Signal Image and Video Project Report

Nardi Davide, Perantoni Giovanni, Zilio Nicola

February 14, 2023

Chapter 1

Project Idea

1.1 Problem Formulation

This project was born from our passion for robotics that has accompanied us in the last year.We wanted to broaden our horizons including some virtual reality inside our environment. The aim of this project is to have a virtualized environment resembling the real one, with the only difference in the real world there is a dummy object that will be remapped in a totally different object when spawned in the simulated world.

In the simulated world it will be possible, using an Oculus Quest 2 VR headset, to interact with the simulated robotic arm, which actions will then be replicated by the real robot. The objective then is to be able to build a tool that will allow to remote control the robotic arm and to have the possibility of creating an operating scenario in which we can insert some objects at our discretion based on the task we want to simulate.

1.2 Project development

In order to develop this project we had to reconstruct our real world environment into the Unity framework, that will then be used by the VR headset. This was a challenging part of the project, since that in order to have the simulated environment as similar as possible to the real one, we needed to implement the Differential Kinematics of our 6 Degrees Of Freedom manipulator and to give the possibility of controlling both the movement of the robot and the gripping procedure. After the simulated environment was completed we shifted our attention towards the recognition of the object. We then started to work on different approaches, both starting from what we learned from the course but also looking for possible approaches on the web. After each of the approaches we were trying to apply seemed to work

on the images we were using for the development we tried to integrate it with the ZED camera to perform the test in our work environment. In this part of the testing we wanted our approaches to be as consistent as possible in order to reduce the precision of the position of our simulated object

Chapter 2

Approaches

2.1 Recognizing the shape of the block

As a first approach we chose as our dummy block three megablocks. Then what we aimed to do was to isolate the block inside the image, and from the isolated part aiming at recognizing the orientation and position of the block.

In order to achieve that goal we decided to apply several masks and filters to our image.

The first thing we applied to the image was a mask that allowed us only to extract the red parts of the image.

```
1   img_hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
2  lower_red = np.array([0,50,50])
3  upper_red = np.array([10,255,255])
4  mask0 = cv2.inRange(img_hsv, lower_red, upper_red)
5  lower_red = np.array([170,50,50])
6  upper_red = np.array([180,255,255])
7  mask1 = cv2.inRange(img_hsv, lower_red, upper_red)
8  mask = mask0 + mask1
9  output_img = image.copy()
10 output_img[np.where(mask==0)] = 0
11 output_hsv = img_hsv.copy()
12 output_hsv[np.where(mask==0)] = 0
13 out_rgb = cv2.cvtColor(output_hsv, cv2.COLOR_HSV2BGR)
```

We then apply a GaussianBlur and a Canny filter in order to isolate the shape of the block. Then the contours are extracted from the shape since our next goal is to apply a HoughLines detection in order to find the continous lines inside our shape.

```
1  blurred=cv2.GaussianBlur(image_gray,(5,5),0)
2  sigma = np.std(blurred)
3  mean = np.mean(blurred)
```

```
4  lower = int(max(0, (mean - sigma)))
5  upper = int(min(255, (mean + sigma)))
6  canny = cv2.Canny(blurred, lower, 180)
7  canny_cp=canny
8  kernel = np.ones((5,5),np.uint8)
9  canny_cp = cv2.erode(canny_cp,kernel,iterations = 2)
10 contours, _ = cv2.findContours(canny, cv2.RETR_LIST, cv2.
       CHAIN_APPROX_NONE)
11 areas = [cv2.contourArea(c) for c in contours]
12 max_index = np.argmax(areas)
13 cnt = contours[max_index]
14 epsilon = 0.01 * cv2.arcLength(cnt, True)
15 approx = cv2.approxPolyDP(cnt, epsilon, closed=True)
16 cv2.drawContours(canny_cp,[approx],-1, (255,255,0), thickness=2)
17  linesP = cv2.HoughLinesP(canny_cp, rho=1, theta=np.pi/180,
        threshold=80, minLineLength=70, maxLineGap=1)
```

After the application of the HoughLines detection we select the highest points corresponding to the lines which describe the two external edges. We then applied some geometric constraints in order to find the three points that allows us to find the rotation of the block.

Since our block's base has a shape of a square, we can, given the two points, draw a line that conjuncts them and rotate it by 45° and by -45° with the rotation centers applied in one point for the first rotation and in the other for the second. The intersection of these two lines allows us to find the third point. From the three points we can find the rotation of our block.

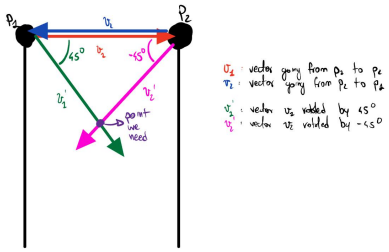


Figure 2.1: Explanation of the above

After a few tries we realized that this approach had limitations. In fact, since we need a real time detection and our work environment has some problem with the light, we decided to take a different approach.

Figure 2.2: Output after mask application

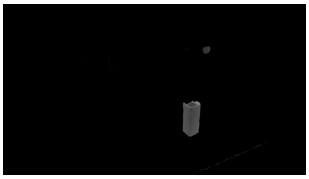


Figure 2.3: Output after Gaussian Blur

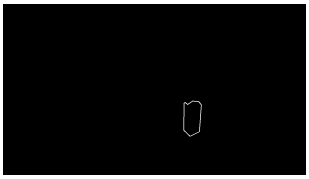


Figure 2.4: Output after Canny Edge Detection

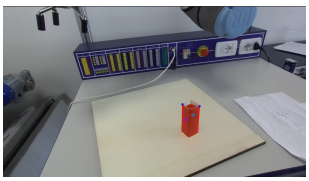


Figure 2.5: Output of Hough Line Detection

2.2 Chessboard Detection

As a second try we decided to exploit the chessboards used for calibrating cameras in order to recognize the orientation of our dummy block. In fact the OpenCV function *cv.findChessboardCorners* is able to detect the position of internal corners of the chessboard.

Using this corners we exploited the *cv.solvePnP* function to reconstruct the position and orientation frame of the chessboard. This chessboard will then be applied to our block in order to find the rotation. In the end applying the *cv.projectPoints* to project the points we found into our image, this will be useful to us for applying the *draw* function to draw the axis frame.

```
1  row = 3
2  col = 4
3  ret, corners = cv.findChessboardCorners(gray, (row,col), cv.
       CALIB_CB_FAST_CHECK)
4  objp = np.zeros((row*col,3), np.float32)
5  objp[:,:2] = np.mgrid[0:row,0:col].T.reshape(-1,2)
6  axis = np.float32([[3,0,0], [0,3,0], [0,0,-3]]).reshape(-1,3)
7  mtx = np.array([[fx, 0, cx], [0, fy, cy], [0, 0, 1]])
8  dist = np.empty(4)
9  if ret == True:
10   ret, rvecs, tvecs = cv.solvePnP(objp, corners, mtx, dist)
11   c1 = int(corners[0].ravel()[0])
12   c2 = int(corners[0].ravel()[1])
13   err, xyz = point_cloud.get_value(c1,c2)
14   imgpts, jac = cv.projectPoints(axis, rvecs, tvecs, mtx, dist)
15   if imgpts is None:
16    continue
17   else:
18    img = draw(img, corners, imgpts)
```

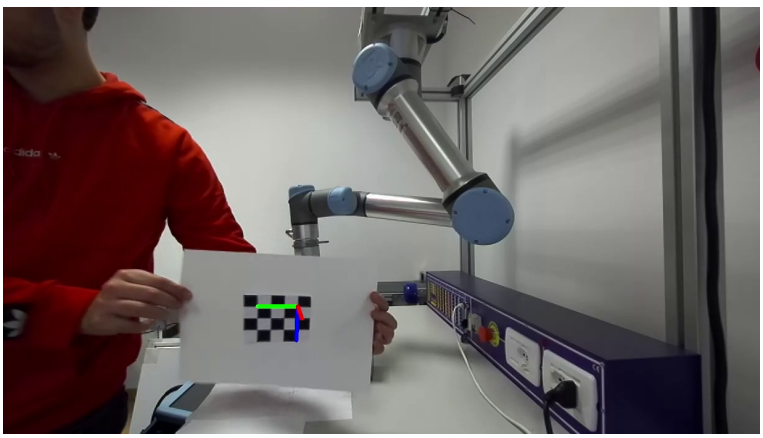


Figure 2.6: Results of the Chessboard Detection

This approach seemed to work, but the fact that we were only able to recognize just a chessboard puts some limits. In fact since we want to recognize the orientation of a randomly placed cube, having the chessboard applied on just one of the sides of the cube requires it to be always in plain sight and being all chessboard equals we cannot discriminate which face of the cube we were seeing. An other issue was coming up when the face with the chessboard assigned was in a lateral position and could not be perfectly seen the algorithm failed to recognize it.

2.3 Red Cube Detection

When looking for other possible approaches we bumped into an algorithm that is able to reconstruct a cube with a certain color. As a reference the algorithm can be found at: https://github.com/kaixindelele/OpenCV-real-world-red-cube-detection– This algorithm has a similar behavior as the first we tried to implement but it applies the goodFeaturesToTrack OpenCV function in order to find the points. The output of this algorithm is for each of the two possible faces of the cube as follows:

- the bottom point
- the upper point
- the left point
- the right point
- the center point

This is an example of the execution of this algorithm with a red cube

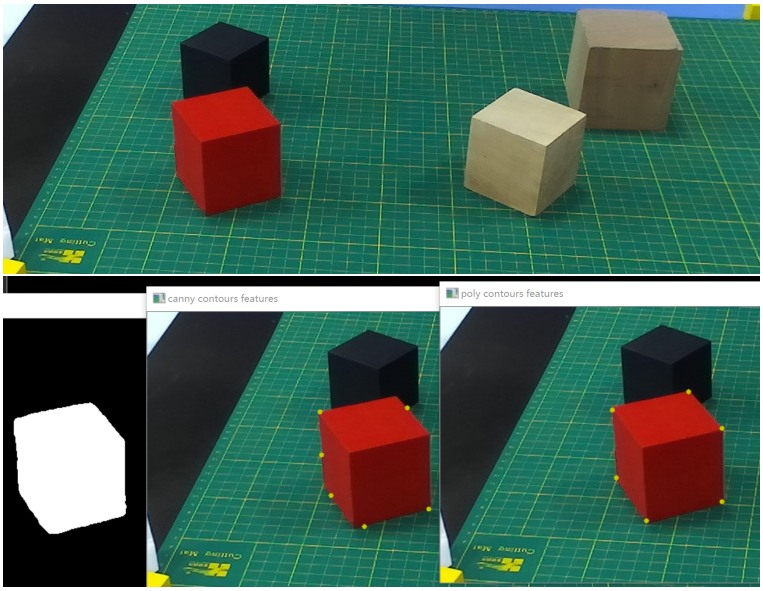


Figure 2.7: Result of the Red Cube Detection

After trying to apply this algorithm in our scenario, the same light problem as the first approach reappeared. In fact the flickering neon light combined with the highly variable intensity makes this approach not as consistent as we need.

2.4 ArUco approach

The last approach that we used was exploiting the ArUco markers. ArUco markers consists of a 6x6 binary grid, a dictionary of ArUco markers is consists of a numbers of such markers. In order to detect one of these markers inside an image what we needed to recognize it corners, correct the perspective of the image. This allows us to obtain the image of the marker as it was seen from the top. This image could now be divided into a grid in order to confront the grid with a given dictionary.

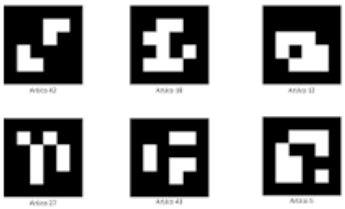


Figure 2.8: ArUco markers example

After the detection of the markers we have decided to apply the following heuristic in order to find the orientation of the cube. Since there is always at least one side of the cube in view, we will always find at least a system of reference axes. We decided then to give a priority to each of the faces of the cube, assigning them the markers with values from 0 to 5. The marker with value 0 in fact is the one that has that will be used as a reference frame in order to extract the rotation of the cube. For each of the other faces of the cube we apply a rotation to the axes in order to bring us back to the original frame.

Once we found the 2d position in which we can apply the axes corresponding to the rotation of the face of the cube we need to find where the centroid of the cube is located in order to correctly remap it into the simulated environment.

The way we proceeded in order to find out the position is to exploit some mathematical tricks. If we take the system of axes we put in the face of the cube, we can move our point by 2.5cm on the incoming axis, then applying an homogeneous transformation we are able to express the point we found in the camera reference system. This is done as follows

$$\tilde{p} = \begin{bmatrix} p \\ 1 \end{bmatrix} \qquad A_1^0 = \begin{bmatrix} R_1^0 & o_1^0 \\ 0^T & 1 \end{bmatrix} \qquad \tilde{p}^0 = A_1^0 \tilde{p}^1$$

Where p^0 is the point expressed in the frame of the camera, and it is obtained by multiplying the rototranslation matrix A_1^0 with the point p^1 that is expressed in the frame of the face of the cube. We then converted the Rodriguez vector coming from the *aruco.estimatePoseSingleMarker* into a quaternion. Once we have obtained the

center, and rotation we create a ROS message which contains both the position and the quaternion expressing the rotation along the axis. This message will be then sent directly to the VR application trough the ROS topic publisher and subscriber system.

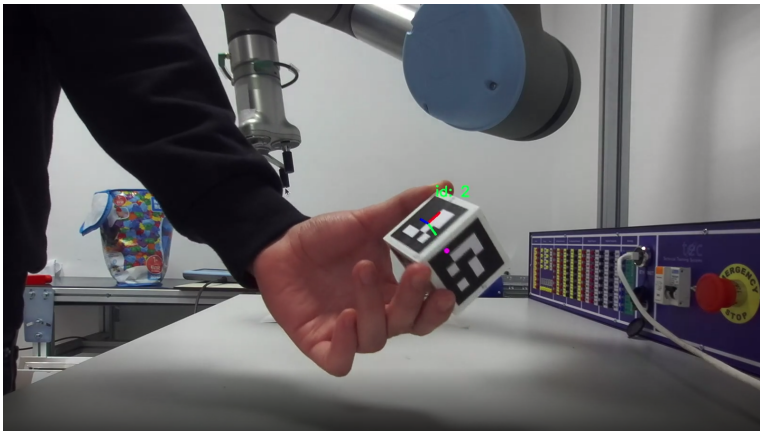


Figure 2.9: Result of the detection

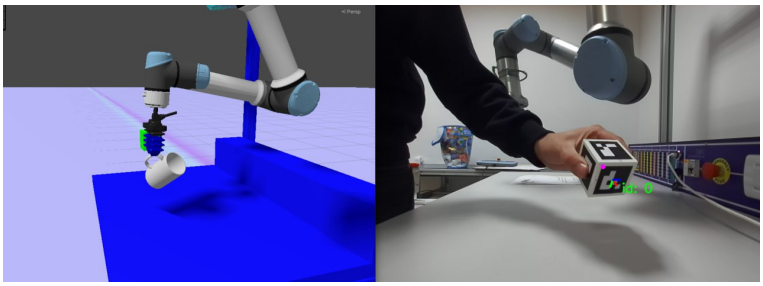


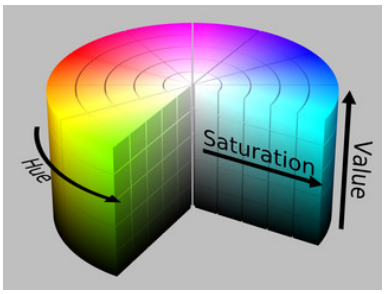
Figure 2.10: Result of applying rotation to unity

Chapter 3

Underlying theory

3.1 HSV mask

In order to only extract only the interesting part of the image we need we applied an HSV mask allowing us to isolate the red parts of the image, this will contain the part of the image containing the dummy block.



3.2 Gaussian Blur

Gaussian Blurring is the result of blurring an image by applying a Gaussian function. The result of this application is to have a blurred version of the image as applying a low-pass filter to the image. The visual effect of this blurring technique is a smooth blur resembling that of viewing the image through a translucent

3.3 Canny Edge Detection

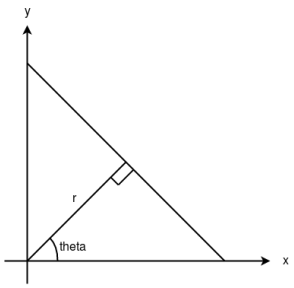
Canny edge detection is a technique used to extract useful structural information from different visions object in order to reduce the amount of data to be processed When the Canny edge detection algorithm is performed the following five steps are applied:

1. Apply a Gaussian filter in order to smooth the image and remove the noise
2. Find the intensity gradient of the image
3. Apply gradient magnitude thresholding or lower bound cut-off suppression to get rid of spurious response to edge detection
4. Apply double threshold to determine potential edges
5. Track edge by hysteresis: finalize the detection of edges by suppressing all the other edges that are weak and not connected to strong edges

3.4 Hough Detection

The Hough Transform is a method that is used in image processing to detect any shape, if that shape can be represented in mathematical form

Since a line can be represented as y = mx + c or in a parametric form as r = xcos(θ) + ysin(θ) , where r is the perpendicular distance from origin to the line and θ is the angle formed by this perpendicular line and the horizontal axis measured in counter-clockwise



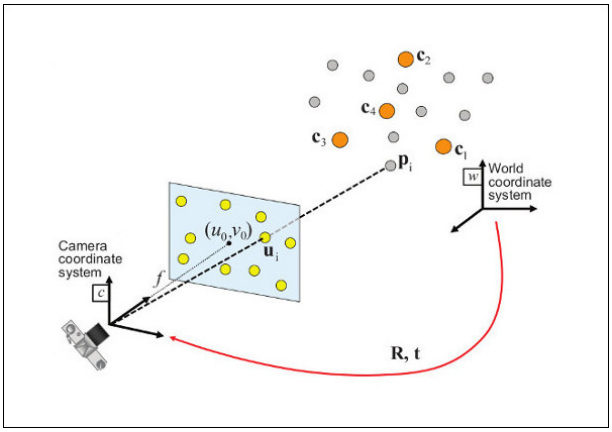
So any line can be represented in two terms as (r,θ) The OpenCV implementation of this algorithm is defines as:

cv2.HoughLines (edges,r,θ, threshold)

where edges is a binary image coming for example by a Canny edge detector, then we set the parameter values for r and θ, and then the last parameter is the one regulating the number of votes to have in order to have this line considered as a line.

3.5 SolvePnP

SolvePnP is a function used to solve the pose computation problem. This method estimates the object pose given a set of points, their corresponding image projections and the camera intrisic matrix and distortion coefficients.



Points expressed in the world frame X_w are projected into the image plane [*u,v*] using the perspective projection model Π and the camera intrinsic parameters matrix A.

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \mathbf{A}\Pi^{C}T_{W} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} r_1 1 & r_1 2 & r_1 3 & t_x \\ r_2 1 & r_2 2 & 3_2 3 & t_y \\ r_3 1 & 3_3 2 & 3_3 3 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

The estimated pose is thus the rotation (rvec) and the translation (tvec) vectors that allow transforming a 3D point expressed in the world frame into the camera frame.

$$\begin{bmatrix} X_C \\ Y_C \\ Z_C \\ 1 \end{bmatrix} = {}^{c}T_{W} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} X_C \\ Y_C \\ Z_C \\ 1 \end{bmatrix} = \begin{bmatrix} r_1 1 & r_1 2 & r_1 3 & t_x \\ r_2 1 & r_2 2 & 3_2 3 & t_y \\ r_3 1 & 3_3 2 & 3_3 3 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

3.6 Project Points

This function computes the 2D projection of the 3D points to the image plane, given intrinsic and extrinsic camera parameters. There is the possibility of computing also the Jacobian -matrices of partial derivatives of image points coordinates with respect to the particular parameters, intrinsic and/or extrinsic.

The camera intrinsic matrix A is defined as: $\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$ and the distortion coefficients are defines as: (k_1, k_2, p_1, p_2)

Chapter 4

Conclusion

We are aware that using a Neural Network could have solved our root problem. But we wanted to challenge ourselves and try to stay as close as possible to the things we have studied in the course. The results we reached made us very proud of how we managed to solve the problem despite the difficulties.

Developing this project is to lay the foundations for subsequent works embedding virtual reality and robotics in order to achieve goals in the field of medicine and teleoperations.