Brat robotic pentru Pick-and-Place,

care detecteaza piese lego prin Yolov5

Content

No table of contents entries found.

1. Introducere

ROS (robot operating system), ROS, sau Sistemul de Operare pentru Roboți, este un cadru middleware open-source proiectat pentru dezvoltarea și controlul roboților. Despreși numele său, ROS nu este un sistem de operare în sine; în schimb, furnizează servicii și instrumente pentru abstractizarea hardware-ului, comunicarea între componente, gestionarea pachetelor și altele. ROS este larg utilizat în comunitatea robotică datorită flexibilității și scalabilității sale, facilitând dezvoltarea și partajarea software-ului între diverse platforme robotice.

Caracteristicile cheie ale ROS includ:

1. **Noduri:** Sistemele ROS sunt compuse din noduri, care sunt procese independente care îndeplinesc sarcini specifice. Nodurile comunică între ele prin transmiterea de mesaje.
2. **Subiecte (Topics):** Nodurile comunică prin subiecte, care sunt canale numite prin care se transmit mesaje. Un nod poate publica mesaje pe un subiect sau se poate abona pentru a primi mesaje de pe un subiect.
3. **Mesaje:** Mesajele sunt unitățile de date schimbate între noduri. Ele pot conține informații despre datele senzoriale, comenzi sau orice altceva necesar în cadrul sistemului robotic.

Roboți Pick and Place: Roboții Pick and Place sunt roboți industriali specializați în preluarea și plasarea obiectelor de la un loc la altul. Acești roboți sunt utilizați în aplicații variate, precum ambalarea, asamblarea, manipularea obiectelor în liniile de producție sau depozitare.

Principalele caracteristici ale sistemelor Pick and Place includ:

1. **Prehensiune:** Roboții Pick and Place sunt echipați cu sisteme de prehensiune sau ventuze care le permit să ridice și să țină diverse tipuri de obiecte.
2. **Viziune:** Unii roboți Pick and Place sunt prevăzuți cu sisteme de viziune pentru a identifica și localiza obiectele în spațiu. Acest aspect este crucial pentru a asigura o preluare și plasare precisă.
3. **Programare flexibilă:** Acești roboți sunt programabili pentru a manipula diferite tipuri de obiecte și pentru a efectua sarcini variate. Programarea lor este adaptabilă la schimbările de producție sau la introducerea de noi produse.
4. **Precizie și viteză:** Roboții Pick and Place sunt proiectați pentru a combina precizia în manipulare cu o viteză eficientă, permițând un flux de lucru optim în cadrul proceselor industriale.

Utilizarea sistemelor ROS poate facilita dezvoltarea și controlul roboților Pick and Place, oferind un mediu de programare robust și instrumente pentru gestionarea comunicațiilor și a datelor într-un mod eficient.

Pentru ca robotul sa poate indeplini functia de Pick and place tinand cont de pozitia pieselor, este nevoie de urmatoarele lucruri:

---

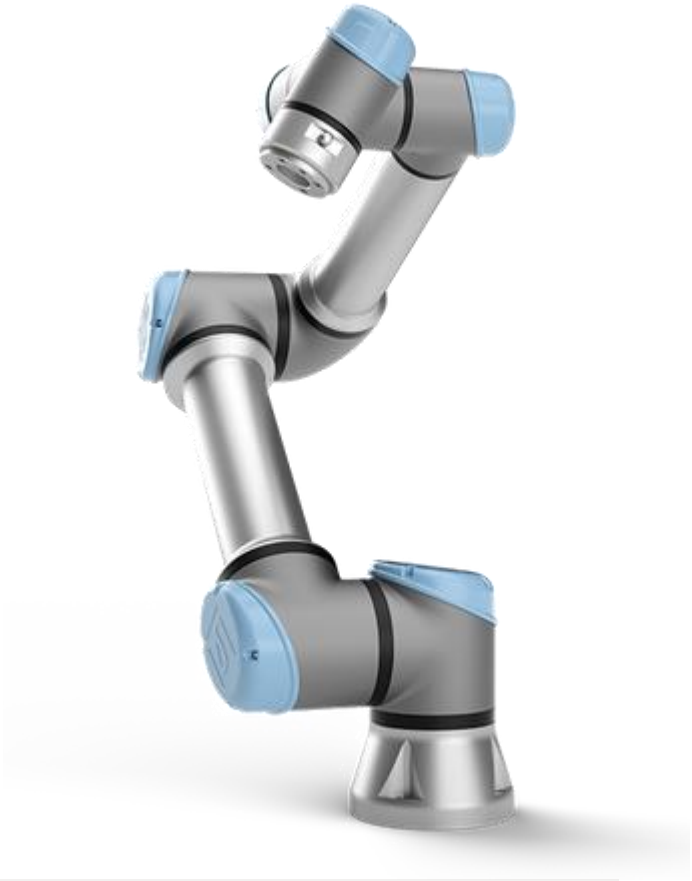
1) O structura robotica cu un gripper operational
2) Un controller al acestui robot, inclusive controller al gripperului
3) O modalitate de a detecta piesele de pe masa
4) Un control intre sistemul de viziune sic el de miscare pentru a se indeplini taskul de pick-and-place

---

I Structura proiect

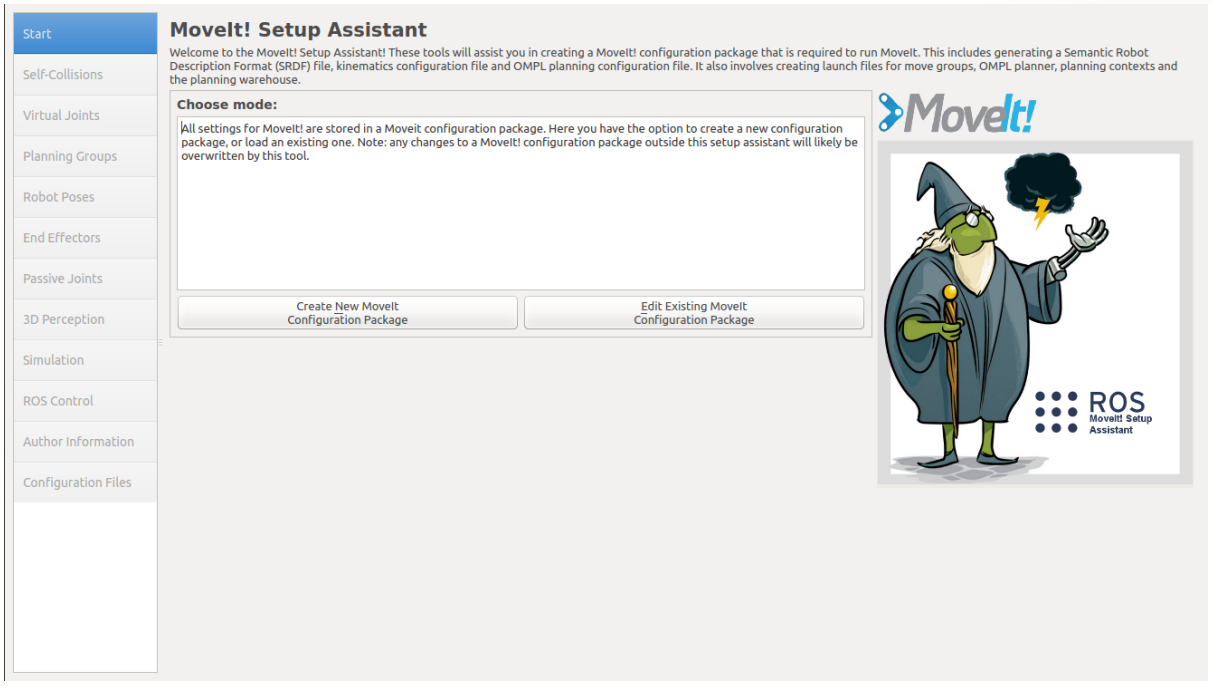
1. Structura robotica

Ca structura robotica am folosit robotul Universal Robot 5 cu 6 grade de libertate.

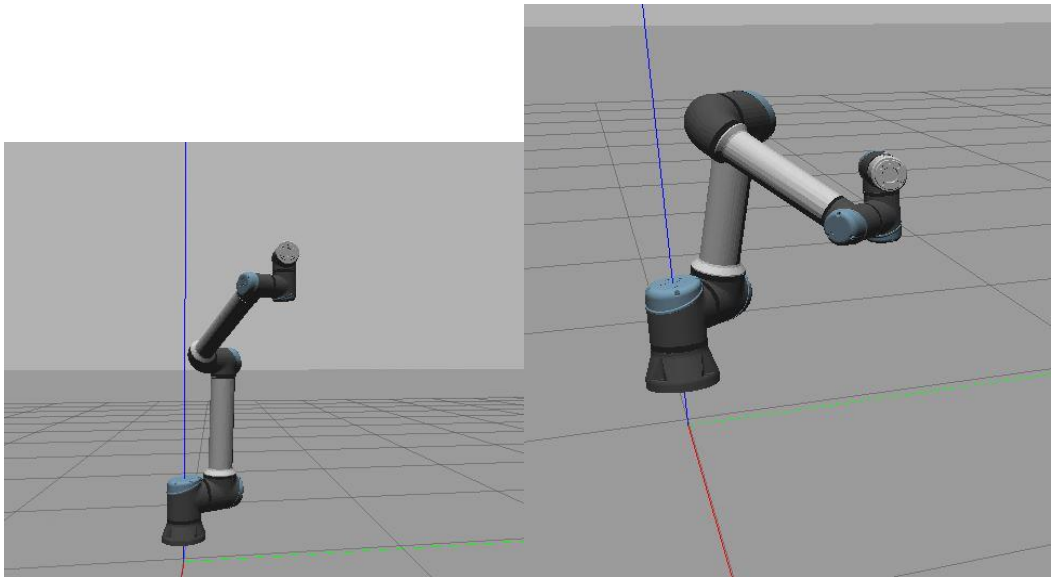


Structura a fost luata de pe https://github.com/ros-industrial/universal_robot

Pentru a construi in ROS configuratia robotului (legatura jointuri, declarare jointure, limite, pose-uri) a fost nevoie de pachetul Moveit.

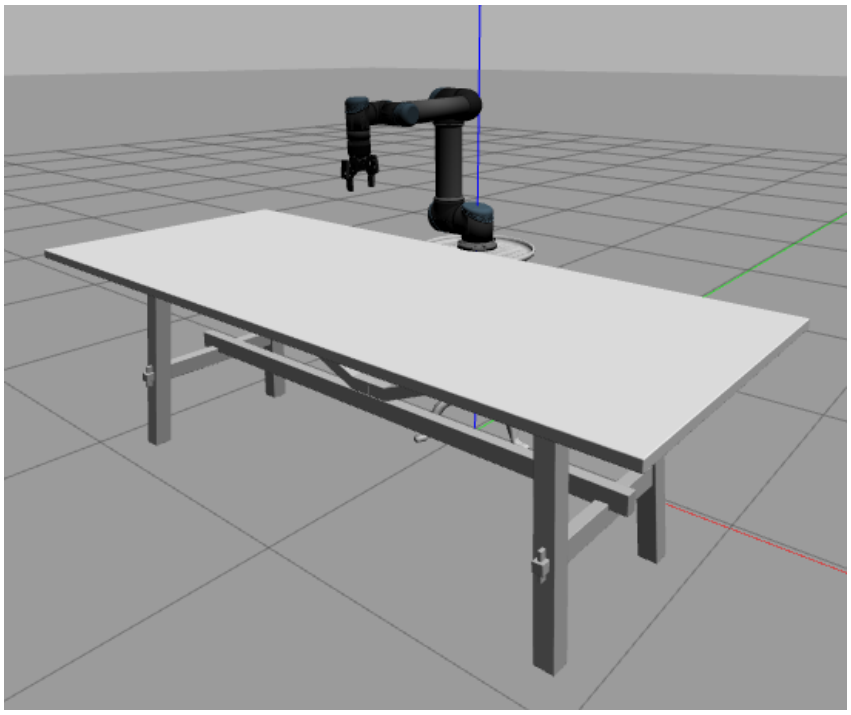


Dupa ce am configurat structura robotica, si robotul se poate misca



Am cautat un controller potrivit pentru acest robot, am gasit cinematica si motion planning-ul pentru Universal Robot 5

Dupa care, am folosit o scena create crobotul UR5, unde acesta se afla langa o masa pe care ulterior vom pune piesele lego si camera.



Pentru a lega elementele intre ele si masa de ground am folosit gazebo ros link attacher.

https://github.com/pal-robotics/gazebo_ros_link_attacher

2.1 Cinematica inversa:

```
def inverse(desired_pos):  # T60
    th = np.zeros((6, 8), dtype=np.float64)
    P_05 = desired_pos @ [0, 0, -d[5], 1] - [0, 0, 0, 1]

    P_05 = np.asarray(P_05).flatten()

    # **** theta1 ****

    psi = atan2(P_05[2 - 1], P_05[1 - 1])
    phi = acos(d[3] / sqrt(P_05[2 - 1] * P_05[2 - 1] + P_05[1 - 1] * P_05[1 - 1]))

    # The two solutions for theta1 correspond to the shoulder
    # being either left or right

    th[0, 0:4] = pi / 2 + psi + phi
    th[0, 4:8] = pi / 2 + psi - phi
    th = th.real

    # **** theta5 ****

    cl = [0, 4]  # wrist up or down
    for c in cl:
        T_10 = linalg.inv(AH(1, th[:, c]))
        T_16 = T_10 @ desired_pos
        th[4, c:c + 2] = +acos((T_16[2, 3] - d[3]) / d[5])
        th[4, c + 2:c + 4] = -acos((T_16[2, 3] - d[3]) / d[5])

    th = th.real

    # **** theta6 ****
    # theta6 is not well-defined when sin(theta5) = 0 or when T16(1,3), T16(2,3) = 0.

    cl = [0, 2, 4, 6]
    for c in cl:
        T_10 = linalg.inv(AH(1, th[:, c]))
        T_16 = linalg.inv(T_10 @ desired_pos)
        th[5, c:c + 2] = atan2(-T_16[1, 2] / sin(th[4, c]), T_16[0, 2] / sin(th[4, c]))

    th = th.real
```

2.2 Controlul robotului :

Printre functiile importante din acest script avem:

```
def get_controller_state(controller_topic, timeout=None):
    return rospy.wait_for_message(
        f"{controller_topic}/state",
        control_msgs.msg.JointTrajectoryControllerState,
        timeout=timeout)
```

Aceasta functie asteapta si returneaza starea(state) controllerului “controller_topic”, controller care furnizeaza starea curenta a robotului (pozitii viteze ale jointurilor).

```
class ArmController:
    def __init__(self, gripper_state=0, controller_topic="/trajectory_controller"):
    # Lista cu numele articulațiilor brațului robotic
        self.joint_names = [
            "shoulder_pan_joint",
            "shoulder_lift_joint",
            "elbow_joint",
            "wrist_1_joint",
            "wrist_2_joint",
            "wrist_3_joint",
        ]

        # Starea inițială a gripperului și topicul controllerului
        self.gripper_state = gripper_state
        print("initializat")

        print("initializat2")
        self.controller_topic = controller_topic

        # Traseul implicit al articulațiilor
        self.default_joint_trajectory = trajectory_msgs.msg.JointTrajectory()
        self.default_joint_trajectory.joint_names = self.joint_names

        # Obține starea actuală a brațului și calculează poziția și orientarea gripperului

        joint_states = get_controller_state(controller_topic).actual.positions
        x, y, z, rot = kinematics.get_pose(joint_states)
        self.gripper_pose = (x, y, z), Quaternion(matrix=rot)

        # Creează un publicator pentru comanda de mișcare a articulațiilor

        self.joints_pub = rospy.Publisher(
            f"{self.controller_topic}/command",
            trajectory_msgs.msg.JointTrajectory, queue_size=10)

    def move(self, dx=0, dy=0, dz=0, delta_quat=Quaternion(1, 0, 0, 0), blocking=True):
     #Realizează o mișcare relativă a brațului
        (sx, sy, sz), start_quat = self.gripper_pose

        tx, ty, tz = sx + dx, sy + dy, sz + dz
        target_quat = start_quat * delta_quat

        self.move_to(tx, ty, tz, target_quat, blocking=blocking)

    def move_to(self, x=None, y=None, z=None, target_quat=None, z_raise=0.0, blocking=True):
        # Mută brațul la o poziție absolută specificată
        """
```

```
    def send_joints(self, x, y, z, quat, duration=1.0):
    # Trimite comenzi pentru a muta brațul la o anumită poziție


        joint_states = kinematics.get_joints(x, y, z, quat.rotation_matrix)

        traj = copy.deepcopy(self.default_joint_trajectory)

        for _ in range(0, 2):
            pts = trajectory_msgs.msg.JointTrajectoryPoint()
            pts.positions = joint_states
            pts.velocities = [0, 0, 0, 0, 0, 0]
            pts.time_from_start = rospy.Time(duration)
            # Set the points to the trajectory
            traj.points = [pts]
            # Publish the message
            self.joints_pub.publish(traj)

        # Așteaptă ca brațul să ajungă la o anumită poziție
    def wait_for_position(self, timeout=2, tol_pos=0.01, tol_vel=0.01):
        end = rospy.Time.now() + rospy.Duration(timeout)
        while rospy.Time.now() < end:
            msg = get_controller_state(self.controller_topic, timeout=10)
            v = np.sum(np.abs(msg.actual.velocities), axis=0)
            if v < tol_vel:
                for actual, desired in zip(msg.actual.positions, msg.desired.positions):
                    if abs(actual - desired) > tol_pos:
                        break
                return
w Applications  .logwarn("Timeout waiting for position")
```

2.3 Motion-planning

1. Preluarea pozitiei piesei logo

```
def get_legos_pos(vision=False):
    #inregistrare topic detectie piesa lego
    if vision:
        legos = rospy.wait_for_message("/lego_detections", ModelStates, timeout=None)
    else:
        models = rospy.wait_for_message("/gazebo/model_states", ModelStates, timeout=None)
        legos = ModelStates()

        for name, pose in zip(models.name, models.pose):
            if "X" not in name:
                continue
            name = get_model_name(name)

            legos.name.append(name)
            legos.pose.append(pose)
    return [(lego_name, lego_pose) for lego_name, lego_pose in zip(legos.name, legos.pose)]
```

2. Configurarea traiectoriei si a gripperului pentru a lua piesa:

```
def straighten(model_pose, gazebo_model_name):
    x = model_pose.position.x
    y = model_pose.position.y
    z = model_pose.position.z
    model_quat = PyQuaternion(
        x=model_pose.orientation.x,
        y=model_pose.orientation.y,
        z=model_pose.orientation.z,
        w=model_pose.orientation.w)

    model_size = MODELS_INFO[get_model_name(gazebo_model_name)]["size"]

    """
        Calculate approach quaternion and target quaternion
    """

    facing_direction = get_axis_facing_camera(model_quat)
    approach_angle = get_approach_angle(model_quat, facing_direction)

    print(f"Lego is facing {facing_direction}")
    print(f"Angle of approaching measures {approach_angle:.2f} deg")

    # Calculate approach quat
    approach_quat = get_approach_quat(facing_direction, approach_angle)

    # Get above the object
    controller.move_to(x, y, target_quat=approach_quat)

    # Calculate target quat
    regrip_quat = DEFAULT_QUAT
    if facing_direction == (1, 0, 0) or facing_direction == (0, 1, 0):  # Side
        target_quat = DEFAULT_QUAT
        pitch_angle = -math.pi/2 + 0.2

        if abs(approach_angle) < math.pi/2:
            target_quat = target_quat * PyQuaternion(axis=(0, 0, 1), angle=math.pi/2)
        else:
            target_quat = target_quat * PyQuaternion(axis=(0, 0, 1), angle=-math.pi/2)
        target_quat = PyQuaternion(axis=(0, 1, 0), angle=pitch_angle) * target_quat

        if facing_direction == (0, 1, 0):
            regrip_quat = PyQuaternion(axis=(0, 0, 1), angle=math.pi/2) * regrip_quat

    elif facing_direction == (0, 0, -1):
        """
```

3.Prepozitionare pentru incepe miscare si apucare piesa inchizand gripperul:

```
            Pre-positioning
        """
        controller.move_to(z=z, target_quat=approach_quat)
        close_gripper(gazebo_model_name, model_size[0])

        tmp_quat = PyQuaternion(axis=(0, 0, 1), angle=2*math.pi/6) * DEFAULT_QUAT
        controller.move_to(SAFE_X, SAFE_Y, z+0.05, target_quat=tmp_quat, z_raise=0.1)
Move to safe position
        controller.move_to(z=z)
        open_gripper(gazebo_model_name)

        approach_quat = tmp_quat * PyQuaternion(axis=(1, 0, 0), angle=math.pi/2)

        target_quat = approach_quat * PyQuaternion(axis=(0, 0, 1), angle=-math.pi)  #
Yaw rotation of 180 deg

        regrip_quat = tmp_quat * PyQuaternion(axis=(0, 0, 1), angle=math.pi)
    else:
        target_quat = DEFAULT_QUAT
        target_quat = target_quat * PyQuaternion(axis=(0, 0, 1), angle=-math.pi/2)

    """
        Grip the model
    """
    if facing_direction == (0, 0, 1) or facing_direction == (0, 0, -1):
        closure = model_size[0]
        z = SURFACE_Z + model_size[2] / 2
    elif facing_direction == (1, 0, 0):
        closure = model_size[1]
        z = SURFACE_Z + model_size[0] / 2
    elif facing_direction == (0, 1, 0):
        closure = model_size[0]
        z = SURFACE_Z + model_size[1] / 2
    controller.move_to(z=z, target_quat=approach_quat)
    close_gripper(gazebo_model_name, closure)
```

3. Detectie piese lego

Pentru partea de vision, am folosit o retea Yolov5,
https://github.com/ultralytics/yolov5

, care a facut posibila detectia pieselor lego utilizand urmatorul script:

```python
def process_item(imgs, item):

    #images
    rgb, hsv, depth, img_draw = imgs
    #obtaining Yolo informations (class, coordinates, center)
    x1, y1, x2, y2, cn, cl, nm = item.values()
    mar = 15
    x1, y1 = max(mar, x1), max(mar, y1)
    x2, y2 = min(rgb.shape[1]-mar, x2), min(rgb.shape[0]-mar, y2)
    boxMin = np.array((x1-mar, y1-mar))
    x1, y1, x2, y2 = np.int0((x1, y1, x2, y2))

    boxCenter = (y2 + y1) // 2, (x2 + x1) // 2
    color = get_lego_color(boxCenter, rgb)
    hsvcolor = get_lego_color(boxCenter, hsv)
```

Unde se colecteaza informatiile din yolo legate de coordonate si centru.

Pentru detectarea orientarii piesei, pentru a configure si orientarea gripperului.

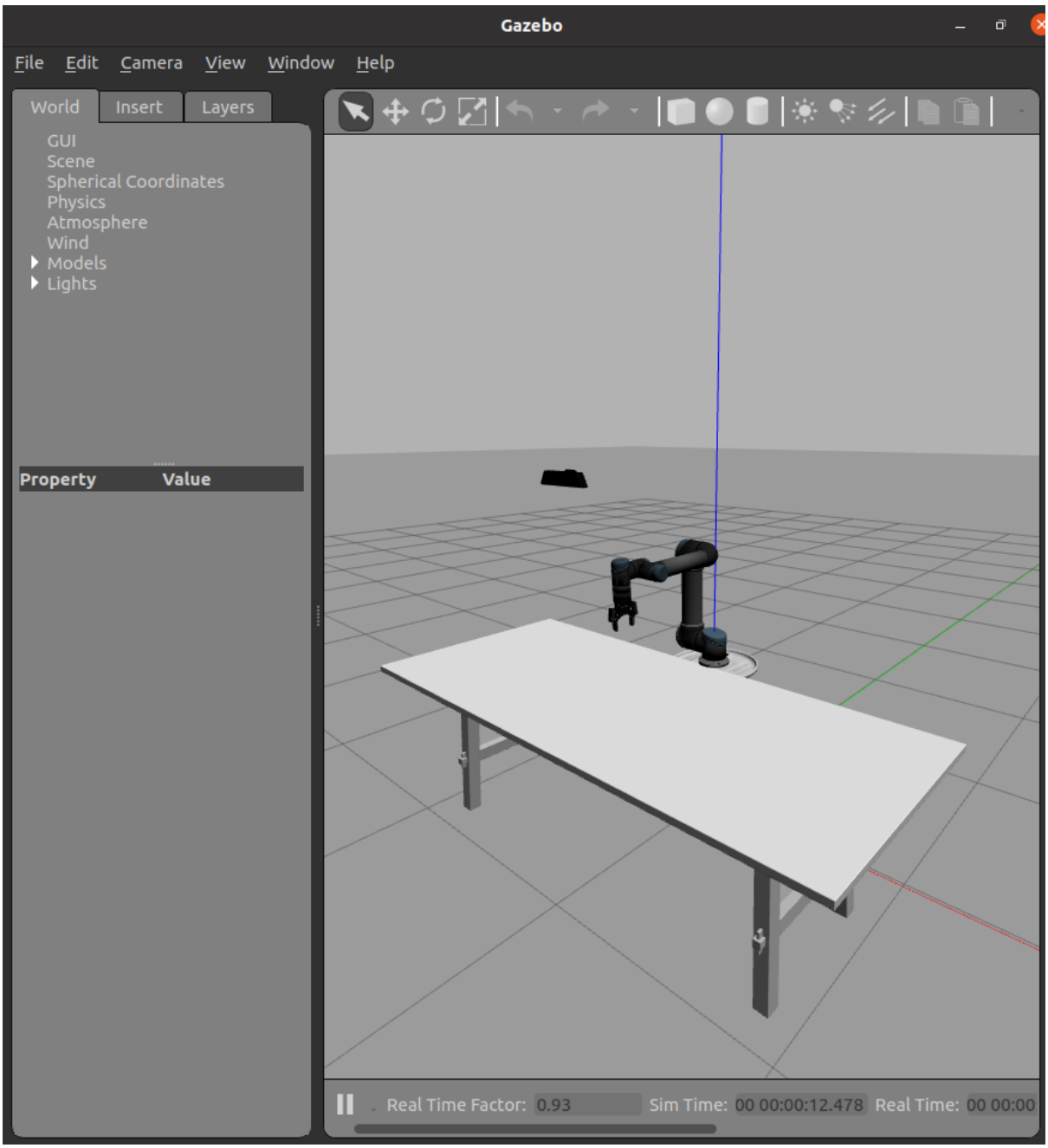
```python
# model detect orientation
depth_borded = np.zeros(depth.shape, dtype=np.float32)
depth_borded[sliceBox] = l_depth

depth_image = cv.normalize(
    depth_borded, None, alpha=0, beta=255, norm_type=cv.NORM_MINMAX, dtype=cv.CV_8U
)
depth_image = cv.cvtColor(depth_image, cv.COLOR_GRAY2RGB).astype(np.uint8)
```

II Utilizare

1. Initializare world, cu robotul si workspace-ul acestuia:

```
odin@ubuntu:~$ cd proiectros/
odin@ubuntu:~/proiectros$ source devel/setup.bash
odin@ubuntu:~/proiectros$ roslaunch levelManager lego_world.launch
```

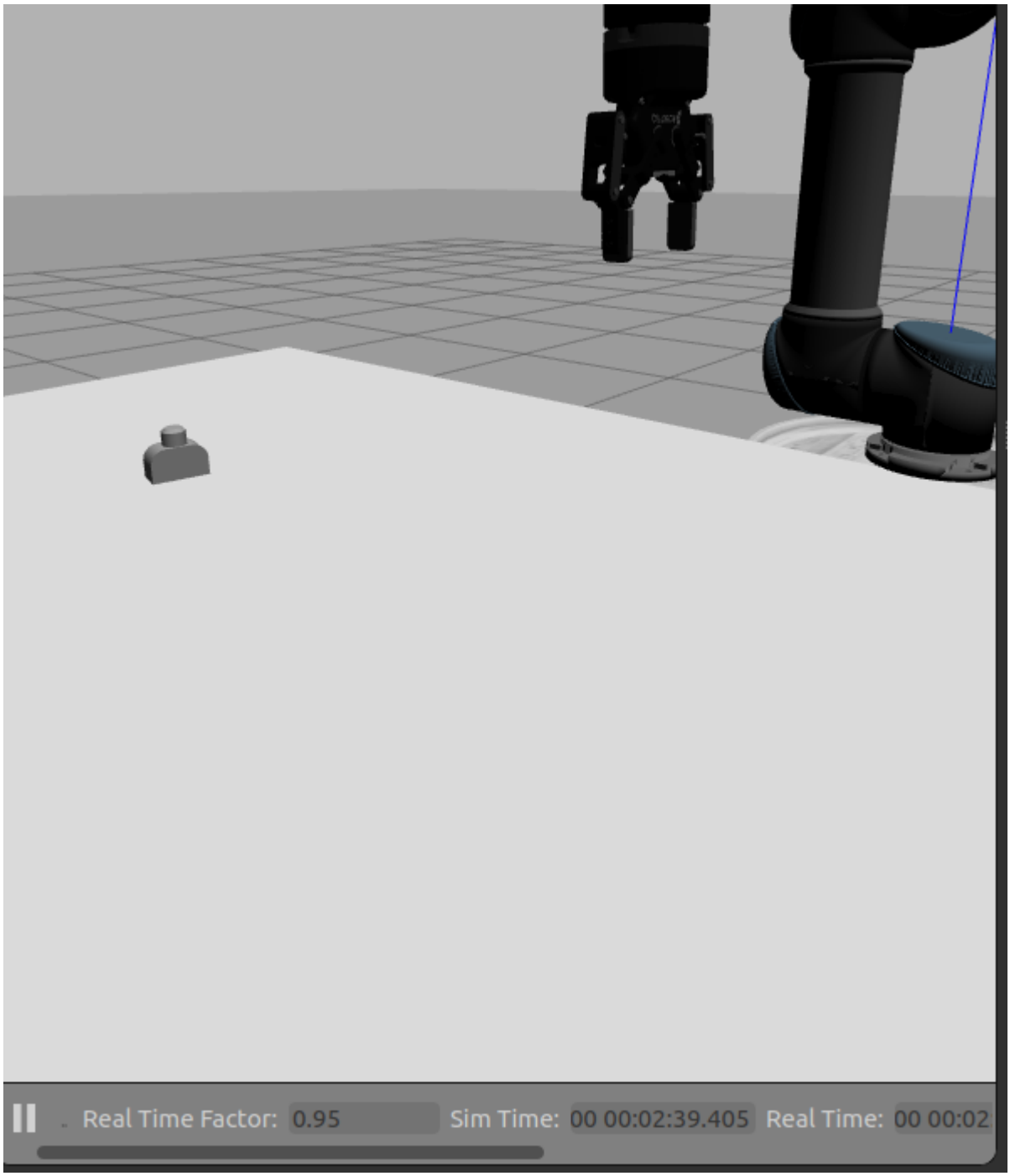
Se deschide Gazebo si se afiseaza worldul configurat:



Am introdus si un model Kinect din libraria gazebo, pentru a putea fii vizualizata camera, ea fiind in originea modelului.

Dupa care se spawneaza o piesa lego:

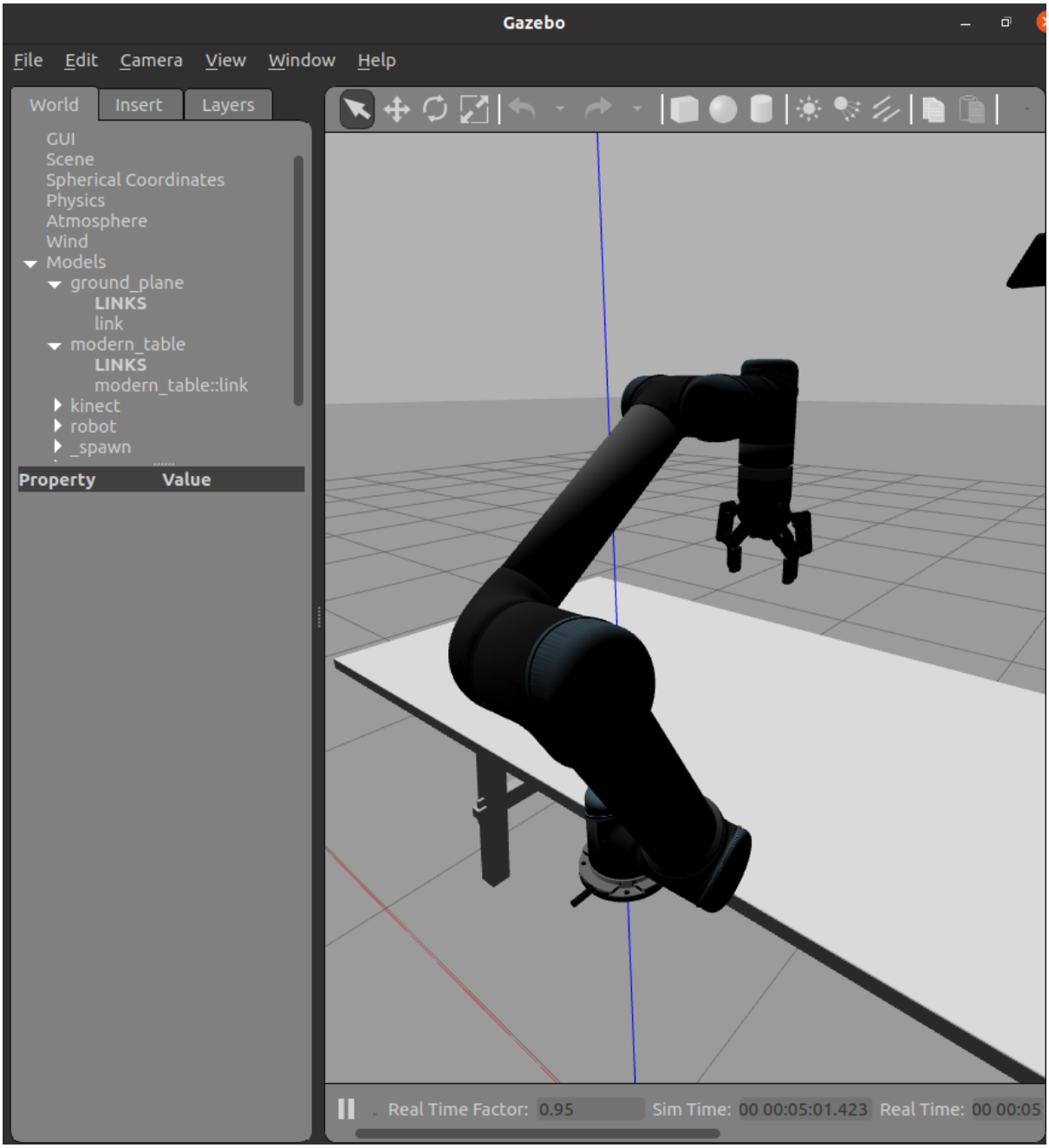
```
odin@ubuntu:~/proiectros$ rosrun levelManager levelManager.py -l 1

Added 1 bricks
All done. Ready to start.
```



3. Apelarea controllerului, prin motion planning

```
odin@ubuntu:~/proiectros$ rosrun motion_planning motion_planning.py
Initializing node of kinematics
initializat
initializat2
```

In urma acestei comenzi, robotul se duce in pozitia de homing.



Dupa aparitia mesajului din terminal :

```
Waiting for detection of the models
```

Se poate trece la detectie.

4. Apelarea scriptului de vision:

```
odin@ubuntu:~/proiectros$
odin@ubuntu:~/proiectros$ rosrun vision lego-vision.py -show
Loading model best.pt
```

Extra-optiunea -show este pentru a putea vizualiza imaginea capturata:

Se trimit coordonatele piesei la scriptul de motion planning, dupa care se executa miscarea :