Fall 2022

Final Project

Introduction to Robot Programming

December 17, 2022

Students:
Sourang Sri Hari

Sandip Sharan Senthil Kumar

Sandeep Thalapanane

Instructors:
Z. Kootbally

Course code:
ENPM809Y

Contents

1 Introduction 3

2 Approach 3

3 Challenges Faced 9

4 Final Results 10

5 Contributions 12

6 Resources 12

7 Course Feedback 12

1 Introduction

The goal of this project is to command the turtlebot to move towards a destination point with the help of aruco markers in the gazebo world. Initially, the turtlebot needs to move to a known destination, and upon reaching the point, it needs to rotate and try to find a fiducial marker. As soon as the turtlebot traces the marker, it needs to move forward and reach any one of the four fiducial marker locations which are represented by the colors green, yellow, blue, and red. The report explains the approach to achieve the goal, the challenges faced while doing this project and how the issues are solved, the Contributions of each team member, and the resources used.

2 Approach

STEP 1

The first task of the project was to connect the frames of the Odom and the base footprint of the robot. Odom is a fixed frame in the gazebo world whereas the base footprint is part of the turtlebot and keeps changing its location as the turtlebot moves around the gazebo world. In order to connect the tf frames tree, a broadcaster needs to be written which connects the base footprint of the turtlebot to the Odom frame.

There are two types of broadcasters that are generally used: static and non-static. Static broadcaster involves frames that are stationary and which do not move with respect to one another. For the project, a non-static broadcaster is used since the base footprint moves from its reference frame, Odom.

This step is initiated by creating a node inside the package named **odom_updater** and adding the required dependencies by making necessary additions and modifications to the *cmake.list* and *package.xml* files. After creating the node, a broadcaster script is written in a cpp file which was added to the node. The script contained the algorithm for the broadcaster which is responsible for creating a parent-child association of the Odom and base footprint respectively.

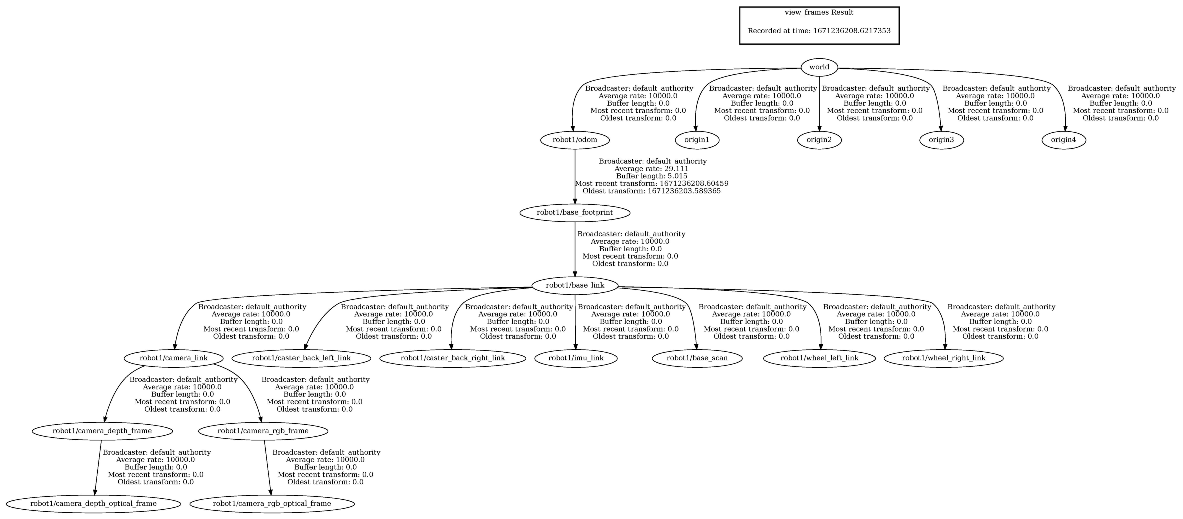

view_frames Result
Recorded at time: 1671236208.6217353
world
Broadcaster: default_authority Average rate: 10000.0 Buffer length: 0.0 Most recent transform: 0.0 Oldest transform: 0.0
Broadcaster: default_authority Average rate: 10000.0 Buffer length: 0.0 Most recent transform: 0.0 Oldest transform: 0.0
Broadcaster: default_authority Average rate: 10000.0 Buffer length: 0.0 Most recent transform: 0.0 Oldest transform: 0.0
Broadcaster: default_authority Average rate: 10000.0 Buffer length: 0.0 Most recent transform: 0.0 Oldest transform: 0.0
Broadcaster: default_authority Average rate: 10000.0 Buffer length: 0.0 Most recent transform: 0.0 Oldest transform: 0.0
robot1/odom
origin1
origin2
origin3
origin4
Broadcaster: default_authority Average rate: 29.111 Buffer length: 5.015 Most recent transform: 1671236208.60459 Oldest transform: 1671236203.589365
robot1/base_footprint
Broadcaster: default_authority Average rate: 10000.0 Buffer length: 0.0 Most recent transform: 0.0 Oldest transform: 0.0
robot1/base_link
Broadcaster: default_authority Average rate: 10000.0 Buffer length: 0.0 Most recent transform: 0.0 Oldest transform: 0.0
Broadcaster: default_authority Average rate: 10000.0 Buffer length: 0.0 Most recent transform: 0.0 Oldest transform: 0.0
Broadcaster: default_authority Average rate: 10000.0 Buffer length: 0.0 Most recent transform: 0.0 Oldest transform: 0.0
Broadcaster: default_authority Average rate: 10000.0 Buffer length: 0.0 Most recent transform: 0.0 Oldest transform: 0.0
Broadcaster: default_authority Average rate: 10000.0 Buffer length: 0.0 Most recent transform: 0.0 Oldest transform: 0.0
Broadcaster: default_authority Average rate: 10000.0 Buffer length: 0.0 Most recent transform: 0.0 Oldest transform: 0.0
Broadcaster: default_authority Average rate: 10000.0 Buffer length: 0.0 Most recent transform: 0.0 Oldest transform: 0.0
robot1/camera_link
robot1/caster_back_left_link
robot1/caster_back_right_link
robot1/imu_link
robot1/base_scan
robot1/wheel_left_link
robot1/wheel_right_link
Broadcaster: default_authority Average rate: 10000.0 Buffer length: 0.0 Most recent transform: 0.0 Oldest transform: 0.0
Broadcaster: default_authority Average rate: 10000.0 Buffer length: 0.0 Most recent transform: 0.0 Oldest transform: 0.0
robot1/camera_depth_frame
robot1/camera_rgb_frame
Broadcaster: default_authority Average rate: 10000.0 Buffer length: 0.0 Most recent transform: 0.0 Oldest transform: 0.0
Broadcaster: default_authority Average rate: 10000.0 Buffer length: 0.0 Most recent transform: 0.0 Oldest transform: 0.0
robot1/camera_depth_optical_frame
robot1/camera_rgb_optical_frame

Figure 1: The connected tree

The following steps were followed to implement the first step of the project:

- A c++ script was created inside the odom updater node and the general format of the tf broadcaster, which was linked to the project report was utilized for the implementations
- Code lines containing information about the parent and the child links were formatted with respect to the requirements of the project (odom and base_footprint)
- The message published by the topic was obtained by running the command ros2 topic info command which will display the message published by the robot1/odom
- The translation coordinates are obtained from the message of the top robot1/odom so that the position of the turtlebot can be obtained as the device moves around the gazebo environment

By writing broadcaster, the tf trees which were disconnected previously get connected at locations robot1/odom and robot1/base_footprint, and as a whole, the required connected tree is obtained. The obtained connected tree image is shown in Figure 1

The pseudo-code for this step is given below:

```
Function handle_turtle_pose in FramePublisher class (input data -
   shared pointer to \robot1\odom topic)
 {
   Parent frame is "/robot1/odom";
   child_frame is "/robot1/base_footprint";

   translation in x is the position of turtlebot in the x-axis of the odom frame;
   translation in y is s position of turtlebot in the y-axis of the odom frame;
   translation  in z is 0.0;

   rotation in x is the orientation of turtlebot in the x-axis of the odom frame;
   rotation in y is the orientation of turtlebot in the y-axis of the odom frame;
   rotation in z is the orientation of the turtlebot in the z-axis of the odom frame;
   rotation in w is the orientation of the turtlebot in the w-axis of the odom frame;
   Broadcast the transformation;
 }
```

STEP 2

After connecting the tf tree, the turtlebot needs to go to a set position from when it can search for the fiducial marker. This step witnesses the utilization of the ‘target reacher package, which is the only package that needs major modification apart from the odom updater in the whole project. The m bot controller function sets the goal of the aruco marker and commands the turtlebot to reach the marker’s location. Upon reaching the first aruco marker, a message gets published to the goal-reached topic and so, it is required to subscribe to this topic to communicate with the turtlebot.

The following procedure involves twisting the robot after reaching the initial goal position so that the robot can identify the aruco marker which is placed arbitrarily around the robot. In order to implement the twist on the robot, a ”twist” message needs to be published on the velocity topic(cmd_vel) of the robot, with an angular velocity of 0.2. Just as the twist message is being published, the robot makes a rotation about its own axis and searches for an aruco marker. It is mandatory to subscribe to the aruco markers topic because the information about the robot detecting the fiducial marker is embedded into the aruco markers topic.

Figure 2 shows the turtlebot reaching the required position to search for the fiducial marker and Figure 3 shows the UML Class Diagram

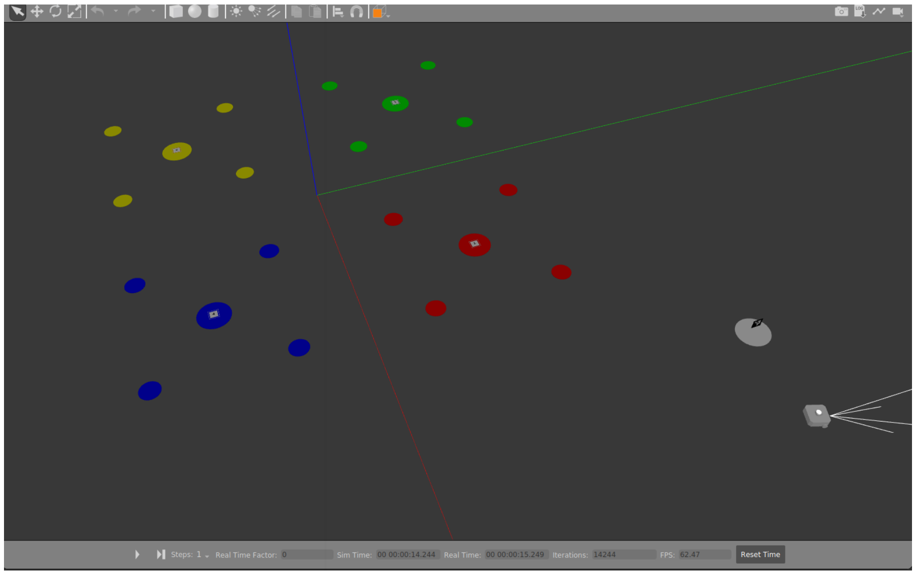


Figure 2: Turtlebot at 1st destination

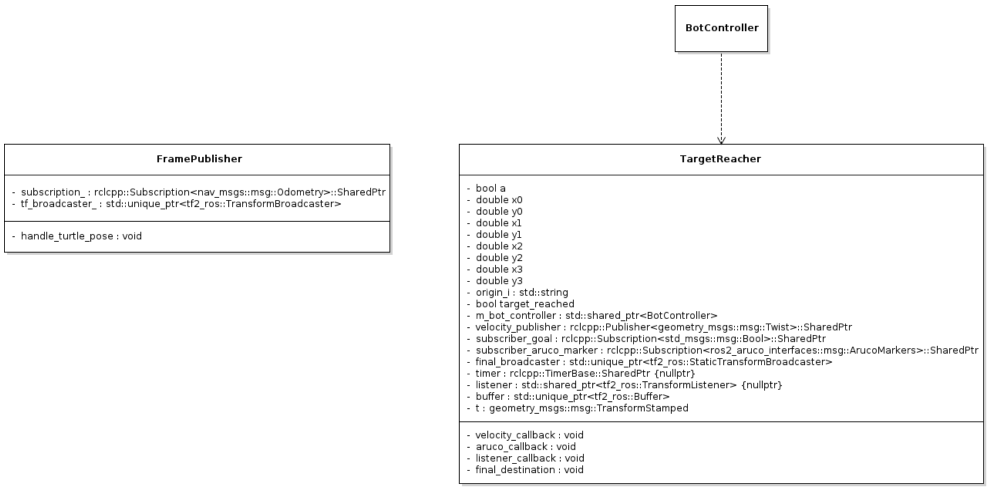

BotController
FramePublisher
- subscription_ : rclcpp::Subscription<nav_msgs::msg::Odometry>::SharedPtr
- tf_broadcaster_ : std::unique_ptr<tf2_ros::TransformBroadcaster>
- handle_turtle_pose : void
TargetReacher
- bool a
- double x0
- double y0
- double x1
- double y1
- double x2
- double y2
- double x3
- double y3
- origin_i : std::string
- bool target_reached
- m_bot_controller : std::shared_ptr<BotController>
- velocity_publisher : rclcpp::Publisher<geometry_msgs::msg::Twist>::SharedPtr
- subscriber_goal : rclcpp::Subscription<std_msgs::msg::Bool>::SharedPtr
- subscriber_aruco_marker : rclcpp::Subscription<ros2_aruco_interfaces::msg::ArucoMarkers>::SharedPtr
- final_broadcaster : std::unique_ptr<tf2_ros::StaticTransformBroadcaster>
- timer : rclcpp::TimerBase::SharedPtr {nullptr}
- listener : std::shared_ptr<tf2_ros::TransformListener> {nullptr}
- buffer : std::unique_ptr<tf2_ros::Buffer>
- t : geometry_msgs::msg::TransformStamped
- velocity_callback : void
- aruco_callback : void
- listener_callback : void
- final_destination : void

Figure 3: UML Class Diagram

The pseudo-code for the 2nd step is given below:

```
 Function velocity_callback in TargetReacher class (input data ->  msg - shared pointer
 to /goal_reached topic, subscribed message type { bool value)
{
    if the bool value is true
    {
        Publish zero linear velocity
        if target_reached is not false
        {
          Publishing angular velocity - 0.2
        }
        else
```

```
                Publishing angular velocity - 0.0

            Publishing the velocity command

             Change the target_reached bool variable to true;
        }
}
```

STEP 3

After locating the fiducial marker, the turtlebot needs to move forward and reach the final destination according to the frame ids in the final_params.yaml file. The topic aruco_markers is subscribed to find the message published and read the field marker_ids in the topic. The field marker_ids is a vector of integers. The elements are accessed using the same procedure as vectors. The field marker_ids contains the information of the final destination corresponding to the frame id in the final_params.yaml. There are four different destinations in one frame id and the destination depends upon the aruco marker displayed.

The pseudo-code for the 3rd step is given below:

```
 Function listener_callback in TargetReacher class (no input)
{
    if the bool value of a is true)
    {
        execute the below

        try the below
        {
            Look up the transformation between \robot1\odom and final_destination frames
            and set a goal for turtlebot to reach the second goal

        Setting the transformed final_destination position as the goal by giving

        the inputs to the \m_bot_controller function
}
```

STEP 4

Once the final destination is subscribed and the marker_ids is retrieved, the goal coordinates should be transformed to robot1 odom frame. The goal coordinates are generally in the frame id and need to be transformed to the odom frame so that the turtlebot moves correspondingly. To transform, a final destination frame is broadcasted in the origin mentioned in the frame ids.

The position of the final destination in the origin mentioned is frame_ids origin is taken from the final_params.yaml file using a pointer. The position of the final destination in the origin_i frame is (final_destination.aruco_i.x, final_destination.aruco_i.y, 0) and the orientation is (0, 0, 0, 1), where i is the integer retrieved from the marker_ids field. A transform listener is used to transform between the final_destination frame and /robot1/odom frame.

The transformed final destination coordinates are used and the goal is given as input in the m_bot_controller function. Once the coordinates are transformed to /robot1/odom frame and the goal is set, the turtlebot reaches the final destination and again publishes the "Goal reached" message in the topic goal_reached as shown in Figure 4. The whole approach is represented in the flow chart as shown in figure 5

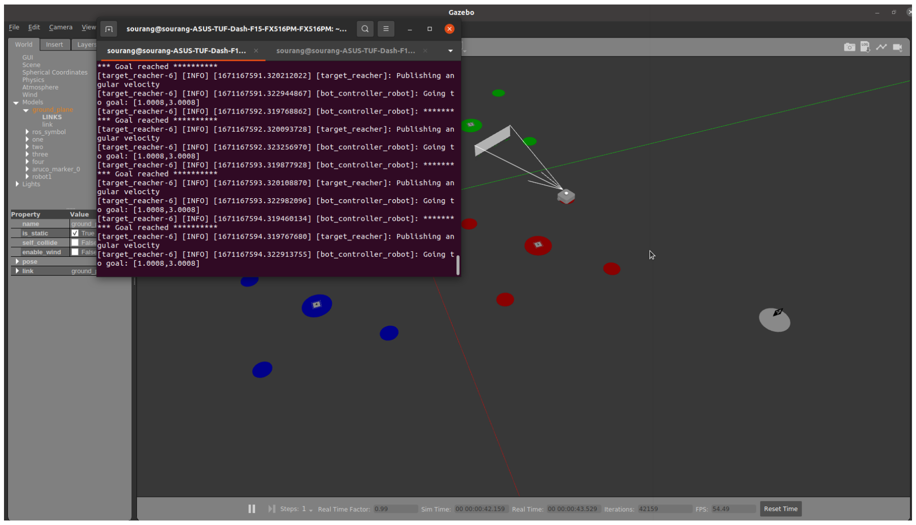
The pseudo-code for the 4th step is given below:

```
 Function for reaching  final position final_destination  in TargetReacher

 class(input integer id)
{
    header frame is origin_i,
    where i is (1,2,3,4)  depending upon the frame_id in the final_params.yaml
    child frame is the final_destination of the turtlebot

    if the value of the marker_id field is equal to 0
    {
    execute the below
    translation in x is the position of the final destination in the x-axis of the header
    frame  is aruco_marker_0 the x-axis coordinate from final_params.yaml
    translation in y is s position of the final destination in the y-axis of  the header
    frame is aruco_marker_0 y-axis coordinate from final_params.yaml
    translation  in z is 0.0;

    rotation in x is the orientation of the final destination in the x-axis of  the header
    frame which is zero;
    rotation in y is the orientation of the final destination in the y-axis of  the header
    frame which is zero;
    rotation in z is the orientation of the final destination in the z-axis of the header
    frame which is zero;
    rotation in w is the orientation of the final destination in the w-axis of  the header
    frame which is one;
    }
    else  if the value of the marker_id field is equal to 1
    {
    Similar to the above if loop;
    }
else  if the value of the marker_id field is equal to 2
    {
    Similar to the above if loop;
    }
else  if the value of the marker_id field is equal to 3
    {
    Similar to the above if loop;
    }
    assign true bool value to ‘a’;
    Broadcast the transformation;
}
```


Gazebo
sourang@sourang-ASUS-TUF-Dash-F15-FX516PM-FX516PM: ~...
sourang@sourang-ASUS-TUF-Dash-F1...
sourang@sourang-ASUS-TUF-Dash-F1...

```
*** Goal reached **********
[target_reacher-6] [INFO] [1671167591.320212022] [target_reacher]: Publishing an
gular velocity
[target_reacher-6] [INFO] [1671167591.322944867] [bot_controller_robot]: Going t
o goal: [1.0008,3.0008]
[target_reacher-6] [INFO] [1671167592.319768862] [bot_controller_robot]: *******
*** Goal reached **********
[target_reacher-6] [INFO] [1671167592.320093728] [target_reacher]: Publishing an
gular velocity
[target_reacher-6] [INFO] [1671167592.323256970] [bot_controller_robot]: Going t
o goal: [1.0008,3.0008]
[target_reacher-6] [INFO] [1671167593.319877928] [bot_controller_robot]: *******
*** Goal reached **********
[target_reacher-6] [INFO] [1671167593.320108870] [target_reacher]: Publishing an
gular velocity
[target_reacher-6] [INFO] [1671167593.322982096] [bot_controller_robot]: Going t
o goal: [1.0008,3.0008]
[target_reacher-6] [INFO] [1671167594.319460134] [bot_controller_robot]: *******
*** Goal reached **********
[target_reacher-6] [INFO] [1671167594.319767680] [target_reacher]: Publishing an
gular velocity
[target_reacher-6] [INFO] [1671167594.322913755] [bot_controller_robot]: Going t
o goal: [1.0008,3.0008]
```

Figure 4: Turtlebot reaching the final_destination

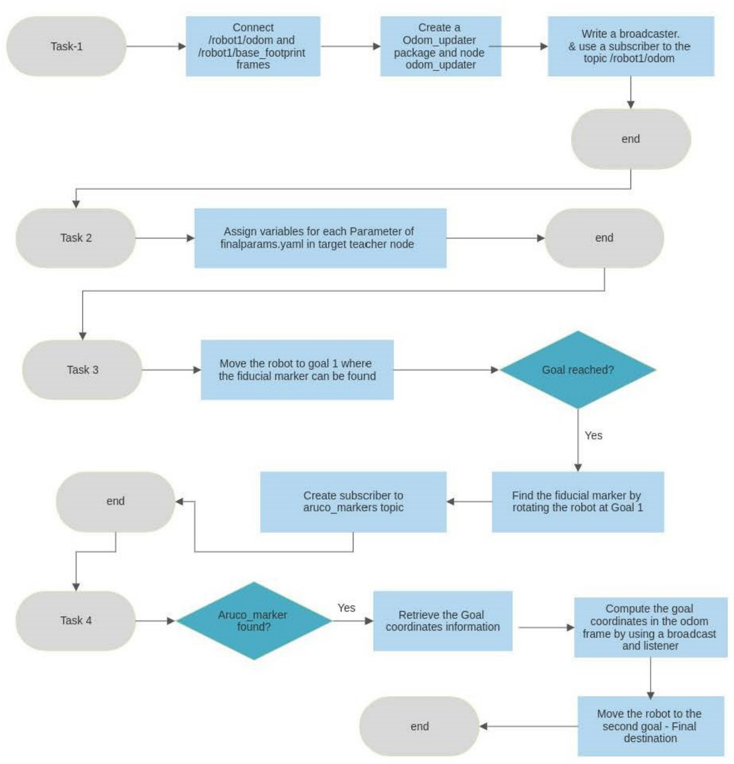

Task-1
Connect /robot1/odom and /robot1/base_footprint frames
Create a Odom_updater package and node odom_updater
Write a broadcaster. & use a subscriber to the topic /robot1/odom
end
Task 2
Assign variables for each Parameter of finalparams.yaml in target teacher node
end
Task 3
Move the robot to goal 1 where the fiducial marker can be found
Goal reached?
Yes
Find the fiducial marker by rotating the robot at Goal 1
Create subscriber to aruco_markers topic
end
Task 4
Aruco_marker found?
Yes
Retrieve the Goal coordinates information
Compute the goal coordinates in the odom frame by using a broadcast and listener
Move the robot to the second goal - Final destination
end

Figure 5: Flow chart for the approach

3 Challenges Faced

1. TURTLEBOT DIDN'T SPAWN IN THE GAZEBO WORLD

After including the ENPM809Y file into the source of the final workspace, while running the command to spawn it in the gazebo, the turtlebot did not spawn and the colcon build frequently failed indicating a few packages were missing. The gazebo had to be uninstalled and this time, all the necessary packages, including a navigator package were installed. The turtlebot eventually spawned in the gazebo environment and the project was initiated.

2. COMPLEXITY WHILE WRITING THE BROADCASTER

The broadcaster syntax was complicated and encountered a number of challenges while adapting the code in accordance with the project's requirements.

- The first mistake was with the names "odom" and "base footprint" which were directly used for the parent and child link respectively instead of mentioning it as **"/robot1/odom"** and **"/robot1/base_footprint"**
- Second error was displayed while trying to include the message type in the broadcaster script. Instead of using a double colon in between the elements, backslashes were written which prevented the code from getting executed (i.e) Instead of using geometry_msgs::msg::Twist and wrote geometry msgs/msg/Twist.
- Another major mistake involved is assuming that the roll pitch yaw needs to be converted to quaternion while the format was already being published in quaternion

This assumption delayed the process of implementation and took a considerable amount of time.

3. ROTATING THE TURTLEBOT AFTER REACHING THE FIDUCIAL MARKER POSITION

While writing the code for the **(this)** pointer an asterisk was added in front of the pointer name (i.e.) (*this) which was then constantly throwing errors and that a considerable amount of time to figure out the cause of the issue. Later the issue was found and then removed the asterisk. Another improper assumption was that the geometry tag in the type of message was replaced by the odometry while geometry was the actual tag for the message type. While defining the parameters in the public class, the program didn't get executed and had to eventually add the parameters in the private class due to which the program was successfully compiled.

4. LISTENER CALLBACK FUNCTION

Instead of using the right command "robot1/odom", a backslash was added to the command ("/robot1/odom) that induced an error in the program. This error was caused due to the error made in the broadcaster where the backslash was required so the backslash was added which was not required for this case.

5. TURTLEBOT ROTATION AFTER REACHING THE FINAL DESTINATION

Upon reaching the final destination in the gazebo environment, the turtlebot started to rotate again searching for an aruco marker. This issue was caused due to the subscriber getting subscribed to the goal_reached publisher again. To fix this issue and make the turtlebot stop rotating once it reaches the final destination, a boolean variable named target_reached was created. This boolean variable was assigned a false value and used in an if-else loop in the velocity_callback frame. The logic was written in such a way that if the loop condition will be executed only if the variable is equal to false, then the turtlebot would stop rotating at the correct location, in this case, the final_destination.

4 Final Results

This section contains images of the turtlebot reaching various final destinations in different origins.

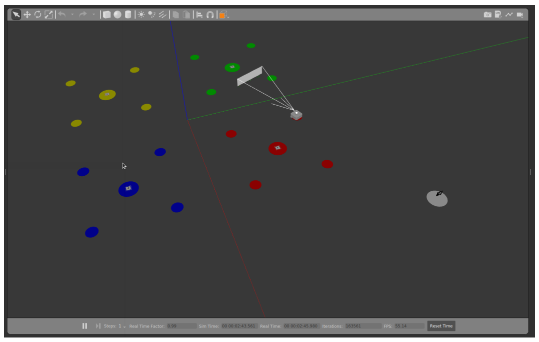


Figure 6

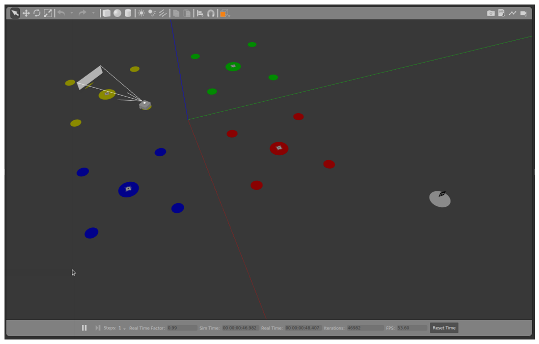


Figure 7

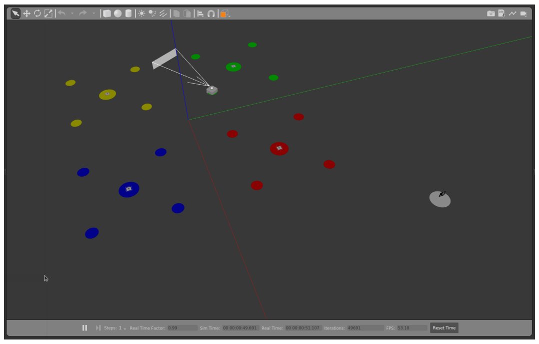


Figure 8

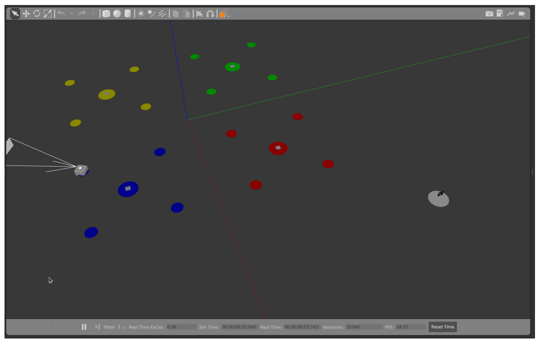
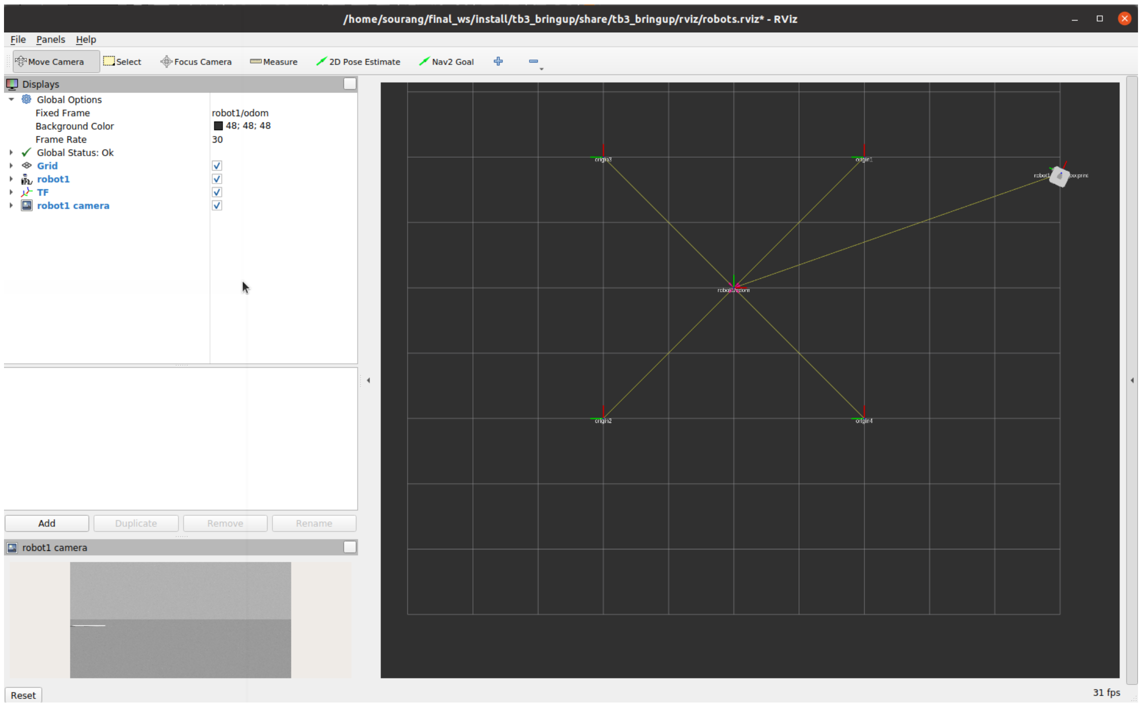


Figure 9

- Figure 6 shows the Turtlebot in the Final destination - Origin 1 and Aruco_marker 0
- Figure 7 shows the Turtlebot in the Final destination - Origin 2 and Aruco_marker 1
- Figure 8 shows the Turtlebot in the Final destination - Origin 3 and Aruco_marker 2
- Figure 9 shows the Turtlebot in the Final destination - Origin 4 and Aruco_marker 3

The turtlebot moving around the gazebo world is visualized in the RViz software in Figure 10 and the final connected tree after connecting odom and base footprint frames is shown in Figure 11



/home/sourang/final_ws/install/tb3_bringup/share/tb3_bringup/rviz/robots.rviz* - RViz

Displays
- Global Options
  - Fixed Frame: robot1/odom
  - Background Color: 48; 48; 48
  - Frame Rate: 30
- Global Status: Ok
- Grid ✓
- robot1 ✓
- TF ✓
- robot1 camera ✓

robot1 camera

Figure 10: Final connected tree after connecting odom and base footprint frames

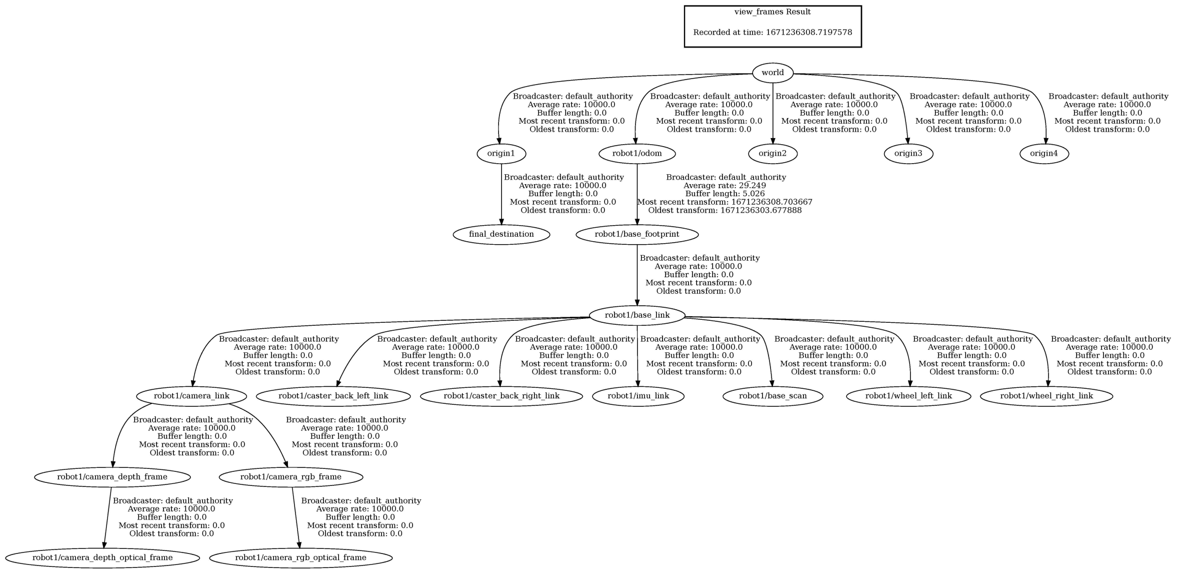

view_frames Result
Recorded at time: 1671236308.7197578
world
origin1
robot1/odom
origin2
origin3
origin4
final_destination
robot1/base_footprint
robot1/base_link
robot1/camera_link
robot1/caster_back_left_link
robot1/caster_back_right_link
robot1/imu_link
robot1/base_scan
robot1/wheel_left_link
robot1/wheel_right_link
robot1/camera_depth_frame
robot1/camera_rgb_frame
robot1/camera_depth_optical_frame
robot1/camera_rgb_optical_frame
Broadcaster: default_authority
Average rate: 10000.0
Buffer length: 0.0
Most recent transform: 0.0
Oldest transform: 0.0
Broadcaster: default_authority
Average rate: 29.249
Buffer length: 5.026
Most recent transform: 1671236308.703667
Oldest transform: 1671236303.677888

Figure 11: Final connected tree after reaching the final_destination

5 Contributions

Contributions of each member			
Tasks	**Sandip**	**Sourang**	**Sandeep**
1	✓	✓	✓
2	✓	✓	✓
3	✓	✓	✓
4	✓	✓	✓
Report	✓	✓	✓

6 Resources

- Writing the Broadcaster - Click_here
- Listener -Click_here
- Velocity_callback - Referred from the given botcontroller.cpp file

7 Course Feedback

The course Intro to Robot programming has been immensely helpful in learning C++ and ROS 2 from absolute scratch. Despite the complicated concepts including object-oriented programming and pointers, with the help of Prof. Dr. Zeid Kootbally who has been extremely helpful and clear in terms of explanation, a considerable amount of confidence has been built up in the respective concepts. The quizzes conducted every alternate week ensured thorough revision of every concept and the project helped in the implementation of these concepts that enabled profound learning.

"It's not the subject itself, but the professor who can make a subject interesting and with complete satisfaction, one can claim that the learning outcome and satisfactory component was completely fulfilled."