

Introduction to ROS - ss25

Autonomous Driving

Team 9

I. Introduction

In this project, the task is to travel a car along the predetermined route as quickly as possible within a simulated urban environment. At the same time, one must ensure not to leave the road, not to collide with other vehicles, and to follow traffic signals. Figure 1.1 shows the top view of the simulated urban environment, as well as the predefined route and waypoints.

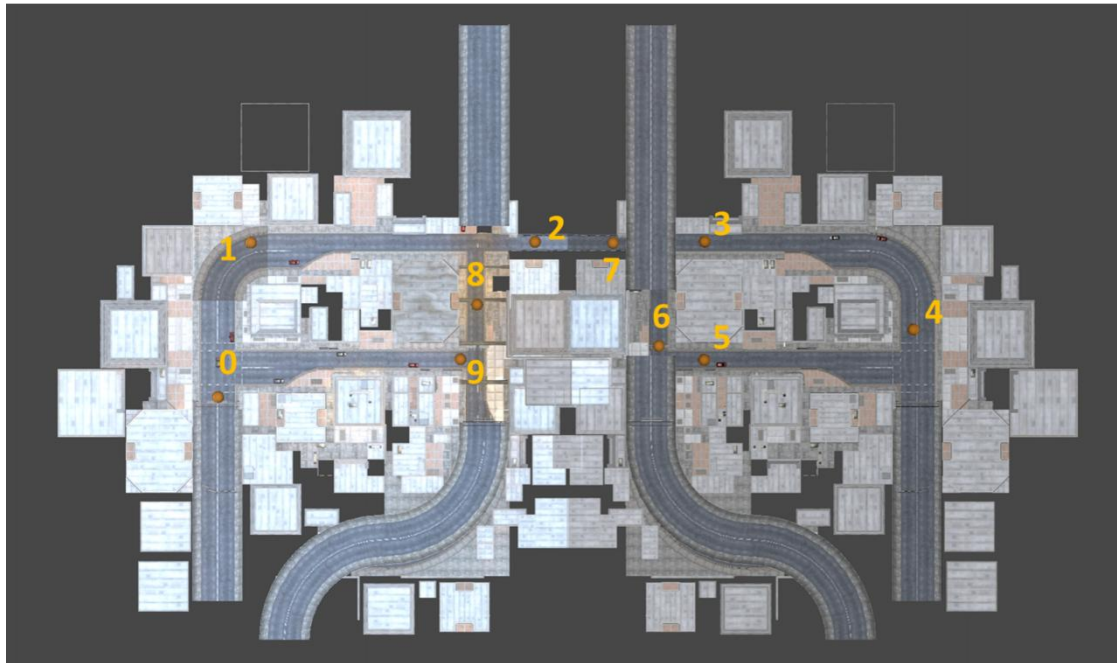


Figure 1.1: Predefined track through the urban environment.

II. Perception Pipeline

Before implementing the perception pipeline, we need to first utilize the tf2 package to define the tf transformations for all coordinate systems as per the project introduction file. This is because the octomap and costmap that will be used later are generated based on the corresponding coordinate systems. The complete tf tree is shown in Figure 2.1. The function of the tf tree “**world - INS - Center - SensorBase - 4Cameras**” is to provide the coordinate system and tf transformation corresponding

to each camera. The function of this tf tree, **“world - odom - base_link - rear_axle_link”**, is to provide the corresponding coordinate systems and tf transformations for the move_base package to implement path&trajectory planning. The origin of the rear_axle_link coordinate system is set at the center of the vehicle's rear axle, which complies with the requirements of the TEB local planner for the coordinate system. However, later during the debugging of the path planning module by other team members, the coordinate frame used to follow the car's movement was changed to OurCar/Center, which also yielded good results. As a result, in the TF tree **“world - odom - base_link - rear_axle_link”**, only world and odom are still actively used. The base_link and rear_axle_link coordinates were retained merely as alternative frames to be chosen and does not serve any practical function in the final submitted code.

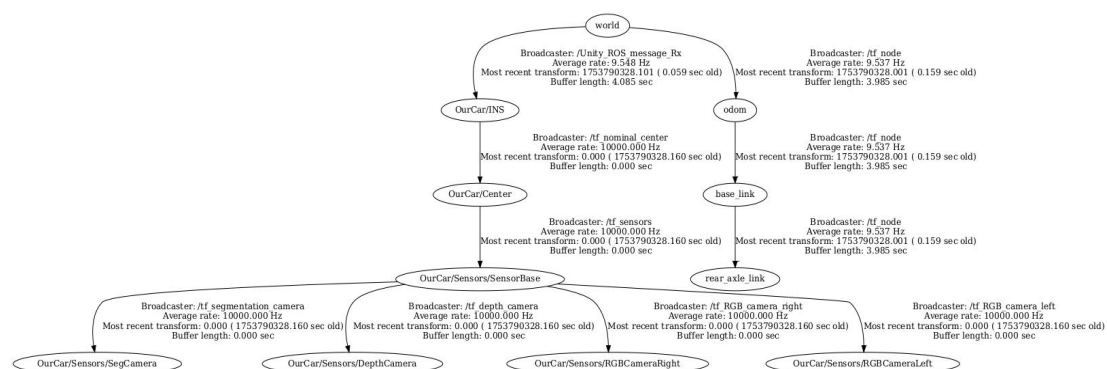


Figure 2.1: TF tree

After defining all the tf coordinate systems, we need to use the existing depth camera to generate point clouds through the depth_image_proc package. The point cloud image is shown in Figure 2.2. Then, we will use the obtained point cloud as input to generate the octomap. When the octomap is ready, it will automatically output many ROS topics, such as /projected_map, which is an important reference for subsequent path planning, and /occupied_cells_vis_array, which is a three-dimensional Occupancy Map and can be directly visualized in rviz. The visualized octomap (2D projected map + 3D occupancy map) can be found in Figure 2.3.

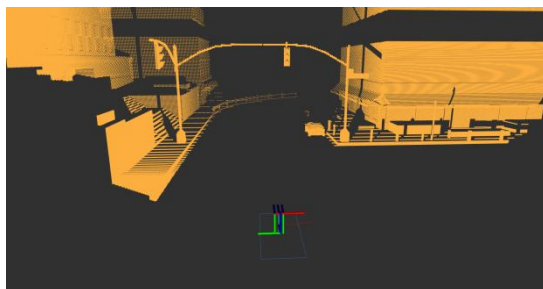


Figure 2.2: Point cloud

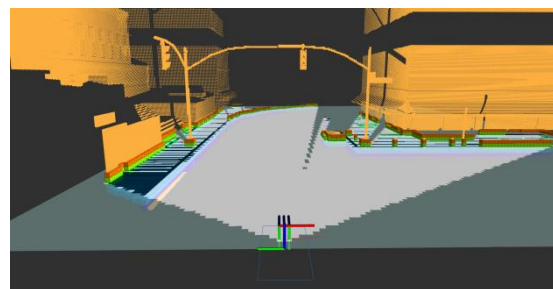


Figure 2.3: Octomap (2D+3D)

Once the octomap is completed, the costmap needs to be generated. The costmap is generated based on the `move_base` package and is also an important component of path planning. On one hand, we need the original point cloud to generate the obstacle layer of the costmap, which is used to detect the situation of obstacles. On the other hand, we also need the octomap to assist in generating the static layer, which is used for basic path planning. In addition, we have customized the inflation layer, which is used to virtually expand the shapes of all obstacles. Although this will to some extent increase the difficulty of trajectory planning, this buffer distance can effectively improve the probability of successfully avoiding obstacles, making driving safer. The visualized images of the global and local costmaps are shown in Figure 2.4.

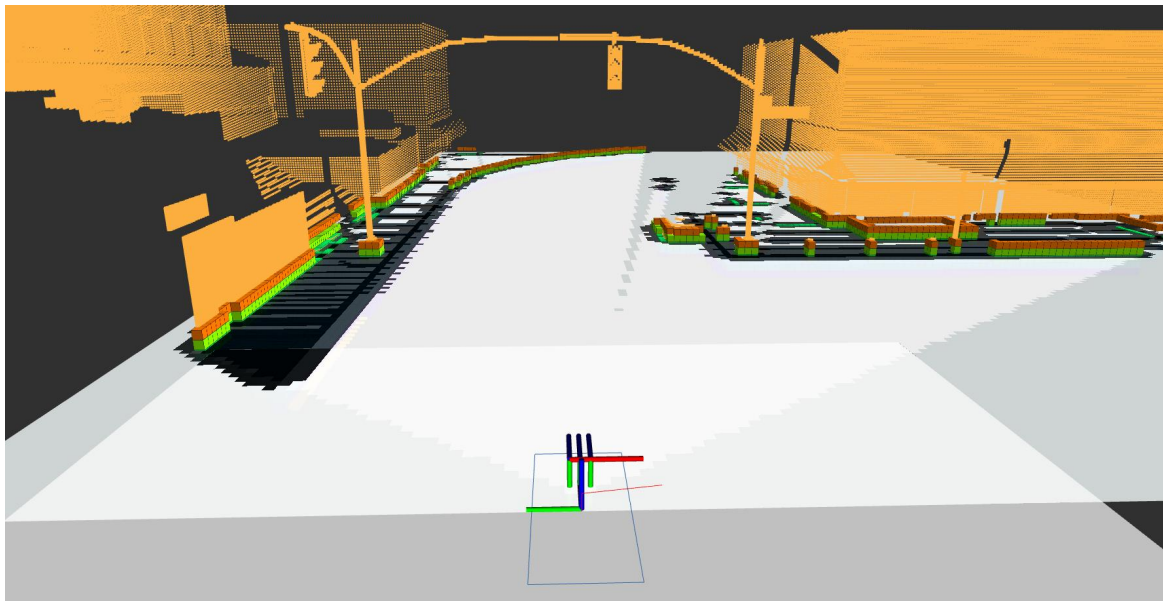


Figure 2.4: Costmap (global + local, with point cloud and Octomap)

Furthermore, since the vehicles in this project need to be capable of complying with traffic signals, the semantic segmentation function is necessary. However, since we abandoned the semantic camera and instead used the RGB image + semantic segmentation neural network model to recognize traffic signals, we will not discuss it in detail in the perception pipeline for now. Instead, we will elaborate on it in the subsequent chapter.

III. Perception_YOLO: Traffic Light Detection

To enable the autonomous vehicle to follow basic traffic regulations, such as stopping at red lights, we implemented the `perception_yolo` module. This module is responsible for detecting traffic lights in the scene and determining whether the vehicle should stop or proceed. Although this function can be structured with traditional methods that rely on semantic cameras, we utilize a standard RGB camera combined with a YOLO-based object detection network, using pure image-vision algorithms that make

the system more adaptable and realistic for real-world applications. The reason why YOLO was selected as the recognition model is: this is a well-trained and relatively reasonable small that can be implemented in the project. Although the YOLO model is more widely used in real-world recognition, still it's better than heavy giant models such as Apollo.

The first step in this pipeline is model loading and image subscription. The detection model is a YOLOv5 network with per-trained parameters, because of the lack of training data and validation set. Training a custom dataset of traffic light images, with special attention given to detecting three-bulb vertical traffic lights, became hardly possible. The trained models (with a training set of 100/500 images) performed also unsatisfied.

As shown in Figure 3.1, the RGB images are obtained in real-time from the front-left mounted camera. These images are then passed through the YOLO model to detect bounding boxes of traffic lights. However, simple detection is not sufficient—hence, a post-processing step is performed using OpenCV to analyze the detected traffic lights and determine the active light color. Specifically, we apply HSV color thresholding and blob analysis to determine if the topmost light in the bounding box is a red circular light, which satisfies our stop condition.

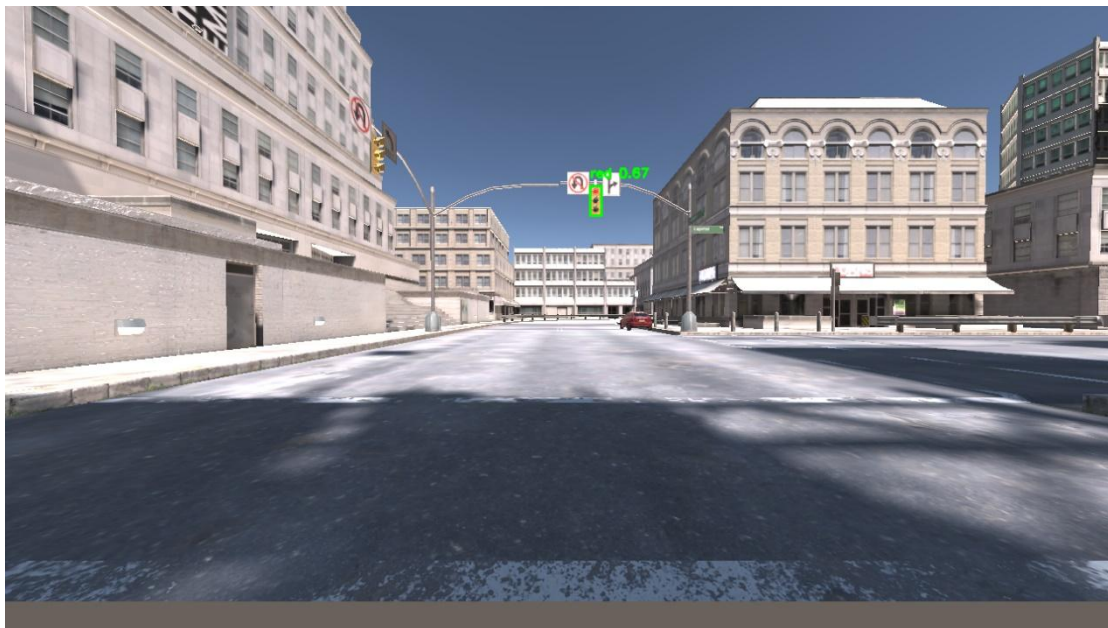


Figure 3.1: Traffic Light Detection with YOLO and Color Analysis

Once a valid red light is detected, a “TrafficLightState” message is published to a dedicated ROS topic `/perception/yolo/traffic_light`, containing both the state (“red”, “not_red”). This topic “traffic_light” is subscribed to by the controller module, which integrates it into the overall vehicle behavior. The controller module only cares if the traffic light is red. If the state is “red”, the vehicle will stop at the intersection, all other situations won’t cause any change of movement.

With all the methods and strategies mentioned, there are still a few problems. To improve reliability, we tried to incorporate a suppression mechanism to avoid erratic

behavior due to detection noise. Once a "red" signal is detected, the system enforces a short suppression period (e.g., 5 seconds) during which the state remains "red" even if transient false "not_red" detections occur. This mechanism was meant to help maintain smooth decision-making and prevents unnecessary stop-and-go actions, but the performance was not good enough. Because of the low resolution of Unity-Simulation, the YOLO model cannot always detect the right traffic light. On the other hand, increasing resolution caused significant function decreasing of the virtual machine.

Mis-identification cannot be avoided due to low resolution. If the traffic light recognition module can be implemented in a better simulation platform (e.g. Carla), the results would be much greater.

In conclusion, the perception_yolo module plays a key role in the perception stack by replacing semantic segmentation with a deep learning-based detection and simply decision logic. It enables the vehicle to comply with traffic light signals in a robust and smooth way.

IV. Path&Trajectory Planning

The navigation pipeline starts by generating waypoint trajectories using cubic Bézier curves to ensure smooth and feasible transitions between checkpoints. Each curve is defined by four control points: P_0 and P_3 set the start and end positions, while $\overrightarrow{P_0P_1}$ and $\overrightarrow{P_2P_3}$ determine direction vectors at the start and end.

$$B(t) = (1 - t)^3 P_0 + 3(1 - t)^2 t P_1 + 3(1 - t) t^2 P_2 + t^3 P_3$$

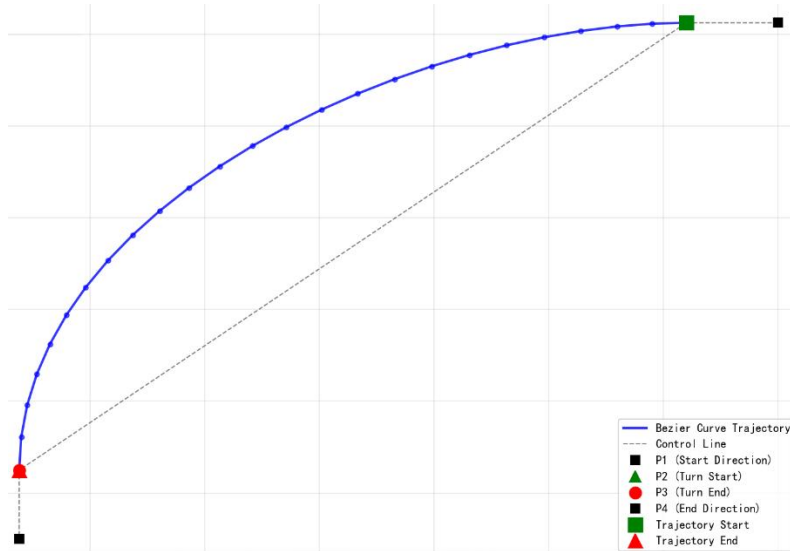


Figure 4.1: A demonstration of cubic Bézier curves

These curves provide smooth acceleration profiles and continuous curvature by determining control points from directional vectors derived from surrounding

waypoints, with the system adjusting intermediate points to maintain smoothness and ensure the vehicle can navigate each segment within its steering constraints.

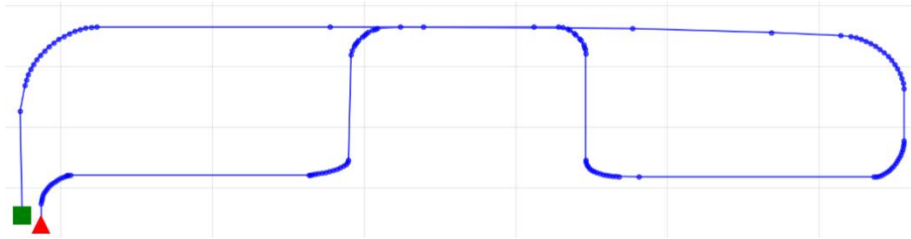


Figure 4.2: A plot of all waypoints

Waypoints are precomputed and stored in YAML files. The `loadWaypoints()` function reads these sequences and calculates orientation vectors for each waypoint. The system publishes `MoveBaseGoal` messages to the `move_base` action server, guiding the robot through defined goals. The current vehicle pose is tracked using the message from `/Unity_ROS_message_Rx/OurCar/CoM/pose`. The system constantly checks curvature to ensure kinematic feasibility and adjusts target points when necessary.

The `move_base` node handles path execution using two costmaps—global costmap and local costmap. These costmaps are built with layered inputs like obstacle detection, inflation zones for safety, and optionally static maps.

Navigation is supported by error recovery mechanisms in both the waypoint navigator and `move_base`. These include costmap clearing, rotational maneuvers, slow traversal through complex terrain, and timeouts to avoid getting stuck. The navigator also supports looped routes and returning to the starting point after completing a mission.

Throughout the navigation process, the system generates comprehensive visualization data through marker arrays published to `"/waypoint_markers"` and planned path information to `"/planned_path"`, enabling real-time mission monitoring. The pipeline culminates in velocity command generation through `"/cmd_vel"`, where `move_base` publishes Twist messages specifying desired linear and angular velocities that represent the transformation of high-level mission objectives into concrete motion directives.

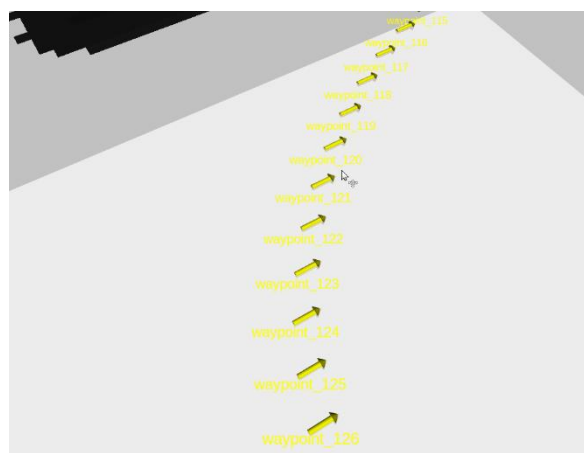


Figure 4.3: Visualization of waypoints

Global Planning

The global planner is responsible for generating a coarse, obstacle-free path from the robot's current position to the target goal using a 2D costmap derived from Octomap projections. The Global Planner plugin was configured with:

- **Dijkstra's algorithm** (use_dijkstra: true) for optimal cost-based search
- **Quadratic distance heuristics** (use_quadratic: true) to improve path smoothness
- **Unknown space handling** (allow_unknown: true) to allow planning through unmapped areas

This setup enabled reliable global path computation even in scenarios where parts of the environment were unknown or only partially observed.

Local Planning with TEB

For local planning, we utilized the **TEB (Timed Elastic Band)** algorithm through the `teb_local_planner`. This local planner performs real-time trajectory optimization while accounting for robot kinematics, obstacle avoidance, and time efficiency. Key configurations included:

- **Velocity and acceleration limits** for safe and feasible control (e.g., `max_vel_x`, `acc_lim_x`)
- **Goal tolerances** (`xy_goal_tolerance`, `yaw_goal_tolerance`) to ensure precise docking
- **Multi-objective optimization** with adjustable weights (`weight_obstacle`, `weight_kinematics_nh`, `weight_optimaltime`, etc.)
- **Kinematic constraints** like minimum turning radius for non-holonomic motion

The TEB planner continuously optimized the local trajectory based on these parameters and allowed the robot to smoothly track the global path while reacting to changes in the environment.

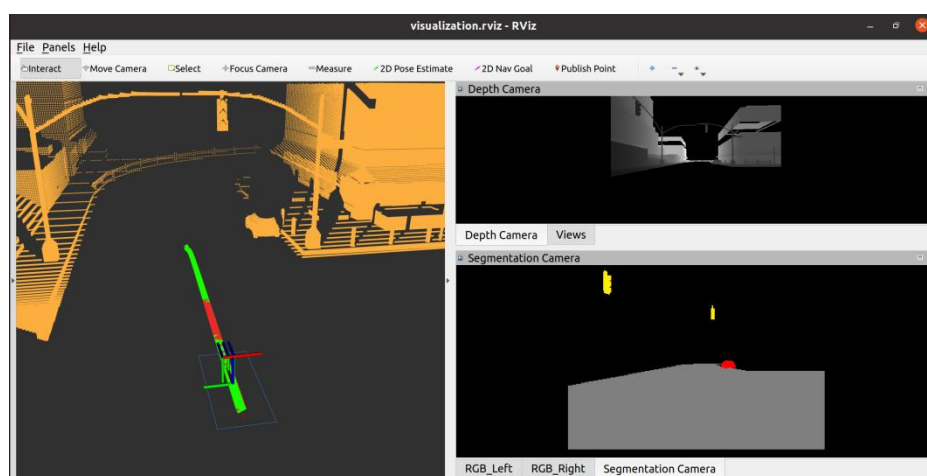


Figure 4.4 local planner TEB

V. Controller

The **controller_node** is the vehicle's low-level closed-loop control program. Running at 100 Hz, it fuses three ROS topics: the real-time odometry `/forward_state`, the linear- and angular-velocity commands `/cmd_vel` generated by the local planner (Teb within `move_base`), and the traffic-light state published by the YOLO perception module on `/perception/yolo/traffic_light`. From these inputs it realizes three key functions—longitudinal PID speed regulation, lateral Ackermann steering, and safe red-light stops. The node first converts `/cmd_vel` into target forward speed and steering angle, limiting both by the vehicle's physical constraints. It then compares the target speed with the actual forward speed, applies a tunable PID controller (kp, ki, kd) to obtain the desired acceleration, and distributes this acceleration into throttle or brake commands. Together with the normalized steering command, these values are packed into a `simulation/VehicleControl` message and sent on `/car_command`, thereby driving the simulated vehicle. When a red light is detected, the node immediately issues full braking, ignores incoming `/cmd_vel` for at least three seconds, and resumes only after a subsequent green signal. If it detects that the current velocity and desired velocity have opposite signs it likewise prioritizes braking to ensure safety. All physical limits and PID gains are read from the ROS parameter server, so they can be adjusted at run-time without recompiling.

The **StateForwarder** node merges the vehicle's center-of-mass pose (`/Unity_ROS_message_Rx/OurCar/CoM/pose``) and twist (`/Unity_ROS_message_Rx/OurCar/CoM/twist``) from the Unity simulation, repackages them into a standard ``nav_msgs/Odometry`` message, and publishes the result on ``/forward_state``, providing downstream **controller_node** with a unified, continuous vehicle-state stream.

VI. Result and Discussion

Note: Since the TEB local planner's moving coordinate frame is defined at the center of the vehicle chassis (OurCar/Center) and behind the depth camera, this means that the origin of the vehicle's motion frame initially lies outside the costmap. As a result, path planning cannot begin immediately. Therefore, after launching the system as described in the README, please open Unity and manually drive the car forward by about 1 meter so that the rear axle is within the costmap range. After this, `move_base` will automatically take over the vehicle's movement.

The goals that we have successfully achieved so far:

- ✓ TF Transformation
- ✓ Perception Pipeline (Point clouds, Octomap and Costmap)
- ✓ Path&Trajectory planning (The car starts from point 0 and completes a lap according to the planned route.)

- ✓ Avoiding static obstacles
- ✓ Avoiding moving cars
- ✓ Car controller
- ✓ Car start/stop at Traffic lights
- ✓ Implement an own message type (In the perception_yolo part)
- ✓ Not using semantic camera

VII. Work Distribution

Original planned task allocation of our group is like this:

- Zetong Zhang: TF transformation and perception pipeline.
- Zhiheng Quan: Path planning.
- Shixuan Fang: Trajectory planning.
- Jiayu Zhao: Traffic lights detection.
- Sizhe Fan: Controller and help with Traffic lights detection.

However, due to various reasons, the actual final contributions of each team member to the project are as follows:

- Zetong Zhang: TF transformation and perception pipeline, part of Path & Trajectory Planning debugging tasks, maintain the remote GitLab repository, generate and integrate all of the project documents.
- Zhiheng Quan: waypoint_navigator node, usage of actionlib, part of Path & Trajectory Planning debugging tasks.
- Shixuan Fang: Initialize two .yaml files for GP and TEB planner with untuned parameters.
- Jiayu Zhao: Traffic lights detection (Yolo5 model).
- Sizhe Fan: Implement the controller and forward_state node, help with Traffic lights detection, part of Path & Trajectory Planning debugging tasks.

Regarding the project development process:

For a long period after the initial work distribution, only Zetong Zhang, Jiayu Zhao, and Sizhe Fan were actively updating the code in GitLab. Zhiheng Quan uploaded the first code related to path planning on July 22nd. Moreover, the initial path planning code he uploaded was not compatible with the rest of the GitLab repository and could not be launched correctly. Zetong Zhang and Sizhe Fan, who were originally in charge of other modules, worked overtime for a week to integrate and debug Zhiheng Quan's code, enabling the Path & Trajectory Planning module to function properly. After their debugging, this vehicle was able to autonomously reach the third turning point (located between the preset points 6 and 5). On July 31st, Zhiheng Quan updated the

path planning method by integrating the actionlib package to assist with planning, which significantly improved the car’s driving performance. Based on this, Zetong Zhang and Sizhe Fan further tuned the parameters, and the car was eventually able to complete the entire course successfully on August 1st.

The traffic light module was mainly handled by Jiayu Zhao, with assistance from Sizhe Fan. Currently, the car can correctly stop and start at traffic lights.

Shixuan Fang was originally assigned to work on trajectory planning. However, his actual contribution was limited to uploading two .yaml files for the GP and TEB planners, both with untuned parameters. In fact, his task was carried out collectively by Zetong Zhang, Sizhe Fan, and Zhiheng Quan.

VIII. Bibliography

- [1] ROS installation: <https://wiki.ros.org/noetic/Installation/Ubuntu> [Accessed: 1-Jul-2025].
- [2] Octomap: https://wiki.ros.org/octomap_server [Accessed: 1-Jul-2025].
- [3] Move_base: https://wiki.ros.org/move_base [Accessed: 1-Jul-2025].
- [4] Yolo5: <https://github.com/ultralytics/yolov5> [Accessed: 10-Jul-2025].
- [5] Actionlib: <https://github.com/ros/actionlib> [Accessed: 30-Jul-2025].

IX. Ros graph

You can find high quality images of the original ROS graph and the annotated ROS graph with team member responsibilities respectively in the “documents” folder.

Original ROS graph:

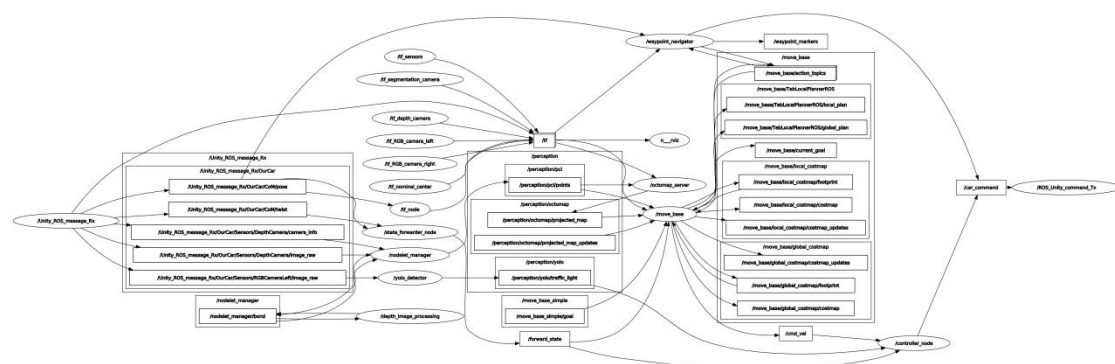


Figure 9.1: Original ROS graph

Annotated ROS graph:

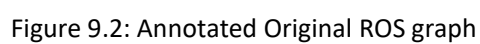


Figure 9.2: Annotated Original ROS graph