CISTER
Research Centre in
Real-Time & Embedded
Computing Systems

BEng Thesis

An ImGui for a Cooperating Vehicle Simulation framework

Orientação Científica: Ricardo Severino

Francisco Cardoso

CISTER-TR-191217

An ImGui for a Cooperating Vehicle Simulation framework

Francisco Cardoso

CISTER Research Centre

Polytechnic Institute of Porto (ISEP P.Porto)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail:

https://www.cister-labs.pt

Abstract

Modern embedded systems, combined with the progressions of digital communicationtechnologies, have been empowering a new generation of systems,firmly connecting with the physical environment via sensing and actuating actions:Cyber Physical Systems (CPS). Cooperative Vehicular Platooning (CoVP)is one of these emerging applications among the new generation of safety-criticalCooperating CPS.CoVP is a methods for driving a group of vehicles together. It can potentiateseveral benefits, such as safety, fuel efficiency and the capacity of roads via anautomated highway system. COPADRIVe is a co-simulation tool that is currentlybeing developed and aims to create a fully functional platooning system.Although, one of the components that slows the development process is relatedto the difficulty of "keeping on track" such complex systems since there are a widenumber of variants that need to be constantly analyzed.Therefore, to strengthen and improve the work done on this simulation tool,this Thesis proposes to implement a Graphical User Interface (GUI) that allowsan easier interconnection between the developer and the simulation developmentsystem, which in this case is the Robot Operating System (ROS). The mainobjective of this work it to facilitate data analysis and processing of COPADRIVe,serving as front-end for the simulation analysis and control.

An ImGui for a Cooperating Vehicle Simulation framework

Francisco Xavier Pinto Cardoso

Licenciatura em Engenharia Eletrotécnica e de Computadores
Departamento de Engenharia Eletrotécnica

Instituto Superior de Engenharia do Porto
2019

Este relatório satisfaz, parcialmente, os requisitos que constam da Ficha de Unidade Curricular de Projeto/Estágio, do 3º ano, da Licenciatura em Engenharia Eletrotécnica e de Computadores

Candidato: Francisco Xavier Pinto Cardoso, Nº 1161253, 1161253@isep.ipp.pt
Orientação científica: Dr. Ricardo Augusto Rodrigues da Silva Severino (PhD), rarss@isep.ipp.pt

Licenciatura em Engenharia Eletrotécnica e de Computadores
Departamento de Engenharia Eletrotécnica
Instituto Superior de Engenharia do Porto
5 Setembro 2019

Acknowledgements

I would like to start by thanking my family and friends for the emotional support.

I also want to thank Dr. Ricardo Severino for his guidance and all the dedication and support. Moreover, i want to thank all my colleagues at Cister for the help and moral support.

Also, one last special thanks to my friend and fellow student Tiago Pinto for all the assistance and patience that he had with me over the last few months.

Resumo

Os sistemas embebidos modernos combinados com as evoluções tecnologias na área da comunicação digital, têm criado uma nova geração de sistemas, que se conectam fortemente ao ambiente físico através de sensores e atuadores: Cyber Physical Systems (CPS). O Cooperative Vehicular Platooning (CoVP) é uma das aplicações emergentes na nova geração de sistemas cooperativos de segurança CPS.

O CoVP é um método para conduzir um grupo de veículos juntos, que pode potencializar vários benefícios, como a segurança, a eficiência no consumo de combustivel e a capacidade das estradas através da automatização das vias de transporte. O COPADRIVe é uma ferramenta de simulação que se encontra atualmente em desenvolvimento e que tem como objetivo criar um sistema de platooning totalmente funcional.

Entretanto, um dos componentes que atrasa o processo de desenvolvimento está relacionado com a dificuldade de "manter o controle" de sistemas tão complexos, pois há um grande número de variaveis que precisam de ser constantemente analisadas.

Portanto, para fortalecer e aprimorar o trabalho realizado nessa ferramenta de simulação, nesta tese é proposto a implementação de uma Graphical User Interface (GUI) que permita facilitar a conexão entre o programador e o sistema de desenvolvimento da simulação, que neste caso é o Robot Operating System (ROS). O principal objetivo deste trabalho é facilitar a análise e o processamento de dados do COPADRIVe, servindo de front-end para a analisa e controlo da simulação.

Palavras-Chave: Cyber Physical Systems, Cooperative Vehicular Platooning, Wireless Sensor Network, COPADRIve, Graphic User Interface, Robot Operating System

Abstract

.

Modern embedded systems, combined with the progressions of digital communication technologies, have been empowering a new generation of systems, firmly connecting with the physical environment via sensing and actuating actions: Cyber Physical Systems (CPS). Cooperative Vehicular Platooning (CoVP) is one of these emerging applications among the new generation of safety-critical Cooperating CPS.

CoVP is a methods for driving a group of vehicles together. It can potentiate several benefits, such as safety, fuel efficiency and the capacity of roads via an automated highway system. COPADRIVe is a co-simulation tool that is currently being developed and aims to create a fully functional platooning system.

Although, one of the components that slows the development process is related to the difficulty of "keeping on track" such complex systems since there are a wide number of variants that need to be constantly analyzed.

Therefore, to strengthen and improve the work done on this simulation tool, this Thesis proposes to implement a Graphical User Interface (GUI) that allows an easier interconnection between the developer and the simulation development system, which in this case is the Robot Operating System (ROS). The main objective of this work it to facilitate data analysis and processing of COPADRIVe, serving as front-end for the simulation analysis and control.

Keywords: Cyber Physical Systems, Cooperative Vehicular Platooning, Wireless Sensor Network, COPADRIve, Graphic User Interface, Robot Operating System

Contents

List of Figures

List of Acronyms

Acronym	Description
ADAS	Advanced Driver-Assistance Systems
API	Application Programming Interface
CoVP	Cooperative Vehicular Platooning
OpenGL	Open Graphics Library
CPS	Cyber Physical Systems
DSPC	Data-Centric Publish-Subscribe
DDS	Data Distribution Service
GDS	Dlobal Data Space
GPS	Global Positioning System
GPU	Graphics Processing UniT
GUI	Graphical User Interface
IMGUI	Immediate Mode Graphical User Interface
IoT	Internet of Things
NUI	Natural User Interfaces
OMG	Object Management Group
OS	Operating System
P2P	Peer-to-Peer
ROS	Robot Operating System
RTPS	Real-Time Publish/Subscribe Protoco
SDL	Simple DirectMedia Layer
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UI	User Interfaces
V2I	Vehicle-to-Infrastructure
V2V	Vehicle-to-Vehicle

Chapter 1

Introduction

1.1 Overview

The last decades are considered to be the years when there has been the biggest technological breakthrough. It's safe to say that we now live in a society where the use of new technologies and tools are a crucial part of our everyday routine. Nowadays, companies such as Tesla are working in a continuous way to reduce the human factor in daily critical applications, approaching us to a reality that was far distant many years ago. One of the most invested areas of these companies is autonomous vehicles.

Although, creating new technological advances is not an easy task. Despite the amount of information and tools available, there is still room for improvement regarding the technological development from a wide number of industrial areas, namely the vehicle sector.

In autonomous vehicles, there are three important types of systems: Advanced Driver-Assistance Systems (ADAS), Vehicle-to-Vehicle (V2V) and Vehicle-to-Infrastructure (V2I). These systems carry the potential to radically improve transportation. From reduced collisions to increased energy efficiency.

ADAS are intelligent systems that reside inside the vehicle and assist the driving in a variety of ways[13]. Moreover, V2V communication settles in a wireless network where vehicles send messages to each other with information about what they are doing [14]. Finally, V2I address the wireless exchange of data between vehicles and road infrastructures [15].

This new communication mechanism is due to a new technological concept: the Internet of Things (IoT). IoT allows the expansion of the interaction between the physical environments with digital systems, which is referred to as Cyber Physical Systems (CPS), via sensors and actuators. The emerging CPS can be

applied throughout a big range of areas. Such as Smart Houses, Medical and health-care, Industrial automation and, as mentioned before, vehicle automation.

Inside this new concept, platooning stands out for its capability to drive a group of vehicles together, in order to provide an automated highway system. Many research project, such as COPADRIVe, has been done to find the best approach to implement this kind of systems with safety insurances. COPADRIVe is a realistic simulation framework for cooperative autonomous driving application, each has a platoon model scenario implemented under the Robot Operating System (ROS) and Gazebo [11].

In the world of autonomous vehicles, there is no single provision of a graphical interface for direct control over the various components of the system. Even Robot Operating System (ROS), only provides standard operating system services such as hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes and package management. Although, ROS has an interface that allows graphical representations, called Rviz, it fails to directly control the various components of a simulation tools, such as the frameworks developed outside of ROS environment.

However, ROS supply packages which allow linking the system with a external Graphical User Interface (GUI) such as Qt, GTK+, wxWidgets and Dear ImGui.

This Thesis propose to develop a GUI interface and develop the software modules which link the GUI to the various software components through the ROS framework. This allows the user to interact with COPADRIVe simulation tool thought graphics and visual indicators and control certain simulation parameters.

1.2 Research Context

This Thesis is being accomplished at the CISTER - Research Centre in Real-Time and Embedded Computing System, synced with the COPADRIve simulation tool which is a realistic simulation framework for cooperative autonomous driving applications with a Gazebo platoon model scenario. This Thesis context comes up with the quality improvement of the cooperative autonomous driving applications.

1.3 Research Objectives

The objectives of this Thesis emphasize the development of a solution that allows the developer interaction with the cooperative autonomous car simulations in the ROS1 and ROS2 development environments through a graphical interface. This goal is intended to be achieved by creating a GUI interface that allows direct connection thought graphics and visual indicators, and also can provide control over certain simulation parameters.

1.4 Research Contributions

The contribution of this Thesis is to provide a first draft tool that allows the developer to ease data and system analysis of connected vehicular application. Usually similar systems are based on complex structures that are difficult to adapt to unpredictable situations. Thus, the application developed in this Thesis must aim to control and configure these applications through the graphical application developed in this Thesis.

1.5 Structure of this thesis

This Thesis is divided into five chapters, including the current one, which is the Introduction chapter. The second chapter introduces the GUI frameworks detailing its main functions and described some options for GUI frameworks. In the third chapter, the technologies required to implementation are present, including ROS1, ROS2, Gazebo and the COPADRIVe. Then, chapter four focus on the full implementation of Dear ImGui over the ROS enabled system (COPADRIVe) including the code development, the compilation in the ROS environment, concluding with experimental visual validation in other to legitimize the implementation. Finally, chapter five, overviews the results and present the future plans for the GUI implementation.

Chapter 2

Graphical User Interface

This chapter starts by presenting and describing the Graphical User Interface (GUI). Then, it presents an overview of some options of GUI Frameworks such as Rviz, Qt, GTK+, wxWidgets and Dear Imgui. Concluding with a brief review over the selected GUI framework, Dear Imgui and introduce the external tools required to render a Dear ImGui program.

2.1 GUI

2.1.1 Overview

Graphical User Interface (GUI) announces a new era for users to interact with machines and computer systems. Gone are the days where operating computers needed a huge knowledge of code and the inner workings of a system to interact with the command-line user interface such as MS-DOS, waiting for a response from the computer. The term GUI was created in the 1970s to distinguish graphical interfaces from text-based ones, such as a command-line interface.

This innovation offered a new form of interaction between the user and the systems, in a way they didn't need to learn code, killing a step learning curve [16].

PARC was the first commercially available GUI, developed by Xerox. It was used by the Xerox 8010, which was released in 1981. During a tour in Xerox, Steve Jobs saw this interface and was inspired by it. Therefore, the first Apple operating system was very similar to the PARC interface.

The first Apple's GUI-based OS was included in the Macintosh, which was released in 1984 becoming the most popular commercial computer since having

a graphical representation of an OS makes it more accessible and easy to interact. Apple was followed by Microsoft that released its first GUI-based OS with Windows 1.0, in 1985 [17].

GUI is a form of user interface that enables a person to communicate with a computer through the use of windows, icons, menus, text and other visual indicator representations to display information and related user controls as opening, deleting and moving files.

During a long time, GUIs were controlled exclusively by a mouse and a keyboard, because these types of inputs are sufficient for desktops computers, but do not work for mobile devices. Mobile devices are projected to use a touchscreen interface, and now some mobile devices can be controlled by spoken commands as well as motion detection. These types of GUIs are called Natural User Interfaces (NUI)[18].

As there are many types of digital devices available, GUIs need to be designed for the appropriate type of input.

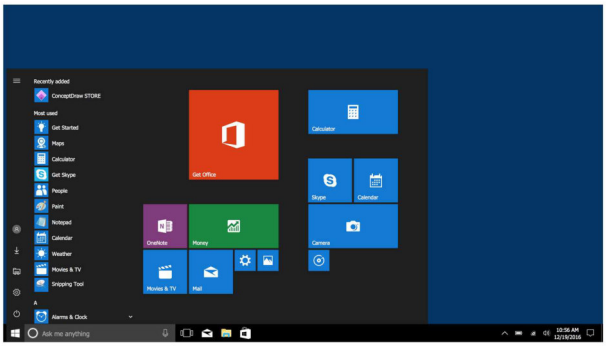


Figure 2.1: Windows 10 GUI interface [1]

2.1.1.1 Qt

Qt [19] is a free and open-source widget toolkit that allows creating GUI and cross-platform applications that runs on all major desktop platforms and most mobile or embedded platforms. Qt is made with the C++ programming language but can be used in several other languages such as python, PHP, Ruby, and Java. It also supports many compilers as well, as Visual Studio suite and GCC C++ Compiler.

It has many non-GUI-related classes, such as data/time, containers, networking, and OpenGL functionality. The GUI projects created by Qt do not use the system provided widgets but emulates it with themes. Each means that Qt draws its own widgets on each platform. Qt has a full-featured embeddable GUI (Qt for embedded Linux) based on GNU/Linux with framebuffer [20]. This means

that having Linux with /dev/fb is possible running examples without additional work. Also, it's possible to develop non-GUI programs, such as command-line tools and consoles for servers.

Figure 2.2: Qt application example [2]

2.1.1.2 GTK +

GTK+ [21], original knows as Gimp toolkit, is a free and open-source, cross-platform toolkit for creating graphical user interfaces. Its primary development and focus are for the Unix platform and it is the primary library used to construct user interfaces in GNOME.

GTK+ offers a complete set of widgets and is appropriate for diverse projects from small one-off tools to complete application suites [22]. Unlike wxWidgets and Qt, GTK+ supports C but is also design to support multi-languages, such as C/C++, Perl, Python, Ruby, and Java. It looks and behaves exactly the same on all platforms unless themes are used. On windows, it has the ability to get the native look with the Wimp theme [23], doesn't using system provided widgets, but emulates them.

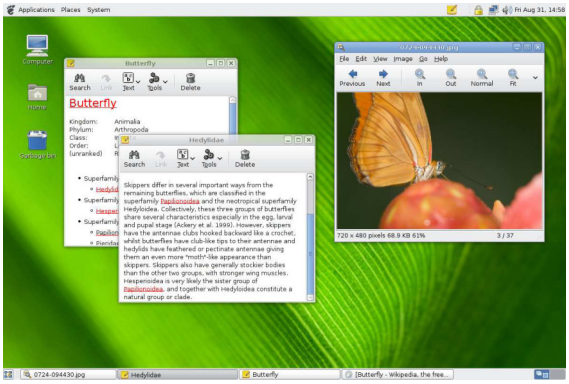


Figure 2.3: GTK+ application example [3]

2.1.1.3 wxWidgets

wxWidgets [24] is open-source and mature widget toolkit and tools library for creating GUIs for cross-platform application, it is a C++ library that allows developers to create applications for Windows, OS X, Linux, and UNIX 32-bits and 64-bits architectures, as well as several mobile platforms. It supports multi-languages such as python, Perl, Ruby, Java, PHP, and some others. It is one of the most complete GUI toolkits with a lot of documentation

This multi-platform toolkit uses the native platform SDK and system provided widgets instead of emulating the GUI, given the applications a native look and feel. This means that a program compiles in Windows will look like a Windows program, and when compiling on a Linux machine, it will get the look of a Linux program. Focusing on native looks also mean wxWidgets may not be a good option for an application that wants a customized look, instead of the system's theme.

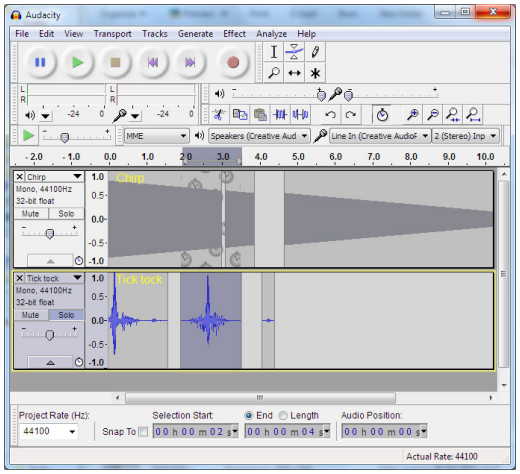


Figure 2.4: wxWidgets application example [4]

2.1.1.4 Rviz

Rviz [25] is a 3D visualizer for the ROS framework, which allows to display sensor data and state information from ROS. Rviz displays 3D sensor data from stereo cameras, lasers, Kinects, and other 3D devices in the form of point clouds or depth images. 2D sensor data from webcams, RGB cameras, and 2D laser rangefinders can be viewed in rviz as image data.

If an actual robot is communicating with a workstation that is running rviz, rviz will display the robot's current configuration on the virtual robot model. ROS topics will be displayed as live representations based on the sensor data published by any cameras, infrared sensors, and laser scanners that are part of

the robot's system. This can be useful to develop and debug robot systems and controllers. Rviz provides a configurable Graphical User Interface (GUI) to allow the user to display only information that is pertinent to the present task.

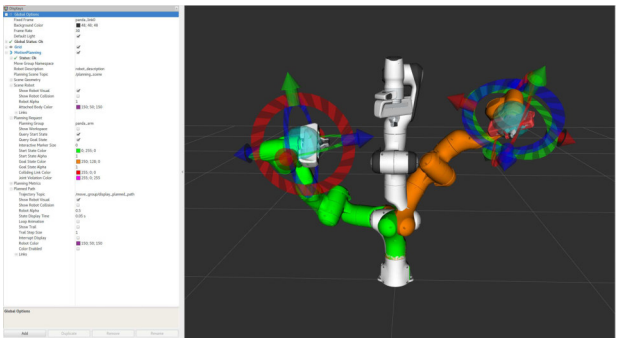


Figure 2.5: Rviz application example [5]

2.1.1.5 Dear Imgui

Dear Imgui [7] is a graphical user interface library accent in C++ language programming. It generates optimized vertex buffers that make possible render anytime in any application. Dear ImGui is designed to enable create content creation tools and visualization/debug tools. Therefore, this framework is fast, portable, render agnostic and does not need external dependencies.

Dear Imgui is appropriate for integration in game engines, real-time 3D, fullscreen and embedded applications, or any applications on consoles platforms where OS features are non-standard. It is an immediate mode GUI (IMGUI) library which allows to adapt GUI for all kind of tools.

An immediate mode GUI (IMGUI) is different from the traditional GUI. The classic way of developing a GUI is through a retained-mode. Unlike IMGUI, which involves creating and drawing widgets in each frame, instead of creating button objects and adding a callback to it [26]. Figure. 2.6 shows the difference between these two different methods.

Dear ImGui is very dynamic (periodic upgrades), allows fast prototyping and is really useful for real-time simulation since IMGUI is very responsive to real-time environment based systems. For that reason, in this Thesis, since the main objective is to provide graphical environment to real-time simulations, Dear ImGui is the obvious GUI framework choice to be implemented on ROS based systems. Moreover, it also has the advantage of being a framework independent of the ROS system, unlike Rviz, allowing the implementation with the various components of the COPADRIVe simulation tool, such as ROS-based robotic simulation (Gazebo) and network simulator (OMNeT++).

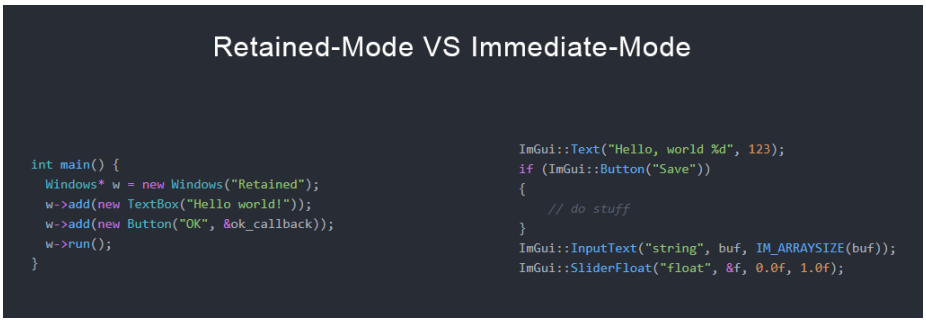


Figure 2.6: Comparison between Retained-mode and Immediate-Mode code

Figure 2.7 shows an example of a game that uses ImGui as graphical interface and debugger.

Figure 2.7: Dear ImGui application example [6]

Programming User Interfaces (UIs) has a reputation of being difficult, perhaps due to the fact that UI toolkits tend to be large and complex software systems. IMGUI represents a new paradigm in GUIs where the user interface is simpler to create and simpler to be implemented.

Most of the problems associated with the design and use of the traditional GUIs are a direct result of such systems to retain state. From the user's point of view, the UI must look like a collection of objects, with encapsulating a state that needs to be frequently synchronized with the application [26].

This synchronization goes both ways. To become visible to the user, states move from the application to the UI and in order to change the state of the application, when a user interacts with the interface, states move from the user interface back to the application.

IMGUI tries to minimize superfluous state duplication, state synchronization and state retention from the user's point of view by requiring the application to

explicitly pass all state required for visualization and interaction with any given widget in real-time.

Widgets are no longer objects. Its take the form of procedural method calls, and the UI goes from retain-state of objects to be a real-time sequence of method calls [27]. The application processes logic and draws its displays at real-time rates.

It is less code and less bugs than the traditional GUI and allows to create dynamic user interfaces [7]. Dear ImGui provides vertex buffers and a small list of draw calls batches that enable an easy render in the application. The number of draw calls and state changes required to render them is fairly small. It never touches the Graphics Processing Unit (GPU) directly, so it is possible to call its functions anywhere in the application code, in the middle of a running algorithm or in the middle of the render process.

Dear ImGui, is not just for tweaking values. It is possible to trace a running algorithm by just emitting text commands, using it along with reflection data to browser the live dataset. Additionally, it can also expose the internals of a subsystem in the engine, create a logger, an inspection tool, a profile, a debugger and an entire game making editor/framework.

However, since Dear ImGui is only a library and renders agnostic, we need to provide external tools to render, which will interact with the GPU. As an example, OpenGL needs a toolkit to guarantee the API related to input, window and events handling libraries, which are included in SDL.

Figure 2.8: Dear ImGui Demo Windows [7]

2.1.1.6 OpenGL and SDL

Open Graphics Library (OpenGL) is a cross-platform-language Application Programming Interface (API) for render 2D and 3D vector graphics. This API is mostly used to interact with a Graphics Processing Unit (GPU).

The specification does not say anything on the subject of obtaining and managing an OpenGL context, leaving this as a detail of the underlying windows system. OpenGL is only concerned with rendering and does not provide APIs related to input, audio or windowing.

To create, manage OpenGL windows and to manage input, there are toolkits design for that, such as SDL, which complements OpenGL by setting up the graphical output and providing mouse and keyboard input. It is a multimedia libraries with a C API that beyond manage OpenGL windows and input. Also can manage sound, file access, event handling and time. In this Thesis, SDL and OpenGL will be important to provide a visual response in software development to the Dear ImGui framework.

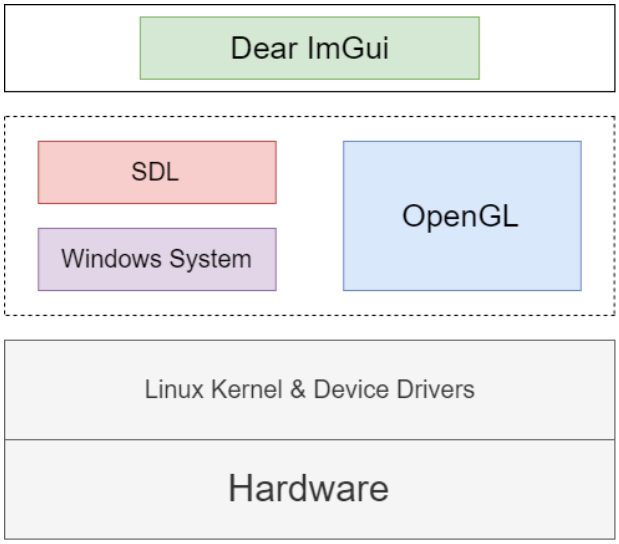


Figure 2.9: SDL and OpenGL in Dear ImGui Framework

Chapter 3

Simulation Tools

This chapter presents and describes the technologies used in this Thesis. Starting by introducing and describing the ROS software, including the ROS1 and its evolution ROS2. Then, the ROS simulation tool Gazebo is presented. Concluding with a brief review over COPADRIVe simulation tool.

3.1 ROS

3.1.1 Overview

Robotic Operating System (ROS) is an open-source Linux based framework for creating high-performance robot applications. ROS is not a traditional operating system (OS) in the sense of process management and scheduling, but it provides services similar to a normal OS such as including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between process and package management [28]. Also, it provides tools and libraries to build, write and run code across multiple computers.

There are available a big variety of other robots frameworks, such as PLAYER, CARMEN, Orca, and MOOS. Although the motivation of ROS is to support software reuse and to build robotic systems with software components in robotics research and development.

The ROS runtime "graph" is a Peer-to-Peer (P2P) network that executes programs called nodes communicate with each other at runtime. P2P systems are very used when it comes to sharing big quantities of data.

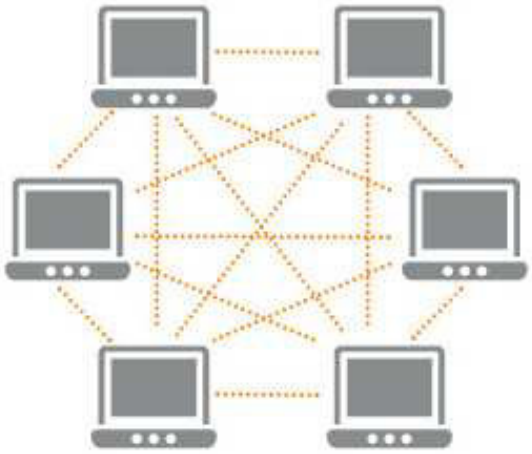


Figure 3.1: P2P Communication [8]

3.1.2 ROS Communication model

The principal concepts of ROS implementation are nodes, messages, topics, subscribers, and publisher. In ROS development, o robot system is designed using a set of components called nodes and a communication channel called topic. It's possible to choose from several distributed packages and build software by connecting them. The communication model of ROS is based on Publish-Subscribe messaging [29].

Publish-Subscriber messaging is an asynchronous messaging in which ROS nodes do not communicate directly with each other but through a topic. The advantage is a dynamic network configuration that allows adding new ROS nodes to the system easily. There are two rules in ROS nodes: publisher and subscriber.

A publisher node, send a message by publishing it to a given topic. The topic is the communication path that holds a sequence of messages. A subscriber is a node that needs a certain kind of data and subscribes to the appropriate topic to receive the message, as shown in Figure 3.2. It is possible to exist a lot of concurrent for publishers and subscribers for just one topic. As ROS nodes do not have information on their communication partners, ROS is able to be configured dynamically, which is very suitable for rapid system prototyping.

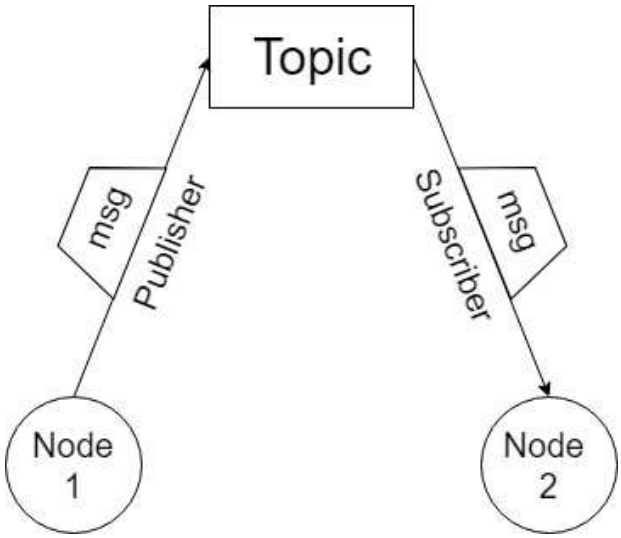


Figure 3.2: ROS Communication concept

3.2 ROS 2

All the developments that have been made in the area of improving cooperative autonomous driving accentuate in the ROS1 development environment. But as this system does not allow real-time communication, in the future all the work developed must be updated to ROS2. Therefore, with this in mind, this Thesis aims to implement the graphical interface developed over this innovative technology.

3.2.1 Overview

ROS does not satisfy real-time run requirements and only runs on a few OSs, besides that ROS cannot guarantee fault-tolerance, deadlines, or process synchronization [9]. Thus, ROS is not suitable for real-time embedded systems, and to satisfy that ROS have a significant upgrade to ROS2.

ROS2 have new functions such as real-time systems, small embedded platforms, non-ideal networks and cross-platform (e.g. Linux, Windows, Mac, Real-time OS, no OS) and to satisfy this requirements ROS (ROS1) has been reconstructed to improve user-interface APIs and incorporate new technology such as Data Distribution Service (DDS), Zeroconf, Protocol Buffers, ZeroMQ, Redis and WebSockets. DSS is the one that replaces the ROS1 transport system, and in this Thesis will focus only on this one.

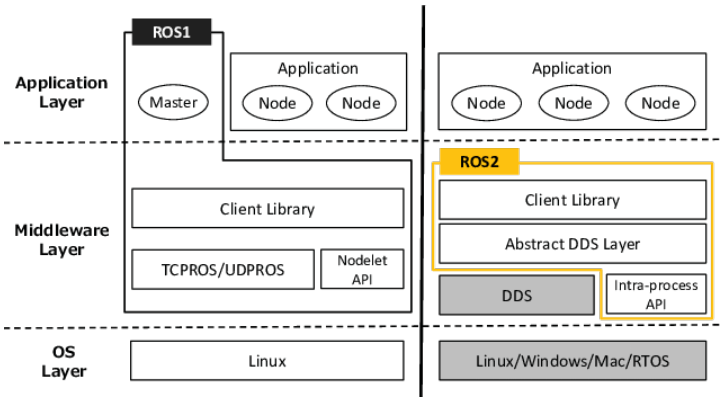


Figure 3.3: ROS1 and ROS2 Architectures [9]

3.2.2 DDS

DDS is an industry-standard real-time communication system and end-to-end middleware, that can provide reliable publish/subscribe transport similar to that of ROS1. DDS has been applied in mission-critical environments (e.g. trains, ships, financial systems, medical service). DDS is appropriate to real-time embedded systems because of its various transport configurations such as deadline, reliability and durability and scalability [30].

DSS collects the requirements of distributed systems for safety, resilience, scalability, fault-tolerance, and security. Also can provide solutions for real-time environments and small/embedded system by reducing library sizes and memory footprints.

The DDS specification is defined for a publish/subscriber system by the Object Management Group (OMG), each manages the definitions and standardized APIs. The core of DDS is a Data-Centric Publish-Subscribe (DCPS) model designed to arrange efficient data transport between processes even in distributed different applications. The DCPS model creates a Global Data Space (GDS) that can be accessed by any independent application, to efficient data distribution.

In DDS, every process that publishes or subscribe to data is called participants, witch in ROS correspond to a node. Participants can read and write from/to the GDS using a typed interface.

As well as ROS1, a Publisher is an application that sends data to one or more Topics. A DataWriter is an object used by the Participant to publish data through a Publisher, and a DataReader is an object used by Participants to receive and access data that must correspond to the DataWritter data.

After this transaction, a DataWriter and a DataReader connect with each other, using the Real-Time Publish/Subscribe Protocol (RTPS), as shown in Figure 3.4. The RTPS protocol and the DDS protocol allow DDS implementations

from a different vendor to operate simultaneously by abstracting and optimizing transport, like TCP, UDP, and IP.

The transport of data between a DataWriter and a DataReader is executed in the RTPS protocol according to the QoS Policy, which represents their data transport behavior and configured the deadlines period, depth of history and communication reliability.

This introduction of DCPS into the ROS model allows the possibility of managing the node communication without the necessity of a central server, with allowing the Real-Time Communication between the publisher and the subscriber.

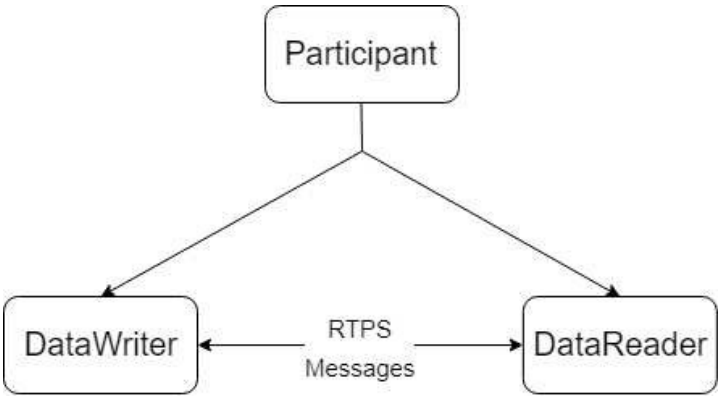


Figure 3.4: DDS Communication

3.3 Gazebo

Gazebo [31] development start in 2002 at the University of Southern California and it is one of the most used robot simulations. The gazebo is a 3D ROS-based robotic simulation with dynamics and kinematic physics that offers the ability to accurately and efficiently simulate robots in indoor and outdoor environments [32], with supports for multiple robots.

Gazebo used ROS features, such as the message interface, which is the same as the ROS system, making the ROS nodes compatible with the simulation, offering simulation synchronization, and communication between different software present on simulation scenarios.

The most relevant features present in Gazebo are Dynamics Simulation each permit access to multiple high-performance physics engines, Advanced 3D Graphics, Sensors and Noise that provide generation of sensor data optionally with noise, Plugins for robot/environment control, Robot Models already developed, TCP/IP transport for simulation running on remote servers, cloud simulation on Amazon AWS and GzWeb to interact with the simulation by a browser and command-line tools.

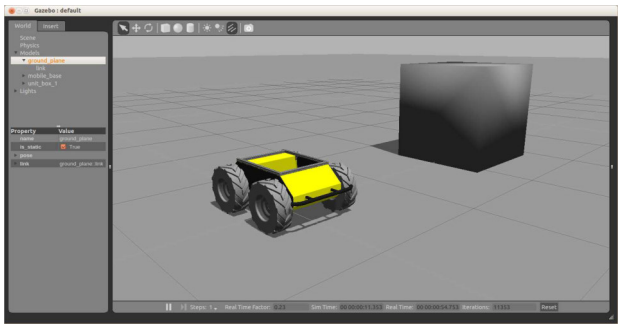


Figure 3.5: Gazebo simulation example [10]

3.4 COPADRIVe

COPADRIVe is a realistic simulation framework for cooperative autonomous driving applications. It is a powerful framework to test and validate cooperative autonomous driving applications, that integrates a well-known ROS-based robotic simulation (Gazebo) with a network simulator (OMNeT++), by extending Artery [33].

In the one hand, COPADRIVe utilize upon Gazebo's robotic simulation most prominent features, such as its support for multiple physics engines, and its rich library of components and vehicles in integration with ROS, which enables to build realistic vehicle control scenarios.

On the other hand, OMNet++ supports the underlying network simulation relying on an ITS-G5 communications stack which is, currently, the standard for C-ITS applications in Europe. This integration provides the support for an accurate analysis of the communications impact upon the cooperative application, and on the other hand, the tools to carry out a thorough evaluation of the network performance using the OMNet++/INET framework. In Figure 3.6 is shown the COPADRIVe Architecture.

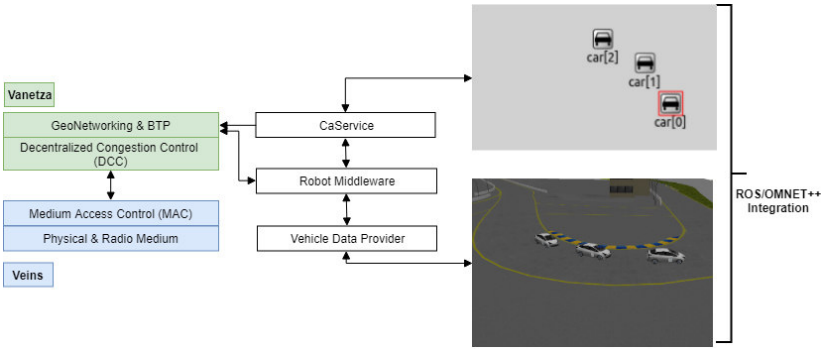


Figure 3.6: COPADRIVe framework Architecture [11]

Chapter 4

Integration between Dear ImGui and COPADRIVe

This chapter presents the integration between Dear ImGui and the COPADRIVe simulation tool. Starting with the development of the GUI in Dear Imgui Framework. Following this all the precedents needed to implement the graphical interface in the ROS environment will be presented. Finishing with the presentation of the implementation of GUI developed into Gazebo Platooning scenario.

4.1 Develop of GUI in Dear ImGui

Dear ImGui should be able to write bindings for pretty much any platform and any 3D graphics APIs. It is highly portable and only needs a few requirements to run and render the graphical component. These requirements are mouse and keyboard input, uploading the font atlas texture into graphics memory and providing a render function to render indexed textured context.

Dear ImGui is self-contained within a few files that can easily be copied and compiled into the application/engine:

- imgui.cpp
- imgui.h
- imgui_demo.cpp
- imgui_draw.cpp
- imgui_widgets.cpp

- imgui_internal.h
- imconfig.h
- imstb_rectpack.h
- imstb_textedit.h
- imstb_truetype.h

4.1.1 Plugging Dear ImGui into SDL2 and OpenGL

Dear ImGui, is renderer agnostic in the way that to create a working program it is necessary to provide the tools to render the data, combining one platform (e.g. GLFW, SDL2, and GLUT) with one renderer (e.g. OpenGL, DirectX, and Vulkan).

In this case, the platform binding that is in charge for mouse and keyboard input, cursor shape, timing, and windowing, is the SDL2 [34] and the renderer binding, in charge of creating the main font texture and rendering ImGui draw data, is the OpenGL [35].

Modern OpenGL does not have a standard header file and for that reason, requires individual function pointers to be loaded manually. In Figure 4.1 it is possible to see how to initialize GL3W [36], which is a helper library used for this purpose. Alternatives are GLEW and Glad.

```
#if defined(IMGUI_IMPL_OPENGL_LOADER_GL3W)
#include <GL/gl3w.h>    // Initialize with gl3wInit()
#else
#include IMGUI_IMPL_OPENGL_LOADER_CUSTOM
#endif
```

Figure 4.1: Initialization of GLW3 library

As mention above, Dear ImGui is independent of the rendering system and platform, and it is necessary to introduce some binding for the rendering system. Fortunately, they are many pre-made bindings in Dear ImGui's repository. In order to use SDL2 and OpenGL theses are the ones used:

- imgui_impl_opengl3.cpp
- imgui_impl_opengl3.h
- imgui_impl_sdl.cpp
- imgui_impl_sdl.h

The code required to integrate the platform and the render is in the source code, main.cpp. First, it is necessary to initialize SDL and OpenGL loader. Then initialize the windows for rendering and initialize a Dear ImGui context, the helper platform, and renderer bindings, like is shown below, in Figures 4.2, 4.3 and 4.4.

```
if (SDL_Init(SDL_INIT_VIDEO | SDL_INIT_TIMER | SDL_INIT_GAMECONTROLLER) != 0)
{
    printf("Error: %s\n", SDL_GetError());
    return -1;
}
```

Figure 4.2: Initialize SDL2

```
#if defined(IMGUI_IMPL_OPENGL_LOADER_GL3W)
    bool err = gl3wInit() != 0;
#else
    bool err = false;
#endif
    if (err)
    {
        fprintf(stderr, "Failed to initialize OpenGL loader!\n");
        return 1;
    }
```

Figure 4.3: Initialize OpenGL loader

```
// Setup Dear ImGui context
IMGUI_CHECKVERSION();
ImGui::CreateContext();
ImGuiIO& io = ImGui::GetIO(); (void)io;

// Setup Dear ImGui style
ImGui::StyleColorsDark();

// Setup Platform/Renderer bindings
ImGui_ImplSDL2_InitForOpenGL(window, gl_context);
ImGui_ImplOpenGL3_Init(glsl_version);
```

Figure 4.4: Setup Dear ImGui context and Platform/Renderer bindings

After that, it is the main application loop, where it is possible to see the difference with the traditional GUI modes. Instead of a retained mode with callbacks, the Dear ImGui creates and draws widgets in every frame, as shown in Figure 4.5, with the rendering process after the initialization of the frame. Additionally, Figure 4.6 presents the cleanup in the end of the loop.

```
bool done = false;
while (!done)
{

    SDL_Event event;
    while (SDL_PollEvent(&event))
    {
        ImGui_ImplSDL2_ProcessEvent(&event);
        if (event.type == SDL_QUIT)
            done = true;
        if (event.type == SDL_WINDOWEVENT && event.window.event == SDL_WINDOWEVENT_CLOSE
          && event.window.windowID == SDL_GetWindowID(window))
            done = true;
    }

    // Start the Dear ImGui frame
    ImGui_ImplOpenGL3_NewFrame();
    ImGui_ImplSDL2_NewFrame(window);
    ImGui::NewFrame();

    // Rendering
    ImGui::Render();
    glViewport(0, 0, (int)io.DisplaySize.x, (int)io.DisplaySize.y);
    glClearColor(clear_color.x, clear_color.y, clear_color.z, clear_color.w);
    glClear(GL_COLOR_BUFFER_BIT);
    ImGui_ImplOpenGL3_RenderDrawData(ImGui::GetDrawData());
    SDL_GL_SwapWindow(window);
}
```

Figure 4.5: Main application loop

```
ImGui_ImplOpenGL3_Shutdown();
ImGui_ImplSDL2_Shutdown();
ImGui::DestroyContext();

SDL_GL_DeleteContext(gl_context);
SDL_DestroyWindow(window);
SDL_Quit();
```

Figure 4.6: Cleanup after end of the loop

4.1.2 Compile Dear ImGui

In order to make an available executable project it is necessary to compile and link all the files that integrate the Dear ImGui framework. The files and their localization, incorporated in this first GUI example have the following filesystem, Figure 4.7.

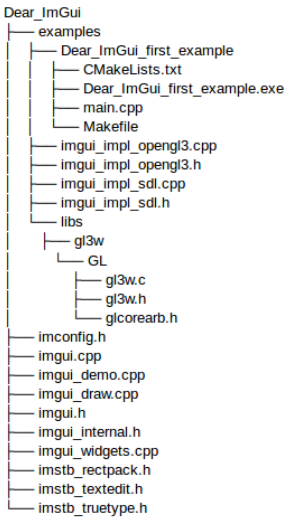


Figure 4.7: Filesystem Dear ImGui first example

To build an executable, a Makefile was created, which directs on how to compile and link the program. The Makefile contains a set of directories that are used by the make build automation tool to generate a target. That process is visible in Figure 4.8.

```
EXE = Dear_ImGui
SOURCES = main.cpp
SOURCES += ../imgui_impl_sdl.cpp ../imgui_impl_opengl3.cpp
SOURCES += ../../imgui.cpp ../../imgui_demo.cpp ../../imgui_draw.cpp ../../imgui_widgets.cpp
OBJS = $(addsuffix .o, $(basename $(notdir $(SOURCES))))
UNAME_S := $(shell uname -s)

CXXFLAGS = -I../ -I../../
CXXFLAGS += -g -Wall -Wformat
```

Figure 4.8: Makefile set directories of sources and flags

In this specific project, is crucial to include in the Makefile specification the project with the SDL2 platform, as shown in Figure 4.9 and the renderer OpenGL, described in Figure 4.10.

After a successful compilation and linking files, the executable program has created and ready to be used. Figure 4.11 shows the final result.

```
ifeq ($(UNAME_S), Linux) #LINUX
  ECHO_MESSAGE = "Linux"
  LIBS += -lGL -ldl `sdl2-config --libs`

  CXXFLAGS += -I../libs/gl3w `sdl2-config --cflags`
  CFLAGS = $(CXXFLAGS)
endif
```

Figure 4.9: Compile with SDL2 library

```
SOURCES += ../libs/gl3w/GL/gl3w.c
CXXFLAGS += -I../libs/gl3w
```

Figure 4.10: Compile OpenGL using GL3W library

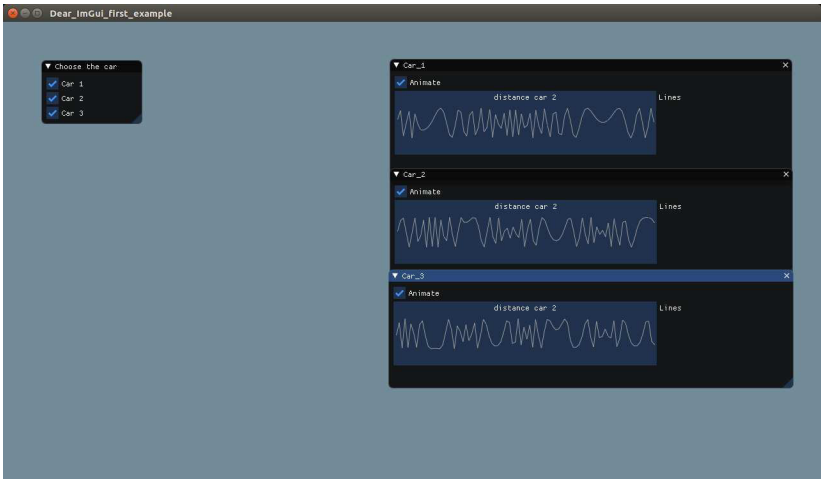


Figure 4.11: Dear ImGui first GUI example

4.2 Implement of Dear ImGui into ROS

In order to implement the GUI developed before into the ROS environment, it was necessary to create a bridge between the Dear ImGui framework and the ROS system, which allows the communication between Dear ImGui and another program through ROS Communication tools. For this purpose it was required a ROS workspace, in order to build a package with Dear ImGui context.

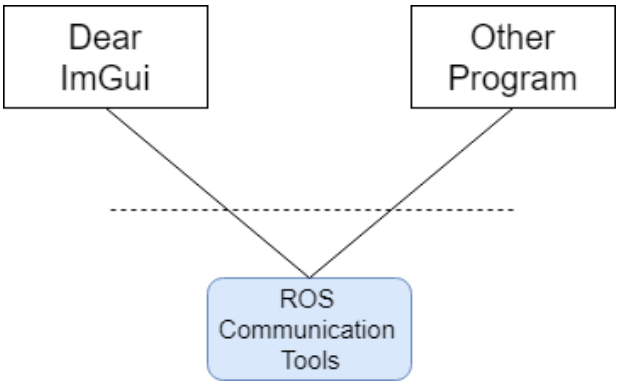


Figure 4.12: ROS Communication tools

4.2.1 Setup the ROS workspace

The next step is to create a catkin workspace. A catkin workspace is a directory in which it is possible to create and modify existing catkin packages [12]. Catkin packages can be built as a standalone project like the normal CMake projects can be built, but catkin also provides the concept of workspace, where it is feasible to build multiple, independent packages together all at once [35]. It structures simplifies the building and installation process for ROS packages.

A catkin workspace can contain three or more different subdirectories, such as build, devel, source, and install, which each one serves a different purpose in the software development process. Figure 4.13 represents the structure of catkin workspace.

```
workspace_folder/        -- WORKSPACE
  src/                   -- SOURCE SPACE
    CMakeLists.txt       -- The 'toplevel' CMake file
    package_1/
      CMakeLists.txt
      package.xml
      ...
    package_n/
      CATKIN_IGNORE      -- Optional empty file to exclude package_n from being processed
      CMakeLists.txt
      package.xml
      ...
  build/                 -- BUILD SPACE
    CATKIN_IGNORE        -- Keeps catkin from walking this directory
  devel/                 -- DEVELOPMENT SPACE (set by CATKIN_DEVEL_PREFIX)
    bin/
    etc/
    include/
    lib/
    share/
    .catkin
    env.bash
    setup.bash
    setup.sh
    ...
```

Figure 4.13: Structure of Catkin Workspace [12]

The source space has the source code of catkin packages. This is where the source code for the packages build is placed. The root of the source space contains a symbolic link to catkin's boiler-plate top-level CMakeLists.txt. This file is invoked by CMake during the configuration of the catkin projects in the workspace.

The build space is where CMake is invoked to build the catkin packages located in the source space. And also it is where CMake and catkin save their cache information and other intermediate files [37].

The devel space is where built targets are placed to being installed. The targets are organized in the devel space in the same way as their layout when they are installed.

The binary catkin package includes a set of environment setup files that are used to extend the shell environment so that they can find and use any resource that has been installed to that location.

4.2.2 Combine Dear ImGui with ROS environment

After the setup of the catkin workspace, it's needed to incorporate Dear ImGui framework in the workspace as shown in Figure 4.14. Adding all the dependencies and files necessary to create the executable project there and build a package with Dear ImGui context.

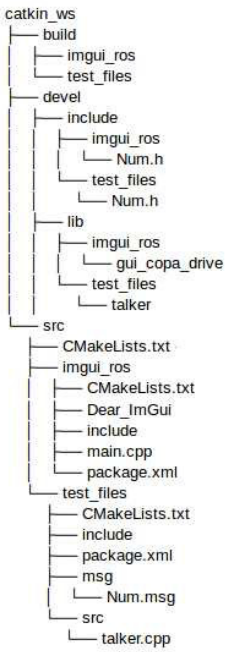


Figure 4.14: Filesystem Dear ImGui and ROS

It also require to make some changes to the main code of the GUI project, in order to make the GUI able to subscribe a topic and display a graph with their values.

Besides that, to test the communication in the ROS environment, a simple Talker node was created. That node generates some random number and publishes them to a certain topic.

The main code of GUI had to add ”ros/ros.h” which include all the headers necessary to use the most usual public pieces of ROS system and the ”std_msgs/Float32.h” that include the std_msgs/Float32 message, which resides in the std_msgs package.

```
#include "ros/ros.h"
#include "std_msgs/Float32.h"
```

Figure 4.15: Includes for ROS environment

To initialize ROS, the function ros::init() is called. The ImGui_ROS node subscribe a messages through the function ”ros::Subscriber sub = n.subscribe(”chatter”, 1000, Callback)”, subscribing to the ”chatter” topic. ROS will call the Callback() function whenever a new message arrives, as Figure 4.16 shows.

```
ros::init(argc, argv, "listener");
ros::NodeHandle n;
ros::Subscriber sub = n.subscribe("chatter", 1000, Callback);
```

Figure 4.16: ROS initialization and subscribing

The Callback() function, Figure 4.17, is, as the name suggests, the callback function that is called when a new message has arrived on the ”chatter” topic. It is where the values are received, stored and processed to be able to use in the ImGui::PlotLine() function. This is a Dear ImGui function that allows to create dynamic graphs.

```
void Callback(const std_msgs::Float32::ConstPtr& msg)
{
    ROS_INFO("I heard: [%f]", msg->data);
    values_ROS_float = msg->data;
    values_ROS[values_offset_ros] = msg->data;
    values_offset_ros = (values_offset_ros+1) % IM_ARRAYSIZE(values_ROS);
}
```

Figure 4.17: Callback function

The Talker node, Figure 4.18, publish a messages through the function ”ros::Publisher chatter_pub = n.advertise<std_msgs::Float32>(”chatter”, 1000)”, which communicate to the master that a message of type std_msgs/Float32 is being published on the topic ”chatter”. This allows the master to communicate to any nodes who is listening to ”chatter” topic.

```
int main(int argc, char **argv)
{
  ros::init(argc, argv, "talker");
  ros::NodeHandle n;
  ros::Publisher chatter_pub = n.advertise<std_msgs::Float32>("chatter", 1000);
  ros::Rate loop_rate(10);

  static float count = 0;
  static float phase = 0;
  static int values_offset = 0;
  static float values[90] = { 0 };
  while (ros::ok())
  {
    std_msgs::Float32 msg;
    msg.data = values[values_offset];
    chatter_pub.publish(msg);

    ros::spinOnce();
    loop_rate.sleep();
    values_offset = (values_offset+1); //% values[count];
    phase += 0.1*values_offset;
    values[values_offset] = cosf(phase);
    count++;
  }
  return 0;
}
```

Figure 4.18: Talker Node

4.2.2.1 Compile packages through CMakeList.txt

As explained above, ROS works with CMake using file CMakeLists.txt as an input to the CMake build system for building software packages. Any CMake-compliant packages contain one or more CMakeLists.txt files that describe how to build the code and where to install it.

As mentioned earlier, Dear ImGui is independent of the rendering system and platform, in such a way to create a functional program that need to provide specifications to compile the project with the SDL2 platform and the OpenGL renderer, as shown on Figure 4.19 and 4.20.

```
set(GL_DIR Dear_ImGui/examples/libs/gl3w)
include_directories(${GL_DIR})
find_package(OpenGL REQUIRED)
```

Figure 4.19: Include GL3W directories to compile

```
find_package (SDL2 REQUIRED)
include_directories(${SDL2_INCLUDE_DIRS})
link_libraries(${SDL2_LIBRARIES})
```

Figure 4.20: Include SDL2 directories and libraries to compile

When it is needed to specify which other CMake packages are needed to be found to build the project, the function find_packages() has to be called. That result in the creation of several CMake environment variables that give information about the founded packages, such as the localization of the headers files exported by the packages, source files, required libraries and the associated path.

To add any directories that need to be included, the function include_directories() serves this purpose. Normally, the argument should be the *_INCLUDE_DIRS variables generated by the find_packages() call.

The function link_libraries() is used to specify libraries to build. By default, catkin builds shared libraries.

To specify an executable target that must be built, we use the add_executable() function. In this project, this builds an executable target called "gui", which is built from the sources of Dear_ImGui framework, as shown in Figure 4.22

```
set(IMGUI_DIR Dear_ImGui)
include_directories(${IMGUI_DIR} ..)
```

Figure 4.21: Include Dear ImGui directories

```
file(GLOB sources *.cpp)

add_executable(gui ${sources}

${IMGUI_DIR}/examples/imgui_impl_sdl.cpp
${IMGUI_DIR}/examples/imgui_impl_opengl3.cpp
${IMGUI_DIR}/imgui.cpp
${IMGUI_DIR}/imgui_draw.cpp
${IMGUI_DIR}/imgui_demo.cpp
${IMGUI_DIR}/imgui_widgets.cpp
${GL_DIR}/GL/gl3w.c)
```

Figure 4.22: Build executable target gui_copa_drive

A target that depends on some other target that needs messages/services/actions to be built, need to add an explicit dependency on target catkin_EXPORTED_TARGETS, through the function add_dependencies(), so they are built in the correct order. This case applies to the target "gui", that need to receive messages. Additionally, function target_link_libraries() is used to specify which libraries the executable target link against, as the Figure 4.23 shows.

```
add_dependencies(gui ${catkin_EXPORTED_TARGETS} gui_copa_drive_gencpp)
target_link_libraries(gui ${catkin_LIBRARIES} ${Boost_LIBRARIES} ${SDL2_LIBRARIES}
${OpenGL_LIBRARIES} GL dl)
```

Figure 4.23: Add dependencies and libraries to target gui

After running the catkin_make command, it builts the target ”gui”. The communication between the talker and the ImGui_ROS, which is the name of the node responsible for subscribing inside the ”gui” target, is made through the publication of a message by the talker node to the ”chatter” topic, which is subscribed by the ImGui_ROS node, as shown in Figure 4.24. In Figure 4.25 it's possible to see the final result of the graphic respond.

Therefore, we can conclude that the implementation of GUI into the ROS environment was successful.

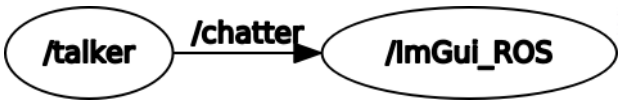


Figure 4.24: Communication between Talker and gui

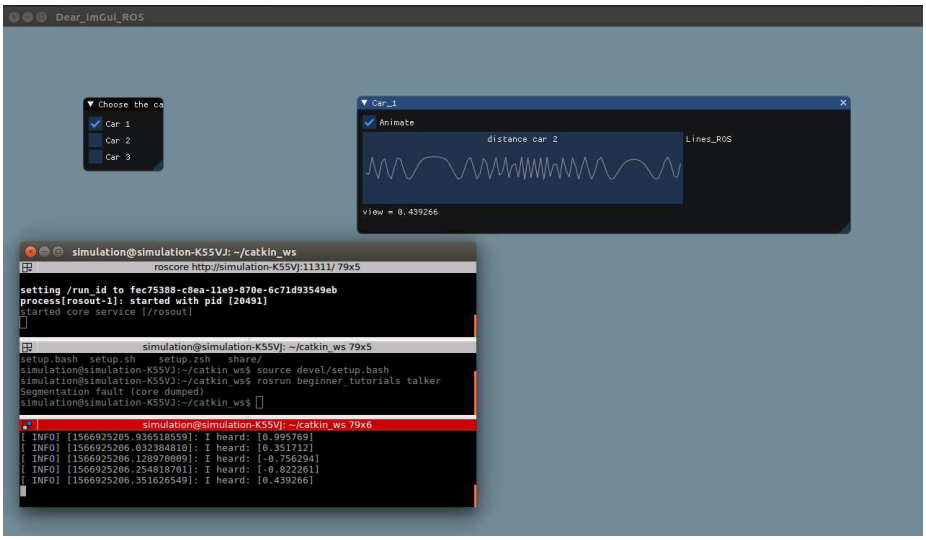


Figure 4.25: Graphic respond from interaction between Talker and gui

4.3 Gazebo Platooning Model

COPADRIVe has mention early on this Thesis, is a realistic simulation framework for cooperative autonomous driving applications which has a platoon model scenario implemented under ROS and Gazebo.

The Gazebo Platooning model is based on a V2V reliant platooning algorithm, featuring a platoon of three vehicles starting in a stop-motion manner already aligned inside a scenario resembling a motor sports track.

In this particular scenario, communication is made between the front car and those following one, which receive information transmitted by the front car

and process that same information in order to actuate properly. An example of vehicles running the platoon model during a simulation run is shown in Figure 4.26.

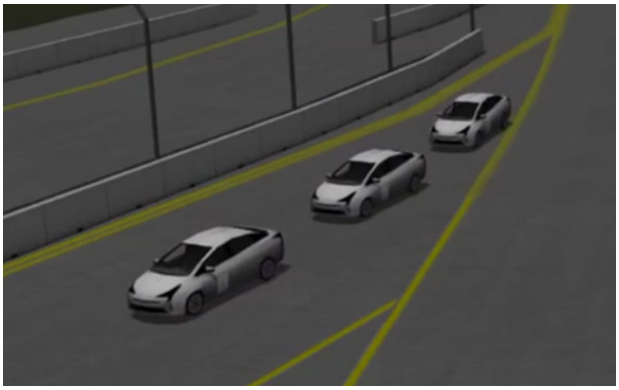


Figure 4.26: Vehicles running in platoon simulation

A rqt_graph demonstrating all the connections are sorted out is appeared in Figure 4.27. The blocks named as "carX" represent the namespace for each vehicle. Inside these, smaller blocks represent topics, which are being published by these same vehicles modules. Additionally, circles represent ROS nodes that may be publishing or subscribing.

Every vehicle has important applications running in their nodes. One module is in charge of the topic publishing, as "carX/carINFO" that is a topic giving information with respect to the present status of the vehicle, steering angle, speed, GPS coordinates, heading, etc. This topic is distributed in a 20Hz recurrence, running exclusively with gazebo and ROS subjects as the main correspondence instruments accessible [11].

Separated from this module, every one of the vehicles include a control application that uses information from several topics in order to control the vehicle actuators to pursue the direction appropriately.

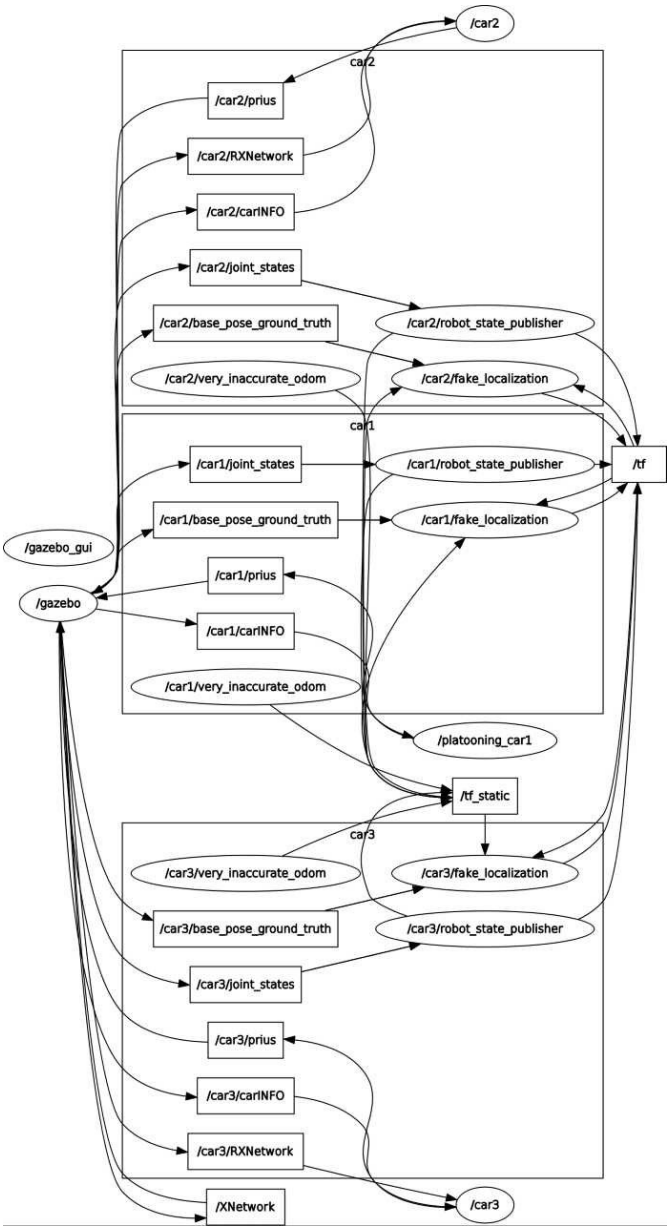


Figure 4.27: Gazebo platoon simulation

4.3.1 Integration Dear ImGui package on Platoon Simulation

The Gazebo Platoon Simulation as mention above is based on a V2V reliant platooning algorithm. The front car share data with the following car, that use it to control the vehicle actuators to seek after the course fittingly.

Due to the lack of any available software platform or architecture that allows the user to monitor its diagnostics, the Dear ImGui packages developed into the ROS system is used to enable a quick and easy interaction with the simulation. In short, the GUI developed is used to congregate the data published by the car nodes in one place, allowed to facilitate the work of data analysis and processing.

The information that matters to be viewed in the graphical interface is the one who represent the present status of the vehicle such as sterring angle, speed and GPS coordinates. These information are published on the topic "carX/carINFO", every vehicle has their own application running in their nodes. Also, it's important know the distance between a car an the front one, that data is published on the topic "carX/Platoon_dist".

For visualize that information, the GUI must include the type of message that are being publish to the topic, as shown on Fig. 4.15 as well as add in the CMakeLists.txt the REQUIRED COMPONENTS "ros_its_msgs" on the find_package() function.

```
#include <ros_its_msgs/CAM_simplified.h>
#include <ros_its_msgs/platoon_dist.h>
```

Figure 4.28: Include librarie

In other to receive the data from that topics the main.cpp of GUI should has subscribers functions appointed to them, as shown on Fig.4.29.

These functions receive the data using the callback() functions, "carX_info_Callback()", Figure 4.30, and "carX_distance_Callback()", Figure 4.31. Processing the collect data and make them available to integrate the ImGui::PlotLines() function.

```
ros::Subscriber sub_car1_INFO = n.subscribe("/car1/carINFO", 100, car1_info_Callback);
ros::Subscriber sub_car2_dist = n.subscribe("/car2/Platoon_dist", 100, car2_distance_Callback);
```

Figure 4.29: Subscribe functions

```
void car1_info_Callback(const ros_its_msgs::CAM_simplified::ConstPtr& msg)
{
  values_steer_car1 = msg->steeringWheelAngle_steeringWheelAngleValue;
  values_speed_car1 = msg->speed_speedValue;
  speed_car1[offset_car1] = msg->speed_speedValue;
  steer_car1 [offset_car1] = msg->steeringWheelAngle_steeringWheelAngleValue;
    offset_car1 = (offset_car1+1) % IM_ARRAYSIZE(speed_car1);
  position_x = msg->latitude;
  position_y = msg->longitude;
}
```

Figure 4.30: Callback ”carX/carINFO” topic

```
void car2_distance_Callback(const ros_its_msgs::platoon_dist::ConstPtr& msg)
{
  values_distance_car2 = msg->leader_distance;
  distance_car2[offset_distance_car2] = msg->leader_distance;
  offset_distance_car2 = (offset_distance_car2+1) % IM_ARRAYSIZE(distance_car2);
}
```

Figure 4.31: Callback ”carX/Platoon_dist” topic

The final product of this implementation, using the Gazebo Platoon simulation integrated with the Graphic User Interface can be shown in Figure 4.32 and 4.33. The first images demonstrate the Gazebo Platoon simulation running and the second images display the graphics response to what's happening on the simulation.

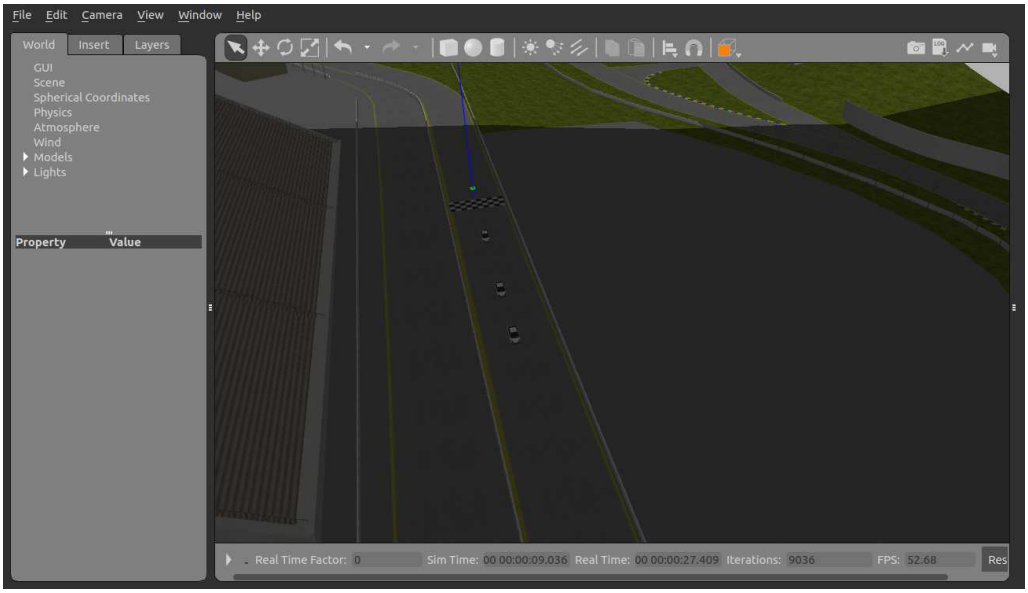


Figure 4.32: Gazebo Platoon simulation running

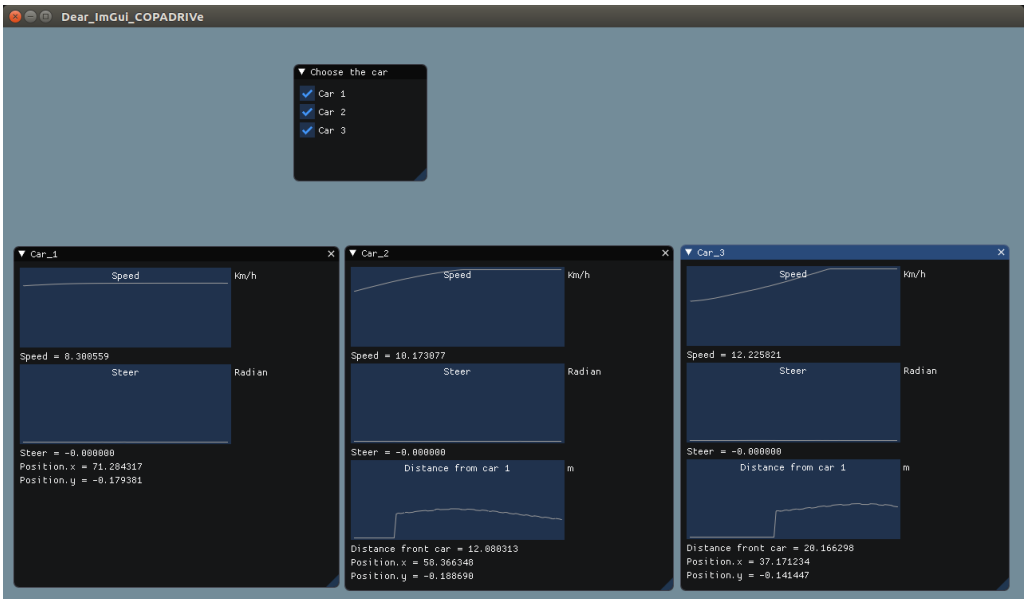


Figure 4.33: GUI developed under Gazebo Platoon simulation

Chapter 5

Conclusion

This chapter reviews the objectives proposed in this Thesis and analyzes its final result, summarizing its main contributions. Finally, some considerations about future work will be presented, considering the work developed in this Thesis

The work presented in this Thesis aims to create a tool that facilitates the intercommunication between the developer and COPADRIVe, by enabling the visualization and data analysis in real time and in a more visual and intuitive way.

Therefore, it was proposed to implement an existing GUI framework named as Dear ImGui, which differs from the traditional frameworks since it provides an immediate mode GUI, making it responsive to real-time environment based systems.

The implementation of this framework in the ROS environment was successful since, as result of Thesis, we developed a package that allows the integration of Dear ImGui framework into any ROS based system. This package has been tested on the COPADRIVe simulation tool. It create a scenario analysis at run-time, which allows the developer know, during the course of the simulation, how data transfers between vehicles and simplifies the error detection and correction process.

Also, this package is useful for simulation tools that test other types of communication systems within the autonomous vehicle environment.

All procedures and documentation for this implementation are available in this Thesis, for future reference when all the work done in the area of cooperative autonomous driving in ROS1 is updated to ROS2, which will enable real-time communications.

5.1 Future Work

As future work, we plan to implement a system capable of controlling certain simulation parameters without the need to manually reprogram these same variants, building a more integrated simulation environment for COPADRIVe. That will allow the GUI to control the various components of a simulation tools.

Additionally, when upgrading from ROS1 to ROS2 developments, the goal will be to adapt this tool to the new environment, which is already prepared to support real-time operations.

References

[1] "Windows 10 user interface solution | conceptdraw.com." https://www.conceptdraw.com/solution-park/software-windows-user-interface. [cited on p. iii, 6]

[2] "Qt quick controls - imagine style example: Automotive | qt quick controls 5.15.0." https://doc-snapshots.qt.io/qt5-dev/qtquickcontrols-imagine-automotive-example.html. [cited on p. iii, 7]

[3] "Getting started with gtk+: Gtk+ 3 reference manual." https://developer.gnome.org/gtk3/stable/gtk-getting-started.html. [cited on p. iii, 7]

[4] "Audacity ® | free, open source, cross-platform audio software for multi-track recording and editing.." https://www.audacityteam.org/. [cited on p. iii, 8]

[5] "Moveit! quickstart in rviz — moveit_tutorials kinetic documentation." http://docs.ros.org/kinetic/api/moveit_tutorials/html/doc/quickstart_in_rviz/quickstart_in_rviz_tutorial.html. [cited on p. iii, 9]

[6] "Gallery: Post your screenshots / code here (part 2) · issue #539 · ocornut/imgui · github." https://github.com/ocornut/imgui/issues/539. [cited on p. iii, 10]

[7] "Github - ocornut/imgui: Dear imgui: Bloat-free immediate mode graphical user interface for c++ with minimal dependencies." https://github.com/ocornut/imgui. [cited on p. iii, 9, 11]

[8] "P2p-accelerated streaming with webrtc | wowza." https://www.wowza.com/resources/guides/p2p-unicast-streaming. [cited on p. iii, 14]

[9] Y. Maruyama, S. Kato, and T. Azumi, "Exploring the performance of ros2," pp. 1–10, 10 2016. [cited on p. iii, 15, 16]

[10] "Sec. 2: Driving the husky robot in gazebo · smartlab-purdue/ros-tutorial-gazebo-simulation wiki · github." https://github.com/SMARTlab-Purdue/ros-tutorial-gazebo-simulation/wiki/Sec.-2:-Driving-the-Husky-robot-in-Gazebo. [cited on p. iii, 18]

[11] E. V. F. A. K. E. T. Bruno Vieira, Ricardo Severino, "Copadrive - a realistic simulation framework for cooperative autonomous driving applications." Cister Research Centre. [cited on p. iii, 2, 18, 31]

[12] "catkin/workspaces - ros wiki." http://wiki.ros.org/catkin/workspaces#Catkin_Workspaces. [cited on p. iii, 25]

[13] "Advanced driver assistance systems - an overview | sciencedirect topics." https://www.sciencedirect.com/topics/engineering/advanced-driver-assistance-systems, 2016. [cited on p. 1]

[14] "V2v: What are vehicle-to-vehicle communications and how do they work?." https://bit.ly/2lvho49. [cited on p. 1]

[15] "What is vehicle-to-infrastructure (v2i) communication and why do we need it?." https://bit.ly/2jZfYyn. [cited on p. 1]

[16] "What is a graphical user interface?." https://www.itpro.co.uk/operating-systems/30248/what-is-a-graphical-user-interface. [cited on p. 5]

[17] "Gui (graphical user interface) definition." https://techterms.com/definition/gui. [cited on p. 6]

[18] "Natural user interfaces – what are they and how do you design user interfaces that feel natural? | interaction design foundation." https://bit.ly/2mcfP7g. [cited on p. 6]

[19] "Qt | cross-platform software development for embedded & desktop." https://www.qt.io/. [cited on p. 6]

[20] "Qt for embedded linux | qt 4.8." https://doc.qt.io/archives/qt-4.8/qt-embedded-linux.html. [cited on p. 6]

[21] "The gtk project." https://www.gtk.org/. [cited on p. 7]

[22] "The gtk open source project on open hub." https://www.openhub.net/p/gtk. [cited on p. 7]

[23] "gtk-wimp.sourceforge.net." http://gtk-wimp.sourceforge.net/. [cited on p. 7]

[24] "wxwidgets: Cross-platform gui library." https://www.wxwidgets.org/. [cited on p. 8]

[25] "rviz - ros wiki." http://wiki.ros.org/rviz. [cited on p. 8]

[26] "Immediate mode model/view/controller." http://www.johno.se/book/imgui.html?fbclid=IwAR04sqs_jb6TThpIcatmFAtxQ_5BlAooFWPfEjydedw1fobbblQ0e-D4L-o. [cited on p. 9, 10]

[27] "Could imgui be the future of guis?." https://games.greggman.com/game/imgui-future/. [cited on p. 11]

[28] "Ros/introduction - ros wiki." http://wiki.ros.org/ROS/Introduction. [cited on p. 13]

[29] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," 2009. [cited on p. 14]

[30] M. P. Thomas White, Michael N. Johnstone, "An investigation into some security issues in the dds messaging protocol," 2017. [cited on p. 16]

[31] "Gazebo." http://gazebosim.org/. [cited on p. 17]

[32] X. W. LINGJIE YANG, ZHIHONG LIU and Y. XU, "An optimized image-based visual servo control for fixed-wing unmanned aerial vehicle target tracking with fixed camera." College of Intelligence Science and Technology, National University of Defense Technology, Changsha 410073, China, 5 2019. [cited on p. 17]

[33] C. F. R. Riebl, H. Gnther and L. Wolf, "Artery: Extending veins for vanet applications," 2015. [cited on p. 18]

[34] "Simple directmedia layer - sdl version 2.0.10 (stable)." https://www.libsdl.org/download-2.0.php. [cited on p. 20]

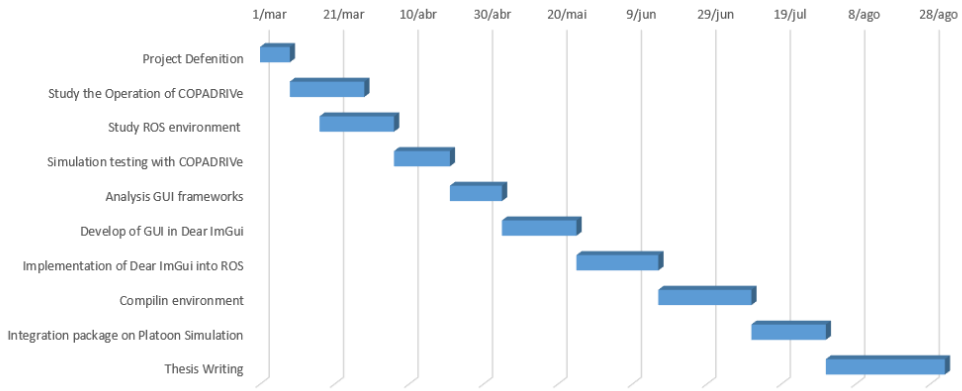
[35] "Khronos opengl® registry - the khronos group inc." https://www.khronos.org/registry/OpenGL/index_gl.php. [cited on p. 20, 25]

[36] "Github - skaslev/gl3w: Simple opengl core profile loading." https://github.com/skaslev/gl3w. [cited on p. 20]

[37] "catkin/cmakelists.txt - ros wiki." http://wiki.ros.org/catkin/CMakeLists.txt. [cited on p. 26]

Appendix A

Project Roadmap



Appendix B

CMakeLists.txt template for Dear ImGui package

```
cmake_minimum_required(VERSION 2.8.3)
project(imgui_ros)

find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  message_generation
)

find_package(catkin REQUIRED COMPONENTS
message_generation
rostime
roscpp
rosbag
rosconsole
roscpp_serialization
rospy
gazebo_ros
ros_its_msgs
)

find_package(Boost REQUIRED)
link_directories(${catkin_LIBRARY_DIRS} )
catkin_package(CATKIN_DEPENDS message_runtime std_msgs)
##SET(CMAKE_CXX_FLAGS "-std=c++0x" )

include_directories(
  include
  ${catkin_INCLUDE_DIRS}
```

```
)


if(NOT CMAKE_BUILD_TYPE)
  set(CMAKE_BUILD_TYPE Debug CACHE STRING "" FORCE)
endif()

set (CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -DVK_PROTOTYPES")
set (CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -DVK_PROTOTYPES")



# GLFW
set(GL_DIR imgui/examples/libs/gl3w)
include_directories(${GL_DIR})

#SDL2

find_package (SDL2 REQUIRED)
include_directories(${SDL2_INCLUDE_DIRS})
link_libraries(${SDL2_LIBRARIES})

# ImGui
set(IMGUI_DIR imgui)
include_directories(${IMGUI_DIR} ..)

# Libraries
set(OpenGL_GL_PREFERENCE GLVND)
find_package(OpenGL REQUIRED)

file(GLOB sources *.cpp)

add_executable(gui_copa_drive ${sources}
${IMGUI_DIR}/examples/imgui_impl_sdl.cpp

${IMGUI_DIR}/examples/imgui_impl_opengl3.cpp
${IMGUI_DIR}/imgui.cpp
${IMGUI_DIR}/imgui_draw.cpp
${IMGUI_DIR}/imgui_demo.cpp
${IMGUI_DIR}/imgui_widgets.cpp
${GL_DIR}/GL/gl3w.c)

add_dependencies(gui_copa_drive ${catkin_EXPORTED_TARGETS} gui_gencpp)

target_link_libraries(gui_copa_drive ${catkin_LIBRARIES} ${Boost_LIBRARIES} ${SDL2_LIBRARIES}

${OpenGL_LIBRARIES} GL dl glfw)
```