Introduction:

This mini project is made by three 8th-semester VGIS students. The mini-project is part of the semester course Robot Vision, where the challenge is to make a robot assemble LEGO figures automatically. The hardware which is provided is a UR5 robot and an RGB camera. Furthermore, an electronic gripper will be used to grasp the LEGO bricks, where two gripper fingers were 3D printed to make the process easier.

Planning the workflow:

Before the real work of the mini-project could begin, a plan to solve the requirements was made. The plan was made to narrow the scope of the mini-project, so it was known exactly what should be investigated beforehand.

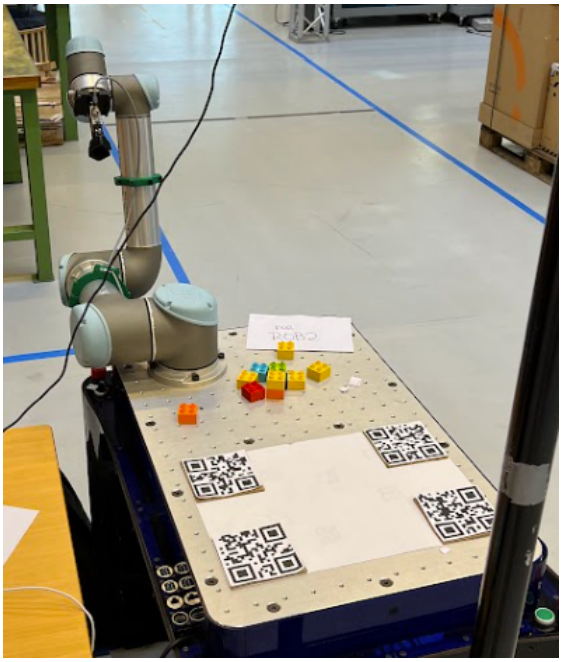
The plan included the following topics:

1. Establish a connection to the robot, making it possible to grasp and move the Lego bricks.
2. Make a computer vision algorithm that can detect each lego brick and its color.
3. Creating a transformation between the robot and the camera, making the bricks coordinates being described with respect to the robot's base frame.

With the three topics in mind, the process of making a system capable of them all could begin. The remainder of the report is therefore divided into these three topics, while the discussion will conduct a summation of these.

Establish a connection to the robot

This section will be concerned with establishing a connection to the robot, and hereafter how the robot can be moved in order to build the lego figures. The robot used in this mini project is a UR 5 and can be seen in the image below.



Two libraries have been used for establishing a connection to the robot and also moving the robot around. These two libraries are robodk.robolink and robodk.robomath and are both provided by RobotDK. A connection to the robot can be established with the following functions:

```
RDK = Robolink()
robot = RDK.Item('UR5')
state = robot.Connect()
```

If a connection is established the robot.Connect() function will return true.
Hereafter, the robot should be ready to pick and place the lego bricks. The following workflow was made for the robot algorithm:

1. Move to the initial joint space position
2. Move to joint space position for specific lego brick
3. Open gripper
4. Make linear movement in cartesian space and close the gripper to grasp the lego brick
5. Make linear movement in cartesian space upward from the picking position
6. Move to end position in joint space.
7. Make linear movement in cartesian space and open the gripper to release the lego brick
8. Make linear movement in cartesian space upward from the releasing position
9. Repeat 1-8 until all lego bricks have been picked

As can be seen in the workflow list above, multiple of the performed movements are conducted in joint space. The reason for this is to force the robot to take the same path from the pick-up area to the release area each time. Without the joint space position, the robot would very often choose a weird solution for the kinematic of the end position.

The gripper is manipulated through the RobotDK libraries described above, with the following commands:

```
def open_gripper():
    print("opening gripper")
    robot.setDO(0, 1)
    robot.setDO(1, 0)
```

```
def close_gripper():
    print("closing gripper")
    robot.setDO(0, 0)
    robot.setDO(1, 1)
```

The first argument in **robot.setDO()** defines the inputs and outputs from the robot, and since an electric gripper is attached to the robot controller, this function can manipulate it. The second argument of the function is a boolean describing if i/o should be true or false.
To make sure that the robot will not crash into a previously placed lego brick when releasing a new lego brick in the releasing zone, a counter is created. This counter keeps track of how many lego bricks have been picked and placed already. By knowing the height of a Lego brick, the counter is multiplied by that height and the releasing height can therefore be found.

For the robot to build a figure, it needs to know the sequence of bricks for each figure and it needs to know what bricks are available to be picked up and where they are placed. Each individual figure have been represented as a list of colours in the sequence which the figure should be build. Additionally, these figure lists have been collected into new list, resulting in a list of lists.

The positions and orientations of bricks found by the vision system, is represented as a dictionary, where the keys are the colors and the values are lists of x, y positions and angles.

A function have been made which is responsible for controlling the sequence of which the figures should be build and which bricks should be used to do so. This function takes the figure lists and the brick position dicitonaray as input.

Pick and place

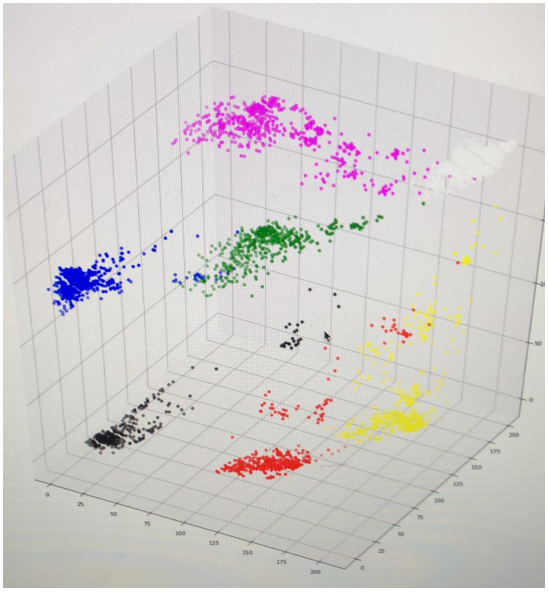
To pick and place the lego bricks, a new gripper for the UR5 was made. In the image below on the left, it can be seen that the gripper has different indention layers and a rounded outer corner. This is done to help guide the lego brick into position so that the angle does not need to be perfect for it to pick it up. Additionally, the gripper is designed to pick up the bricks in the corners. This eliminates unknowns in where the brick could be in the gripper, as if it were to pick it up on the sides it could be different each time.

Detect each lego brick and its color

In this section, the goal is to be able to detect the bricks and which color they have. The initial idea to do so was to perform simple color thresholding. However, due to changing lighting conditions and similar colors, this showed to be inconsistent. The next idea was to use the HSV channel from the colors. This showed to be a bit more consistent, but again due to the changing lighting conditions, the values would have to be adjusted at times. The next idea was to gather RGB values for all of the pixels containing the colors and set up a k-NN algorithm. This was shown to be able to find the bricks more consistently than the other mentioned methods. To be able to do these methods a Logitech C922 webcam was used and can be seen in the following Figure.

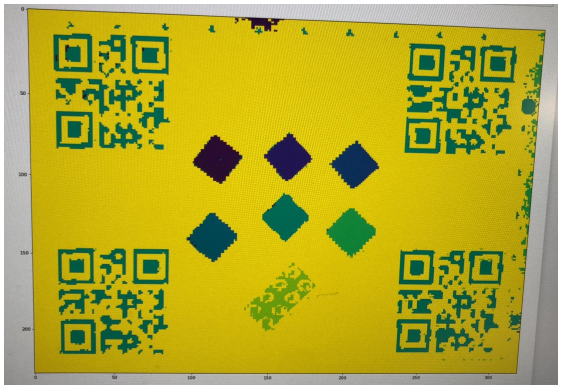
Creating a threshold for the bricks using k-NN

The first step to being able to make the k-NN is gathering data. The data was gathered by slicing images containing the bricks and then storing all of the RGB values for each pixel containing the brick. The values from this can then be visualized in 3D using matplotlib.



As can be seen in the Figure above, the colors are mostly grouped with their respective colors and seem to be separable. The RGB information is stored in a sav file type as a model. The values used on the model are gathered from the webcam frame. All of the pixels from the start frame are then classified accordingly using the stored model and classified pixels are stored as masks. The results from this can be seen in the Figure

below on the left where each color except white has been found pretty well and the white color is seen to have some issues, this is thought to be because of the white background. Therefore moving on the white brick has been removed along with the black brick, as that one also caused issues at times.



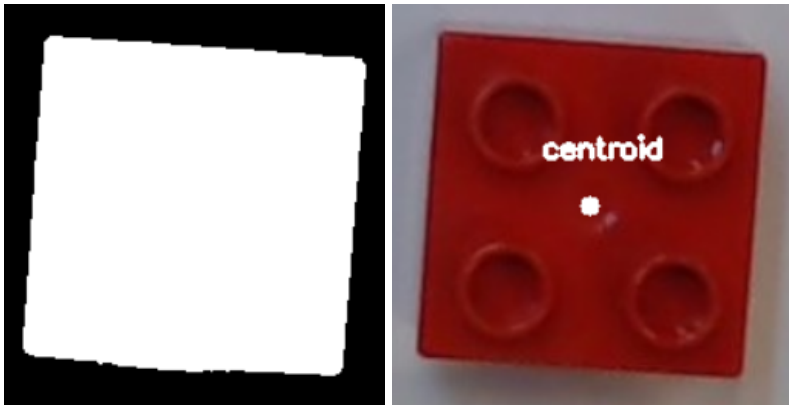

Getting the center point and rotation of the brick

After getting the results from the k-NN algorithm, it is possible to determine the center points by using the image moments on the BLOBs. The image moment is a weighted average of image pixel intensities. These allow for finding specific properties of an image and among these is the centroid of a BLOB. The equation to calculate the centroid is given by:

$C_x = \frac{M_{10}}{M_{00}}$ and $C_y = \frac{M_{01}}{M_{00}}$,

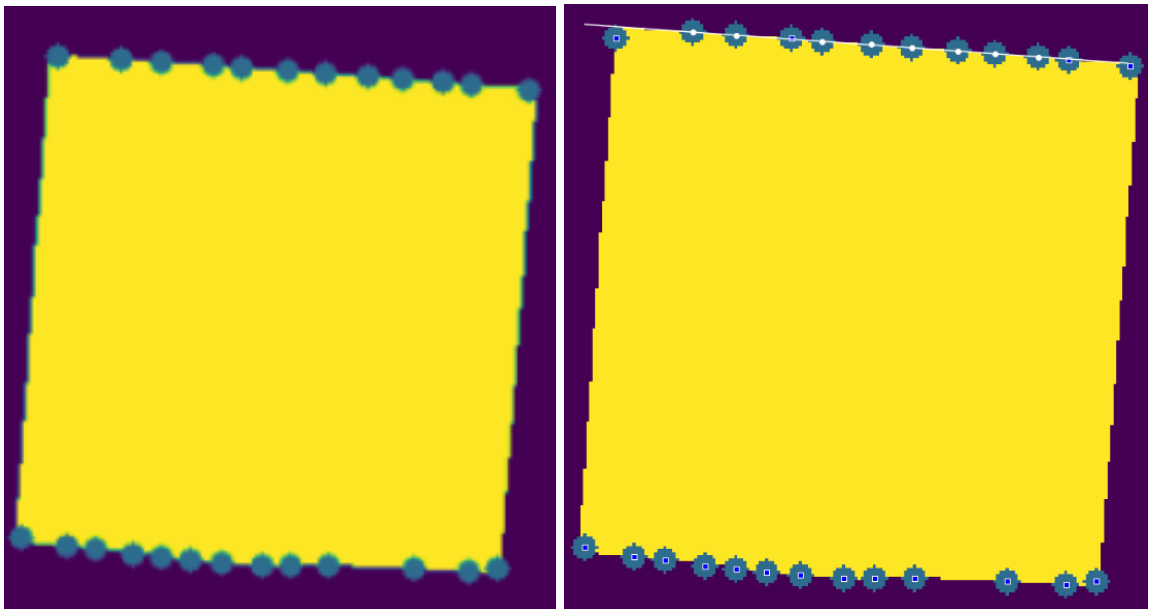
where C_x and C_y are the x and y coordinates of the centroid and M is the Moment.

The results can be seen in the following Figure.



With the center of the brick, the robot can use that information to have a more accurate estimate of where the brick is in its workspace. However, for the centroid coordinates to be functional, the camera needs to be positioned straight above the workspace, as otherwise the center of the BLOB would not be representable for the real world center.

To be able to pick up the brick with the gripper in the same way each time it is necessary to find the rotation of the brick. A method to find the rotation is by first finding edges on the blobs and then using Random sample consensus (RANSAC). From the openCV function goodFeaturesToTrack the following edges are found on the outer horizontal part of the blob.



The edge points are stored and used for the RANSAC algorithm. The RANSAC algorithm is a method that can estimate the parameters of a mathematical model from a given set of data with inliers and outliers. In this case, the line with the most amount of inliers is wanted. This will give a line, from which the slope can be found. The slope of the line can be used to calculate the angle of inclination for a line. The angle will always be between 0° and 180°. The angle is measured counterclockwise from the part of the x-axis to the right of the line. The equation to calculate the angle is given by:

$$\theta = Tan^{-1}(slope)$$

The angle of inclination for a vertical line is 90°, so this knowledge can be used to determine how much to rotate the gripper so it has the same rotation as the brick. In cases where the calculation of the angle might be off, the gripper design can adjust for it.

Creating a transformation between the camera and robot

This section will be concerned with creating a transformation between the camera frame and the robot coordinate system. It is essential to create such a transformation since it will otherwise not be possible for the robot to grasp the lego bricks. To create this transformation the camera described earlier is used to detect four QR-codes placed on a flat surface. The QR codes can be seen in the Figure below:

Each of the four QR codes in the Figure above has unique texts stored within it. This is done for the camera to distinguish between them. The method utilized to determine the transformation between the camera and robot is called Homography.

Homography

Homography is a method very often used to estimate a transformation between two planes. The method needs at least four known points in each frame to work, hence the four QR codes are used. The homography matrix is a 3x3 matrix as shown below:

$$H = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix}$$

The matrix can be used to perform translation, rotation, scaling, shear, and projection, and it is, therefore, possible to transform points from one plane to another by multiplying the matrix with a specific point.
The homography matrix is found by using the four QR codes' center coordinates and comparing these with the corresponding pixel coordinates in the image frame. The QR coordinates are found by measuring the x and y positions relative to the robot's base frame. This can be done by simply manually moving the end effector to be perpendicular to the center of each QR code and reading the x and y values from the teach-pendant.
The x and y pixel coordinates are found by utilizing a python library called PyZbar. This library makes it straightforward to find QR codes in a camera frame. Furthermore, the PyZbar can be used to find the x and y corner and the width and height of each QR code. The library can also read the information stored in each QR code. The center coordinate is found by using the coordinate information from each QR code in the following way:

$$center = (x + (width/2), y + (height/2))$$

Since the center pixel coordinates are now known, and the real-world coordinates with respect to the robot's base frame are also known, a homography matrix can be found. To calculate the homography the library OpenCV is utilized. This library has made many complex computer vision algorithms relatively easy to use. Opencv also features a function for homography estimation. The implementation of this function can be seen below:

```
h, status = cv2.findHomography(np.asarray(pts_src), np.asarray(pts_dst))
```

As can be seen in the code snippet above, the OpenCV homography function is relatively simple to implement. The pts_src array contains the pixel coordinates of the QR code center in the camera frame, whereas the pts_dst contains the real-world coordinates with respect to the robot base frame. The cv2.findHomography function returns h, which is the homography matrix, found between the eight points.
It is very important that the order of the four positions in each array corresponds to each other. If this is not the case the homography matrix will not return the correct transformation. By utilizing QR-codes the text encrypted inside can be used to make sure the points are correctly stored in the array. The text inside each QR code is encrypted to be ‘QR_1’, ‘QR_2’, and so on. In the following code snippet, it can be seen how the two arrays are made.

```python
def findHomography(QRCodes):
    pts_src = []
    pts_dst = [[-47.6, -442.57], [-173.2, -583.77], [-400.2, -365], [-267, -229.75]]

    print(QRCodes.name)
    print(QRCodes.center)

    k = {"qr_1": 0, "qr_2": 1, "qr_3": 2, "qr_4": 3}

    for name in QRCodes.name:
        pts_src.append(QRCodes.center[k[name]])

    print(pts_src)
    h, status = cv2.findHomography(np.asarray(pts_src), np.asarray(pts_dst))
    #print(status, h)
    return h, status
```

With the homography matrix, any pixel coordinate inside the camera frame can be transformed into the robot's base frame.

Discussion

As can be seen from the results some more work on the detection of bricks could have been done as only five out of the seven wanted colors were found consistently. A factor that affected this process was the lighting condition, however, this would have been a similar issue for the other methods of detecting the bricks mentioned earlier in the report. In this mini-project, the lighting was not fixed, meaning that the RGB values would vary a little each time. This problem could be fixed by having no external light and making the internal light static. Another improvement could be made to the area in which the bricks can be in. With more time this could have been changed, as the bricks should be able to be anywhere within the camera frame as long as the QR codes are not obstructed. With the trajectory planning from the robot, it would sometimes choose trajectories that would not make it possible for the robot to pick and place the legos correctly. One way that this problem could be solved is to place specific waypoints, which the end-effector would have to pass through to get to the releasing area. By having the waypoints, the robot controller might calculate the same kinematic solutions each time moving from the picking position to the release position, and by doing the same weird behavior could be eliminated. Furthermore, the RoboDK features the ability to place collision objects around the robot. This functionality could also have been used to make the robot avoid finding kinematic solutions which would make the robot move through specific areas. By placing collision objects in the unwanted areas, the robot controller would calculate paths avoiding these areas.

Conclusion

Although improvements can still be made to the system designed during this mini-project, it was managed to build a system capable of assembling Lego bricks in different combinations of color. This was done using a camera system to identify the colors, positions, and orientations of the Lego bricks before stacking them on top of each other.