UNIVERSITÀ DI TRENTO

February 2024

Fundamentals of Robotics

Final Report

UR5 Motion and Block Recognition

Marco Morandin
228160

Marco Soldera
226651

Introduction **2**
Assignment 2
Github Repository 2
High-Level Organization **3**
Spawn of the Legos 4
Motion **5**
Kinematics 5
Movement 5
Euler's Singularities 6
Motion Planning 6
Vision **8**
Dataset Creation 8
Training 8
Vision Execution 10
Area Detection 10
Lego Detection 10
Vision issues 11
Conclusions **12**

Introduction

For the course “Fundamentals of Robotics” it has been asked to develop a project in groups. The aim of this report is to summarize the work behind the project about the motion of the UR5 robot based on lego detection. The students Marco Morandin and Marco Soldera make up our group. We decided to divide the tasks into sections related to robot vision and motion. Every component of the group developed all the tasks related to a certain part.

Assignment

A number of objects (legos), one for each class, are stored without any specific order on the initial stand located within the workspace of a robotic manipulator. The only requirement of the blocks is that the base of the object is “naturally” in contact with the ground. The manipulator is an anthropomorphic arm, with a spherical wrist and a two-fingered gripper as end-effector. The objects belong to different classes but have a known geometry (coded in the STL files). The objective of the project is to use the manipulator to pick the objects in sequence and to position them on the final stand in a specific position based on their silhouettes. A camera is used to locate the different objects and to detect their position in the initial stand.

Github Repository

The project is uploaded in the following public GitHub repository: LINK GITHUB.
The instructions for installation and usage are written in the README.md file, but in case of any problem please contact marco.soldera@studenti.unitn.it or marco.morandin@studenti.unitn.it. The first part is based on the installation of “locosim” and is very similar to it. The needed files from “locosim” are already included in our repository, so aren’t needed and can cause conflicts during Repository Setup.

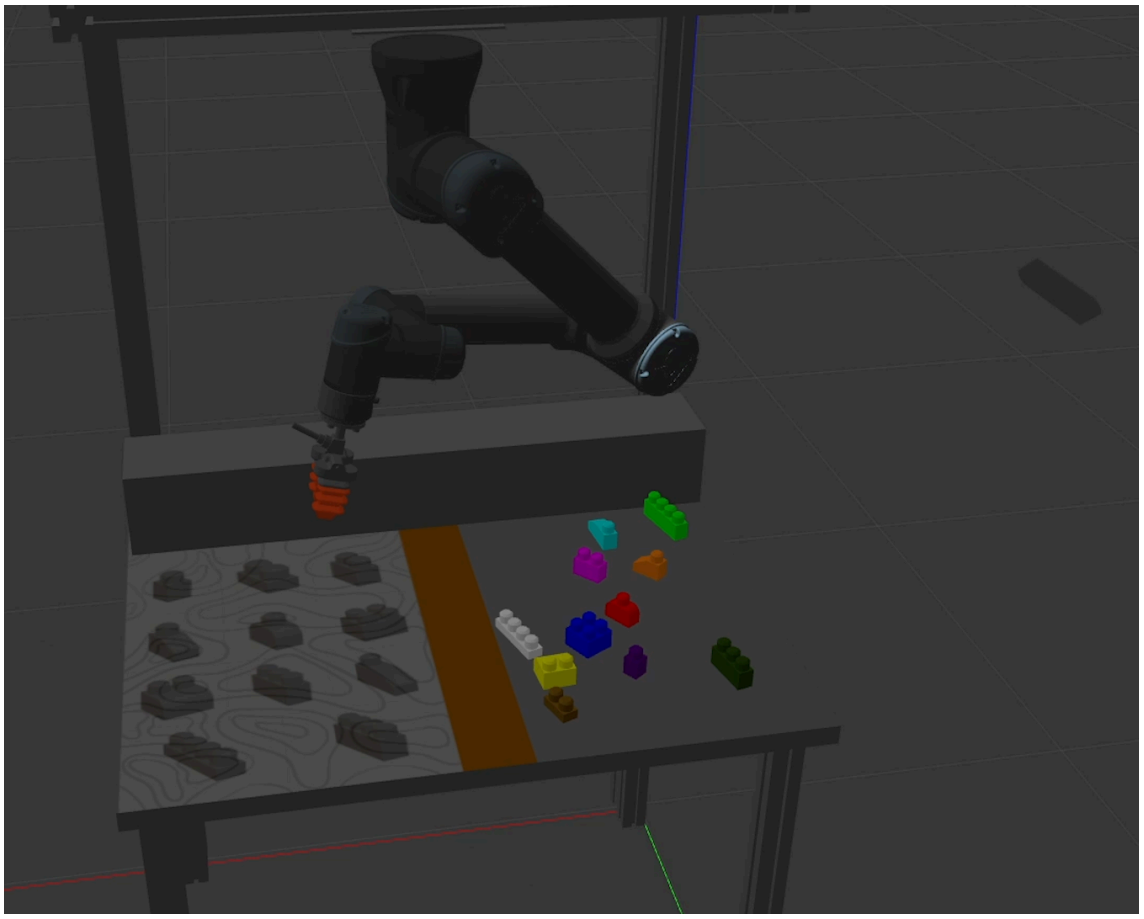
High-Level Organization

The entire project is divided in vision module and planner/motion module. The first one gets images from ZED camera to find positions of the blocks. The second one gets positions from vision module with the topic “/lego_position”, plans the trajectory and actuates the movement strategy with quaternions that we have chosen for our project. The positions published by vision are comprehensive of cartesian coordinates with respect to the base frame of the world and rotation in quaternion notation.
Since also the UR5 robot uses a topic based communication we have used a publisher to send positions through the “/ur5/joint_group_pos_controller/command” topic. For the inverse process the angles of the joints are read from the topic “/ur5/joint_states”. The precise strategy for picking and placing the blocks will be described later in the dedicated section.
The system has the following workflow:

1. Launch the robot controller, Gazebo and RVIZ through the script ur5_generic.py
2. Run the spawnLego.py script to generate all the blocks after the robot moved to home position
3. Start the planner from planner_pkg; it will wait for some positions from the vision module
4. Execute the vision.py script; it gets the image captured from the ZED camera and starts recognizing the legos. After that, it publishes the found positions to the specific topic
5. Now the movement part receives a list of legos with their respective class, position and rotation in quaternion notation
6. The movement part computes the direct kinematics to know where the robot is from the angles of the joints
7. The path planning finds the best trajectory to move the robot from the actual position to catch the lego and sends to the UR5 all the intermediate joint positions.
8. The robot moves to the position of the lego and picks it up
9. To know where to place the lego, the movement part computes the new path from the current position to a specified position in the final stand related to that type of lego
10. After the placement of every block in the final stand the script restarts from point 6
11. When all the perceived pieces are placed in the final stand the placement module waits for the vision to send other blocks to pick up

Spawn of the Legos

The legos are spawned with random positions in the initial stand and rotations between 0 and $\frac{\pi}{4}$ around the Z axis paying attention to collisions between bricks and keeping them at a minimum distance. To spawn them, the script uses the service "/gazebo/spawn_sdf_model" where the types and their respective positions are sent. Every lego has his own color different from the others that is set by the script. First of all every piece found on the table is removed through the service "/gazebo/delete_model" to start then with the placement procedure. Every time the script makes more than 500 tries to place the pieces without collisions unsuccessfully, it removes everything and restarts from zero. The program ends when 11 pieces (one per class) are placed.



Motion

The motion part can be divided into three main tasks: the kinematics of the UR5 robot, the movement task that is used to convert the desired trajectory from the operational space to the joint space to trace a rectilinear movement and the motion planning task that computes the trajectory of the end-effector in the operational space to avoid obstacles and critical points. This whole module is like a “translator” between the vision that tells where to move and the UR5 robot that wants to know how to go there.

Kinematics

In this task, that is coded in the files “kinematics.h” and “kinematics.cpp”, we defined all the functions related to the physical characteristics of the robot. We have the transformation matrix for every link and the function that computes the direct kinematics by multiplying these matrices. For the UR5 the transformation matrix is found as follows, in a different way from the standard procedure with the listed DH parameters.

$$T_{i-1}^{i}(\theta_i) = \begin{bmatrix} \cos(\theta_i) & -\sin(\theta_i) & 0 & a_{i-1} \\ \sin(\theta_i)\cos(\alpha_{i-1}) & \cos(\theta_i)\cos(\alpha_{i-1}) & -\sin(\alpha_{i-1}) & -d_i\sin(\alpha_{i-1}) \\ \sin(\theta_i)\sin(\alpha_{i-1}) & \cos(\theta_i)\sin(\alpha_{i-1}) & \cos(\alpha_{i-1}) & d_i\cos(\alpha_{i-1}) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

i	α_{i-1}	a_{i-1}	d_i	θ_i
1	0	0	d_1	θ_1
2	$\alpha_1 = 90°$	0	0	θ_2
3	0	a_2	0	θ_3
4	0	a_3	d_4	θ_4
5	$\alpha_4 = 90°$	0	d_5	θ_5
6	$\alpha_5 = -90°$	0	d_6	θ_6

Then we have the functions for inverse kinematics, the computation of the Jacobian matrix and the conversion functions from Euler angles to rotation matrix and vice versa.

Movement

To plan the trajectory from the current position to a desired one we use quaternions and differential kinematics. The first function called “invDiffKinematicControlSimCompleteQuaternion” gets the desired position and rotation. It divides the path with the given time step and computes joint positions for every tick to accomplish the movement. This approach is based on the computation of the velocity of the end-effector to have a straight trajectory from start to finish point. Then we can pass all this data to the function “invDiffKinematicControlCompleteQuaternion” that computes the coefficient (velocity of the links) to find the new joint configuration starting from the last one with the

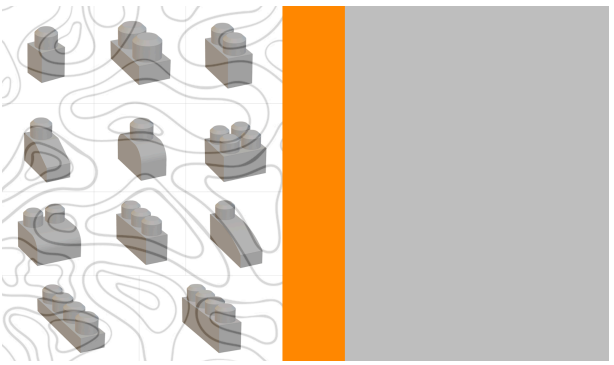
usage of the Jacobian matrix. We have chosen the quaternion strategy to avoid singularities and plan the trajectory in operational space. During the movement we have to be sure that the angles don't go over the limit, especially the final joint of the gripper. To avoid this every time the angle exceeds $\pm \pi$, it is decreased to a lower but equivalent angle. Sometimes the implemented movement can result not really linear. This happens because the sequence of joint configurations is sent directly to the robot without leaving some time between them. This resulted in the best strategy for us because we lost a little in precision, but gained a lot in speed of the movements. All the functions have been translated from their respective MatLab versions.

Euler's Singularities

As mentioned before all the singularities are avoided since quaternions and differential kinematics are used for movement. Every linear path of the end-effector is planned in operational space and converted into joint space to avoid collisions with obstacles.

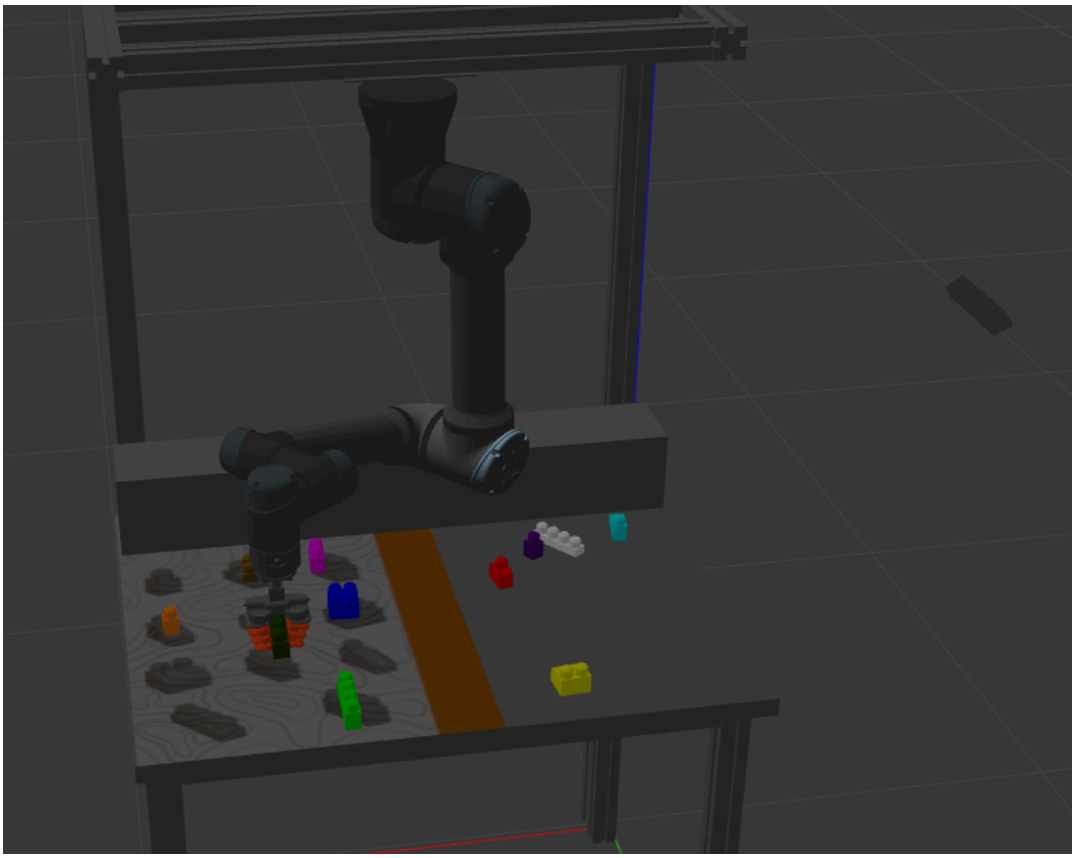
Motion Planning

When the vector containing all the legos and their positions is received by the planner, the trajectory strategy is actuated. First of all, for every lego, we take the robot to a homing position at the X coordinate of 0.00 with respect to the robot frame in front of it. If we start from a position behind the robot we need to turn around it to avoid collisions with the base of the UR5. From this home position then we move over the block that we want to reach. Then we can go down to the brick and grab it. Now we compute another trajectory from the current position to turn around the base joint and go over the placing position that can be composed of multiple segments. The robot goes down and releases the piece in his location on the final stand. At this point we return back to the home position to restart this process for another lego. As shown below the final stand (on the left) has all the marked positions based on the silhouettes of the pieces; the initial stand is on the right side. On the final stand there is a camouflage texture to avoid the vision module to recognise those legos.



Before starting the computation of the next joint configuration for another movement the program waits for the desired position to be reached by the end-effector. When

all the tasks in the queue are satisfied, the module waits for others from the vision module.



Vision

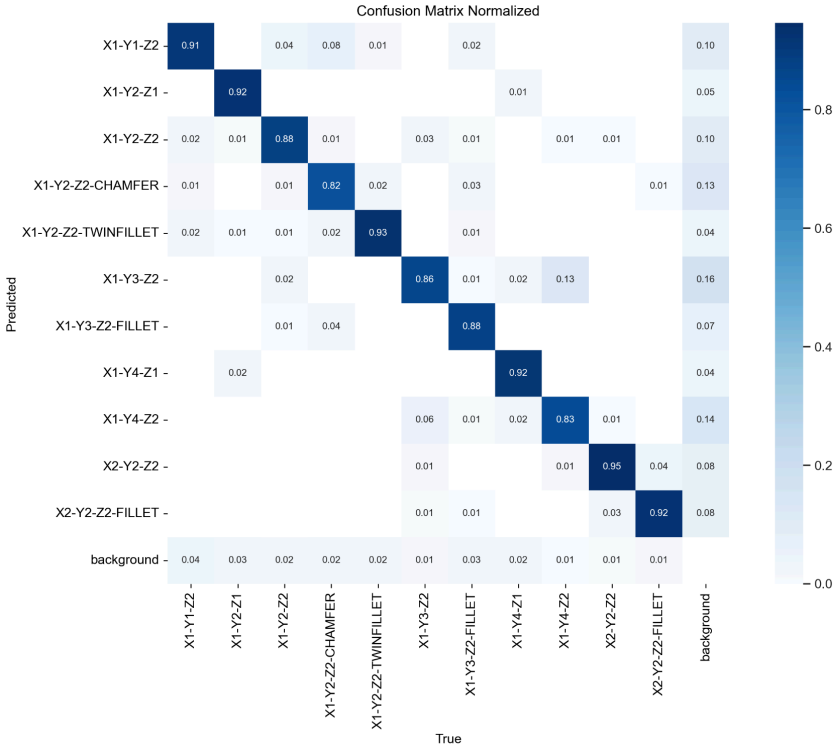
The vision part deals with the detection of the legos from an image obtained from the ZED camera. When the script receives the image it shrinks the photo to the table to avoid confusion on recognizing the legos. After that, it tries to recognize the legos in the scene and computes its position in the Gazebo world with the help of the PointCloud. In the end it publishes a list of legos and their position in the topic “/lego_position”.

Dataset Creation

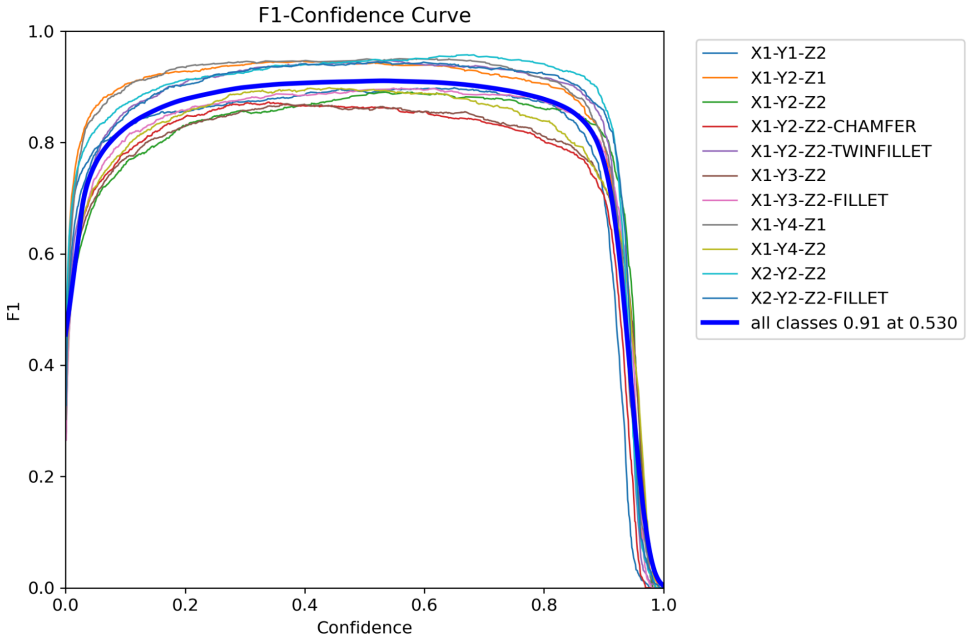
The training of the Yolov8 model requires a dataset in the Yolo format that is composed of: a data.yaml file in which there is the path to labels and images and a list of all the possible classes. Moreover it needs a labels text file with the same name of the image in which there are (for each line) a class id and the coordinates of the bounding box.
To convert the given dataset to a Yolo dataset we created a script that reads all the json files in the assignment folder, extracts useful information and organizes them into the Yolo format structure.

Training

The training of the dataset is done with the k-fold cross validation technique that splits the dataset into k folds; some of them for the train and some of them for the evaluation, looping through the folds, to avoid overtraining. This happens because the training dataset changes on each iteration and this is especially useful when one deals with a small dataset like our situation. The training phase is done with 20 epochs over each fold and the model from where we start is the YoloV8m; we noticed that it is the most complex model that our hardware can manage in reasonable time.
The advantage of choosing YoloV8m instead of the smallest one is that this model has more hidden layers that make it more accurate in detection. The disadvantage is that the model is slower to train and the final result is heavier; another disadvantage is that it has a higher latency, but in our case this isn’t relevant.
It follows the normalized confusion matrix that represents the accuracy of the detection during training in the validation phase; from it we can observe that the training has obtained good results because the highest values (in dark blue) are in the diagonal of the matrix.



Moreover the F1-curve graph, reported below, shows that the model is more confident in its predictions when the confidence score is high. However, the models are also more likely to make mistakes when the confidence score is high but not recall. A good model has high confidence and high recall. The graph shows that the models are more confident in their predictions when the confidence score is high. This model is 91% confident when its predictions are right and it correctly identifies 53% of the objects in the dataset (recall).



Vision Execution

The vision starts by waiting for the image from the ZED camera that is saved; after that the script executes the function pointCloudCallBack(). That function runs the detection of the blocks, finds all the points (in pixels) from the top left vertex to the bottom right vertex of each bounding box and counts the number of points inside a bounding box. Moreover it gets the point from the PointCloud and converts it from the camera frame to the world frame. In the end it finds the coordinates of the ends of the blocks and thanks to those points it computes the coordinates of centers of blocks and their orientation in Euler angles. To do so it finds the coordinates of the smallest (p1) and the biggest (p2) point of the blocks along Y axis and the biggest (p3) point along X axis of the blocks (respectively to the ZED camera); thanks to them it can find the coordinates of the center with the formula shown below.

$$C = (\frac{x_1 + x_2}{2}, \frac{y_1 + y_2}{2})$$

The orientation of the blocks in Euler angles can be found with the following formula.

$$\alpha = atan2((y_2 - y_3), (x_2 - x_3))$$

Once it computes them, it sends in the “/lego_position” topic to be used by the movement part.
Last detail: all the variables used in the vision part are stored in the params.py file to easily change it.

Area Detection

The detection of the area happens when the vision script receives an image from the ZED camera. To detect the area in which the blocks would be recognized there is a dedicated class with a method that creates a black mask that covers everything except the initial stand and saves it.

Block Detection

BlockDetection is the class that initializes the YoloV8 model and has a method that detects the lego in the image captured by the ZED camera. In the end it prints the results like below.

```
0: 576x1024 1 X1-Y1-Z2, 1 X1-Y2-Z1, 1 X1-Y2-Z2, 1 X1-Y2-Z2-CHAMFER, 1 X1-Y2-Z2-TWINFILLET, 1 X1-Y3-Z2-FILLET, 1 X1-Y4-Z1, 2 X2-Y2-Z2-FILLETs, 1972.2ms
Speed: 11.9ms preprocess, 1972.2ms inference, 2.2ms postprocess per image at shape (1, 3, 576, 1024)
Results saved to runs/detect/predict15
Found 9 objects!

   idx             Category  Category ID  Confidence                        Bounding Box
0    0             X1-Y2-Z2            2    0.930843  [1154.08, 631.60, 1204.33, 698.67]
1    1      X1-Y3-Z2-FILLET            6    0.915624  [1032.22, 585.74, 1104.25, 638.72]
2    2  X1-Y2-Z2-TWINFILLET            4    0.911876    [920.52, 566.26, 965.09, 609.50]
3    3             X1-Y1-Z2            0    0.894410  [1172.26, 703.74, 1215.12, 767.22]
4    4     X1-Y2-Z2-CHAMFER            3    0.849351  [1006.35, 692.17, 1065.90, 763.35]
5    5      X2-Y2-Z2-FILLET           10    0.790911    [904.28, 667.87, 973.39, 745.88]
6    6             X1-Y4-Z1            7    0.400663  [1234.31, 569.42, 1323.09, 634.72]
7    7      X2-Y2-Z2-FILLET           10    0.394425  [1265.92, 639.77, 1350.50, 723.86]
8    8             X1-Y2-Z1            1    0.332453    [771.65, 751.37, 841.25, 817.44]

Category: X1-Y2-Z2
Position: ['0.28', '0.37']
Rotation:      0.08 yaw
```

Vision issues

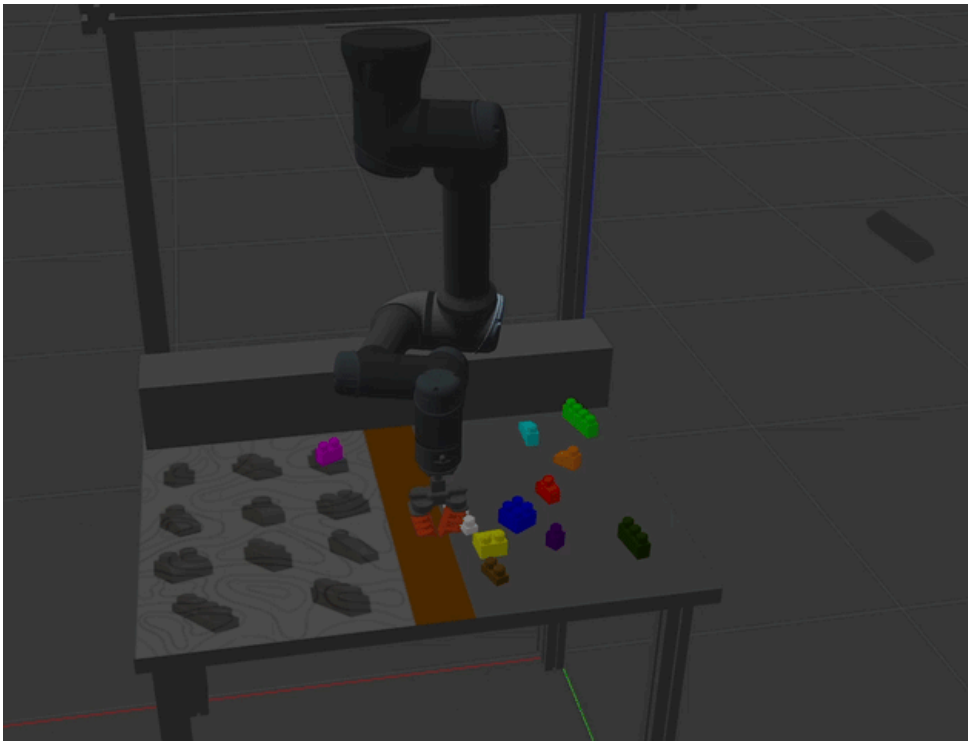
Due to the limited dimension of the dataset the trained model hasn't very high performance so sometimes it fails to recognize some blocks or does not detect them at all. To solve this problem we decided to scale down blocks so there is more space between them to prevent them from covering each other. one can re-run the vision module that sends new data to the movement part. To improve the performance of the detection it is necessary to increase the dimension of the labeled dataset or add more viewpoints and compare the results.

Conclusions

In the end we can say that our project achieved the requirements of the assignment. The vision module can recognise with a pretty good precision the spawned block and if it is not really sure about something with a low confidence, it has a second try after the first block of movements has ended. The main problem of the motion module that can leave some legos in the initial stand is the gripper that sometimes can't grab blocks in the correct way. A possible future challenge can be to combine precision and a faster movement sending joint configurations only when the previous one has been reached. This requires finding the optimal compromise between transmission frequency and the maximum time of every movement.

Team tasks:

- Morandin Marco: vision module, organization of the dataset and training of the model, scaling the blocks
- Soldera Marco: planner module, movement and kinematics, spawning of the blocks



This is a little demo of the movement of the robot. This video is speeded up by 500% with respect to reality. If not visible the gif video can be found here.