

# **A VISUALIZATION TOOL FOR PERCEPTION SYSTEM DEVELOPMENT AND OPTIMIZATION**

A Thesis  
Presented to  
The Academic Faculty

By

Michelle D. Warren

In Partial Fulfillment  
of the Requirements for the Degree  
Master of Science in the  
School of Electrical and Computer Engineering

Georgia Institute of Technology

May 2023

© Michelle D. Warren 2023

# **A VISUALIZATION TOOL FOR PERCEPTION SYSTEM DEVELOPMENT AND OPTIMIZATION**

Thesis committee:

Dr. David Taylor, Co-Advisor  
Electrical and Computer Engineering  
*Georgia Institute of Technology*

Dr. Samuel Coogan  
Electrical and Computer Engineering  
*Georgia Institute of Technology*

Dr. Antonia Antoniou, Co-Advisor  
Mechanical Engineering  
*Georgia Institute of Technology*

Dr. Patricio Vela  
Electrical and Computer Engineering  
*Georgia Institute of Technology*

Date approved: April 28, 2023

Transformation is intellect, will, purpose, desire. Die. Be born. Bring forth labors and love. Let the invisible be in the visible. Name yourself and know who you are.

*The Egyptian Book of the Dead*

For my grandmother Annette Warren



## ACKNOWLEDGMENTS

I would like to thank my co-advisors for their persistent help in preparation of this work – Dr. David Taylor and Dr. Antonia Antoniou – without whom I would not have been able to organize my thoughts and minimize my scope. They provided eyes for things I could not see and grounding for one of the hardest periods of my life. I would also like to thank all members of my thesis committee, including Dr. Samuel Coogan and Dr. Patricio Vela, for their added expertise in the field and openness to be on my committee.

A very special thanks is due to the Georgia Tech EcoCAR team, whose invaluable teamwork and effort allowed us all to secure success. The team’s willingness to complete weekend testing, their patience with tools and hardware, and their go-getter, can-do, attitude of grit does not go unnoticed. Particularly, I’d like to thank Nick Hummel, who I worked alongside on the CAVs team, and who could give insight as a controls engineer as to what he needed from the perception system. He quickly learned the new tools and skills that I was showing him, took those, and ran, helping to create a robust CAVs system from end-to-end.

Special thanks are due to the friends and colleagues who made this work possible. Neque Willis, Melat Abraham, and Brandon Bland were invaluable both as friends and as sounding boards for my ideas.

The author gratefully acknowledges the support for this work offered by both the department through a Graduate Research Assistantship and the GEM National Consortium Fellowship. Any views and conclusions contained herein are those of the author, and do not necessarily represent the official positions, expressed or implied, of the funders.

## TABLE OF CONTENTS

<b>Acknowledgments</b> . . . . .	v
<b>List of Tables</b> . . . . .	ix
<b>List of Figures</b> . . . . .	x
<b>List of Acronyms</b> . . . . .	xii
<b>Summary</b> . . . . .	xiv
<b>Chapter 1: Introduction and Background</b> . . . . .	1
1.1 EcoCAR Competition . . . . .	1
1.2 Perception Systems in Autonomy . . . . .	2
<b>Chapter 2: GT EcoCAR Perception System</b> . . . . .	5
2.1 Hardware Architecture . . . . .	5
2.1.1 Platform . . . . .	5
2.1.2 Compute . . . . .	5
2.1.3 Sensor Suite . . . . .	7
2.2 Existing Software . . . . .	9
2.2.1 ROS Overview . . . . .	9
2.2.2 Sensor Fusion Node in Simulink Overview . . . . .	10

2.2.3	Automation . . . . .	11
<b>Chapter 3: Tank Viz Description</b>	<b>. . . . .</b>	<b>13</b>
3.1	Purpose . . . . .	13
3.2	Data Displayed on Tank Viz . . . . .	16
3.3	Software Architecture . . . . .	24
3.3.1	Python Subscribers and Publishers . . . . .	25
3.3.2	RViz Package Overview . . . . .	25
3.3.3	Catkin Workspace and ROS Packages Needed . . . . .	26
3.4	Real-Time and Post Processing Usage . . . . .	27
3.5	Line-By-Line Software Description . . . . .	28
3.5.1	Automation Script . . . . .	28
3.5.2	Visualization Script . . . . .	29
<b>Chapter 4: Conclusion and Future Work</b>	<b>. . . . .</b>	<b>35</b>
4.1	Tool Modification . . . . .	37
<b>Appendices</b>	<b>. . . . .</b>	<b>39</b>
<b>Chapter A: Tank Viz Scripts</b>	<b>. . . . .</b>	<b>40</b>
A.0.1	viz_d.py: Detection Visualization Script . . . . .	40
A.0.2	viz_lanes_v2.py: Lane Detection Visualization Script . . . . .	47
A.0.3	viz_lv.py: Lead Vehicle Visualization Script . . . . .	51
A.0.4	viz_tracks.py: Track Visualization Script . . . . .	59

<b>References</b>	67
-------------------	----

## **LIST OF TABLES**

3.1	Tank Viz markers and descriptors . . . . .	17
-----	--	----

## LIST OF FIGURES

1.1	An example of a MATLAB simulation of a forward facing perception system. On the left is the chase camera view, showing simulated obstacles ahead. On the right is the bird's eye view, showing the field-of-view for selected sensor modalities. . . . .	4
2.1	The Georgia Tech EcoCAR Team's re-engineered 2019 Chevrolet Blazer, affectionately named "Poppy", an acronym inspired by the vehicle's P0 motor and P4 motor regenerative braking architecture. . . . .	6
2.2	Left: Bosch Mid-Range RADAR (MRR4). Right: Intel Mobileye camera. . . . .	8
2.3	CAV hardware architecture. . . . .	9
2.4	A screenshot of the sensor fusion node in Simulink, specifically the adjustable parameters within the IMM block. . . . .	12
3.1	A video monitor mounted on the back of the passenger seat headrest for engineer observation from the backseat. . . . .	15
3.2	Intel Tank startup screen . . . . .	18
3.3	This is a typical view of what is perceived by the vehicle during highway driving. The Lead Vehicle is the nearest obstacle within the lane boundaries. Outside of the lane boundaries, other detections and tracks can be observed from obstacles in other lanes. In some instances, one can observe a cluster of camera and RADAR detections with an accompanying track. This shows that the sensor fusion node is taking those nearby detections and creating tracks. Although this is a freeze-frame and all objects may not reflect what could be observed even a second later, it gives a good view of how the perception system provides information about the environment. . . . .	19

3.4	This is a capture emphasizing the green lane boundaries that are provided from camera detection data while driving through a neighborhood. Obstacles on the right side of the lane boundaries are objects on the side of the road, such as signs or bike riders. Objects on the left side of the lane boundaries represent oncoming traffic. . . . .	20
3.5	This is a capture emphasizing the magenta lane boundaries that are calculated from steering angle data while driving through a neighborhood. Obstacles on the right side of the lane boundaries are objects on the side of the road, such as signs or bike riders. Objects on the left side of the lane boundaries represent oncoming traffic. . . . .	20
3.6	Visualization screen with populated data. On the left of the screen is the current configuration after it has been expanded by the left arrow. . . . .	21
3.7	Configuration options within the config panel. . . . .	23
3.8	Config file open window . . . . .	24
3.9	Scripts within the using_markers package . . . . .	25
3.10	The automation script <code>script.sh</code> included in the <code>Tank startup.sh</code> shell script used to open programs at system start up. . . . .	28
3.11	The first chunk of code that imports required ROS packages, initializes a publisher, sets up a ROS node, and sets the ROS publish rate . . . . .	30
3.12	The next chunk of code that defines track and ego-vehicle (referred to within the code as “ownship”) marker initialized parameters. . . . .	31
3.13	The next chunk of code that defines callback functions. . . . .	32
3.14	The final chunk of code that updates marker arrays and publishes the markers. This chunk also handles clearing of markers before updating again. . .	34

## LIST OF ACRONYMS

<b>6DOF</b>	six degrees of freedom
<b>ACC</b>	Adaptive Cruise Control
<b>AVTC</b>	Advanced Vehicle Technology Competitions
<b>CAVs</b>	Connected Automated Vehicles
<b>DSRC</b>	Dedicated Short-Range Communications
<b>EPA</b>	Environmental Protection Agency
<b>EV</b>	Electric Vehicle
<b>fps</b>	frames per second
<b>GPS</b>	Global Positioning System
<b>GT</b>	Georgia Tech
<b>GUI</b>	Graphical User Interface
<b>IMM</b>	Interacting Multiple Model
<b>IMU</b>	Inertial Measurement Unit
<b>KVM</b>	keyboard, video (monitor), mouse
<b>LiDAR</b>	Light Detection and Ranging
<b>MAP</b>	intersection geometry
<b>MIO</b>	Most Important Object
<b>OBU</b>	On-Board Unit
<b>PCM</b>	Propulsion Controls and Modeling
<b>RADAR</b>	Radio Detection and Ranging
<b>RGB</b>	red/blue/green
<b>ROS</b>	Robotics Operating System



**SPaT** Signal Phase and Timing

**STEM** science, technology, engineering, and math

**V2X** Vehicle-To-Everything

## SUMMARY

Perception systems are necessary for advanced autonomous technologies. During the integration, development, and testing phases of creating a perception system, it has been proven to be necessary to create a means for visualizing sensor data and perception information obtained from sensor fusion algorithms. The proposed research objective is to discuss the development and use of a visualization tool for the EcoCAR project, my individual contribution. To develop this visualization tool, I implemented a cluster script architecture, assisting in environment automation, that handled subscription to detected object information that may be in the form of sensor detections; sensor fused tracks; the Most Important Obstacle (MIO) track, also referred to as the lead vehicle; lane information; and publishing of markers based on the position of objects to be reflected on the visualization display with respect to the car being controlled, or the ego-vehicle. The visualization system, referred to in this thesis as Tank Viz, provided the team with a complete perception system for continued development, troubleshooting, and testing that helped lead to the team's eventual success in Year 4 of the Mobility Challenge Connected Automated Vehicles section of the competition and provides a foundation for the team's perception system for the 2022-2026 Electric Vehicle Challenge.

# **CHAPTER 1**

## **INTRODUCTION AND BACKGROUND**

### **1.1 EcoCAR Competition**

EcoCAR is a competition that is part of the Advanced Vehicle Technology Competitions (AVTC), a series of North America's premier collegiate automotive engineering competitions. AVTCs engage students from middle school through higher education, creating a pipeline that both encourages students to pursue careers in science, technology, engineering, and math (STEM) and has seeded more than 30,000 graduates into industry, helping to build the workforce needed for the U.S. to be competitive in the global marketplace [1]. Sponsored by the U.S. Department of Energy, General Motors, MathWorks, and Argonne National Labs, along with other partners and contributors, EcoCAR is one of the AVTC competitions that Georgia Tech is involved with on a large scale. The goal of this competition is to stimulate the development of advanced vehicle technologies that reduce the overall impact of transportation on the environment by designing, building, and refining an alternative fuel and connected/autonomous vehicle that reduces energy consumption, and greenhouse gas and tailpipe emissions while maintaining consumer acceptability, utility, and safety. The Georgia Tech (GT) EcoCAR team has been a competitor in EcoCAR for the past 3 consecutive EcoCAR challenges: EcoCAR 3, EcoCAR Mobility Challenge, and now the EcoCAR Electric Vehicle (EV) Challenge. Each cycle of the EcoCAR competition lasts for 4 total years, with goals and deliverables for each year and an annual end-of-the-year competition. The Georgia Tech EcoCAR team works with upwards of 40-50 undergraduate students and 10 graduate students in a highly technical environment to address issues including but not limited to optimized component sizing and subsystem design based on performance versus cost tradeoffs, design of methodologies to ensure reliability

and safety, virtual prototyping and hardware-in-the-loop testing, and vehicle fabrication and on-road testing. The EcoCAR Mobility Challenge took place from the start of the Fall 2018 semester in August to the conclusion of the Spring 2022 semester in May amongst 12 university teams. One of the technological tracks, Connected Automated Vehicles (CAVs), requires multidisciplinary groups of students to lead the development of the connected and automated driving features for each of their teams. Year 4 of the Mobility Challenge was focused on a complete automated system that performs Adaptive Cruise Control (ACC) capabilities and Vehicle-To-Everything (V2X). The Georgia Tech team's perception system work spans over the course of two competition cycles, the Mobility Challenge and the Electric Vehicle Challenge, and will be discussed in this thesis.

## **1.2 Perception Systems in Autonomy**

Any autonomous system for any one of the six SAE J3016 levels of driving automation requires a way to perceive the environment around it. This requires the robot or vehicle to have a perception system. The perception system is made up of sensors and processes for collecting information about the environment and using it to inform decision making. The perception system, as described by Rosique et al. is a "positioning estimation system", perceiving and identifying obstacle distances from the ego-vehicle [2].

A rich perception system takes a multi-modal approach, making use of many variable sensors. As Campbell et al. describe in their review of sensor technology in autonomous systems, a perception system can consist of a mixture of exteroceptive sensors and proprioceptive sensors [3]. Exteroceptive sensors can include Light Detection and Ranging (LiDAR), Radio Detection and Ranging (RADAR), and a variety of types of cameras. Proprioceptive sensors include Global Positioning System (GPS), Inertial Measurement Unit (IMU), and encoders.

Sensor fusion is the next critical component of perception systems in autonomy by combining data from multiple sensors to provide a more accurate and robust representation

of the environment. As outlined by Yeong et al. “multi-sensor fusion is effectively now a requisite process”, allowing perception systems to overcome the limitations of individual sensors, such as noise, occlusion, and limited field of view, and to integrate complementary information from multiple sensors to provide a more complete understanding of the environment [4]. In this context, sensor fusion is a fundamental technique that enables perception systems to operate effectively in complex and dynamic environments.

Visualization of perceived information from an autonomous system is an important part of the development of autonomous technologies that “helps scientists and engineers to investigate physical systems through a process of geometric abstraction” according to R.B. Haber [5]. Bertoline et al. have suggested that visualization ability is central to design [6]. Visualization is a process that includes three main steps: preparing the data, simplifying the object, and then creating a visual representation of it. Visualization systems are used during simulation, real-time operations, and post-processing and analysis. Figure 1.1 shows an example of a MATLAB simulation of a forward facing perception system, which shows the developer a 2-D and 3-D representation of a vehicle and the sensor suite fields of view. The depiction of data in this visualization is effective for simulation, where the 3-D representation provides greater insight for engineers on realistic scenarios and the 2-D representation maps the sensor capabilities.

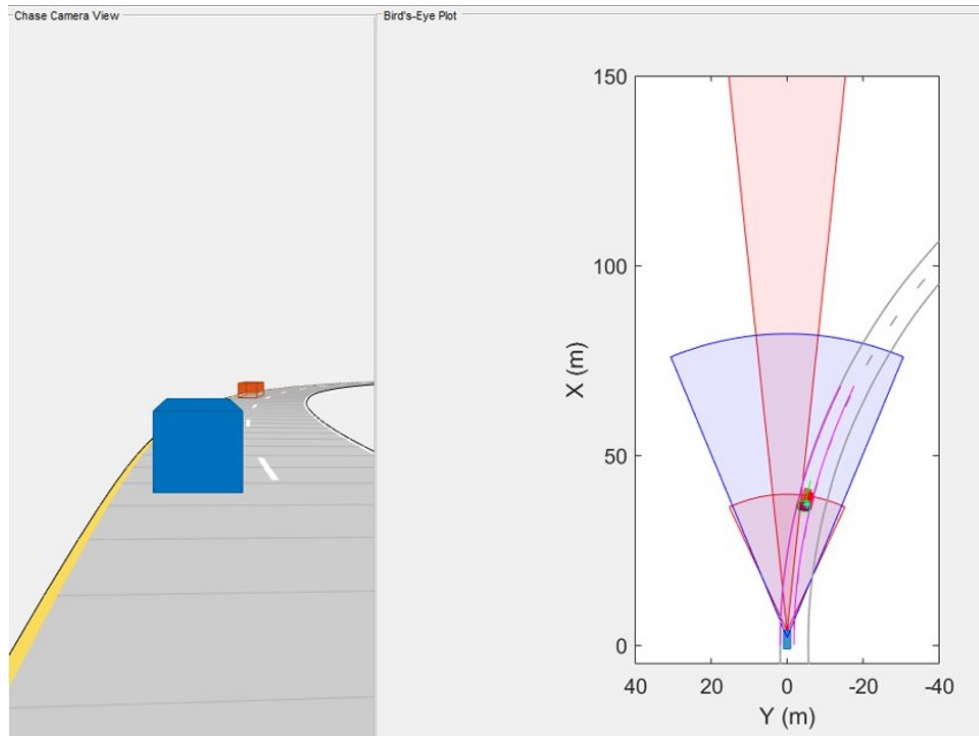


Figure 1.1: An example of a MATLAB simulation of a forward facing perception system. On the left is the chase camera view, showing simulated obstacles ahead. On the right is the bird's eye view, showing the field-of-view for selected sensor modalities.

## **CHAPTER 2**

### **GT ECOCAR PERCEPTION SYSTEM**

#### **2.1 Hardware Architecture**

##### 2.1.1 Platform

Our platform is a 2019 Chevrolet Blazer. Our Blazer originally came with its general specifications, such as a 3.6-liter LGX 6 cylinder engine with Environmental Protection Agency (EPA) estimates of 20 mpg city and 26 mpg highway (22 combined) with all-wheel drive [7], and standard safety features including a rear view camera, rear-seat reminder, and Teen Driver system. By Year 4 of the competition, the major modifications that we made to the Blazer include: advancing propulsion systems to achieve hybrid electrification with an optimized fuel efficiency of 32 mpg combined; and SAE level 2 automation and vehicle connectivity via perception system situational awareness, V2X communication, and adaptive cruise control. Our platform was required to be returned to a consumer-ready end-product after modifications were made to be driven and evaluated at the competition. All this was done with emissions and safety factors in mind. Figure 2.1 shows the test platform in Yuma, AZ for the Year 4 EcoCAR Competition Dynamic Vehicle Testing.

##### 2.1.2 Compute

The Intel IEL Tank AIO is a high-performance embedded computer designed for industrial applications that require a rugged, reliable, and powerful system. It is powered by a 5th generation Intel Core processor and R680E chipset, which provides excellent computing power and graphics performance. The IEL Tank AIO also offers a range of connectivity options, including multiple LAN (ethernet) ports, USB ports, and expansion slots for additional peripherals [8]. The expansion slots proved to be useful for the team, as the Tank



Figure 2.1: The Georgia Tech EcoCAR Team’s re-engineered 2019 Chevrolet Blazer, affectionately named “Popy”, an acronym inspired by the vehicle’s P0 motor and P4 motor regenerative braking architecture.

originally came with no peripherals for CAN bus connections and were able to be easily updated to include many. With its robust design and powerful performance, the Intel IEI Tank AIoT is an ideal solution for industrial automation, machine vision, and other demanding applications.

Both sensor fusion and longitudinal controller algorithms necessary to implement ACC are implemented in the primary compute unit, the Intel Tank. The Intel Tank runs on the Linux operating system. The Tank interfaces with actuators in the vehicle via a hybrid supervisory controller, the dSPACE MicroAutoBox II. The Cohda MK5 On-Board Unit (OBU) is a wireless radio that communicates via Dedicated Short-Range Communications (DSRC) and provides V2X capability to the team architecture. The OBU is used to receive Signal Phase and Timing (SPaT) and intersection geometry (MAP) messages from signalized intersections and communicate them to the Tank to enable autonomous traversal of



signalized intersections.

### 2.1.3 Sensor Suite

The forward-facing sensors on the Blazer include a Bosch Mid-Range RADAR (MRR4) and an Intel Mobileye 6 camera.

The Bosch MRR4 is a high-performance RADAR sensor designed for use in automotive applications. It has two field-of-view ranges: a short-range of 45 meters with a field of view of +/- 45 degrees, and a long range of 160 meters with a field of view of +/- 10 degrees. The RADAR operates in the 77 GHz frequency band, providing high accuracy and resolution. The MRR4 features advanced signal processing algorithms, which enable it to distinguish between different types of objects<sup>1</sup> and accurately determine their position and velocity. It also includes multiple detection zones, allowing it to track multiple objects simultaneously. The tracking referenced here has minimal time-filtering, and is used only to reduce the bandwidth from the raw data [9].

The Intel Mobileye 6 camera is a cutting-edge vision-based sensor system designed for use in automotive applications. With its high-resolution 8-megapixel CMOS sensor and wide field of view of up to 100 degrees, the Mobileye 6 camera can capture a broad area in front of the vehicle with exceptional image quality and detail. The camera is capable of detecting and classifying a wide range of objects, including vehicles, pedestrians, and cyclists, using advanced computer vision algorithms. It uses highly effective proprietary algorithms to estimate the distance to objects using mono vision technology, which provides 3D information of the surrounding environment from a single camera[10]. The Mobileye 6 camera is designed to work in a variety of lighting conditions, including low light and high glare, thanks to its advanced image processing capabilities.

The RADAR and camera sensor modalities, shown in Figure 2.2, both provide obstacle

---

<sup>1</sup>The MRR4 user manual specifies an object message for obstacles that includes a classification signal within the CAN bus message. This classification signal has enumerations for an obstacle that is 0 = unknown, 1 = a moving 4-wheel vehicle, 2 = a moving 2-wheel vehicle, 3 = a moving pedestrian, or 4 = a constant element [9].



Figure 2.2: Left: Bosch Mid-Range RADAR (MRR4). Right: Intel Mobileye camera.

positional and velocity data relative to the ego-vehicle. They are both designed for ease of integration into automotive systems, boasting compact form factors, reliability and durability in extreme temperatures, vibrations, and shocks, and low power consumption. The camera sensor, with its own internal compute system, also provides lane information. When the camera is unable to provide lane information, lanes are extrapolated from the steering angle input and a fixed value for the left and right offsets. For reference, the equation for lane extrapolation from steering angle information is given below:

$$\text{left\_lane} = (\text{curvature} * x^2) + (\text{heading} * x) + \text{left\_offset}$$

$$\text{right\_lane} = (\text{curvature} * x^2) + (\text{heading} * x) + \text{right\_offset}$$

where left\_lane and right\_lane refer to an equation for the lane markings to the left and right of the ego-vehicle respectively, curvature refers to the approximated rate of change of direction in meters, heading refers to the bearing in degrees, and left\_offset and right\_offset refer to a pre-determined constant representing the width from the center of the car to the left or right lane markings respectively. The left\_offset and right\_offsets are set to -1.8

and 1.8 and respectively to match the average lane width of 3.6m on both urban and rural freeways as determined by the Department of Transportation [11], with the assumption that the ego-vehicle is in the center of the lane at all times.

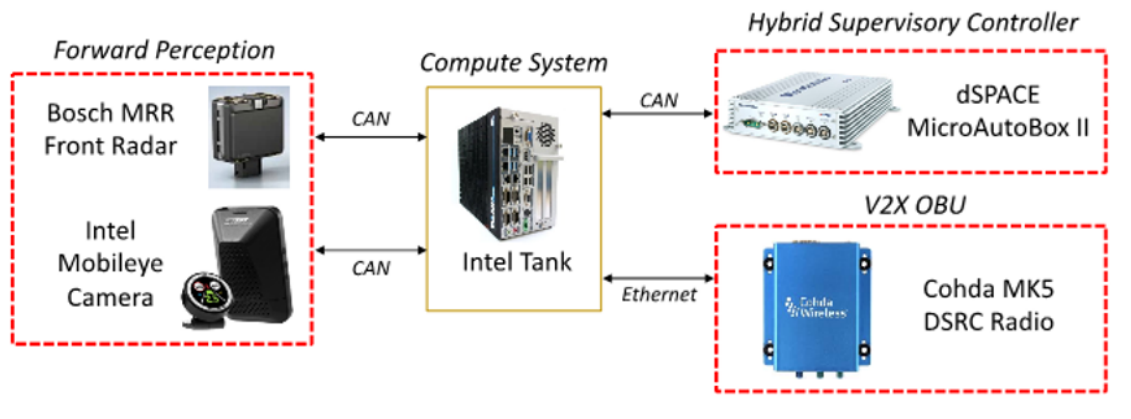


Figure 2.3: CAV hardware architecture.

Figure 2.3 shows a visual representation of the connections made between the sensors and compute systems for the vehicle's perception system. The purpose of the perception system is to find the next closest vehicle in the ego-vehicle's lane. The inputs to the perception system for sensor fusion are sensor object detections and lane detections. The output is the displacement measurement to the next vehicle in the current lane.

## 2.2 Existing Software

### 2.2.1 ROS Overview

The Robotics Operating System (ROS) is an open-source software framework that provides a collection of libraries and tools to help software developers build robotic applications [12]. ROS was initially developed in 2007 by Willow Garage, a robotics research lab in California, and is now maintained by the Open Robotics organization [13]. ROS supports a wide range of programming languages, including C++, Python, and Java, and has a large community of developers who contribute to its development and maintenance.

ROS provides a powerful and flexible framework for building robotic applications, en-

abling developers to focus on the high-level logic of their applications rather than the low-level details of hardware and communication protocols. ROS provides a set of libraries and tools that help manage the complexity of robot software development. These include a message-passing system, a parameter server, a package management system, and various tools for visualization, simulation, and debugging. ROS also provides a standardized architecture for building robot software, which includes a modular structure with nodes, topics, and services.

Nodes are individual programs that perform specific tasks, such as reading sensor data or controlling a robot's actuators. Nodes communicate with each other through a message-passing system, where messages are published on topics and subscribed to by other nodes. This enables a distributed architecture, where different nodes can run on different computers and communicate with each other over a network.

### 2.2.2 Sensor Fusion Node in Simulink Overview

The sensor fusion algorithm is a critical component of autonomous vehicle technology. It combines data from multiple sensors, such as cameras, LiDAR, RADAR, and GPS, to provide a more accurate and complete view of the environment around the vehicle. The algorithm processes and analyzes the data from each sensor to identify objects, such as other vehicles, pedestrians, and road signs, and creates a comprehensive understanding of the surroundings. The algorithm then fuses this data together to create a real-time, high-definition 3D map of the environment. This map is used by the vehicle's control system to make decisions about steering, acceleration, and braking, allowing the vehicle to navigate safely and effectively. The development of effective sensor fusion algorithms is essential for the advancement of autonomous vehicle technology and the safe deployment of self-driving cars on public roads.

The sensor fusion node on our platform was created using Mathworks' Simulink. The graphical interface provided by Simulink provided a means for simplifying the design and

simulation process for models that would be challenging to model using traditional programming languages. Simulink models can also be easily modified, and changes can be quickly implemented, which helps speed up the development process.

The sensor fusion node takes in parsed CAN bus detection data from sensors and the vehicle, and provides tracks and information regarding the Most Important Object (MIO), also referred to as the lead vehicle. “Tracks” refer to the process of detecting and identifying objects in a sensor’s field of view over time. The tracking process involves associating the object’s location in different sensor frames to create a trajectory or “track” of its movement over time. This trajectory can be used to predict the object’s future position and velocity, which is essential for autonomous vehicle control and decision-making. The sensor fusion node creates tracks from the multiple sensor modality (front-facing camera and RADAR) detections and fuses tracks that fit into the identified parameters for fusion, such as a detection assignment threshold based on the detection’s normalized distance from an existing track, the chosen Kalman filter for fusion, and many more. For reference, the Kalman filtering technique initialized for sensor fusion within the Interacting Multiple Model (IMM) block was the constant-velocity extended Kalman filter. The selectable parameters for setting up the filter can be seen in the open panel in Figure 2.4.

### 2.2.3 Automation

Automation scripts on the Intel Tank enabled our team to save time and effort by automating manual tasks, reducing errors and improving productivity. Automation scripts also improved the consistency and reproducibility of workflows, ensuring that the same set of steps is followed every time the script is executed. This was vital during start-up for testing and even on the back-end with our final product, enabling ACC without operator input. Our team’s visualization tool to be discussed, further referred to as Tank Viz, interfaced with a few of the automation scripts, namely the boot script that runs at system startup and a script for starting all nodes of Tank Viz.

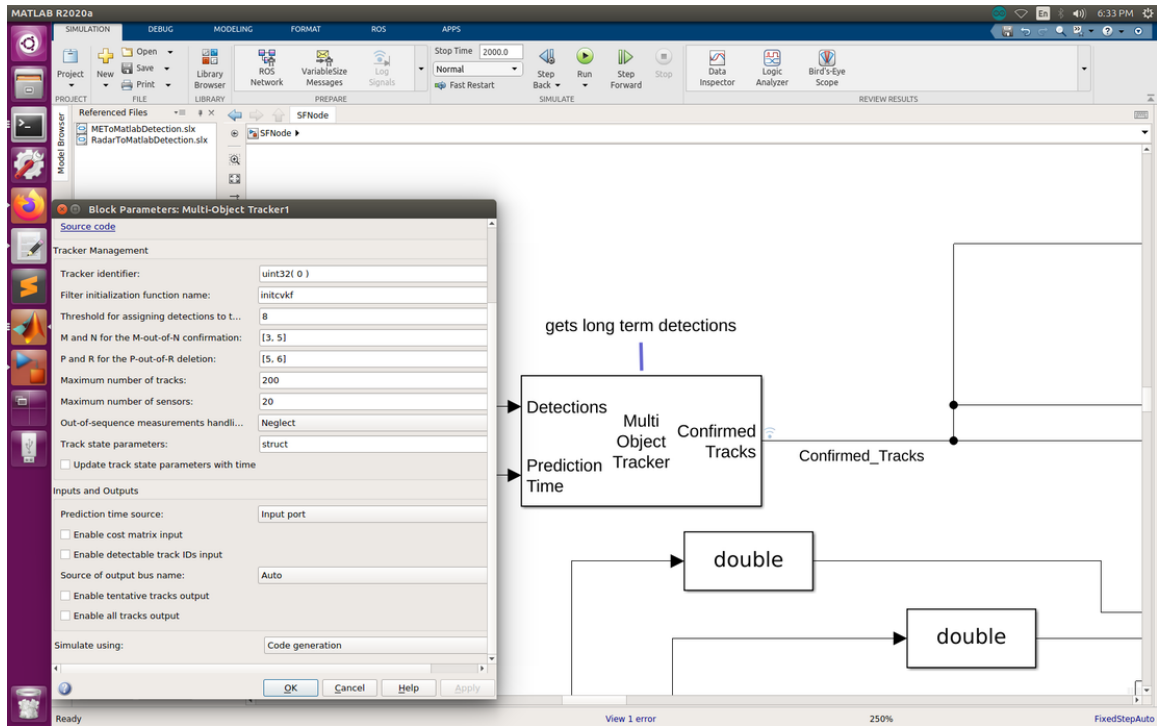


Figure 2.4: A screenshot of the sensor fusion node in Simulink, specifically the adjustable parameters within the IMM block.

The Intel Tank has a mechanical switch on the back panel for toggling ACC Mode. The ACC mode switch on the Intel Tank is a unique feature that allows the user to toggle the ACC mode on and off, modifying the Tank's performance and power consumption according to their needs. When the ACC mode switch is toggled off, the Intel Tank functions like a normal processor, using maximum performance and outputting information to a graphical display such as a monitor. When the ACC mode switch is toggled on, the Intel Tank is ready to be used in the background for Automated Cruise Control and is put into a mode that better balances performance and energy consumption (for example, information is not passed to an external graphical display). The consumer-ready platform uses the switch toggled on, while testing and optimization phases of the engineering process use the switch toggled off so that we have access to the system configuration that allows graphical display.

## **CHAPTER 3**

### **TANK VIZ DESCRIPTION**

#### **3.1 Purpose**

In recent years, perception systems have gained increasing attention in a variety of fields, including robotics, autonomous driving, and medical imaging, among others. These systems rely on various sensors, such as cameras, LiDARs, and RADARs, to capture data from the surrounding environment and process it to make informed decisions. However, interpreting the raw sensor data can be challenging, especially for complex and dynamic environments.

The purpose of a visualization tool in a perception system is to transform the raw sensor data captured by sensors, such as cameras, LiDARs, and RADARs, into graphical representations that can be easily understood by humans. A visualization tool presents sensor data in a visual format, such as 2D/3D visualizations, heatmaps, or augmented reality overlays, allowing the user to interpret and analyze the data more effectively. Tank Viz as discussed in this thesis is a 2D/3D visualization.

The use of visualization tools can enhance the performance of perception systems by assisting engineers in building systems that accurately and efficiently detect, track, and classify objects in complex and dynamic environments. Visualization tools are one of the ways engineers can make heuristic determinations of what the perception system sees. Visualization tools can also help to identify patterns and anomalies in the data, providing insights into the behavior of the environment and facilitating decision-making processes for engineers and software-developers when considering the controls aspect of autonomy.

Tank Viz as discussed in this thesis is not intended for use by a driver in an autonomous car end-product. Therefore the visualization system described in this thesis does not im-

prove the user experience by presenting information that will lighten the cognitive burden of the user. The scope of Tank Viz described is that the tool is used only for the validation and optimization phases of the engineering design process.

For testing, a keyboard, video (monitor), mouse (KVM) was added to the backseat for engineer observation as shown in Figure 3.1. The monitor was fixed to the back of the passenger seat headrest using an in-house fabricated mounting apparatus. The wired keyboard and mouse were free-floating in the backseat for ease of access for the engineer. In the future we may want to implement a wireless remote-in setup using a laptop. The large graphical display is necessary for the engineer to observe all visualized icons and text.



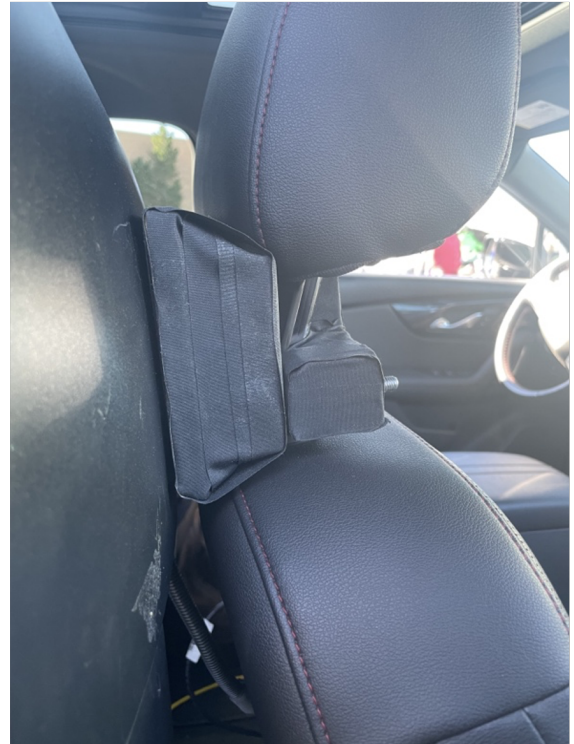
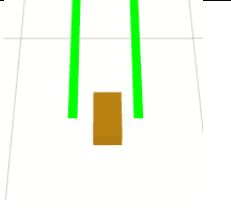
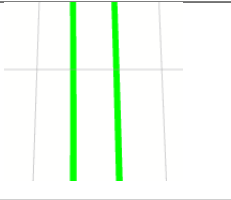
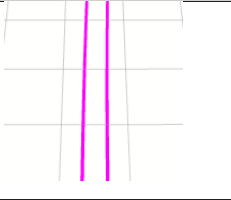
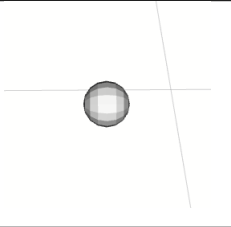
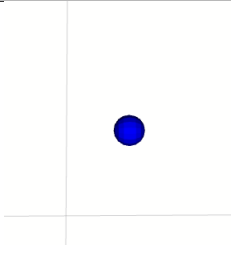
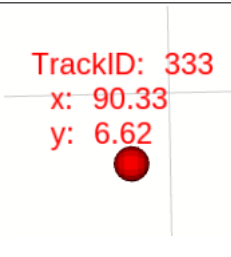
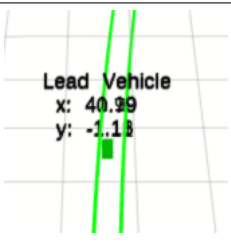


Figure 3.1: A video monitor mounted on the back of the passenger seat headrest for engineer observation from the backseat.

### **3.2 Data Displayed on Tank Viz**

Tank Viz is able to display the ego-vehicle, the lane boundaries of the lane it is in, front RADAR detections, front camera detections, tracks, and the lead vehicle. Table 3.1 shows what objects are displayed on Tank Viz.

Table 3.1: Tank Viz markers and descriptors

Marker	Marker Name & Description
	<p><b>Ego-Vehicle:</b> The ego-vehicle is represented as a brown box.</p>
	<p><b>Camera-Detected Lanes:</b> Lane boundaries appear as two green lines when lane data is provided by the Mobileye camera.</p>
	<p><b>Steering Angle Extrapolated Lanes:</b> Lane boundaries appear as two magenta lines when lane data is not provided by the Mobileye camera and is calculated from the steering angle.</p>
	<p><b>Camera Detection:</b> Camera detections are represented as small white spheres.</p>
	<p><b>RADAR Detection:</b> RADAR detections are represented as small blue spheres.</p>
	<p><b>Track:</b> Tracks are represented by medium red spheres accompanied by text that includes the track number and the track x-y position in meters, where x is the longitudinal offset and y is the lateral offset relative to the ego-vehicle.</p>
	<p><b>Lead Vehicle:</b> The Lead Vehicle is represented as a green box derived from the Sensor Fusion Node as the MIO, accompanied by text that includes the x-y position in meters, where x is the longitudinal offset and y is the lateral offset relative to the ego-vehicle.</p>

At startup, many windows are brought up automatically within the boot scripts, including terminals to see the live data values ticking as data is received, a window to visualize torque requests, and the Tank Viz window within the RViz package, which is further described in detail in the RViz Package Overview at <http://wiki.ros.org/rviz>. Figure 3.2 shows the windows that open when starting the Intel TANK, with Tank Viz in the top left corner.

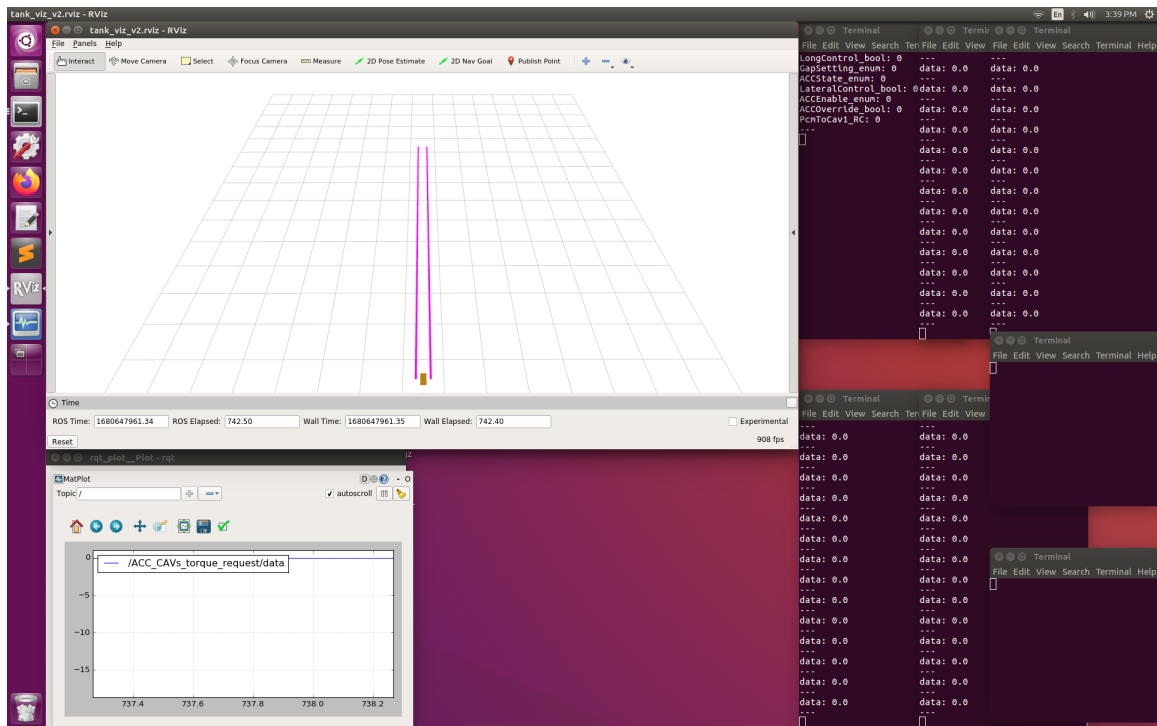


Figure 3.2: Intel Tank startup screen

Once data starts to come in, the visualization screen automatically fills up with icons referred to as markers representing the different objects perceived. A few examples of the screen and what to expect are shown in Figure 3.3, Figure 3.4, and Figure 3.5.

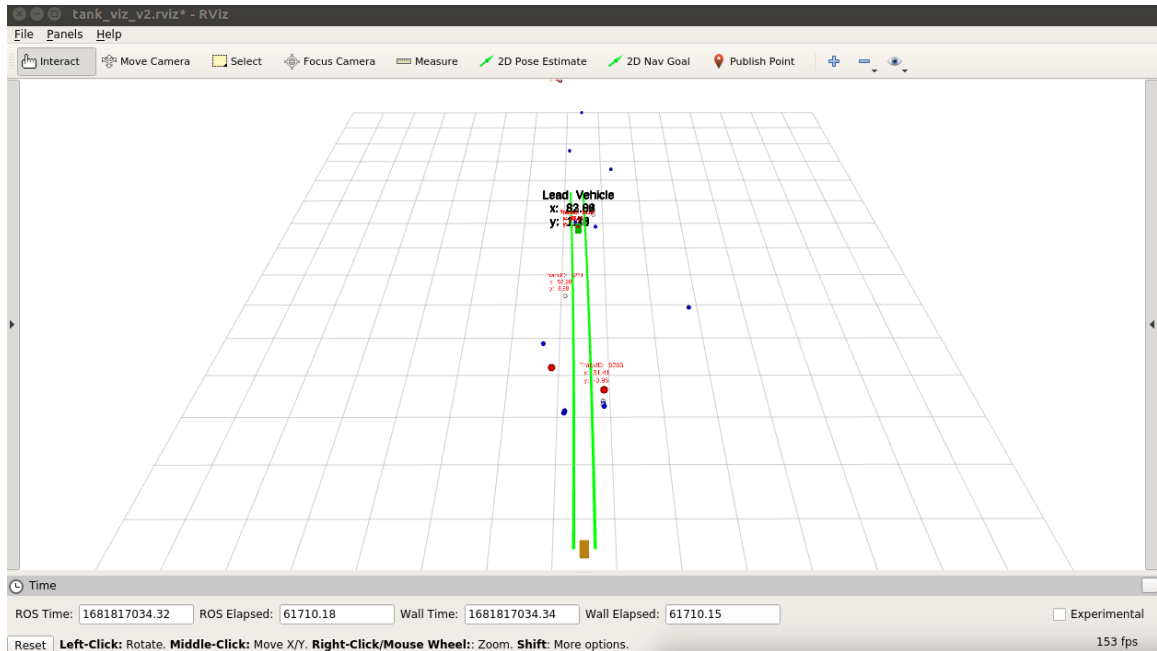


Figure 3.3: This is a typical view of what is perceived by the vehicle during highway driving. The Lead Vehicle is the nearest obstacle within the lane boundaries. Outside of the lane boundaries, other detections and tracks can be observed from obstacles in other lanes. In some instances, one can observe a cluster of camera and RADAR detections with an accompanying track. This shows that the sensor fusion node is taking those nearby detections and creating tracks. Although this is a freeze-frame and all objects may not reflect what could be observed even a second later, it gives a good view of how the perception system provides information about the environment.

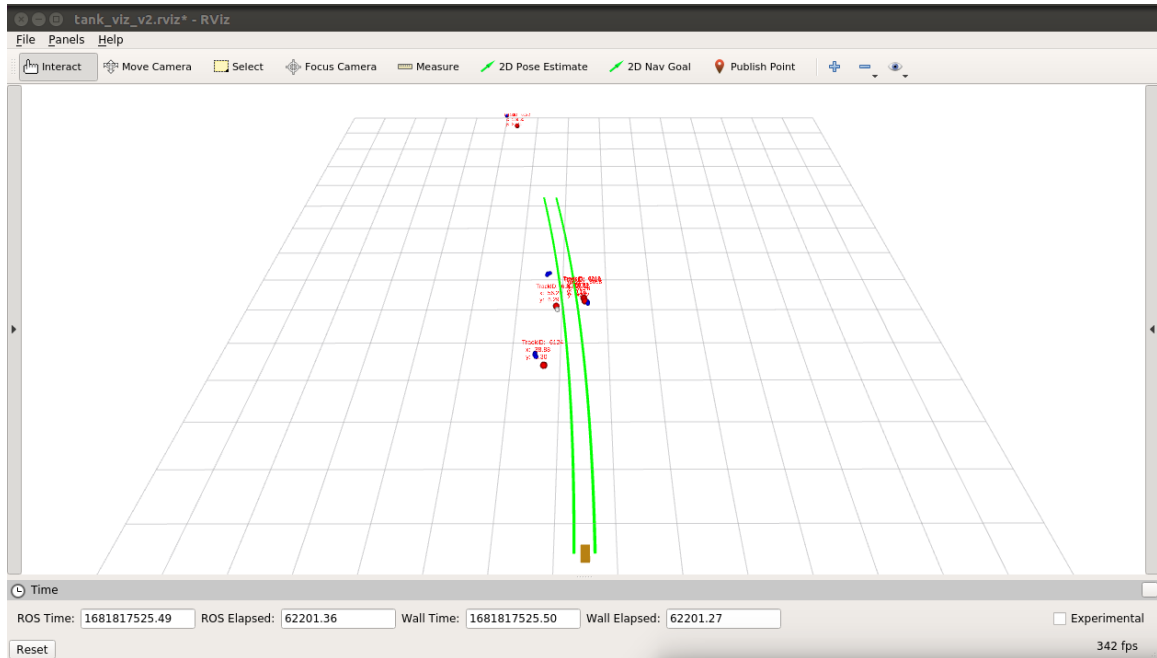


Figure 3.4: This is a capture emphasizing the green lane boundaries that are provided from camera detection data while driving through a neighborhood. Obstacles on the right side of the lane boundaries are objects on the side of the road, such as signs or bike riders. Objects on the left side of the lane boundaries represent oncoming traffic.

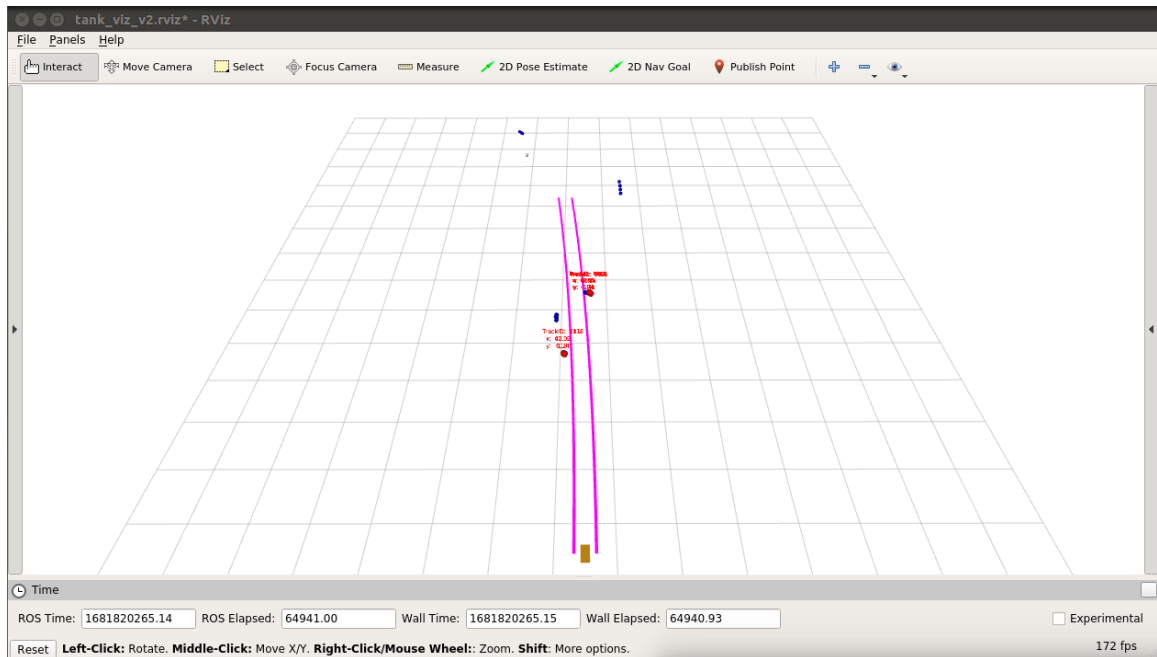


Figure 3.5: This is a capture emphasizing the magenta lane boundaries that are calculated from steering angle data while driving through a neighborhood. Obstacles on the right side of the lane boundaries are objects on the side of the road, such as signs or bike riders. Objects on the left side of the lane boundaries represent oncoming traffic.

Tank Viz comes with the option to update configurations. When the arrow on the left side of the window is expanded, a configuration panel is shown, and seen in Figure 3.6.

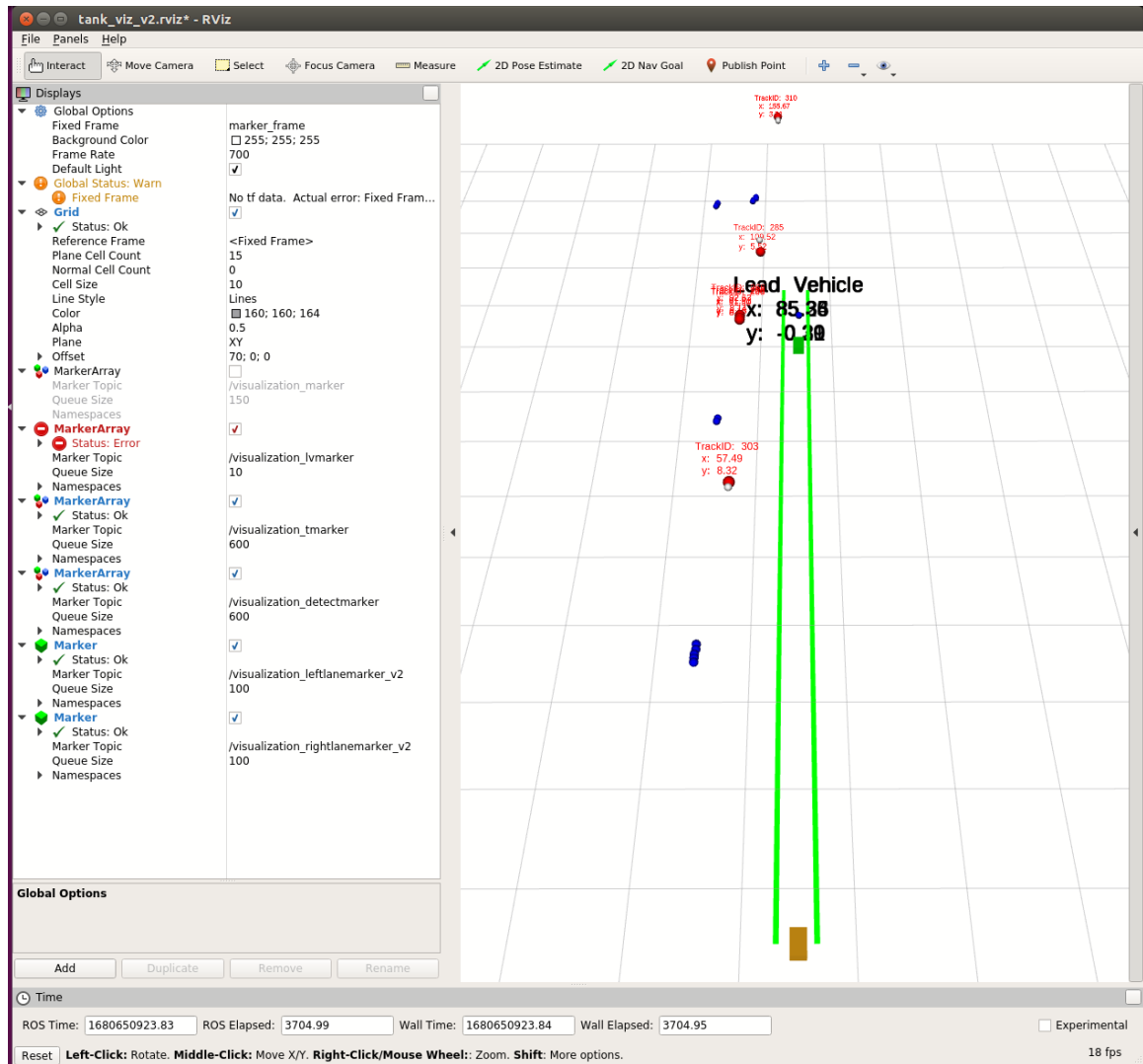


Figure 3.6: Visualization screen with populated data. On the left of the screen is the current configuration after it has been expanded by the left arrow.

The Tank Viz configuration panel, shown in Figure 3.7 handles all markers displayed on the screen. This visualization configuration includes the background, the grid, and markers. Markers and groups of markers may be added to the configuration by using the “Add” button at the bottom. Here we can see that one marker is used for the left lane boundary, the right lane boundary, and the lead vehicle. A marker array is used for detections and tracks. Object type boxes must be checked for the type to be seen on the display.

Under *Global Options*, the *Fixed Frame* name is used within the code to add markers to the correct environment. Essentially, it is the reference frame used to denote the “world” frame and should not be moving relative to the world. This is described later in the line-by-line software description. The background color can be chosen by the red/blue/green (RGB) color values. The frame rate in frames per second (fps) can also be modified here. A frame rate of 700 fps was chosen for the global environment for improved graphics that have smooth movements, comparable to popular games.

The Grid cell count, size, line style, RGB color, alpha, plane, and offset can be shown under *Grid*. The Alpha is set to 0.5 for the grid lines to be 50% transparent. The offset is at [ 70,0,0 ] in the configuration to move the grid back toward the ego-vehicle for operator cognition. The cell size is 10, representing 10 units of measure that are used within our code. For our purposes, the unit of measure is meters. This helps the operator easily see where detections are without requiring added text that will clutter up the screen. All objects appear on the x-y plane.

The markers and marker arrays take a marker topic as input, to tell RViz what ROSTopic to subscribe to for marker visualization. The ROSTopic will be further described in the line-by-line software description. The Queue Size handles how many of each type we are able to see in the frame at once. Their values here are greatly exaggerated to account for graphical smoothness as an object may move from one position to the next.



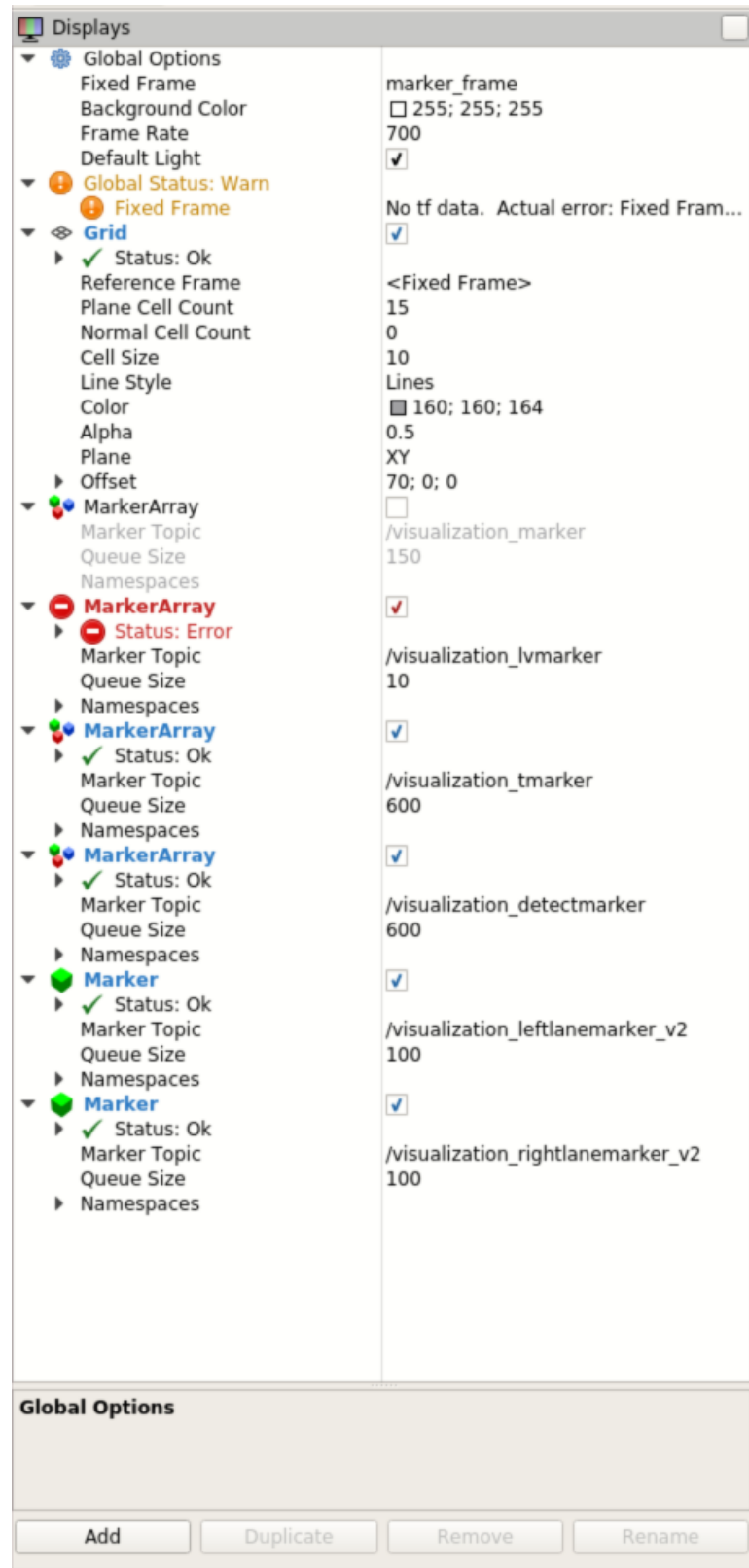


Figure 3.7: Configuration options within the config panel.

The configuration is saved in the *home/ < user > /.rviz* directory. The *< user >* name for the Tank is *ieisw*. The file open/save window is shown in Figure 3.8. The file can be changed by simply changing any of the parameters on the config user interface. When any parameters are changed or the visualization display has been moved to a different frame of view, the title of the window will include an asterisk \*, signifying that the file has been changed and needs to be saved.

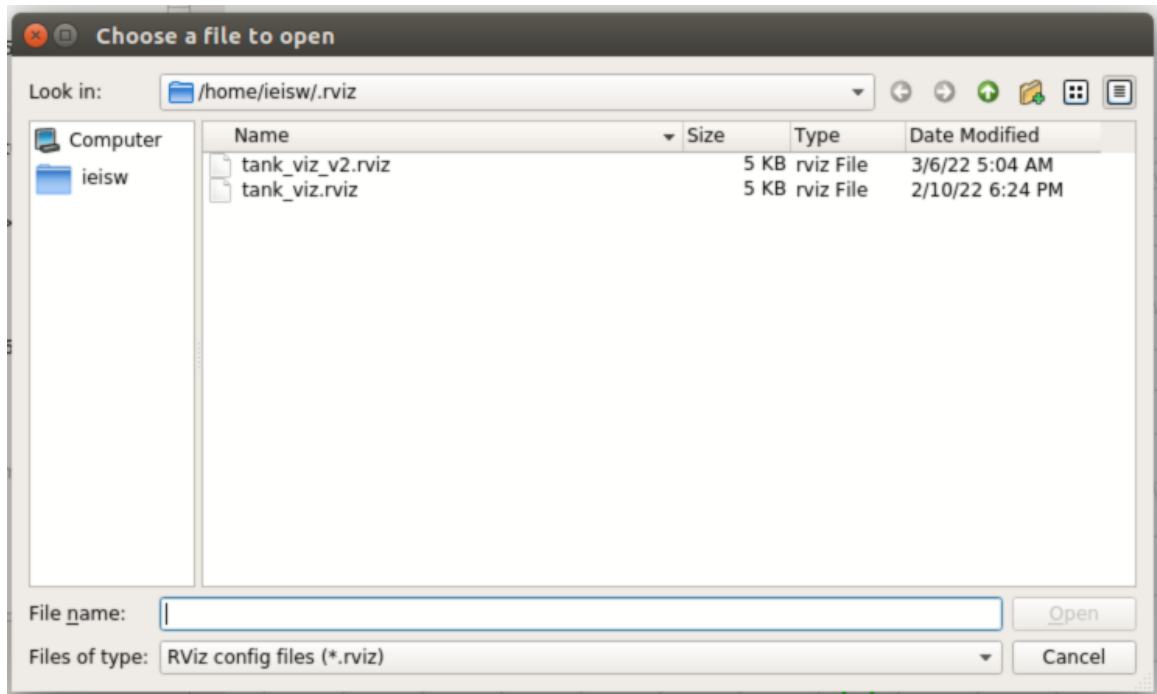


Figure 3.8: Config file open window

### 3.3 Software Architecture

The visualization scripts, as previously mentioned, are part of a cluster architecture, meaning multiple scripts are run to perform concurrent tasks. It was written this way because of two main reasons: 1) different items to be displayed on Tank Viz require different frame rates, and 2) software for graphics generation involves a great amount of throughput and output, thus having multiple scripts helps with load balancing by organizing the incoming and outgoing data under its own category. All scripts for visualization can

be found at `~/catkin_ws/src/using_markers/src/`, as shown in Figure 3.9. There are separate scripts for visualizing sensor detections (`viz_d.py`), sensor fused tracks (`viz_tracks.py`), lanes (`viz_lanes_v2.py`), and the lead vehicle (`viz_lv.py`). Version 1 of the lane detection script (`viz_lanes.py`) is obsolete and was replaced by the updated lane script (`viz_lanes_v2.py`) and therefore should be ignored.

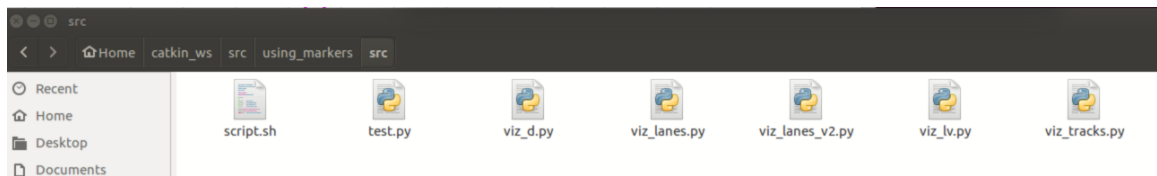


Figure 3.9: Scripts within the `using_markers` package

Before understanding the scripts, prior knowledge about Python subscribers and publishers, the RViz package, the Catkin workspace, and tool usage needs to be discussed.

### 3.3.1 Python Subscribers and Publishers

In ROS, communication between different nodes (software components) is achieved using a publish/subscribe model. Nodes can publish messages to a specific ROStopic, and other nodes can subscribe to that ROStopic to receive the messages. This approach provides a highly decoupled and modular system, allowing nodes to communicate without having to know about the specific details of the other nodes. A publisher node creates a ROStopic and sends messages to it, while a subscriber node receives the messages from the ROStopic. The messages sent and received can be of any data type, including custom message types defined by the developer. Many publishers and subscribers will be discussed in the line-by-line software description.

### 3.3.2 RViz Package Overview

RViz is a 3D visualization tool in ROS that allows users to visualize and interact with robot data in a 3D environment. RViz provides a Graphical User Interface (GUI) that enables

users to display, interpret, and interact with various types of data, such as robot models, sensor data, and maps.

RViz has a modular architecture that allows users to add and remove different displays and plugins according to their needs. Some of the displays available in RViz include the robot model display, which allows users to visualize the robot and its joints; the point cloud display, which enables users to visualize data from 3D sensors, such as LiDARs or depth cameras; and the map display, which displays 2D occupancy grid maps or 3D point clouds.

RViz also allows users to interact with the displayed data, enabling them to move the robot, modify the map, or adjust the parameters of the displays. Additionally, RViz provides a comprehensive set of tools for debugging and monitoring the robot's behavior, such as the TF tree display, which shows the robot's coordinate frames and transformations, and the plot display, which allows users to plot and analyze various types of data.

RViz is a powerful tool for robotics development and research, as it allows users to quickly prototype, debug, and test their algorithms in a realistic 3D environment. RViz also integrates seamlessly with other ROS packages, such as the Gazebo simulator, enabling users to simulate and test their robot systems in a virtual environment before deploying them on real hardware. Overall, RViz is a versatile and flexible tool for 3D visualization in ROS, providing users with a powerful set of features for interacting with and analyzing robot data.

### 3.3.3 Catkin Workspace and ROS Packages Needed

A Catkin workspace is a directory hierarchy used to develop and build ROS packages. Catkin is the build system used in ROS, and a Catkin workspace is the top-level directory where all the packages, build files, and other resources for a particular ROS project are stored.

A Catkin workspace typically contains two directories: `src` and `build`. The `src` directory contains all the source code for the ROS packages that are being developed, while the `build`

directory is where the build system generates the binaries and libraries.

When a new package is created in the `src` directory, the Catkin workspace needs to be re-built to reflect the changes. This can be done using the `catkin_make` command in the `catkin_ws` directory from terminal command line window. `catkin_make` will generate the necessary makefiles and build the packages in the workspace.

Catkin workspaces are used extensively in ROS development, as they provide an organized and modular way to develop, build, and distribute ROS packages. For visualization, we need to include the `using_markers` package to the catkin workspace. The `using_markers` package is a Catkin package in ROS that provides an example of how to use markers in RViz, which is a popular 3D visualization tool in ROS. Markers are visual objects such as arrows, spheres, and lines that can be used to represent various aspects of a robotic system, such as the robot itself, obstacles, and waypoints. The `using_markers` package has a wiki page that shows users how to use the `visualization_msgs/Marker` message type to create and display markers in RViz. When the `using_markers` package is built and launched in ROS, the `basic_shapes` node runs and publishes the markers to the chosen `/visualization_marker` ROS topic. These markers are then displayed in RViz, where they can be manipulated and viewed in 3D.

### **3.4 Real-Time and Post Processing Usage**

Tank Viz can be used during real-time operation while the vehicle platform is powered on, and in post-processing at lab test benches. ROS, with its capability to sync multiple nodes to the same virtual clock, is able to record data for synchronized playback amongst all actors. This recorded data can be played using simple ROS commands within the terminal. More information on ROS terminal commands can be found on the ROS wiki. When data is played back, simulated in Simulink to create tracks, or being fed in real-time from real obstacles in the environment, Tank Viz will be able to display these items.

### 3.5 Line-By-Line Software Description

Many of the Python scripts will contain the same general structure, so for the purpose of this thesis software description I will provide commentary for the automation script and one of the python pub-sub scripts. The remaining scripts can be found in the Appendix.

#### 3.5.1 Automation Script

---

```
1  #!/bin/bash
2
3  cd ~/catkin_ws/src/using_markers/src/ &
4  rosrun rviz rviz -d /home/ieisw/.rviz/tank_viz.rviz &
5  ./viz_d.py &
6  ./viz_tracks.py &
7  ./viz_lv.py &
8  ./viz_lanes.py &
```

Figure 3.10: The automation script `script.sh` included in the Tank `startup.sh` shell script used to open programs at system start up.

Figure 3.10 shows a bash script that runs several ROS nodes for visualizing markers, tracks, and lanes. Here’s a line-by-line breakdown of what the script does:

1. `#!/bin/bash`: This is known as a “shebang” line and tells the shell that this script should be run with the bash interpreter.

2. `cd ~/catkin_ws/src/using_markers/src/ &`: Changes the working directory to the folder where the ROS nodes are located. The `&` character at the end runs the command in the background so that the script can continue to run.

3. `roslaunch rviz rviz -d /home/ieisw/.rviz/tank_viz.rviz &`: Runs the RViz display that Tank Viz is built on with a configuration file that sets up the visualization on the tank. The `&` character at the end runs the command in the background so that the script can continue to run.

4. `./viz_d.py &`: Runs a Python script named `viz_d.py` that visualizes markers in RViz. The `&` character runs the command in the background.

5. `./viz_tracks.py &`: Runs a Python script named `viz_tracks.py` that visualizes tracks in RViz. The `&` character runs the command in the background.

6. `./viz_lv.py &`: Runs a Python script named `viz_lv.py` that visualizes lanes in RViz. The `&` character runs the command in the background.

7. `./viz_lanes.py &`: Runs a Python script named `viz_lanes.py` that visualizes a lane in RViz. The `&` character runs the command in the background.

Overall, this script sets up a ROS environment for visualizing several components in RViz.

### 3.5.2 Visualization Script

For the sake of describing how the visualizations scripts are written, I will use the `viz_tracks.py` script for the line-by-line description.

This code is a Python script that defines callbacks that handle data received from various ROS topics. The data is then used to update markers on Tank Viz using the RViz package.

Figure 3.11 shows that the script first imports the required ROS and `visualization_msgs` packages, as well as the `message_filters` package and the custom PCM message type, created by the Propulsion Controls and Modeling (PCM) EcoCAR subteam for vehicle diagnostic data publishing. It then initializes a publisher for a marker array on a specific ROStopic, sets up a ROS node, and sets a rate at which to publish markers. The publish rate is set to control the processor energy consumption. The node is then initialized, as well as the clock time, `markerArray` objects, and marker objects.

```

1  #!/usr/bin/env python
2  import rospy
3  import time
4  from visualization_msgs.msg import MarkerArray
5  from visualization_msgs.msg import Marker
6  import message_filters
7  from pcm_messages.msg import *
8
9  topic = '/visualization_tmarker'
10 publisher_t = rospy.Publisher(topic, MarkerArray, queue_size=600) #queue size 100
11 rospy.init_node('rviz_markers')
12 rate = rospy.Rate(200) # ROS Rate at 200Hz
13
14 global time_now
15 time_now = rospy.Time.now()
16
17 global markerArray
18 global marker
19 markerArray = MarkerArray()
20 markerArray.markers= []
21
22 marker = Marker()
23 ownship = Marker()
24 text = Marker()
25

```

Figure 3.11: The first chunk of code that imports required ROS packages, initializes a publisher, sets up a ROS node, and sets the ROS publish rate

Several markers are then defined, shown in Figure 3.12, including a sphere marker type, a text marker type, and a cube marker type to represent the ego-vehicle. Each marker is given a specific color and scale, as well as a position and orientation. They are all assigned to the same ROS time clock, shown in the `*.header.stamp` assignment. They are also added to the same frame, shown in the `*.header.frame` assignment so that they may be seen in the same global environment.



```

26 #         marker = Marker()
27 marker.header.frame_id = "marker_frame"
28 marker.header.stamp = time_now
29 marker.type = marker.SPHERE
30 marker.scale.x = 2
31 marker.scale.y = 2
32 marker.scale.z = 2
33 marker.color.a = 1.0 #make it visible
34 marker.pose.orientation.w = 1.0
35 marker.pose.position.z = 0.0
36 #         text = Marker()
37 text.header.frame_id = "marker_frame"
38 text.header.stamp = time_now
39 text.type = marker.TEXT_VIEW_FACING
40 text.scale.x = 2
41 text.scale.y = 2
42 text.scale.z = 2
43 text.color.a = 1.0 #make it visible
44 text.pose.orientation.w = 1.0
45 text.pose.position.z = 0.0
46 #         ownship = Marker()
47 ownship.header.frame_id = "marker_frame"
48 ownship.header.stamp = rospy.Time.now()
49 ownship.type = ownship.CUBE
50 ownship.scale.x = 2.5
51 ownship.scale.y = 1.5
52 ownship.scale.z = 1.0
53 ownship.color.a = 1.0 #make it visible
54 ownship.color.r = 1.0
55 ownship.color.g = 0.7
56 ownship.color.b = 0.1
57 ownship.pose.orientation.w = 1.0
58 ownship.pose.position.x = 0.0
59 ownship.pose.position.y = 0.0
60 ownship.pose.position.z = 0.0

```

Figure 3.12: The next chunk of code that defines track and ego-vehicle (referred to within the code as “ownship”) marker initialized parameters.

Next, the code screenshot in Figure 3.13 shows that a callback function is defined. Callback functions handle incoming data from different ROS topics and use the data to update the appropriate marker. For example, the `rtrackCallback` function updates the blue sphere marker with the data received from the `/trackDetection` topic. It also updates the position with data from the `/trackDetection` ROS topic, shown here to be the `dx` and `dy` values from the track data. The text markers are updated to also appear

near the position of the spherical track marker. In the code it is shown that the position updates using the dx and dy values from the track data, with a shift up 4 in the x direction to make the text appear above the spherical track marker. The lifetime duration for both the spherical marker and the text is set to 0.3 seconds, so that the track may linger on screen long enough for the engineering operator to see it. The team may choose to modify the lifetime duration of marker objects to match the collision avoidance industry standard for full autonomy of 50Hz (0.02 seconds) in the future [14].

```

115 def trackCallback(data):
116     if (data.dx < 1000000 and data.dy > 0):
117         marker.type = marker.SPHERE
118         marker.header.stamp = time_now
119         marker.scale.x = 1.5
120         marker.scale.y = 1.5
121         marker.scale.z = 1.5
122         marker.pose.position.x = data.dx
123         marker.pose.position.y = data.dy
124         #marker.id = data.trackNum #use this when every piece has a unique identifier
125         #make red markers
126         marker.color.r = 1.0
127         marker.color.g = 0.0
128         marker.color.b = 0.0
129         marker.lifetime = rospy.Duration(0.3)
130         text.scale.x = 1.5
131         text.scale.y = 1.5
132         text.scale.z = 1.5
133         text.pose.position.x = data.dx+4
134         text.pose.position.y = data.dy
135         text.color.r = 1.0
136         text.color.g = 0.0
137         text.color.b = 0.0
138         text.lifetime = rospy.Duration(0.3)
139         text.text = ("TrackID: %i \n x: %0.2f \n y: %0.2f" % (data.trackNum, data.dx, data.dy))
140         markerArray.markers.append(marker)
141         markerArray.markers.append(text)

```

Figure 3.13: The next chunk of code that defines callback functions.

Each callback function checks if the incoming data is valid before updating the marker. If the data is not valid, the function returns and the marker is not updated.

Finally, shown in Figure 3.14, the subscriber subscribes to all track ROSTopics, here showing that 20 are used, and each updated marker is appended to the marker array and published. While ros is operating, the marker ID is updated so not to replace markers but add to the end of a list. Then, the marker array is published. If this marker array is larger than 100, this will bog down the compute system, overfill the display, and cause

visualization performance to diminish, so the array is cleared and the most recent marker objects are added to the fresh list. The marker array is cleared before each update to prevent previous markers from lingering in the visualization. At this refresh time, the clock time is also refreshed. Within the main function, a try-except block tells the program to run unless there is a ROS interrupt that kills the process. A ROS interrupt may be initiated by either killing the process within a terminal using the signal number of the process, or by pressing ctrl-c within the terminal that the process is running in the foreground in. This exception call keeps the program from continuing to run despite the a call to kill the process and was a result of a prior implementation error.

```

142 def viz():
143     global markerArray
144     global time_now
145
146     # # #20 track topics
147     rospy.Subscriber("/Tracks1", Track, trackCallback)
148     rospy.Subscriber("/Tracks2", Track, trackCallback)
149     rospy.Subscriber("/Tracks3", Track, trackCallback)
150     rospy.Subscriber("/Tracks4", Track, trackCallback)
151     rospy.Subscriber("/Tracks5", Track, trackCallback)
152     rospy.Subscriber("/Tracks6", Track, trackCallback)
153     rospy.Subscriber("/Tracks7", Track, trackCallback)
154     rospy.Subscriber("/Tracks8", Track, trackCallback)
155     rospy.Subscriber("/Tracks9", Track, trackCallback)
156     rospy.Subscriber("/Tracks10", Track, trackCallback)
157     rospy.Subscriber("/Tracks11", Track, trackCallback)
158     rospy.Subscriber("/Tracks12", Track, trackCallback)
159     rospy.Subscriber("/Tracks13", Track, trackCallback)
160     rospy.Subscriber("/Tracks14", Track, trackCallback)
161     rospy.Subscriber("/Tracks15", Track, trackCallback)
162     rospy.Subscriber("/Tracks16", Track, trackCallback)
163     rospy.Subscriber("/Tracks17", Track, trackCallback)
164     rospy.Subscriber("/Tracks18", Track, trackCallback)
165     rospy.Subscriber("/Tracks19", Track, trackCallback)
166     rospy.Subscriber("/Tracks20", Track, trackCallback)
167
168     markerArray.markers.append(ownship)
169     while not rospy.is_shutdown():
170         i_d = 0
171         for m in markerArray.markers:
172             m.id = i_d
173             i_d += 1
174         publisher_t.publish(markerArray)
175         #print(len(markerArray.markers))
176         if (len(markerArray.markers) >= 100 ):
177             markerArray.markers = []
178             markerArray.markers.append(ownship)
179             time_now=rospy.Time.now()
180
181         rate.sleep()
182
183 if __name__ == '__main__':
184     try:
185         viz = viz()
186     except rospy.ROSInterruptException:
187         marker.action = Marker.DELETEALL
188         ownship.action = Marker.DELETEALL
189     pass

```

Figure 3.14: The final chunk of code that updates marker arrays and publishes the markers. This chunk also handles clearing of markers before updating again.

## **CHAPTER 4**

### **CONCLUSION AND FUTURE WORK**

In conclusion, a visualization tool is a necessary component for rapid prototyping in autonomous vehicle applications. As autonomous vehicle development continues to accelerate, the use of a visualization tool becomes increasingly important to ensure that engineers can rapidly and accurately design and test various scenarios, without the need for expensive physical prototypes or simulations. By providing a clear, intuitive representation of complex data, a visualization tool can help engineers identify potential issues, optimize performance, and ultimately speed up the development process. Therefore, the use of a visualization tool is essential for any team looking to develop effective and efficient autonomous vehicle applications.

The results are centered around the engineering operator experience. Being able to have a graphical representation of what the vehicle is perceiving allowed the team to make rapid adjustments during testing, rather than continuing extensive effort of collecting data to be post-processed for code revisions. For example, during a day of parking lot testing with highly reflective and metallic obstacles, we were not able to perceive RADAR detections on the display. This raised questions concerning RADAR calibration, and helped us to troubleshoot an issue and the necessary solution. Due to parking lot testing that caused us to make many consecutive left turns, the forward-facing RADAR had been knocked into a horizontal alignment fault. This was also an issue that support at Bosch was not aware of, since their testing does not include many consecutive single-direction turns. Without being able to perceive the lack of RADAR data on Tank Viz, we would have had to take empty data and try to troubleshoot the issue after the effort of gathering the needed data was carried out, adding more time and hassle for the engineer. Having a means of visualization helped to minimize erroneous results.

It's also important to note the approaches that were taken at visualizing perception data that did not lead to the desired results, so as to minimize the added effort of the team to try tactics that have already been tried. The first data representation that was tried was simply outputting text data to a terminal. With this configuration, a terminal window was opened for each output type – camera detection objects, RADAR detection objects, fused tracks, and the lead vehicle. The most important information given through this method was the x-y position of the object. With high frame rates from the sensors, along with the ability to output several objects at a time from each sensor and up to 20 fused tracks at a time, it was impossible and highly inefficient to follow any one object by observing the text, causing us to pause often during testing to try to keep track of objects. Another visualization tactic attempted involved recording the CAN data from the sensors during testing and replaying the logged data on the Vector CANoe program after data collection is complete. The first issue with this is that it does not provide real-time data visualization. While Vector CANoe is great for checking the CAN data flow, it is not great for visualizing the environment being perceived. A last attempt at visualizing the environment was using Simulink after data collection for sensor data playback. While this was a good method for understanding the environment in the early stages of perception system development, it again was only useful for after testing and not real-time visualization. It's possible that Simulink could be used for real-time visualization, but the team steered away from this approach after observing the heavy load that MATLAB was already putting on the processor and the great potential for a faulty or slow visualization program in the field. The RViz package was decided upon as a visualization package to build on because of its lightweight and easy integration into the ROS environment. The learning curve was low, which was perfect for rapid prototyping since our team was already behind from the previous year with CAVs development. Also, this simplified visualization package was perfect for operator use, and did not require rich or interactive visualization such as what may be obtained from the Foxglove Studio ROS visualization package for example.

Tank Viz provided a means for validation, by allowing the operator to have six degrees of freedom (6DOF) visual of what the Blazer is perceiving. In the long-run, we were able to create a more robust perception system for CAV system development, testing, and optimization. Only two versions were made within the year, with many git pushes of modifications. Modifications within the two versions can be observed on the pushes to other experimental branches. Having this tool was a contributing factor to our rapid prototyping from a 9th place CAVs system at the 2021 May competition to a successful 1st place CAVs system at the 2022 May final competition, securing more than 50% of 1st place awards in sub-categories and the 1st place overall award for Year 4 of the competition [15].

Succeeding engineers on the Georgia Tech EcoCAR team may find the referenced code on the Georgia Tech GitHub on the EcoCAR-Mobility private page. Members must be invited to view these resources. Once on the page, the referenced code can be found in the `Tank_catkin_ws` repository. Then navigate to `using_markers/src/` to find the visualization scripts.

#### **4.1 Tool Modification**

The team may find reasons to modify this tool that are outside of its original scope. For example, the team may want to add text to detections or change detection marker shapes to letters to represent the detected classifications of objects provided by the sensors. A use-case for this would be to record and analyze the effectiveness and possibly bias of pedestrian detection from the sensors. Another modification the team may want to make is adding neighboring lane boundaries from camera detection data when available, to give a more full and representative display of the environment. Modifications may be made and pushed to a new branch in GitHub to preserve the original contents. Modifications in the Python coding language may be made in the Tank Viz scripts. Modifications in the Bash scripting language may be made in the startup script for Tank Viz within the same folder. New markers may be added by simply importing necessary ROS packages, initializing

a publisher for the marker, defining marker initialization parameters, subscribing to the desired ROSTopic, and appending new markers to a marker array for display.



# Appendices

## APPENDIX A

### TANK VIZ SCRIPTS

#### A.0.1 viz\_d.py: Detection Visualization Script

```
1  #!/usr/bin/env python
2  import rospy
3  import time
4  from visualization_msgs.msg import MarkerArray
5  from visualization_msgs.msg import Marker
6  import message_filters
7  from pcm_messages.msg import *
8
9  topic = '/visualization_detectmarker'
10 publisher_d = rospy.Publisher(topic, MarkerArray, queue_size=600) #queue
    size 100
11 rospy.init_node('rviz_dmarkers')
12 rate = rospy.Rate(420) # ROS Rate at 420Hz
13
14 global time_now
15 time_now = rospy.Time.now()
16
17 global markerArray
18 global marker
19 markerArray = MarkerArray()
20 markerArray.markers= []
21
22 marker = Marker()
23 ownship = Marker()
24
25 # marker = Marker()
```

```

26 marker.header.frame_id = "marker_frame"
27 marker.header.stamp = time_now
28 marker.type = marker.SPHERE
29 marker.scale.x = 2
30 marker.scale.y = 2
31 marker.scale.z = 2
32 marker.color.a = 1.0 #make it visible
33 marker.pose.orientation.w = 1.0
34 marker.pose.position.z = 0.0
35 # ownship = Marker()
36 ownship.header.frame_id = "marker_frame"
37 ownship.header.stamp = rospy.Time.now()
38 ownship.type = ownship.CUBE
39 ownship.scale.x = 2.5
40 ownship.scale.y = 1.5
41 ownship.scale.z = 1.0
42 ownship.color.a = 1.0 #make it visible
43 ownship.color.r = 1.0
44 ownship.color.g = 0.7
45 ownship.color.b = 0.1
46 ownship.pose.orientation.w = 1.0
47 ownship.pose.position.x = 0.0
48 ownship.pose.position.y = 0.0
49 ownship.pose.position.z = 0.0
50
51 def rDetectionCallback(data):
52     if (data.wObstacle > 0.5):
53         marker.type = marker.SPHERE
54         marker.header.stamp = time_now
55         marker.scale.x = 1
56         marker.scale.y = 1
57         marker.scale.z = 1
58         marker.pose.position.x = data.dx - 4 #- 4.12

```

```

59     marker.pose.position.y = data.dy # + 1
60
61     #make blue markers
62     marker.color.r = 0.0
63     marker.color.g = 0.0
64     marker.color.b = 1.0
65     marker.lifetime = rospy.Duration(0.3)
66     markerArray.markers.append(marker)
67     #markerArray.markers.append(ownship)
68 else:
69     return
70 def mDetectionCallback(data):
71     marker.type = marker.SPHERE
72     marker.header.stamp = time_now
73     marker.scale.x = 1
74     marker.scale.y = 1
75     marker.scale.z = 1
76     marker.pose.position.x = data.PosX #+5.5#+ 1.38
77     marker.pose.position.y = data.PosY #opposite lateral direction as
78     radar
79     #make purple markers
80     marker.color.r = 1.0
81     marker.color.g = 1.0
82     marker.color.b = 1.0
83     marker.lifetime = rospy.Duration(0.3)
84     markerArray.markers.append(marker)
85     #markerArray.markers.append(ownship)
86 def leadVehCallback(data):
87     if (data.leadVehDist < 1000000):
88         marker.type = marker.CUBE
89         marker.header.stamp = time_now
90         marker.scale.x = 2.5

```

```

91     marker.scale.y = 1.5
92     marker.scale.z = 1.0
93     marker.pose.position.x = data.leadVehDist
94     marker.pose.position.y = data.leadVehLateralDist
95
96     #make green marker
97     marker.color.r = 0.0
98     marker.color.g = 1.0
99     marker.color.b = 0.0
100    marker.lifetime = rospy.Duration(0.3)
101    markerArray.markers.append(marker)
102 else:
103     return
104 def trackCallback(data):
105     if (data.dx < 1000000):
106         marker.type = marker.SPHERE
107         marker.header.stamp = time_now
108         marker.scale.x = 1.5
109         marker.scale.y = 1.5
110         marker.scale.z = 1.5
111         marker.pose.position.x = data.dx
112         marker.pose.position.y = data.dy
113         #marker.id = data.trackNum #use this when every piece has a unique
        identifier
114
115         #make red markers
116         marker.color.r = 1.0
117         marker.color.g = 0.0
118         marker.color.b = 0.0
119         marker.lifetime = rospy.Duration(0.3)
120         markerArray.markers.append(marker)
121     else:
122         return

```

```

123
124 def viz():
125     global markerArray
126     global time_now
127     # #1 lead vehicle verbose topic
128     # rospy.Subscriber("/lead_vehVERBOSE", LeadVehicleVerbose,
129         leadVehCallback)
130
131     #32 radar detection topics
132     rospy.Subscriber("/Radar1_Obj00_A", RadObjA, rDetectionCallback)
133     rospy.Subscriber("/Radar1_Obj01_A", RadObjA, rDetectionCallback)
134     rospy.Subscriber("/Radar1_Obj02_A", RadObjA, rDetectionCallback)
135     rospy.Subscriber("/Radar1_Obj03_A", RadObjA, rDetectionCallback)
136     rospy.Subscriber("/Radar1_Obj04_A", RadObjA, rDetectionCallback)
137     rospy.Subscriber("/Radar1_Obj05_A", RadObjA, rDetectionCallback)
138     rospy.Subscriber("/Radar1_Obj06_A", RadObjA, rDetectionCallback)
139     rospy.Subscriber("/Radar1_Obj07_A", RadObjA, rDetectionCallback)
140     rospy.Subscriber("/Radar1_Obj08_A", RadObjA, rDetectionCallback)
141     rospy.Subscriber("/Radar1_Obj09_A", RadObjA, rDetectionCallback)
142     rospy.Subscriber("/Radar1_Obj10_A", RadObjA, rDetectionCallback)
143     rospy.Subscriber("/Radar1_Obj11_A", RadObjA, rDetectionCallback)
144     rospy.Subscriber("/Radar1_Obj12_A", RadObjA, rDetectionCallback)
145     rospy.Subscriber("/Radar1_Obj13_A", RadObjA, rDetectionCallback)
146     rospy.Subscriber("/Radar1_Obj14_A", RadObjA, rDetectionCallback)
147     rospy.Subscriber("/Radar1_Obj15_A", RadObjA, rDetectionCallback)
148     rospy.Subscriber("/Radar1_Obj16_A", RadObjA, rDetectionCallback)
149     rospy.Subscriber("/Radar1_Obj17_A", RadObjA, rDetectionCallback)
150     rospy.Subscriber("/Radar1_Obj18_A", RadObjA, rDetectionCallback)
151     rospy.Subscriber("/Radar1_Obj19_A", RadObjA, rDetectionCallback)
152     rospy.Subscriber("/Radar1_Obj20_A", RadObjA, rDetectionCallback)
153     rospy.Subscriber("/Radar1_Obj21_A", RadObjA, rDetectionCallback)
154     rospy.Subscriber("/Radar1_Obj22_A", RadObjA, rDetectionCallback)
155     rospy.Subscriber("/Radar1_Obj23_A", RadObjA, rDetectionCallback)

```

```

155 rospy.Subscriber("/Radar1_Obj24_A", RadObjA, rDetectionCallback)
156 rospy.Subscriber("/Radar1_Obj25_A", RadObjA, rDetectionCallback)
157 rospy.Subscriber("/Radar1_Obj26_A", RadObjA, rDetectionCallback)
158 rospy.Subscriber("/Radar1_Obj27_A", RadObjA, rDetectionCallback)
159 rospy.Subscriber("/Radar1_Obj28_A", RadObjA, rDetectionCallback)
160 rospy.Subscriber("/Radar1_Obj29_A", RadObjA, rDetectionCallback)
161 rospy.Subscriber("/Radar1_Obj30_A", RadObjA, rDetectionCallback)
162 rospy.Subscriber("/Radar1_Obj31_A", RadObjA, rDetectionCallback)
163
164 # #10 mobileye detection topics
165 rospy.Subscriber("/ObstacleDataA1", ME_A2, mDetectionCallback)
166 rospy.Subscriber("/ObstacleDataA2", ME_A2, mDetectionCallback)
167 rospy.Subscriber("/ObstacleDataA3", ME_A2, mDetectionCallback)
168 rospy.Subscriber("/ObstacleDataA4", ME_A2, mDetectionCallback)
169 rospy.Subscriber("/ObstacleDataA5", ME_A2, mDetectionCallback)
170 rospy.Subscriber("/ObstacleDataA6", ME_A2, mDetectionCallback)
171 rospy.Subscriber("/ObstacleDataA7", ME_A2, mDetectionCallback)
172 rospy.Subscriber("/ObstacleDataA8", ME_A2, mDetectionCallback)
173 rospy.Subscriber("/ObstacleDataA9", ME_A2, mDetectionCallback)
174 rospy.Subscriber("/ObstacleDataA10", ME_A2, mDetectionCallback)
175
176 # # #20 track topics
177 # rospy.Subscriber("/Tracks1", Track, trackCallback)
178 # rospy.Subscriber("/Tracks2", Track, trackCallback)
179 # rospy.Subscriber("/Tracks3", Track, trackCallback)
180 # rospy.Subscriber("/Tracks4", Track, trackCallback)
181 # rospy.Subscriber("/Tracks5", Track, trackCallback)
182 # rospy.Subscriber("/Tracks6", Track, trackCallback)
183 # rospy.Subscriber("/Tracks7", Track, trackCallback)
184 # rospy.Subscriber("/Tracks8", Track, trackCallback)
185 # rospy.Subscriber("/Tracks9", Track, trackCallback)
186 # rospy.Subscriber("/Tracks10", Track, trackCallback)
187 # rospy.Subscriber("/Tracks11", Track, trackCallback)

```

```

188 # rospy.Subscriber("/Tracks12", Track, trackCallback)
189 # rospy.Subscriber("/Tracks13", Track, trackCallback)
190 # rospy.Subscriber("/Tracks14", Track, trackCallback)
191 # rospy.Subscriber("/Tracks15", Track, trackCallback)
192 # rospy.Subscriber("/Tracks16", Track, trackCallback)
193 # rospy.Subscriber("/Tracks17", Track, trackCallback)
194 # rospy.Subscriber("/Tracks18", Track, trackCallback)
195 # rospy.Subscriber("/Tracks19", Track, trackCallback)
196 # rospy.Subscriber("/Tracks20", Track, trackCallback)
197
198 markerArray.markers.append(ownship)
199 while not rospy.is_shutdown():
200     i_d = 0
201     for m in markerArray.markers:
202         m.id = i_d
203         i_d += 1
204     publisher_d.publish(markerArray)
205     #print(len(markerArray.markers))
206     if (len(markerArray.markers) >= 420 ):
207         markerArray.markers = []
208         markerArray.markers.append(ownship)
209         time_now=rospy.Time.now()
210
211     rate.sleep()
212
213 if __name__ == '__main__':
214     try:
215         viz = viz()
216     except rospy.ROSInterruptException:
217         marker.action = Marker.DELETEALL
218         ownship.action = Marker.DELETEALL
219     pass

```



## A.0.2 viz\_lanes\_v2.py: Lane Detection Visualization Script

```
1 #!/usr/bin/env python
2 import rospy
3 import time
4 from visualization_msgs.msg import MarkerArray
5 from visualization_msgs.msg import Marker
6 from geometry_msgs.msg import Point
7 from pcm_messages.msg import *
8 from GT_CAVs_ROS_Messages.msg import PcmToCav2,
   SFNode_Utilized_Lane_Info
9 import math
10
11 # initialize publishers
12 topicl = '/visualization_leftlanemarker_v2'
13 topicr = '/visualization_rightlanemarker_v2'
14 publisher_leftlane = rospy.Publisher(topicl, Marker, queue_size=100) #
   queue size 100
15 publisher_rightlane = rospy.Publisher(topicr, Marker, queue_size=100) #
   queue size 100
16
17 # initialize global vars
18 curvature = 0
19 heading = 0
20 leftOffset = 0
21 rightOffset = 0
22 laneSource = 0
23
24 def sfNodeLaneCallback(data):
25     global curvature
26     global heading
27     global leftOffset
28     global rightOffset
```

```

29  global laneSource
30
31  # get parameters from ROS message
32  curvature = data.curvature
33  heading = data.heading
34  leftOffset = data.leftOffset
35  rightOffset = data.rightOffset
36  laneSource = data.source
37
38
39
40
41  def plot():
42      global curvature
43      global heading
44      global leftOffset
45      global rightOffset
46      global laneSource
47
48      # setup marker arrays for lanes
49      markerArray = MarkerArray()
50      leftLane = Marker()
51      rightLane = Marker()
52
53      # set opacity
54      leftLane.color.a = 1.0
55      rightLane.color.a = 1.0
56
57      # set color based on source
58      if laneSource == 1: # data coming from ME
59          leftLane.color.r = 0.0
60          leftLane.color.g = 1.0
61          leftLane.color.b = 0.0

```

```

62     else:
63         leftLane.color.r = 1.0
64         leftLane.color.g = 0.0
65         leftLane.color.b = 1.0
66
67     if laneSource == 1: # data coming from ME
68         rightLane.color.r = 0.0
69         rightLane.color.g = 1.0
70         rightLane.color.b = 0.0
71     else:
72         rightLane.color.r = 1.0
73         rightLane.color.g = 0.0
74         rightLane.color.b = 1.0
75
76     # set other necessary fields
77     leftLane.type = leftLane.LINE_STRIP
78     leftLane.header.stamp = rospy.Time.now()
79     leftLane.header.frame_id = "marker_frame"
80     leftLane.action = leftLane.ADD
81     leftLane.id = 1
82     leftLane.pose.orientation.w = 1.0
83     leftLane.scale.x = 0.5
84     leftLane.scale.y = 0.5
85     leftLane.scale.z = 0.5
86     leftLane.pose.position.x = 0
87     leftLane.pose.position.y = 0
88     leftLane.pose.position.z = 0
89
90     rightLane.type = rightLane.LINE_STRIP
91     rightLane.header.stamp = rospy.Time.now()
92     rightLane.header.frame_id = "marker_frame"
93     rightLane.action = rightLane.ADD
94     rightLane.id = 1

```

```

95     rightLane.pose.orientation.w = 1.0
96     rightLane.scale.x = 0.5
97     rightLane.scale.y = 0.5
98     rightLane.scale.z = 0.5
99     rightLane.pose.position.x = 0
100    rightLane.pose.position.y = 0
101    rightLane.pose.position.z = 0
102
103    # create points for left lane
104    for i in range(100):
105        p = Point()
106        equation = (curvature*i**2) + (heading*i) + leftOffset
107        p.x = i
108        p.y = equation
109        p.z = 0
110        leftLane.points.append(p)
111
112    # publish left lane
113    # print("publishing left lane")
114    publisher_leftlane.publish(leftLane)
115
116
117    # create points for left lane
118    for i in range(100):
119        p = Point()
120        equation = (curvature*i**2) + (heading*i) + rightOffset
121        p.x = i
122        p.y = equation
123        p.z = 0
124        rightLane.points.append(p)
125
126    # publish left lane
127    publisher_rightlane.publish(rightLane)

```

```

128
129 def viz():
130     rospy.init_node('rviz_lane')
131     rate = rospy.Rate(10) # ROS Rate at 10Hz
132
133     lane_subscriber = rospy.Subscriber("/SFNode_Utilized_Lane_Data",
134                                         SFNode_Utilized_Lane_Info, sfNodeLaneCallback)
135
136     while not rospy.is_shutdown():
137         plot()
138         # print("in main loop")
139         rate.sleep()
140
141 if __name__ == '__main__':
142     try:
143         viz = viz()
144     except rospy.ROSInterruptException:
145         marker.action = Marker.DELETEALL
146         ownship.action = Marker.DELETEALL
147     pass

```

### A.0.3 viz\_lv.py: Lead Vehicle Visualization Script

```

1 #!/usr/bin/env python
2 import rospy
3 import time
4 from visualization_msgs.msg import MarkerArray
5 from visualization_msgs.msg import Marker
6 from pcm_messages.msg import *
7
8 topic = '/visualization_lvmarker'
9 #publisher = rospy.Publisher(topic, MarkerArray,queue_size=10) #queue
    size 100

```

```

10 publisher_leadveh = rospy.Publisher(topic, MarkerArray, queue_size=10) #
    queue size 100
11 rospy.init_node('rviz_leadveh')
12 rate = rospy.Rate(10) # ROS Rate at 10Hz
13
14 global markerArray
15 global marker
16 markerArray = MarkerArray()
17 markerArray.markers= []
18
19 marker = Marker()
20 ownship = Marker()
21 text = Marker()
22
23 # marker = Marker()
24 marker.header.frame_id = "marker_frame"
25 marker.header.stamp = rospy.Time.now()
26 marker.type = marker.SPHERE
27 marker.scale.x = 2
28 marker.scale.y = 2
29 marker.scale.z = 2
30 marker.color.a = 1.0 #make it visible
31 marker.pose.orientation.w = 1.0
32 marker.pose.position.z = 0.0
33 # text = Marker()
34 text.header.frame_id = "marker_frame"
35 text.header.stamp = rospy.Time.now()
36 text.type = marker.TEXT_VIEW_FACING
37 text.scale.x = 2
38 text.scale.y = 2
39 text.scale.z = 2
40 text.color.a = 1.0 #make it visible
41 text.pose.orientation.w = 1.0

```

```

42 text.pose.position.z = 0.0
43 # ownship = Marker()
44 ownship.header.frame_id = "marker_frame"
45 ownship.header.stamp = rospy.Time.now()
46 ownship.type = ownship.CUBE
47 ownship.scale.x = 2.5
48 ownship.scale.y = 1.5
49 ownship.scale.z = 1.0
50 ownship.color.a = 1.0 #make it visible
51 ownship.color.r = 1.0
52 ownship.color.g = 0.7
53 ownship.color.b = 0.1
54 ownship.pose.orientation.w = 1.0
55 ownship.pose.position.x = 0.0
56 ownship.pose.position.y = 0.0
57 ownship.pose.position.z = 0.0
58
59 def rDetectionCallback(data):
60     if (data.wExist > -0.7):
61         marker.type = marker.SPHERE
62         marker.header.stamp = rospy.Time.now()
63         marker.scale.x = 1
64         marker.scale.y = 1
65         marker.scale.z = 1
66         marker.pose.position.x = data.dx
67         marker.pose.position.y = data.dy
68
69         #make blue markers
70         marker.color.r = 0.0
71         marker.color.g = 0.0
72         marker.color.b = 1.0
73         marker.lifetime = rospy.Duration(0.1)
74         markerArray.markers.append(marker)

```

```

75     #markerArray.markers.append(ownship)
76 else:
77     return
78 def mDetectionCallback(data):
79     marker.type = marker.SPHERE
80     marker.header.stamp = rospy.Time.now()
81     marker.scale.x = 1
82     marker.scale.y = 1
83     marker.scale.z = 1
84     marker.pose.position.x = data.PosX
85     marker.pose.position.y = data.Posy
86
87     #make purple markers
88     marker.color.r = 1.0
89     marker.color.g = 1.0
90     marker.color.b = 1.0
91     marker.lifetime = rospy.Duration(0.1)
92     markerArray.markers.append(marker)
93     #markerArray.markers.append(ownship)
94 def leadVehCallback(data):
95     marker.type = marker.CUBE
96     marker.header.stamp = rospy.Time.now()
97     marker.scale.x = 2.5
98     marker.scale.y = 1.5
99     marker.scale.z = 1.0
100    marker.pose.position.x = data.leadVehDist
101    marker.pose.position.y = data.leadVehLateralDist
102    #make green marker
103    marker.color.r = 0.0
104    marker.color.g = 1.0
105    marker.color.b = 0.0
106    marker.lifetime = rospy.Duration(0.3)
107    text.scale.x = 3.5

```



```

108 text.scale.y = 3.5
109 text.scale.z = 3.5
110 text.pose.position.x = data.leadVehDist+8
111 text.pose.position.y = data.leadVehLateralDist
112 text.color.r = 0.0
113 text.color.g = 0.0
114 text.color.b = 0.0
115 text.lifetime = rospy.Duration(0.3)
116 text.text = ("Lead Vehicle\n x: %0.2f \n y: %0.2f" % (data.leadVehDist
    , data.leadVehLateralDist))
117 if (marker.pose.position.x > 10000):
118     pass
119 else:
120     markerArray.markers.append(marker)
121     markerArray.markers.append(text)
122     #lvma.markers.append(marker)
123     #lvma.markers.append(ownship)
124 def trackCallback(data):
125     if (data.dx < 1000000):
126         marker.type = marker.SPHERE
127         marker.header.stamp = rospy.Time.now()
128         marker.scale.x = 1.5
129         marker.scale.y = 1.5
130         marker.scale.z = 1.5
131         marker.pose.position.x = data.dx
132         marker.pose.position.y = data.dy
133         #marker.id = data.trackNum #use this when every piece has a unique
            identifier
134
135         #make red markers
136         marker.color.r = 1.0
137         marker.color.g = 0.0
138         marker.color.b = 0.0

```

```

139     marker.lifetime = rospy.Duration(0.1)
140     markerArray.markers.append(marker)
141 else:
142     return
143
144 def viz():
145     global markerArray
146     #1 lead vehicle verbose topic
147     rospy.Subscriber("/lead_vchVERBOSE", LeadVehicleVerbose,
148                     leadVehCallback)
149
150     #32 radar detection topics
151     # rospy.Subscriber("/Radar1_Obj00_A", RadObjA, rDetectionCallback)
152     # rospy.Subscriber("/Radar1_Obj01_A", RadObjA, rDetectionCallback)
153     # rospy.Subscriber("/Radar1_Obj02_A", RadObjA, rDetectionCallback)
154     # rospy.Subscriber("/Radar1_Obj03_A", RadObjA, rDetectionCallback)
155     # rospy.Subscriber("/Radar1_Obj04_A", RadObjA, rDetectionCallback)
156     # rospy.Subscriber("/Radar1_Obj05_A", RadObjA, rDetectionCallback)
157     # rospy.Subscriber("/Radar1_Obj06_A", RadObjA, rDetectionCallback)
158     # rospy.Subscriber("/Radar1_Obj07_A", RadObjA, rDetectionCallback)
159     # rospy.Subscriber("/Radar1_Obj08_A", RadObjA, rDetectionCallback)
160     # rospy.Subscriber("/Radar1_Obj09_A", RadObjA, rDetectionCallback)
161     # rospy.Subscriber("/Radar1_Obj10_A", RadObjA, rDetectionCallback)
162     # rospy.Subscriber("/Radar1_Obj11_A", RadObjA, rDetectionCallback)
163     # rospy.Subscriber("/Radar1_Obj12_A", RadObjA, rDetectionCallback)
164     # rospy.Subscriber("/Radar1_Obj13_A", RadObjA, rDetectionCallback)
165     # rospy.Subscriber("/Radar1_Obj14_A", RadObjA, rDetectionCallback)
166     # rospy.Subscriber("/Radar1_Obj15_A", RadObjA, rDetectionCallback)
167     # rospy.Subscriber("/Radar1_Obj16_A", RadObjA, rDetectionCallback)
168     # rospy.Subscriber("/Radar1_Obj17_A", RadObjA, rDetectionCallback)
169     # rospy.Subscriber("/Radar1_Obj18_A", RadObjA, rDetectionCallback)
170     # rospy.Subscriber("/Radar1_Obj19_A", RadObjA, rDetectionCallback)
171     # rospy.Subscriber("/Radar1_Obj20_A", RadObjA, rDetectionCallback)

```

```

171 # rospy.Subscriber("/Radar1_Obj21_A", RadObjA, rDetectionCallback)
172 # rospy.Subscriber("/Radar1_Obj22_A", RadObjA, rDetectionCallback)
173 # rospy.Subscriber("/Radar1_Obj23_A", RadObjA, rDetectionCallback)
174 # rospy.Subscriber("/Radar1_Obj24_A", RadObjA, rDetectionCallback)
175 # rospy.Subscriber("/Radar1_Obj25_A", RadObjA, rDetectionCallback)
176 # rospy.Subscriber("/Radar1_Obj26_A", RadObjA, rDetectionCallback)
177 # rospy.Subscriber("/Radar1_Obj27_A", RadObjA, rDetectionCallback)
178 # rospy.Subscriber("/Radar1_Obj28_A", RadObjA, rDetectionCallback)
179 # rospy.Subscriber("/Radar1_Obj29_A", RadObjA, rDetectionCallback)
180 # rospy.Subscriber("/Radar1_Obj30_A", RadObjA, rDetectionCallback)
181 # rospy.Subscriber("/Radar1_Obj31_A", RadObjA, rDetectionCallback)
182
183 #10 mobileye detection topics
184 # rospy.Subscriber("/ObstacleDataA1", ME_A2, mDetectionCallback)
185 # rospy.Subscriber("/ObstacleDataA2", ME_A2, mDetectionCallback)
186 # rospy.Subscriber("/ObstacleDataA3", ME_A2, mDetectionCallback)
187 # rospy.Subscriber("/ObstacleDataA4", ME_A2, mDetectionCallback)
188 # rospy.Subscriber("/ObstacleDataA5", ME_A2, mDetectionCallback)
189 # rospy.Subscriber("/ObstacleDataA6", ME_A2, mDetectionCallback)
190 # rospy.Subscriber("/ObstacleDataA7", ME_A2, mDetectionCallback)
191 # rospy.Subscriber("/ObstacleDataA8", ME_A2, mDetectionCallback)
192 # rospy.Subscriber("/ObstacleDataA9", ME_A2, mDetectionCallback)
193 # rospy.Subscriber("/ObstacleDataA10", ME_A2, mDetectionCallback)
194
195 #20 track topics
196 # rospy.Subscriber("/Tracks1", Track, trackCallback)
197 # rospy.Subscriber("/Tracks2", Track, trackCallback)
198 # rospy.Subscriber("/Tracks3", Track, trackCallback)
199 # rospy.Subscriber("/Tracks4", Track, trackCallback)
200 # rospy.Subscriber("/Tracks5", Track, trackCallback)
201 # rospy.Subscriber("/Tracks6", Track, trackCallback)
202 # rospy.Subscriber("/Tracks7", Track, trackCallback)
203 # rospy.Subscriber("/Tracks8", Track, trackCallback)

```

```

204 # rospy.Subscriber("/Tracks9", Track, trackCallback)
205 # rospy.Subscriber("/Tracks10", Track, trackCallback)
206 # rospy.Subscriber("/Tracks11", Track, trackCallback)
207 # rospy.Subscriber("/Tracks12", Track, trackCallback)
208 # rospy.Subscriber("/Tracks13", Track, trackCallback)
209 # rospy.Subscriber("/Tracks14", Track, trackCallback)
210 # rospy.Subscriber("/Tracks15", Track, trackCallback)
211 # rospy.Subscriber("/Tracks16", Track, trackCallback)
212 # rospy.Subscriber("/Tracks17", Track, trackCallback)
213 # rospy.Subscriber("/Tracks18", Track, trackCallback)
214 # rospy.Subscriber("/Tracks19", Track, trackCallback)
215 # rospy.Subscriber("/Tracks20", Track, trackCallback)
216
217 markerArray.markers.append(ownship)
218 while not rospy.is_shutdown():
219     i_d = 0
220     for m in markerArray.markers:
221         m.id = i_d
222         i_d += 1
223     publisher_leadveh.publish(markerArray)
224     #print(len(markerArray.markers))
225     if (len(markerArray.markers) >= 1000 ):
226         markerArray.markers = []
227     rate.sleep()
228
229 if __name__ == '__main__':
230     try:
231         viz = viz()
232     except rospy.ROSInterruptException:
233         marker.action = Marker.DELETEALL
234         ownship.action = Marker.DELETEALL
235     pass

```

#### A.0.4 viz\_tracks.py: Track Visualization Script

```
1 #!/usr/bin/env python
2 import rospy
3 import time
4 from visualization_msgs.msg import MarkerArray
5 from visualization_msgs.msg import Marker
6 import message_filters
7 from pcm_messages.msg import *
8
9 topic = '/visualization_tmarker'
10 publisher_t = rospy.Publisher(topic, MarkerArray, queue_size=600) #queue
    size 100
11 rospy.init_node('rviz_markers')
12 rate = rospy.Rate(200) # ROS Rate at 200Hz
13
14 global time_now
15 time_now = rospy.Time.now()
16
17 global markerArray
18 global marker
19 markerArray = MarkerArray()
20 markerArray.markers= []
21
22 marker = Marker()
23 ownship = Marker()
24 text = Marker()
25
26 # marker = Marker()
27 marker.header.frame_id = "marker_frame"
28 marker.header.stamp = time_now
29 marker.type = marker.SPHERE
30 marker.scale.x = 2
```

```

31 marker.scale.y = 2
32 marker.scale.z = 2
33 marker.color.a = 1.0 #make it visible
34 marker.pose.orientation.w = 1.0
35 marker.pose.position.z = 0.0
36 # text = Marker()
37 text.header.frame_id = "marker_frame"
38 text.header.stamp = time_now
39 text.type = marker.TEXT_VIEW_FACING
40 text.scale.x = 2
41 text.scale.y = 2
42 text.scale.z = 2
43 text.color.a = 1.0 #make it visible
44 text.pose.orientation.w = 1.0
45 text.pose.position.z = 0.0
46 # ownship = Marker()
47 ownship.header.frame_id = "marker_frame"
48 ownship.header.stamp = rospy.Time.now()
49 ownship.type = ownship.CUBE
50 ownship.scale.x = 2.5
51 ownship.scale.y = 1.5
52 ownship.scale.z = 1.0
53 ownship.color.a = 1.0 #make it visible
54 ownship.color.r = 1.0
55 ownship.color.g = 0.7
56 ownship.color.b = 0.1
57 ownship.pose.orientation.w = 1.0
58 ownship.pose.position.x = 0.0
59 ownship.pose.position.y = 0.0
60 ownship.pose.position.z = 0.0
61
62 def rDetectionCallback(data):
63     if (data.wExist > 0.7):

```

```

64     marker.type = marker.SPHERE
65     marker.header.stamp = time_now
66     marker.scale.x = 1
67     marker.scale.y = 1
68     marker.scale.z = 1
69     marker.pose.position.x = data.dx
70     marker.pose.position.y = data.dy
71
72     #make blue markers
73     marker.color.r = 0.0
74     marker.color.g = 0.0
75     marker.color.b = 1.0
76     marker.lifetime = rospy.Duration(0.1)
77     markerArray.markers.append(marker)
78     #markerArray.markers.append(ownship)
79 else:
80     return
81 def mDetectionCallback(data):
82     marker.type = marker.SPHERE
83     marker.header.stamp = time_now
84     marker.scale.x = 1
85     marker.scale.y = 1
86     marker.scale.z = 1
87     marker.pose.position.x = data.PosX
88     marker.pose.position.y = data.Posy
89
90     #make purple markers
91     marker.color.r = 1.0
92     marker.color.g = 0.0
93     marker.color.b = 1.0
94     marker.lifetime = rospy.Duration(0.1)
95     markerArray.markers.append(marker)
96     #markerArray.markers.append(ownship)

```

```

97 def leadVehCallback(data):
98     if (data.leadVehDist < 1000000):
99         marker.type = marker.CUBE
100         marker.header.stamp = time_now
101         marker.scale.x = 2.5
102         marker.scale.y = 1.5
103         marker.scale.z = 1.0
104         marker.pose.position.x = data.leadVehDist
105         marker.pose.position.y = data.leadVehLateralDist
106
107         #make green marker
108         marker.color.r = 0.0
109         marker.color.g = 1.0
110         marker.color.b = 0.0
111         marker.lifetime = rospy.Duration(0.3)
112         markerArray.markers.append(marker)
113     else:
114         return
115 def trackCallback(data):
116     if (data.dx < 1000000 and data.dx > 0):
117         marker.type = marker.SPHERE
118         marker.header.stamp = time_now
119         marker.scale.x = 1.5
120         marker.scale.y = 1.5
121         marker.scale.z = 1.5
122         marker.pose.position.x = data.dx
123         marker.pose.position.y = data.dy
124         #marker.id = data.trackNum #use this when every piece has a unique
125         #identifier
126         #make red markers
127         marker.color.r = 1.0
128         marker.color.g = 0.0
129         marker.color.b = 0.0

```



```

129     marker.lifetime = rospy.Duration(0.3)
130     text.scale.x = 1.5
131     text.scale.y = 1.5
132     text.scale.z = 1.5
133     text.pose.position.x = data.dx+4
134     text.pose.position.y = data.dy
135     text.color.r = 1.0
136     text.color.g = 0.0
137     text.color.b = 0.0
138     text.lifetime = rospy.Duration(0.3)
139     text.text = ("TrackID: %i \n x: %0.2f \n y: %0.2f" % (data.trackNum,
140         data.dx, data.dy))
141     markerArray.markers.append(marker)
142     markerArray.markers.append(text)
143
144 def viz():
145     global markerArray
146     global time_now
147
148     # #1 lead vehicle verbose topic
149     # rospy.Subscriber("/lead_vehVERBOSE", LeadVehicleVerbose,
150         leadVehCallback)
151
152     #32 radar detection topics
153     # rospy.Subscriber("/Radar1_Obj00_A", RadObjA, rDetectionCallback)
154     # rospy.Subscriber("/Radar1_Obj01_A", RadObjA, rDetectionCallback)
155     # rospy.Subscriber("/Radar1_Obj02_A", RadObjA, rDetectionCallback)
156     # rospy.Subscriber("/Radar1_Obj03_A", RadObjA, rDetectionCallback)
157     # rospy.Subscriber("/Radar1_Obj04_A", RadObjA, rDetectionCallback)
158     # rospy.Subscriber("/Radar1_Obj05_A", RadObjA, rDetectionCallback)
159     # rospy.Subscriber("/Radar1_Obj06_A", RadObjA, rDetectionCallback)
160     # rospy.Subscriber("/Radar1_Obj07_A", RadObjA, rDetectionCallback)
161     # rospy.Subscriber("/Radar1_Obj08_A", RadObjA, rDetectionCallback)
162     # rospy.Subscriber("/Radar1_Obj09_A", RadObjA, rDetectionCallback)
163     # rospy.Subscriber("/Radar1_Obj10_A", RadObjA, rDetectionCallback)

```

```

160 # rospy.Subscriber("/Radar1_Obj11_A", RadObjA, rDetectionCallback)
161 # rospy.Subscriber("/Radar1_Obj12_A", RadObjA, rDetectionCallback)
162 # rospy.Subscriber("/Radar1_Obj13_A", RadObjA, rDetectionCallback)
163 # rospy.Subscriber("/Radar1_Obj14_A", RadObjA, rDetectionCallback)
164 # rospy.Subscriber("/Radar1_Obj15_A", RadObjA, rDetectionCallback)
165 # rospy.Subscriber("/Radar1_Obj16_A", RadObjA, rDetectionCallback)
166 # rospy.Subscriber("/Radar1_Obj17_A", RadObjA, rDetectionCallback)
167 # rospy.Subscriber("/Radar1_Obj18_A", RadObjA, rDetectionCallback)
168 # rospy.Subscriber("/Radar1_Obj19_A", RadObjA, rDetectionCallback)
169 # rospy.Subscriber("/Radar1_Obj20_A", RadObjA, rDetectionCallback)
170 # rospy.Subscriber("/Radar1_Obj21_A", RadObjA, rDetectionCallback)
171 # rospy.Subscriber("/Radar1_Obj22_A", RadObjA, rDetectionCallback)
172 # rospy.Subscriber("/Radar1_Obj23_A", RadObjA, rDetectionCallback)
173 # rospy.Subscriber("/Radar1_Obj24_A", RadObjA, rDetectionCallback)
174 # rospy.Subscriber("/Radar1_Obj25_A", RadObjA, rDetectionCallback)
175 # rospy.Subscriber("/Radar1_Obj26_A", RadObjA, rDetectionCallback)
176 # rospy.Subscriber("/Radar1_Obj27_A", RadObjA, rDetectionCallback)
177 # rospy.Subscriber("/Radar1_Obj28_A", RadObjA, rDetectionCallback)
178 # rospy.Subscriber("/Radar1_Obj29_A", RadObjA, rDetectionCallback)
179 # rospy.Subscriber("/Radar1_Obj30_A", RadObjA, rDetectionCallback)
180 # rospy.Subscriber("/Radar1_Obj31_A", RadObjA, rDetectionCallback)
181
182 # #10 mobileye detection topics
183 # rospy.Subscriber("/ObstacleDataA1", ME_A2, mDetectionCallback)
184 # rospy.Subscriber("/ObstacleDataA2", ME_A2, mDetectionCallback)
185 # rospy.Subscriber("/ObstacleDataA3", ME_A2, mDetectionCallback)
186 # rospy.Subscriber("/ObstacleDataA4", ME_A2, mDetectionCallback)
187 # rospy.Subscriber("/ObstacleDataA5", ME_A2, mDetectionCallback)
188 # rospy.Subscriber("/ObstacleDataA6", ME_A2, mDetectionCallback)
189 # rospy.Subscriber("/ObstacleDataA7", ME_A2, mDetectionCallback)
190 # rospy.Subscriber("/ObstacleDataA8", ME_A2, mDetectionCallback)
191 # rospy.Subscriber("/ObstacleDataA9", ME_A2, mDetectionCallback)
192 # rospy.Subscriber("/ObstacleDataA10", ME_A2, mDetectionCallback)

```

```

193
194 # # #20 track topics
195 rospy.Subscriber("/Tracks1", Track, trackCallback)
196 rospy.Subscriber("/Tracks2", Track, trackCallback)
197 rospy.Subscriber("/Tracks3", Track, trackCallback)
198 rospy.Subscriber("/Tracks4", Track, trackCallback)
199 rospy.Subscriber("/Tracks5", Track, trackCallback)
200 rospy.Subscriber("/Tracks6", Track, trackCallback)
201 rospy.Subscriber("/Tracks7", Track, trackCallback)
202 rospy.Subscriber("/Tracks8", Track, trackCallback)
203 rospy.Subscriber("/Tracks9", Track, trackCallback)
204 rospy.Subscriber("/Tracks10", Track, trackCallback)
205 rospy.Subscriber("/Tracks11", Track, trackCallback)
206 rospy.Subscriber("/Tracks12", Track, trackCallback)
207 rospy.Subscriber("/Tracks13", Track, trackCallback)
208 rospy.Subscriber("/Tracks14", Track, trackCallback)
209 rospy.Subscriber("/Tracks15", Track, trackCallback)
210 rospy.Subscriber("/Tracks16", Track, trackCallback)
211 rospy.Subscriber("/Tracks17", Track, trackCallback)
212 rospy.Subscriber("/Tracks18", Track, trackCallback)
213 rospy.Subscriber("/Tracks19", Track, trackCallback)
214 rospy.Subscriber("/Tracks20", Track, trackCallback)
215
216 markerArray.markers.append(ownship)
217 while not rospy.is_shutdown():
218     i_d = 0
219     for m in markerArray.markers:
220         m.id = i_d
221         i_d += 1
222     publisher_t.publish(markerArray)
223     #print(len(markerArray.markers))
224     if (len(markerArray.markers) >= 100 ):
225         markerArray.markers = []

```

```
226     markerArray.markers.append(ownship)
227     time_now=rospy.Time.now()
228
229     rate.sleep()
230
231 if __name__ == '__main__':
232     try:
233         viz = viz()
234     except rospy.ROSInterruptException:
235         marker.action = Marker.DELETEALL
236         ownship.action = Marker.DELETEALL
237     pass
```

## REFERENCES

- [1] “Advanced vehicle technology competitions (avtc),” Retrieved March 03, 2023, from <https://avtcseries.org>.
- [2] F. Rosique, P. J. Navarro, C. Fernández, and A. Padilla, “A systematic review of perception system and simulators for autonomous vehicles research,” *Sensors*, vol. 19, p. 648, 2019, doi:10.3390/s19030648.
- [3] S. Campbell, “Sensor technology in autonomous vehicles: A review,” *29th Irish Signals and Systems Conference (ISSC)*, pp. 1–4, 2018, doi:10.3390/s21062140.
- [4] D. Yeong, G. Velasco-Hernandez, J. Barry, and J. Walsh, “Sensor and sensor fusion technology in autonomous vehicles: A review,” *Sensors*, 2021, doi:10.3390/s21062140.
- [5] R. Haber, “Visualization techniques for engineering mechanics,” *Computing Systems in Engineering*, vol. 1, pp. 37–50, 1 1990, doi:10.1016/0956-0521(90)90046-N.
- [6] G. R. Bertoline, E. Wiebe, C. Miller, and L. Nasman, *Engineering Graphics Communication*. Chicago: Irwin, 1995.
- [7] “2023 chevrolet blazer review, pricing, and specs,” 2023, Retrieved April 11, 2023, from [https://www.caranddriver.com/chevrolet/blazer/specs/2019/chevrolet\\_blazer\\_chevrolet-blazer\\_2019](https://www.caranddriver.com/chevrolet/blazer/specs/2019/chevrolet_blazer_chevrolet-blazer_2019).
- [8] “TANK AIoT developer kit: TANK-XM811AI-i5AD/2A-R10,” Retrieved March 3, 2023, from <https://www.ieiworld.com/en/product-ns/model.php?II=2>.
- [9] *Sensor gateway unit interface specification: MRR/LRR4-SGU*, Robert Bosch LLC.
- [10] “Quad cam vision for robocars comes to CES,” Retrieved March 3, 2023, from <https://www.eetimes.com/quad-cam-vision-for-robocars-comes-to-ces/>.
- [11] “Mitigation strategies for design exceptions - safety: Federal highway administration,” Retrieved April 11, 2023, from [https://safety.fhwa.dot.gov/geometric/pubs/mitigationstrategies/chapter3/3\\_lane\\_width.cfm](https://safety.fhwa.dot.gov/geometric/pubs/mitigationstrategies/chapter3/3_lane_width.cfm).
- [12] A. Dattalo, “ROS introduction,” Retrieved February 23, 2023 from <http://wiki.ros.org/ROS/Introduction>.
- [13] J. D’Onfro, “How a billionaire who wrote google’s original code created a robot revolution,” Retrieved February 23, 2023, from <https://www.businessinsider.com/a-look-back-at-willow-garage-2016-2>.

- [14] B. Soner and S. Coleri, “Visible light communication based vehicle localization for collision avoidance and platooning,” *IEEE Transactions on Vehicular Technology*, vol. 70, 3, doi:10.1109/TVT.2021.3061512.
- [15] J. Maderer, “Student team wins department of energy ecocar mobility challenge,” Retrieved December 1, 2023, from <https://coe.gatech.edu/news/2022/05/student-team-wins-department-energy-ecocar-mobility-challenge>.