ENPM673 - PERCEPTION

PROJECT 1

PABLO SANHUEZA

ANDRE GOMEZ FERREIRA

ADHEESH CHATTERJEE

Table of Contents

Introduction

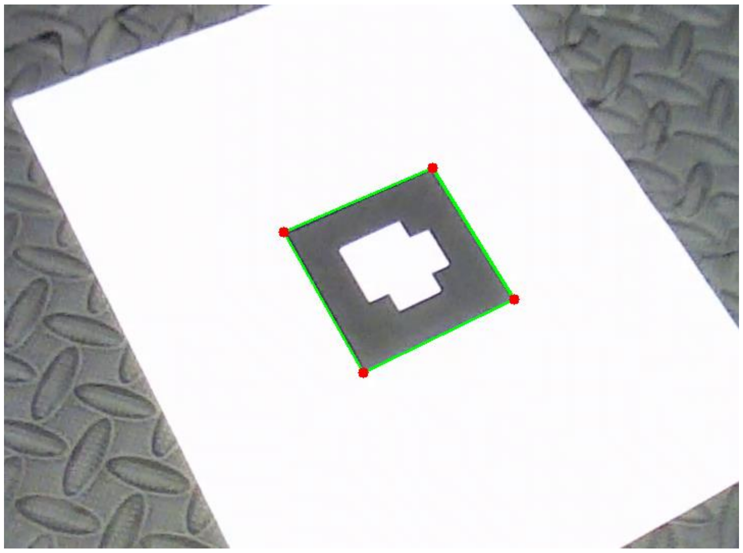
The goal of this project was to use computer Vision to identify a custom AR tag and perform some desired tasks. We will cover important concepts and mechanisms utilized in this project

Detection

Corner Detection

The initial step of this project consists in finding the four corners of the tag placed on the ground on a white paper. In order to find the desired corners we first changed the image to gray scale and applied the cv2.findContours function. This function retrieves contours from a binary image, which is very useful for shape analyses and object detection. Converting image to gray helps to avoid undesired contours.

After finding the contours, cv2.approxPolyDP is used. This function approximates a polygon/curve to another polygon/curve with less or equal precision making possible to detect corners of the resulting polygons. After finding the points we filtered them by area, hierarchy and convexity which gave us the desired points.



Before achieving success, we had a lot of issues to trying to find the points of the tag. The main problem was trying to detect the contours and the points that show up in the back part of at the end of the video. To solve that problem we tried to apply blob detection and crop the image taking out all the contours that were not necessary. The solution worked well but after a while we found that the actual solution applied to the project was much simpler and worked much well than blob detection

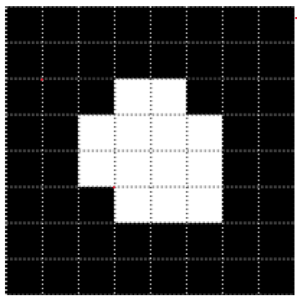
Another big problem we had is that for some frames of the video the does no find the points in the corner and because of that the video crashed at the same frame all the time.

The solution for that was to make sure that at the time where the points from the corners are not found, the code will use the points from previous corners.

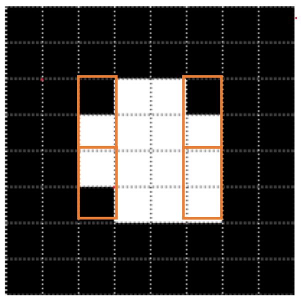
Encoding Scheme

After successfully detecting the tag using contours, area, convexity and length equaling a number of 4 points, we were able to encode the data provided by each unique tag. As mentioned in the project description, the AR Tag can be decomposed with an 8x8 grid, which is where the most important data lies.

To achieve and 8x8 grid the group resized the tag from a 200x200 to an 8x8. This can be achieved by either averaging the values of each 25x25 matrix within the 200x200 original image. Averaging each 25x25 matrix will result in an 8x8 image. An even easier solution is to use the cv2.resize() function, which automatically resizes the image, and by setting the size to an 8x8, the data within the image will automatically average and give a proper 8x8. Using the given reference marker as an initial test, the group later then utilizes this function to encode the data within for each frame during the videos provided.

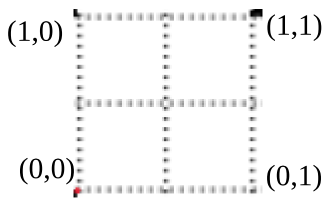


The inner 4x4 provides the orientation of the only white corner on the bottom right of the 4x4 grid. To detect this corner the team created a function that takes the average of each corner within the 4x4 grid with its respective neighbor, up or down (these are enclosed with orange/red in the figure below).

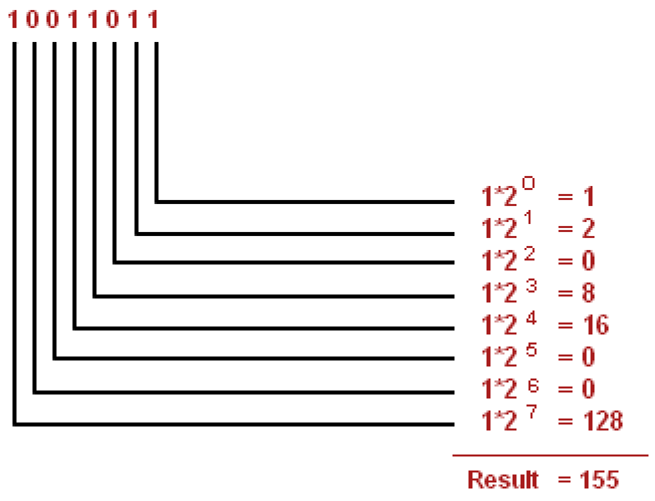


When bringing the tag upright from the video frame, the tag becomes very blurry and warped at certain times of the video, and therefore it is very hard detect the corner at certain times of each video. This usually happens when the person recording the camera rotates in a fast manner, and therefore the detection method to obtain the index of the corner fails due to the fact that there may be noise in the rest of the corners of the upright tag. Due to these uncertainties the function made by the group failed for several parts of the video, where Lena rotated in random ways.

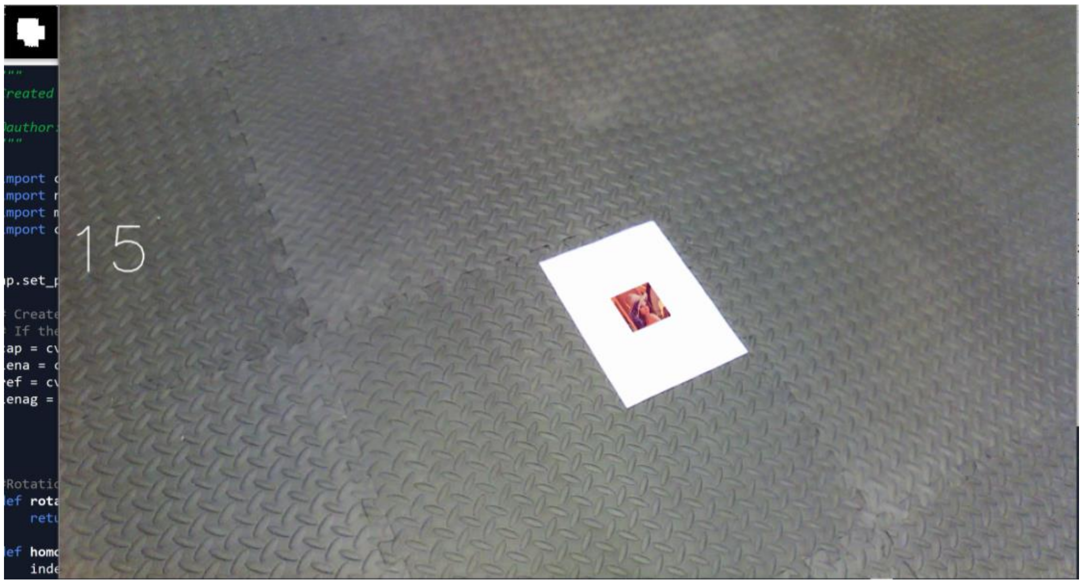
Lastly, the inner-most 2x2 grid provides the binary representation of the tag's ID. These are stored in a clockwise direction, in the order of significant bits. The top section of the 2x2 has the least significant bit, and the top left corner has the most significant bit.



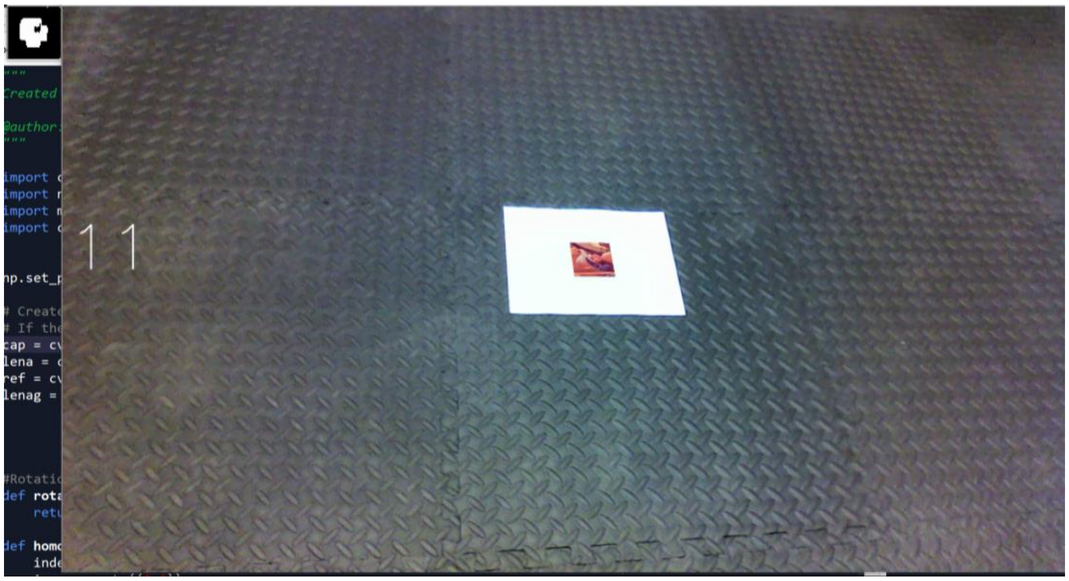
The function that checks for this section of the encoding scheme checks for each individual coordinate as the figure shown below. This can be seen in the code as scheme2x2(), which takes as input an image, and resizes it to a 2x2 matrix. It then checks for each location and checking for either values of 255 or 0. From these we can get binary values of 0 and 1's. As a result, we interpret these as the number of the tag, but they're being converted to an integer. Please see example below.



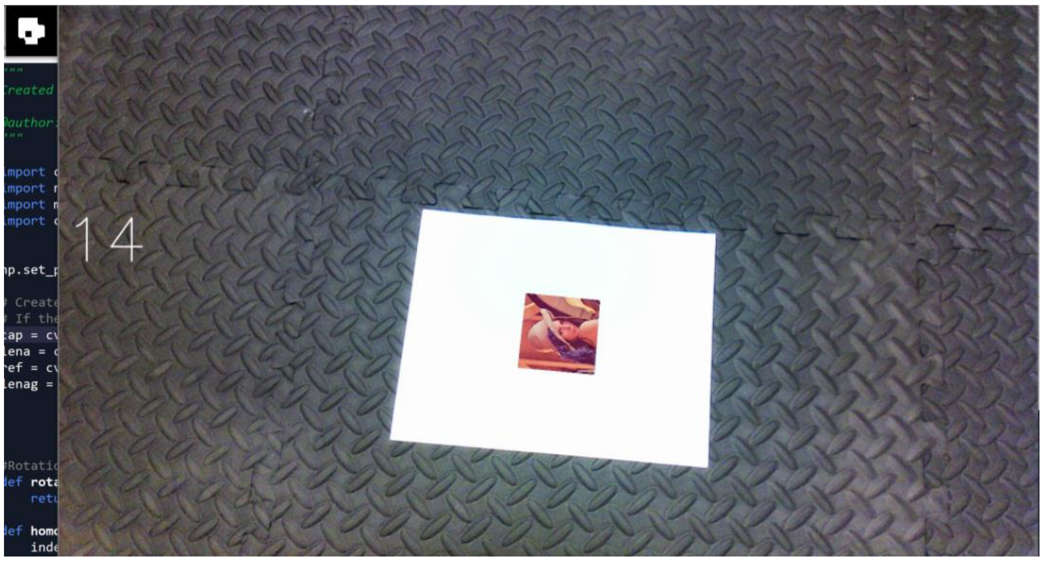
One example from one of the given tags is that it will give you a binary value of 1111. This converted to an actual tag number is: $1 * 2^0 + 1 * 2^1 + 1 * 2^2 + 1 * 2^3 = \mathbf{15}$. This conversion is what is displayed in the videos generated by the group's code. Shown below are some screenshots of the actual values obtained from tracking the tag.



The binary value from the Tag 0 video is 1111. This is the same as displaying the number 15 as an integer.



The binary value for Tag 1 video is 1011. This is the same as displaying the number 11 as an integer.



The binary value for Tag 2 video is 1110. This is the same as displaying the number 14 as an integer.

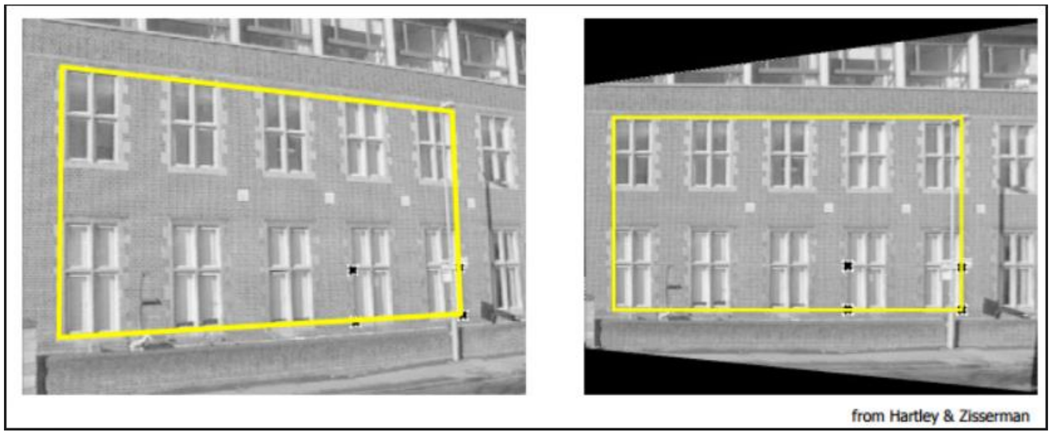
The rotation aspect for this encoding scheme was done by using three functions. Two of these functions were written by the group, which are rotate() and orientation(). The third, which is only to rotate the Lena and maintain it in the original state, is done by cv2.getRotationMatrix2D().

The rotate() function returns an in integer which gives the amount of rotations in 90 degrees to reach original state. The index fed to the rotate function is given by the orientation function. This inputs the upright tag and resizes the tag to a 5x5. From the 5x5 we extract the inner 4x4, and then analyzing the corners. If one of the corners is greater than 200 (close to a white value), it will return the corner given by the direction of the tag. With the rotations given from the rotate function, we are able to maintain the image in the right position throughout the video by using the cv2.getRotationMatrix2D().

Tracking

The second step on this project was to try to find the relationship between the reference tags gave to us and the tag of the video which can be found by calculating the homography.

In the field of computer vision homography relates two images of the same space. Calculating this relationship has many practical applications like image ratification, image registration, computation of motion and localization of a desired object where this last one is the main focus of our project.



In the step the goal is to find the relationship between a point of the Reference tag and it's respectively position in the video. Since we were able to find the 4 positions in the corner of the tag and we can easily find the positions of the corners of the reference tag and then calculate the homography matrix.

$$\boldsymbol{x}_k^{(c)} = \lambda.\boldsymbol{H}_c^w \boldsymbol{x}_k^{(w)};$$

Where $x_k^{(c)}$ represents the point from the image coordinates and $X_k^{(w)}$ and H represents the 3x3 homography matrix that we want to find.

$$\boldsymbol{H}_w^c = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{32} & h_{32} & h_{33} \end{bmatrix}$$

Since H_w^c has only 8 degrees of freedom we can fix the value of $h_{33}\ as\ 1$.
Once we have the 8 points necessary to calculate the homography, we just need to solve the following system to find the values of the matrix.

$$\begin{bmatrix} x_1^{(w)} & y_1^{(w)} & 1 & 0 & 0 & 0 & -x_1^{(c)}x_1^{(w)} & -x_1^{(c)}y_1^{(w)} & -x_1^{(c)} \\ 0 & 0 & 0 & x_1^{(w)} & y_1^{(w)} & 1 & -y_1^{(c)}x_1^{(w)} & -y_1^{(c)}y_1^{(w)} & -y_1^{(c)} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \\ x_4^{(w)} & y_4^{(w)} & 1 & 0 & 0 & 0 & -x_4^{(c)}x_4^{(w)} & -x_4^{(c)}y_4^{(w)} & -x_4^{(c)} \\ 0 & 0 & 0 & x_4^{(w)} & y_4^{(w)} & 1 & -y_4^{(c)}x_4^{(w)} & -y_4^{(c)}y_4^{(w)} & -y_4^{(c)} \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

This system can be easily solved via singular value decomposition (SVD). Once the decomposition is made the values of the matrix can be obtained by the last column of the V. Since we require that the last element of V to be one we need to normalize with the value of this element.

$$\boldsymbol{h} = \frac{[v_{19}, \ldots, v_{99}]^T}{v_{99}}.$$

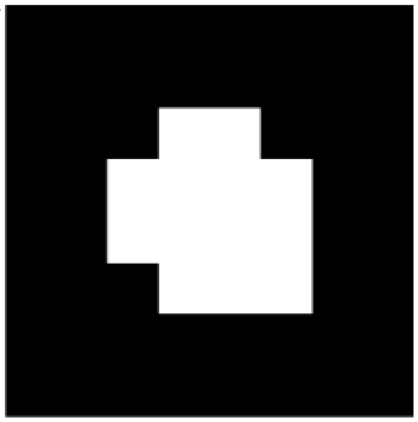
We calculated the homography matrix a little different from what was explained before as you can see in the following equation.

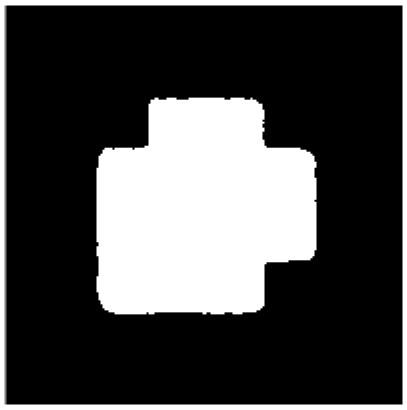
$$\begin{bmatrix} x_1^{(w)} & y_1^{(w)} & 1 & 0 & 0 & 0 & -x_1^{(c)}x_1^{(w)} & -x_1^{(c)}y_1^{(w)} \\ 0 & 0 & 0 & x_1^{(w)} & y_1^{(w)} & 1 & -y_1^{(c)}x_1^{(w)} & -y_1^{(c)}y_1^{(w)} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_4^{(w)} & y_4^{(w)} & 1 & 0 & 0 & 0 & -x_4^{(c)}x_4^{(w)} & -x_4^{(c)}y_4^{(w)} \\ 0 & 0 & 0 & x_4^{(w)} & y_4^{(w)} & 1 & -y_4^{(c)}x_4^{(w)} & -y_4^{(c)}y_4^{(w)} \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{bmatrix} = \begin{bmatrix} x_1^{(c)} \\ y_1^{(c)} \\ \\ x_4^{(c)} \\ y_4^{(c)} \end{bmatrix}$$

We solved this using psudo-inverse:

$$AH = B$$
$$A^T AH = A^T B$$
$$H = (A^T A\,)^{-1}\,(A^T B)$$

Once we have the homography matrix it is possible to unwarp the tag from the image by finding the respect position of the video tag in the frame of reference Tag.


Referance_Tag


Unwarped Tag

After unwarping the tag, it is easier to find the orientation and to place Lena on the respective position of the tag on the image frame.

Superimposing

One of the Goals of the project is to place the image of Lena on the position of the tag in the video. To solve this problem, we unwarp the video tag putting in the desired frame (just like in the previous photo), then we resize the image to 200x200 which is the same size of the tag and place pixel by pixel of the desired location. The final step of this process consists in placing the tag with the image of Lena, back to its original position.

To complete this task we apply homography again but this time trying to find the position of the of the points from the tag to its respective position on the original frame.

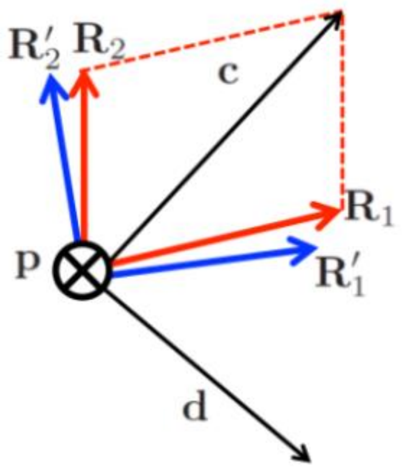
We have a problem with that part, since we were not allowed to use cv2.WarpPerspective to place the picture in the tag, we created a method to place the image pixel by pixel in the desired location. However since we have a large number of pixels it requires a lot of computation time making the video very slow. We tried to recode this process in order to make it faster, getting rider of some unnecessary for loops and trying different approaches for some parts of the code but we still were not able to accomplish big improvements in the speed. We talked to the TA about the problem and he said there was no problem to leave it is.

Placing Virtual Cube on Tag

To place the virtual cube on the tag, we need to first do pose estimation from the plane. However, the way we do it is a little different from the document given. Instead after a lot of searching, we found what we thought was an easier algorithm of calculating the projection matrix.

We first compute the rotations along x and y axis as well as the translation. Then we normalize the vectors and compute an orthonormal basis. We do this by taking R1=G1, R2=G2 and t=G3
Now, since the external calibration matrix [R1 R2 R3 t] is an homogeneous transformation that maps points amongst two different reference frames we can be sure that [R1 R2 R3] have to be orthonormal.
Hence, theoretically we can compute R3 as the cross product of R1 and R2. However, we cannot be sure that R1 and R2 are orthonormal to begin with. So we try to find a pair of vector that are close to G1 and G2 and orthonormal to each other.

From G1 and G2 we can easily compute an orthogonal basis i.e the angle between the basis vectors will be exactly 90 degrees - that will be rotated approximately 45 degrees clockwise with respect to the basis formed by G1 and G2. This basis is the one formed by c=G1+G2 and d = c x p = (G1+G2) x (G1 x G2). If the vectors that form this new basis (c,d) are made unit vectors and rotated 45 degrees counterclockwise, we will have an orthogonal basis which is pretty close to our original basis (G1, G2). If we normalize this rotated basis, we will finally get the pair of vectors we were looking for.



Here R1 and R2 are our original R1 and R2 but they may not be perpendicular. So we assume them to be equal to G1 and G2 and compute the R1' and R2' which are perpendicular and close to R1 and R2.

We finally compute the projection matrix as P=[R1,R2,R3,t]

Now we form an array of 8 points which has all the points of the cube. We multiply this array with the projection matrix to convert all the points into the camera frame. Then we normalize it form a red circle at all the points using cv2.circle
Then using a for loop and the cv2.line we join the points such that it forms a cube and display it.