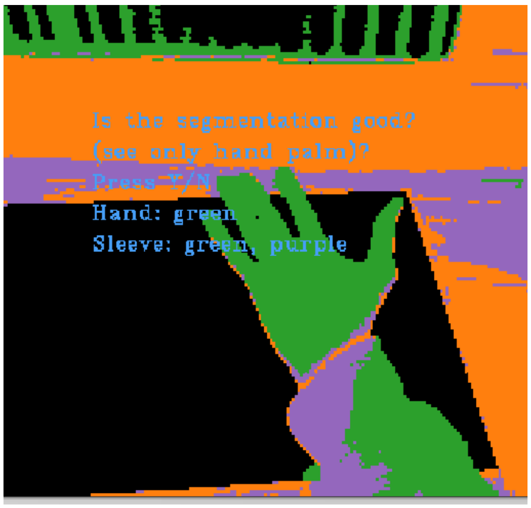
## Results

The result of the hand segmentation was pretty good but sensitive to light conditions.



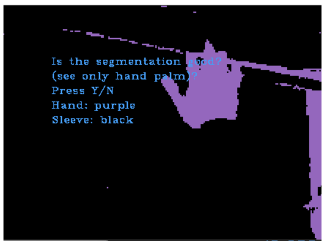
```
Is the segmentation good?
(see only hand palm)?
Press Y/N
Hand: orange
Sleeve: green, purple
```

Some GMM models segment both hand and sleeve contour with the same color, apparently because of darker shades of shadows on the hand and the sleeve. We had to choose if to take only the inner segmentation but lose the resolution of the segmentation or recalibrate the GMM model.

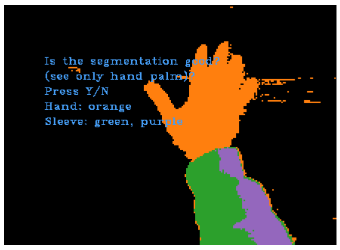


```
Is the segmentation good?
(see only hand palm)?
Press Y/N
Hand: green
Sleeve: green, purple
```

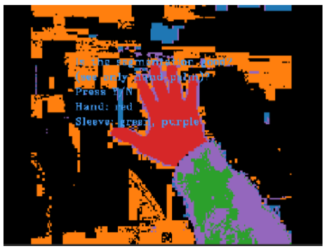
We had to choose the best number of components for the GMM while we had the trade-off resolution and computation time.



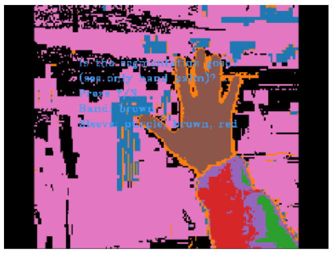
```
Is the segmentation good?
(see only hand palm)?
Press Y/N
Hand: purple
Sleeve: black
```



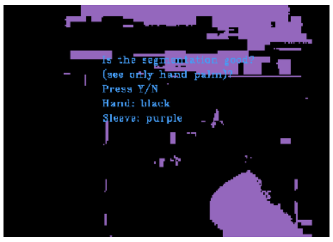
```
Is the segmentation good?
(see only hand palm)?
Press Y/N
Hand: orange
Sleeve: green, purple
```

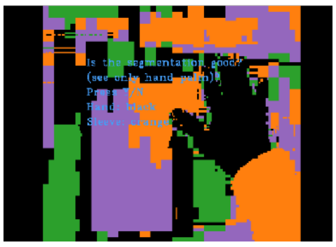


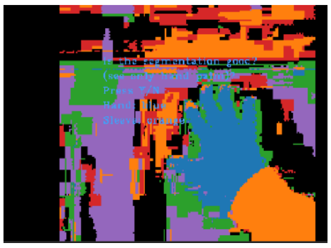
```
Is the segmentation good?
(see only hand palm)?
Press Y/N
Hand: red
Sleeve: green, purple
```

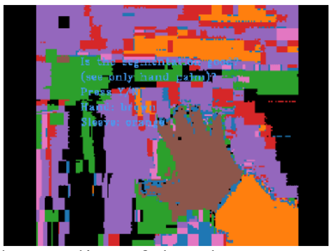


```
Is the segmentation good?
(see only hand palm)?
Press Y/N
Hand: brown
Sleeve: purple, brown, red
```


Is the segmentation good?
(see only hand palm)?
Press Y/N
Hand: black
Sleeve: purple


Is the segmentation good?
(see only hand palm)?
Press Y/N
Hand: black
Sleeve: orange


Is the segmentation good?
(see only hand palm)?
Press Y/N
Hand: blue
Sleeve: orange

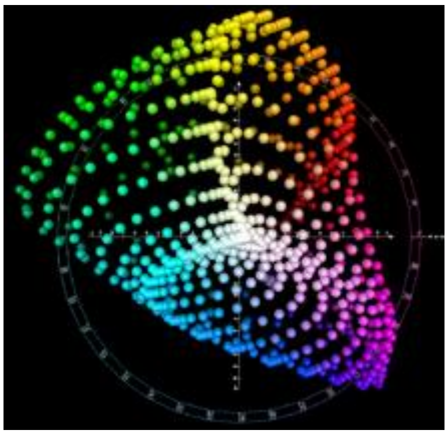

Is the segmentation good?
(see only hand palm)?
Press Y/N
Hand: brown
Sleeve: orange

The main difference between the first row (LAB color space) and the second row (HSV) is the quality of the clusters. The LAB color space has better correlation with the human color perception (distances between color in our perception). Also, the Hue channel of the HSV is not continues and it may affect the output.

Two components in the most robust with noise on the hand but not possible because we needed at least three components.

A high number of components will cause high noise in the image and won’t distinguish between objects in the image.

We tried several sleeves for the calibration, and in the end, we used a blue sleeve as it gave us the best distinguish between the hand and the sleeve. We can see from the LAB color space top view that the shades of blue are most far from the body color.



## Hands and Sleeve Segmentation Mask

### General Information

The clustering results in a noisy image. Therefore, we had to seclude only the hand from the noisy image using morphological operations.

### Implementation

Once we had the segmented image, we wanted to create an image that contains only the hand, to achieve that first we preformed morphological operations on the segmented image to reduce the noise.

The morphological operations are erosion and dilation, both operations used an 5x5 elliptic kernel because it simulates the shape of the hand and fingers well, the input of the operations is a binary image and the kernel, and the output is a binary image. In erosion the kernel slides through the image. A pixel in the image (either 1 or 0) will be considered 1 only if all the pixels under the kernel is 1, otherwise it is eroded (made to zero). Dilation It is just opposite of erosion. Here, a pixel element is '1' if at least one pixel under the kernel is '1'.

Second, we applied gaussian blur also to reduce the noise. Third, we applied the function 'findContours' by OpenCV that relays on [8]. This algorithm gets a binary image as an input, scans the image and for every pixel check if it part of a contour or not based on the number of neighbor pixels which is '1'. The output is list of points that contract contours.

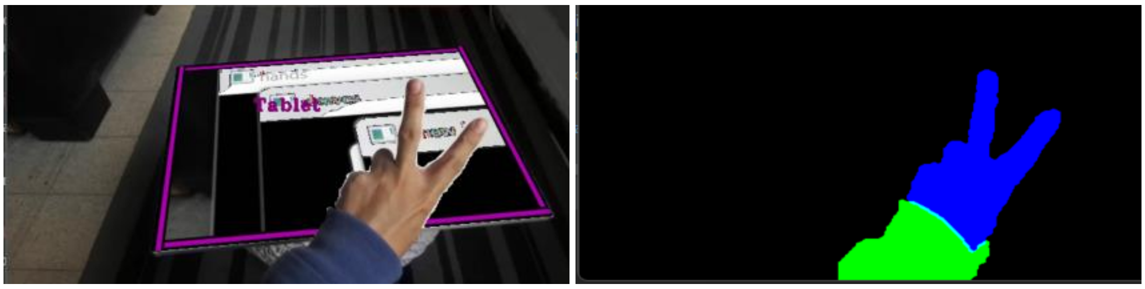
Fourth, we found the largest contour based on its length, we drew it on a zero image and fill the internal area with '1'.

The output of the prosses is a clean binary image that contains only the hand.

### Assumptions and Limitations

In this part we assumed that the hand has the largest contour in the image, and it is reasonable because the main detail in our images are the hands.

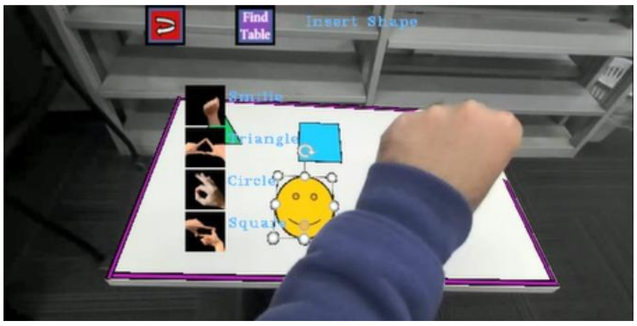
### Results



## Hands Shapes classifier

### General Information

One feature of our application is inserting a shape to the table. The shape is chosen by the user, from a set of 4 shapes, by preforming a specific hands gesture. To apply this action, we needed to make a machine learning model that maps hand gestures to shape labels. The machine learning model we applied is called "Bag-Of-Visual-Words" (BOVW). The general idea BOVW is to represent an image as a set of features that are made from key-points and descriptors. We use the key-points and descriptors to construct vocabularies and represent each image as a frequency histogram of its features. From the frequency histogram, later, we can find another similar image or predict the label of the image [9].



### Implementation

First, we collected images that contain the hands gestures we chose, we than applied different augmentations like zooming, rotating, shifting and flips to enrich our data set. Then we applied the GMM on the images to get only the hand to exclude background noises for better results.

Second, we turned all the images to gray scale and extracted features using SIFT, we tried several features extractors such as ORB and SURF and found that SIFT finds the largest number of good features (features that are unique and repeats in all the gestures) and it is fast enough for real time implementation. The input for SIFT feature extractor if a grayscale image, and the output is the key-points and descriptors of the given image. SIFT algorithm uses Difference of Gaussians which approximates Laplacian of Gaussian to find local extrema, it is done for different octaves of the image in Gaussian Pyramid. Next, the algorithm uses Taylor series expansion of scale space to get more accurate location of extrema, and if the intensity at this extremum is less than a threshold value it is rejected. Now an orientation is assigned to each key-point to achieve invariance to image rotation. A neighborhood is taken around the key point location depending on the scale, and the gradient magnitude and direction is calculated in that region. An orientation histogram with 36 bins covering 360 degrees is created. The highest peak in the histogram is taken and any peak above 80% of it is also considered to calculate the orientation. Now key-point descriptor is created. A 16x16 neighborhood around the key-point is taken. It is divided into 16 sub-blocks of 4x4 size. For each sub-block, 8 bin orientation histogram is created. So, a total of 128 bin values are available [10] [11].

Third, we stack all the images features in matrix that every row represents a feature in an image.

Then we clustered the features using K-means. we used K-means because we wanted to use classic machine learning algorithms that gives good results. We checked over the internet and found that this algorithm is widely used for image classification of small number of classes. K-means get as input set of points to be clustered and the number of clusters and gives as output the means points. First, k initial "means" are randomly generated within the data domain. Second, k clusters are created by associating every observation with the nearest mean. Third, the centroid of each of the k clusters becomes the new mean. Finally, Steps 2 and 3 are repeated until convergence. then we build the feature histogram, we check for each feature to each cluster it belongs and count the number of clusters that belong to every image. After swiping through various K values, we found that K=230 satisfies the best accuracy over the whole algorithm.

Finally, we preformed SVM on the histogram to divided it to 4 classes. We chose SMV for the same reasons we chose the K-means algorithm. The input of the algorithm is set of points, and the output is a hyperplane that divide the plane to 4 classes. We used the linear SVM that find a line that divide the data set into 2 in a way that the max the distance between the line and nearest points in each group. To create 4 classes, we used the One-vs-All method which means that we apply SVM 4 times for each class and all the other classes considered as one class, and to get prediction we search all the 4 SVMs and find the best.

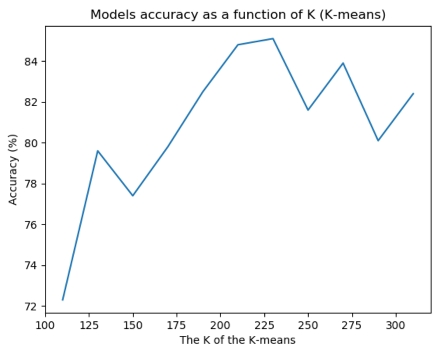
We created the model and saved it. On our application we loaded the model and created a function that get an image that contains only the hands (using the hand mask the pixelwise multiplication) and return the prediction. The function preforms the same steps as the learning excluding the augmentations and get the prediction from the SVM model.

## Assumptions and Limitations

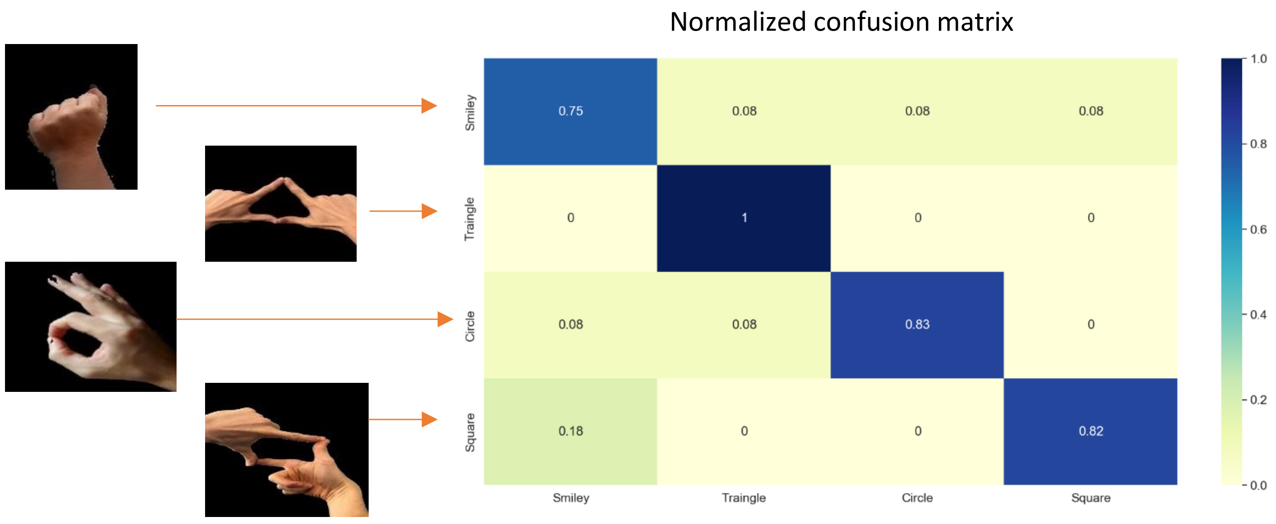
In this part we assumed that the hands segmentation is accurate, and that the user will copy the exact figure that he sees in the menu. One more assumption is that each shape has a features combination that defines it and separates it well enough from the other shapes.

## Results

Here we present the graph that helped us choosing the optimal number of classes for the K-means algorithm:



Although, this part wasn't the main part of the project, we managed to get good results as shown in the figure below:



The accuracy that we got is: 85.1% on the test data set.

To overcome the inaccuracy, we used predictions from 20 consecutive frames and only if over 75% predictions were the same, a prediction was considered as valid.

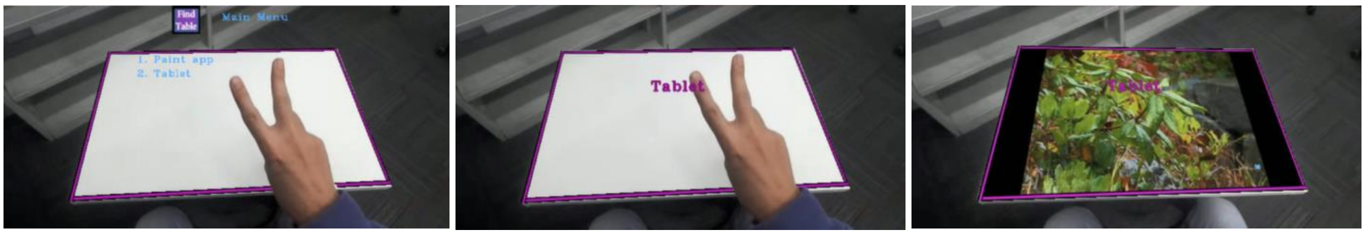
## Realtime Adaptations

To achieve Realtime performance we trained the model on small images (scale down to around 150x150 but with the original proportions) so the prediction process will by fast. For small images the SIFT features extraction is fast enough and extracts enough features (but not to many) so all the steps mentioned above for prediction are fast.

## Finger Count

### General information

In our application we change states by raising fingers and counting the number of fingers raised. The input of this function is a binary image that represents the hand, and the output is a number representing the number of fingers raised.



Switching to 'Tablet' raising two fingers (as in the menu on the left image)

### Implementation

The first step in the function is to find the largest contour the same way we did on the hand mask.

The next step is to find the convex hull of the contour found in previous step; convex hull of a shape is the smallest convex polygon that contains all the points of the shape. We used the function 'cv2.convexhull' on our application, the input is a set of point represents the contour, and the output is also a set of points represents the convex hull. The function is based on the algorithm [12], which take the bottom most point of the input set and sort all the other points by the angle created with the bottom point. Then the algorithm iterates in sorted order, placing each point on a stack, but only if it makes a counterclockwise turn relative to the previous 2 points on the stack, pop the previous point off the stack if making a clockwise turn and finally returns the stack. The runtime of this algorithm is O(NlogN).

Once having the contour and the convex hull we computed the areas created by the polygons by using the function 'cv2.contourArea' which uses Green's formula to compute the area.

Now we calculated the convexity defects between the contour and the convex hull, any deviation of the contour from its convex hull is known as the convexity defect. We used the function 'cv2.convexityDefects' which takes as input the contour and its corresponding hull indices and returns an array containing the convexity defects as output, an array where each row contains 4 values [start point, endpoint, farthest point, approximate distance to the farthest point]. The implementation is:

- for each pair (H[i], H[i+1]) of adjacent hull points
    - for each contour point (C[n]) that lies between [H[i], H[i+1])
        - Find the point with the maximum distance from the line joining (H[i], H[i+1])
    - store the index of (H[i], H[i+1], max distance point) and the max distance in an array
- Return the array

The final step is to count the fingers using the cosine theorem, we looped over all the convexity defects and calculated all the sides created by the start point, end point and farthest point. Once having all the sides, we calculated the angle between the side created from the start point and the end point, and the side created from the start point and the farthest point. If the angle is less than 90 degrees, we knew that two fingers were raised, and accumulated the number of fingers. If the accumulated number was zero, we checked the ratio (hull area – contour area)/ contour area, if the ratio was larger than 0.12, we know that only one finger was raised.

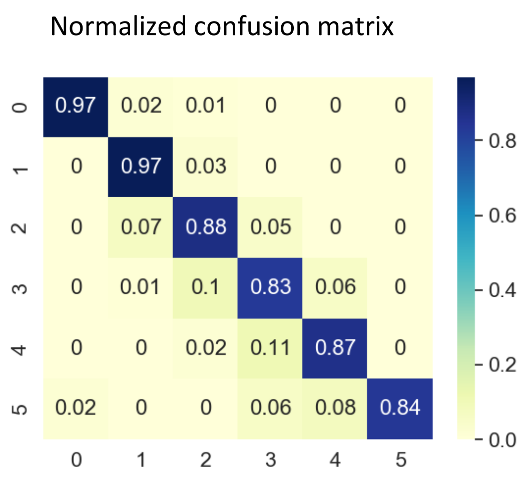
### Assumptions and limitations

In this part we assumed that the mask is clear and doesn't contain spikes meaning it smooth. To reduce false predictions, we added a condition to consider fingers raising, in addition to the angle we created a threshold based on the ratio between the finger length and the length of the palm (using the maximum and minimum point of the y axis on the contour set, and the sides that already calculated). Only if the ratio is larger than 0.25, we considered finger raising.

Another limitation for that method is that it relies on a very good segmentation. For example, if the segmentation is a bit thick, two fingers that are closed together may be confused as only one finger. Therefor, for good prediction the segmentation needs to be as tight as possible, and the raised fingers must be separated.

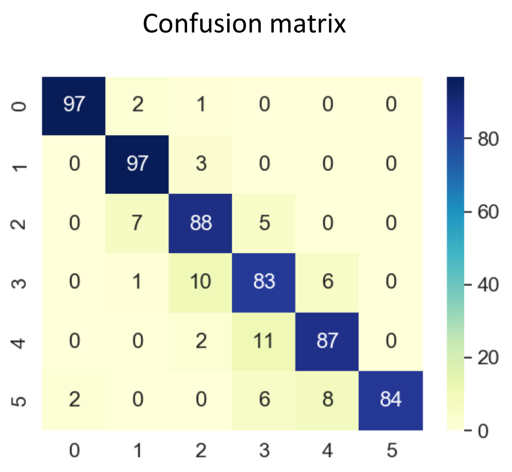
## Results

We preform tests to measure the function success, we tested each fingers permutation for 100 frames for each number, the results are shown in the figures below:



Normalized confusion matrix

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0.97 | 0.02 | 0.01 | 0 | 0 | 0 |
| 1 | 0 | 0.97 | 0.03 | 0 | 0 | 0 |
| 2 | 0 | 0.07 | 0.88 | 0.05 | 0 | 0 |
| 3 | 0 | 0.01 | 0.1 | 0.83 | 0.06 | 0 |
| 4 | 0 | 0 | 0.02 | 0.11 | 0.87 | 0 |
| 5 | 0.02 | 0 | 0 | 0.06 | 0.08 | 0.84 |



Confusion matrix

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 97 | 2 | 1 | 0 | 0 | 0 |
| 1 | 0 | 97 | 3 | 0 | 0 | 0 |
| 2 | 0 | 7 | 88 | 5 | 0 | 0 |
| 3 | 0 | 1 | 10 | 83 | 6 | 0 |
| 4 | 0 | 0 | 2 | 11 | 87 | 0 |
| 5 | 2 | 0 | 0 | 6 | 8 | 84 |

To overcome the inaccuracy, we used predictions from 20 consecutive frames and only if over 75% predictions were the same, a prediction was considered as valid.

As you can see, we were able to get good results thanks to the threshold mentioned above, the accuracy of the test is 89.3%.

## Realtime Adaptations

The function works on a binary image, and all the algorithms used in this function are efficient, so we didn’t need to make extra adaptions to achieve real-time performance for this function.

## Table segmentation and corner detection

### General information

This algorithm detects the table and its corners. The algorithm receives a regular three-channel image with a table in it, and outputs the table corners and a segmentation mask.

### Implementation

To obtain a segmentation image of the table and its corners, we first used the Sharpen filter, to highlight the edges, then we used the OpenCV threshing function, on each of the channels (R, G, B) according to the THRESH_OTSU method, the threshing function's input is one channel image and outputs binary image. Otsu's method avoids having to choose a value and determines it automatically. Consider an image with only two distinct image values (bimodal image), where the histogram would only consist of two peaks. A good threshold would be in the middle of those two values. Similarly, Otsu's method determines an optimal global threshold value from the image histogram. In this way the segmentation of the table did not depend on the color of the table but only to have a uniform and prominent color in the image, in addition with this method the segmentation will work well also if there are objects on the table. Then to clear noises, we created a threshing image of the three channels by the AND function between their binary images. Testing several color spaces (such as LAB and HSV) and different combination of channels, we found that RGB with all its channels the best one for that specific task.

In addition, we used Morphological Transformations of OpenCV. Morphological transformations are some simple operations based on the image shape. It is normally performed on binary images. It needs two inputs, one is our original image, second one is called structuring element or kernel which decides the nature of operation We used open morphologic which is basically using erosion in and then dilation.

The basic idea of erosion is just like soil erosion only, it erodes away the boundaries of foreground object, and dilation . is it is just opposite of erosion. So, it increases the white region in the image or size of foreground object increases.
In addition we used two additional kernels, one is representing horizontal lines, and the other is representing vertical lines, which features square table. In order to emphasis the edges we used Canny algorithm, with quiet small parameters, because we wanted to get as much as line we can for better binary image.
Once we have a segmentation image, we use a function which finds the largest connected components, and using Convex Hull algorithm, we fill the largest components. Convex Hull using Jarvis's Algorithm (gift wrapping algorithm) [13].

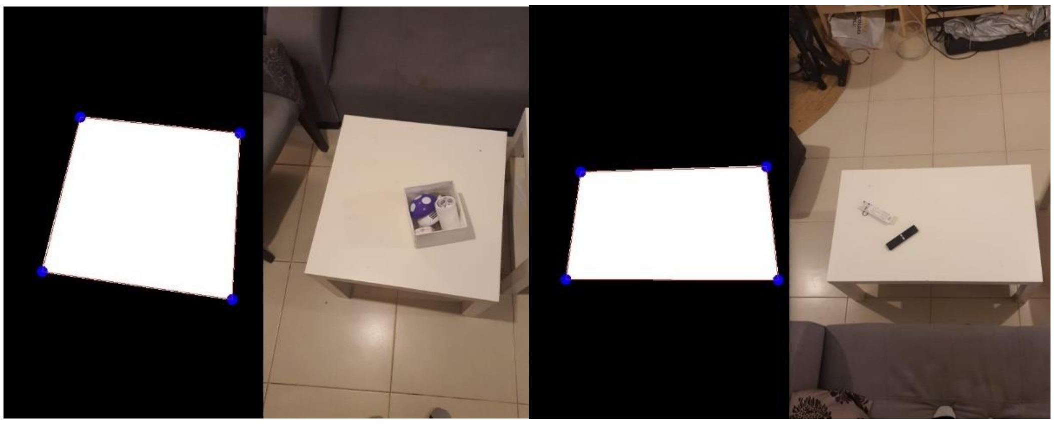
The next step is to obtain the coordinates of the table corners using the segmentation image. To do this we use OpenCV's find contour algorithm the function is basically a curve joining all the continuous points (along the boundary), having the same color or intensity, therefore works well with binary images. We used CHAIN_APPROX_SIMPLE parameter, it removes all redundant points and compresses the contour, thereby saving memory.
Then, we used OpenCV's ApproxPolyDP which approximate the contour using 4 point which represents the corners.

### Assumptions and limitations

During the implementation we used several assumptions on the table so that the algorithms we worked with would work well.

The first is that the table will be significant in the image. As we explained above, the threshold algorithm we used, selects the threshold according to the 2 main Gaussians in the image histogram. Second, we assume that the table has a relatively uniform color. In addition, a key assumption is that in order for the segmentation to work well, all 4 corners of the table must be in the image. The algorithm is robust to objects that are on the table and tables in different colors.

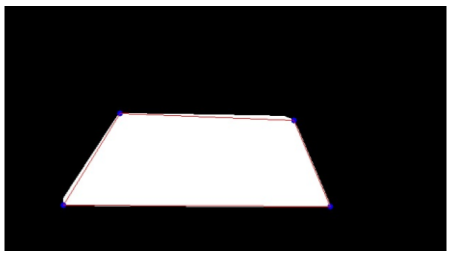
### Results

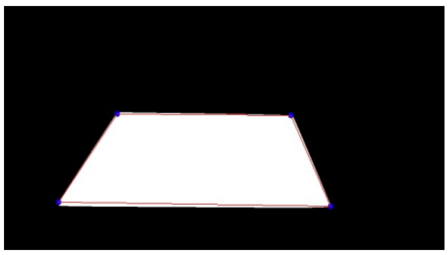


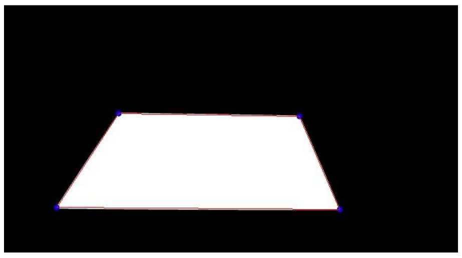
Realtime adaptations

Due to high latency, the table segmentation has not been performed every frame, but only when needed (calibration time and when we calculated that it is needed - explained in detail in table tracking section).

We realize that different scale of the image, results better segmentation and thus better corner detection. However, there is a trade-off between the scale and run time and a larger scale helped for better segmentation and corners tracking but also significantly increases the computation time.


Scale = x1


Scale = x2


Scale = x3

## Table tracking

### General information

One of the greatest challenges was to find the table corners in real-time (each frame). For that task we tried segmenting the table every frame but that was problematic for several reasons. First, due to the table segmentation limitations, in case of hands that hides pieces of the table, the segmentation is not good enough. Second, the segmentation computation time is long and does not fit the real-time allowance for that task. Therefore, we decided to track the table most of the time and to renew the segmentation only when the tracking goes wrong, or the user wants to 'find' the table again.

### Implementation

The tracking is based on both optical flow and Hough lines. The tracking algorithm implemented as follows:

1. initialization – segmenting the table and extracting its corners (as explained in Table segmentation section)
2. in each frame:
   a. draw line between each two consecutive corners and get the indexes of all the points of each line (each side of the table).
   b. Apply Lucas Kanade (LK) to track each point in each table-side at the next frame.
   c. For each tracked side points, use linear regression to find the best line that fits (excluding outliers).
   d. Find the intersections of the lines to get 4 table corners.
   e. Create 4 different side masks using the corners (thick line mask that locally concludes the real table side).
   f. Apply Canny edges on the frame and multiply each mask by the edges map to get the real table sides.
   g. Apply Hough lines to find the strongest line in each table side.
   h. Find the intersections of the lines to get 4 table corners and update them as the current corners.

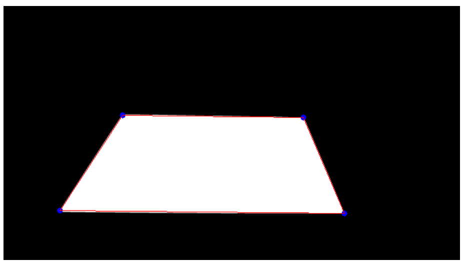
This algorithm has a high noise tolerance due to several operation that we done - excluding outliers, staying close to the real table side, and defining 'reasonable change' of the corners and retain the former corners if the change was not 'reasonable' (explained below).

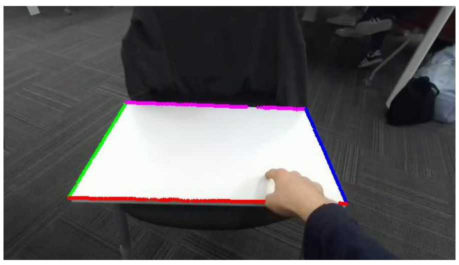
The outlier's exclusion is done with several points filters. First, if the distance from the same point in the former frame was not around the same distance as most of the points the point was considered outlier. Second, if the point was on the sleeve or hand segmentation mask.

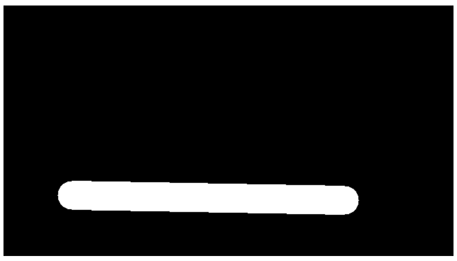
We have added the Canny edge detection and Hough lines because we found that using only the optical flow, there is a cumulative error that grows each frame in the tracking. That's allows us to 'reset' every frame and stay close to the real table sides.

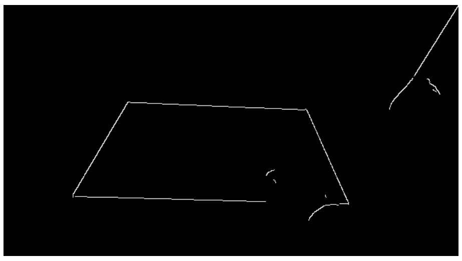
The fact that the algorithm follows every point in the tables outline makes it invariant to partially covered table edges and covered corners. It can follow the table even when a parts of it is out of the frame.

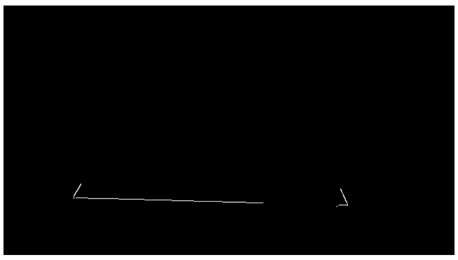
### Results

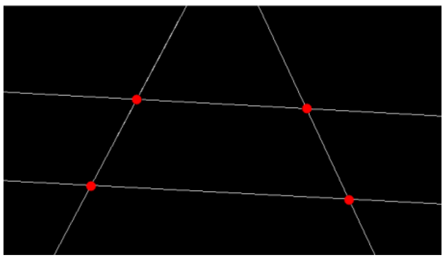

1. Initialization


2. b. After LK


2. e. one side mask


2.f. Canny


2. f. multiplies all the masks


2.g, h. using Hough and find intersections

## Assumptions and limitations

Under the assumption that the table is rectangular, and the changes of its corners are small perspective transformations (assuming small changes between frames) we defined that reasonable corner change has the following properties. There must be at least two corners that changes. The differences between the largest and the smallest change must be relatively small (around 10% of the image x size).

We used high Canny thresholds under the assumption that the table has strong edges. One of its limitations stems from that (for example, if the table will merge in its background). After trying different thresholds on different tables and backgrounds we found that the parameters that gives us good enough edges map are 400 and 500 (low and high relatively). The sigma parameter that was used was relatively high to blur the small edges.

We wanted an algorithm that helps following some chosen key points and the fact that LK's assumptions are "mostly" (explained below) fulfilled, and it is a fast algorithm that specializes in following good features made us choose it. LK's inputs are a set of coordinates and two consecutive gray scaled frames. It tracks the coordinates on the first frame and outputs the location of the features that were in the input coordinates and outputs the new coordinates on the next frame. The main idea of the algorithm is solving the optical flow equations to a small window around each point.

The assumptions made using LK are that the brightness is the same in each frame, the motion of the points is small and the points that are neighbors, moves the same. Those assumptions are forcing the lighting of the scene to remain constant in order of the algorithm to work properly. In addition, the small motion translates to small changes between two frames which either means to move the camera slowly or to increase the frames rate. The frame rate is determined by the computation time of each frame so for the LK to work properly, the real-time computation must be fast enough (we managed to speed the computations up to 15 frames per second). Speeding up the computations per frame we managed to use a relatively small window (30x30) and not a lot of levels (6 pyramid levels) as the LK parameter. The third assumption of the LK algorithm is the one that probably created the cumulative error that we got and made us use Hough lines also (to stick to the true table corners).

For good features like corners, the LK should perform well, the problem with that is that it is very likely that the user will cover the corner and it won't be visible for the algorithm to track. That is why we added the rest of the side points. Considering the LK's aperture problem[1] we understand that the algorithm will not perform so good tracking the sides of the table. Although that the assumption doesn't hold here, it is not a problem because the tracking will still be on the table's side. As long as the point track lands on the side of the table, it counts as a good point to our algorithm.

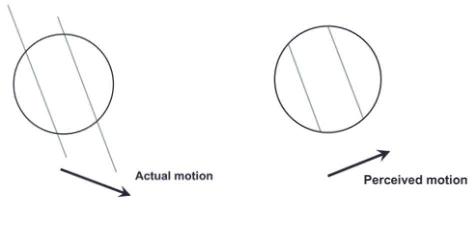
The Hough lines parameters we chose were low $\theta$ and $r$ resolution (1 deg and 1 $r$ resolution) to enable a fast computation needed by the real-time property. An assumption we made using the Hough lines here is that there is one main line in each side-edges image. This is true if the optical flow and Canny algorithms are working properly. The threshold is a small one (50) because there is no need on filtering lines if there is only one main line and it is the relevant one.

One more limitation that the algorithm has is that if a hand covers the whole side of the table, it won't be able to track that side because of the outliers filter we made. If the whole side is completely out of the frame, it will be also non trackable by the algorithm.

## Realtime adaptations

Most was discussed above the rest of the topics. In addition, we are using threads to follow both the right and left image's table corners for parallel computation.

[1] The aperture problem is that the perceived motion is not the same as the real motion and happens due to the small window size. it best described by the following images.



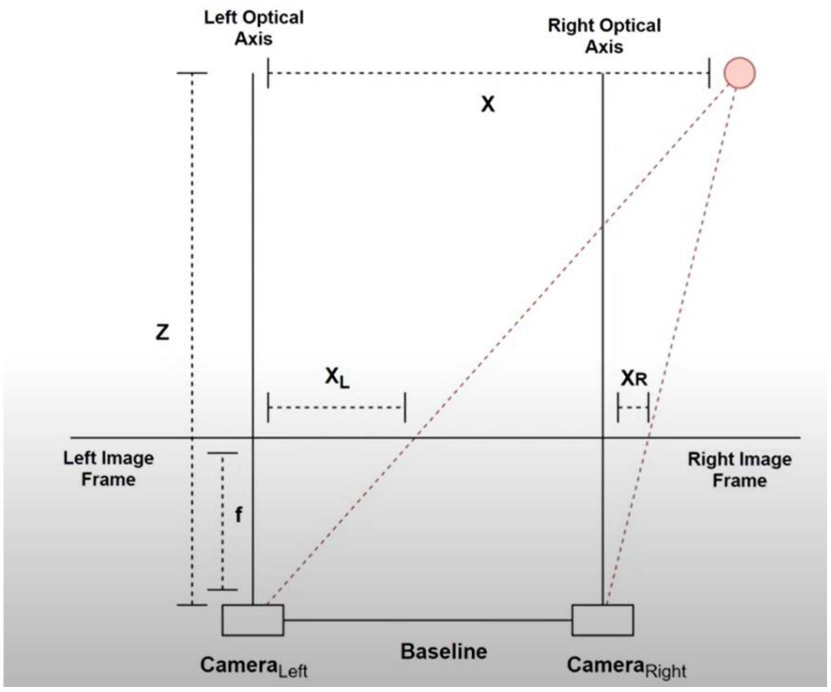
## Distance calculation

### General Information

In order to detect touches, we needed to create a distance map. Since we did not use sensors, we calculated the distances using the two images (right and left). The function gets x vectors of both left and right camera and return distances vector.

### Implementation

To calculate the distance, we used a triangle similarity for 2 points (first point the side in the right image and second in the left image).



$$X_L = \frac{X}{Z} * f \qquad X_R = \frac{X - b}{Z} * f$$

$$Disparity = X_L - X_R$$

$$Z = f * \frac{b}{Disparity}$$

when

- $X_L - x\ coordinates\ of\ left\ image$
- $X_R - x\ coordinates\ of\ right\ image$
- F – focal length of camera (350 pixels in our case)
- B – base distance between the to lens of the camera ( 12 pixels in our case)

### Results

Calculate the distance we performed using this method, with an accuracy level of up to 5 cm on average, and Std around 1 cm, relative to the actual distance. This level of accuracy is sufficient for us because we end up using this function to calculate a distance map of the hands and of the table, and the error is relative to both so that it hardly affects the result.

### Realtime Adaptations

In order to speed up the calculation as part of real time adaptation, we performed the calculation in vector form.

## Hand distances map

### General Information

To create a good distance map of the hands we needed to match each pixel of the hand in the right image to the same pixel in the left image. To do that, we started by using feature detector to find matching features between the hand in both images. Then, with the help of projective transformations we could find the pairs of matching pixels for the whole hand and send it to the distances function above.

Then, by estimating the projective transformation for the hand between both images we can create a list of corresponding hand's pixels for both the right and left images. This list can be the input of the distances function to create a distance map for the hands.

### Implementation

First, we used the "FAST" algorithm [14] to identify the key points on the hands area using the binary hands segmentation. This algorithm inputs an image, a mask and a threshold and outputs a set of key-points. In this algorithm we can use the max suppression flag to get key-points that are distributed more uniformly across the mask area. Using ORB we computed the descriptors of those key-points. ORB computes the descriptors using Binary Robust Independent Elementary Features, BRIEF [19-20]. The input of BREIF are the key-points and its output are the descriptors.

Then, using the Brute-Force match function we found matches between the features of the right and left image, Brute-Force matcher is simple. It takes the descriptor of one feature in first set and is matched with all other features in second set using Hamming distance calculation. The distance is measured by the distance between the 2 descriptors, the smaller the distance it means they are more suitable for each other, so we sort the matches by distance, and took the first 35.

With those matches we can estimate the projective transformation between the hand in the right and left image using the pseudo inverse method. When we have the transformation matrix, we are able to find the corresponding pixel in the right image to each pixel that is on the hand's segmentation in the left image. That set of pixels can be sent to the distances function (mentioned before) to obtain the hand's distances map.

### Assumptions and Limitations

The FAST algorithm selects a pixel $p$ in the image which is to be identified as an interest point or not. Let its intensity be $I_p$. It selects appropriate threshold value t, consider a circle of 16 pixels around the pixel under test, now the pixel p is a corner if there exists a set of n contiguous pixels in the circle (of 16 pixels) which are all brighter than $I_p + t$ , or all darker than $I_p - t$. If $p$ is a corner, then at least three of these must all be brighter than $I_p + t$ or darker than $I_p - t$. If neither of these is the case, then $p$ cannot be a corner. This algorithm is faster than other existing feature detectors and is very compatible for real-time application. However, it is not robust to noise, and it is dependent on a threshold. The threshold selection was made by trial and error while the base line was to find a relatively large number of key points (so in high probability, at least some of them will be good enough).

The BRIEF algorithm is also targeted for real-time applications and its main idea is like the SIFT. It uses 128-dim vector for descriptor, with normalized orientation. The difference is that for fast matching, the vector needs to be converted to a corresponding binary string so a simple XOR can be applied to find the Hamming distance between two features. Here, the BRIEF It provides a shortcut to find the binary strings directly without finding descriptors. It takes smoothened image patch and selects a set of $n_d(x, y)$ location pairs in a unique way (explained in paper [20]). Then some pixel intensity comparisons are done on these location pairs. For example, let first location pairs be p and q. If I(p)<I(q), then its result is 1, else it is 0. This is applied for all the $n_d$ location pairs to get a nd-dimensional bitstring. One important point is that BRIEF is a feature descriptor, it doesn't provide any method to find the features so it needs to be combined with key-points finder (such as FAST) as we did here.

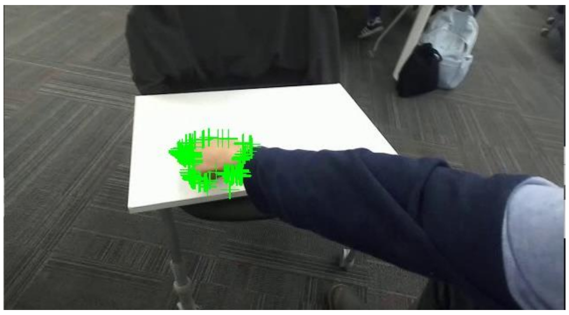
We assume that there are no scale differences between the hands in the right and left image, this is a reasonable assumption due to the proximity between the two lenses of the camera.

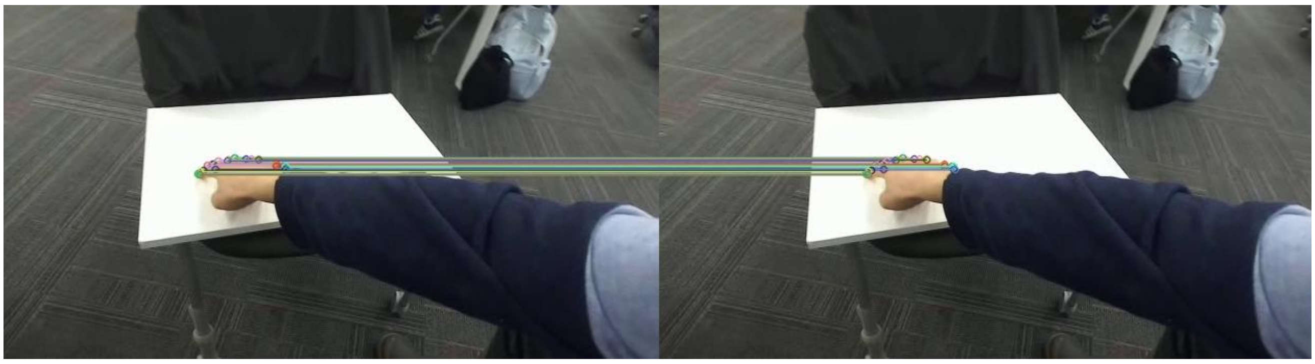
In addition, this algorithm significantly relies on the quality of the segmentation of the hands, i.e., since the search for features is performed solely on the segmentation image, if the segmentation of the hands is not sufficient, no good enough features will be found. Moreover, if no good features are found in this part, the distance map of the hands will not be good enough to detect the touches.

Calculating the distance, we assumed that the projective transformation of the hand's segmentation map was 'good enough'. To get a measure of the quality of the transformation we used a Jaccard measure between the transformed left segmentation map to the right segmentation map. If the Jaccard index was not good enough that means that something went wrong along the way (bad segmentation, not good enough descriptors, not good enough matched descriptors, etc.).

In this case, we didn't consider the hand's distances map as valid. The threshold Jaccard index for a valid transformation was 0.4 and reached by trial and error with respect to the accuracy of the touches that we defined (detailed in the touches section).

## Results


Key- Point detection with Fast algorithm

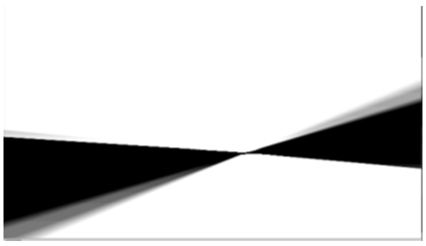

Best ORB descriptor matches

left image hand's segmentation

left hand's good transformation (0.97 Jaccard)

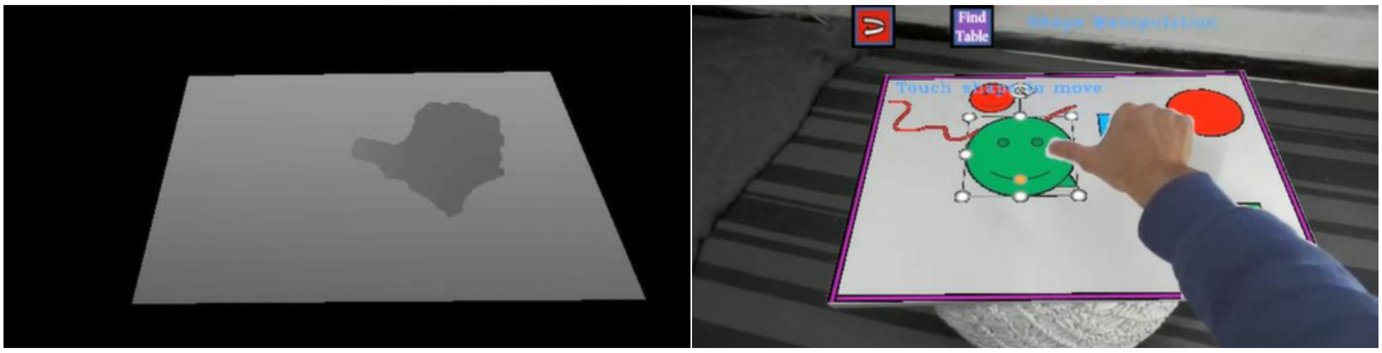
right image hand's segmentation

left image hand's segmentation


left hand's bad transformation (0.03 Jaccard)

right image hand's segmentation

After trial and error with the hand's matching parameters we found the best ones for us.



Realtime Adaptations

To reduce latency, we used Fast algorithm which is not robust to high levels of noise. But considering the tradeoff between speed and accuracy it suits us the best.

To improve the descriptors' calculation time, we used `0` as the scale parameter of the ORB, because we know that both images are in the same scale.

## Table Distance Map

### General information

Generating a distances map for the table using the four table corners of each image (left and right). The map will be used for the detection of table touching indexes.

### Implementation

The algorithm uses the four table corners of each image (left and right) to estimate the 4 distances to each corner (with the distance function as explained before). The distances and the corners coordinates create 3D points in the image space with distance $(x, y, z)$. From linear algebra we know that only 3 points are required to determine a 3D plane equation $z = ax + by + c$. Using pseudo inverse we calculate the best fitted 3D plane according to the following equation:

$$A = \begin{pmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \\ x_4 & y_4 & 1 \end{pmatrix}$$

$$\begin{pmatrix} a \\ b \\ c \end{pmatrix} = (A^T A)^{-1} A^T \begin{pmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{pmatrix}$$

Creating a 2D 'heat map' of the whole plane and cutting it as a rectangle with the table corners we get the distances map of the whole table.

We measured the tables width and length after calculating the distances to the corners at the first time using the equations presented in [15] and used them to clear noises of bad corners following/detecting.

### Assumptions and limitations

Assuming that the table is in one 3D plane enabled us to use the equations above.

We created an error detection mechanism to prevent bad corners following to mess the distances map and thereby add noise resistance to the algorithm. Under the assumption that the table is must be close enough but not too close (most of the table still must be inside the frame) we skipped the calculations and returned the former map if one of the corners distances were more than 6m or less than 20cm. in addition the difference between the farthest corner and the closest one must be less than (or equal) to the tables diagonal (equalization can be achieved when looking at the table from the side and the corners are in aligned). Comparing the width and height of the first measure we can clear even more noises of bad corners following.

### Results

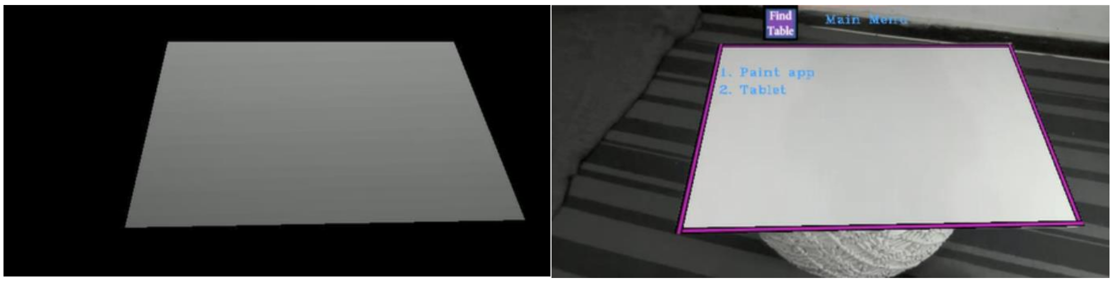


Table distances 'heat map'

### Realtime adaptations

At first, we tried to use the 4 corners in the left and right image to estimate the projective transform between the left and right images. Then use the transform on the segmentation map of the left image's table to create pairs of points and their corresponding points in the right image and send all of them to the distances function. After more thinking we found that using the assumption mentioned above we can use much less computations to get the map.

## Touches detection, clicking and projecting

### General information

Using both the table and the hands distances map we calculate the indexes in which the hand is touching the table. We use the indexes to generate mouse clicks in the computer in the relevant location and then project the screenshot of the relevant area of the computer to the table in the images (both left and right).

### Implementation

We found the tip of the finger via center mass calculation of the hand segmentation and the defined the farthest point from this center mass as the fingertip. Then, we cross-check the tip indexes with the corresponding indexes at the table map to find how close the tip is, to the table. A touch is found if there are nonzero values in the table distances map at the fingertip indexes and the distance on the table there and the distance of the fingertip is closer than the tolerance we decided (4 cm tolerance). We consider the touch index as the center of the whole touching area.

Using the four table corners we estimate the projective transformation of the table to the relevant computer screen allocated area and project the touch index from the table to the relevant place on the screen where the mouse clicks occur.

After that, we use the same estimated projective transformation to project a screenshot of that screen area into the images (both left and right). The screenshot is thereby projected to the table are in the image.

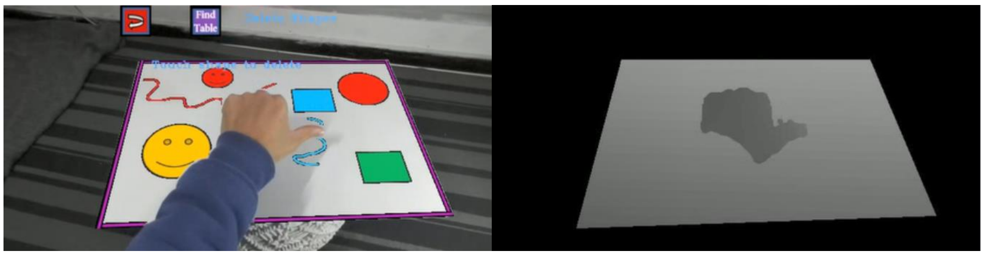
### Assumptions and limitations

The most relevant assumption here is that both the distances maps are good and valid. In case of invalid hands distances map the touch is considered as the last frame touch.

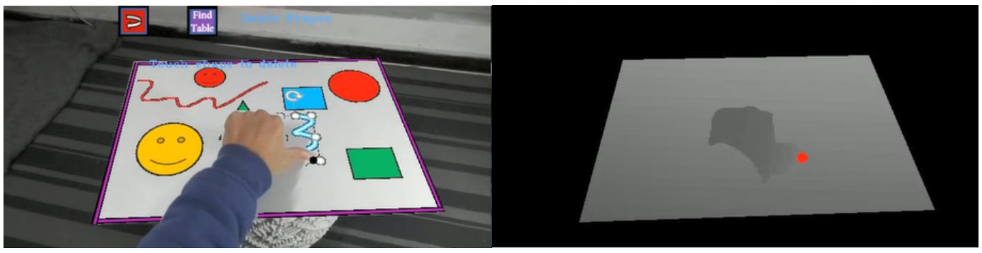
The touch is made when the hand is fisted but one finger (for the fingertip detection). This was done to clear touching noises.

The accuracy of the touch is derived from the accuracy of the distances that we have calculated and some noise. Finger that is less than 4cm from the table considered as a touch.

### Results



No touches have been detected



A touch has been detected

### Realtime adaptations

The screenshot projection to each image is done in parallel using threading because they are independent. In addition, we tried many projection functions, screenshot functions and mouse clicking from different libraries, compared their performances and used the fastest that we found.

## AR on Binocular Optical See-Through Displays

### General information

The main idea of the project is to create augmented reality. Most of the AR applications these days are with cameras and displays. To enable good AR applications with 3D perception, it is needed to overlay correct 2D positional alignment. In our project, we screen the output image as a stereo image, and each eye gets a different image with the correct alignment.

### Implementation

As we learned in the project, the eyes see the same object in different positions due to the distance between the eyes. The positions vary between objects, depending on their distance to the eyes.

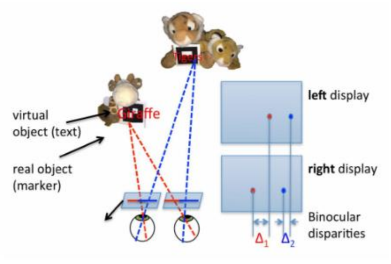


*Figure1 : Different offset accordingly to the distance*

We can separate the image correction into two parts: GUI & table projection. On the GUI, we had to add an offset to the right image for all the text. The offset is constant, and it gives a perception that the GUI is far enough to read and ease the eye.

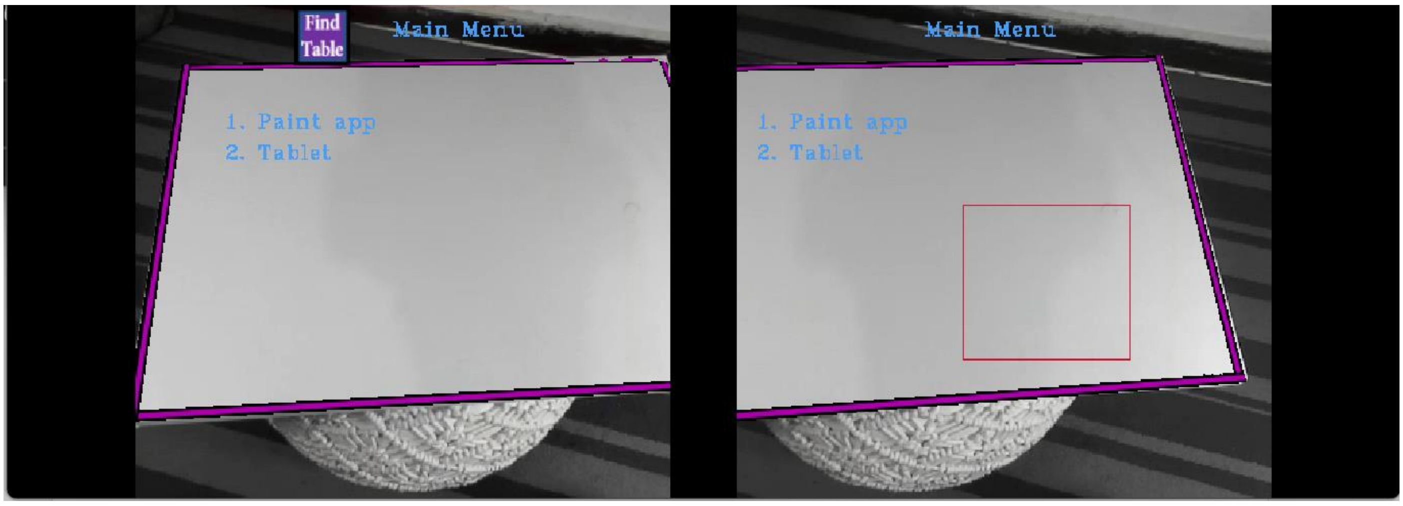
Table projection was more important for integrating virtual content into a real scene but easier. We did the corner detection on each image from the stereo image for a distance map, and by that, we received the accurate projection and position of the table. Therefore, the offset of the content between each image changed accordingly to the distance to the object. (Opposite the GUI that was constant).

The last part was to modify the image to fit in a VR headset using a phone as a display. When the display is right on the face, there is a content in the middle and the sides the eyes can't reach. After removing the dead zones, we helped the eyes by focusing only on the important things.

### Assumptions and limitations

The only assumption that we have is the screen display of the phone. Different screen sizes will change our calculation. We chose the constant GUI offset according to our phone setup.

### Results



## Video streaming

### General information

The AR system is built from three main parts: camera, PC, and phone. These three components communicate between them and transfer data. This data is video streaming.

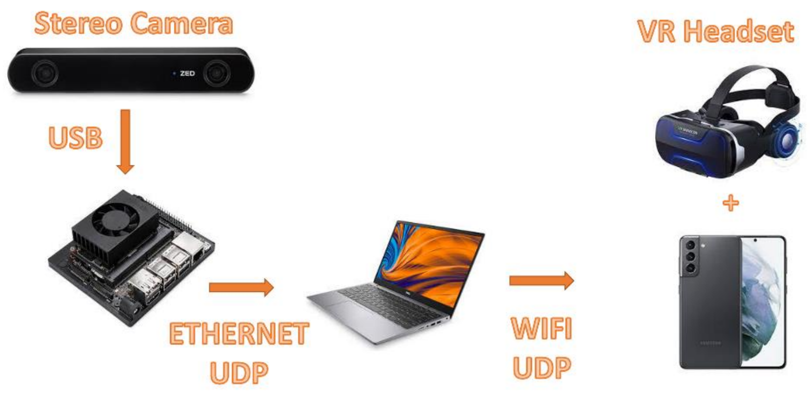
### Implementation

First, we need to capture a frame from the camera as eyes work. Our camera is a stereo camera that works in high resolution and fps. To support the advanced camera, we had to use GPU to receive the camera's data. We connected the camera to Jetson Xavier by Nvidia with a USB peripheral. A wired network cable connected the Jetson to the PC.

One of the most video compression standards is Advanced Video Coding (AVC), also referred to as H.264. This standard is based on block-oriented, motion-compensated coding. Some main components improve the compression as DCT (As we learned in class), motion compensation, intra prediction (neighborhood prediction, and CABAC.

We implemented a pipeline using the Gstreamer framework to send the compressed camera frames in efficiency compression over UDP protocol to ensure that we will have the lowest latency. The PC received the stream in OpenCV with the Gstreamer plugin, and then we implemented the AR algorithm.

The last part is streaming the content to the phone. We created an Android app with a Gstreamer framework to receive streams from the PC to the phone wireless in UDP protocol.



### Assumptions and limitations

The problem with H.264 coding is that if the processing time of each frame is too big, the video preview starts to pixelized. To overcome that problem, we had to capture a new frame only when the last frame was processed. That way, we managed to keep low latency and good image resolution but a low frame rate that can interfere with the user experience.

Another limitation is the network bandwidth. While the PC was connected to the GPU wired, the phone was connected with Wi-Fi. If the network bandwidth was low, we miss UDP packets, resulting in many bad phone pixels.

## Conclusions

We chose to do a project that involves many challenges and problems that forced us to use a wide variety of algorithms and methods some of which we learned during the course and some of which we learned while working.

One of the main challenges was real-time adaptation as the project type forces fast computation. During the work, we realized that there is a clear tradeoff between the speed of running the algorithms and their results accuracy. Along the way we tried to find the right balance for us. While trying to get the best of both worlds we used parallel processing tools, looked for faster identical algorithms and even implemented several functions ourselves to reduce latency. On the other hand, we have seen that proper adaptation of the algorithms' parameters is significant both for run time and for accurate in results.

Despite the large computational complexity required to perform at each frame we were able to output about 15 frames per second without the use of a GPU during the processing.

Another challenge was that the flow of the program was built in the form of algorithms that are based on the outputs of previous algorithms (for example, touches detection relies on the distances maps which relies on the table tracking). If there are errors or noises in some of the algorithms, the noise seeps and increases as we progress through the processing flow. This problem has forced us to find solutions for detecting and cleaning those noises at every stage in order to get the results that we got.

In addition, we learned that the use of color-based segmentations is very sensitive to lighting changes and requires calibration tailored to lighting conditions and appropriate use of color spaces.

The project was meaningful and instructive for us. Working on it, we experienced in working as a team and acquired a wide range of applied and research tools.

Algorithms and methods we use: GMM, Morphological Transformations, findcontour, Color spaces, Jaccard, Bag-Of-Visual-Words, SIFT, Kmeans, SVM, convexHull, THRESH OTSU, Optical Flow, Hough lines, Canny, Fast, Projective transformation, ORB, Brute-Force matches.

## Bibliography

[1] https://en.wikipedia.org/wiki/CIELAB_color_space#Where_CIELAB_is_used.

[2] https://en.wikipedia.org/wiki/HSL_and_HSV#Disadvantages.

[3] https://en.wikipedia.org/wiki/Helmholtz%E2%80%93Kohlrausch_effect.

[4] https://scikit-learn.org/stable/modules/generated/sklearn.mixture.GaussianMixture.html.

[5] https://en.wikipedia.org/wiki/Mixture_model#Gaussian_mixture_model.

[6] Interduce to image processing – Ben Gurion University.

[7] https://en.wikipedia.org/wiki/EM_algorithm_and_GMM_model.

[8] article "Satoshi Suzuki and others. Topological structural analysis of digitized binary images by border following. *Computer Vision, Graphics, and Image Processing*, 30(1): 32–46, 1985".

[9] https://docs.opencv.org/4.x/da/df5/tutorial_py_sift_intro.html

[10] Lowe, D.G. Distinctive Image Features from Scale-Invariant Keypoints. International Journal of Computer Vision 60, 91–110 (2004).

[11] https://medium.com/analytics-vidhya/bag-of-visual-words-bag-of-features-9a2f7aec7866

[12] Jack Sklansky. Finding the convex hull of a simple polygon. *Pattern Recognition Letters*, 1(2): 79–83, 1982.

[13] https://en.wikipedia.org/wiki/Gift_wrapping_algorithm

[14] https://vovkos.github.io/doxyrest-showcase/opencv/sphinx_rtd_theme/page_tutorial_py_fast.html

[15] https://stackoverflow.com/questions/42035721/how-to-measure-object-size-in-real-world-in-terms-of

[16] https://www.researchgate.net/publication/283117341_Depth-Disparity_Calibration_for_Augmented_Reality_on_Binocular_Optical_See-Through_Displays

[17] Wu, Wanmin & Tosic, Ivana & Berkner, Kathrin & Balram, Nikhil. (2015). Depth-Disparity Calibration for Augmented Reality on Binocular Optical See-Through Displays. Proceedings of the 6th ACM Multimedia Systems Conference, MMSys 2015. 10.1145/2713168.2713171.

[18] https://en.wikipedia.org/wiki/Advanced_Video_Coding

[19] https://www.cs.ubc.ca/~lowe/525/papers/calonder_eccv10.pdf

[20] https://docs.opencv.org/3.4/dc/d7d/tutorial_py_brief.html