Introduction to ROS Project — Group 7

Team member:
Hao Tian,
Youran Wang,
Yihang Xu,
Yiping Zhou,
Zhiyu Zhao

Perception	Yihang Xu
Planning	Zhiyu Zhao, Youran Wang
Decision making	Youran Wang, Yiping Zhou
Control	Youran Wang,Hao Tian
Overall structure and other documentations	Zhiyu Zhao,Hao Tian,Yiping Zhou

Contents

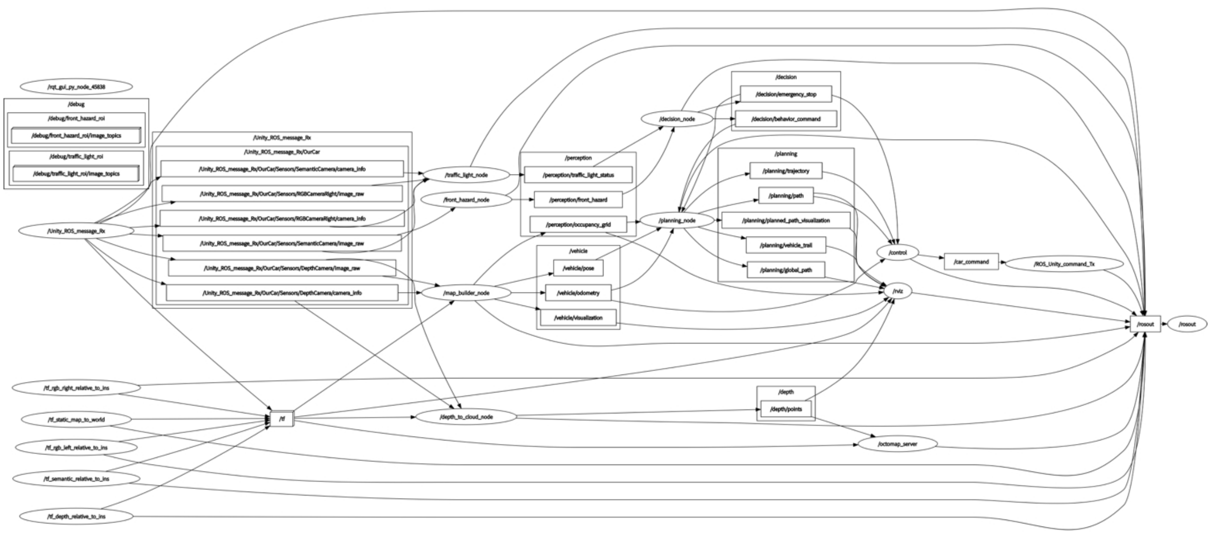
1 Introduction to repository structure

Link to repository:
https://gitlab.lrz.de/zhiyu_zhao/i2ros_project/-/tree/develop?ref_type=heads

Repository Structure:

```
i2ros_project/
└── src/
 |     ├── perception/
 |     ├── planning/
 |     ├── decision_making/
 |     ├── control/
 |     ├── dummy_controller/
 |     ├── msg_interfaces/
 |     ├── start/
 |     |   └── launch/
 |     |     └── all.launch
 |     ├── CMakeList.txt
 |     ├── setup_script.sh
├── devel/
├── build/
├── logs/
├── .gitignore/
└── README.md
```

2 rqt graph



/rqt_gui_py_node_45838
/debug
/debug/front_hazard_roi
/debug/front_hazard_roi/image_topics
/debug/traffic_light_roi
/debug/traffic_light_roi/image_topics
/Unity_ROS_message_Rx
/Unity_ROS_message_Rx
/Unity_ROS_message_Rx/OurCar
/Unity_ROS_message_Rx/OurCar/Sensors/SemanticCamera/camera_info
/Unity_ROS_message_Rx/OurCar/Sensors/RGBCameraRight/image_raw
/Unity_ROS_message_Rx/OurCar/Sensors/RGBCameraRight/camera_info
/Unity_ROS_message_Rx/OurCar/Sensors/SemanticCamera/image_raw
/Unity_ROS_message_Rx/OurCar/Sensors/DepthCamera/image_raw
/Unity_ROS_message_Rx/OurCar/Sensors/DepthCamera/camera_info
/traffic_light_node
/front_hazard_node
/map_builder_node
/perception
/perception/traffic_light_status
/perception/front_hazard
/perception/occupancy_grid
/vehicle
/vehicle/pose
/vehicle/odometry
/vehicle/visualization
/decision_node
/decision
/decision/emergency_stop
/decision/behavior_command
/planning_node
/planning
/planning/trajectory
/planning/path
/planning/planned_path_visualization
/planning/vehicle_trail
/planning/global_path
/control
/car_command
/ROS_Unity_command_Tx
/rviz
/rosout
/rosout
/tf_rgb_right_relative_to_ins
/tf_static_map_to_world
/tf_rgb_left_relative_to_ins
/tf_semantic_relative_to_ins
/tf_depth_relative_to_ins
/tf
/depth_to_cloud_node
/depth
/depth/points
/octomap_server

Figure1. rqt graph

3 Tasks division

3.1 Perception

3.1.1 Overview

The **perception_node** group is responsible for interpreting sensor data and building an environmental understanding essential for navigation and planning. It includes modules for 3D obstacle detection, occupancy grid generation, and traffic light recognition.

The perception pipeline processes inputs from depth, RGB, and semantic cameras. It publishes a real-time local occupancy map and interprets semantic information to detect traffic signals. All perception outputs are structured to integrate seamlessly with planning and control modules.

3.1.2 Role in the System

The **perception system** provides a comprehensive and up-to-date representation of the surrounding environment to support decision-making and safe trajectory planning.

1 Input

Sensor data from the Unity simulator, including:

- Depth images
- RGB images
- Semantic segmentation images
- Odometry data from the INS

2 Processing

- **3D Point Cloud Generation**: Depth images are converted into 3D point clouds using intrinsic parameters. The resulting points are then transformed into the world or vehicle coordinate frame using tf2.
- **Occupancy Grid Construction**: Points higher than a given threshold (e.g., >2 meters) are filtered out to ignore irrelevant overhead objects. A local 2D sliding window occupancy grid is generated by accumulating the transformed point clouds over time, allowing consistent environment modeling even with temporary occlusions.
- **Traffic Light Detection**: Red and green lights are detected by combining semantic segmentation masks with RGB image-based HSV filtering, improving robustness under varying lighting and visibility conditions.
- **Front Vehicle Hazard Detection**: The semantic camera is also used to detect nearby vehicles in front of the ego car. Their angular position relative to the centerline is analyzed, and potential collision risks (frontal hazard) are identified based on proximity and heading.

3 Output

- **/perception/occupancy_grid**: A ROS OccupancyGrid representing the local environment for use by the planner.
- **/perception/traffic_light_status**: A ROS String message indicating the current traffic light state: "red"and "green".
- **/perception/front_hazard_status**:A ROS message indicating whether a frontal collision hazard exists, along with the estimated angle of the obstructing vehicle.

3.1.3 Subscribed and Published Topics

The Subscribed and Published topic of **perception_node** showed as following tables:

Table1. Subscribed Topics

Topic Name	Message Type	Description	Purpose
/Unity_ROS_message_Rx/OurCar/Sensors/DepthCamera/image_raw	**sensor_msgs/Image**	Raw depth image from Unity simulation	Used to generate 3D point cloud
/Unity_ROS_message_Rx/OurCar/Sensors/DepthCamera/camera_info	**sensor_msgs/CameraInfo**	Intrinsic parameters of the depth camera	Required for depth projection
/Unity_ROS_message_Rx/OurCar/Sensors/SemanticCamera/image_raw	**sensor_msgs/Image**	Semantic segmentation image	Used for color-based traffic light detection
/Unity_ROS_message_Rx/OurCar/Sensors/Camera/image_raw	**sensor_msgs/Image**	RGB image	Used to refine pixel localization and visualize bounding boxes
/tf	**tf2_msgs/TFMessage**	Transformation tree	Used to transform points into world or vehicle frame

Table2. Published Topics

Topic Name	Message Type	Used By /	Purpose

/perception/occupancy_grid	**nav_msgs/OccupancyGrid**	**/planning_node**	Consumed by the path planner for collision-free local navigation.
/perception/traffic_light_status	**std_msgs/Stringl**	**/decision_node**	Used by decision-making module or state machine for intersection behavior.
/perception/front_hazard	**std_msgs/Bool**	**/decision_node**	Triggers emergency slow-down or stop logic in decision module.
/perception/front_hazard_angle	**std_msgs/Float64**	**/decision_node**	Can be visualized or used to evaluate hazard position (e.g., for lateral avoidance planning).

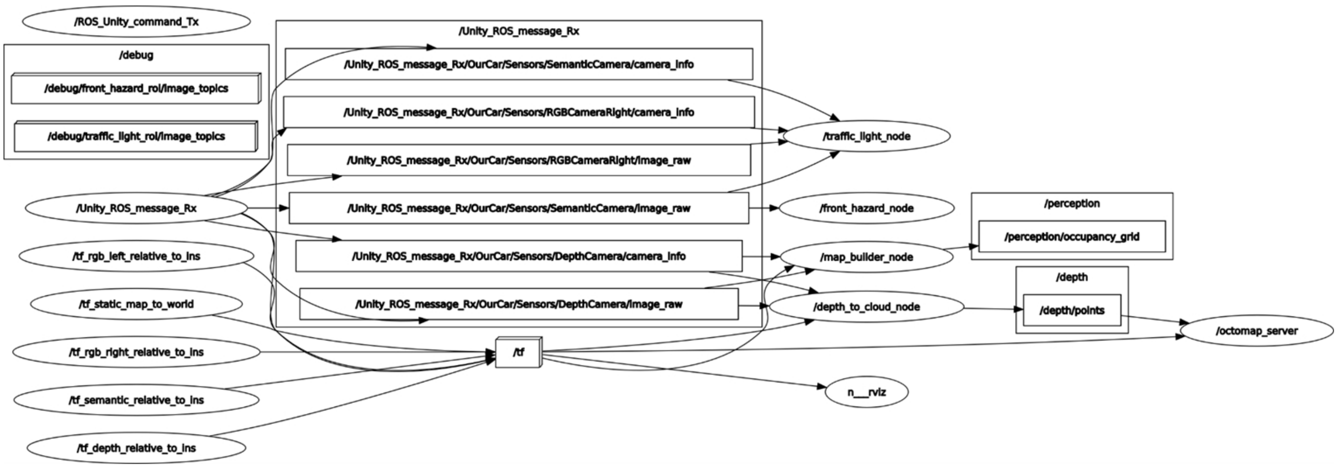

/ROS_Unity_command_Tx
/debug
/debug/front_hazard_roi/image_topics
/debug/traffic_light_roi/image_topics
/Unity_ROS_message_Rx
/Unity_ROS_message_Rx/OurCar/Sensors/SemanticCamera/camera_info
/Unity_ROS_message_Rx/OurCar/Sensors/RGBCameraRight/camera_info
/Unity_ROS_message_Rx/OurCar/Sensors/RGBCameraRight/image_raw
/Unity_ROS_message_Rx/OurCar/Sensors/SemanticCamera/image_raw
/Unity_ROS_message_Rx/OurCar/Sensors/DepthCamera/camera_info
/Unity_ROS_message_Rx/OurCar/Sensors/DepthCamera/image_raw
/Unity_ROS_message_Rx
/tf_rgb_left_relative_to_ins
/tf_static_map_to_world
/tf_rgb_right_relative_to_ins
/tf_semantic_relative_to_ins
/tf_depth_relative_to_ins
/tf
/traffic_light_node
/front_hazard_node
/map_builder_node
/depth_to_cloud_node
/perception
/perception/occupancy_grid
/depth
/depth/points
/octomap_server
n__rviz

Figure2. Perception Rostopic graph

3.1.4 Map Generation

This module implements a custom ROS node map_builder_node that generates an expanding 2D occupancy grid map from depth images using OctoMap as the backend.The resulting map is used for planning and navigation tasks, and is continuously updated as the ego vehicle moves through the environment.

1 Input

Depth Image: **/Unity_ROS_message_Rx/OurCar/Sensors/DepthCamera/image_raw**
(Format: sensor_msgs/Image with 16UC1 encoding)

Camera Info: **/Unity_ROS_message_Rx/OurCar/Sensors/DepthCamera/camera_info**
(Provides intrinsic parameters: fx, fy, cx, cy)

TF: Required transformation from the depth camera frame to the map frame.

2 Processing Pipeline

Depth Projection:
Each pixel of the input depth image is back-projected into 3D coordinates using the provided intrinsic parameters (fx, fy, cx, cy).

TF Transformation:
The 3D points are transformed into the global **map** frame using a real-time transform from the TF tree. This ensures that the mapping is spatially consistent and aligns with the robot's global pose.

OctoMap Insertion:
All valid 3D points are inserted into an instance of **octomap::OcTree**, which efficiently maintains a 3D occupancy representation using a probabilistic model and an adaptive tree structure.

2D Projection to OccupancyGrid:

The 3D OctoMap is flattened into a 2D **nav_msgs/OccupancyGrid** by selecting occupied cells below a height threshold (e.g., z ≤ 2.0 m). This 2D grid is suitable for use by planners or visualization tools like RViz.

Automatic Map Expansion:

As the vehicle moves through the environment, the map is automatically resized if any new data point or the vehicle's position falls outside the current occupancy grid bounds. Old map data is preserved and copied into the expanded map, with the origin updated accordingly. This enables coverage of large or unknown environments without requiring prior knowledge of their size.

3 Output Topic

Table3. Output Topic

Topic	**Type**	Description
/perception/occupancy_grid	**nav_msgs/OccupancyGrid**	2D occupancy grid dynamically updated and expanded as the vehicle moves

Map resolution: **0.2 m/cell**

Default initial map size: **100 × 100 meters** (will Automatic Expansion)

The final result is visualized in RViz as follows:

White regions represent the real-time point cloud generated from the depth camera. These points reflect the ego vehicle's current perception of the environment.

Black shaded areas correspond to occupied cells in the accumulated occupancy map. These regions are identified as obstacles and are considered non-traversable.

The occupancy grid is built incrementally on the ground plane using the OctoMap algorithm, which integrates transformed 3D points over time to create a consistent global map.

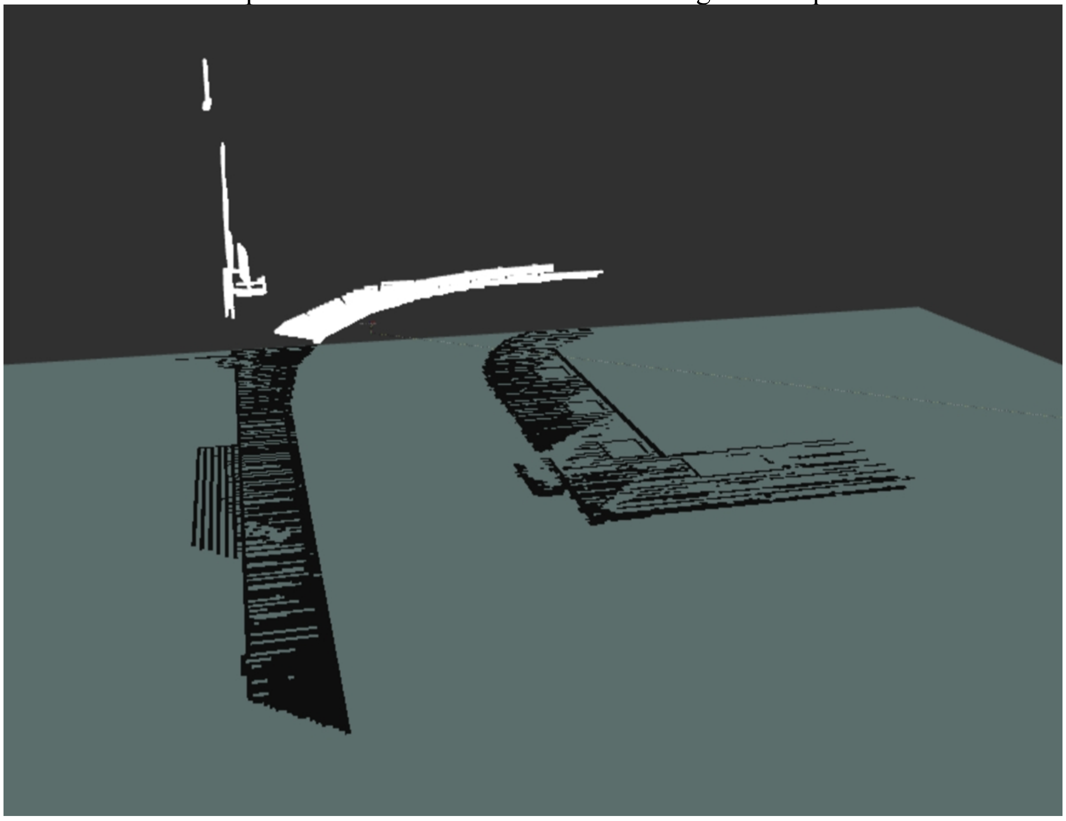


Figure3. RViz visualization of the occupancy map. White points indicate real-time point cloud from the depth camera, while black shaded areas show accumulated obstacles

3.1.5 Traffic Light Recognition Module

This module implements a ROS node that detects and classifies traffic light states from the simulated urban driving environment. It combines information from both semantic segmentation and RGB image analysis to robustly identify red, green and publish them in real-time.

1 Input

/Unity_ROS_message_Rx/OurCar/Sensors/SemanticCamera/image_raw:

This topic provides semantic segmentation images in which each pixel is labeled with its object class. It is used to identify and locate traffic lights in the scene based on their semantic labels.

/Unity_ROS_message_Rx/OurCar/Sensors/CameraCenter/image_raw:

This topic provides RGB images from the center camera of the vehicle. It is used to determine the color state (red/green) of the identified traffic lights by analyzing the pixel content in the region of interest (ROI).

TF Transformations:

Although not always required in the basic version, the tf2_ros system may be used to transform camera frames or align ROIs across sensor views, especially for multi-view or spatial tracking extensions.

2 Processing Pipeline

Traffic Light Localization via Semantic Camera:

Traffic lights appear in the semantic image as yellow-colored regions. The module identifies potential lights by counting the number of yellow pixels in the frame. When multiple candidate regions are detected, the one closest to the center of the image is selected to minimize ambiguity.

For each candidate, its pixel coordinates and bounding box size (pixel area) are extracted and recorded.

Coordinate Mapping to RGB Image:

Since the semantic and RGB cameras may have different resolutions or fields of view(Semantic:320x240, RGB:640x380), the selected bounding box must be projected from the semantic image coordinate space to the RGB image space. The converted coordinates and size are used to extract the corresponding ROI in the RGB image for further analysis. The Visualization of mapping is shown below, for debugging the bounding box is circled in blue

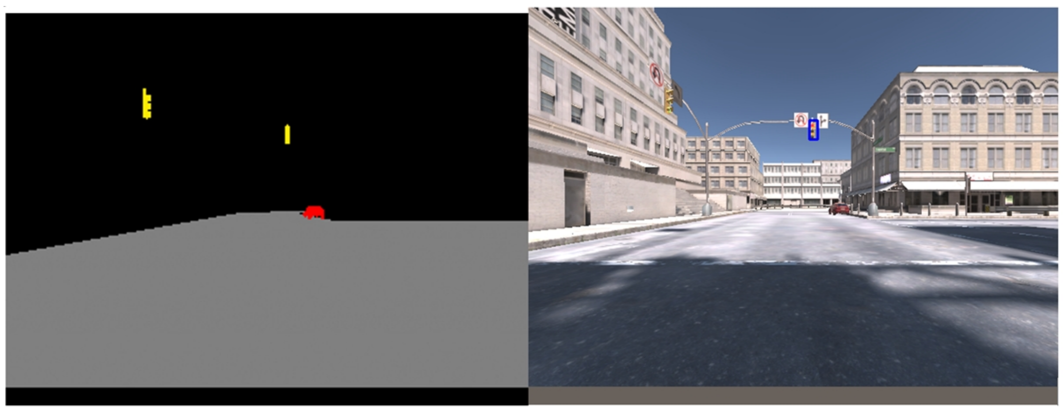


Figure4. Traffic Light Recognition

Color Analysis (HSV-based):

The RGB ROI is converted into HSV color space. Based on the average hue and saturation, the light is classified as:

Red: Hue in [0–10] or [160–180], with high saturation

Green: Hue in [40–90], with high saturation

State Classification:

The traffic light state is labeled as RED and GREEN and published to a dedicated topic.

3 Output Topic

Table4. Output Topic

Topic	**Type**	Description
/perception/traffic_light_status	**std_msgs/String**	Indicates the current traffic light state: "red", "green".

3.1.6 Front hazard detection

This module implements a ROS node that detects potential front hazards, such as stationary or slow-moving vehicles in the direct path of the ego car. It analyzes semantic segmentation images to identify red-labeled cars and estimates their angular position relative to the vehicle's centerline. This helps the decision-making system to determine whether an emergency brake or obstacle avoidance maneuver is needed.

1 Input

/Unity_ROS_message_Rx/OurCar/Sensors/SemanticCamera/image_raw

This topic provides semantic segmentation images in which each pixel is color-coded according to its object class. Red-colored regions typically represent cars in the simulated environment. These images are the sole input used for front hazard detection.

2 TF Transformations

No explicit tf transformations are required for this module, as it operates entirely in image space. All detections and angle computations are derived from pixel coordinates within a fixed region of interest (ROI) in the semantic camera image.

3 Processing Pipeline

ROI Definition and Masking:

A fixed Region of Interest (ROI) is defined in the lower central part of the semantic image, where vehicles directly ahead would appear. The module converts the ROI to HSV color space and generates a binary mask by thresholding for red hues (both low-H and high-H ranges).

Hazard Detection:

The number of red pixels within the ROI is counted. If the red pixel count exceeds a defined threshold (e.g., 200), a front hazard is detected and published. This approach helps eliminate false positives from small or distant red patches.

Angle Estimation:

The spatial distribution of red pixels is analyzed using image moments to compute the horizontal center of mass. This x-coordinate is projected into an angle based on the camera's horizontal field of view (default FOV = 90°). The resulting angle indicates whether the obstacle is slightly left, right, or directly ahead.

Continuous Output:

Unlike binary detection modules, the front hazard node continuously publishes the estimated angle, even when no hazard is detected. In such cases, the angle defaults to zero. This ensures consistent data flow for downstream control modules.

Visualization:

For debugging and visualization, the ROI is outlined in red or white depending on hazard state. The estimated angle is overlaid as text on the image, and the red pixel centroid is marked with a blue circle. This debug image is published to /debug/front_hazard_roi just like below shown.

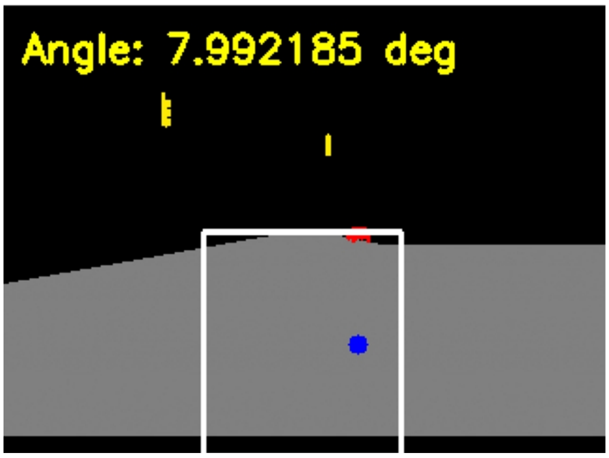


Figure5. Visualization

3.1.7 Configurable Parameters

Table5 abc

Parameter	Description	Typical Value
map_width	Initial occupancy grid width (number of cells)	500
map_height	Initial occupancy grid height (number of cells)	500
sensor_max_range	Maximum depth range to include (in meters)	20.0
height_filter_max	Filter out points above this Z value	2.0
map_expand_buffer	Minimum buffer before map expands (in meters)	5.0
yellow_min_area	Minimum pixel area for yellow box to be considered a light	10
roi_expand_margin	Pixel margin to expand the bounding box for HSV classification	3
hsv_red_thresh	Minimum red pixel count in HSV ROI to classify as red	10
hsv_green_thresh	Minimum green pixel count in HSV ROI to classify as green	10
hazard_pixel_thresh	Minimum red pixels in ROI to consider it a hazard	200
camera_fov_deg	Horizontal field of view of semantic camera used for angle estimation	90.0

3.2 Planning :planning_node: Local Path Planning Based on Enhanced DWA

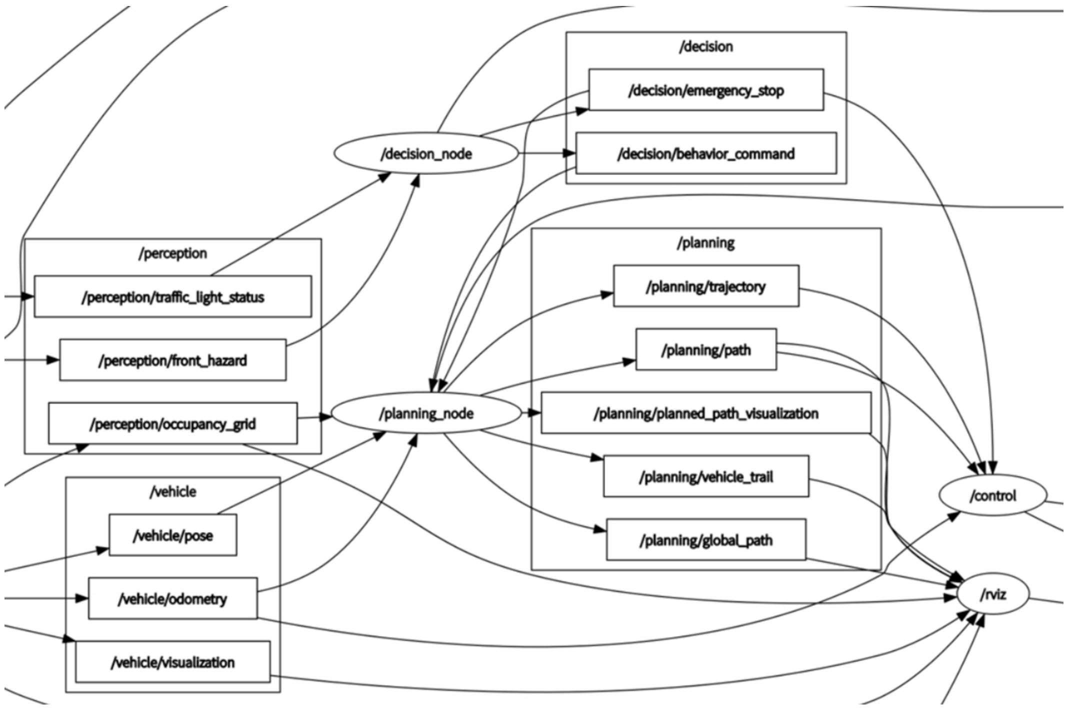


Figure6. planning_node Rostopic graph

3.2.1 Overview

The **planning_node** is the core local planning module responsible for generating safe and feasible motion plans based on real-time perception input. Acting as the intermediate layer between perception and control, it receives environmental data from the perception system, computes motion trajectories using an enhanced Dynamic Window Approach (DWA), and sends planned trajectories or local paths to downstream control modules for execution.

This document outlines the functionality, inputs/outputs, architecture, and tuning parameters of the **planning_node**.

3.2.2 Role in the System

The **planning_node** plays a central role in the navigation pipeline:

- Input: It takes real-time obstacle and localization information from the perception module.
- Processing: It uses the DWA algorithm to evaluate possible velocity and trajectory commands under dynamic constraints.
- Output: It publishes either a velocity-annotated trajectory or a local path to the control module, which then handles execution through throttle and steering control.
- This decoupling allows each module to focus on its domain: perception detects and maps the environment, planning selects a feasible and safe motion strategy, and control executes it precisely.

3.2.3 Subscribed Topics

The **planning_node** typically listens to the following topics:

1 /perception/obstacles

- Type: Custom message or point cloud (e.g., **sensor_msgs/PointCloud2**)
- Description: Provides the spatial positions of static and dynamic obstacles detected by LiDAR or camera.
- Usage: These obstacles are transformed into a 2D costmap and used to evaluate the safety of trajectories in DWA.

2 /localization/odometry

- Type: **nav_msgs/Odometry**
- Description: Provides the current position, orientation, and linear velocity of the vehicle.
- Usage: Used as the starting point for trajectory simulation and cost evaluation.

3 /global_path

- Type: **nav_msgs/Path**
- Description: A high-level path generated by the global planner, consisting of sparse waypoints to the final goal.
- Usage: Serves as the reference for local trajectory alignment. DWA attempts to stay close to this path while avoiding obstacles.

3.2.4 Published Topics

1. /planning/trajectory

- Type: **msg_interfaces/Trajectory** (custom)
- Description: A detailed trajectory including a list of future poses and corresponding velocities.
- Target: This message is consumed by the control module to guide the vehicle's motion.
- Use case: Published when full velocity-aware planning is enabled (e.g., for dynamic maneuvering or tight spaces).

2. /planning/path

- Type: **nav_msgs/Path**
- Description: A simpler fallback path without velocity information.
- Target: Used by the controller when full trajectory information is unavailable or unnecessary.
- Use case: When trajectory generation fails, or during low-speed operation.

3.2.5 Planning Algorithm: DWA (Dynamic Window Approach)

The **planning_node** is based on the DWA local planner, which simulates possible motion commands over a short time horizon and evaluates them using a multi-cost function. The selected trajectory balances progress toward the goal, obstacle avoidance, and alignment with the global path. This algorithm is not using the API from ros,rather we write by ourselves. The waypoints form the global path and then as a guide for DWA.

The trajectory generation and selection process involves:
1. Dynamic Window Construction: Based on current velocity and acceleration limits, generate candidate velocity samples.
2. Forward Simulation: For each (linear, angular) velocity pair, simulate the vehicle’s motion for a few seconds.
3. Cost Evaluation: Assign a cost to each simulated trajectory using the following factors:
4. Trajectory Selection: Choose the lowest-cost, collision-free trajectory and publish it.

3.2.6 Configurable Parameters（problem of sharp corners, not stable）

To adapt DWA behavior to different environments or vehicle dynamics, the following parameters can be tuned via ROS:

Table6. Ros parameters in planning

Parameter	Description	Typical Value
dwa/cost_heading	Weight for aligning trajectory end direction with goal	1.0
dwa/cost_clear	Weight for distance to obstacles (higher = more cautious)	2.0
dwa/cost_vel	Preference for faster velocities	0.5
dwa/cost_path	Preference for staying near the global path	2.7
dwa/obstacle_range	Max range to consider obstacles (meters)	20.0
dwa/inflation_radius	Radius around obstacles to be considered "danger zones"	2.5

dwa/forward_check_distance	How far ahead to evaluate simulated trajectories	2.0–5.0
path_progress_threshold	Distance threshold to advance along the global path	1.0–1.5

If the vehicle turns too late at sharp corners (e.g., 90-degree turns), consider:
- Increasing **forward_check_distance**: Enables the planner to "see" further ahead and start turning earlier.
- Increasing **cost_clear** and **inflation_radius**: Makes the planner more conservative near walls, encouraging earlier turns.
- Reducing **cost_path**: Gives the planner more freedom to temporarily diverge from the global path to complete safe turns.

3.3 Decision making and control

3.3.1 Overview

This part describes the functionality and architecture of a simplified autonomous driving decision-making node developed using ROS (Robot Operating System) in C++. The node integrates inputs from perception modules—namely, traffic light status and front hazard detection—and determines the appropriate driving behavior: proceeding straight, avoiding obstacles, or executing an emergency stop.

3.3.2 System Architecture

The decision node subscribes to two perception topics and publishes three outputs, including a behavior command, an emergency stop flag, and a visualization marker for RViz display:

Table7. Subscribed Topics

Topic Name	**Message Type**	**Published By**	Purpose
/perception/traffic_light_status	**std_msgs::String**	**/traffic_light_node**	Receive traffic light state
/perception/front_hazard	**std_msgs::Bool**	**/front_hazard_node**	Receive front hazard detection

Table8. Published Topics

Topic Name	**Message Type**	**Used By / Visualized In**	Purpose
/decision/behavior_command	**std_msgs::String**	**/planning_node**	Send current behavior decision
/decision/emergency_stop	**std_msgs::Bool**	**/control**	Signal emergency stop
/decision/status_display	**visualization_msgs::Marker**	**/rviz**	Visualize current status and decision reason

3.3.3 Behavior States

The node internally uses an enumerated state machine with three possible values:
- **STRAIGHT:** Normal driving.
- **AVOIDANCE:** Obstacle detected, evasive maneuver required.
- **EMERGENCY_STOP:** Red light detected, stop immediately.

Decision Priority:
1. If red light detected → EMERGENCY_STOP
2. Else if hazard detected → AVOIDANCE

3. Else → STRAIGHT
It inherently functions as a Finite State Machine (FSM).

Table9 State Transition

Current State	Condition	Next State
Any	traffic_light_red == true	EMERGENCY_STOP
Any	traffic_light_red == false && front_hazard_detected == true	AVOIDANCE
Any	All conditions false	STRAIGHT

3.3.4 Visualization

The node publishes a Marker of type TEXT_VIEW_FACING to RViz, placed above the car. The content includes:

– Current behavior status
– Reason for behavior (e.g., "Red Light", "Front Hazard")
– Current sensor state
– Duration in the current behavior

Each behavior uses distinct colors:

– STRAIGHT: Green
– AVOIDANCE: Orange
– EMERGENCY_STOP: Red

Core Functions Explained

– **trafficLightCallback()**

Triggered when a new traffic light status message arrives.
Converts the string to lowercase and sets the internal flag **traffic_light_red_**.

– **frontHazardCallback()**

Updates the **front_hazard_detected_ flag** upon message reception.

– **decideBehavior()**

Implements the core decision-making logic based on sensor states.

– **publishBehaviorCommand()**

Publishes the current behavior state as a string for other modules.

– **publishEmergencyStop()**

Publishes a Boolean message indicating whether an emergency stop is needed.

– **publishStatusVisualization()**

Sends a visualization marker for RViz with text indicating the current state, reason, duration, and sensor info.

– **logBehaviorChange()**

Logs transitions between behavior states and their causes.

3.4 Control

3.4.1 Overview

The control_node is the final execution module in the motion pipeline. It converts the planned trajectory or local path received from the planning module into direct vehicle control commands—specifically throttle, brake, and steering.
The node supports both full-trajectory tracking and simplified path-following, depending on available planning output. Additionally, it monitors emergency stop flags from the decision module to enforce safety overrides.

3.4.2 Role in the System

The control_node acts as the actuator interface between planning and simulation:

– Input: Receives trajectory or path data from planning and current vehicle odometry
– Processing: Computes steering angles and speed commands using adaptive control strategies
– Output: Sends VehicleControl messages to the simulator to control the ego vehicle

The controller ensures accurate trajectory tracking while handling dynamic constraints and safety responses.

3.4.3 Subscribed Topics

Table10. Subscribed Topics

Topic	Type	Description
/planning/trajectory	msg_interfaces/Trajectoryl	Velocity-annotated local trajectory generated by the planner.
/planning/path	nav_msgs/Path	Backup local path without velocity information.
/vehicle/odometry	nav_msgs/Odometry	Vehicle's current position, orientation, and speed.
/decision/emergency_stop	std_msgs/Bool	true triggers emergency brake, overriding planner output.

3.4.4 Published Topics

Topic	Type	Description
/car_command	simulation/VehicleControl	Final vehicle command including target speed and steering angle. Sent to Unity simulator.

Table11. Published Topics

3.4.5 Control Algorithm and Strategy

1 Steering Control

The control system implements multiple steering strategies:

Traditional Hierarchical Control (when **stability_mode = false**):

– **Micro angles** (< 0.6°): Dead zone, no steering output
– **Small angles** (0.6° - 1.0°): Low gain control (80.0)
– **Medium angles** (1.0° - 1.7°): Medium gain control (80.0)
– **Large angles** (> 1.7°): High gain control (120.0)

Advanced PID Control (when **stability_mode = true**):

– Replaces simple proportional gains with full PID control
– Includes integral term for steady-state error elimination
– Enhanced derivative term for improved damping
– Adaptive gain scheduling based on angle magnitude

2 Speed Control

Target Speed Determination:

– Primary: Uses trajectory velocity if available
– Fallback: Uses path-based default speed
– Constraints: Limited to safe operational range (1.5 - 8.0 m/s)

Speed Adaptation:

– Reduces speed during detected instability (15% reduction)
– Further reduction for large steering angles (10% additional)
– Low-speed boost factor for improved responsiveness
– High-speed reduction factor for safety

3 Control Output Processing

Smoothing Pipeline:

1. **Sliding Window Average**: Averages recent steering commands
2. **Rate Limiting**: Constrains maximum steering change rate
3. **Low-pass Filtering**: Applies exponential smoothing
4. **Range Clamping**: Enforces maximum steering limits

Command Generation:

- **Throttle:** Proportional to speed error with saturation
- **Brake:** Activated for negative throttle commands
- **Steering:** Final processed steering with sign correction

3.4.6 Configurable Parameters

All parameters are loaded from the ROS parameter server and can be set in the launch file or .yaml.

Table12 abc

Parameter	Description	Typical Value
control_gain	Basic steering gain	2.0
speed_gain	Speed control responsiveness	0.8
max_steering	Max steering output (rad)	0.5
steering_sign	Direction multiplier (+1 or -1)	-1.0
steering_deadzone	No steering below this angle (degrees)	1.5
small_angle_gain	Gain inside deadzone (set to 0)	0.0
large_angle_gain	Gain outside deadzone	15.0
angle_transition	Transition zone between no-steer and full-steer	1.5

3.4.7 PID Control Method

1 PID Control Theory

The Proportional-Integral-Derivative (PID) controller is a fundamental control algorithm that provides robust tracking performance for dynamic systems. In the context of vehicle steering control, PID helps maintain accurate heading while ensuring stability and smooth operation.

Mathematical Formulation:

$u(t) = Kp \cdot e(t) + Ki \cdot \int e(\tau) d\tau + Kd \cdot de(t)/dt$

Where:

- u(t): Control output (steering command)
- e(t): Error signal (heading error)
- Kp: Proportional gain
- Ki: Integral gain
- Kd: Derivative gain

2 PID Parameter Tuning

Proportional Gain (Kp = 25.0):

- Effect: Primary tracking response
- High Kp: Fast response, potential overshoot
- Low Kp: Slow response, steady-state error
- Tuning: Set for desired response speed without oscillation

Integral Gain (Ki = 1.0):

- Effect: Eliminates steady-state error

- High Ki: Fast error elimination, potential instability
- Low Ki: Slow error elimination, persistent offset
- Tuning: Set to eliminate bias while maintaining stability

Derivative Gain (Kd = 3.0):
- Effect: Provides damping and anticipation
- High Kd: Strong damping, noise sensitivity
- Low Kd: Weak damping, potential overshoot
- Tuning: Set for optimal damping without noise amplification

3.4.8 Failure Recovery

1 Planning Data Loss

No Trajectory Available:
- Automatically switches to Path Following Mode
- Uses /planning/path with default velocity profile
- Logs warning: "No trajectory - using path fallback"
- Continues operation with reduced precision

No Path Available:
- Switches to Safe Stop Mode
- Applies gentle braking (30% brake force)
- Maintains last known steering angle
- Logs error: "No planning data - gentle stop"

2 Sensor Data Loss

Odometry Loss:
- Stops publishing control commands
- Maintains last known vehicle state
- Logs critical error: "Vehicle odometry lost"
- Waits for odometry recovery before resuming

Communication Timeouts:
- Implements watchdog timers for critical topics
- Triggers safe stop if timeout exceeded
- Configurable timeout values (default: 1.0 second)

3 Control System Failures

Actuator Saturation:
- Detects persistent control saturation
- Activates anti-windup mechanisms
- Reduces control aggressiveness temporarily
- Logs performance warnings

Oscillation Detection:
- Monitors for persistent oscillatory behavior
- Automatically increases damping
- Reduces control gains if necessary
- Provides diagnostic feedback

Instability Recovery:
- Detects growing tracking errors
- Switches to conservative control mode
- Resets integral terms and history buffers
- Gradually returns to normal operation

4 Result figures

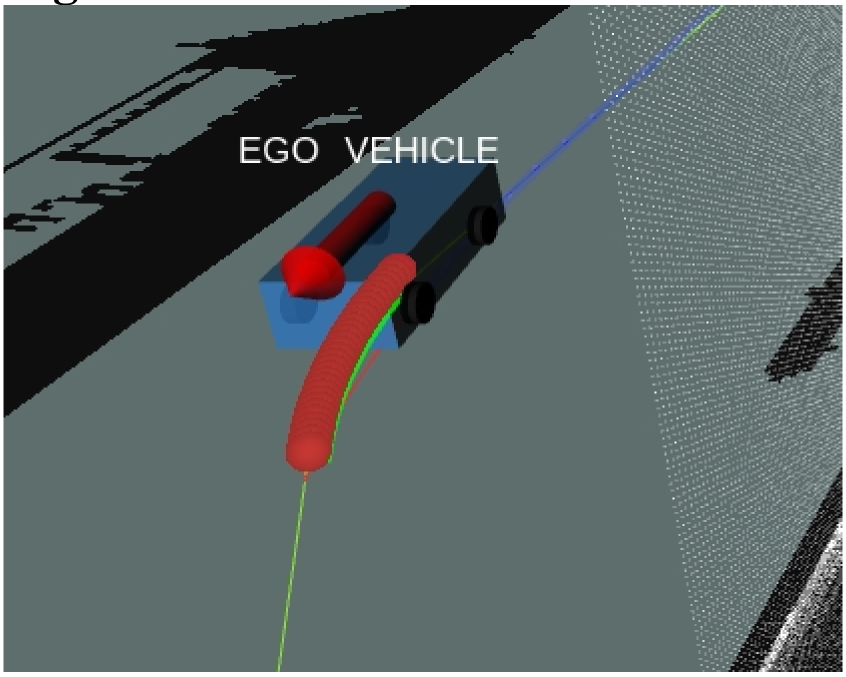


Figure7. local planner(red line)

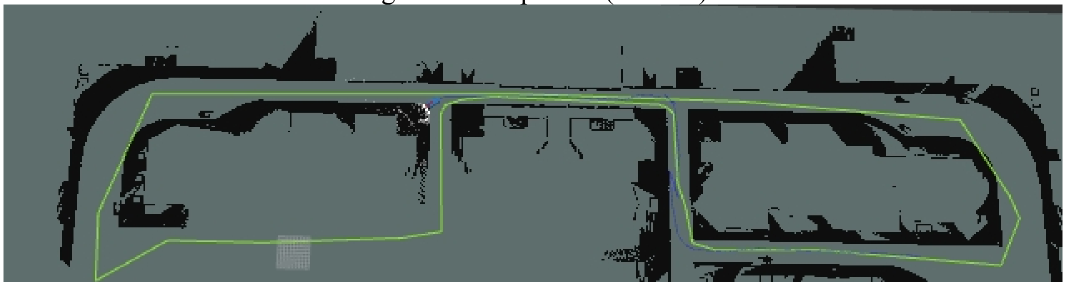


Figure8. dense global planner path

5.Conclusions and Outlook

Key Accomplishments:
Integrated perception, planning, decision making and control into a single system that drives the Unity city-course vehicle from start to finish without human intervention.

Real-time perception:
Fused depth, RGB and semantic images to build a 2-D occupancy grid and detect traffic-light states at > 10 Hz while publishing obstacle angles for the planner.

Enhanced Dynamic Window Approach (DWA) planner:
Sampled the vehicle’ s dynamic window, simulated candidate motions with curvature penalties and selected the lowest-cost, collision-free trajectory that follows dense global way-points at ≈ 30 Hz.

Finite-State Machine (FSM) for behaviour selection:
Implemented CRUISE, STOP_AT_LIGHT, FOLLOW_VEHICLE and EMERGENCY_BRAKE states; each state publishes both a drive command and an RViz marker for transparency.

Hierarchical / PID control:
Lateral and longitudinal PID controllers track chosen trajectories, apply anti-wind-up and smoothing, and obey emergency-stop overrides.

Verified performance:

The vehicle completed the entire city track in 172 s (under the 180 s target), stayed in its lane, respected traffic lights, and avoided static or slow obstacles—zero collisions.

Custom ROS interfaces:
Created a TrafficLightState.msg for semantic results and a /sim_bridge/reset service for rapid scene resets, fully decoupling perception from decision layers.

6.Bibliography

– https://en.wikipedia.org/wiki/Robot_Operating_System
– https://chatgpt.com/