EECS 4421 LAB4

Name: Mahfuz Rahman

Student No.: 217847518

My aruco_target.py file

```
import math
import numpy as np
import rclpy
from rclpy.node import Node
from rclpy.parameter import Parameter
import cv2
from cv_bridge import CvBridge, CvBridgeError
from sensor_msgs.msg import Image
from sensor_msgs.msg import CameraInfo
from packaging.version import parse
from std_msgs.msg import Bool  # Import Bool message type


if parse(cv2.__version__) >= parse('4.7.0'):
    def local_estimatePoseSingleMarkers(corners, marker_size, mtx, distortion):
        marker = np.array([[-marker_size /2, marker_size / 2, 0],
                           [marker_size /2, marker_size / 2, 0],
                           [marker_size /2, -marker_size / 2, 0],
                           [-marker_size /2, -marker_size / 2, 0]],
                           dtype = np.float32)
        trash = []
        rvecs = []
        tvecs = []
        for c in corners:
            nada, R, t = cv2.solvePnP(marker, c, mtx, distortion, False, cv2.SOLVEPNP_IPPE_SQUARE)
            rvecs.append(R)
            tvecs.append(t)
            trash.append(nada)
        return rvecs, tvecs, trash
```

```
class ArucoTarget(Node):
  _DICTS = {
    "4x4_100" : cv2.aruco.DICT_4X4_100,
    "4x4_1000" : cv2.aruco.DICT_4X4_1000,
    "4x4_250" : cv2.aruco.DICT_4X4_250,
    "4x4_50" : cv2.aruco.DICT_4X4_50,
    "5x5_100" : cv2.aruco.DICT_5X5_100,
    "5x5_1000" : cv2.aruco.DICT_5X5_1000,
    "5x5_250" : cv2.aruco.DICT_5X5_250,
    "5x5_50" : cv2.aruco.DICT_5X5_50,
    "6x6_100" : cv2.aruco.DICT_6X6_100,
    "6x6_1000" : cv2.aruco.DICT_6X6_1000,
    "6x6_250" : cv2.aruco.DICT_6X6_250,
    "6x6_50" : cv2.aruco.DICT_6X6_50,
    "7x7_100" : cv2.aruco.DICT_7X7_100,
    "7x7_1000" : cv2.aruco.DICT_7X7_1000,
    "7x7_250": cv2.aruco.DICT_7X7_250,
    "7x7_50": cv2.aruco.DICT_7X7_50,
    "apriltag_16h5" : cv2.aruco.DICT_APRILTAG_16H5,
    "apriltag_25h9" : cv2.aruco.DICT_APRILTAG_25H9,
    "apriltag_36h10" : cv2.aruco.DICT_APRILTAG_36H10,
    "apriltag_36h11" : cv2.aruco.DICT_APRILTAG_36H11,
    "aruco_original" : cv2.aruco.DICT_ARUCO_ORIGINAL
  }

  def __init__(self, tag_set="apriltag_36h10", target_width=0.20):
    super().__init__('aruco_target')
    self.get_logger().info(f'{self.get_name()} created')

    self.declare_parameter('image', "/mycamera/image_raw")
    self.declare_parameter('info', "/mycamera/camera_info")

    self._image_topic = self.get_parameter('image').get_parameter_value().string_value
    self._info_topic = self.get_parameter('info').get_parameter_value().string_value

    self.create_subscription(Image, self._image_topic, self._image_callback, 1)
    self.create_subscription(CameraInfo, self._info_topic, self._info_callback, 1)

    self._bridge = CvBridge()
```

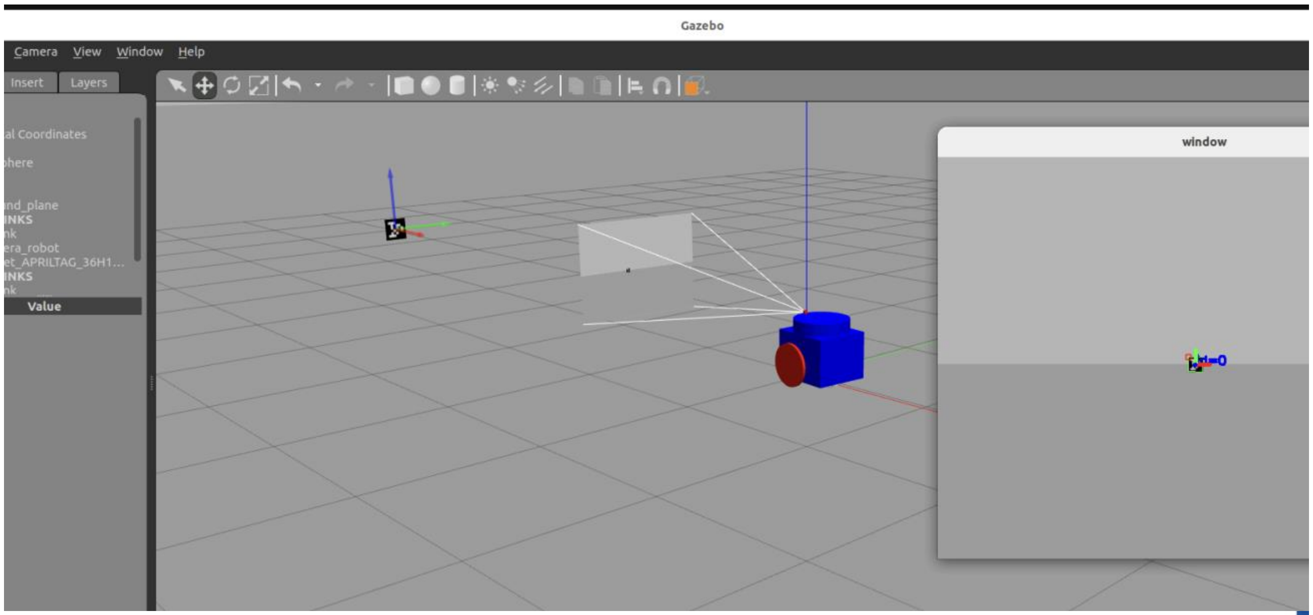
```
        dict = ArucoTarget._DICTS.get(tag_set.lower(), None)
        if dict is None:
            self.get_logger().error(f'ARUCO tag set {tag_set} not found')
        else:
            if parse(cv2.__version__) < parse('4.7.0'):
                self._aruco_dict = cv2.aruco.Dictionary_get(dict)
                self._aruco_param = cv2.aruco.DetectorParameters_create()
            else:
                self._aruco_dict = cv2.aruco.getPredefinedDictionary(dict)
                self._aruco_param = cv2.aruco.DetectorParameters()
                self._aruco_detector = cv2.aruco.ArucoDetector(self._aruco_dict, self._aruco_param)
            self._target_width = target_width
            self._image = None
            self._cameraMatrix = None
            self.get_logger().info(f"using dictionary {tag_set}")

        # Create a publisher for target visibility
        self._visibility_publisher = self.create_publisher(Bool, '/target_visible', 1)


    def _info_callback(self, msg):
        if msg.distortion_model != "plumb_bob":
            self.get_logger().error(f"We can only deal with plumb_bob distortion {msg.distortion_model}")
        self._distortion = np.reshape(msg.d, (1,5))
        self._cameraMatrix = np.reshape(msg.k, (3,3))

    def _image_callback(self, msg):
        self._image = self._bridge.imgmsg_to_cv2(msg, "bgr8")

        grey = cv2.cvtColor(self._image, cv2.COLOR_BGR2GRAY)
        if parse(cv2.__version__) < parse('4.7.0'):
            corners, ids, rejectedImgPoints = cv2.aruco.detectMarkers(grey, self._aruco_dict)
        else:
            corners, ids, rejectedImgPoints = self._aruco_detector.detectMarkers(grey)
        frame = cv2.aruco.drawDetectedMarkers(self._image, corners, ids)
        if ids is None:
            self.get_logger().info(f"No targets found!")
            self.set_target_visible(False)  # Set visibility to False
            return
        if self._cameraMatrix is None:
            self.get_logger().info(f"We have not yet received a camera_info message")
            return

        # Set visibility to True when a target is found
        self.set_target_visible(True)

        if parse(cv2.__version__) < parse('4.7.0'):
            rvec, tvec, _objPoints = cv2.aruco.estimatePoseSingleMarkers(corners, self._target_width, self._cameraMatrix,
self._distortion)
        else:
            rvec, tvec, _objPoints = local_estimatePoseSingleMarkers(corners, self._target_width, self._cameraMatrix,
self._distortion)
        result = self._image.copy()
        for r,t in zip(rvec,tvec):
            self.get_logger().info(f"Found a target at {t} rotation {r}")
            if parse(cv2.__version__) < parse('4.7.0'):
                result = cv2.aruco.drawAxis(result, self._cameraMatrix, self._distortion, r, t, self._target_width)
            else:
                result = cv2.drawFrameAxes(result, self._cameraMatrix, self._distortion, r, t, self._target_width)
        cv2.imshow('window', result)
        cv2.waitKey(3)

    def set_target_visible(self, visible):
        """ Update the visibility of the target and publish the status. """
        msg = Bool()
        msg.data = visible
        self._visibility_publisher.publish(msg)
        self.get_logger().info(f"Target visibility set to {visible}")

def main(args=None):
    rclpy.init(args=args)
    node = ArucoTarget()
    try:
        rclpy.spin(node)
        rclpy.shutdown()
    except KeyboardInterrupt:
        pass

if __name__ == '__main__':
    main()
```

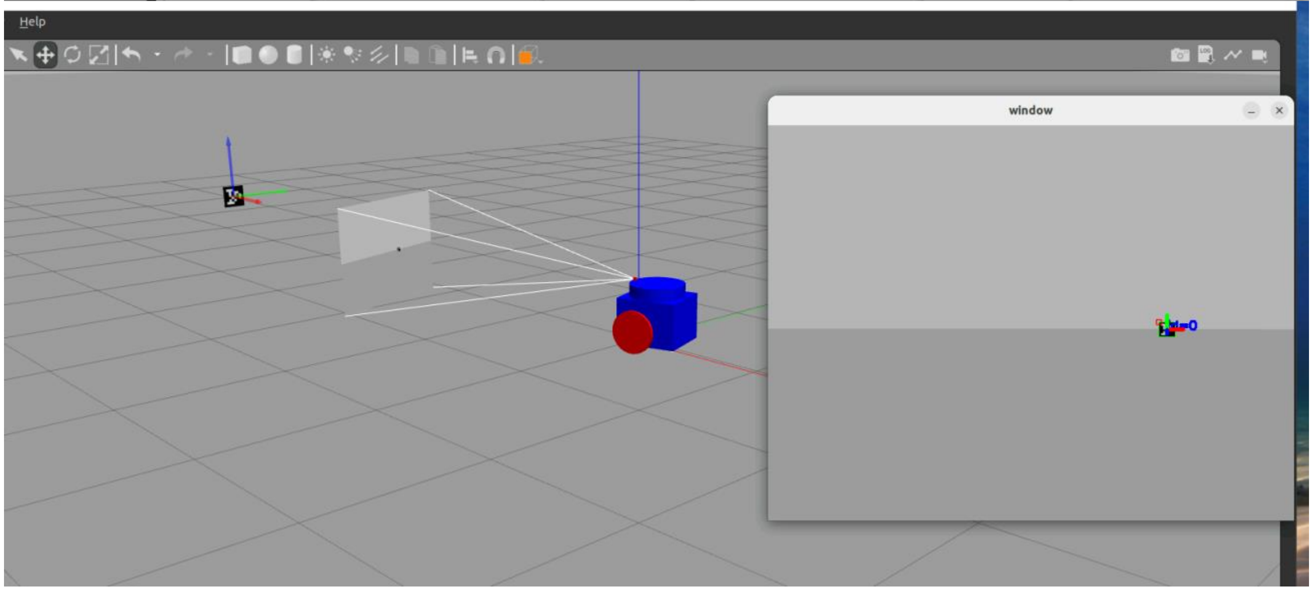
The drive_to_goal file:

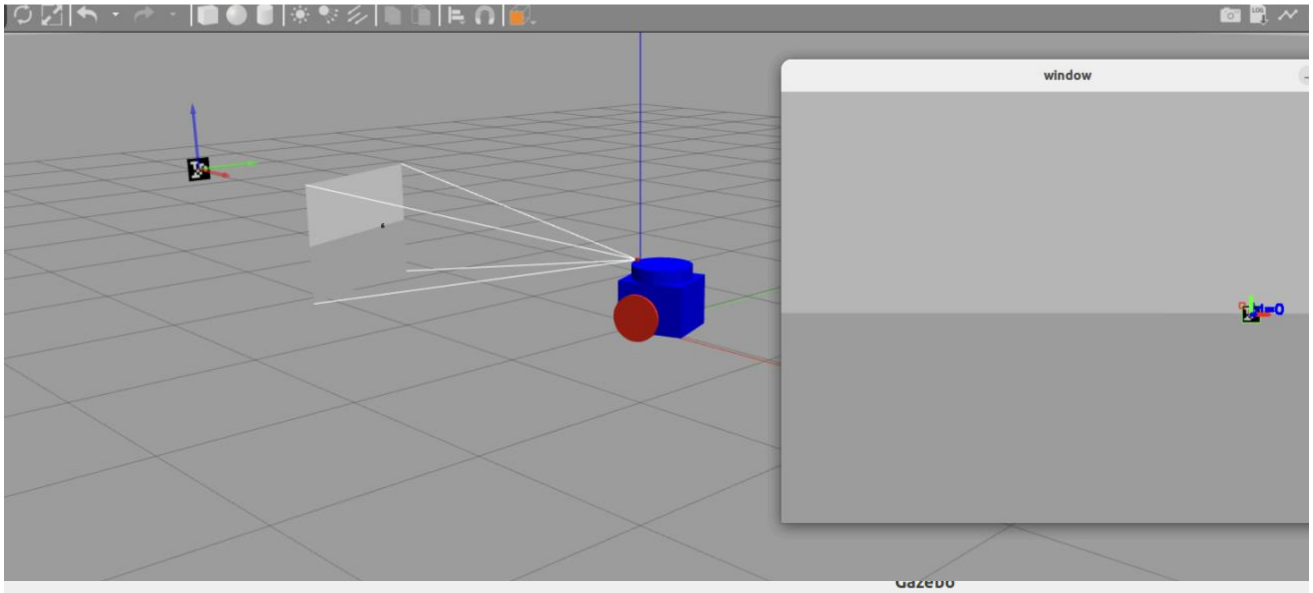
```
import math
import numpy as np
import rclpy
from rclpy.node import Node
from rclpy.parameter import Parameter
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Twist, Pose, Point, Quaternion
from std_msgs.msg import Bool
from geometry_msgs.msg import TransformStamped

def euler_from_quaternion(quaternion):
    """
    Converts quaternion (w in last place) to euler roll, pitch, yaw
    quaternion = [x, y, z, w]
    """
    x = quaternion.x
    y = quaternion.y
    z = quaternion.z
    w = quaternion.w

    sinr_cosp = 2 * (w * x + y * z)
    cosr_cosp = 1 - 2 * (x * x + y * y)
    roll = np.arctan2(sinr_cosp, cosr_cosp)

    sinp = 2 * (w * y - z * x)
    pitch = np.arcsin(sinp)

    siny_cosp = 2 * (w * z + x * y)
    cosy_cosp = 1 - 2 * (y * y + z * z)
    yaw = np.arctan2(siny_cosp, cosy_cosp)

    return roll, pitch, yaw


class move_to_goal(Node):
```

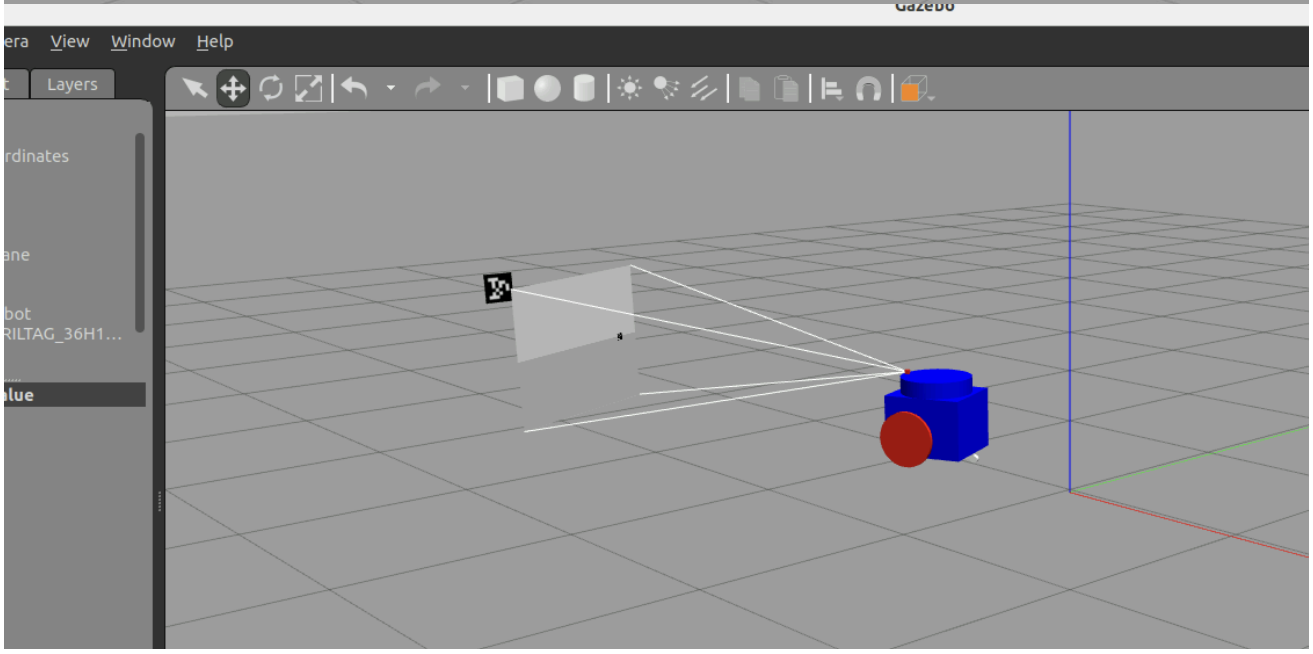
```python
    def __init__(self):
        super().__init__('aruco_robot_controller')

        # Publisher for cmd_vel to control robot velocity
        self.cmd_vel_pub = self.create_publisher(Twist, '/cmd_vel', 10)

        # Subscriber to know if the target is visible
        self.create_subscription(Bool, '/target_visible', self.target_visible_callback, 10)

        # Subscriber to get the position and orientation of the target
        self.create_subscription(TransformStamped, '/target_pose', self.target_pose_callback, 10)

        # Subscriber to get the odometry data
        self.create_subscription(Odometry, '/odom', self.odom_callback, 10)

        self.target_visible = False  # Whether the target is visible or not
        self.target_distance = None  # Distance to the target (None if not visible)
        self.stop_distance = 0.01  # Stop 1 cm away from the target

        self.robot_pose = None  # Robot's current pose
        self.robot_yaw = None  # Robot's current yaw angle
        self.target_position = None  # Target's position (x, y)

        self.timer = self.create_timer(0.1, self.control_loop)  # Timer for control loop

    def target_visible_callback(self, msg):
        self.target_visible = msg.data

    def target_pose_callback(self, msg):
        # Assuming the target's pose is relative to the robot's coordinate frame
        # Translation (x, y, z) gives the position of the target
        self.target_position = (msg.transform.translation.x, msg.transform.translation.y)
        self.target_distance = math.sqrt(msg.transform.translation.x ** 2 + msg.transform.translation.y ** 2)

    def odom_callback(self, msg):
        # Extract position and orientation (yaw) from odometry data
        self.robot_pose = msg.pose.pose.position
        _, _, self.robot_yaw = euler_from_quaternion(msg.pose.pose.orientation)

    def control_loop(self):
        if self.robot_pose is None or self.robot_yaw is None:
            self.get_logger().info('Waiting for robot odometry...')
            return

        twist = Twist()

        if not self.target_visible:
            # Target is not visible, spin the robot
            twist.angular.z = 0.5  # Rotate at 0.5 rad/s
            self.get_logger().info('Spinning to search for target...')
        elif self.target_visible and self.target_position is not None:
            target_x, target_y = self.target_position
            self.get_logger().info(f'Target visible at distance {self.target_distance:.2f}m')

            # Calculate angle to the target
            angle_to_target = math.atan2(target_y, target_x)

            # Adjust the angle based on the robot's current orientation (yaw)
            angle_difference = angle_to_target - self.robot_yaw

            # Normalize the angle difference to the range [-pi, pi]
            angle_difference = (angle_difference + np.pi) % (2 * np.pi) - np.pi

            if self.target_distance > self.stop_distance:
                # Move forward and adjust angular velocity to turn towards the target
                twist.linear.x = 0.2  # Move forward at 0.2 m/s
                twist.angular.z = 0.5 * angle_difference  # Proportional control to turn towards target
                self.get_logger().info('Approaching the target...')
            else:
                # Stop the robot when within 1 cm distance
                twist.linear.x = 0.0  # Stop moving forward
                twist.angular.z = 0.0  # Stop rotating
                self.get_logger().info('Stopping the robot 1cm in front of the target.')

        # Publish the velocity command
        self.cmd_vel_pub.publish(twist)


def main(args=None):
    rclpy.init(args=args)
    node = move_to_goal()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()


if __name__ == '__main__':
    main()
```

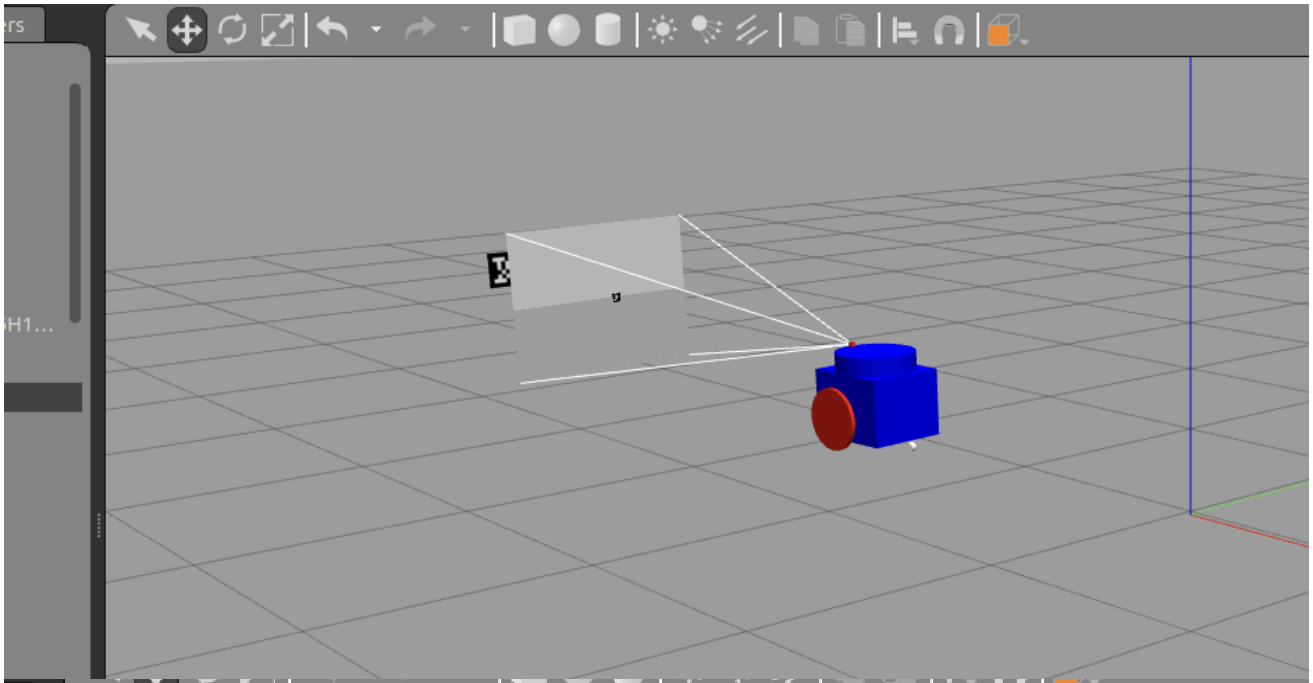
Screen dump:

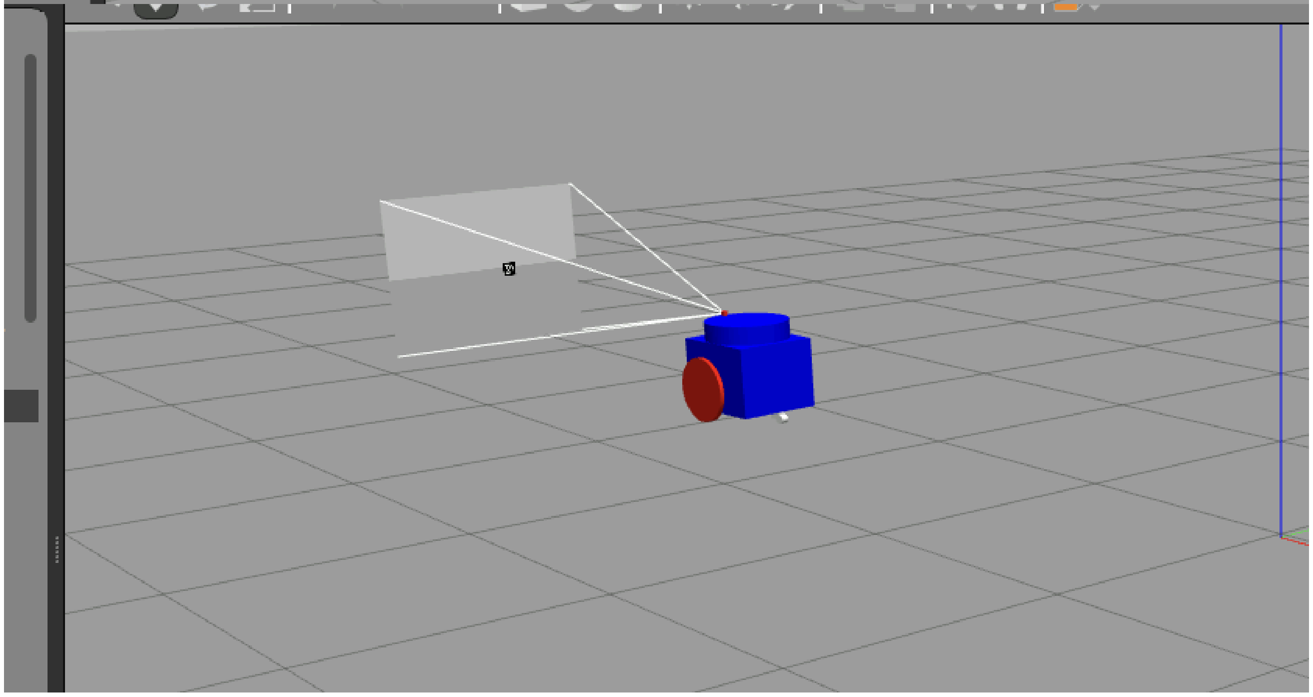





window
id=0


Gazebo
era
View
Window
Help
Layers
rdinates
ane
bot
RILTAG_36H1...
lue





To execute a long-duration mission, I structured the code by creating modular functions for distinct tasks such as target detection, position tracking, and robot odometry updates. I implemented a continuous control loop that checks if the target is visible, calculates the direction and distance to it, and adjusts the robot's movement accordingly. A timer was used to ensure the control loop runs regularly (e.g., every 0.1 seconds), allowing for quick responses to environmental changes. Additionally, I set stopping conditions to halt the robot when it approaches within 1 m of the target, preventing it from spinning indefinitely. This organized approach allows the robot to effectively handle the mission over an extended period.