ELEC330

Assignment 2 detailed report

Turtle in water –Teri–Transformative, Efficient, Robust, Intelligent Underwater Bio-Robot

Date: ***December 13, 2024***

Abstract

This report introduces TERI, a biomimetic underwater robot developed for autonomous navigation and mapping. Using Gazebo, a realistic environment, Atlantis.world, was created with obstacles and underwater objects to simulate real-world scenarios. The robot's motion was stabilized through key plugins, including hydrodynamics, joint position control, and thrusters, effectively addressing challenges like buoyancy and propulsion. Autonomous navigation was achieved using LiDAR for mapping and a camera for obstacle detection, with OpenCV enabling color-based object identification. TERI demonstrated reliable obstacle avoidance in various test scenarios, supported by a state memory mechanism to enhance navigation in confined spaces. This project highlights the integration of physical simulation, motion control, and sensing for robotic systems, providing a solid foundation for future advancements in underwater biomimetic robotics.

Introduction

With the advancement of robotics technology, biomimetic systems inspired by nature have become a focal point for researchers. These systems leverage the principles of biological structures and movements to solve complex engineering problems. In this report, we present the Transformative, Efficient, Robust, Intelligent Underwater Bio-Robot (TERI), designed to emulate the efficient locomotion and adaptability of aquatic creatures, specifically sea turtles. This project explores TERI's design, development, and testing within a simulated underwater environment.

TERI is developed as a biomimetic underwater robot capable of autonomous navigation, mapping, and obstacle avoidance in complex underwater environments. To achieve this, the Gazebo simulator was used to create a custom world file named *Atlantis.world*, which replicates realistic underwater conditions. The Atlantis environment includes diverse obstacles to simulate real-world challenges. Essential underwater physical dynamics, including buoyancy and fluid resistance, were incorporated into the simulation using advanced Gazebo plugins. These features allowed TERI to achieve stable motion and realistic behavior under simulated fluid forces.

The propulsion system for TERI was simulated using *thrusters*, which accurately modeled the forces generated by fin oscillation. By carefully tuning the robot's physical parameters and control systems, TERI achieved stable and efficient movement, demonstrating its capacity for realistic underwater exploration.

To enhance TERI's autonomy and decision-making capabilities, sensing and navigation algorithms were implemented. A GPU-accelerated LiDAR was used for real-time 3D mapping, while an onboard camera combined with OpenCV enabled obstacle detection and color-based object identification. These sensory systems allowed TERI to perceive its surroundings and navigate autonomously while avoiding obstacles.

This report details the development process, from constructing the Atlantis.world simulation environment to defining TERI's physical properties and integrating its control and sensing systems. The challenges encountered during testing, such as achieving neutral buoyancy and addressing motion stability, are analyzed alongside solutions implemented. The results highlight TERI’s ability to autonomously navigate complex environments, locate predefined objects, and perform obstacle avoidance.

Through the TERI project, this report provides insights into the integration of biomimetic design, physics simulation, and autonomous control systems for underwater robotics. The methods and results presented serve as a foundation for future advancements in underwater exploration and biomimetic robotic design.

World File (Atlantis.world)

Plugins utilized in the world

Atlantis is the .world file used for the robot to navigate. To simulate an underwater world, we will need the physics of the buoyancy of objects when immersed in the fluid by adding the `gz-sim-physics-system` and `gz-sim-buoyancy-system` plugins onto the world.

- **Physics** `(gz-sim-physics-system)`
  Responsible for loading and managing the physics engine, enabling Gazebo to simulate the dynamic behaviors of objects, such as gravity, collisions, and motion. This is a fundamental plugin for simulating physical environments

- **Buoyancy** `(gz-sim-buoyancy-system)`
  Used to simulate the buoyancy of objects in a fluid. The plugin calculates buoyancy based on the volume of the object submerged in the liquid, and it allows the specification of the fluid's density [1].

  To achieve neutral buoyancy for the robot (where the buoyancy force equals the weight, allowing it to hover), we can check the robot's volume in the Gazebo simulation (Figure 1) and calculate the buoyant force. Then, we adjust the mass and center of gravity of the robot's components so that the gravitational force matches the buoyant force.

  It is worth noting that during the robot's design phase, we incorporated ballast tanks inside the robot and designed attachment points for external floats and ballast weights. This allows the robot's weight and center of gravity to be easily adjusted to achieve neutral buoyancy during actual development.

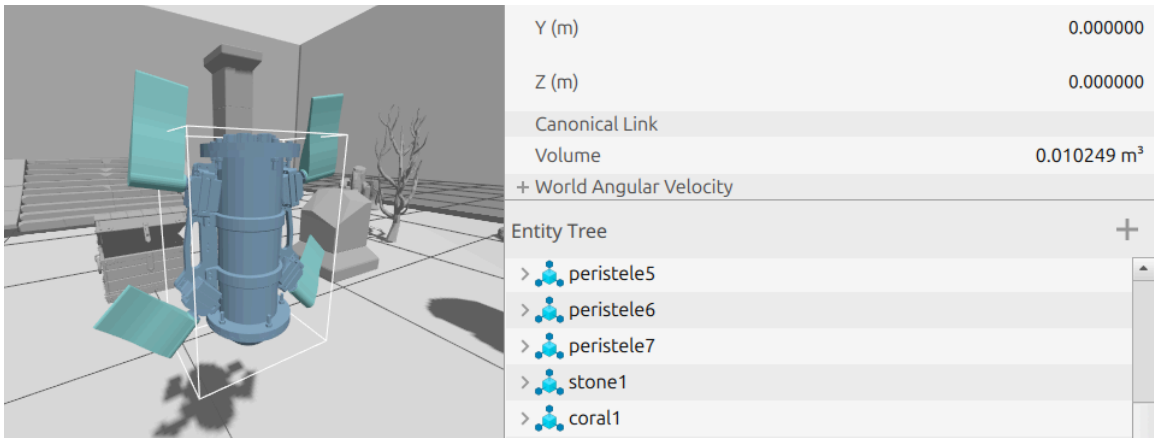

Figure 1. Checking the robot's volume

Implementation of Building the World (Atlantis.world)

Then, it is necessary to create an appropriate living environment for the robot, as shown in Figure 2.

The concept of this world is to resemble the lost city of Atlantis—an advanced civilization of its time that disappeared for centuries and was recently discovered at the bottom of the ocean—providing an environment for TERI to explore, simulating the robot's application scenarios.

We set the world's limit to be 10x10x3 units (meters), with ground_plane as the 10x10 base and wall1, wall2, wall3, and wall4 as the 10x3 side barriers. As shown in Figure 2, we can see that the walls are transparent. This is done by setting the material's opacity value to 0.1.

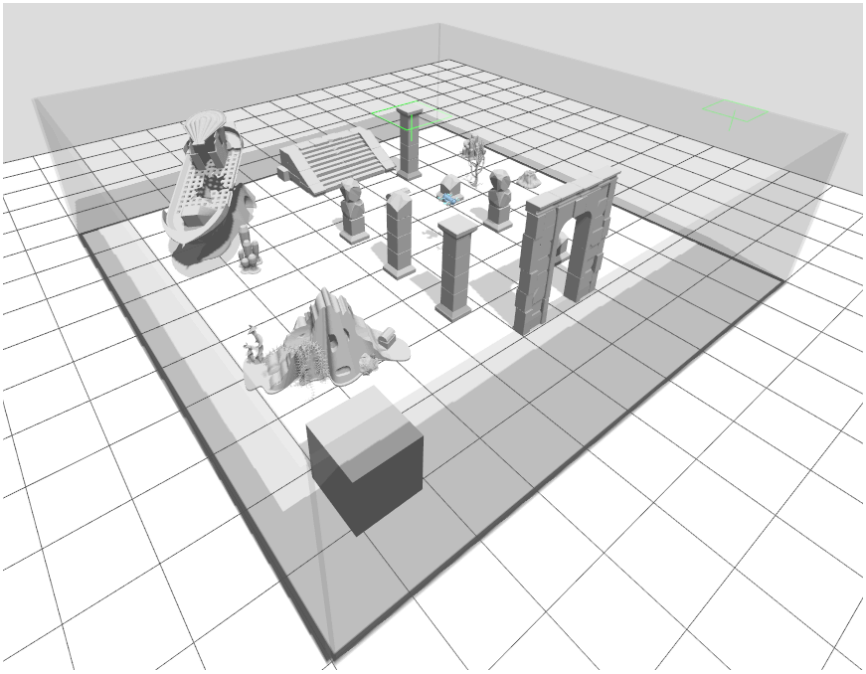

Figure 2. Atlantis.world

Atlantis has different obstacles that are exported from Gazebo's database and STL models.

- For simple objects like spheres and cubes, they can be directly defined in the `world` file.
- For more complex objects, such as arches and shipwrecks, SolidWorks is used to edit the model (Figure 3) and export it as an STL file. The STL file should then be placed into the `mesh` directory of the work package, specifically under the `collision` and `visual` subfolders, and referenced in the `world` file with the correct file path [2].

  It is worth noting that when exporting models from SolidWorks as STL files, the unit of measurement must be changed to meters (the default is millimeters). Otherwise, this

may result in the STL file failing to load in Gazebo or having excessively large dimensions.

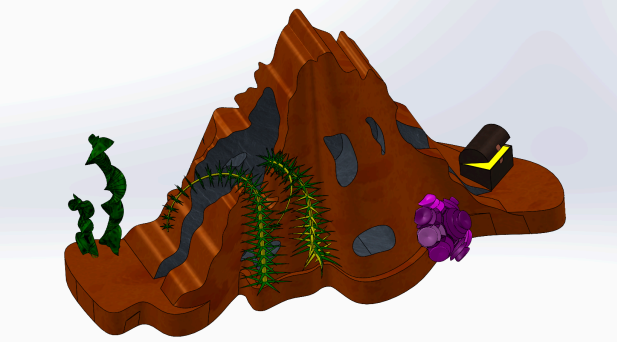

Figure 3. Stone in SolidWorks

To simulate a more realistic underwater exploration setting while aligning with the objectives outlined in Assignment 1, additional elements such as a shipwreck, broken peristyles, an arch, and a treasure box are included. Furthermore, a 1×1×1 cube, a sphere with a radius of 0.25, and a stair step with a slope are added to fulfill the required object specifications . Each object is precisely positioned by adjusting its X, Y, and Z axes, as well as its pitch, yaw, and roll rotations. The arrangement of all objects in the Atlantis environment is shown in Figure 4.

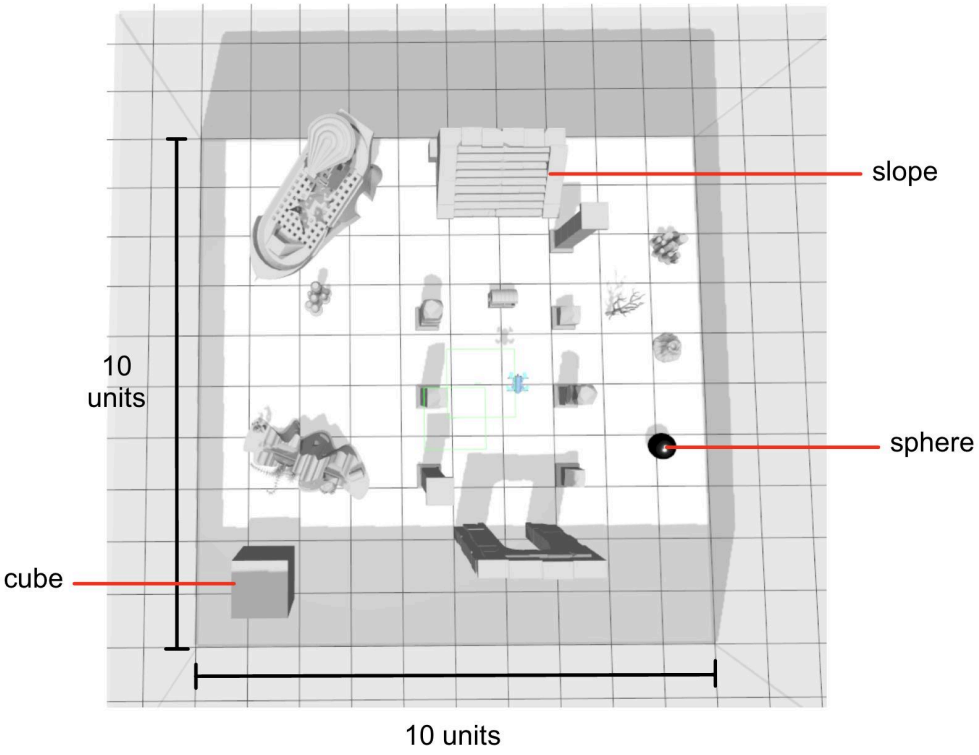

Figure 4. Bird's eye view of the world

During testing, it was observed that more than three environment-related obstacles of various sizes and shapes appeared in the robot's line of sight at certain points along its navigation path (Figure 5).

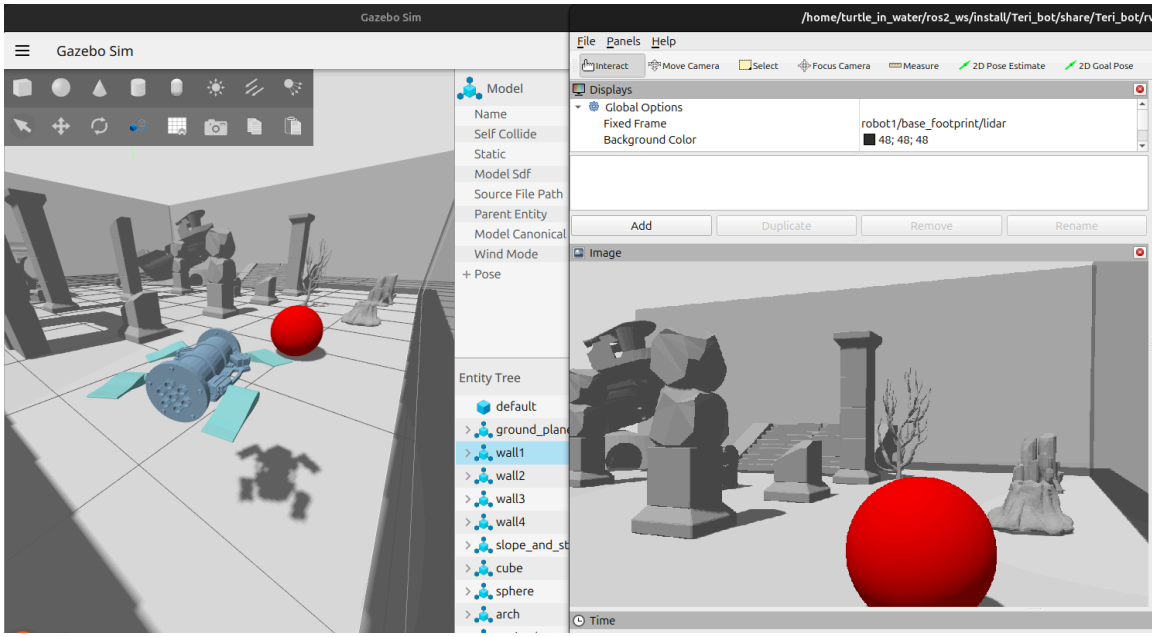

Figure 5. More than three obstacles in the robot's sight

Motion control (Teri_urdf.sdf)

Utilized plugins

Teri_urdf.sdf file is used to define the physical and visual properties of the TERI robot in the simulation environment. It includes necessary plugins for motion control and sensors. The file defines key parameters for the robot, such as joint configurations, environmental interactions, control types, and sensor types. This file is crucial for achieving realistic simulation of TERI's behavior, interaction with the environment, and compatibility with the simulation framework.

Hydrodynamics `(gz-sim-hydrodynamics-system)`

In the simulation of underwater environments, the buoyancy plugin can simulate the buoyant force acting on objects in the fluid. However, during testing, it was observed that: :

- When the robot is loaded into the Atlantis simulation environment, if the center of mass and the center of volume are not aligned, the robot oscillates continuously back and forth due to imbalance.

This phenomenon occurs because the buoyancy plugin does not account for the forces exerted by fluid flow on the robot's motion, resulting in less realistic simulation outcomes. In real environments, fluid forces, including motion resistance and torques, significantly influence the dynamic behavior of objects. To address this issue, the Hydrodynamics plugin [3] was introduced .

The Hydrodynamics plugin simulates hydrodynamic behavior by defining the forces and torques between the robot and the fluid. This enables a more realistic representation of the resistance, acceleration response, and stability changes induced by the fluid flow acting on the object in the water.

Several parameters need to be calculated and simulated:

1. **Linear Velocity-Related Parameters**:
   `xDotU`, `yDotV`, and `zDotW` define the contribution of velocity changes along each axis to drag forces.
2. **Angular Velocity-Related Parameters**:
   `kDotP`, `mDotQ`, and `nDotR` are used to model the stabilizing torques exerted by the fluid during rotational motion.
3. **Nonlinear Components**:
   `xUabsU`, `yVabsV`, and `zWabsW` describe the nonlinear behavior of fluid resistance, where the drag increases proportionally to the square of velocity at high speeds.

Given the complexity of accurately calculating these parameters for a robot with a complex structure, we approximated the robot's main body structure to a cylinder after analysis. Additionally, formulas and results for calculating these parameters for cylinders were obtained through research [4].

After integrating the Hydrodynamics plugin and conducting the simulation again, the robot's motion was found to align with real-world expectations:

- By simulating the damping torques generated by fluid flow during rotation, the robot's oscillation gradually stabilized and ultimately came to rest (Figure 6).

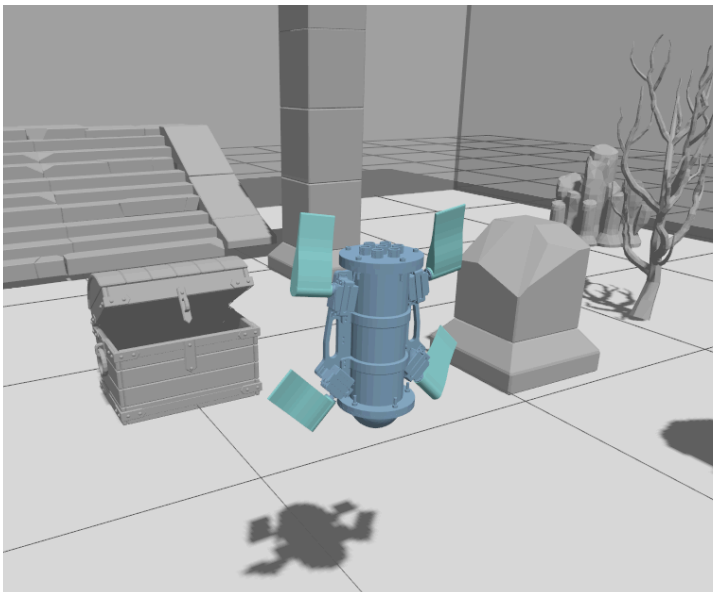

Figure 6. Rest robot

Joint-position-controller (`gz-sim-joint-position-controller-system`)

The plugin [5] is used to control the angular position (Position) of a specified joint. By employing a closed-loop control algorithm (PID controller), the plugin ensures that the joint reaches and maintains the target position accurately.

Plugin configurations
To achieve control over a specific joint, the following steps and configurations are required:

- **Reference to Joint Name**

  The joint name (e.g., `joint3`) must be specified in the SDF file to indicate which joint the plugin controls.

- **ROS 2 Topic for Target Position**

  A ROS 2 topic (e.g., `joint3_move`) needs to be defined to receive the target position commands.

- **PID Parameter Configuration**

  The plugin requires tuning of the PID parameters to achieve optimal control:

  - `p_gain` (Proportional gain): Determines the system's responsiveness to position error.
  - `i_gain` (Integral gain): Addresses steady-state error over time.

- `d_gain` (Derivative gain): Dampens rapid changes to prevent overshooting.

Proper tuning ensures:

- Faster response times.
- High precision.
- System stability.

- **Command Signal Limits**

  `cmd_max` and `cmd_min` are used to define the maximum and minimum control signals, preventing the application of excessive force or torque on the joint.

Plugin Workflow
After subscribing to the specified topic, the plugin computes and applies control signals to the selected joint using the PID controller formula:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt}$$

Where:

- e(t): The error between the current joint position and the target position.
- Kp,Ki,Kd: The proportional, integral, and derivative gains.

The computed control signal u(t) is then applied to the joint to adjust its position until the error approaches zero.

Joint Limit Settings

In addition to configuring the plugin, it is essential to properly set the limit parameters for the joint in the SDF file. These parameters ensure that the joint operates within realistic and safe boundaries (If these parameters are not set, `<effort>` and `<velocity>` default to `0`, causing the joint to remain stationary, as no motion or force is allowed.).

- `<lower>` and `<upper>`:

  Define the joint's motion range.

  Prevent the joint from moving beyond its physical or design limits.

- `<effort>`:

  Specifies the maximum allowable force or torque the joint can apply.

  Protects the joint from excessive loads.

- `<velocity>`:

  Limits the maximum speed of the joint.

  Ensures smooth and realistic movement.

Thruster (gz-sim-thruster-system)

After applying the Hydrodynamics and Joint-position-controller plugins, the robot's fin movements did not produce the expected forward motion, and errors were observed in the simulation.

This indicated that the Hydrodynamics plugin could not fully simulate the force generated by water on the robot, particularly the propulsion force created by the fin oscillation.

To accurately simulate the force exerted by the water on the robot (reactive thrust), the thruster plugin was introduced. This plugin provides realistic dynamic behavior for propulsion systems, enabling the robot model to move effectively in the simulation [3]. It is especially suitable for underwater robots, ships, and other applications requiring thrust.

As described in Assignment1, the propulsion force is generated by the oscillating motion of the fins, producing a reactive force from the water perpendicular to the surface of the fins (as illustrated in Figure 7). To simulate this propulsion force, the thruster plugin is utilized to generate the reactive thrust caused by fin movements.

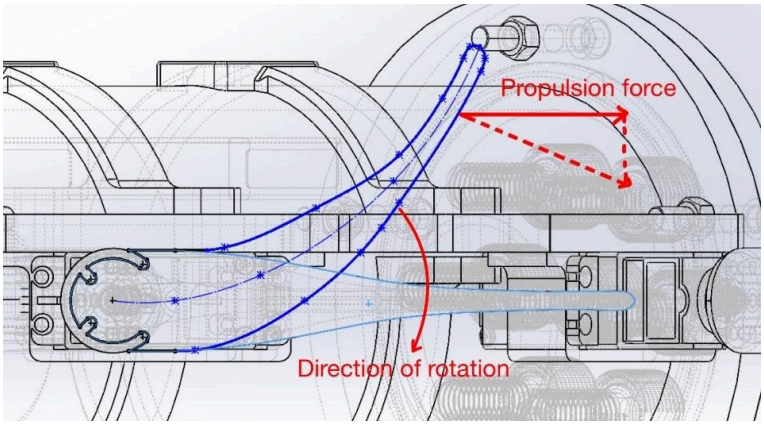

Figure 7. Propulsion force

Implementation

- **Modeling**

  In the SDF file, define the thruster model using simple cylindrical geometry.

  Assign the thruster appropriate mass, inertia, and connect it to the robot model through a joint.

- **ROS 2 Topic:**
  Bind the joint_name (e.g., pp2_joint) to the plugin and subscribe to ROS 2 thrust command topics.
  The thruster plugin automatically generates a control topic (/model/robot1/joint/pp2_joint/cmd_thrust) that does not require manual definition but must be bridged to Gazebo in the launch file.

Reaction Torque

After adding thrusters to four fins, the robot could produce forward propulsion in the simulation. However, it also exhibited rotation along the **Roll axis**. The Gazebo tutorial does not provide an explanation for this phenomenon.

Upon analysis, it was determined that this issue arose due to the **reaction torque** generated by the rotating propellers, which was not balanced between the fins.

After analysis, it was determined that this issue was caused by the reaction torque generated by the rotating propellers. Since all four propellers rotated in the same direction, the reaction torques could not cancel each other out.

To resolve this issue, we changed the `thrust_coefficient` of the thrusters on one side to a negative value. This allowed the propellers on that side to rotate in the opposite direction while maintaining the same thrust direction, thereby canceling out the reaction torques between the left and right sides.

After re-running the simulation, we observed that the robot achieved stable horizontal forward motion, and the roll-axis rotation issue disappeared.

Optimization

After confirming the position and direction of the thrusters, we reduced the size of the four propellers to minimize their impact on buoyancy. Additionally, we made them invisible to facilitate further development.

Sensors (Teri_urdf.sdf)

Lidar (gpu_lidar)

The GPU Lidar sensor plugin simulates a LiDAR device, generating real-time 3D point cloud data of the environment for robotic navigation, obstacle avoidance, and environment perception. By leveraging GPU acceleration, this plugin enables faster simulation of laser scanning with higher resolution and update rates[6].

Position of the LiDAR

The position and orientation of the LiDAR relative to the robot's reference point (`base_footprint`) must be defined using the `<pose>` tag.

It is worth to note that the LiDAR should not be placed inside the robot's body, as its emitted laser beams could be blocked by the robot's shell, resulting in incorrect or incomplete data.

LiDAR Parameter Configuration

- **Update Rate (`<update_rate>`)**
  This parameter determines how many laser scans the LiDAR generates per second. For low-dynamic environments or limited computational resources, a lower value is recommended (e.g., 10 scans per second are used in this project due to computational constraints). In high-dynamic scenarios, this value can be increased to capture more frequent updates.
- **Scanning Parameters (`<scan>`)**
  The `<scan>` tag defines the horizontal and vertical scanning behaviors:
  - Horizontal Scan (`<horizontal>`): Defines the resolution (`<samples>`) and the angular range (`<min_angle>` and `<max_angle>`).
  - Vertical Scan (`<vertical>`): Similarly defines resolution and angular range in the vertical direction.

  By combining horizontal and vertical scans, the LiDAR can generate comprehensive point cloud data (Figure 8).

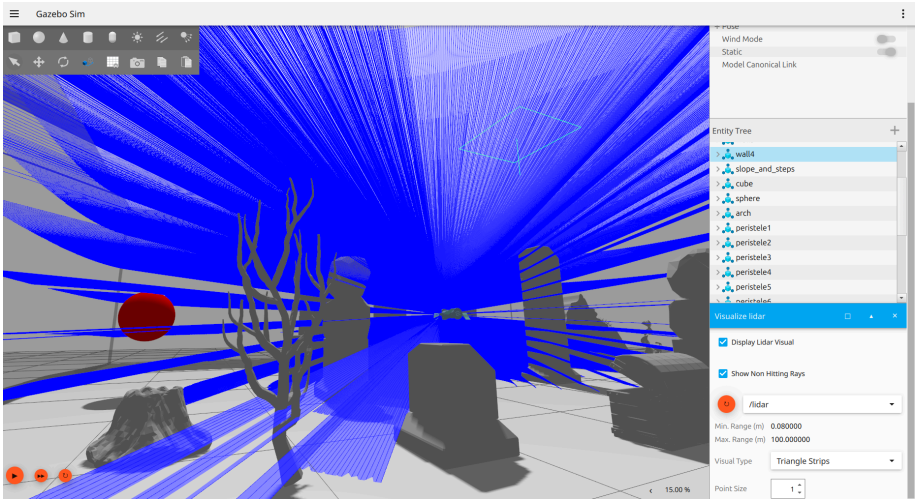

Figure 8. Lidar hitting rays

- **Range Settings (`<range>`)**
  These parameters set the minimum and maximum distances the LiDAR can measure, along with the resolution of the measurements. Configuring these parameters ensures the LiDAR's behavior closely matches that of real-world sensors.

Data Publishing Topic

The LiDAR publishes its scan data to a ROS 2 topic, specified as `lidar(/lidar/points)` in this configuration.

The robot's controllers or algorithms can subscribe to this topic to receive and process LiDAR data for tasks such as navigation or mapping.

To enable data visualization, we use the `ros_gz_bridge` node to convert Gazebo's internal data format (`gz.msgs.PointCloudPacked`) into ROS 2's standard format (`sensor_msgs/PointCloud2`). RViz subscribes to the LiDAR point cloud data topic (`/lidar/points`) and displays the 3D environment in real time. This provides an intuitive visualization of the LiDAR output, facilitating the verification and debugging of LiDAR functionality.

After completing the above steps, as shown in Figure 9, the point cloud on the left clearly displays the arch and two stone columns on the right side of the actual environment, indicating that the LiDAR successfully generates a map of the world.

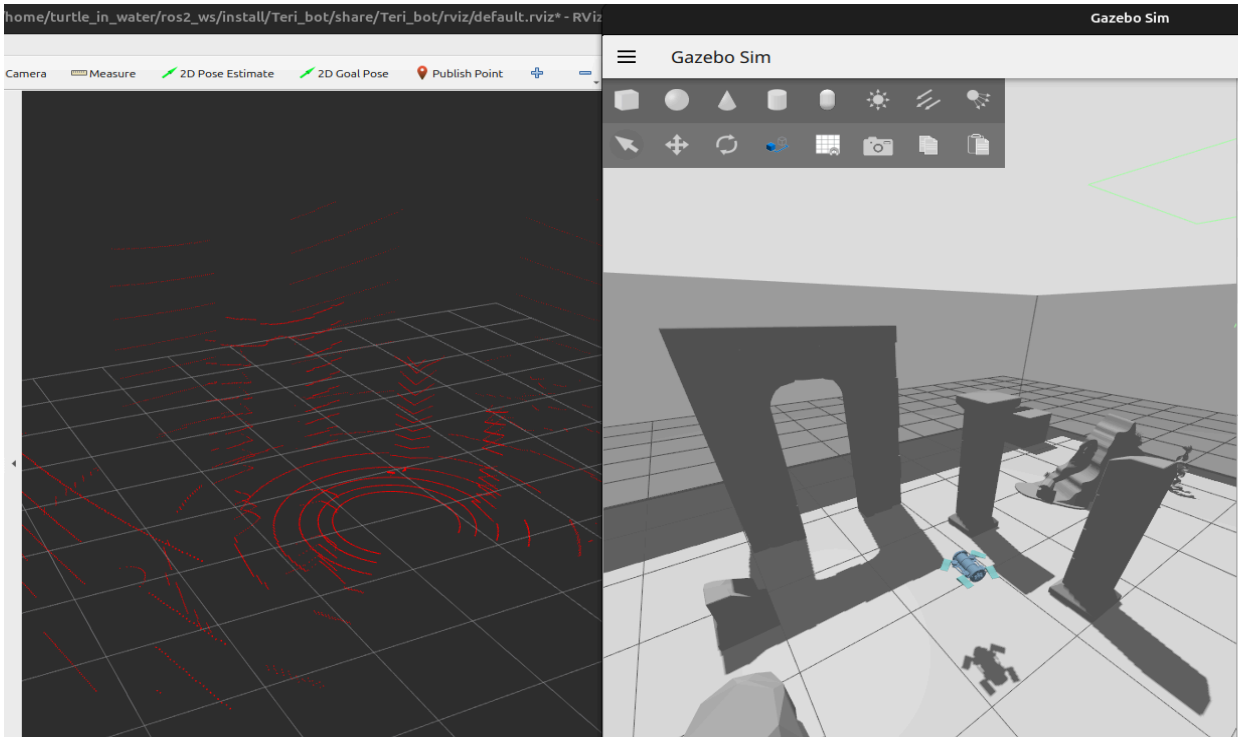

Figure 9. Comparison of point cloud and actual environment

Camera (`boundingbox_camera`)

The `boundingbox_camera` plugin is a camera sensor plugin primarily used in Gazebo simulations to simulate object detection functionality and capture images of the environment [7].

Position of the Camera

Similar to LiDAR, the camera's relative position must be defined. It should also not be placed inside the robot's body to prevent its field of view from being obstructed by the robot's structure.

Camera Parameter Configuration

The plugin supports configuration of parameters such as the field of view, resolution, and clipping distances, allowing it to closely match the characteristics of a real-world camera.

Data Publishing Topic

A topic needs to be specified for publishing the camera data. In this configuration, the topic is defined as `camera1`. The sensor publishes data to this topic, which can be subscribed to by RViz (Figure 10) or other nodes for visualization and processing.

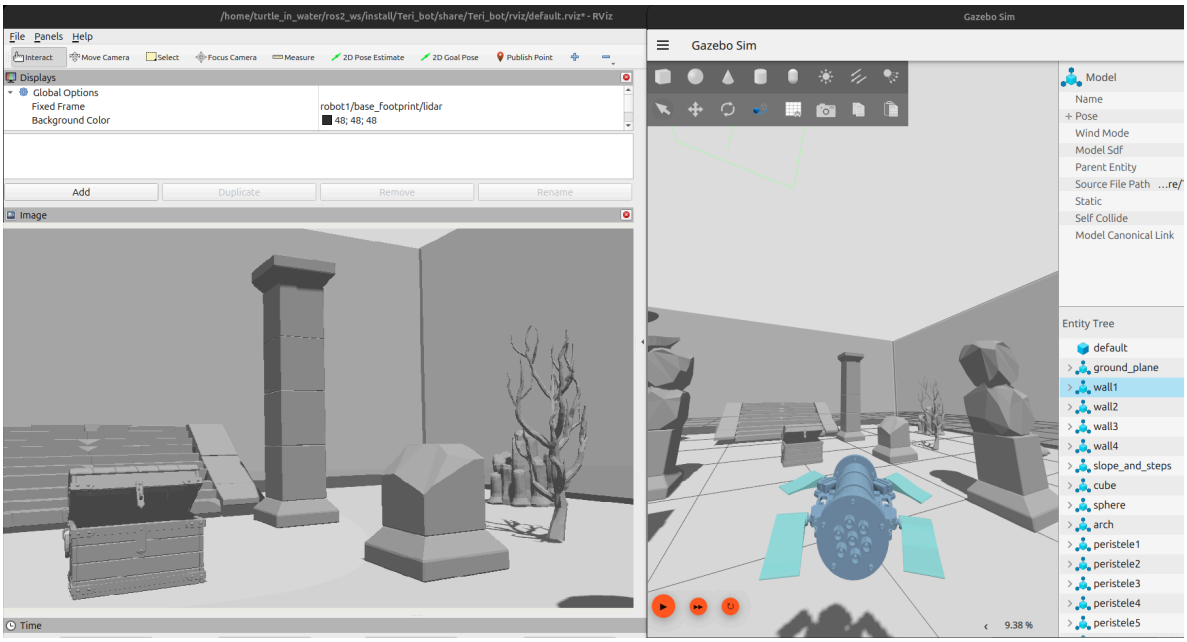

Figure 10. Camera view in RViz

Navigation Algorithm(joint_controller.py)

Code Structure

First, a class named JointController is defined, inheriting from ROS 2's base class Node. This class is responsible for subscribing to sensor data, making decisions, and publishing control commands. The following functions are included in the class:

1. __init__(self)

- Purpose:
  Constructor function used to initialize the node joint_controller, set up subscribers, publishers, and initial parameters.
- Key Functions:
  - Subscribe to LiDAR point cloud data:
    Subscribes to the /lidar/points topic to receive point cloud data.
  - Create joint control and thrust publishers:
    Publishes joint movement and thruster thrust values to relevant topics.
  - Initialize parameters:
    Includes thrust values, obstacle distance thresholds, etc.
  - Set up timers:
    Creates a timer for periodically updating joint movements.

2. set_thrust(self, forward=False, left=False, right=False)

- Purpose:
  Sets the thrust values for the actuators and publishes them to the corresponding ROS 2 topics.

3. timer_callback(self)

- Purpose:
  Timer callback function used to periodically update the oscillation state of the robot's joints

4. point_cloud_callback(self, msg)

- Purpose:
  Callback function to process subscribed point cloud data, implementing obstacle detection and general logic(Described in detail in the following section).
- Key Functions:

  Parse and filter point cloud data:
  Extracts coordinates from the point cloud data and filters points

  Detect obstacles:
  Identifies obstacles within the robot's surroundings.

  Update movement state:
  Based on the obstacle distance, determines whether to turn or continue moving straight.

General logic

By importing `sensor_msgs_py.point_cloud2`, the `point_cloud_callback` function is executed each time the `JointController` node receives a new message from the `/lidar/points` topic. Therefore, to promptly respond to changes in the environment, the control logic of this project is primarily implemented within the `point_cloud_callback` function. Figure 12 illustrates the detailed logic of this function:

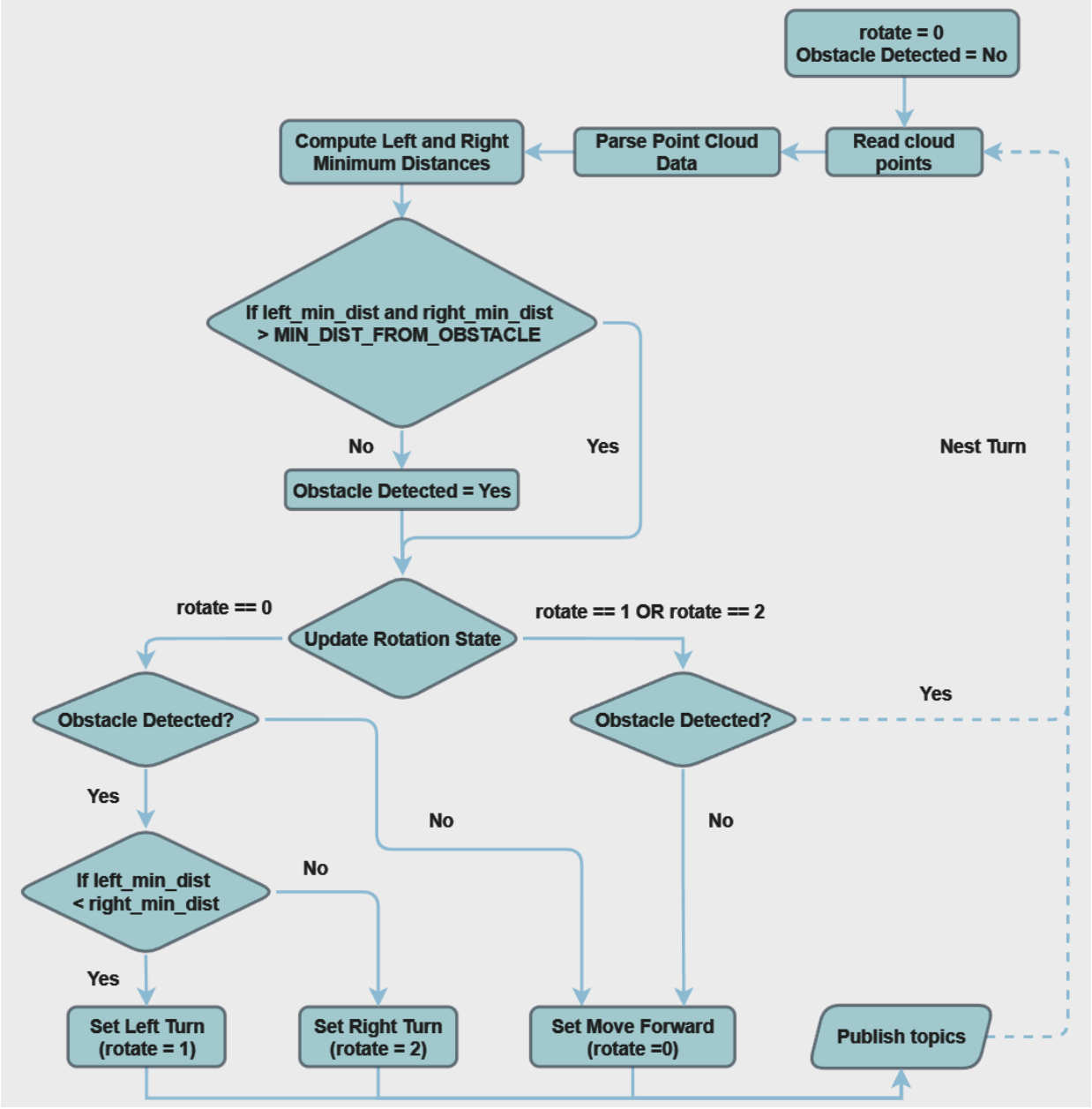

Figure 12. Logic of the **point_cloud_callback** function

Data Processing and Filtering

First, LiDAR data needs to be further processed to identify the relative positions of obstacles in the environment.

- **Point Cloud Data Parsing**

  The `read_points` function is used to parse `PointCloud2` messages.

  Note: Although the definition of LiDAR involves an angular range (<min_angle> and <max_angle>), the returned values are not in polar coordinates but in (x, y, z) coordinates.

- **Data Filtering**

  Horizontal Scanning Range:

  The minimum horizontal scanning range is determined using the robot's width (`robot_front_width`) and the `MIN_DIST_FROM_OBSTACLE` threshold.

  The effective horizontal range is calculated to identify potential obstacles that fall within the robot's movement area.

  Points outside the calculated horizontal scanning width (`±robot_front_width / 2`) are discarded. (Figure 13).

  Vertical Scanning Range:

  Similarly, a vertical range is defined based on the robot's height (`robot_front_height`), ensuring detection is limited to obstacles that are within the robot's frontal vertical field of view.

  Points outside the calculated vertical height (`±robot_front_height / 2`) are excluded (Figure 13).

  Distance Threshold:

  Any points where the distance (`x`) is greater than `MIN_DIST_FROM_OBSTACLE` are considered safe and filtered out.

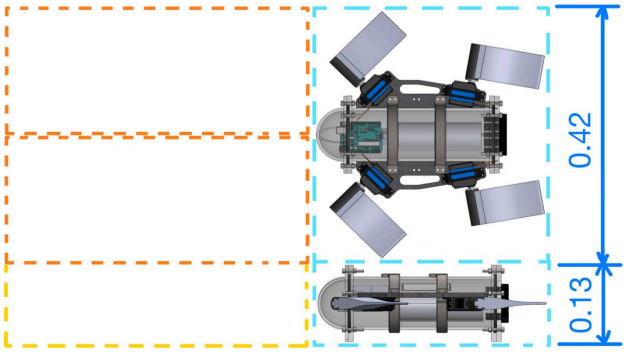

Figure 13. Minimum horizontal scanning range

Obstacle Avoidance

The algorithm iterates through all valid scanned distances on both the left and right sides, comparing each distance with the current minimum distance for that side to select the smaller value:

- Left Side:
  Compute the minimum distance for points where `y > 0`.
- Right Side:
  Compute the minimum distance for points where `y ≤ 0`.

Motion State Updates

The robot's motion state is updated based on the minimum distances to obstacles on the left and right sides:

- Move Forward:
  If the minimum distances on both the left and right sides are greater than the threshold `MIN_DIST_FROM_OBSTACLE`, the robot continues moving forward.
- Obstacle Avoidance:
  If the minimum distance on either side is less than the threshold:
  - If the left-side distance is smaller

    The robot turns right. The right fin will stop swinging, and the left fin will increase the swing amplitude, so that the left and right forces of the robot are inconsistent, and the differential rotation is achieved.。

  - If the right-side distance is smaller,

    The robot turns left.

State Memory

Testing has shown that without state memory during decision-making, the robot may get trapped when obstacles are present on both sides with minimal difference in distance (as illustrated in Figure 14).

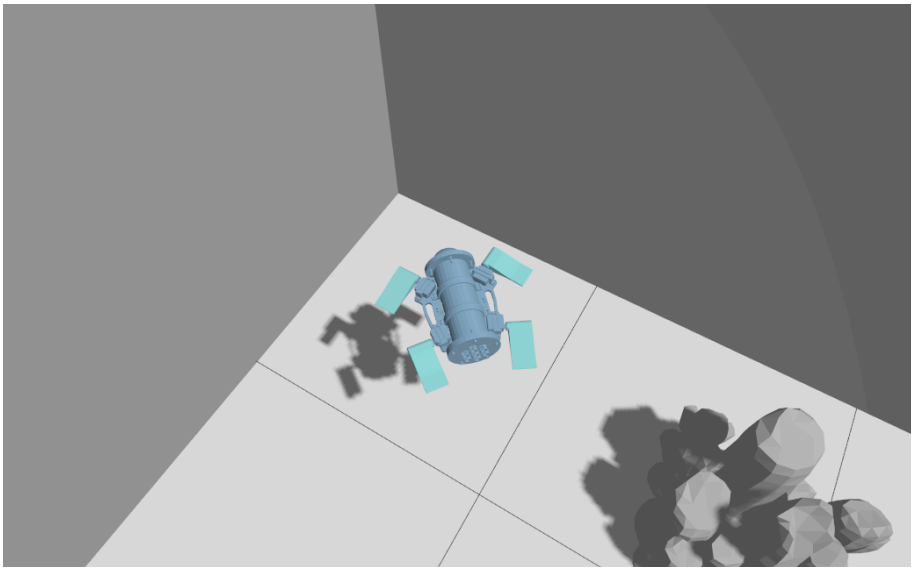

Figure 14. The trapped robot

To address this issue, a variable `rotate` is introduced to track the robot's rotational state. This method allows the robot to adjust its rotational behavior based on both the previous state and the current position of obstacles. The optimized state update logic is as follows:

- **No Rotation (0)**:
  - If there are no obstacles in front, the robot continues to move straight.
  - If an obstacle is detected, the robot stops forward thrust and selects a rotation direction based on the relative position of the obstacle. The `rotate` state is updated accordingly to reflect this change.
- **Rotate Left (1)**:
  - The robot continues turning left until there are no obstacles in front(minimum distances on both sides are greater than the threshold distance).
  - Once the path is clear, the `rotate` state is reset to `0`, and the robot resumes straight movement.
- **Rotate Right (2)**:
  - The robot continues turning right until no obstacles are detected in front(minimum distances on both sides are greater than the threshold distance).
  - Once the path is clear, the `rotate` state is reset to `0`, and the robot resumes straight movement.

This state memory mechanism ensures that the robot can respond continuously and consistently when encountering obstacles. After avoiding obstacles, it can effectively resume straight-line motion. By maintaining the `rotate` state, the robot's behavior becomes more intelligent, improving both the efficiency and safety of obstacle avoidance.

Autonomous Navigation Result

After implementing the above algorithm on the robot, it successfully achieved autonomous navigation in different testing scenarios and was able to drive autonomously to a previously defined object.

In Scenario 1 (Figure 15), the robot effectively explored the entire map through continuous autonomous obstacle avoidance, successfully avoiding collisions while ultimately locating the defined object.

After placing obstacles in different positions and adjusting the robot's starting point (Figure 16), the robot was still able to autonomously navigate, avoid collisions, and ultimately reach the defined object.

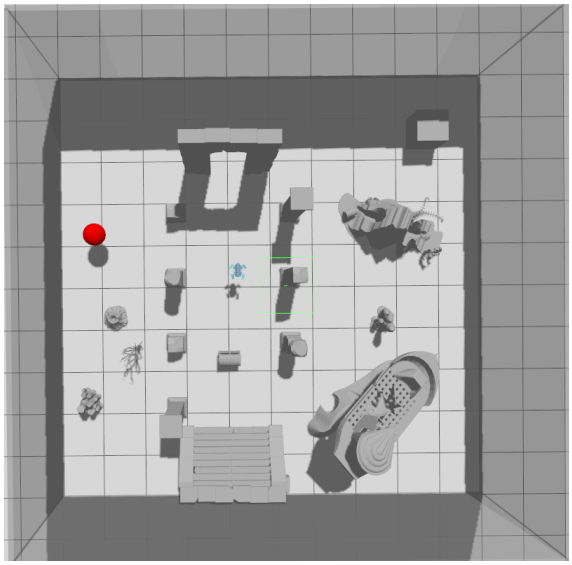

Figure15. scenarios1

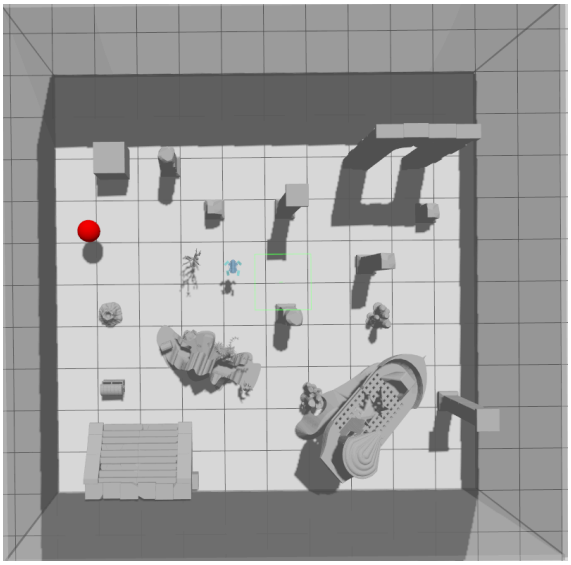

Figure16. scenarios2

Object Detection (color_detector_node.py)

For TERI to detect objects, it must have a camera sensor to allow TERI to see its surroundings. A camera sensor is defined in TERI's SDF file as `type="camera"`. Different from `type="boundbox_camera"`, the camera sensor is a standard camera that simulates a typical visual camera. It can capture raw image data which will be necessary for object detection. A camera plugin called `"camera_controller"` is then added to allow TERI to capture the target object. An object detection algorithm is then created.

Code Structure

ColorDetectionNode is defined to inherit from ROS 2's base class Node. This Node is responsible for detecting the red color from the camera's vision and taking a picture of it.

The following functions are included in the class:

1. __init__(self)

- Purpose:
  Connects Node to the camera, allowing images from the robot to be received. Sends messages to other nodes.
- Key Functions:
  - Subscribe to Camera images:
    Subscribes to the `/camera1/image_raw` topic to receive messages of type `Image`.
  - Publishes processed image:
    It publishes the processed image to `/processed_image`.
  - CvBridge:
    Converts ROS Image messages to OpenCV format for image processing [8]. OpenCV is a computer vision library that is easier to use for tasks like color detection.

2. image_callback(self, msg)

- Purpose:
  Converts ROS Image to OpenCV format. It has a callback function to receive and process the images from `/camera1/image_raw` for real-time analysis.
- Key Functions:
  - Convert Image to HSV Color Space:
    `cvtColor()` is OpenCV's function that converts BGR (Blue, Green, Red) into HSV (Hue, Saturation, Value) color space.

- Red Color Range:
  `red_lower` and `red_upper` define the darkest and brightest saturation of the red color spectrum.
- Masking Red Objects:
  `cv2.inRange()` creates a binary mask to separate detected red objects by converting them into white and all other colors are set to black.
- Detect Contours on Original Image:
  `cv2.drawContours()` draws the contours of the detected red object on the original BGR image.

General logic

By importing `rclpy`, `cv2`, and `numpy` the `image_callback` function is executed each time the `ColorDetection` node receives a new message from the `/camera/image_raw` topic. Processed images that detect red will send messages to the node to convert the image back to ROS format and take a picture of the object with an output file of `camera1_image.jpg`.

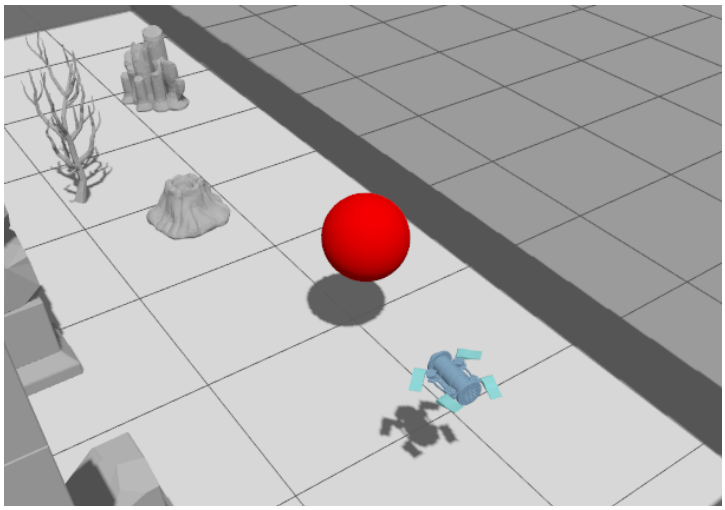

Figure 17. TERI with target object

We chose a sphere as the target object for the robot as it is the easiest shape to detect, and it has the same 2D shape as a circle from any point of view. We also set the sphere to have a true red color as shown in Figure 17.

By running `ros2 run rqt_image_view rqt_image_view` we can see TERI's camera vision of the red sphere as shown in Figure 18. TERI's vision can also be viewed directly through Rviz.

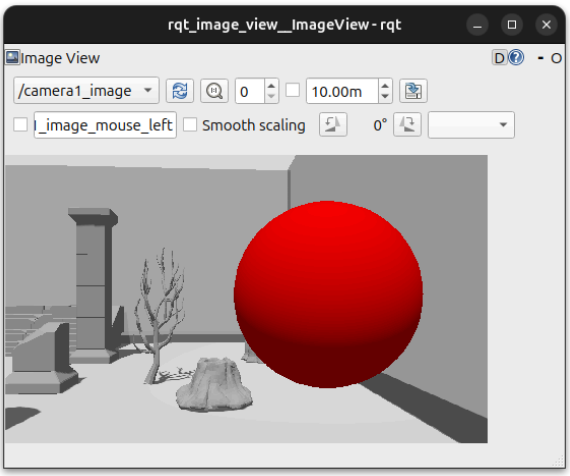

Figure 18. Image of target object

Launch File (launch.py)

In order to initialize and startup various components of a robotic system, `launch.py` is designed to serve as the launch file. The script includes executes multiple nodes to successfully run the robotic simulation. The nodes includes:

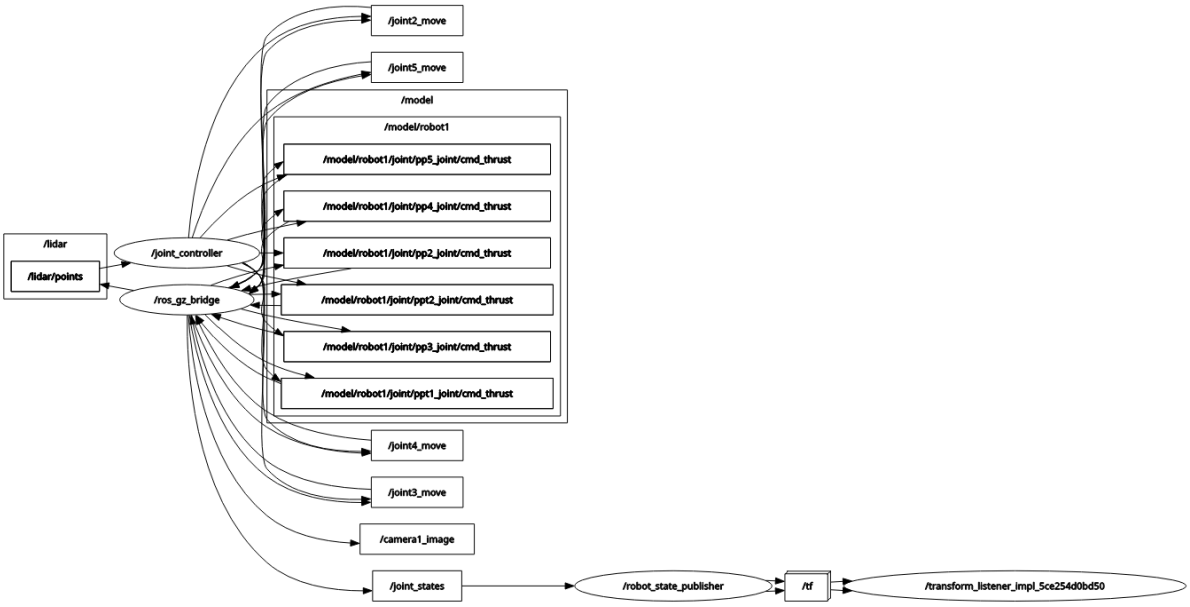

Figure 11. rqt_graph

1. robot_state_publisher

- Publishes the state of robot to track its pose and joint configurations

2. gz_sim

- Launch the Gazebo simulation. It is responsible in loading the world and robot model.

3. bridge

- Consist of `ros_gz_bridge`, it serves as a bridge between ROS2 and Gazebo, allowing the robot to move in the simulated world

4. joint_controller_node

- Responsible for controlling the robot's joints to precisely move the robot. Includes `ExecuteProcess` which launch a new terminal for manual movement.

5. rviz2

- A tool to visualize the robot's sensor data and state in the simulated world. It plays an important role in mapping and computer vision.

6. vision_node

- Consists of `color_detector_node`, it is responsible in finding the target object.

The launch file serves as the foundation of TERI's testing in a simulated environment by integrating sensor data, visualization, and joint control.

Reflection

Environment Construction

The `Atlantis.world` simulation environment was successfully developed to meet the assignment's specifications. The environment included a variety of obstacles, such as slopes, cubes, and spheres, as well as additional elements like shipwrecks, treasure chests, and pillars, which provided a realistic underwater exploration setting. Custom STL models were designed and imported using SolidWorks to enhance the environmental complexity and fidelity. Furthermore, Gazebo's buoyancy and hydrodynamics plugins were integrated to achieve a highly accurate simulation of underwater physical dynamics. This project facilitated the development of practical skills in designing and simulating complex physical environments, balancing functional requirements with environmental realism. Future work will focus on incorporating dynamic obstacles and moving targets to evaluate the robot's navigation capabilities in more complex scenarios.

Robot Motion Control

The project successfully implemented essential motion control mechanisms by integrating the joint position controller and thruster plugins. The joint position controller provided precise joint manipulation via PID control, while the thruster plugin simulated propulsion forces and solved issues such as rolling caused by reaction torque, ensuring motion stability and efficiency. This aspect of the project reinforced expertise in developing and optimizing control systems for complex and dynamic environments. To further enhance the robot’s motion control, future iterations will include an IMU sensor to improve attitude detection and system robustness.

Autonomous Navigation

Autonomous navigation was achieved using LiDAR for environmental mapping, enabling the robot to perform effectively in various testing scenarios. The robot successfully avoided obstacles and reached predefined targets. Additionally, a state memory mechanism was introduced to address scenarios where the robot risked becoming trapped in confined spaces, thereby improving its navigation reliability. This project provided valuable insights into the design and implementation of efficient navigation algorithms, with a focus on balancing obstacle avoidance and goal-oriented behavior. Future enhancements will include optimizing path planning algorithms for complex environments, incorporating dynamic obstacle detection and prediction, and refining map generation techniques to improve real-time accuracy and responsiveness.

Object Detection

Object detection posed a significant technical challenge during the project. While the robot successfully identified the target object using OpenCV by converting ROS image formats to OpenCV and back, it was unable to autonomously capture images of the detected object due to time and computational constraints. This experience highlighted the importance of integrating and bridging diverse software frameworks for advanced robotic functionalities. Future improvements will aim to enable automatic image capture and expand the detection capabilities to accommodate more complex object characteristics.

Conclusion

With this Assignment 2, we have demonstrated the development and simulation of a biomimetic robotic system. Further improvements from the previous Assignment 1 are made.

Atlantis.world is created as the world file in Gazebo. It represents the appropriate living environment for a sea turtle to navigate. Atlantis is confined with four walls and has more than three environment related obstacles,and a sphere to be used as the target object.

We then implement sensing in the robotic system in order to navigate autonomously through computer vision. This is done by first allowing the robot to move through the open space by adding key plugins, such as the hydrodynamics, joint controller and thruster systems. With these plugins, TERI is able to control its movement under a stable and realistic motion under fluid forces, addressing the challenges of buoyancy and propulsion. In order to allow autonomous navigation, sensors such as LiDAR and a camera are added to TERI. The LiDAR sensor enables TERI to navigate autonomously by performing environment mapping while a camera is used for obstacle detection of a different color with the help of OpenCV. Rviz is used to show all TERI's perception of the world through the sensors.

Overall, This project not only deepened our understanding of robotic simulation but also addressed real-world engineering challenges. It highlighted the importance of rigorous testing and iterative improvement in developing complex systems. Moving forward, TERI serves as a solid foundation for advancing underwater biomimetic robotics, with potential applications in environmental monitoring, underwater exploration, and disaster response. Future iterations will focus on enhancing path planning, incorporating dynamic obstacle detection, and optimizing real-time performance.

Reference

[1] Gazebo Tutorials, "Hydrodynamics in Gazebo," [Online]. Available: https://classic.gazebosim.org/tutorials?tut=hydrodynamics&cat=physics.

[2] Gazebo Tutorials, "Guided Tutorials - Level 2," [Online]. Available: https://classic.gazebosim.org/tutorials?tut=guided_i2.

[3] Gazebo Tutorials, "Underwater Vehicles," [Online]. Available: https://gazebosim.org/api/sim/7/underwater_vehicles.html.

[4] L. Hong et al., "Advances in Marine Engineering Research," *Journal of Marine Science and Engineering*, vol. 10, no. 8, pp. 1049, Aug. 2022. [Online]. Available: https://www.mdpi.com/2077-1312/10/8/1049?utm_source.

[5] Gazebo API Documentation, "Joint Controllers," [Online]. Available: https://gazebosim.org/api/sim/8/jointcontrollers.html.

[6] RobotecAI, "Robotec GPULidar," GitHub Repository, [Online]. Available: https://github.com/RobotecAI/RobotecGPULidar?tab=readme-ov-file.

[7] Gazebo API Documentation, "Bounding Box Camera," [Online]. Available: https://gazebosim.org/api/sensors/9/boundingbox_camera.html.

[8] ROS Tutorials, “Converting between ROS images and OpenCV images (Python)”, [Online]. Available: https://wiki.ros.org/cv_bridge/Tutorials/ConvertingBetweenROSImagesAndOpenCVImagesPython