Full Stack engineering in Robot Open Autonomous Racing

Michael Wu
Allen Yang, Ed.
S. Shankar Sastry, Ed.

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2022-4
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-4.html

April 14, 2022

Copyright © 2022, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Full Stack engineering in Robot Open Autonomous Racing

by

Michael Wu

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Masters of Electrical Engineering and Computer Science

in

Electrical Engineering and Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Shankar Sastry, Chair
Dr. Allen Yang

Spring 2022

Full Stack engineering in Robot Open Autonomous Racing

Copyright 2022
by
Michael Wu

Abstract

Full Stack engineering in Robot Open Autonomous Racing

by

Michael Wu

Masters of Electrical Engineering and Computer Science in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Shankar Sastry, Chair

Autonomous Driving is a research topic that daunts many generations of engineers. With the advancement of AI and computing power, academia and industry has made significant progress in recent decade. However, high fragmentation, incompatibility across fields, and high initial cost still poses a tall barrier to entry prevents junior engineers such as student researcher or even grade school learners to get started, not even so to contribute to real world problems.

The purpose of this paper is to introduce my contributions in the development of the novel system ROAR [23] that attempts to address the problem of fragmentation, incompatibility, and high cost in the autonomous driving industry. ROAR, name originally created for Robot Open Autonomous Racing, is a competition that encourages young students to get their first experience with engineering, allow researchers to advance their project without spending years on infrastructure, and a platform where extra components can be added with ease.

This paper first will go over the high level design principles and rationals in the Introduction section. Then it will go into details, touch upon a wide variety of technical topics, including 3D design, electrical engineering, vehicle dynamics, embedded systems, wireless communications, software architectures, advanced algorithms, and much more. This paper will outline the engineering effort in ROAR from design to actual implementation and the eventual verification and testing.

To my Family

I would like to express my sincere gratitude for my parents to support me through one the best education possible and support me during many difficulties. From being a shy, timid, boy out of musician family, I had the dream of becoming an engineer – an engineer who can create and innovate. From the first "engineering" project that I remember, a percussion instrument made from door handles parts at the age of 6, to the very complex project involving the entire stack, from hardware, embedded systems, to high level algorithms and system architecture, I have truly grown a lot.

Contents

Contents ii

List of Figures iii

List of Tables v

1 Introduction 1
- 1.1 Motivation . . . 2
- 1.2 Related Work . . . 3

2 ROAR 9
- 2.1 ROAR with Jetson Nano . . . 9
- 2.2 ROAR with iPhone . . . 11
- 2.3 ROAR Overall Design . . . 12
- 2.4 Hardware . . . 14
- 2.5 Embedded Systems . . . 16
- 2.6 Communication Pipeline . . . 16
- 2.7 Software (Python) . . . 21
- 2.8 Simulation . . . 22
- 2.9 ROAR Gym Integration . . . 25
- 2.10 ROS2 Integration . . . 26
- 2.11 Virtual Reality Integration . . . 27
- 2.12 Miscellaneous Discussions . . . 30

3 Autonomous Driving Algorithms 33
- 3.1 Perception . . . 33
- 3.2 Planning . . . 37
- 3.3 Control . . . 41

4 Appendix 46
- 4.1 ROAR Junior . . . 46

Bibliography 51

List of Figures

1.1 Comparisons of low end platofmrs 1
1.2 Coursera Self Driving Car Specialization Autonomous Driving pipeline 3
1.3 Betz et. al Autonomous Vehicles on the Edge: A Survey on Autonomous Vehicle Racing Autonomous driving pipeline including both hardware and software that provides the categorization for the survey in the field of autonomous racing . . . 4
1.4 Yolo with LSTM diagram 5
1.5 DeepSORT algorithm 5
1.6 Edward Fang Person behind Car Scenario Hybrid A* 6
1.7 Map Representation Comparisons 7

2.1 System diagram of ROAR Software Virtualization architecture. Through a carefully designed virtualization layer, learners who may not have skills or access to vehicle hardware can still effectively learn and practice in Python, and test and deploy their experiments on either graphics simulator or reinforcement learning gym environments. 10
2.2 Original ROAR vehicle design with Vive Tracker 10
2.3 Original ROAR vehicle design with Vive Tracker 11
2.4 ROAR Car with iPhone mount 12
2.5 ROAR Car with iPhone mount 13
2.6 Signal Wiring for ROAR with iPhone 14
2.7 New iPhone mount with more structural integrity 15
2.8 Left is ESP-32 Cam, Right is Arduino Nano BLE Sense 17
2.9 Embedded system Ensure Smooth Transition 18
2.10 A screenshot of the CARLA simulator environment. The red car is controlled by a ROAR Python agent. Other vehicles can also be added to follow certain pre-programmed waypoints that share the same road. The simulator can also generate common vehicle sensor data, which is not shown here. 23
2.11 Carla Server and Client model 23
2.12 Berkeley Minor in Carla 24
2.13 Model Based RL Pipeline 25
2.14 ROAR ROS2 transform hierarchy and message pipeline 26
2.15 ROS2 Robot Model Display 27
2.16 ROAR VR Demonstration. Also available on Youtube 28

2.17 System Diagram for ROAR with Virtual Reality integration 29
2.18 Single Image Anchor. Left is individual run; Right is three run overlapped. Clear deviation as time goes on and low precision between the runs. 31
2.19 Multi Image Anchor. Left is individual run; Right is three run overlapped. Deviation is lower, precision is higher. 32

3.1 IOU Visual Diagram 34
3.2 Appearance Descriptor Vector 35
3.3 DeepSort Example. Full video available here 36
3.4 Obstacle detection in the camera frame with LiDAR sensor. On the upper left is Point Cloud, on the lower left is ground plane detection with three regions of interest 37
3.5 Waypoint Following Example. r = searching radius. Red dots are waypoints. Triangle is ego vehicle 38
3.6 Lane Mask Example. Full video can be found at this link 39
3.7 Lane Following Finite State Machine 39
3.8 Vehicle Platooning using ARUCO Marker. Full video can be found here 40
3.9 PID Controller Output example 41
3.10 Pure Pursuit Geometric Relationship Diagram 42
3.11 Finite State Machine for Lead vehicle and follower 44
3.12 System Integration of Lane Following, Obstacle Avoidance, and Platooning. The Lead vehicle is doing Lane Following and obstacle Avoidance. The following vehicle is doing obstacle avoidance and following the lead vehicle. Complete video available in demonstration video 45

4.1 ROAR Junior 46
4.2 L298N motor driver 48

List of Tables

2.1 Arduino Wiring Table 15
2.2 No packet drop UDP communication 20
2.3 Send Ack dropped UDP Sequence example 20
2.4 Content Drop UDP Communication Example 20

4.1 Bill of Material (BOM) of the ROAR Junior base layer 47

Acknowledgments

I would like to take the opportunity to thank everyone who helped me and helped me throughout this adventure. Note that there are way too many who I'd like to thank, and this is just a short list:

- Dr. Allen Yang – my mentor who guided me through a variety of difficulties from computer vision problems all the way to life questions. Thanks you so much!
- Adam Curtis – Genius in any hardware related tasks, thank you so much for teaching me how to use 3D printer and support me learning it!
- Aman Saraf – My go-to ROS2 correspondent, excellent in coordinate transformation and very hardworking
- Star Li – My go-to person for any math, statistic related questions. Good luck on your PhD!
- And so much more, thanks all!

Forwards

This paper introduces my approaches and thoughts in designing and implementing the ROAR platform and all of its associations. There might not be much ground-breaking new discoveries, but will have details on the roadblocks and solutions discovered during my journey in all engineering parts of a product. This paper's goal is to encourage more researches to get first person view into what is it like to be "Full Stack" in the field and avoid some pitfalls that I might have encountered.

Summary of My Contribution

It would be difficult to itemize my contributions since I worked on almost all aspects of ROAR's design from hardware to software. I have started working on the ROAR project since the summer of my Sophomore to Junior year in college – it has been almost 3 years since I worked on the project.

Initially joining the project in effort to bring up a simulation to satisfy research need during the COVID pandemic, I gradually expanded my role into the software architecture design of the entire system that unifies the data types, pipelines and processes. And then got my hands dirty with actual hardware and the embedded systems, personally leading over two iterations of hardware major renovations and a complete rewrite of the embedded systems. I also created a robust, lo-latency communication systems for the different components battle tested by dozens of other students. And lastly I contributed several modules to the Perception and Controls stack, with light breezes in the Plannings field.

Chapter 1

Introduction

Ralph Teetor introduced Cruise Control in 1948; 50 years later, in 1995, GPS was introduced US manufactured cars; fast forward now after the boom of AI, 5G and other technologies, we are on the verge of achieving level 5 autonomy. One major problem is the difficulty in accessing corner cases in complex real life scenarios. And that problem root itself in the fact that the development of autonomous driving technology is **Fragmented**, **incompatibilities across ecosystems**, **high cost for entry**.

Platform	Duckiebot	Donkey	MIT Racecar	F1tenth	AWS Deepracer
Processing Unit	Raspberry Pi	Raspberry Pi	NVidia Jetson TX2	Nvidia Jetson TX2	Intel Atom
Sensors	Webcam	Webcam	ZED depth + Hokuyo LIDAR	Intel D345i + Hokuyo LIDAR	Webcam
Chassis & Scale	Magician 1/24	Exceed 1/16	Traxxas Slash 1/10	Traxxas Rally 1/10	Proprietary 1/18
AI Functions	Lane following	Lane following, DNN imitation learning	MPC, SLAM, Reinforcement Learning	MPC, SLAM, Reinforcement Learning	Lane following, Reinforcement learning (on AWS)

Figure 1.1: Comparisons of low end platofmrs

1.1 Motivation

In recent decades, organizations and events such as the DARPA Grand Challenge [27], the Indy Autonomous Challenege [14] provided many hypes and hopes around when would level 5 capabilities finally come. There are a variety of companies leading the development of level 4 or 5 capabilities such as Tesla, Cruise, Waymo, Baidu, and et cetera. The core issue is that there needs to be more qualified engineer who can uncover all the edge cases and make the technology safe. Unfortunately the road to from academia research to real world application is a very **Fragmented**. Figure 1.1 demonstrates some popular platforms that people uses. RACECAR and F1tenth both are rooted from autonomous driving research at MIT and UPenn, respectively. Their goals are most similar to those of ours. However, employing Nvidia TX2 and Hokuyo LIDAR makes these cars a lot more heavy, power-hungry, and expensive. For example, a single Hokuyo 10LX LIDAR itself would be three times as expensive as the BOM cost of ROAR reference design.

When evaluating these scale model AI car platforms, we concluded that none of the existing platforms is an ideal platform to develop the EIR curriculum at Berkeley. For example, Duckiebot and Donkey use very low-cost chassis (1/16 or smaller) and low-end compute systems (Raspberry Pi plus a webcam).AWS Deepracer is a proprietary platform that is difficult to expand beyond its single webcam setup. The system prioritizes to perform most deep learning tasks on the AWS cloud, hence is not suitable to pursue extreme performance and extreme compatibility.

On a cheaper end, many Arduino or ESP based vehicles such as Freenove Smart Car [12] also emerged with cost typically under 100 dollars, unfortunately, most of those vehicles are differential drive, which is very different from the Ackermann drive mechanism that real vehicle uses.

The ROAR reference design is designed to pair with a variety of 1/10 scale RC car chassis. The 1/10 scale was chosen to have sufficient onboard space to later integrate DL-compatible compute modules and VR-compatible sensing and streaming modules. In our development, we have chosen Traxxas 4-Tec RC platform, which can be configured very flexibly to use a brushed motor to reach 35 mph or a brushless motor to reach 70 mph speed (the choice also affects its price to be either $226 or $376). Other similar RC car chassis at even lower prices can also be adopted to pair with ROAR reference design, such as Traxxas Rustler 2WD at $179 and Hosim Cross-Country truck at even lower $110.

Therefore the original ROAR platform came out of the need for a platform where starting up is easy, learning curve is gradual and not dependent on previous experience, and advanced research and add-ons are possible.

ROAR Junior emerged out of the finding that much of what ROAR do are in the software. Our design made it difficult for users to alter the embedded system's behaviors, and that is not what we'd like to see in an education experience. Therefore, we studied closely some Arduino based robots on the market and created our custom version where incremental updates is possible, thus lowering the learning curve and add interest and joy in the user experience. Since the majority of the disucssion in this paper will be centered around ROAR,

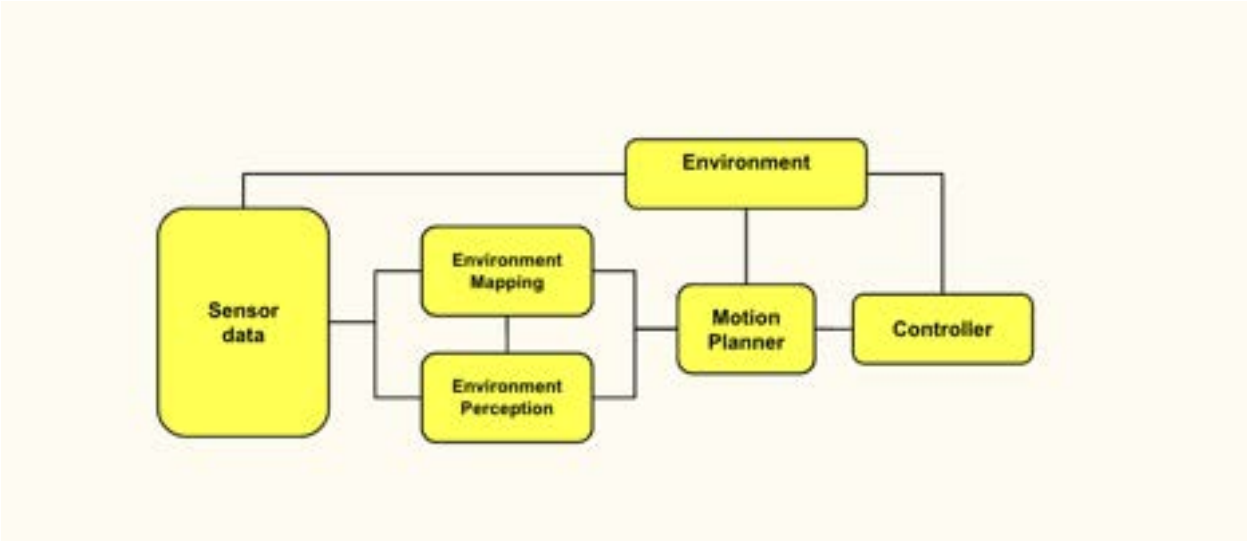

Figure 1.2: Coursera Self Driving Car Specialization Autonomous Driving pipeline

details of ROAR Junior will be moved to the Appendix section.

Lastly the collaboration with University of Hawaii (UHI) and University of California, San Diego (UCSD) let us realize that more complex software solutions require a common middleware, and it seems like ROS2 is the future direction to go [6]. We advanced on development of the *ROAR-ROS2-Bridge* that made our softwares compatible and made progress in further developments under the ROS2 middleware.

1.2 Related Work

In this section, I will go over briefly the entire pipeline of autonomous driving and what inspired me to dig deeper and try out certain algorithms. This section is meant to cover on a high level the algorithms that I've looked into. More detailed discussion is available in later chapters.

Autonomous Driving Life Cycle

Coursera's Self Driving Car Specialization Course [34] is probably one of the most famous courses that teach beginners about the whole lifecycle of self driving. One of the main knowledge that they proposed is the autonomous driving data pipeline which I will summarize in Figure 1.2. Similar pipeline is also proposed by Betz et.al in the paper Autonomous Vehicles on the Edge: A Survey on Autonomous Vehicle Racing [1], a figure of is attached in 1.3. The overall designs are very similar where sensors capture environmental data, which passes into perception for Localization and Detection modules; the output then is fed into

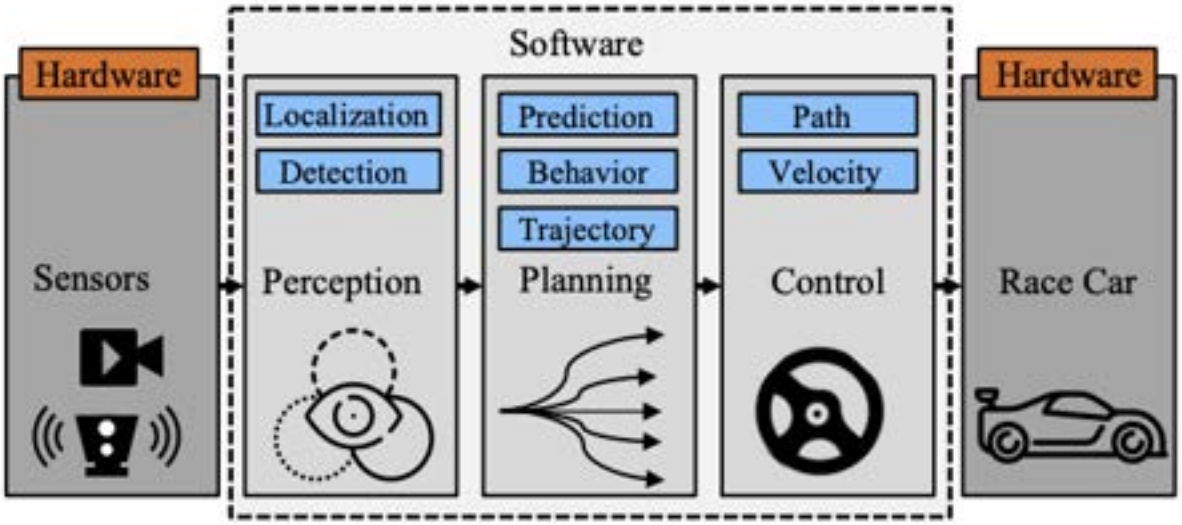

Figure 1.3: Betz et. al Autonomous Vehicles on the Edge: A Survey on Autonomous Vehicle Racing Autonomous driving pipeline including both hardware and software that provides the categorization for the survey in the field of autonomous racing

the planning modules for prediction and motion control; and lastly is fed into a Controls module that will eventually actuate the motor and steering.

Perception

Perception is a general terminology related to two major tasks:

- Detecting & Tracking obstacles
- Localization

For detecting and tracking obstacles, a neural net is most often being used. Author Ankit Sachan detailed the variety of object tracking methods in his article Zero to Hero [37], and stated that the difficulties may include Occlusion, Identity Switches, Motion Blur, ViewPoint variation, Scale Change, Background Clutters, Illumination Variation, and Low Resolution. He further proposed that a Yolo + Long Short Term Memory (LSTM) + CNN based neural net is very popular for the task of detecting and Tracking obstacles. The image pipeline is demonstrated in Figure 1.4. DeepSORT, on another hand, is also a pipeline popular for the Detection and Tracking task. Originally proposed by Wojke et. al, DeepSORT [35] is an extension on Simple Online Realtime Tracking for multiple object detection with an addition of a deep association metric for more accuracy. Ritesh Kanjee gave a really great beginner's

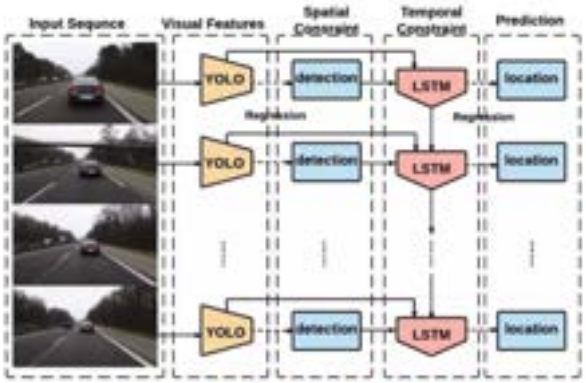

Figure 1.4: Yolo with LSTM diagram

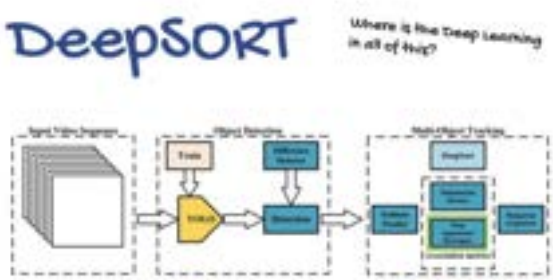

Figure 1.5: DeepSORT algorithm

introduction on the algorithm [8]. An summary of DeepSORT will be explained later and the algorithm figure is attached in Figure 1.5

For our application, we are going to be mostly indoors, Biswas et. al's Depth Camera Based Indoor Mobile Robot Localization and Navigation [2] looked particularly interesting. Their approach proved to us that using depth camera might be sufficient for our need. At the same time, we also looked into Intel's T265 Tracking Camera [30] to see if we can get a solution "out-of-box". However the easiest and the most reliable solution would be one that is "out-side-in". Similar how GPS would work, there will be base stations placed around a room that can detect some sort of trackers that is attached to the vehicle. Kunhoth et. al wrote a survey on Indoor Positioning and wayfinding systems [17] that introduced the variety indoor localization methods.

Figure 1.6: Edward Fang Person behind Car Scenario Hybrid A*

Planning

Path planning for vehicles is a robotic decision making problem. Mincheul Kim reviewed the major path planning algorithms in his thesis quite well [16], Edward Fang experimented the variety of algorithms and their possible enhancements in his paper Dynamic Deadlines in Motion Planning for Autonomous Driving Systems [11]. An example of the trajectory he generated with in the Carla simulator is shown in Figure 1.6.

Behind all these algorithms lies the deep question of data structure. How do I represent my world in an efficient way such that I can both easily update and also execute computation using the data structure. Katrakazas et. al did a survey on Real-time motion planning methods for autonomous on-road driving: State-of-the-art and future research directions [15]. They listed a set of generic algorithms and their pros and cons as demonstrated in Figure 1.7

Furthermore, many examples with code on both representation and path planning algorithms are available on AtsushiSakai's github [24].

Representation	Advantages	Disadvantages
Voronoi Diagrams	• Completeness • Maximum distance from obstacles	• Limited to static environments • Discontinuous edges
Occupancy Grids Cost Maps	• Fast discretisation • Small computational power[a]	• Problems with vehicle dynamics • Errors in the presence of obstacles
State lattices	• Efficiency without increasing computational time[b] • Pre-computation of edges is possible	• Problems with curvature • Restrict motion • Difficulties in dealing with evasive manoeuvres
Driving corridors	• Continuous collision free space for the vehicle to move	• Computational cost[c] • Constraints on motion

Figure 1.7: Map Representation Comparisons

Controls

Controls has been a long studied problem in the field. I want to introduce a Medium Article by Yan Ding [29] that compared three popular controls algorithms: Pure Pursuit, Stanley, and MPC. All of these algorithms are similar in that they have a model of the vehicle trajectory and is trying to fit or solve that optimization problem. At the same time, there is another very generic controls scheme named PID controls that also proven useful in a autonomous vehicles [19]

Chapter 2

ROAR

The purpose of Robot Open Autonomous Racing (ROAR) is to bring autonomous driving techniques and ideas to more audience, lowering the barrier to entry. Since I joined the project, there has been two major updates of the hardware and software solutions. In this chapter, I will first go over chronologically our advancements and then go into details regarding our current setup. An overview of the system architecture can be found in Figure 2.1

2.1 ROAR with Jetson Nano

The design of the original ROAR is heavily influenced by the F 1/10 racer vehicles. Our hardwares include

- Jetson Nano for on-board computing
- Vive Tracker for localization
- Arduino Nano for embedded systems communicating from computing unit to hardwares
- Traxxas 4-tec Chassis, motor, and Servo
- Intel RealSense D435i camera
- Custom designed upper chassis
- Arduino Cam for back camera
- 2 x 7.2V LiPo battery to power the ESC and the Jetson Nano seperately

Figure 2.2 and 2.3 demonstrate the full setup of the original ROAR vehicle.

This setup is on the expensive side: one vehicle of this setup will cost upward of $1500 dollars and the assembly time will be over 3 hours, not counting the 12 hour plus 3D printing time for the custom designed chassis. However, this setup does prove to be simple to debug

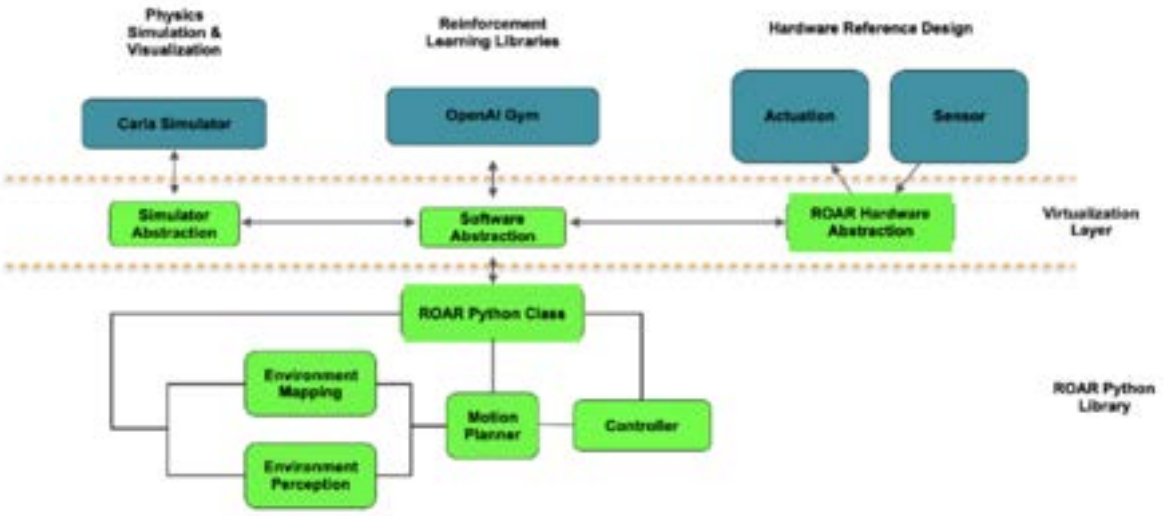

Figure 2.1: System diagram of ROAR Software Virtualization architecture. Through a carefully designed virtualization layer, learners who may not have skills or access to vehicle hardware can still effectively learn and practice in Python, and test and deploy their experiments on either graphics simulator or reinforcement learning gym environments.

Figure 2.2: Original ROAR vehicle design with Vive Tracker

Figure 2.3: Original ROAR vehicle design with Vive Tracker

since all components work seperate from another and that the computation of autonomous driving features happens onboard in Jetson Nano, which means that there will be little, if none at all, in transmission delay. However, this setup is on-par, if not better than the F1/10th setup due to better localization accuracy and computation power. We can do better by further reducing the cost and assembly time for achieving the goal of reduced cost.

The Github for the ROAR Jetson repo can be found at this link: ROAR Jetson but is currently deprecated.

2.2 ROAR with iPhone

Apple announced that all iPhones are going to have Lidar cameras on it starting from iPhone 12 Pro. That's when we started considering the possibility of simplifying our hardware to just using an iPhone because it has a depth sensor, a RGB camera, an IMU on-board, and iPhone itself is a really strong computing unit with GPU support. And most likely, users might have their own iPhone such that it does not add toward the overall cost of the vehicle.

Therefore, we designed our first generation iPhone mount as demonstrated in Figure 2.4. This design significantly shortend the assembly time from several hours to minutes. and the complexity of the system. The user simply need do four steps to get the vehicle up and running:

1. Download the ROAR application

2. Wire the Arduino

Figure 2.4: ROAR Car with iPhone mount

3. Install the iPhone mount
4. Plug the iPhone into the mount

And furthermore, the overall cost was reduced 75% from over $1500 to around $400 dollars excluding the cost of iPhone.

However, naturally, there will be pros and cons of this modification and the specifics will be discussed in the Hardware section in this chapter.

2.3 ROAR Overall Design

The core design philosophies for the ROAR architecture are Simple, Versatile, and Extensible. It is meant to be a platform where first-time users have a pleasant time on-boarding,

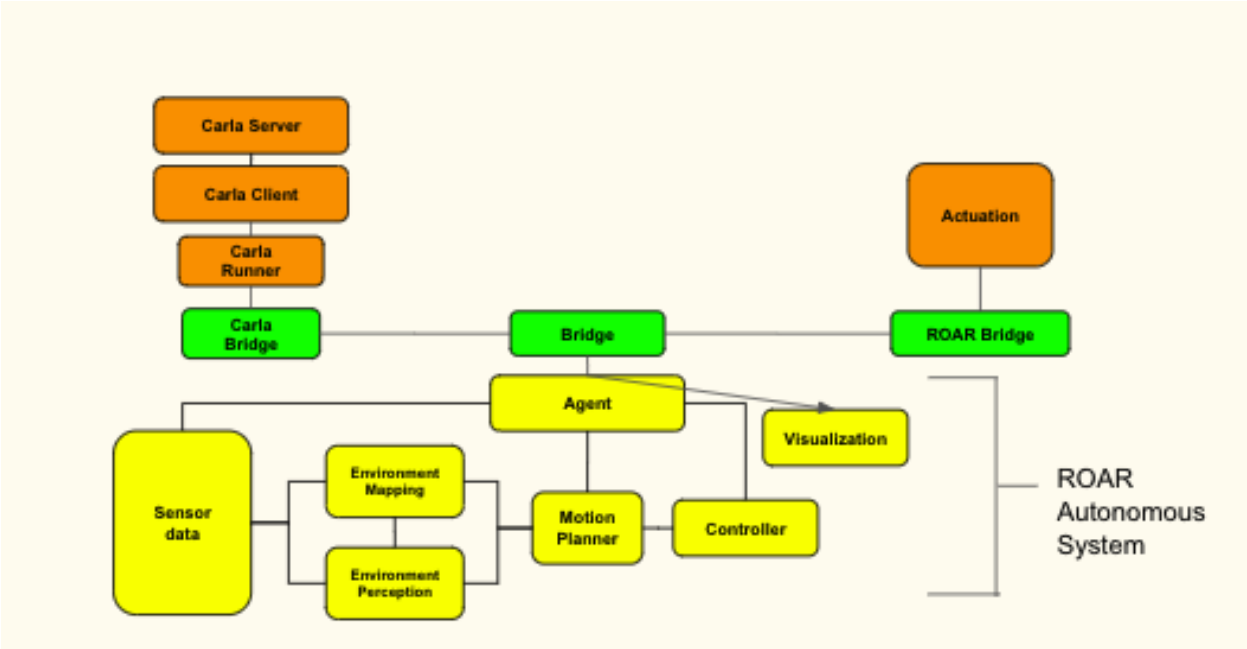

Figure 2.5: ROAR Car with iPhone mount

researchers have the ability to carry out in-depth studies in their particular interest, and further add-ons can be easily integrated. Therefore, we have produced a system architecture as shown in Figure 2.5.

We gave the pipeline that process the environment and output an actuation the name ROAR Autonomous Systems. The ROAR Autonomous Systems will talk to various environment, be a simulation or an actual vehicle via a dedicated Bridge. For simulation, we have chosen the Carla Simulator [10], and for Actuation, it could be either the Jetson version of the ROAR vehicle or the iPhone version.

Furthermore, our code base will reflect this design principle using submodules. The ROAR Autonomous System code base can be found at augcog/ROAR while its dependencies to other environments can be found as a submodule such as ROAR Sim. The main autonomous driving logic will be all in the main repo inside the ROAR directory which further split out into *agent_module*,*control_module*,*perception_module*,*planning_module*, and various other supporting modules that user can choose to combine for their specific use cases. And to run the ROAR system, user simply need to interact with a *runner_xxx.py* file where xxx represent different platform (jetson, iOS, or sim)

This methodology of exposing only a runner script in the outer most layer introduce abstraction barriers that exposes a short and easy-to-understand runner script for first time user, lowering the on-boarding technical barrier. The design of using submodules creates abstraction barriers so that high level engineers don't need to worry about the cumber-

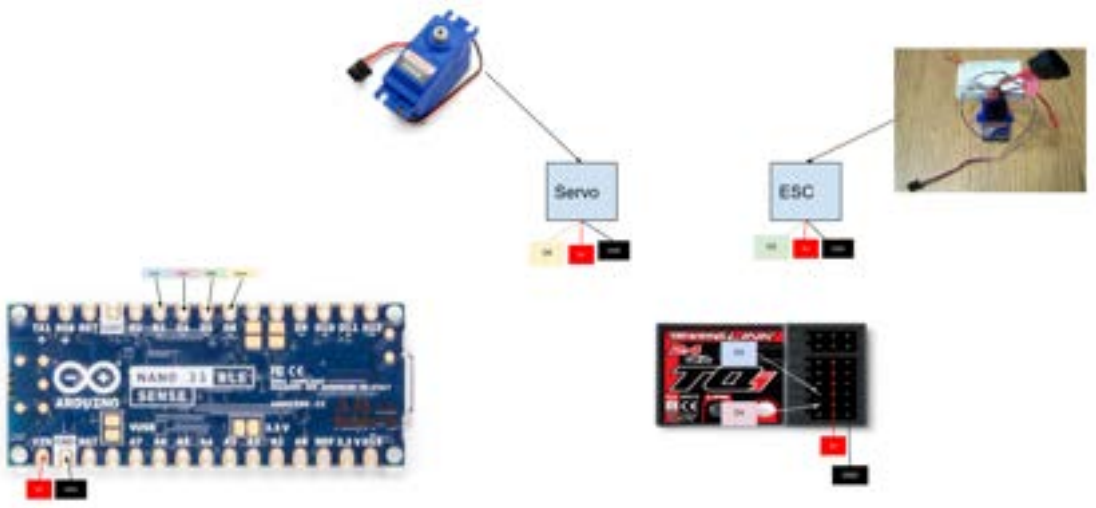

Figure 2.6: Signal Wiring for ROAR with iPhone

some pipeline that sends their signal to the motors. And the distinct divide in Perception, Plannning and Controls module encourage researchers to focus on their area while simply conforming their input and output to the existing working code.

However, as we later found out, there are downsides in designing this entire pipeline as well. Although this design proves to be easy to get-started, it is hard to implement parallelism, and the core culprit is due to Python's Global Interpreter Lock (GIL) [9]. And this is one of the reason why we started working on a ROS2 Bridge integration which will be discussed later in the chapter.

2.4 Hardware

As previously discussed, iPhone replaced our old setup of using multiple individual components and the impact this design change led to are reduced assembly time and system complexity.

The only wiring the user needs to do would be to wire up the signal outputs from the Arduino to the Servo and Electronic Speed Control (ESC) as demonstrated in Figure 2.6 and also in Table 2.4. One can safely disregard D3 and D4 if remote control is not needed at all.

Aside from wiring, the iPhone mount was easy to put on but comes at some disadvantages as we expanded the program and had more students to test it.

Arduino Pin	Destination Pin	Description
D3	Receiver Ch2	Remote control Steering
D4	Receiver Ch1	Remote control Throttle
D5	ESC Signal	Throttle command from Arduino
D6	Servo Signal	Steering command from Arduino

Table 2.1: Arduino Wiring Table

Figure 2.7: New iPhone mount with more structural integrity

The first disadvantage is that because the iPhone is placed in the very front, user's can't actually see the front of the car and therefore lost sense of "depth", and led to difficulty in manual control in some cases.

The second disadvantage relate to impact absorbance of the iPhone mount. We have taken extreme precaution when designing the iPhone protector. However, when impact happens, very often the iPhone is fine, but the mount is broken in two. And after extensive testing, we found that it is due to the fact that the iPhone sits at 90 degrees in the front of the vehicle, when experiencing impact, all the forces will turn into torque and be exerted onto the middle of the iPhone mount, which is simply a 3D printed 5 mm thick PLA plastic that will easily break. Our solution is to move the iPhone more toward the back so we let the vehicle's front bumper to absorb more impact and also to design a triangular support structure such that the mount itself is more structurally in-tact as seen in Figure 2.7

2.5 Embedded Systems

The embedded side of the ROAR system is relatively simple. In the original ROAR where we have Jetson Nano communicating to Aruidno via a Serial connection, we had the message to be in the format of (THROTTLE, STEERING). And after switching to iPhone, we decided to use Bluetooth Low Energy (BLE) [3] as our communication scheme, but keeps our message format.

We later switched from Arduino to ESP32-Cam since they can be substituted with a 84% decrease in cost from $30 to $6. In this paper, will refer to both Arduino and ESP32 as Arduino since it is a more commonly used chip, but note that in actual implementation, ESP32 is used. A picture of both chip can be found in Figure 2.8

As demonstrated in Figure 2.9, we utilize Arduino's Servo [26] library to issue commands from the board to the ESC. 2000 represent full throttle forward, or full right in steering. 1500 represent neutral in throttle, or roughly straight in steering (in reality, the steering rod might not be attached to the center of the servo, and therefore, the reading is not accurate. The steering rod itself would require tuning to be straight); 1000 represent full throttle backward, or full left in steering.

According to the Traxxas documentation [20], the braking function is enabled if the ESC is in the training mode and will be activated when user suddenly shift from forward to backward mode. This is an un-desired behavior because the higher level algorithm is going to expect that when it issues a backward command, the vehicle will execute as is, and not having any extra steps to overwrite it. Therefore, it is critical to do so in the embedded systems to hide it from higher level algorithms. As demonstrated in Figure 2.9, the *ensure_smooth_transition* function's job is to ensure the execution of the Finite State Machine. If the vehicle transit from the Forward state to the Reverse state, the transition will involve an execution of neutral, slight reverse, and neutral in order and each of 10 ms.

2.6 Communication Pipeline

There are two main communication that happens in real time: the communication between iPhone and the Arduino; and the communication between the iPhone and the PC.

The communication between the iPhone and the Arduino is carried out via Bluetooth Low Energy. In BLE, a message is of length 20 bytes or less. Fortunately, our scheme of (THROTTLE, STEERING) is gauranteed 11 bytes long because "(", "," and ")" takes 3 bytes, and throttle and steering are 4 byte char from 1000 to 2000. Therefore 4+4+3 = 11 bytes.

It is indeed not the optimal approach in sending data – the best approach should be to send it as a concatenated 8 byte Integer. However, at this point, we had many vehicles on the ground already and the parsing of string seems to be trivial in computation time and therefore we decided to follow the idiom "if it isn't broken, don't fix it".

The Arduino code can be found here: ROAR ios arduino.

Figure 2.8: Left is ESP-32 Cam, Right is Arduino Nano BLE Sense

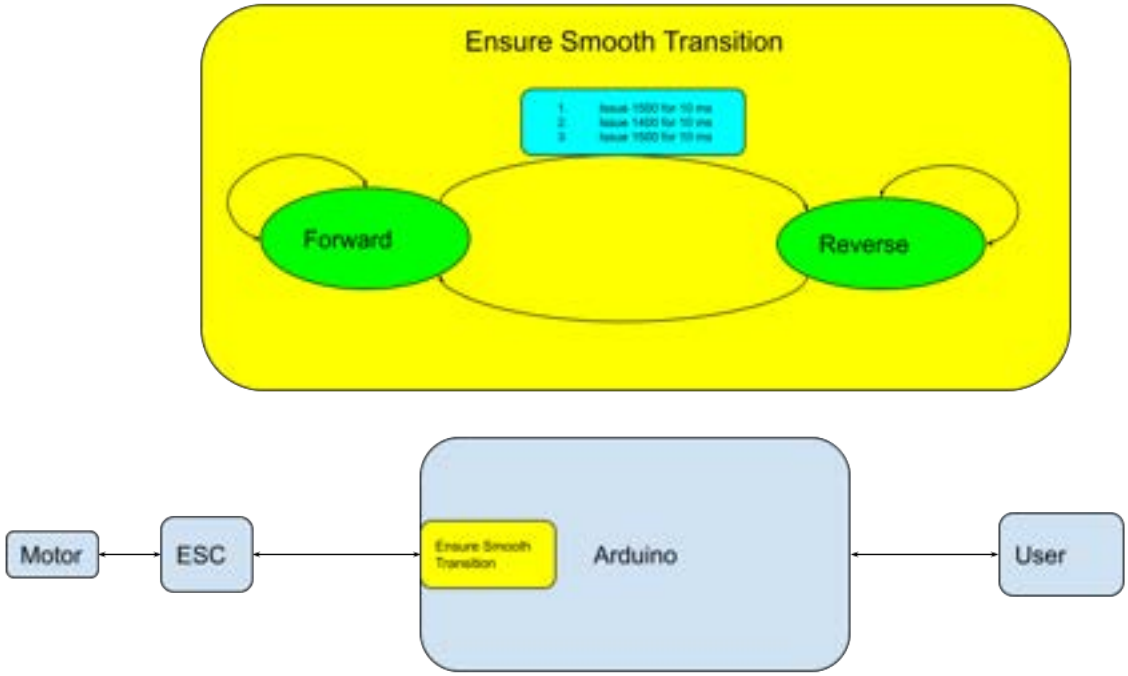

Figure 2.9: Embedded system Ensure Smooth Transition

The first version of communication between the iPhone and the PC was implemented using WebSockets [28]. WebSockets are easy to use and get started. Unfortunately, in the back end, it uses TCP communication, which guarantees delivery but not speed and therefore in reality, we often see lagged frames. And this phenomenon was not discovered during development, where signal noise was minimized, and communication was near perfect with no lost packets. However, when more students started using the application in Cory Hall, where wireless signals are very crowded, we start seeing bad behaviors.

We attempted to look into Google's WebRTC, Apple's HTTP Live Stream, or GStreamer, but unfortunately, they are either too complicated to setup, not a real time (latency ¡ 10 ms) solution, or just don't have sufficient ecosystems for us to start development.

Our conclusion is that we need to start off with the UDP protocol and design a transmission algorithm on top of that. A visual figure of the algorithm can be found in Table 2.2, 2.3, 2.4.

It is assumed that reader knows the difference between UDP and TCP and the characteristics of UDP. A good summary of the UDP protocol can be found on Wikipedia [32].

The general idea of our algorithm is that the Server (iPhone) will receive a request to send data. And the server will then chunk and send over the data with each chunk containing a header for later decoding purpose. Now there are three cases that we need to consider:

1. No packet loss 2.2
2. ACK packet is lost (The client sent request but the server did not receive it) 2.3
3. Content loss (There is a content packet lost in transmission) 2.4

For normal sequence where no packet is dropped, the client (PC) will first send an ACK to the iPhone, and wait within $T_timeout$ to hear back from the server. The message will be in the format of

```
xxxyyyzzzDATA
```

where xxx represent the current chunk counter, yyy represent the total number of chunks, and zzz is temporarily left as a placeholder, and DATA will have a maximum length of 9000 (since Mac has a default UDP packet size of 9216, if we want all systems to be compatible by default, we have to choose a lower bound).

Take RGB image as a concrete example, assume that the RGB image, along with the camera intrinsics needs to be sent over. The RGB data needs to be sliced into 10 chunks. Therefore, the number of chunks required for an RGB image will be maximum 11 (10 for sliced up RGB data, 1 for intrinsics). Then it will look like the following:

```
'|' is a delimeter for visual explanation purpose
001|011|000|INTRINSICS_DATA|RGB_DATA1 --> Chunk 1
002|011|000|RGB_DATA2 --> Chunk 2
003|011|000|RGB_DATA3 --> Chunk 3
...
011|011|000|RGB_DATA10 --> Chunk 11
```

As for the client side, we not only need to decode the content correctly, but also needs to find a good balance of $T_{timeout}$ so that we are not simply discarding all of our messages due to long waits, or are not just waiting for a message that was lost in transmission. A good $T_{timeout}$ with some experiment turned out to be 0.1 seconds. And it makes sense because it implies that the algorithm can lag for at most 10 frame per second and still be useful. This number can be further tuned if wireless environment is known to be not congested.

The code for the client can be found ROAR_iOS and the code for the iPhone server can be found ROAR_xcode_iOS

Time	A	B
1	Send Ack	
2		Recv Ack and send data 0 with header
3		data 1
4		data 2
...		...
n		data n

Table 2.2: No packet drop UDP communication

Time	A	B
1	Send Ack	
2		did not receive ACK
3	TImeout, send ACK again	
4		Recv Ack and send data 0 with header
5		data 1
6		data 2
...		...
n		data n

Table 2.3: Send Ack dropped UDP Sequence example

Time	A	B
1	Send Ack	
2		Recv Ack and send data 0 with header
3		data 2
4		
...		...
n		data n
	data drop discovered, abandon	

Table 2.4: Content Drop UDP Communication Example

2.7 Software (Python)

About Entry points

As previously discussed, one of the design philosophy is to allow easy on-boarding for new users. Therefore, all the entry points for the program are easily identified as runner_xxx.py. As of now, we support runner_sim.py and runner_ios.py and have deprecated the support for runner_jetson.py.

About Connection to environments

Here, let's define environment to be the downstream pipeline that supplies ROAR the data it needs, and takes the throttle and steering it outputs.

For each runner (or environment), there is a respective xxx_runner.py file in the submodule that supports the connection between the higher level algorithms to the actual environment. For example, the carla_runner.py in the ROAR_sim repository supports the connection of ROAR against the Carla environment. And similarly, ios_runner in the ROAR_ios repo will support the connection for iOS environment.

Every runner will follow similar format with PyGame [22] as the main driver for display and parse user inputs:

```
func setup() // setup environment & connections
func start() // run an infinite loop that drives the game
func finish() // safely tears down connections
```

About Abstractions and Hierarchy

The purpose of Abstractions and Hierarchy is to encourage researchers to focus more on the algorithm rather than the infrastructure that ties together the algorithms into the system.

Most classes in ROAR conforms to a Module where it defines

```
func init() // safely initialize the class
func run_in_series() // the step function
func shutdown() // safely tears down
func save() // called every step for saving data
```

The run_in_series function defines what should this particular module do on every step. For example, if the user were to implement an object detection algorithm, they will need to implement the run_in_series and don't need to worry about how to run it in parallel from the other modules. We also have a generic definition of Agent, where a predefined method for sensor initialization, recording and tear down is implemented so that user don't have to dig deep into the infrastructure.

For all the data types, we have a predefined data structure models implemented on top of the library Pydantic [21]. Pydantic is a great library for data validation and the usage for us

is to ensure that the data types entering ROAR is compatible across different platforms. For example, the Depth data entering the ROAR system via the simulator and the real world should be at least similar in their shape and format.

2.8 Simulation

The need for simulation rose out of the COVID pandemic, when the research group decided that it was unfeasible to build and ship everyone an individual vehicle and maintain the inventory. We wanted a simulator that can provide the same type of sensor data as the actual sensors in the world, namely, Depth data, Front and back RGB data, GPS signal, and the ability to customize track and environment such that we can test on a variety of scenarios.

Carla [10] is a simulation that satisfy all those criteria with a massive open sourced ecosystem. Furthermore, Carla has a very well documented Python API and ROS bridge that will help us to get started quickly.

CARLA has been developed by both industrial stakeholders such as Intel and Toyota and other individual contributors. Using Unreal 4 engine, the simulator can simulate multiple vehicles driving in urban environments (including programmable traffic lights and added pedestrians), and provide ground truth vehicle localization and RGB and depth camera data. The system is implemented in a client/server architecture, where multiple vehicle agents can be connected as clients to a common simulated driving environment controlled by a CARLA server. The software supports Python programming of the client algorithm and further uses pygame module to render the simulated environment, as shown in Figure 2.10

Carla uses a Server-Client model as seen in Figure 2.11 the server is the actual game, and developers can use a client to talk to the server and thus controls the actors in the server. Carla is built on Unreal Engine, which is renown for its ability to produce photo-realistic rendering, computation of physics and much more. A more complete description of Carla can be found on their official documentation.

After looking through some example files in Carla, we decided to fit it all into the ROAR Infrastructure model by creating a Simulation submodule. The specific implementation will be left for the reader to explore in the code, here is an overview of what the submodule contains:

1. carla_runner.py that allows ROAR to interact with Carla
2. *.egg files that represents the compiled Python API corresponding to each Server release
3. camera_manager.py that manages the different cameras
4. world.py that manages the interaction with the Carla World

Figure 2.10: A screenshot of the CARLA simulator environment. The red car is controlled by a ROAR Python agent. Other vehicles can also be added to follow certain pre-programmed waypoints that share the same road. The simulator can also generate common vehicle sensor data, which is not shown here.

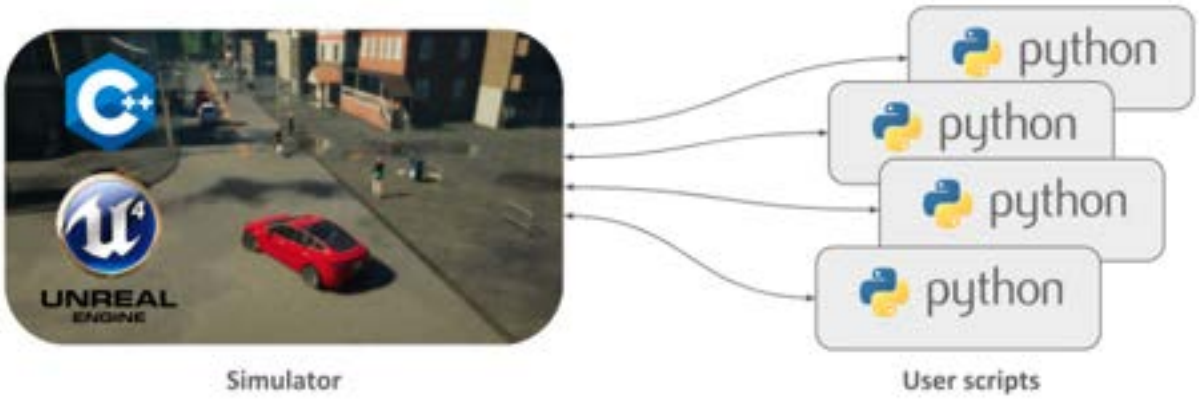

Figure 2.11: Carla Server and Client model

Figure 2.12: Berkeley Minor in Carla

5. sensors.py that creates definites for sensors
6. hud.py that manages the display shown to user

With this set of API, we are able to stream the data from Carla Server, conform it to what ROAR expect and use the simulation as our alternative to actual vehicles.

Map making in Carla

Besides setting up the connection, we also attempted to create our own map in Carla. Figure 2.12 showcase Berkeley Minor that we constructed in Carla.

We took 4 steps in generating this map:

1. Go to Google Map to find out the rough outline of the map
2. Use RoadRunner to import the map in OpenStreetMap format
3. Create the building assets in 3D modeling software such as Maya or TinkerCAD
4. Import the xodr file generated from RoadRunner into Carla and drag and drop the 3D assets onto the map.

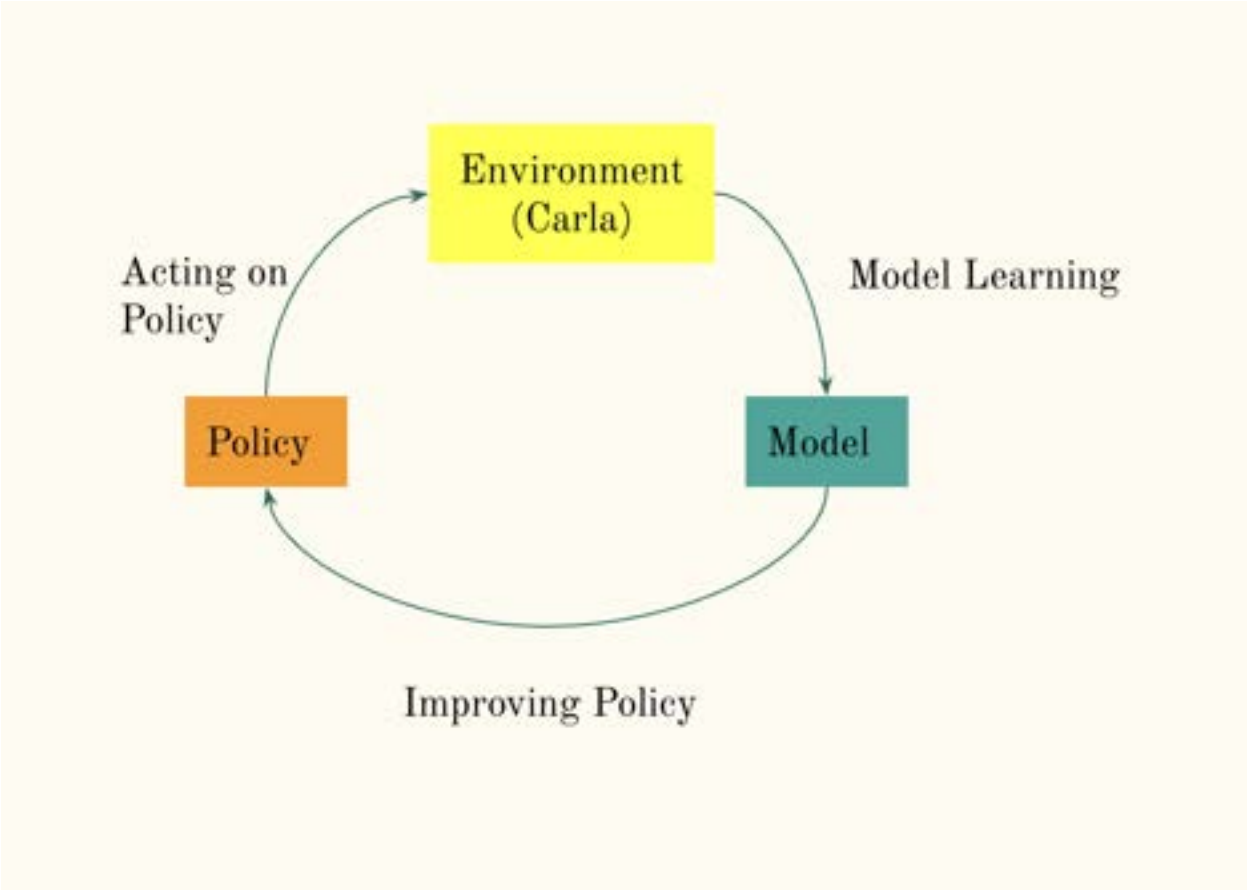

Figure 2.13: Model Based RL Pipeline

This method of generating map does not give Carla the information of semantic segmentation and therefore of course in the downstream pipeline, ground truth semantic segmentation will return meaningless data. However, sensors such as depth sensor, Lidar, etc still works as intended. We found that for racing purpose, we don't really need to do fine grained semantic segmentation and therefore this did not yield significant roadblock for us.

2.9 ROAR Gym Integration

OpenAI Gym [5] is a framwork that supports a variety of Reinforcement Learning algorithms. In order to support continuous development in a common infrastructure for RL, we decided to implement a bridge between ROAR and the OpenAI Gym environment.

In essence, a gym environment is a framework that outlines the neccesary steps for reinforcement learning. Figure 2.13 demonstrate a model based RL pipeline. The role that

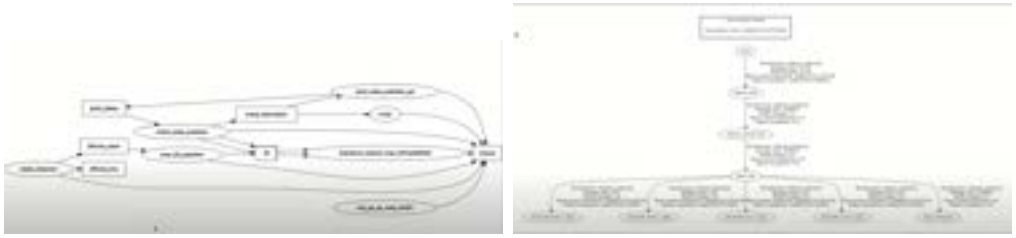

Figure 2.14: ROAR ROS2 transform hierarchy and message pipeline

the OpenAI Gym play in this particular diagram is to define pipeline for model and policy and upstream user can implement the exact data types of the input and output.

Below are some functions that user can choose to implement

```
def __init__() // initialize the Gym environment
def reset() // function called when reset of the environment is triggered
def step(action) // function called when stepping in the environment
def is_terminate() // function called to decide whether an episode has ended
```

The ROAR_Gym package interact with ROAR_Sim to help user define a set framework for re-initializing the Carla environment and fetching the correct output from the simulation. A more detailed example of RL based PID control and a model based end-to-end agent will be discussed in later chapter.

2.10 ROS2 Integration

As our system expand, we found that the Python Global Interpreter Lock (GIL) [9] really limit what we can do.

For example, out algorithm for ground plane detection will work at a maximum of 10 FPS, however, the will impact our down stream controller to run slower as well. This causes significant problem in the parallelism of the system despite having Multi Threading implemented.

Furthermore, the need for more advanced and ready-to-use solutions for planning is also on the rise. ROS2's Nav2 [18] package served as a great example of the package that we wanted to try.

Figure 2.14 and 2.15 demonstrate our setup of the ROAR vehicle. The link between iPhone and all other components on the vehicle such as the wheels, base, and iPhone mount are all statically defined. The transform sent from the iPhone will be used to define the transform between the odometry frame and the iPhone Link frame.

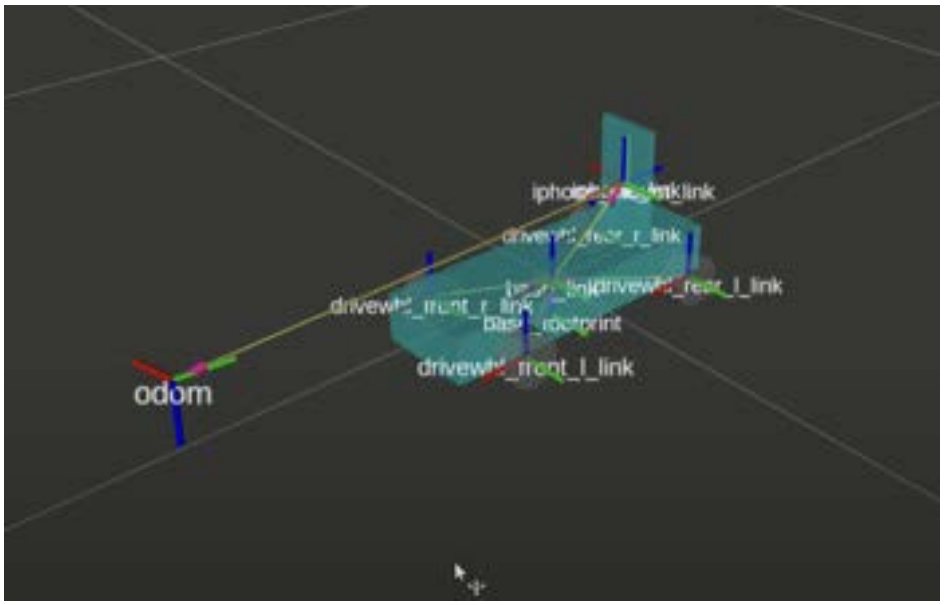

Figure 2.15: ROS2 Robot Model Display

2.11 Virtual Reality Integration

Virtual Reality can provide a safe yet low-cost alternative for testing autonomous driving features and customer experience. VR based test platforms can reduce cost, accelerate development, and mitigate the risk of testing autonomous vehicles on the road [36]. VR allows the users to drive the vehicle and supervise the autopilot in a safe manner, and the immersive environment can engage drivers deeply.

Our system is based on the Unity platform using Oculus Rift S as our headset. Our software contain three major modules: control, display, and data streaming. An example is shown in Figure 2.16 and also a video of the demo can be found in youtube.

Figure 2.17 demonstrate a overview of how all the components tie together

The requirement for data streaming is that it has to be low latency. We decided that since we still wanted to use Python as our decision making pipeline, and use Unity simply as a display, we are going to run a UDP server on Python to stream out the data that it received from iPhone. But at the same time, this pipeline can also directly skip Python to have the Unity to directly control the vehicle as well. Therefore, we ensured that Python will encode information the exact same way that the iPhone encodes it. The code for Unity can be found at this link. We are able to achieve over 25 FPS, and that does satisfy our requirement for real-time, low-latency streaming service.

The display and control modules in our system take image and vehicle status as input and gather user control as output. The image streamer receives encoded RGB images and then decodes them into original images. The image then is converted to texture in the virtual

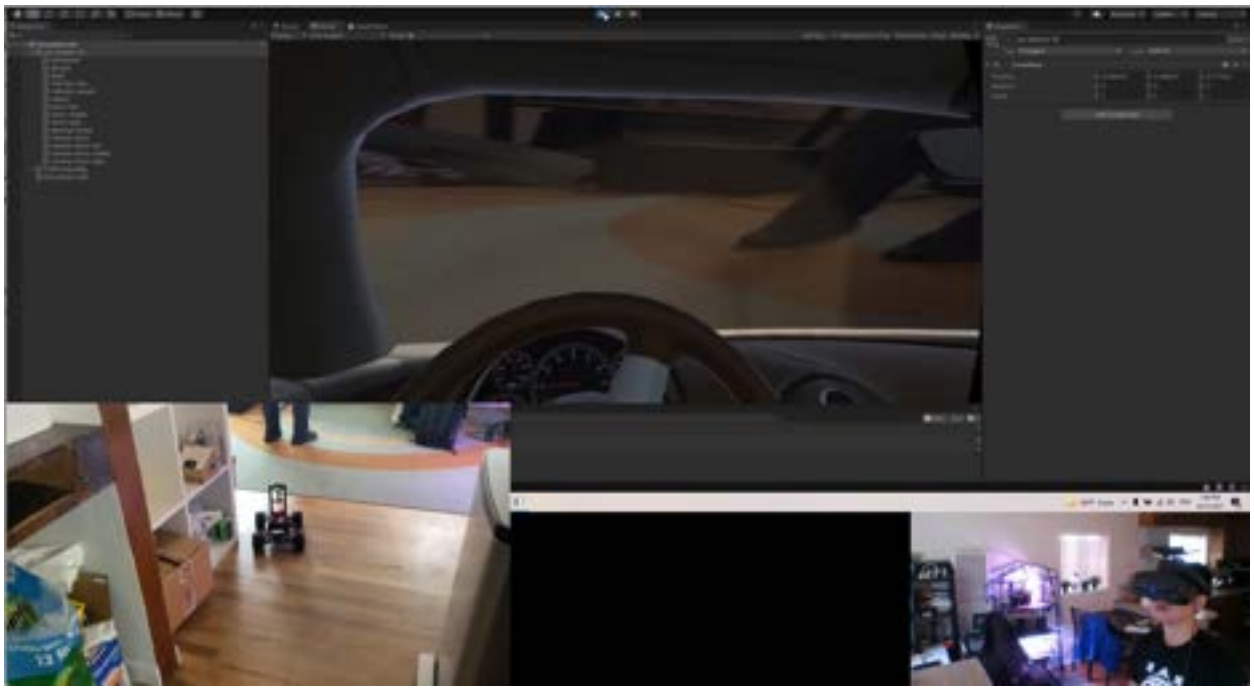

Figure 2.16: ROAR VR Demonstration. Also available on Youtube

3D environment. The up-to-date driving environment will be displayed in the Oculus Rift headset for the user. The control module takes input from the VR controller and maps them into control signals, throttle and steering. The throttle is determined by the trigger on the controller and has a larger magnitude when the trigger is pulled tighter. The steering can be controlled by the joystick and rotation of the controller. Using the rotation of the controller provides more freedom to precisely control steering.

In our virtual driving environment, users can control the vehicles manually or supervise the autopilot, which is semi-autonomous driving. Users sit in a simulated vehicle with realistic car interiors. The steering wheel represents the control steering and the dashboard with indicators shows the speed. When the car is moving, the car itself maintains stability and only the outside environment, the streaming RGB image, is changing. By this manner, our VR system can reduce discomfort associated with intensive locomotion.

The VR Integration is part of the Berkeley CS 294 Augmented Reality and Virtual Reality final course project [7]. The final paper can be found on this link and we also have a website on behance

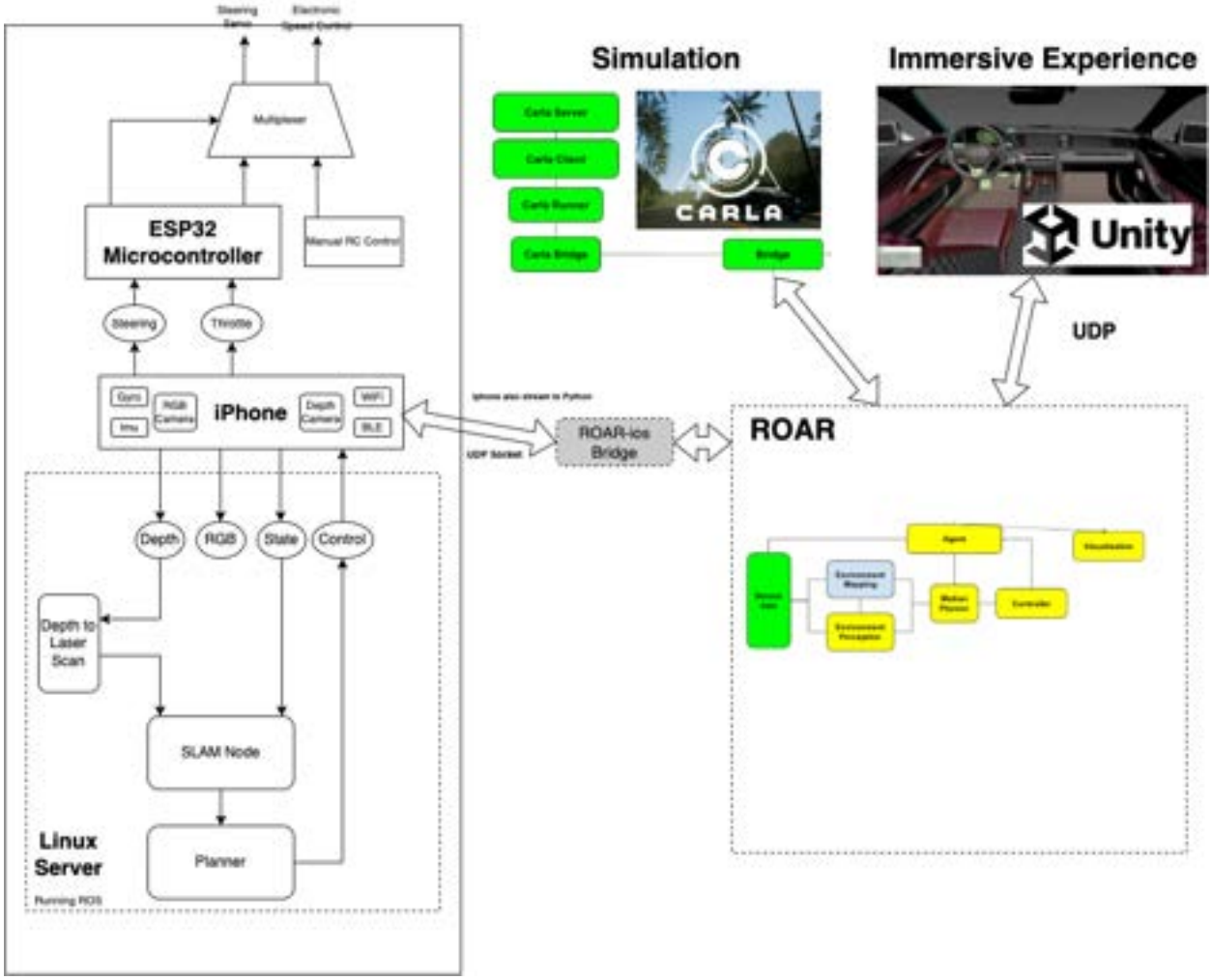

Figure 2.17: System Diagram for ROAR with Virtual Reality integration

2.12 Miscellaneous Discussions

Visual Odometry SLAM accuracy and precision on iOS

Visual Inertia SLAM, as the name suggest, is to combine the use of visual elements, or RGB image, and Odometry information, or IMU, to produce a localization estimate.

For our purpose, we needed the localization to be precise, but not necessarily accurate. The reason behind is that the data can deviate from actual ground truth, but between each run, they should deviate with the same amount. For example, if all data are drifted by 5 meters, that is fine. But if Trial 1 drift by 5 meters, trial 2 by 2 meters, then there will be a problem.

We decided to take advantage of Apple's AR Kit's World Tracking feature to help us achieve Visual Odometry SLAM. Apple's Understanding World Tracking [31] has a great description of the feature. We have found that although this is a great technique in most scenarios, it does not always work.

This technique does not work outdoors, especially where there is direct sunlight exposure, uneven ground, and non-distinguishable features. We have tested in an outdoor parking lot. The result was discouraging – the ARKit returned us data that are very noisy and does not look like the track that we intended to drive on. And looking from the camera's perspective, due to the asphalt ground, we see that the phone vibrates a lot, which probably introduced extensive noise into the IMU. Furthermore, the featureless scene and the differences in sun exposure further deteriorate the performance of such system. Therefore, for small outdoor space, an outside-in localization system such as RTK GPS localization system is still preferred over the inside-out SLAM.

When we moved indoor, the amount of features and the ability for loop closure still play an important role in obtaining accurate localization data. We have done an extensive experiment to understand the kind of environment we need.

We divided SLAM into distinct two parts – Calibration and Localization. In the Calibration phase, we drove the vehicle 4 laps around the same track, attempting the have the iPhone "remember" the passive features of the track and thus build a map. Then, we will do 3 more runs, recording the x, y coordinates in which the iPhone reported, and plot it out.

For our controls group, we will have a scene where there is only one active Anchor point – the world center. In the testing group, we will place active Anchor points at strategic locations, such as places where turns, up and down ramp, or other movement that can unstablize the iPhone.

Figure 2.18 demonstrate the result of our controls group with only single image anchor. It is clear to see that although the general shape of the trajectories are similar, their precision is very low. And also, as time goes on (the vehicle start at roughly (1,0) on the right hand side, and then goes down toward (1,-8) in the lower right hand side, the deviation becomes more severe. There are sometimes drops in data, and that might be caused by both jumps in actual localization output or due to UDP transmission losses. These are all expected – as more vibration and noise are accumulated without seeing another calibration marker, the

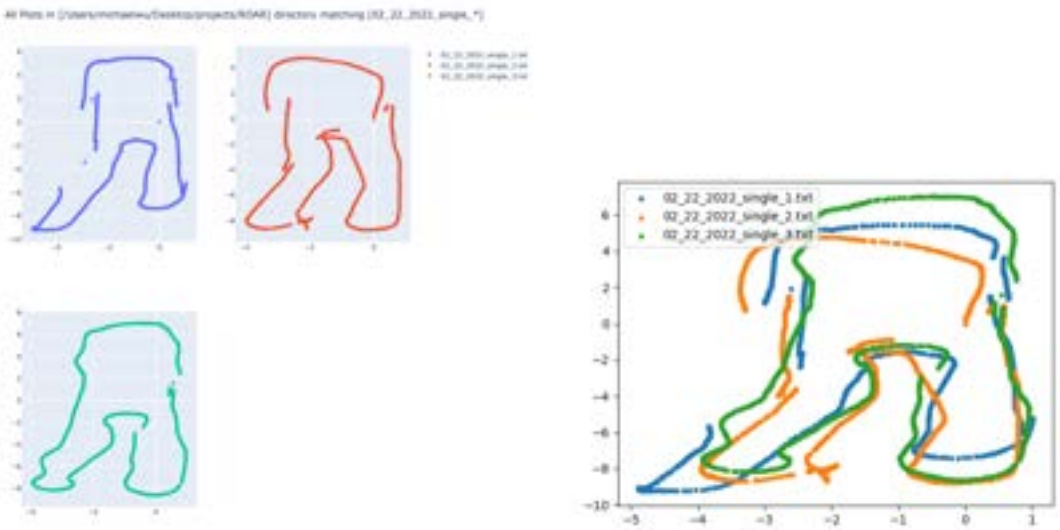

Figure 2.18: Single Image Anchor. Left is individual run; Right is three run overlapped. Clear deviation as time goes on and low precision between the runs.

Visual Odometry SLAM is doomed to experience deviation.

Figure 2.19 demonstrate the result of our testing group with multiple image anchors placed at strategic locations. It is clear to see that the precision increased dramatically although there are still some minor differences, and also as time goes on, the effect of drifting starts to appear. We attribute some of the drifting due to the fact that the vehicles were driven by hand and that there might be manual error introduced in the trajectory. Furthermore, we found that the more laps we drive, the more accurate it is. This might be because when we are driving, the image is not that stable, and by driving lots of time, ARKit is able to build a better represenation of the world using feature points collected from each trial. And we can see that happening because on Lap 1, when the vehicle has driven a lap in the real world, in the localization data plot, we can see that it was still pretty far away from the starting point – an effect of drifting. But as we drive more in the same scenario, in the localized data plot, the vehicle begin to close the loop.

From comparing the result of this experiment, we can conclude the following:

- Multiple image anchors can enhance localization accuracy, but ONLY of calibration is done right
- Image anchors must be placed at strategic locations, such as turns and up or down ramp, and must be clearly visible.
- Calibration is successful if the world center AR marker sticks in the origin when driven a lap around.

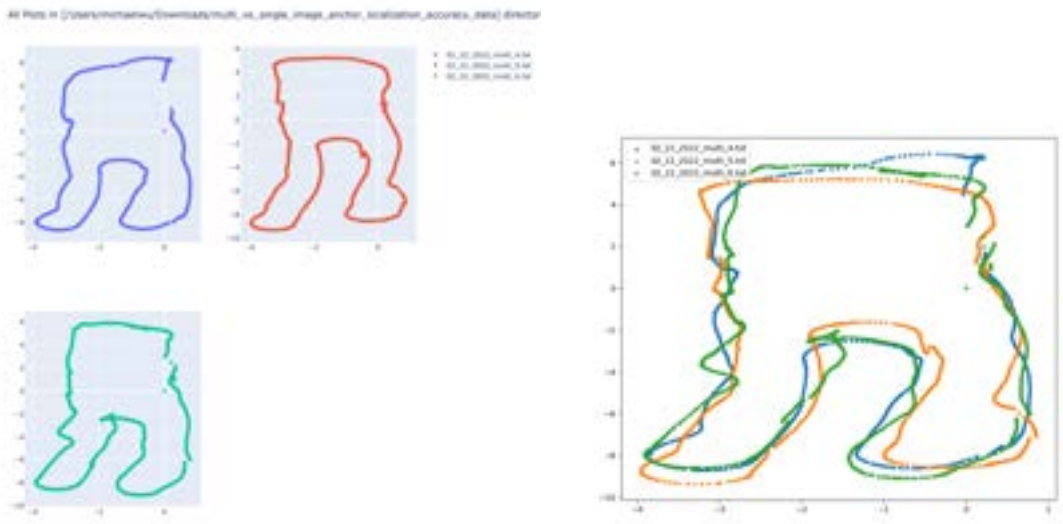

Figure 2.19: Multi Image Anchor. Left is individual run; Right is three run overlapped. Deviation is lower, precision is higher.

- Visual Inertia SLAM suffer from drifting IMU and blurry image readings, therefore we need to stabilize iPhone at all cost and have a very robust calibration map.

Chapter 3

Autonomous Driving Algorithms

My work during my 5th year master was mostly on infrastructure. However, I have also touched upon some interesting autonomous algorithms. Here are a list of them that I wanted to talk deeper about.

3.1 Perception

DeepSORT

DeepSORT, proposed by Wojke et. al [35] is an improvement upon Simple Online and Realtime Tracking (SORT) algorithm featuring using a Deep Association Metric.

To understand the problem, we need to understand what are the traditional methods' pitfall, and how did DeepSORT solve it.

Mean shift and Optical flow are two traditional methods for object tracking.

Mean shift operate using a similarity function to figure out the best chunk of pixels or features in the next frame. For example, in Frame 1, the detector found an object of interest, the in Fame 2, we will start searching in a bigger area of interest, and use that similarity function to compute the smallest bounding area that minimize our similarity function. Unfortunately, this method, although simple, has the pitfall that it is potentially computationally intensive and will lose tracking during times of occlusion.

Optical Flow is another technique that looks at the motion of features due to the relative motion across frames. If the object of interest is moving in a certain constant velocity, then we will be able to track, even predict the object in the next frame. However, the major downside is that this method is prone to noise, and the motion of the object has to stay relatively stable.

Simple Online Sort Realtime Tracking (SORT) is an algorithm that can mitigate issues mentioned above. SORT operate in four steps:

1. Detection
2. Estimation

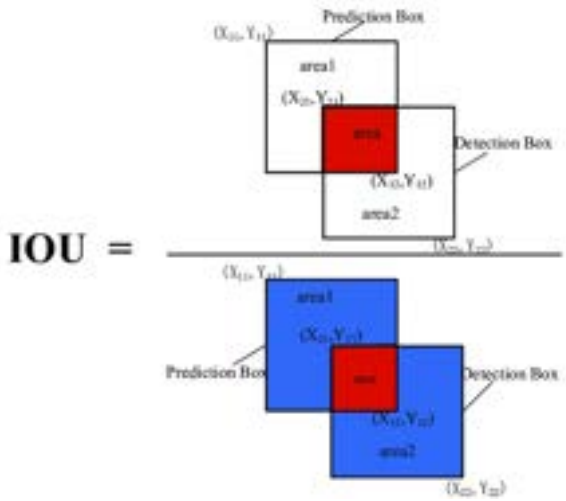

Figure 3.1: IOU Visual Diagram

3. Association

4. Track identity creation and destruction

For detection, there are many options to choose from. Neural nets such as Yolo, FRCNN, and many more can produce great results. However, the quality of detection plays a big role in tracking performance. And the output of this module is bounding boxes and confidence ratings.

For the estimation step, we need to propagate the detections from the current frame to the next using a linear constant velocity model with Kalman Filter. If a detection is associated to a target, then its velocity model will be updated via the Kalman Filter framework. However, if no detection is associated to the target, it is simply predicted without correction to its linear model. We would assign detections to existing targets by using a cost matrix computed using intersection-over-union (IOU) distance. The assignment is solved optimally using the Hungarian Algorithm [25]. Figure 3.1 demonstrate visually what an IOU is.

Now lastly, Track Identity Life Cycle is a process where we assign and destroy each identity. We consider any detection with IOU greater than IOU_{min} to trigger the existence of an untracked object. The tracker is initialized with velocity covariance being infinity since at frame 0, the tracker just initialized and does not have an velocity estimate. Furthermore, the new tracker will undergo a probation period $tracker_init_time$ where target associate with detection to accumulate enough evidence that tracking a false positive. For destruction, if a tracker is lost for T_{lost} frames, we deem it lost and destroy the identity.

The Deep association metrics aims to resolve the issue that SORT still experiences problems during times of occlusion and different view points. We will build an Appearance Description Vector by training a classifier until it reached good result and strip away the

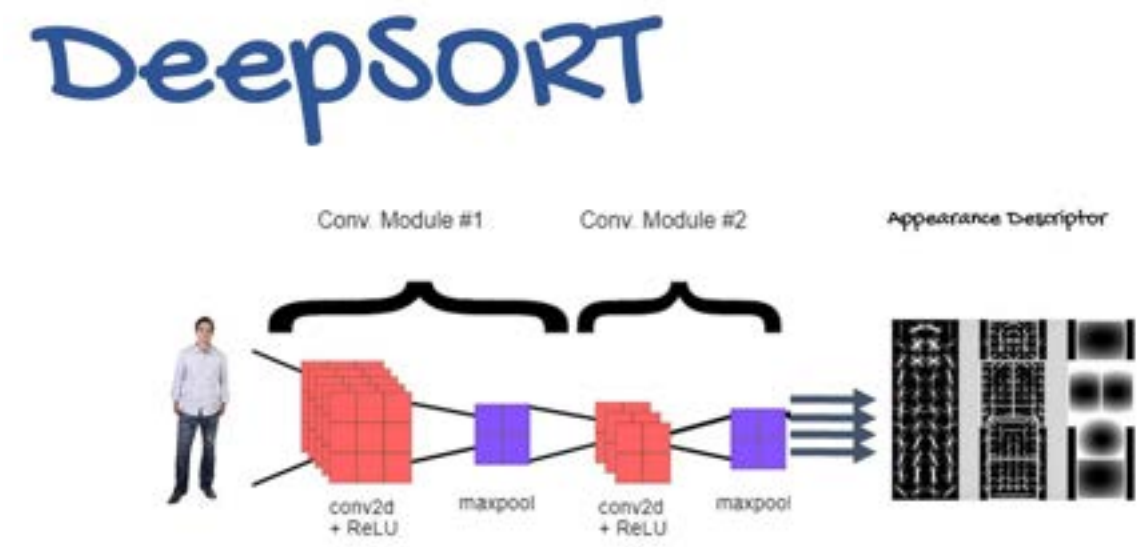

Figure 3.2: Appearance Descriptor Vector

final classification layer. The dense layer will produce a single feature vector. Figure 3.2 demonstrate the Appearance Descriptor Vector. Now, we use Mahalanobis distance as opposed to Euclidean distance for Measurement-to-track association (MTA) for determining the relation between a measurement and an existing track.

Figure 3.3 demonstrate the end result. The code is available here.

Traffic Cone Detection

Obstacle Detection Via Ground Plane Detection

For a racing environment, we found that most of the view is going to be ground, with only occasionally other vehicles in-sight.

The pipeline that we designed is to transform a depth image into a point cloud and then find all the ground points in the point cloud and project it back onto a free space map. The inverse of that free map would be simply the "occupancy map". To simply the language, we will be referring to the free space map interchangeably with the occupancy map. We utilized the package OpenCV [4] and Open3D [38] to help us to process images and point cloud data efficiently.

The equation $s * [u, v]^T = K@[R, T]@[x, y, z]$ states the relationship of coordinate trans-

Figure 3.3: DeepSort Example. Full video available here

formation from depth images to world coordinate. s represent the scale, or the depth reading at pixel u, v. R, T represent the Rotation matrix and the Translation vector from the camera coordinate to the world coordinate, respectively. Krepresent the intrinsics matrices of the camera, which is being streamed over with each depth image. X, Y, Z are the coordinates of points in the world frame of reference. Unfortunately, the readings for R and T are not that reliable – we suspect it is due to SLAM's drifting and vehicle's significant vibration leading to unstable sensor reading – causing the data transformed into the world coordinate system to be unstable. Therefore, we decided that we are going to do use the data in the camera frame of reference, thus the equation simply becomes $s * [u, v]^T = K@[x, y, z]$

We are interested in the x and z coordinates. However, notice that x is in the range of [-DEPTH, DEPTH]and z is in the range of (0, DEPTH]. Therefore, we need to do an affine transformation of Coordoccu = SCALE * Coordcamera + SHIFT where we define SCALE and SHIFTaccording to the desired occupancy map size and maximum depth. The resulting image can be seen in Figure 3.4

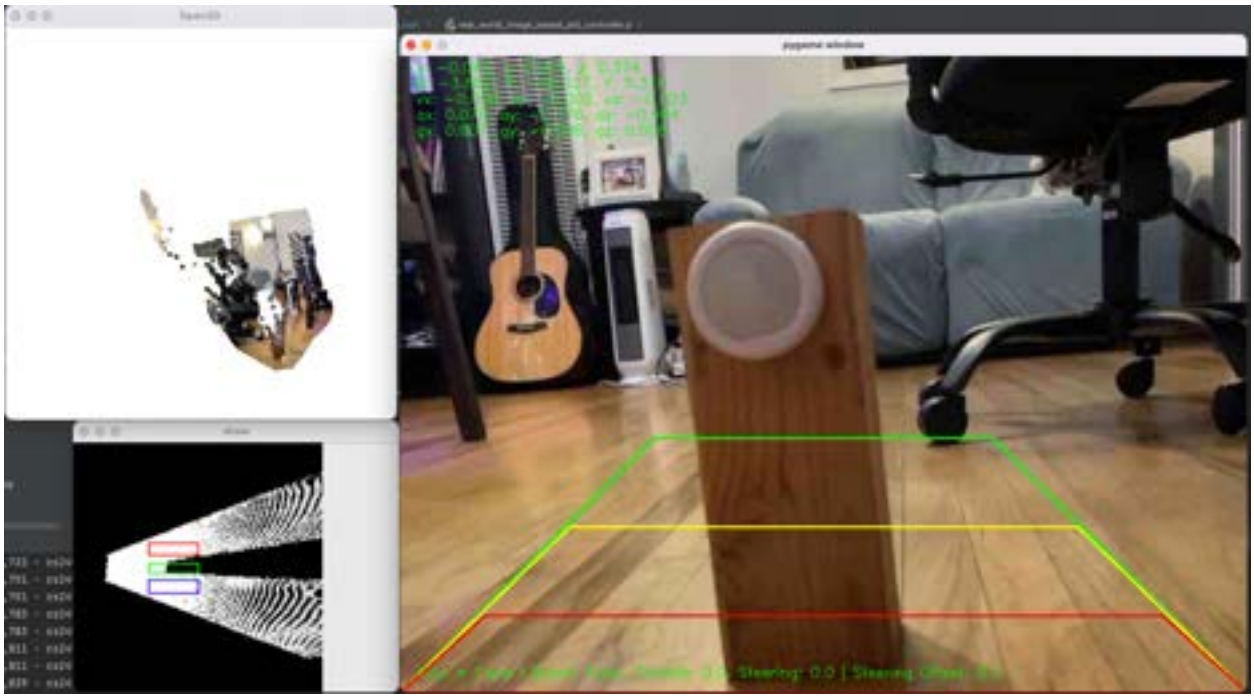

Figure 3.4: Obstacle detection in the camera frame with LiDAR sensor. On the upper left is Point Cloud, on the lower left is ground plane detection with three regions of interest

3.2 Planning

Waypoint Following

Figure 3.5 provides a visualization of the planning waypoint following plannint algorithm.

It will be assumed that this module will receive a list of waypoints at the start, and will keep receiving updates for vehicle position. Furthermore, the list of waypoint will be ordered, meaning the it will be A, B, C, D in sequence of contact during collection.

In classical waypoint following methods, in order to mitigate the issue brought forward by messy GPS readings, engineers often calculate an average of the next n waypoints to account for jitterness in the short term movements.

For ROAR, we found it more effective to revise this algorithm and base it off of distance where r is a radius of next waypoints that we are going to skip. For example, point A in Figure 3.5 is going to skipped, and the first point outside of searching radius point B is going to be set as our target waypoint to follow in our controller.

Here is the pseudo code for the algorithm. The actual code can be found in the LoopSimpleWaypointFollowingLocalPlanner class

There are two values for users to adjust:

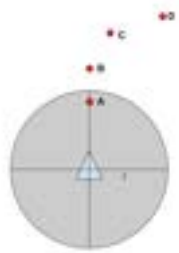

Figure 3.5: Waypoint Following Example. r = searching radius. Red dots are waypoints. Triangle is ego vehicle

- NUM_TO_SKIP = threshold value for number of waypoints to skip in the beginning. This is to prevent the vehicle from turning in circle for trying to follow a point right on top of it self.
- CLOSENESS_THRESHOLD = the search radius r for how close the next waypoint should be.

```
func find_next_waypoint():
    closest_dist = inf
    while True:
        if current_waypoint_index == len(waypoints_queue):
            # loop through from the start
            current_waypoint_index = 0 + NUM_TO_SKIP
        curr_dist = find_dist(waypoints_queue[current_waypoint_index]

        if curr_dist < closest_dist:
            # enter on first iteration only
            curr_closest_dist = curr_dist
        else if curr_dist < CLOSENESS_THRESHOLD:
            # skip close waypoints
            current_waypoint_index += 1
        else:
            # found waypoint that is far enough away
            break
    target_waypoint = waypoints_queue[current_waypoint_index]
    return target_waypoint
```

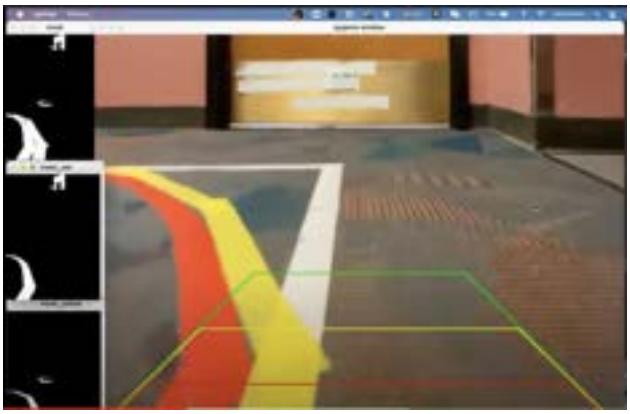

Figure 3.6: Lane Mask Example. Full video can be found at this link

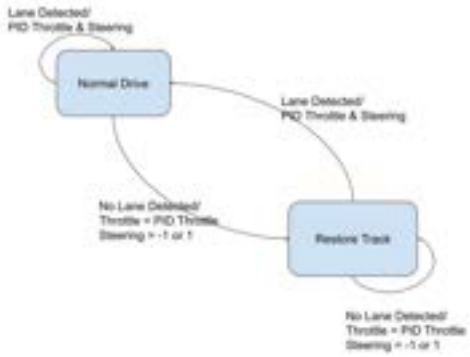

Figure 3.7: Lane Following Finite State Machine

Lane Following

Lane Following is implemented in a Finite State Machine format. The input are RGB Image for lane segmentation and to produce steering, and vehicle velocity for throttle control.

For detecting the lane, we used OpenCV's thresholding function and applied a erode and dilate mask for further refinement of the threshold mask. Furthermore, in certain scenarios with significant light reflections that distort RGB colors, we also lay down two colored lanes to ensure that we can reliably detect the lane. A video can be found at this link.

From obtaining a mask, we now want to compute the cost of deviation and translate that to a steering command. It is not hard to imagine that the optimal line to follow should be at the dead center of the image. And deviation on either the right or the left means that

Figure 3.8: Vehicle Platooning using ARUCO Marker. Full video can be found here

the vehicle has deviated to either direction. To increase efficiency of the algorithm, we took 3 samples – one close, one mid distance, and one far. We scan, from the starting line, where did we find the lane first on the masked image, and lastly, we take the priority in using the pixel difference produced by the longest distance. Then we feed this error number into a PID controller to find the steering output. The code can be accessed here.

However, it is very likely that at some point, the vehicle is going to miss the lane – either because there is a sharp turn (for example, a 180 degree turn) or simply because the underlying controller is not well tuned such that the PID steering does not perform well. Figure 3.7 demonstrate the finite state diagram transitions

Platooning

One of the core issue in Platooning is communication. One way of communicating is by attaching ARUCO markers on the rear end of the vehicle so that the follower can see how much apart are the two vehicles.

A demonstration video can be found in Figure 3.8.

But there are significant limitations with this method:

1. If the two vehicles are too far apart, ARUCO marker will not be visible
2. On tight turns, the lead vehicle will turn a lot, causing the follower to lose sight of the ARUCO easily
3. At high speed where the RGB image is unclear, ARUCO detection might be problematic.

Therefore, we need a Vehicle to Vehicle (V2V) [33] communication method. We decided to use UDP Multicast to simulate the V2V communication.

Multicast is a communication technique where data transmission is addressed to a group of destination computers simultaneously. Multicast can be one-to-many or many-to-many in form factor. In our specific application, we have enabled a many-to-many communication despite having only 2 vehicles on hand to test with.

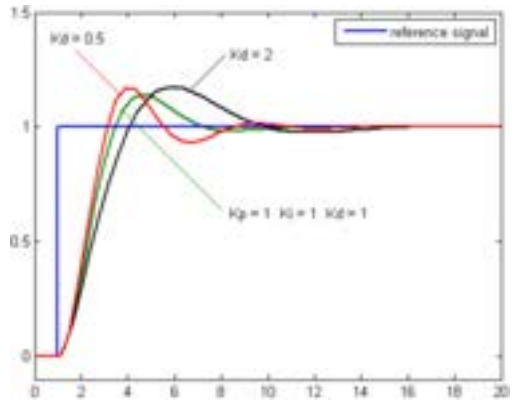

Figure 3.9: PID Controller Output example

With this method, the two vehicles can now transmit their locations to each other.

Our algorithm for platooning is relatively simple: The follower will simply move to a certain distance D away from the lead vehicle and stop. It does not consider the vehicle dynamics or any path planning components. However, this simple algorithm will demonstrate the simple vehicle platooning scheme. A demonstration video can be found in Figure 3.12

3.3 Control

PID Controller

Proportional, Integral, and Derivative Controller is one of the most classical and widely adapted controller for its simplicity.

To understand PID control, I think the easiest way is to look at its formula, and apply it into a real example.

The PID Equation is as follows:

$$u(t) = K_p e(t) + K_i \int e(t)\, dx + K_e \frac{de}{dt} \tag{3.1}$$

Where $e(t)$ defines the error, K_p, K_i, and K_e represent three gains. Put it into the context of longitudinal controller, we can see that we would like to use the PID controller to adjust our speed. For example, if our target speed (V_{target}) is 1 m/s. We can find out the error by simply finding out $e(t) = V(t) - V_{target}$

And if we define some good K_p, K_i, and K_e after experimentation and plug it back into the equation, we should get a plot that roughly looks like Figure 3.9.

PID can also be allied to steering. We can simply define the error as the angular difference between the actual heading and the desired heading. Namely the following code snippet demonstrate the purpose:

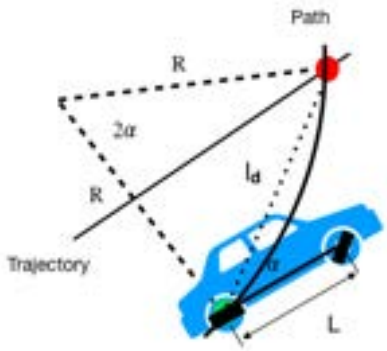

Figure 3.10: Pure Pursuit Geometric Relationship Diagram

```
v_begin = self.transform.location.to_array()
direction_vector = [-sin(yaw), 0, -np.cos(yaw)]
v_end = v_begin + direction_vector
v_vec = [(v_end[0] - v_begin[0]), 0, (v_end[2] - v_begin[2])]

# calculate error projection
w_vec =
    [
        next_waypoint.x - v_begin[0],
        0,
        next_waypoint.z - v_begin[2],
    ]
)

v_vec_normed = v_vec / np.linalg.norm(v_vec)
w_vec_normed = w_vec / np.linalg.norm(w_vec)
error = np.arccos(v_vec_normed @ w_vec_normed.T)
_cross = np.cross(v_vec_normed, w_vec_normed)

if _cross[1] > 0:
    error *= -1
```

The code for PID can be found here and the video link of PID working on a ROAR vehicle can be found here.

Pure Pursuit Controller

Pure Pursuit Controller, as explained by Coulter in his paper Implementation of Pure Pursuit Path Tracking Algorithm [13] using only the geometry of the vehicle kinematics and the reference path. Ding Yan [29] also did a great job in explaining it in a visual way. Using a fixed distance look-ahead point, we can find steering angle δ by the following equation:

$$\delta = arctan(\frac{2Lsin(\alpha)}{l_d}) \tag{3.2}$$

where L is the distance between the wheel base, α is the current heading, and l_d is the distance to the look ahead waypoint. Figure 3.10 showcase the geometric relationship.

Now as vehicle dynamics is different under different speed, we would want to introduce a varied look ahead distance as speed changes, therefore, the equation would be modified to be:

$$\delta = arctan(\frac{2Lsin(\alpha)}{K_{dd} * V}) \tag{3.3}$$

where K_{dd} is the gain for user to tune for the term velocity(V)

Brake

Traxxas 4-tec, the standard vehicle we use, does not have physical brake. According to Traxxas documentation [20], the braking function is enabled if the ESC is in the training mode and will be activated when user suddenly shift from forward to backward mode. From an autonomous driving perspective, having brakes does not equate to sudden shift from forward to backward mode, it would be very confusing to the higher level algorithm engineer. Therefore, as indicated in Chapter 2, the decision was to overwrite the ESC's default braking function in the embedded systems level, and we have to create customized algorithm at higher level for faking the braking effect.

The method we decided to pursue is outlined as below.

```
Keep a queue of the past N frames of the throttle executed
When Brake is issued
    # if we are going at low speed, no need to brake, friction will do the trick
    if current_speed < SPEED_THRESHOLD:
        return 0

    # this must be a positive number since we are using speed as our error
    output_throttle = Run a PID on the current speed reading as error
    if the sum of the past N frames of throttle is positive:
        output_throttle = -1 * output_throttle
```

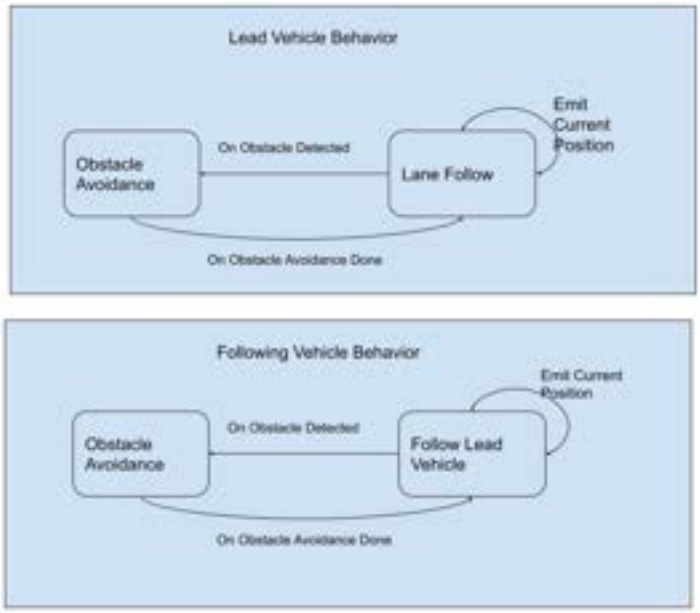

Figure 3.11: Finite State Machine for Lead vehicle and follower

```
return output_throttle
```

Surprisingly, this simple logic worked quite well in reality. A Youtube video demonstrating different level of PID gains is available here.

System Integration on Lane Following, Obstacle Avoidance and Platooning

Previous sections described the implementation of the Lane Following, Obstacle Avoidance, and Platooning functionalities individually. This section's goal is to use a Finite State Machine to connect these individual functions together. A complete demo video cam be found in Figure 3.12

System FSM can be found in Figure 3.11

Code can be found at this github

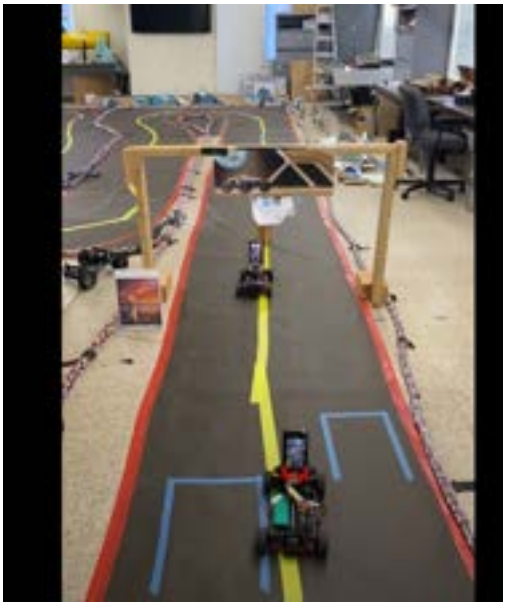

Figure 3.12: System Integration of Lane Following, Obstacle Avoidance, and Platooning. The Lead vehicle is doing Lane Following and obstacle Avoidance. The following vehicle is doing obstacle avoidance and following the lead vehicle. Complete video available in demonstration video

Chapter 4

Appendix

4.1 ROAR Junior

The purpose of ROAR Junior is to even lower the barrier of entry into the autonomous driving world by providing a solution that has a overall cost of under $100 dollars. Furthermore, this platform will also have the capability for expansion so that complex maneuvers and algorithms can be realized via higher level algorithms and add-on hardwares. The end goal is to teach algorithms and principles in a low cost manner and having a flexible platform that can adjust to the different requirements.

ROAR Junior Overall Design

As demonstrated in Figure 4.1 ROAR Junior consist of a 3D printed chassis, ESP32 CAM, ultrasonic sensor, two 9V batteries, L298N motor controller, and four DC motors with wheels. The design principle for the chassis is one that is simple yet extensible in layers. For ex-

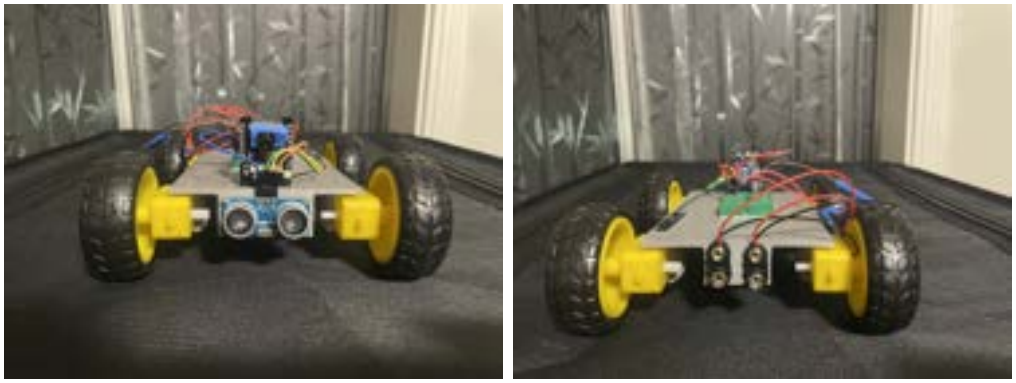

Figure 4.1: ROAR Junior

Name	num	Total Price	num_per_pack	Individual Price
3D printed Chassis	1	$2.00	1	$2.00
L298N + motor + wheels	1	$15.00	1	$15.00
ESP32 Cam	1	$36.00	6	$6.00
9V battery	2	$20.00	2	$20.00
9V battery plug	2	$6.00	10	$1.20
ultrasonic sensor	1	$15.00	5	$3.00
small breadboard	1	$12.00	12	$1.00
Total				$48.20

Table 4.1: Bill of Material (BOM) of the ROAR Junior base layer

ample, user can choose to mount a second layer for installation of robotic arm without any interference of the lower level, or user can choose to mount a bulldozer shovel in the front for a game of obstacle removal. The program burnt on the chip is designed to be as simple as possible so that the complex logic would go at user's computer. This way, we not only ensure that user can use a more friendly language such as Python to write higher level logic, but also to have a method of remotely controlling the vehicle for faster debugging sessions.

Hardware

One of the key requirement is that the robot has to be of low cost. Table 4.1 list out the price of each component at the time of writing this paper. The price is under $50 dollars if user simply want to experience with the basic, yet there's a lot that can be taught already:

- How does DC motor work
- How does ultrasonic sensor work
- How to process RGB image feeds
- How to implement a Bang Bang controller

For more advanced learners, tasks such as visual SLAM is possible because the ESP32 is not the main computing unit, client side's computer can take the heavy lift of running a visual SLAM algorithm for localization.

Embedded Systems

The core functionality of the embedded ESP32 Chip is to

1. Drive the vehicle by sending correct ENA, IN1,IN2,IN3, IN4, ENB signals

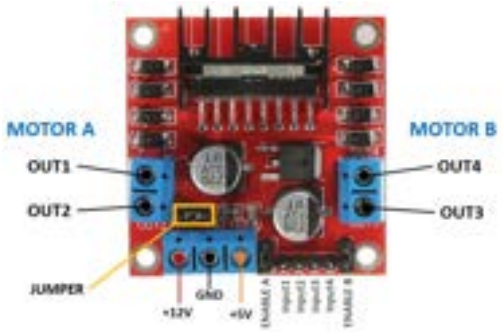

Figure 4.2: L298N motor driver

2. Asynchronously receive and process HTTP_GET command for vehicle control
3. Asynchronously manage camera captures and data stream
4. Asynchronously poll the Ultrasonic distance sensor

As Figure 4.2 shows, the input signals to the L298N motor are ENA, IN1,IN2, IN3, IN4, and ENB.

For ROAR Junior, users would connect the two motors on the left to OUT1 and OUT2, and the two on the right to OUT3 and OUT4. User would supply power to the motor driving unit by placing 9V battery's positive into +12V and Ground into GND. Although it is possible to use the regulated 5V output for the ESP, it is not recommended for ROAR Junior because in our testing, we found that in scenario where voltage drop is significant, such as when motor needs to spin from forward to backward direction immediately, L298N will have short second of short circuiting and causing the +5V to be not stable, thus shutting down anything connected to it.

The ENA and ENB are for the PWM signals for MotorA and MotorB respectively. IN1, IN2, IN3, and IN4 are to control the direction of the motor. Combining all these, users are able to control the direction of each side of the differential drive, and also the magnitude of the force exerted onto the drive train.

Communication Pipeline

For The Asynchronous HTTP server, we decided to use the ESPAsyncWebServer package. Below is a list of routes being published:

```
/forward -> set IN pins such that the vehicle is driving forward
/backward -> set IN pins such that the vehicle is driving backward
/left -> set IN pins such that the vehicle is driving left
/right -> set IN pins such that the vehicle is driving right
```

```
/left_spd -> left RPM (from 0 - 255)
/right_spd -> right RPM (from 0 - 255)
```

On the client side, user will simply need to send a HTTP GET request with parameters as data. For example, here is a message that will tell the car to go forward:

```
#!/usr/bin/python
import requests
requests.get("http://192.168.1.24:81",
              params={"forward":True,
                      "left_spd":1,
                      "right_spd":1})
```

Software (Python)

Now the ROAR platform originally only considered Ackermann control. Therefore, we need to map Ackermann Control signals from ROAR to Differential commands.

In Ackermann control, we have steering and throttle. In differential control, we can control the speed and direction of each side of the drivetrain.

Therefore, the conversion will be as the follow logic:

```
ackerman_to_diff_gain = A_NUMBER_BY_TEST_AND_TRIAL
message = {}
if throttle == 0:
   message["stop"] = True
elif throttle > 0:
    message["forward"] = True
else:
    message["backward"] = True

if steering == 0:
    msg["left_spd"] = 1
    msg["right_spd"] = 1
elif steering < 0:
    # turning left
    msg["left_spd"] = 0
    msg["right_spd] = ackerman_to_diff_gain * steering
else:
    msg["right_spd"] = 0
    msg["left_spd] = ackerman_to_diff_gain * steering
```

We will conform to the logic mentioned in the previous chapter regarding ROAR's architecture, where we have a *setup()*,*start_game_loop()*, and *finish()* function.

In the setup function, we will setup the PyGame and read in all the configs.
In the start game loop function, we will

1. fetch user input from pygame
2. fetch the newest sensor data (for the base version, just the RGB and the distance sensor reading)
3. run an agent step (this is where user can write their autonomous driving code)
4. transform the Ackermann control to differential command
5. send the differential command to the ROAR Junior vehicle.

And in the finish function, we will send calls to stop the vehicle and shut down any other threads running.

Bibliography

[1] Johannes Betz et al. "Autonomous Vehicles on the Edge: A Survey on Autonomous Vehicle Racing". In: (Feb. 2022). DOI: 10.48550/arxiv.2202.07008. URL: https://arxiv.org/abs/2202.07008v1.

[2] Joydeep Biswas and Manuela Veloso. "Depth Camera Based Indoor Mobile Robot Localization and Navigation". In: ().

[3] *Bluetooth Low Energy - Wikipedia*. URL: https://en.wikipedia.org/wiki/Bluetooth_Low_Energy.

[4] G. Bradski. "The OpenCV Library". In: *Dr. Dobb's Journal of Software Tools* (2000).

[5] Greg Brockman et al. *OpenAI Gym*. 2016. eprint: arXiv:1606.01540.

[6] *Changes between ROS 1 and ROS 2*. URL: http://design.ros2.org/articles/changes.html.

[7] *CS 294 ROAR on Behance*. URL: https://www.behance.net/gallery/132784211/CS-294-ROAR?tracking_source=search_projects_recommended%5C%7CCS294-137.

[8] *DeepSORT — Deep Learning applied to Object Tracking — by Ritesh Kanjee — Augmented Startups — Medium*. URL: https://medium.com/augmented-startups/deepsort-deep-learning-applied-to-object-tracking-924f59f99104.

[9] *Demystifying Python Multiprocessing and Multithreading — by David Chong — Towards Data Science*. URL: https://towardsdatascience.com/demystifying-python-multiprocessing-and-multithreading-9b62f9875a27.

[10] Alexey Dosovitskiy et al. "CARLA: An Open Urban Driving Simulator". In: *Proceedings of the 1st Annual Conference on Robot Learning*. 2017, pp. 1–16.

[11] Edward Fang. "Dynamic Deadlines in Motion Planning for Autonomous Driving Systems". In: (2020). URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-58.html.

[12] *Freenove — Home*. URL: https://www.freenove.com/.

[13] "Implementation of the Pure Pursuit Path 'hcking Algorithm". In: (1992).

[14] *Indy Autonomous Challenge - Official Website*. URL: https://www.indyautonomouschallenge.com/.

[15] Christos Katrakazas et al. "Real-time motion planning methods for autonomous on-road driving: State-of-the-art and future research directions". In: *Transportation Research Part C: Emerging Technologies* 60 (Nov. 2015), pp. 416–442. ISSN: 0968-090X. DOI: 10.1016/J.TRC.2015.09.011.

[16] Mincheul Kim. "PATH PLANNING AND FOLLOWING FOR AUTONOMOUS VEHICLES AND ITS APPLICATION TO INTERSECTION WITH MOVING OBSTACLES". In: (2017).

[17] Jayakanth Kunhoth et al. "Indoor positioning and wayfinding systems: a survey". In: (). DOI: 10.1186/s13673-020-00222-0. URL: https://doi.org/10.1186/s13673-020-00222-0.

[18] Steven Macenski et al. "The Marathon 2: A Navigation System". In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2020.

[19] *PID control explained. PID control is a mathematical approach... — by Mattia Maldini — Medium*. URL: https://maldus512.medium.com/pid-control-explained-45b671f10bc7.

[20] *Programming Your Traxxas Electronic Speed Control — Traxxas*. URL: https://traxxas.com/support/Programming-Your-Traxxas-Electronic-Speed-Control.

[21] *pydantic*. URL: https://pydantic-docs.helpmanual.io/.

[22] *pygame/pygame: pygame (the library) is a Free and Open Source python programming language library for making multimedia applications like games built on top of the excellent SDL library. C, Python, Native, OpenGL*. URL: https://github.com/pygame/pygame.

[23] *Robot Open Autonomous Racing (ROAR™) — FHL Vive Center for Enhanced Reality*. URL: https://vivecenter.berkeley.edu/research1/roar/.

[24] Atsushi Sakai. *AtsushiSakai/PythonRobotics: Python sample codes for robotics algorithms*. URL: https://github.com/AtsushiSakai/PythonRobotics.

[25] *SciPy v0.18.1 Reference Guide*. URL: https://docs.scipy.org/doc/scipy-0.18.1/reference/generated/scipy.optimize.linear_sum_assignment.html.

[26] *Servo - Arduino Reference*. URL: https://www.arduino.cc/reference/en/libraries/servo/.

[27] *The Grand Challenge*. URL: https://www.darpa.mil/about-us/timeline/-grand-challenge-for-autonomous-vehicles.

[28] *The WebSocket API (WebSockets) - Web APIs — MDN*. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API.

[29] *Three Methods of Vehicle Lateral Control: Pure Pursuit, Stanley and MPC — by Yan Ding — Medium*. URL: https://dingyan89.medium.com/three-methods-of-vehicle-lateral-control-pure-pursuit-stanley-and-mpc-db8cc1d32081.

[30] *Tracking camera T265 – Intel RealSense Depth and Tracking Cameras*. URL: https://www.intelrealsense.com/tracking-camera-t265/.

[31] *Understanding World Tracking — Apple Developer Documentation*. URL: https://developer.apple.com/documentation/arkit/configuration_objects/understanding_world_tracking.

[32] *User Datagram Protocol - Wikipedia*. URL: https://en.wikipedia.org/wiki/User_Datagram_Protocol.

[33] *Vehicle-to-Vehicle Communication — NHTSA*. URL: https://www.nhtsa.gov/technology-innovation/vehicle-vehicle-communication.

[34] Steven Waslander. *Coursera Self Driving Cars Specialization*.

[35] Nicolai Wojke, Alex Bewley, and Dietrich Paulus. *SIMPLE ONLINE AND REAL-TIME TRACKING WITH A DEEP ASSOCIATION METRIC*.

[36] Shouwen Yao et al. "Autonomous-driving vehicle test technology based on virtual reality". In: *The Journal of Engineering* 2018 (16 Nov. 2018), pp. 1768–1771. ISSN: 2051-3305. DOI: 10.1049/JOE.2018.8303. URL: https://onlinelibrary.wiley.com/doi/full/10.1049/joe.2018.8303%20https://onlinelibrary.wiley.com/doi/abs/10.1049/joe.2018.8303%20https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/joe.2018.8303.

[37] *Zero to Hero: A Quick Guide to Object Tracking: MDNET, GOTURN, ROLO – CV-Tricks.com*. URL: https://cv-tricks.com/object-tracking/quick-guide-mdnet-goturn-rolo/.

[38] Qian-Yi Zhou, Jaesik Park, and Vladlen Koltun. "Open3D: A Modern Library for 3D Data Processing". In: *arXiv:1801.09847* (2018).