Alset IoT

"Hugs the Lanes"

Software Development Process Documentation

version 1.5.4

Steven DeFalco, Lucas Hope,

Jude Lee, Daniel Storms

Table of Contents

1. Introduction

1.1 Purpose

According to the World Health Organization, there are 1.2 *million* automobile fatalities each year worldwide. However, the development of self-driving technology is predicted to save upwards of 10 million lives throughout the next decade. With our product, we hope to contribute to increased safety on the roads through the means of creating self-driving software to be used in *Alset* automobiles. We believe that software engineers can have a critical role in designing a safer future and we have hope to be a part of this progression.

1.2 Project Overview

In this project, our team will develop iterable software with the goal of creating effective and safe self-driving technology for the *Alset* automobile company. To accomplish this, we will develop both front and back end software that uses Internet of Things (IoT) technology. IoT is a powerful tool that will allow us to rapidly detect and process significant amounts of data. From an array of sensors on the car, we will be able to accept constant streams of information regarding the car's surrounding environment. This information will be passed into our software, resulting in quick decision making that will apply necessary adjustments to the car's motion. The project is a mission critical, real-time embedded system; thus, we will take measures to ensure the utmost reliability and safety of our product. This process will involve rigorous testing: both with our software digitally and physical components as well.

1.3 Product Specifications

Our software will be adaptable and reliable for any circumstance that may be encountered by the automobile. Our software will be trained to avoid major obstacles, identify lane markers, identify pace of traffic, and more. IoT will allow the car to act on this training in real time using live data and our backend software. Our overall goal is to emulate the driving of a human being while providing additional features that appeal both to safety concerns and creating a positive user experience. This will involve automatic lane checking, automatic braking, and other actions that can be executed better by our product than by a human driver. Although *not exhaustive*, the following list outlines examples of the functionality we hope to implement in our product:

1.3.1 List of features

→ Lane detection

→ Automatic braking

→ Adaptive cruise control

→ Blind spot detection

→ Parking assist

→ Traffic sign recognition

1.4 Project Scope

We expect to complete this project within the next 3 months. In order to achieve this, we will assess our progress on a weekly basis. In addition, we will be thorough in our planning and communication, with a schedule laid out to accurately track our progress. Due to the collaborative nature of this production, we expect to hold both ourselves and each other accountable during our weekly check-ins. We intend to follow a modified spiral planning model, as to continually advance our software based on feedback from testing. We plan to make software updates weekly barring lack of major progress or excessive rapid progress that would warrant delayed or more frequent releases. This document will be updated as we develop our product and narrow down the specifications; these updates will happen frequently to ensure that this document closely represents the current state of our product. It must be noted that early iterations of our product *will not* be capable of meeting all of our specifications; we will rely on frequent modifications and improvements to promote product development.

1.5 Our Team

Our team consists of four members: Steven DeFalco, Lucas Hope, Jude Lee, and Daniel Storms. Together, we boast a strong knowledge of programming languages, software development, project management, and effective communication. Our collective learning experience includes but is not limited to the following subjects: data structures, algorithms, computer architecture, deep learning, database management, and operating systems. Furthermore, we each have individually applied our skills to projects past, and therefore expect little issue in developing code as a team. Given our discipline and plan to follow a scheduled software development process, we expect to complete this project within the expected time frame. If we encounter any issues, we will consult with each other and outside resources to guarantee a swift recovery.

2. System Architecture

2.1 General Architecture Overview

Both our functional and software architecture are crucial components in the development of this system. The architecture is what will dictate the flow of data and information through different modules in the system eventually resulting in adjustments to the vehicle’s motion. Architecture is essential in ensuring the safety and effectiveness of our vehicle system and will be the means through which the system will meet our specifications (to be outlined in **Section 3**). This section explains the physical components, along with the data transformations and analysis that will take place in our system.

2.2 Model of the Architecture

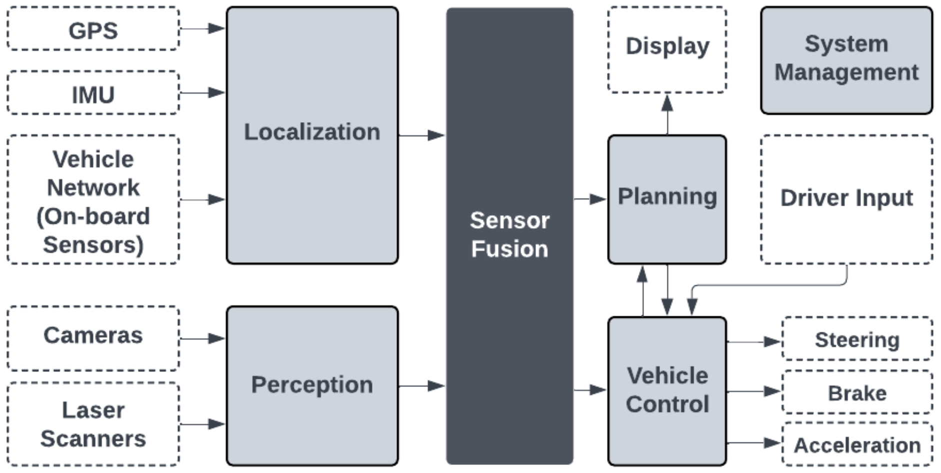


Figure 1: Functional Architecture Diagram (adapted from Guo, Qiaochu: see **references**)

This model roughly approximates the architecture the system will maintain throughout the development of our self-driving system. Examining the figure from left to right, there are physical components and monitors that feed into two primary branches of the system’s underlying capabilities: **localization** and **perception.** Together, these capabilities are joined in the **sensor fusion** step which makes decisions regarding **movement planning** and **vehicle control** with the help of **driver input.** Finally, this information will be used together to formulate the vehicle's actual movements including **steering**, **braking**, and **acceleration. Figure 1** (above) concisely illustrates the overall flow of data through the system and demonstrates how data will be transformed, evaluated, and result in adjustments to the vehicle’s movement.

2.3 Architecture Components

2.3.1 Localization

a. Localization is an integral component of autonomous vehicles. Localization allows vehicles to find their location within a map which introduces a greater understanding of the immediate and relative environment. This allows for the use of a GPS.

2.3.2 GPS

a. GPS, short for Global Positioning System, allows users with a GPS receiver to obtain geolocation and time information.
b. Users can use a GPS to help navigate to an intended destination by combining real time GPS coordinates with digital maps.
c. GPS feeds essential information into the Localization component of the system architecture.

2.3.3 IMU

a. IMU, short for **Inertial Measurement Unit**, measures acceleration, angular velocity, and magnetic fields.

b. IMUs have three primary components: <u>Gyroscopes</u>, <u>Accelerometers</u>, and <u>Magnetometers</u>.

c. IMU data provides *vital* information including the vehicle’s instantaneous movement in the physical space. Vehicles rely on IMU for vehicle stability through use of gyroscopes and accelerometers.

d. IMU data is fed into the localization component where it is used to establish a model of the vehicle with respect to the surrounding environment

2.3.4 Vehicle Network

a. Vehicle network collects information about and from surrounding vehicles and landmarks. This information also serves as a key input to the localization component as it will allow the system to better understand the immediate environment.

b. Although unrealistic in a non-standardized car market, the vehicle network component will be able to allow for communication with other cars should this technology be standardized in cars. This would help the system ensure cars maintain a safe distance.

2.3.5 Perception

a. Perception extracts environmental information from sensors such as cameras, LiDAR, GPS, and IMU.

b. Perception allows the vehicle to scan its surroundings and make decisions based on information provided by scanners.

2.3.6 Cameras

a. Cameras allow for the vehicle to recognize its surroundings and thus establish a 360-degree view of its surroundings. The system makes driving decisions with this information.
b. Cameras capture color; LiDAR cannot. However, cameras are less consistent in different light conditions.

2.3.7 Laser Scanners

a. LiDAR, short for Light Detection and Ranging, allows vehicles to measure distances using light.
b. Using light to measure distances allows for rapid measurements via non contact data capture.

2.3.8 Sensor Fusion

a. Sensor Fusion aggregates input from multiple radars, sensors, LiDar, and other perception devices and forms a single model of the environment around a vehicle.
b. Combining raw data in the sensor fusion component is essential in allowing for the system to make intelligent and well-informed decisions.

2.3.9 Planning

a. Planning encompasses the use of already aggregated sensor data to make decisions for the system. The planning component synthesizes the next steps that the system should make and then once confirmed feeds these steps into vehicle control to be acted on.

2.3.10 System Management

a. The system management component collects and stores the vehicle data. This includes but is not limited to the driver's performance, fuel management, and GPS tracking.
b. System management includes fault management, logging facility, and the human-machine interface.
c. System management promotes software that mitigates human and computer resource shortages.

2.3.11 Driver Input

a. The driver input component collects and processes all of the drivers current input into the car system. Driver input includes any movement of the steering wheel, acceleration pedal, breaking pedal, or any interaction with a digital user interface (DUI) system.
b. Inputs from the driver are transferred into vehicle control where they may be augmented or completely

ignored before possibly being executed by the vehicle in the form of steering, breaking, or acceleration (to be detailed below)

   i. Note that all driver input is closely monitored before being acted on. This is a crucial aspect of how the system implements driver assistance.

2.3.12 Vehicle Control

a. Vehicle control component combines the system's plans with the driver's input and precisely details the exact adjustments that will be made to the vehicle’s motion in the form of steering, braking, or acceleration.

2.3.13 Steering

a. The steering component represents one of the three main outputs from the data transformations in the system
b. The steering component receives instructions from the vehicle control component and makes physical adjustments to the trajectory/direction of the car by manipulating the angle of the wheels

2.3.14 Braking

a. Braking is the second primary output of the system.

b. The braking component receives instructions from the vehicle control component and will engage the brakes at variable levels as indicated in the instructions.
c. Braking occurs when the planning component or driver input component relay to the vehicle control component that deceleration is required/desired. The vehicle control component determines the magnitude required and instructs this component.

2.3.15 Acceleration

a. Acceleration is the third and final primary output of the system.
b. The acceleration component receives instructions from the vehicle control component and increases the speed of the vehicle through controlling the physical systems.
c. Acceleration occurs when the planning component or driver input component relay to the vehicle control component that acceleration is required/desired. The vehicle control component determines the magnitude required and instructs this component.

2.4 Analysis Framework

The following is a list of the analysis framework components that we intend to apply to our project:

2.4.1 Interoperability

- ➔ Interoperability refers to the ability for our systems to meaningfully exchange information so that the systems run as expected. Interoperability is a crucial concept that manages the data our sensors constantly capture and orchestrates its transfer to all systems which are waiting for such input.

2.4.2 Modifiability

- ➔ Modifiability refers to our software's adaptability to new updates and changes. We design our software so that it includes functionality for the implementation of new technology and the ability for existing technology to be updated.

2.4.3 Performance

- ➔ As a real-time software, our software must be extremely efficient and have little latency. Our sensors must have high refresh rates for quick vision processing and to ensure lack of missed data. Our software must similarly be optimized so that systems are able to quickly act on the current data and make decisions for the vehicle.

2.4.4 Security

- ➔ Due to the extreme risk inevitable when designing self-driving automobile software, we must maintain a high level of security to prevent outside malice or ignorance from posing danger to the consumer. Hence, we will provide encryption and two-factor authentication when using the software.

2.4.5 Testability

- ➔ Our software must be thoroughly tested before being released to the public. As a result, we intend to design it in a way that can be safely and easily tested while risking no harm to real humans. This includes the use of rigorous testing software systems and simulated driving environments to test interoperability of our systems.

3. Requirements

3.1 Functional Requirements

3.1.1 Vehicle adaptively brakes when object is detected

Vehicle Control System commands child modules to decrease speed or stop when an object is detected within a range that causes imminent danger to the machine or user considering the car's current speed.

System must correctly identify objects that require braking over 99% of the time in testing cases.

Preconditions:

- System is powered on
- $\gamma \neq 0$ miles per hour (i.e. vehicle is in motion)
  - Where γ represents real-time system speed as collected and processed from the Inertial Measurement Unit (IMU)
- Object of *significant size* is detected within the space which vehicle is expected to occupy within the *immediate future*

- ➞ Let ***significant size*** be defined so that it will exclude objects that pose no threat to the user or machine’s safety, while including objects that could damage the vehicle’s structure.
- ➞ Let ***immediate future*** be defined as the time that it will take for the car to come to a complete stop given current speed. *Immediate future* is calculated by the Planning module with information received from Cameras, Laser Scanners , and IMU and is interpreted by the Perception module.
- ➞ Any object that is within the distance the vehicle requires to come to a complete stop and is of considerable stature will prompt the vehicle’s Planning module to instruct VCS to begin braking the car accordingly.

Postcondition: Vehicle brakes in accordance with the braking component outlined in 2.3.14. The vehicle will also make an audible sound to alert the driver that they are/were driving too close to an object.

3.1.1.1 On-board sensor, laser scanners, and cameras detect object and send data through Localization and Perception modules and subsequently into the Sensor Fusion module

3.1.1.2 IMU sends real-time vehicle motion data through the Localization module and into Sensor Fusion module.

3.1.1.3 Sensor Fusion upon receiving object location normalizes the data and sends it to planning module

3.1.1.4 Planning module will instruct the Vehicle Control System (VCS) to adaptively apply the brakes if the object is within the the distance the car requires to come to a complete stop as calculated by the Planning module based on current velocity (Υ) and road conditions. Planning will also instruct the system to make an audible beeping sound alerting the driver that they were driving too close to an object.

3.1.1.5 Planning module will log metadata of the detected object to the System Management module for record-keeping. The Planning module will also log the instructions sent to the VCS module.

3.1.2 Vehicle automatically detects lanes in real-time

Sensors and Cameras continually scan the ground surrounding the vehicle and detect lanes. The vanishing point of these *lines is considered the "lane boundary" and is used by the systems modules to determine the car's position relative to the lanes.*

System must correctly identify lines with 6 inches of precision in all test cases.

Preconditions:

- System is powered on
- $\gamma \neq 0$ miles per hour (i.e. vehicle is in motion)
- Vehicle is on marked road as determined by GPS

Postcondition: Vehicle utilizes sensors to determine its position relative to the real-time "lane boundary" and alerts the user via the display module that they have exited their lane.

3.1.2.1 On-board sensors, laser scanners, and cameras scan the ground on either side of the car and send the information to the Perception module.

3.1.2.2 The Perception module identifies the lane boundary (or lack thereof), then sends information encoding the relative location of the *lane boundary* to the Planning module.

3.1.2.3 The Planning module stores data denoting the vehicle’s position relative to the *lane boundaries*. This data is requested in other requirements, but is stored in real-time in the Planning module.

3.1.2.4 The Planning module displays the car’s current location within the lane on Digital User Interface (Display module). The Display module will sound alerts if the Planning module detects that the vehicle is moving out of its current lane without the use of a turn signal.

3.1.3 Vehicle Prevents User From Merging Into Object

Sensors and cameras continually scan vehicle surrounding in real-time to detect objects which do or may occupy the space that the vehicle may merge into or is currently merging into.

System must adequately prevent the vehicle from merging into objects in 99% of the test cases.

Preconditions:

- ↪ System is powered on
- ↪ $\curlyvee \neq 0$ miles per hour (i.e. vehicle is in motion)

- An object of *significant size* detected in either blind spot of the vehicle or an object approaching the blind spot of the vehicle.

Postcondition: Blind spots warning lights turn on. Considering driver input, the vehicle either reduces or nullifies current steering to prevent the vehicle from entering the occupied lane. System may apply additional steering to prevent collision.

3.1.3.1 On-board sensor, lasers scanners, and cameras detect object in blind spot and transfer information to the Perception and Localization Modules

3.1.3.2 IMU sends real-time vehicle motion data through Localization module and into Sensor Fusion module to be combined with data about the detected object.

3.1.3.3 Sensor fusion upon receiving object location normalizes the data and sends it to planning module

3.1.3.4 Planning module instructs vehicle control system (VCS) to override user steering input in order to prevent collision.

3.1.3.5 Planning module instructs Display module (including both DUI and side mirror lights) to alert the user of an impending object in the collision path via on-screen alert and rapidly flashing LEDs.

3.1.4 Vehicle maintains lane position and speed

Cruise control system will keep the vehicle positioned approximately centered in the lane and at desired speed. Car will accelerate or apply brakes as necessary.

Cruise control must keep the car centered within 1 foot of the true center of the lane in all test cases and within 6 inches of the true center in over 80% of the test cases.

Preconditions:

- ↪ System is powered on
- ↪ $\curlyvee \neq 0$ miles per hour (i.e. vehicle is in motion)
- ↪ Vehicle is on highway as learned from GPS reading in tandem with knowledge of local roadmaps
    - ➙ ***Highway*** will be defined as a road with an average speed limit of 55 mph or more within the next 10 miles of roadway. Additionally, *highway* only refers to roads without traffic lights or other road signs that demand frequent stops from the user.

Postcondition: Car maintains appropriate speed on the given roadway and in accordance with the driver’s input while also keeping the car centered in the line and navigating away from any objects.

3.1.4.1 Planning module waits for and accepts input from Driver input module requesting the car to activate cruise control.

3.1.4.2 Planning module reads real-time information from sensor fusion including that which has been gathered from GPS and IMU to decide whether the car is in a state capable of cruise control.

3.1.4.3 Planning module instructs the Display module to display to the driver that the car is entering cruise control at the current speed.

3.1.4.4 Planning module continuously accepts real-time data pooled in the Sensor Fusion module and instantaneously instructs Vehicle Control module to operate Steering, Braking, and Acceleration in order to hold speed. Speed should not exceed, that which was set when the driver enters cruise control by more than 2 mph nor should it fall more than 5 mph below the set speed.

3.1.4.5 Planning and Vehicle Control module continuously check for inputs from the Driver Input module requesting to exit cruise control. If a request is made through application of the brake pedal, then the Vehicle Control module will slowly disengage all acceleration and non-normal brake application. The Planning module will alert the user by sending a signal to the Display module that the cruise control is being disengaged.

3.1.4.6 Planning module continuously accepts real-time data pooled in the Sensor Fusion module and determines whether it is feasible for the system to remain in cruise control. If cruise control is no longer deemed feasible, cruise control will disengage and the driver will be notified as detailed in 3.1.4.5 above.

3.1.5 Vehicle changes lanes when instructed

Vehicle will utilize the blind spot detection outlined in 3.1.3 to perform an automatic lane change upon driver input when the vehicle is in the cruise control mode outlined in 3.1.4.

System must correctly and fully execute lane change from lane center to lane center within 8 seconds and must pass *all test cases that have a vehicle in the blind spot of the lane the car wishes to merge into.*

Precondition:

- System is powered on
- $v \neq 0$ miles per hour (i.e. vehicle is in motion)
- Cruise control is active
- Driver input detected via turn signal

Postcondition: The vehicle steers into either the left or right lane depending on the turn signal usage while maintaining the vehicle’s real-time speed.

3.1.5.1 On-board sensor, lasers scanners, and cameras utilize the blind spot detection technology outlined in *3.1.3* to inform the Planning module if the lane chosen by the user is clear to merge.
3.1.5.2 If the lane is determined to be clear, the Planning module informs the vehicle control system (VCS) to steer into the new lane which is then detected by the lane detection technology outlined in 3.1.2.

3.1.6 Vehicle automatically activates lights

Vehicle will detect light using Laser Scanners to determine whether or not to enable or disable in-car display lights and headlights unless the driver has manually activated the lights.

System must correctly activate/deactivate lights when the ambient light is within 30 lux of the desired cutoff brightness and defined below.

Precondition:

- System is powered on
- $X < 200$ *lx*
  - Where X is the unit of illumination per unit area that is detected.
  - Where *lx* is the symbol for lux which is the unit of illumination per unit area which is equal to one lumen per square meter. This is used as a measure of intensity of the light that hits or passes a surface to the human eye.
- Driver has not manually activated the in-car display or headlights.

Postcondition: The vehicle automatically turns on display lights within the car and activates headlights depending on the illumination detected. If the user has manually activated the lights or headlights, the vehicle will not change the state of the manually activated component.

3.1.6.1 Laser scanners scan the surrounding environment to calculate the lux and send the information to the Perception module.

3.1.6.2 Perception module allows the vehicle to obtain information from the environment and sends data to Sensor Fusion.

3.1.6.3 Sensor Fusion forms the environment which includes light detection and sends data to the planning module.

3.1.6.4 The planning module then determines if the lux of the environment is determined to be lower than the required threshold. If so, the vehicle automatically turns on in-car display lights and activates the headlights if the user has not manually activated headlights through the Digital User Interface.

3.1.7 Vehicle automatically activates high beams

Vehicle system will activate high beams in instances that do not adversely impact the driver or the drivers of other vehicles.

System correctly identifies the lack of other vehicles (oncoming vehicles or other vehicles that lead the vehicle) with 95% accuracy. Must pass all cases where there is an oncoming vehicle or one that is being followed.

Precondition:

- System is powered on
- Vehicle lights are turned on either manually or automatically, as described in section <u>3.1.6</u>.
- Y is greater than 25 miles per hour.
- It is dark outside (X < 200 lx), and weather that affects the functionality of high beams is not detected (i.e. fog, rain, snow, etc.)
- Sensor fusion does not sense oncoming vehicles with headlights on and does not sense vehicles ahead with tail lights on.

Postcondition: The vehicle automatically turns on high beams. If any of the following are detected by the system, the high beams will turn off:

- Oncoming vehicles or vehicles ahead are detected.
- The vehicle reduces its speed to below 15 mph.

- Weather that impacts high beam functionality is detected
- Headlights are turned off
- If daylight is detected outside (X > 200 lx)

3.1.7.1 Cameras and Sensors send data to the Perception module and Sensor Fusion module, which extracts information relating to nearby cars, light levels, and the weather.
3.1.7.2 The IMU sends the current speed of the vehicle to the Sensor Fusion module.
3.1.7.3 Planning receives all of the information necessary to determine if the automatic high beams should be enabled.
3.1.7.4 Planning sends a signal to the Vehicle Control System to activate the high beams.
3.1.7.5 If any of the modules detect a condition where the high beams should be turned off, then the data will be sent to the Planning module, which will send a signal to VCS to turn off the high beams.

3.1.8 Windshield wipers turn on in rain

System detects rain on the windshield and will turn on or ensure that the windshield wipers are active at an appropriate rate.

Windshield wipers turn on within the first 5 seconds of consistent rainfall in all test instances and turn off within 15 seconds of final rainfall in all test cases.

Precondition:

- System is powered on
- On-board sensors detect an average of 5 drops/square inch/second across a 5 second period
- Windshield wiper settings are set to automatic
- Windshield wipers have not been manually turned on

Postcondition: Windshield wipers activate at a variable frequency for the duration of the time that rainfall is detected on the windshield sensor

3.1.8.1 On-board Sensors on the top-center of the windshield detect rainfall in real time and send data into the Sensor Fusion module where it will be stored in the form of liquid drops per square inch per second.

3.1.8.2 Sensor Fusion stores previous 30 seconds of *potential* rainfall data from On-Board Sensors and if averages to more than 5 drops per square inch per second for 5 concurrent seconds then will instruct Vehicle Control

System to activate windshield wipers. Windshield wiper frequency increases when/if 12 drops per square inch per second and 25 drops per square inch per second are achieved as the running average.

3.1.8.3 Sensor Fusion module will instruct VCS to decrease windshield frequency as each of the thresholds is met in a decreasing rainfall scenario. Wipers are completely turned off, if drops per square inch per second is on average less than 1 for a sustained period of 15 seconds.

3.1.9 Rear View Camera

When the car is in reverse, a backup camera displays on the DUI detailed in section 3.2.1.

DUI displays the live backup feed with less than 50 ms of latency and never disengages before the car is shifted out of reverse

Preconditions:

- System is powered on
- Vehicle’s is in reverse gear

Postcondition: The rear view camera is displayed on the screen. When the car is shifted out of reverse. The DUI will return to what it was displaying before the car was set to reverse.

3.1.9.1 The Driver's Input shifts the gear to reverse, sending a signal to Vehicle Control, which sends this information to Planning.
3.1.9.2 Cameras send the data from the camera on the back of the car through Perception and Sensor Fusion to Planning.
3.1.9.3 Planning instructs the Display to show the backup camera.

3.1.10 Parking sensors

When in reverse, the DUI, detailed in section 3.2.1, will display nearby objects that are detected by sensors around the car.

Parking sensors throw appropriate warnings in all of the test cases and relative vehicle position is within 4 inches of accuracy 95% of the time during testing.

Precondition:

- System is powered on
- Vehicle is in reverse gear
- Sensors detect nearby objects

Postcondition: System will make an audible sound and the Display will show the position of the detected object(s).

3.1.10.1 The Driver’s Input shifts the gear to reverse, sending a signal to Vehicle Control, which sends this information to Planning.
3.1.10.2 Sensors send nearby object data to Perception, which will interpret if the object is a threat to the car. This data is sent through Sensor Fusion into Planning.
3.1.10.3 Planning instructs the DUI to display sensor information around the vehicle. Planning will also instruct the system to make audible beeping sounds varying in intensity based on how close the vehicle is to the object.

Note that if the vehicle detects that that car will hit an object while parking (whether in drive or reverse), the car will stop due to the automatic braking functional requirement detailed in section 3.1.1. The vehicle will also make beeping sounds if the vehicle comes too close to an object while in drive.

3.2 Non-functional Requirements

3.2.1 Display

Digital User Interface (DUI) as introduced in Section 2.2 is the hub for the majority of the information attained by cameras and sensors. This includes navigation, lane detection alerts, and a basic interactive GUI. The following is a list of some of the features that will be implemented as part of the display:

3.2.1.1 Interactive Radio

- The vehicle will have a radio which the user can use the display to interact with. This can also be connected via AUX cord or Bluetooth.

3.2.1.2 Camera Display

- The vehicle will be able to display the view of the cameras on the display.

3.2.1.3 Lane Detection and Warning

- The vehicle will display lane detection warnings if the car drifts out of the lane as well as a digital image of the car driving and its surroundings, as gathered by the sensors

3.2.1.4 Navigation

- The vehicle will display a map that updates the vehicle's location. Additionally, the user will be able to navigate to a given destination using the map software.

3.2.2 Performance

The performance of the vehicle must be consistent and fast, as it is a real-time system. As a result, the technology in the cameras and sensors must be fully optimized. This is done through the utilization of LIDAR, Radar, and Sonar technologies.

3.2.2.1 LIDAR

- LIDAR utilizes infrared beams to scan the surrounding environment. LIDAR uses the speed of light to calculate the distance to nearby objects. This allows the car to track small objects with *immense* precision.

3.2.2.2 Radar

- Radar utilizes radar waves to help determine the position of environmental objects and their angles, ranges, and velocities.

3.2.2.3 Sonar

- Sonar detects sound waves and surrounding objects. Detects other vehicles as well.

3.2.3 Reliability

The reliability of the car is extremely important considering it is a real-time system. Every feature in the car, from the seatbelts to the brakes must function as intended at all times. The following are some of the most critical features to ensure the reliability of.

3.2.3.1 Communication

- The software must communicate reliably between all components of the system. The information gathered from the cameras and sensors must be interpreted reliably.

3.2.3.2 Responsiveness

- The DUI must reliably respond to user inputs and utilize the software to perform their intended action.

3.2.3.3 Vehicle Control System

- The Vehicle Control System must work reliably. This includes the brakes, steering wheel, and seatbelts.

3.2.4 Security

The security of the car is essential to the safety of the driver. This takes place in a variety of aspects, including the functionality of door and window locks and the key fob being unique to each vehicle. Additionally, the material of the window is strong to prevent break-ins.

3.2.4.1 Locks

- The window and the door locks are secure and do not allow for easy break-ins. This is done through creating a uniquely shaped keyhole.

3.2.4.2 Key Fob

- The key fob is unique to each individual vehicle as to prevent the ability to open the car with a fob from the same manufacturer.

3.2.4.3 Window Material

- The side windows are made of laminated glass to prevent break-ins. Additionally, the windshield is made of tempered glass.

3.2.5 Quality of Life

The vehicle will have a number of quality of life features to make for a better driver experience. The following are some examples:

3.2.5.1 Remote Start

- The vehicle will have the ability to remotely start and warm the car prior to the driver entering. This will be implemented with a button on the key.

3.2.5.2 Seat Warming

- The vehicle will have seat warmers. These can be turned on automatically when the temperature falls below 60 degrees inside the car.

3.2.5.3 Warmed Steering Wheel

- The vehicle will have a warming steering wheel. This can be manually turned on using a setting available in the DUI.

4. Requirement Modeling

4.1 Use Case Scenarios

4.1.1 Forward and rear-facing sensors detect object(s) in path of vehicle

Precondition:

- Vehicle systems are turned on
- Vehicle is in motion
- Object (or objects) of significant size detected by sensor network

Postcondition:

- Car brakes in accordance with the adaptive braking outlined in 3.1.1

Trigger: Vehicle Control System (VCS) detects object(s) that may pose danger in the vehicle's path.

1. Sensor Fusion detects object of significant size within short distance within the vehicle's projected path

2. Network of sensors sends information about object through Sensor Fusion module for interpretation and normalization
3. Sensor Fusion module aggregates data about the potential object and the vehicle’s motion and passes it into the Planning module
4. Planning module calculates the required adaptive braking speed (if necessary) in accordance with 3.1.1
5. Planning module sends the brake command to the VCS
6. VCS brakes in accordance with the commands received from the VCS
7. Planning sends data about the object and the current braking status to the Display module
8. Display module commands display unit to show object location and the vehicle’s current braking status in order to alert driver of the braking
9. Planning module sends all information about the current braking to System Management module for logging
10. Sensor Fusion sends normalized sensor data to System Management module for logging

4.1.2 Driver requests automatic lane change

Precondition:

- Vehicle systems are turned on
- Vehicle is in motion
- Cruise Control turned On
- Driver requests automatic lane change

Postcondition:

- Vehicle performs lane change while maintaining speed

Trigger: Turn signal activated while in Cruise Control

1. Driver activates the turn signal while in Cruise Control mode
2. System Control sends request to Planning to perform lane change
3. Sensor Fusion aggregates data from the sensor network, normalizes the data, and sends it to the Planning module
4. Planning module uses normalized data from Sensor Fusion to determine of a lane change is possible given the car’s current environment
    a. If the lane change is not possible, Planning module will send pertinent data to the Display module to inform the user that the vehicle is unable to perform

lane change (reason for denied request is displayed on central display unit)

5. Knowing the lane change can be safely performed, the Planning module sends commands to the VCS to perform the lane change
6. Planning module sends command to Display module to show the live progression of the lane change
7. Planning sends related data to System Management module
8. VCS sends related data to System Management

4.1.3 Sensors below the vehicle maintain lane position during Cruise Control

Precondition:

- Vehicle system is turned on
- Vehicle is in motion
- Cruise control is active

Postcondition:

- Vehicle stays positioned relatively centered within the lane at all times within accordance with requirements set in 3.1.4.

Trigger: The driver has activated cruise control

1. The driver turns on the cruise control system with the cruise control button on the control panel
2. Sensor network sends live sensor data to Sensor Fusion module for normalization
3. Sensor fusion passes normalized data to Planning module which determines if the car can enter cruise control
4. Car enters cruise control if valid conditions for cruise control (defined in Functional Requirements)
5. When in cruise control, Sensor Fusion instantaneously receives data from sensors, normalizes the data, and sends the data to the Planning module
6. Planning module will interpret sensor data and calculate the car’s real time position in the lane
7. Planning module will send commands to the VCS to adjust the vehicle’s trajectory in the case that the vehicle is tending towards one side of the lane
8. Lane position is monitored and managed until the cruise control system is disengaged or deemed unsustainable by the Planning module and consequently terminated by the vehicle
9. Planning module sends all information about the current lane position to System Management module for logging

10. Sensor Fusion sends normalized sensor data to System Management module for logging

4.1.4 Light sensors turn on headlights in low-light conditions

Precondition:

- Vehicle system is turned on
- Light sensors detect light levels below 200 lux

Postcondition:

- Headlights are activated
  - If requirements are met, high beams may be activated in accordance with section 3.1.7

Trigger: Light sensors detect that the light level outside the car is less than 200 lux

1. Light sensors constantly send light level data to Sensor Fusion for normalization and then to the Planning module for interpretation.

2. When the Planning module learns that the light level has gone below 200 lux, it sends a command to the VCS to turn the headlights on.
    a. Planning will also determine if high beams should be activated if other conditions apply.
3. Planning module sends information regarding headlight status to the Display module
4. Display module shows headlight status on the display screen.
5. If at any point the Planning module determines that the light level has increased above 200 lux, the Planning module will command the VCS to turn of the headlights and send data to the Display module to update the current headlight status displays

4.1.5 Technician logs in to monitor system status and history

Precondition:

- Vehicle System is turned on
- Technician requests information regarding the vehicle
- System Management is in working order

Postcondition:

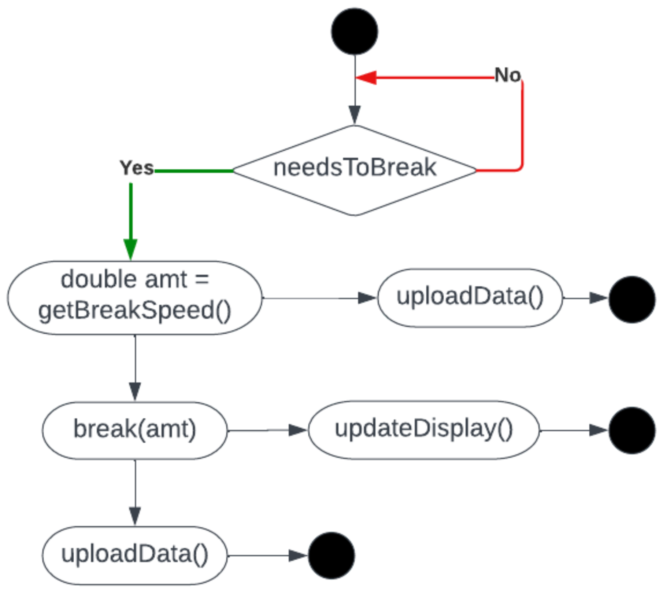
- Technician can observe system status and history

Trigger: Technician logs in to obtain information regarding the system status and history of the vehicle

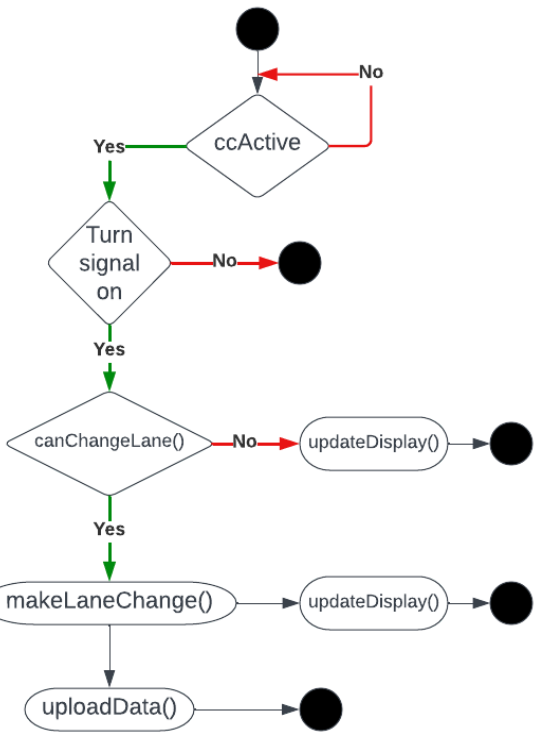
1. Display module prompts technician to enter password and user id.
2. Display module attempts to validate the technician's id and password
    a. If validation fails, technician is prompted again and the counter for number of attempts is incremented. Five subsequent attempts without success will lock the login feature for 30 minutes.
    b. If validation is successful, prompt technician with ability to select module to examine history of
3. Display module puts requested information on the display.
4. After data is displayed, the technician has the ability to logout or two choose other module data to be displayed.
    a. If logout, then the display will return to the login screen
    b. If the technician chooses to continue browsing, the selected module screen will be displayed again on the display module.

4.2 Activity Diagram

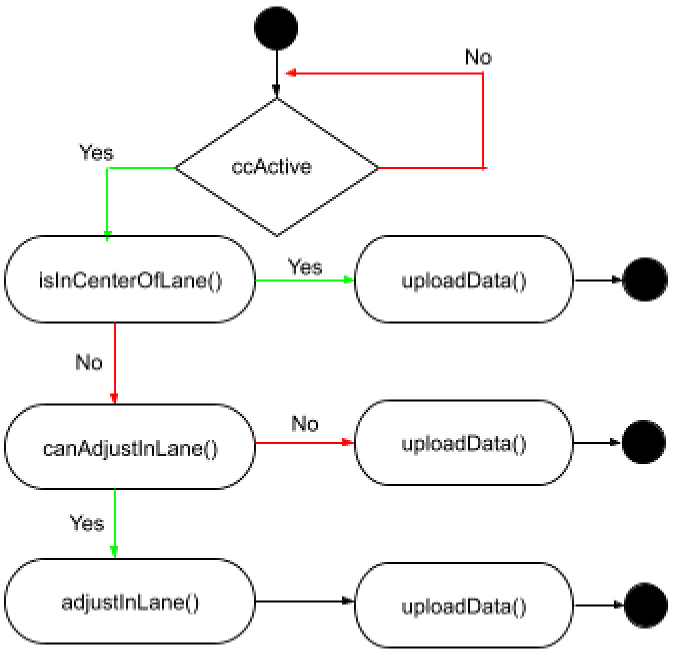
4.2.1 Forward and rear-facing sensors detect object(s) in path of vehicle



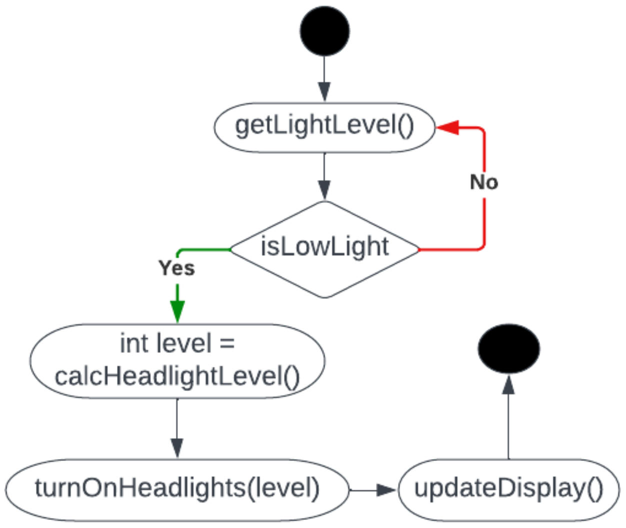
4.2.2 Driver requests automatic lane change



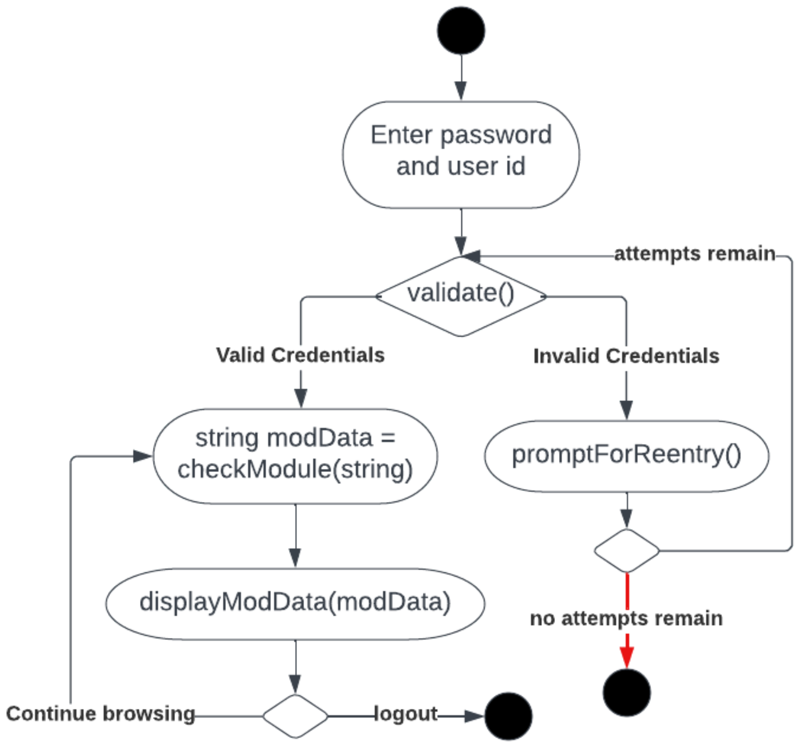
4.2.3 Sensors below the vehicle maintain lane position during Cruise Control



4.2.4 Light sensors turn on headlights in low-light conditions

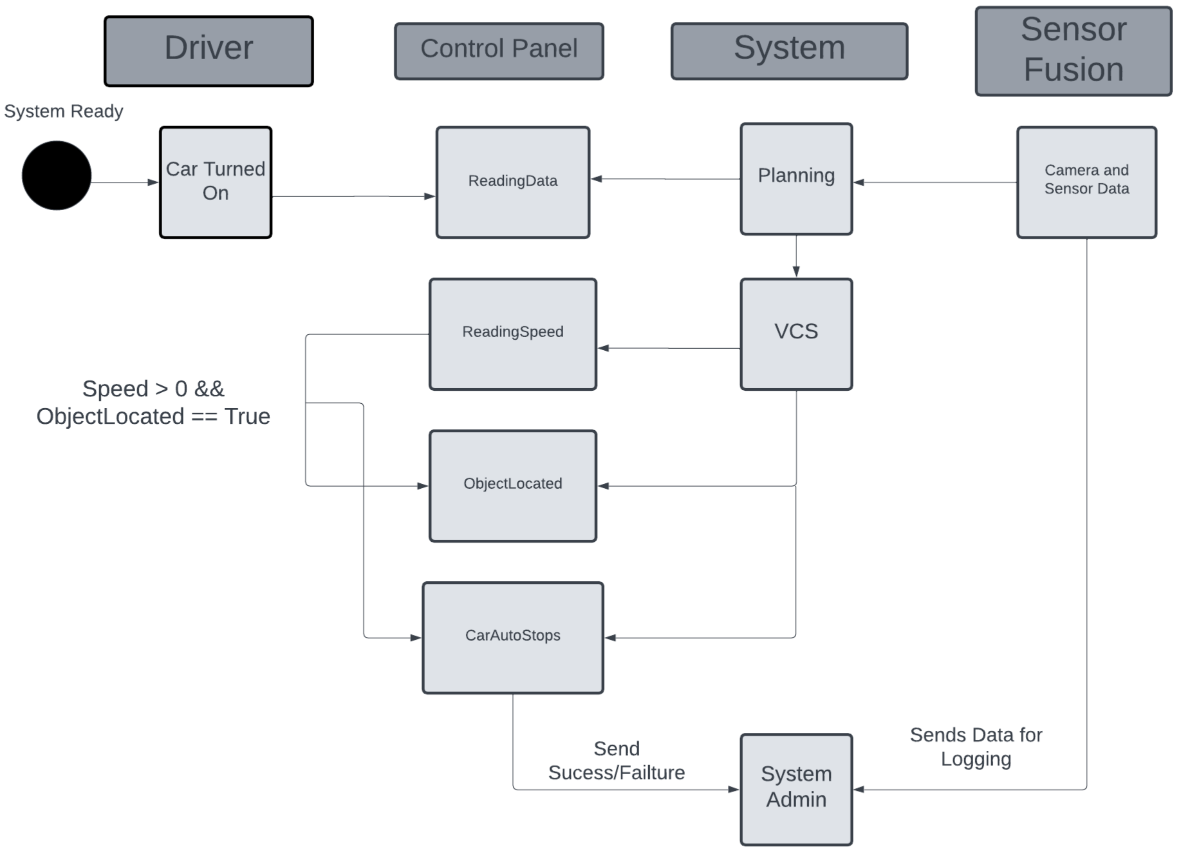


4.2.5 Technician logs in to monitor system status and history

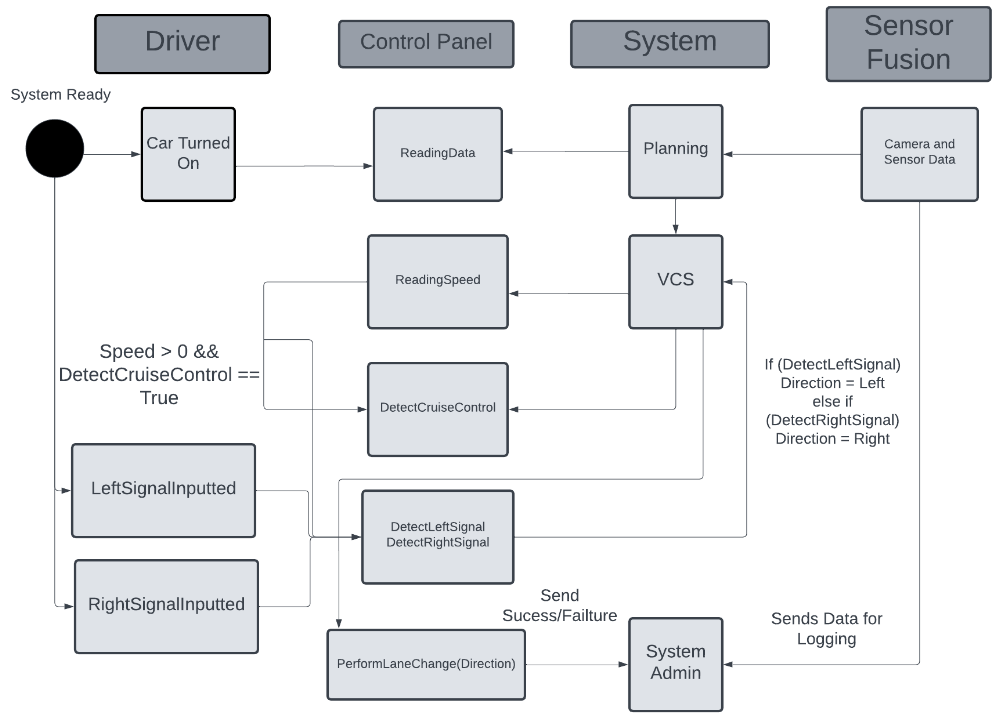


4.3 Sequence Diagrams

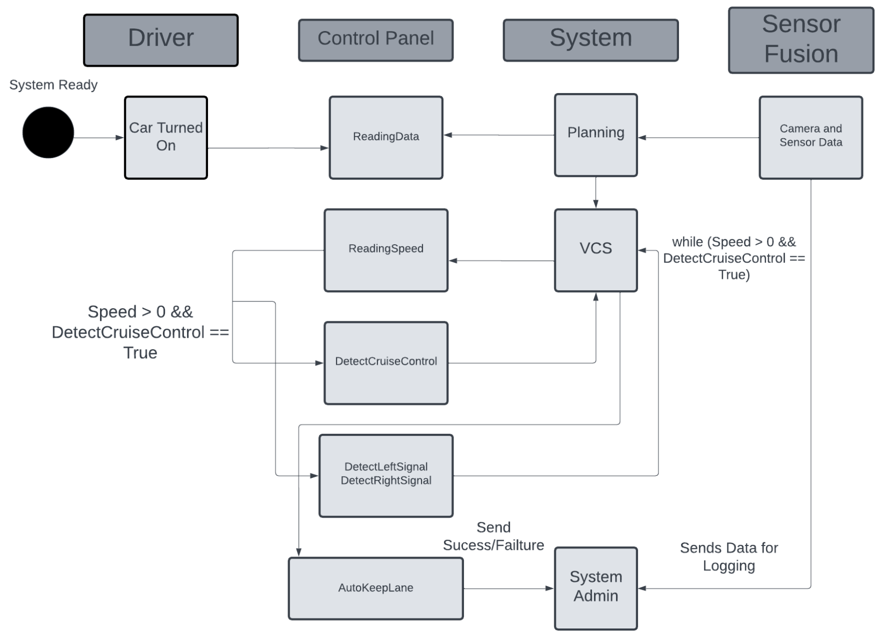
4.3.1 Forward and rear-facing sensors detect object(s) in path of vehicle



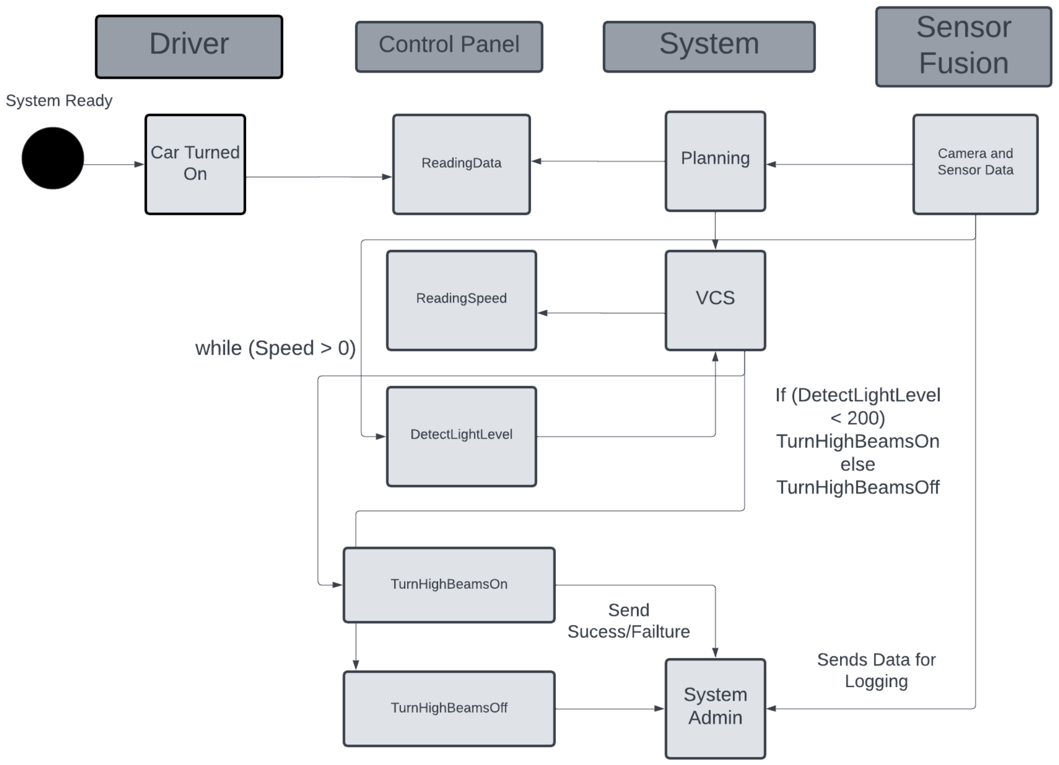
4.3.2 Driver requests automatic lane change



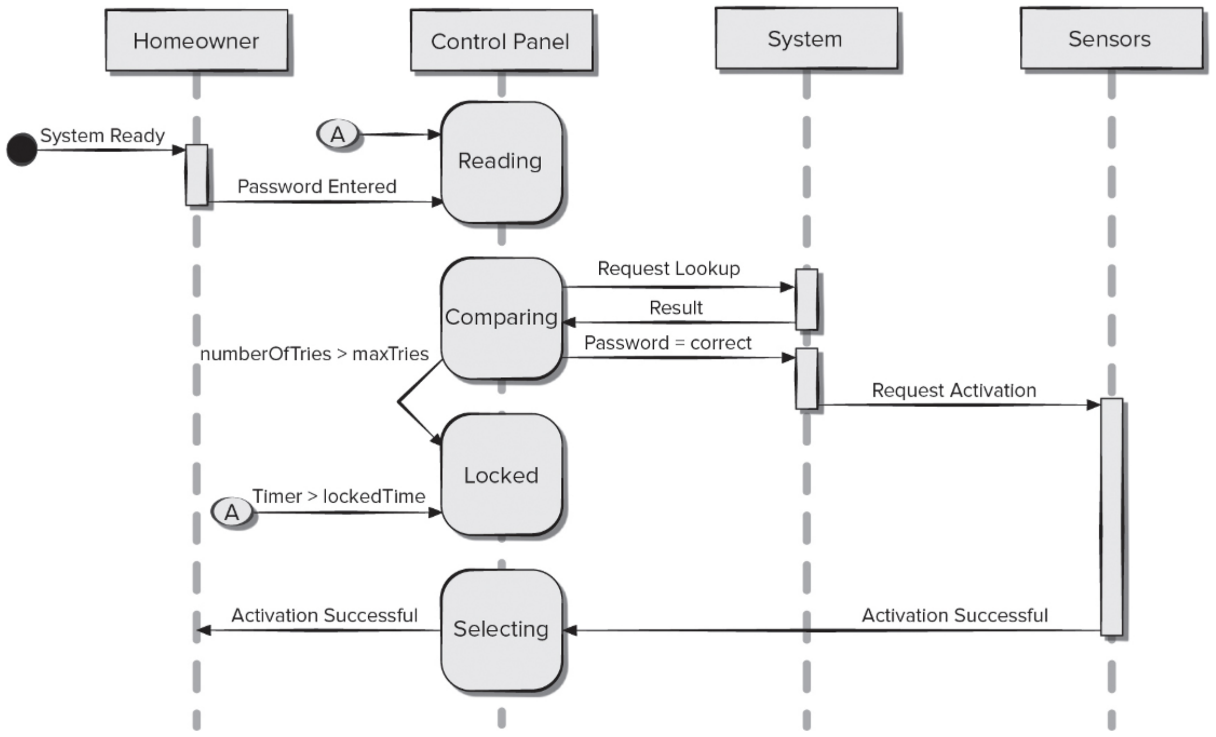
4.3.3 Sensors below the vehicle maintain lane position during Cruise Control



4.3.4 Light sensors turn on headlights in low-light conditions



4.3.5 Technician logs in to monitor system status and history



(Pressman)

4.4 Classes

System Management
loggedIn : bool
validate() : bool **promptForReentry() :** **checkModule(string) : string**

Vehicle Control
isBreaking : bool **isAccelerating : bool**

steeringDegree : double **ccActive : bool** **headlightLevel : int**
applyBreak(double, int) : bool **accelerate(double, int) : bool** **steer(double) : bool** **uploadData() : void** **startCC() : bool** **makeLaneChange() : bool** **turnOnHeadLights(int) : void** **turnOffHeadlights() : void**

Display
displayOn : bool
updateDisplay() : void **displayModData(string) : void**

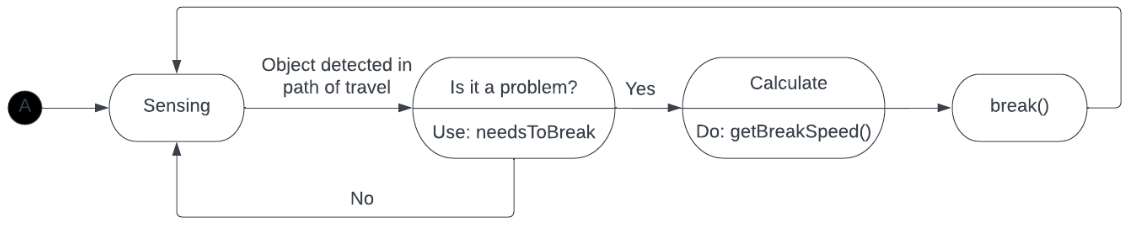
Sensors
lightLevel : double **distanceInFront : double** **distanceBehind : double** **distanceLeft : double** **distanceRight : double** **needsToBreak : bool** **isLowLight : bool**
getSensorData() : void **getLightLevel() : double** **getObjectDistanceInFront() : double** **getObjectDistanceBehind() : double** **getObjectDistanceLeft() : double** **getObjectDistanceRight() : double** **uploadData() : void** **getTimeUntilCollision() : double** **canChangeLane() : bool** **getLightLevel() : double** **calcHeadlightLevel(double) : int** **isInCenterOfLane() : bool**

canAdjustInLane() : bool

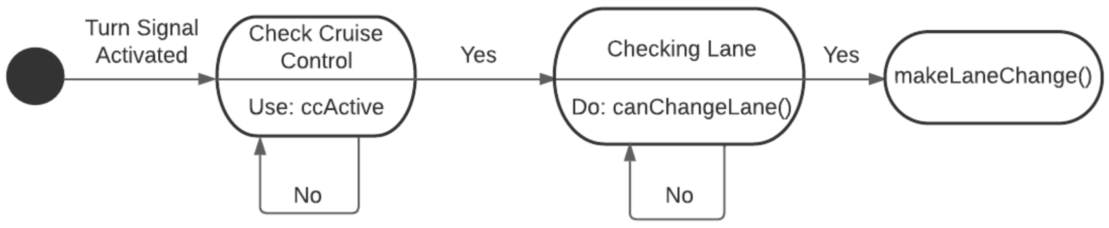
IMU
currentVelocity : double
adjustInLane() : bool **makeInertialMeasurement() : void** **getCurrentVelocity() : double**

4.5 State Diagrams

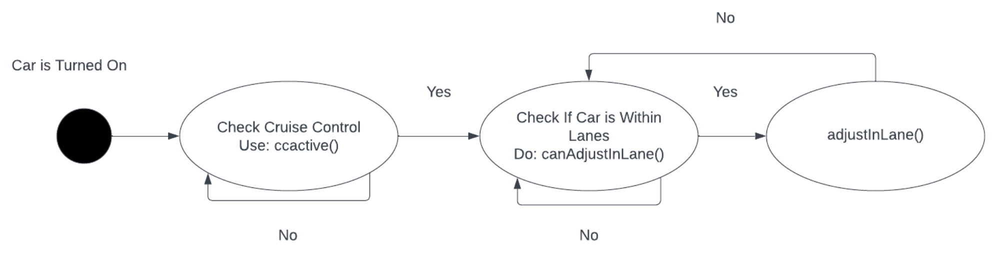
4.5.1 Forward and rear-facing sensors detect object(s) in path of vehicle



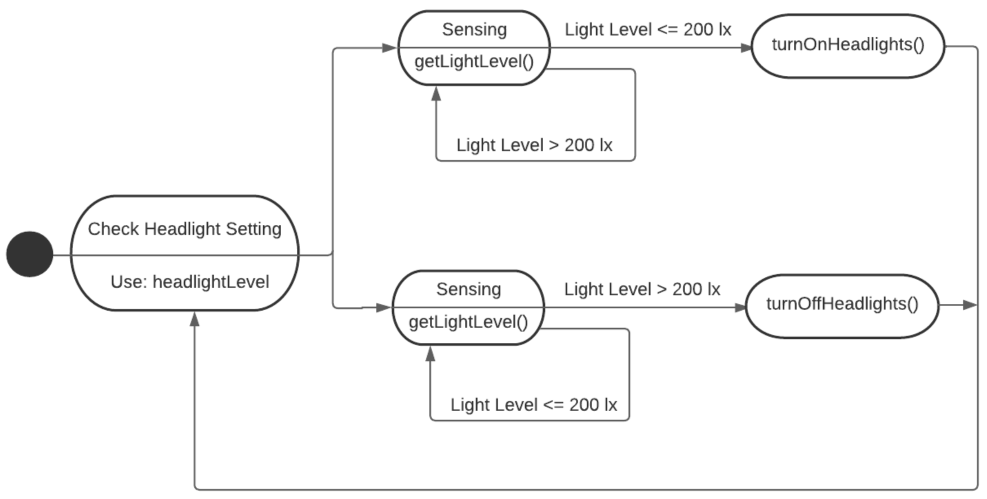
4.5.2 Driver requests automatic lane change



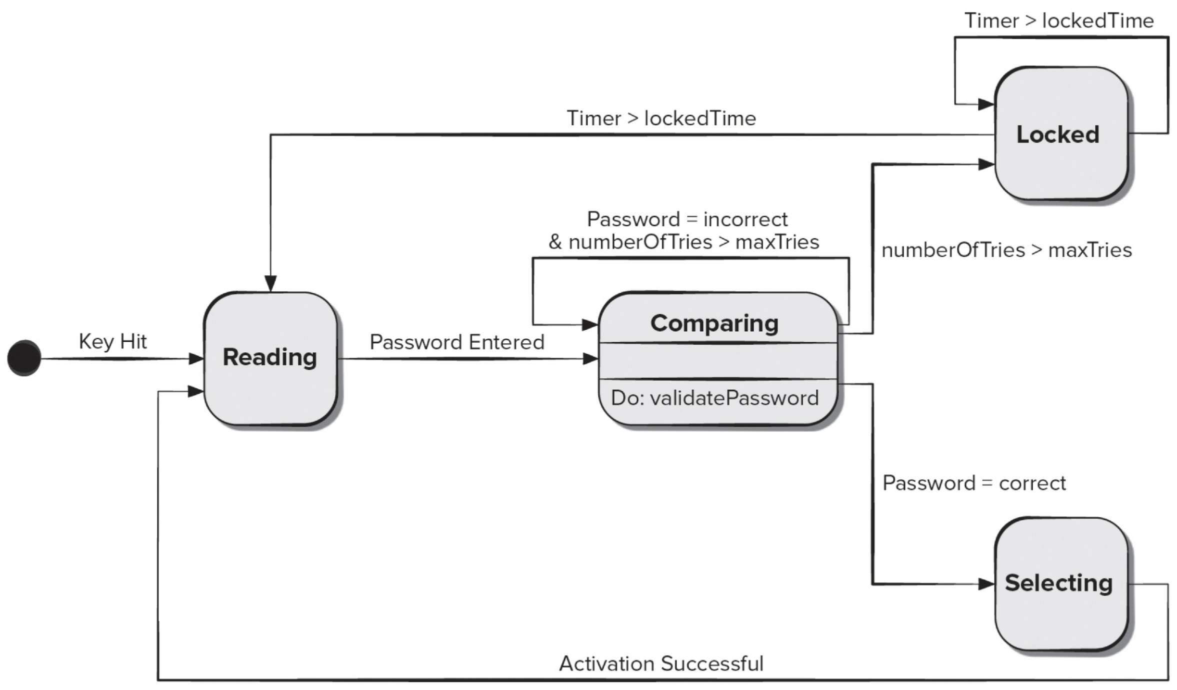
4.5.3 Sensors below the vehicle maintain lane position during Cruise Control



4.5.4 Light sensors turn on headlights in low-light conditions



4.5.5 Technician logs in to monitor system status and history



(Pressman)

5. Design

5.1 Software Architecture

The software architecture can be viewed as one system that manages and interacts with many smaller subsystems. These subsystems include the sensors, the module which transforms sensor data (sensor fusion), the modules which interpret sensor and geographical data (localization/planning), the modules which display information to the vehicle's user, the modules which command motor systems to activate/deactivate, and the systems which move the car themselves. All of these *subsystems* as we have defined above are essentially equivalent to modules that we have defined in previous descriptions of our software architecture. Within these modules, there is not significant hierarchical structure, but instead these modules can be represented by the iterative flow of data between them. One case where a hierarchical relationship does exist is with the sensors. The sensors will each be represented in the code as individual classes, but all sensor data will be made accessible to other modules through a parent class. This will serve as a form of abstraction that prevents modules looking to read from the sensors from seeing any unnecessary data.

The *sensor fusion* class which implements data from the *sensor* class will then interpret that data and store it so that the data may be requested by other classes. Namely, data fields maintained by the *sensor fusion* class will be used by the *localization* class which will use this data and combine it with GPS data to determine a more exact understanding of the vehicle's surroundings. This data is constantly retrieved by the *planning* class which includes many methods dealing with calculating/predicting the potential future movements of the vehicle. *Planning* is arguably the most essential class as it is whose decisions are acted on and therefore is responsible for most major movement of the vehicle. Instead of simply storing data for requests as many other classes so far have done, the *planning* class makes requests to other classes further down the pipeline to perform actions and update other systems' statuses. Most importantly, the *planning* class is what gives the *vehicle control* class meaning. *Planning* sends commands, when it deems it necessary, to the *vehicle control* class which serves as an interface for the vehicle's actual automotive systems (engine, brakes, steering, etc.). This *vehicle control* class is another example of how we use abstraction to provide simple interfaces to allow easier communication between modules. More specifically, the *vehicle control* system commands the vehicle's automotive systems to change state as specified by the *planning* class. Given the above outline of our class relationships, we will provide descriptions and feasibility of various models.

5.1.1 Data Centered Architecture

Description: Data centered architecture uses a shared data source to allow for different components of the project to communicate.

Pros:

- ⇨ Easy to adjust or add to data without changing the code
- ⇨ Set rules and concepts can be maintained in a table rather than throughout a program

Cons:

- ⇨ Looking up data in a table whenever the program is run may cause issues and bugs
- ⇨ If the data table is broken, the components of the project will not work.

Feasibility: A data centered architecture is to some degree feasible for our internet of things (IoT) software product since our system must rely on data to make decisions. However, most decisions will be made based on *real-time* data provided by sensors and cameras. A *centralized* database would be *unnecessary* as the main component of the architecture for our system.

5.1.2 Data Flow Architecture

Description: Data flow architecture uses a series of computational or manipulative components using pipes and filters to transform input data into output data.

Pros:

- ⇨ Reduces development time
- ⇨ Can easily transfer from design and implementation
- ⇨ Can be broken down into subsystems, these subsystems can all work independently to take input data and produce output data

Cons:

- ⇨ Data flow architecture is not well suited for dynamic interactions
- ⇨ Provides high latency and low throughput

Feasibility: A data flow architecture would not be feasible for our IoT software product. The ability to break certain components down into subsystems would be useful and would allow for greater flexibility and testing on individual components. It would also reduce development time which could lead to an increase in actual implementation where many tests could be run. However, data flow architecture is not ideal due to its high latency where our vehicle would have to make split second decisions involving breaking, steering, and other important actions. Data flow architecture is also not ideal for dynamic interactions which would be a hindrance in our software.

5.1.3 Call Return Architecture

Description: Call return architecture divides the main program into a number of components which, in turn, may invoke other program components and subcomponents

Pros:

- ⇨ Code is modular and, thus, it is easy to add and remove certain functionalities without breaking the entire project
- ⇨ Easy to create large scale code iteratively
- ⇨ Easy to modify existing code

Cons:

- ⇨ Different modules can be highly dependent on each other and, thus, core changes to data structures for example may require large scale refactoring
- ⇨ Modules are dependent on each other which can lead to long traces to debug code

Feasibility: A call return architecture is feasible for our internet of things (IoT) software product. There are many different modules in our car which, similar to those in a call return architecture, are dependent on each other. Therefore, we could model our software in a way such that each hardware module corresponds to a software module capable of calling for necessary information. However, there are some instances where our modules would like to use a more two-way, interactive line of communication in which case a call return architecture would have to be slightly modified to allow for this to happen easily.

5.1.4 Object Oriented Architecture

Description: Object Oriented Architecture utilizes the division of responsibilities to reusable, cooperating objects throughout a system.

Pros:

- ⇨ Objects are reusable and prevents repetitive code
- ⇨ Objects can be easily maintained and modified
- ⇨ Objects help with security of data

Cons:

- ⇨ Initial development takes time as object oriented programs are of great size
- ⇨ Object oriented programs can be slower than other programs, depending on the size

Feasibility: An object oriented architecture is feasible for our internet of things (IoT) software product. Objects would be able to handle each of the main components of the vehicle system, which could communicate so that the system could work together. UML diagrams in section 4.4 already provide a basic outline for a class structure that could be used for object oriented design.

5.1.5 Layered Architecture

Description: Layered architecture is an architectural pattern composed of components which are organized in horizontal layers that are independent of each other.

Pros:

- ⇨ Components are interconnected however they do not depend on each other
- ⇨ Modifying one layer of the architecture will not affect downstream layers
- ⇨ Testing is easier due to the separated components and layers allowing for individual testing on components

Cons:

- ⇨ Performance speed decreases as more layers are added
- ⇨ The structure of the framework does not allow for much growth, scalability is difficult

Feasibility: A layered architecture pattern would be feasible for our IoT software product. Due to the breakdown of components into horizontal layers, this architectural pattern would allow users to run a multitude of tests on individual components. This pattern also allows for modifications on layers to be made without affecting any downstream layers as layers are interconnected but do not depend on each other. However, adding more layers would negatively impact performance speed and the growth would be limited due to the framework.

5.1.6 Model View Controller Architecture

Description: Model view controller architecture separates software into three components: the model, the view, and the controller. The model component is used for all the data-related logic that the system uses. The view component is used for all the user interface logic that the system uses. The controller component provides communication between the model and view components.

Pros:

- ⇨ Divides front and back end code into separate components, so components can be developed simultaneously
- ⇨ Easy to modify one component without interfering with the other(s)

Cons:

- ⇨ Framework navigation can be complex due to multiple layers of abstraction
- ⇨ Increased complexity, so bad for small projects

Feasibility: The model view controller architecture is not very feasible for our product because there is not an even division of work between the user interface and the data parts of our software. Our software will run largely without considering information from the user interface, and the model view controller architecture does not give much definition for how this component of the code should be structured within itself, therefore our code will be largely unguided by this model if we were to choose it.

5.1.7 Finite State Machine Architecture

Description: Finite state machine architecture is a programming architecture that allows dynamic flow to states depending on values from previous states or user inputs.

Pros:

- Very flexible
- Easy determination of reachability of a state

Cons:

- Implementation of huge systems is hard without proper organization
- The orders of state conversions are inflexible

Feasibility: The finite state machine architecture would be feasible for our IoT software product. Using a flow-chart like system to make decisions based on user input or previous states would allow the software to make decisions based on data fed. It would allow for a flexible decision making process, however implementing large systems of finite state machines would be difficult to implement without a substantial amount of planning.

5.1.8 Our Architecture

Our team has elected to use object oriented architecture because different vehicle components can be represented using objects which will, in turn, be able to communicate with each other using *real-time* data.

5.2 Interface Design

5.2.1 Technician Interface

Our technician interface will contain more detailed information about the vehicle and its current status. This will mostly be contained on the digital display, which will show the pertinent information about the vehicle including its diagnostics for repair.

Classes: SystemManagement, VehicleControlSystem, Display, IMU, Sensors

Data: (Int) oilLevel, (Int) gasLevel, (Int) washerFluidLevel, (String) carOwner, (Bool) brakeLightDashLight, (Bool) antiLockBrakeDashLight, (Bool) steeringLockDashLight

Functions: getGasLevel(), getAllWarnings(), getCarOwner(), getGasLevel(), getOilLevel(), getWFLevel()

5.2.2 Driver Interface

Our driver interface will contain basic information about the car, including gas level, any lights to indicate issues, current speed, and a digital image showing the car relative to the lanes it is in. This will allow for increased functionality and information for the driver without being too overwhelming or confusing.

Classes: VehicleControlSystem, Display

Data: (Int) gasLevel, (Bool) brakeLightDashLight, (Bool) antiLockBrakeDashLight, (Bool) steeringLockDashLight

Functions: getGasLevel(), getAllWarnings()

5.2.3 Sensor Fusion-Planning Interface

In order to make calculations based on the data captured by the cameras and sensors contained in Sensor Fusion, there must be an interface to communicate between these two modules. This interface is more of a one way line of communication, as

Planning sends the results to other Modules but not back to Sensor Fusion.

Classes: SensorFusion, Planning

Data: (bool) objectDetected, (double) DistanceToObject, (double) lightLevel (double) distanceInFront, (double) distanceBehind, (double) distanceLeft, (double) (double) distanceRight

Functions: (double) getSensorData(), (double) getLightLevel(), (double) getObjectDistanceInFront(), (double) getObjectDistanceBehind() (double) getObjectDistanceLeft(),(double) getObjectDistanceRight(), (void) uploadData(), (double) getTimeUntilCollision(), (bool) canChangeLane(), (double) getLightLevel(), (int) calcHeadlightLevel(double), (bool) isInCenterOfLane()

5.2.4 Vehicle Control-Planning Interface

The Vehicle Control System (VCS) is another essential component for making calculations regarding the vehicle and its surroundings. This is why the information must again be acquired from the VCS before being sent to Planning for calculating. After the calculations have been made, Planning sends the results back to the VCS in order to perform certain actions (i.e. applying the break, steering, etc.)

Classes: VehicleControlSystem, Planning

Data: (double) velocity, (bool) isBreaking (bool) isAccelerating (double) steeringDegree, (bool) ccActive, (int) headlightLevel

Functions: (double) getVelocity(), (bool) applyBreak(double), (bool) accelerate(double), (bool) steer(double) (void) uploadData() (bool) startCC(), (bool) makeLaneChange(), (void) turnOnHeadLights(int), (void) turnOffHeadlights()

5.2.5 IMU-Planning Interface

Once again, the IMU Module is used to attain data regarding the vehicle’s current status, including the vehicle’s current velocity. This information is then once again sent to planning to make calculations and perform required tasks. This information is then sent back to the IMU to perform certain actions, i.e. adjusting in the lane.

Classes: IMU, Planning

Data: (double) currentVelocity

Functions: (bool) adjustInLane(), (void) makeIntertialMeasurement(), (double) getCurrentVelocity()

5.2.6 Planning-Display Interface

Once the information has been obtained from the previous interfaces and the necessary calculations and actions have taken place, certain information must be displayed to the user. This is done through the Display Module.

Classes: Planning, Display

Data: (bool) displayOn

Functions: (void) updateDisplay(), (void) displayModData(string)

5.3 Component-Level Design

5.3.1 Introduction

In this section, we will provide outlines of the classes that will be used in our software design. Each class shows the data fields and methods that it contains. Following each class, there will be a brief description of the class and its contents that will explain the purpose of this class in the larger architecture and the use of each of its contents.

5.3.2 System Management

```
class SystemManagement {

        private:

        bool loggedIn;
        string user;
        string username;
        string password;

        map<string,string> passwords;
        map<string,string> users;

        void init_credentials();

        public:

        SystemManagement();

        void login_prompt();

        string get_user();
 };
```

The SystemManagement class is used to check basic information about the status of the car's software. This includes whether the user is logged in, as well as the function login_prompt() which prompts the user for a username and password and validates the inputs.

5.3.3 Vehicle Control

```
 class VehicleControl {

     private:

     bool ccActive;               // true is on, false is off
     bool windshieldWipers;       // true is on, false is off

     int headlightLevel;          // 0 is off, 1 is on, 2 is high beams
     int gear;                    // 0 P, 1 R, 2 N, 3 D
     int turnSignal;              // -1 left, 0 none, 1 right

     public:
```

```
    VehicleControl();

    VehicleControl(bool cc, bool inDrive);

    void startCC(IMU &imu, GPS &gps);
    void stopCC();

    void setGear(int val);

    void turnOffHeadlights();
    void turnOnHeadLights(int level);

    void leftTurnSignal();
    void rightTurnSignal();
    void turnComplete();

    void turnOnWindshieldWipers(bool wipers);

    bool getccActive();

    int getHeadLightLevel();

    int getGear();

    int getTurn();

    bool windshieldWipersOn();

    void brake(IMU &imu, SensorsAndCameras &sensorsAndCameras, int intensity);
    void brakeTo(IMU &imu, SensorsAndCameras &sensorsAndCameras, int speed);

    void accelerateTo(IMU &imu, SensorsAndCameras &sensorsAndCameras, int speed);
};
```

The VehicleControl class is used to influence decisions about the vehicle. This includes basic functions such as breaking, accelerating, gear shifting, and managing the headlights and windshield wipers. It also controls if the vehicle is in cruise control mode.

5.3.4 Display

```
class Display {

private:

        struct status_struct status;

public:

        Display();

        void set_status(status_struct stat);

        void set_speed(int speed);

        void set_gear(int gear);

        void set_braking(int brakeForce);

        void set_cruise_control_active(bool active);

        void set_wipers(bool on);

        void set_cars_in_front(bool in_front);

        void set_cars_in_back(bool in_back);

        void set_cars_on_left(bool on_left);

        void set_cars_on_right(bool on_right;

        void set_lane_warning(int warning);

        void set_headlights(int level);

        void set_rearview(bool needsRearView);

        void set_lane(int lane);

        void set_num_lanes(int num_lanes);

        void print_display();
};
```

The Display class is used to display the data from specific modules. It will show data that the vehicle collects, including all of the classes within sensor fusion and the vehicle control. Planning will constantly update the display using the set functions so it has accurate data. print_display(), as the name suggests, prints the display.

5.3.5 Sensors and Cameras

```
class SensorsAndCameras {

    private:

    double lightLevel;          // light level outside
    double distanceInFront;     // distance of car in front
    double distanceBehind;      // distance of car behind

    bool objectRight;       // true if object to right, false if not
    bool objectLeft;        // true if object to left, false if not
    bool rainDetected;      // true if its raining, false if its not

    public:

    SensorsAndCameras();

    void setLightLevel(double level);

    void setDistanceInFront(double distance);

    void setDistanceBehind(double distance);

    void setObjectRight(bool value);

    void setObjectLeft(bool value);

    void setRain(bool value);

    double getLightLevel();

    double getDistanceInFront();
    double getDistanceBehind();

    bool isObjectRight();

    bool isObjectLeft();

    bool getRainDetected();
 };
```

The SensorsAndCameras class, a component of sensor fusion, is used to receive and hold data of the sensor and camera system of the vehicle. Sensors can be set using the set methods and can be accessed with the get methods. SensorsAndCameras stores the light level, the distance of cars in front and behind, if there is a car to the left or right, and if it is raining.

5.3.6 IMU

```
 class IMU {

      private:

      double currentVelocity;

      public:

      IMU();

       IMU(double velo);

      double getCurrentVelocity();

      void setCurrentVelocity(double velo);

 };
```

The IMU class, another component of sensor fusion, contains information about the vehicle’s current velocity. Planning uses the data stored by the IMU to make decisions for the vehicle.

5.3.7 Scanners

```
 class Scanners {

     private:

     double lane_width;    // total lane width of the current road
     double right_line;    // distance to right line of lane
     double left_line;     // distance to left line of lane

     bool marked_road;     // if the lanes are marked on the current road

     public:

     Scanners();

     Scanners( double width, double right, double left, bool marked);

     void setLaneWidth(double width);

     void setMarkedRoad(bool marked);

     double getLaneWidth();

     double distanceFromLineRight();

     double distanceFromLineLeft();

     bool onMarkedRoad();
 };
```

The Scanners class, the third component of sensor fusion, stores information about the lane that the vehicle is driving in. Specifically, it holds the total width of the lane that the vehicle is driving in, and the distance of the vehicle from each line to the right and the left. It also stores if the vehicle is on a marked road with distinct lanes or not.

5.3.8 GPS

```
class GPS {

    private:

    bool onHighway;             // true if on a highway, false if not
    bool onLocalRoute;          // true if on a local road, false if not

    int numberOfLanes;          // number of lanes on the current road
    int laneNumber;             // the lane the car is in on the current road

    public:

    GPS();

    GPS(bool H, bool L, int num_lanes, int lane);

    void setOnHighway();

    void setOnLocalRoad();

    void setOnUnregisteredRoad();

    void setNumberOfLanes(int num);

    void setLaneNumber(int lane);

    bool isOnHighway();

    bool isOnLocalRoute();

    bool isOnUnregisteredRoad();

    int getNumberOfLanes();

    int getLaneNumber();

};
```

The GPS class, the last component of sensor fusion, stores data about the road that the vehicle is currently on. It stores if the vehicle is on a highway, local road, or unregistered road. It also stores the number of lanes the current road has and what lane the vehicle is in.

5.3.9 Planning

```
 class Planning {

     private:

     VehicleControl vehicleControl;
     IMU imu;
     Scanners scanners;
     GPS gps;
     SensorsAndCameras sensorsAndCameras;
     Display display;

     bool wantsToAcc;        // when car is told to accelerate
     bool wantsToBrk;        // when car is told to break
     int speed_wanted;       // the speed to accelerate or break to

     public:

     Planning();

     /* vehicle control updates */

     void brakeWhenObjectDetected();

     void automaticallyChangeLane();

     void automaticHeadLights();

     void automaticaHighBeams();

     void automaticWindshieldWipers();

     void gearControl();

     void acc();
```

```
    void brk();

    void check_all();

    /* updates display */

    void automaticObjectDetection();

    void wipersOn();

    void headlightLevel();

    void currentSpeed();

    void automaticRearCamera();

    void checkLanes();

    void checkTurn();

    void detectLaneDeparture();

    void checkGear();

    void checkCC();

    void checkWarnings();

    void updateDisplay();

    /* runs the system */

    void run_systems();
};
```

The Planning class is what the vehicle is run through. Planning has an instance of every sensor fusion class, the display class, and the vehicle class. Planning can be broken up into 3 parts. The first section updates the vehicle control when Planning determines it must. The second section updates the display based on changes to the vehicle and in sensor fusion. Finally, the last section runs the car and takes in inputs from the driver and changes the outside environment as the vehicle runs.

6. Project Code

6.1 system.cpp

```
#include <iostream>
#include <map>
#include <string>

#include "vehicle.cpp"

using namespace std;

class SystemManagement {

    private:

    bool loggedIn;
    string user;

    string username;
    string password;

    map<string,string> passwords;
    map<string,string> users;

    void init_credentials()
    {
        passwords["lhope"] = "171717";
        passwords["falcon"] = "goblin";
        passwords["jlee"] = "who?";
        passwords["danny"] = "dimez";

        users["lhope"] = "Lucas Hope";
        users["falcon"] = "Steven DeFalco";
        users["jlee"] = "Jude Lee";
        users["danny"] = "Daniel Storms";
    }
    public:

    SystemManagement() {
        loggedIn = false;
        init_credentials();
    }

    void login_prompt()
    {
        while(!loggedIn)
        {
            cout << "\033[2J\033[1;1H";
            cout << "Vehicle System Management" << endl;
            cout << "Username: ";
            cin >> username;
            cout << "Password: ";
            cin >> password;

            if(passwords.count(username) == 0) {
                cout << "Login Failed: Invalid username." << endl;
            } else if(passwords[username].compare(password) == 0) {
                loggedIn = true;
                user = users[username];
            } else {
                cout << "Login Failed: Invalid password." << endl;
            }
        }
    }

    string get_user() { return user; }

};

int main(int argc, char *argv[]) {

    int total_steps = 100;
    int curr_step = 0;
    SystemManagement system;
    system.login_prompt();
    cout << "\nWelcome " << system.get_user() << endl;
    while (curr_step <= total_steps) {
        float percent_complete = static_cast<float>(curr_step) /
total_steps * 100;
        std::cout << "[";
        int bar_width = 30;
        int num_symbols = static_cast<int>(percent_complete / 100 *
bar_width);
        for (int i = 0; i < bar_width; ++i) {
            if (i < num_symbols) {
                std::cout << "=";
            } else {
                std::cout << " ";
            }
        }
        std::cout << "] " << percent_complete << "%\r";
        if (percent_complete == 100) {

std::this_thread::sleep_for(std::chrono::milliseconds(1000));
        } else {

std::this_thread::sleep_for(std::chrono::milliseconds(30));
        }
        curr_step++;
        std::cout.flush();
    }
    std::this_thread::sleep_for(std::chrono::milliseconds(500));

    Planning running_vehicle = Planning();

    running_vehicle.run_systems();

}
```

System.cpp holds the main function of the program. System.cpp prompts for login credentials and if provided with a correct username and password, it will start an instance of planning, which will run the vehicle and all of its components.

6.2 vehicle.cpp

```
#include <string>
#include <iostream>
#include <cstring>
#include <chrono>
#include <thread>
#include <iomanip>
#include <csignal>
#include <limits.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "vehicle.hpp"


/* Signal handling for test inputs */

volatile bool wantsEnvironmentInput;
volatile bool wantsVehicleInput;

void environment_handler(int signum) {
    wantsEnvironmentInput = true;
}

void vehicle_handler(int signum) {
    wantsVehicleInput = true;
}


/* Sensor Fusion */

class IMU {
    private:

    double currentVelocity;    // current speed of the vehicle

    public:

    IMU() {
        currentVelocity = 0;
    }

    IMU(double velo) {
        currentVelocity = velo;
    }

    double getCurrentVelocity() { return currentVelocity; }

    void setCurrentVelocity(double velo) { currentVelocity = velo; }

};


class Scanners {

    private:

    double lane_width;    // total lane width of the current road
    double right_line;    // distance to right line of lane
    double left_line;     // distance to left line of lane

    bool marked_road;     // if the lanes are marked on the current
road

    public:

    Scanners() {
        lane_width = 8;
        right_line = 1;
        left_line = 1;
        marked_road = false;
    }

    Scanners( double width, double right, double left, bool marked) {
        lane_width = width;
        right_line = right;
        left_line = left;
        marked_road = marked;
    }

     void setLaneWidth(double width) {
        if(!marked_road) return;
        this->lane_width = width > 7 ? width : 7;
        this->right_line = (width - 6) / 2;
        this->left_line = (width - 6) / 2;
    }

    void setMarkedRoad(bool marked) { this->marked_road = marked; }

    double getLaneWidth() {
        if(!marked_road) return -1.0;
        return this->lane_width;
    }

    double distanceFromLineRight() {
        if(!marked_road) return -1.0;
        return this->right_line;
    }

    double distanceFromLineLeft() {
        if(!marked_road) return -1.0;
        return this->left_line;
    }

    bool onMarkedRoad() { return this->marked_road; }

};


class GPS {
    private:

    bool onHighway;             // true if on a highway, false if not
    bool onLocalRoute;          // true if on a local road, false if
not

    int numberOfLanes;          // number of lanes on the current road
    int laneNumber;             // the lane the car is in on the
current road (1 is lane furthest left)

    public:

    GPS() {
        onHighway = false;
        onLocalRoute = false;
        numberOfLanes = 1;
        laneNumber = 1;
    }

    GPS(bool H, bool L, int num_lanes, int lane) {
        if(H && L) {
            onHighway = false;
            onLocalRoute = false;
        } else {
            onHighway = H;
            onLocalRoute = L;
        }
        numberOfLanes = num_lanes > 0 ? num_lanes : 1;
        if(lane < 1) laneNumber = 1;
        else if (lane > numberOfLanes) laneNumber = numberOfLanes;
        else laneNumber = lane;
    }

    void setOnHighway() {
        this->onHighway = true;
        this->onLocalRoute = false;
    }
    void setOnLocalRoad() {
        this->onHighway = false;
        this->onLocalRoute = true;
    }

    void setOnUnregisteredRoad() {
        this->onHighway = false;
        this->onLocalRoute = false;
    }

    void setNumberOfLanes(int num) { if(num > 0) this->numberOfLanes
= num; }

    void setLaneNumber(int lane) { if(lane > 0 && lane <=
this->numberOfLanes) this->laneNumber = lane; }

    bool isOnHighway() { return this->onHighway; }

    bool isOnLocalRoute() { return this->onLocalRoute;}

    bool isOnUnregisteredRoad() { return !this->onHighway &&
!this->onLocalRoute; }

    int getNumberOfLanes() { return this->numberOfLanes; }

    int getLaneNumber() { return this->laneNumber; }

};


class SensorsAndCameras {

    private:

    double lightLevel;          // light level outside
    double distanceInFront;     // distance of car in front
    double distanceBehind;      // distance of car behind

    bool objectRight;       // true if object to right, false if not
    bool objectLeft;        // true if object to left, false if not
    bool rainDetected;      // true if its raining, false if its not

    public:

    SensorsAndCameras() {
        this->lightLevel = 200;
        this->distanceInFront = INT_MAX;
        this->distanceBehind = INT_MAX;
        this->objectRight = false;
        this->objectLeft = false;
        this->rainDetected = false;
    }

    void setLightLevel(double level) { this->lightLevel = level; }

    void setDistanceInFront(double distance) { this->distanceInFront
= distance; }

    void setDistanceBehind(double distance) { this->distanceBehind =
distance; }

    void setObjectRight(bool value) { this->objectRight = value; }

    void setObjectLeft(bool value) { this->objectLeft = value; }

    void setRain(bool value) { this->rainDetected = value; }

    double getLightLevel() const { return this->lightLevel; }

    double getDistanceInFront() const { return this->distanceInFront;
}

    double getDistanceBehind() const { return this->distanceBehind; }

    bool isObjectRight() const { return this->objectRight; }

    bool isObjectLeft() const { return this->objectLeft; }
    bool getRainDetected() { return this->rainDetected; }

};


/* Vehicle Control */

class VehicleControl {

    private:

    bool ccActive;              // true is on, false is off
    bool windshieldWipers;      // true is on, false is off

    int headlightLevel;         // 0 is off, 1 is on, 2 is high beams
    int gear;                   // 0 P, 1 R, 2 N, 3 D
    int turnSignal;             // -1 left, 0 none, 1 right

    public:

    VehicleControl() {
        this->ccActive = false;
        this->headlightLevel = 0;
        this->gear = 0;
        this->windshieldWipers = false;
        this->turnSignal = 0;
    }

    VehicleControl(bool cc, bool inDrive) {
        this->ccActive = cc;
        this->headlightLevel = 0;
        this->gear = inDrive ? 3 : 0;
        this->windshieldWipers = false;
        this->turnSignal = 0;
    }

    void startCC(IMU &imu, GPS &gps) { if(gps.isOnHighway() &&
imu.getCurrentVelocity() > 0) ccActive = true; }
    void stopCC() { ccActive = false; }
    void setGear(int val) { if (val >= 0 && val <= 3) this->gear =
val; }

    void turnOffHeadlights() { this->headlightLevel = 0; }
    void turnOnHeadLights(int level) {
        if(level == 0) this->headlightLevel = 0;
        if(level == 1) this->headlightLevel = 1;
        if(level > 1) this->headlightLevel = 2;
    }

    void leftTurnSignal() { this->turnSignal = -1; }
    void rightTurnSignal() { this->turnSignal = 1; }
    void turnComplete() { turnSignal = 0; }

    void turnOnWindshieldWipers(bool wipers) { this->windshieldWipers
= wipers; }

    bool getccActive() { return this->ccActive; }

    int getHeadLightLevel() { return this->headlightLevel; }

    int getGear() { return this->gear; }

    int getTurn() { return this->turnSignal; }

    bool windshieldWipersOn() { return this->windshieldWipers; }

    void brake(IMU &imu, SensorsAndCameras &sensorsAndCameras, int
intensity) {  // intensity = 1, 2, 3,  - for if in reverse
        bool reversing = imu.getCurrentVelocity() < 5;
        if(intensity == 1) {
            imu.setCurrentVelocity(imu.getCurrentVelocity() * .95);
            if(this->gear == 3 || this->gear == 2 || !reversing) {
                if(imu.getCurrentVelocity() < 5)
imu.setCurrentVelocity(0);
                else
sensorsAndCameras.setDistanceInFront(sensorsAndCameras.getDistanceInF
ront() + 10);
```

```
            } else if (this->gear == 1) {
                 if(imu.getCurrentVelocity() > -5)
imu.setCurrentVelocity(0);
            }
        } else if( intensity == 2) {
            imu.setCurrentVelocity(imu.getCurrentVelocity() * .90);
            if(this->gear == 3 || this->gear == 2 || !reversing ) {
                if(imu.getCurrentVelocity() < 5)
imu.setCurrentVelocity(0);
                else
sensorsAndCameras.setDistanceInFront(sensorsAndCameras.getDistanceInF
ront() + 15);
            } else if (this->gear == 1) {
                 if(imu.getCurrentVelocity() > -5)
imu.setCurrentVelocity(0);
            }
        } else {
            imu.setCurrentVelocity(imu.getCurrentVelocity() * .85);
            if(this->gear == 3 || this->gear == 2 || !reversing) {
                if(imu.getCurrentVelocity() < 5)
imu.setCurrentVelocity(0);
                else
sensorsAndCameras.setDistanceInFront(sensorsAndCameras.getDistanceInF
ront() + 20);
            } else if (this->gear == 1) {
                 if(imu.getCurrentVelocity() > -5)
imu.setCurrentVelocity(0);
            }
        }
    }

    void brakeTo(IMU &imu, SensorsAndCameras &sensorsAndCameras, int
speed) {
        brake(imu, sensorsAndCameras, 2);
        if((imu.getCurrentVelocity() <= speed && this->gear == 3) ||
            imu.getCurrentVelocity() >= speed && this->gear == 1) {
                imu.setCurrentVelocity(speed);
        }
        if(speed < 5 && this->gear == 3 && imu.getCurrentVelocity() <
5) imu.setCurrentVelocity(speed);
        if(speed > -5 && this->gear == 1 && imu.getCurrentVelocity()
> -5) imu.setCurrentVelocity(speed);
    }

    void accelerateTo(IMU &imu, SensorsAndCameras &sensorsAndCameras,
int speed) {
        if(imu.getCurrentVelocity() <= 10 && this->gear == 3 )
imu.setCurrentVelocity(10 * 1.2);
        else if (imu.getCurrentVelocity() >= -10 && this->gear == 1 )
imu.setCurrentVelocity(-10 * 1.2);
        else imu.setCurrentVelocity(imu.getCurrentVelocity() * 1.10);
        if(this->gear == 3) {

sensorsAndCameras.setDistanceInFront(sensorsAndCameras.getDistanceInF
ront() - 10);

sensorsAndCameras.setDistanceBehind(sensorsAndCameras.getDistanceBehi
nd() + 10);
            if(imu.getCurrentVelocity() >= speed) {
                imu.setCurrentVelocity(speed);
            }
        } else if (this->gear == 1) {

sensorsAndCameras.setDistanceInFront(sensorsAndCameras.getDistanceInF
ront() + 10);
            if(imu.getCurrentVelocity() <= speed) {
                imu.setCurrentVelocity(speed);
            }
        }
    }

};


/* Display */

class Display {
    private:

    struct status_struct status;

    public:

    Display() :

status{0,0,false,false,false,false,false,false,-1,0,false,1,1,false,f
alse}
    {}

    void set_status(status_struct stat) { status = stat; }

    void set_speed(int speed) { status.speed = speed; }

    void set_gear(int gear) { status.gear = gear; }

    void set_cruise_control_active(bool active) {
status.cruise_control_active = active; }

    void set_wipers(bool on) { status.wipers_on = on; }

    void set_cars_in_front(bool in_front) { status.cars_in_front =
in_front; }

    void set_cars_in_back(bool in_back) { status.cars_in_back =
in_back; }

    void set_cars_on_left(bool on_left) { status.cars_on_left =
on_left; }

    void set_cars_on_right(bool on_right) { status.cars_on_right =
on_right; }

    void set_lane_warning(int warning) { status.lane_warning =
warning; }

    void set_headlights(int level) { status.headlights = level; }
    void set_rearview(bool needsRearView) { status.rear_view =
needsRearView; }

    void set_lane(int lane) { status.lane = lane; }

    void set_num_lanes(int num_lanes) { status.num_lanes = num_lanes;
}

    void set_right_turn(bool right_turn) { status.rightTurn =
right_turn; }

    void set_left_turn(bool left_turn) { status.leftTurn = left_turn;
}

    void print_display() {

        // Clear the console
        std::cout << "\033[2J\033[1;1H";

        std::cout << "\n
Alset-IoT Simulation:\n
Ctrl+C to change the environment\n
Ctrl+Z to make a vehicle input\n
-1 after sending signal to exit\n";

        std::cout <<
                "\n
" << status.speed << " mph\n\n";

        if (status.gear == 0) {
        std::cout <<
                "
[P]ark\n\n";
        } else if (status.gear == 1) {
        std::cout <<
                "
[R]everse\n\n";
        } else if (status.gear == 2) {
```

```
        std::cout <<
                "
[N]eutral\n\n";
        } else if (status.gear == 3) {
        std::cout <<
                "
[D]rive\n\n";
        }


        if (status.cars_in_front) {
        std::cout <<
                "
|       CAR HERE        |\n";
        } else {
        std::cout <<
                "
|                       |\n";
        }

        if (status.lane_warning == 0) {
        std::cout <<
                "                        ALERT! Lane Change Warning
|                       |\n";
        } else if (status.lane_warning == 1) {
        std::cout <<
                "
|                       | ALERT! Lane Change Warning\n";
        } else {
        std::cout <<
                "
|                       |\n";
        }

        if (status.headlights == 2) {
        std::cout <<
                "
|      \\   / \\   /     |\n";
        } else {
```

```
        std::cout <<
                "
|                    |\n";
        }

        if (status.headlights == 1 || status.headlights == 2) {
        std::cout <<
                "
|      \\ /   \\ /     |\n";
        } else {
        std::cout <<
                "
|                    |\n";
        }


        std::cout <<
                "
|      --------      |\n";

        std::cout <<
                "
|    (|        |)    |\n";

        if (status.leftTurn) {
            std::cout <<
                "
| <-- |        |     |\n";
        } else if (status.rightTurn) {
            std::cout <<
                "
|     |        | --> |\n";
        } else {
            std::cout <<
                "
|     |        |     |\n";
        }

        if (status.cars_on_left && status.cars_on_right) {
            std::cout <<
                "                                     CAR
|     |        |     |      CAR\n";
        } else if (status.cars_on_left) {
            std::cout <<
                "                                     CAR
|     |        |     |\n";
        } else if (status.cars_on_right) {
            std::cout <<
                "
|     |        |     |      CAR\n";
        } else {
            std::cout <<
                "
|     |        |     |\n";
        }

        if (status.cars_on_left && status.cars_on_right) {
            std::cout <<
                "                                     HERE
|     |        |     |      HERE\n";
        } else if (status.cars_on_left) {
            std::cout <<
                "                                     HERE
|     |        |     |\n";
        } else if (status.cars_on_right) {
            std::cout <<
                "
|     |        |     |      HERE\n";
        } else {
            std::cout <<
                "
|     |        |     |\n";
        }

        std::cout <<
                "
|    (|        |)    |\n";
```

```
        std::cout <<
                "
|      --------      |\n";

        std::cout <<
                "
|                    |\n";

        std::cout <<
                "
|                    |\n";

        if (status.cars_in_back) {
            std::cout <<
                "
|      CAR HERE      |\n";
        } else {
            std::cout <<
                "
|                    |\n";
        }

        std::cout <<
                "
|                    |\n";

        std::cout <<
                "
|      lane: " << status.lane << "       |\n";

        std::cout <<
                "
|                    |\n\n";

        if (status.cruise_control_active) {
            std::cout <<
                "
Cruise Control Active\n\n";
        }
        if (status.wipers_on) {
            std::cout <<
                    "
Wipers on\n";
        }

        if (status.rear_view && status.cars_in_back) {
        std::cout <<
                "\n
--------------------\n";
        std::cout <<
                "
|  Rear View Camera  |\n";
        std::cout <<
                "
|      CAR HERE      |\n";
        std::cout <<
                "
|                    |\n";
        std::cout <<
                "
--------------------\n";
        } else if (status.rear_view && !status.cars_in_back) {
        std::cout <<
                "\n
--------------------\n";
        std::cout <<
                "
|  Rear View Camera  |\n";
        std::cout <<
                "
|                    |\n";
        std::cout <<
                "
|                    |\n";
        std::cout <<
                "
--------------------\n";
```

```
        }
    }

};


/* Planning */

class Planning {

    private:

    VehicleControl vehicleControl;
    IMU imu;
    Scanners scanners;
    GPS gps;
    SensorsAndCameras sensorsAndCameras;
    Display display;

    bool wantsToAcc;          // when car is told to accelerate
    bool wantsToBrk;          // when car is told to break
    int speed_wanted;         // the speed to accelerate or break to

    public:

    /* initialize an instance of every class */

    Planning() {
        vehicleControl = VehicleControl(true,true);
        imu = IMU(60);
        scanners = Scanners(12, 3, 3, true);
        gps = GPS(true, false, 4, 2);
        sensorsAndCameras = SensorsAndCameras();
        display = Display();
    }


    /* updates vehicle when case detected */
    void brakeWhenObjectDetected() {
        if(vehicleControl.getGear() == 2 || vehicleControl.getGear()
== 3) {
            if (imu.getCurrentVelocity() != 0 &&
sensorsAndCameras.getDistanceInFront() > 20 &&
sensorsAndCameras.getDistanceInFront() < 100) {
                vehicleControl.brake(imu, sensorsAndCameras, 1);
                wantsToAcc = false;
            } else if (imu.getCurrentVelocity() != 0 &&
sensorsAndCameras.getDistanceInFront() > 10 &&
sensorsAndCameras.getDistanceInFront() <= 20) {
                vehicleControl.brake(imu, sensorsAndCameras, 2);
                wantsToAcc = false;
            } else if (imu.getCurrentVelocity() != 0 &&
sensorsAndCameras.getDistanceInFront() > 0 &&
sensorsAndCameras.getDistanceInFront() <= 10) {
                vehicleControl.brake(imu, sensorsAndCameras, 3);
                wantsToAcc = false;
            }
        } else if (vehicleControl.getGear() == 1) {
             if (imu.getCurrentVelocity() != 0 &&
sensorsAndCameras.getDistanceBehind() > 0 &&
sensorsAndCameras.getDistanceBehind() < 20) {
                vehicleControl.brake(imu, sensorsAndCameras, 3);
                wantsToAcc = false;
             }
        }
    }

    void automaticallyChangeLane() {
        if (imu.getCurrentVelocity() != 0 &&
vehicleControl.getccActive() && gps.getNumberOfLanes() > 1) {
            //if one of the turn signals is active
            if (vehicleControl.getTurn() < 0) {  //left turn
                if(!sensorsAndCameras.isObjectLeft() &&
gps.getLaneNumber() > 1) {
                    gps.setLaneNumber(gps.getLaneNumber() - 1);
                    vehicleControl.turnComplete();
                    sensorsAndCameras.setObjectRight(false);
                } else {
                    vehicleControl.turnComplete();
                }
            } else if (vehicleControl.getTurn() > 0) {  //right turn
                if(!sensorsAndCameras.isObjectRight() &&
gps.getLaneNumber() < gps.getNumberOfLanes()) {
                    gps.setLaneNumber(gps.getLaneNumber() + 1);
                    vehicleControl.turnComplete();
                    sensorsAndCameras.setObjectLeft(false);
                } else {
                    vehicleControl.turnComplete();
                }
            }
        }
    }

    void automaticHeadLights() {
        if ((sensorsAndCameras.getLightLevel() < 200 ||
sensorsAndCameras.getRainDetected()) &&
vehicleControl.getHeadLightLevel() == 0) {
            vehicleControl.turnOnHeadLights(1);
        } else if((sensorsAndCameras.getLightLevel() >= 200 &&
!sensorsAndCameras.getRainDetected()) &&
vehicleControl.getHeadLightLevel() > 0) {
            vehicleControl.turnOffHeadlights();
        }
    }

    void automaticaHighBeams() {
        if (sensorsAndCameras.getLightLevel() < 50
            && imu.getCurrentVelocity() > 25
            && !sensorsAndCameras.getRainDetected()
            && sensorsAndCameras.getDistanceInFront() >= 100) {
                if(vehicleControl.getHeadLightLevel() == 1 ) {
                    vehicleControl.turnOnHeadLights(2);
                }
            }
        else if(vehicleControl.getHeadLightLevel() == 2) {
            vehicleControl.turnOnHeadLights(1);
        }
    }

    void automaticWindshieldWipers() {
        if (sensorsAndCameras.getRainDetected()) {
            vehicleControl.turnOnWindshieldWipers(true);
        } else {
            vehicleControl.turnOnWindshieldWipers(false);
        }
    }

    void gearControl() {
        // No need to worry about neutral, not implemented yet
        if(vehicleControl.getGear() == 0 && imu.getCurrentVelocity()
!= 0) {
            imu.setCurrentVelocity(0);
        }
        else if(vehicleControl.getGear() == 1 &&
imu.getCurrentVelocity() > 0) {
            imu.setCurrentVelocity(0);
        }
        else if(vehicleControl.getGear() == 3 &&
imu.getCurrentVelocity() < 0) {
            imu.setCurrentVelocity(0);
        }

        if((vehicleControl.getGear() == 0 || vehicleControl.getGear()
== 1 || vehicleControl.getGear() == 2)
            && vehicleControl.getccActive()) vehicleControl.stopCC();
        else if(vehicleControl.getGear() == 3 &&
!vehicleControl.getccActive()) vehicleControl.startCC(imu,gps);

    }

    void acc() {
        int gear = vehicleControl.getGear();
        if(wantsToAcc) {
            vehicleControl.accelerateTo(imu, sensorsAndCameras,
speed_wanted);
```

```
            if(imu.getCurrentVelocity() >= speed_wanted && gear == 3)
wantsToAcc = false;
            if(imu.getCurrentVelocity() <= speed_wanted && gear == 1)
wantsToAcc = false;
        }
    }

    void brk() {
        int gear = vehicleControl.getGear();
        if(wantsToBrk) {
            vehicleControl.brakeTo(imu, sensorsAndCameras,
speed_wanted);
            if(imu.getCurrentVelocity() <= speed_wanted && gear == 3
) wantsToBrk = false;
            if(imu.getCurrentVelocity() >= speed_wanted && gear == 1
) wantsToBrk = false;
        }
    }

    void check_all() {
        brakeWhenObjectDetected();
        acc();
        brk();
        automaticHeadLights();
        automaticallyChangeLane();
        automaticaHighBeams();
        automaticWindshieldWipers();
        gearControl();
    }


    /* updating display*/

    void automaticObjectDetection() {

display.set_cars_in_front(sensorsAndCameras.getDistanceInFront() <
100);

display.set_cars_in_back(sensorsAndCameras.getDistanceBehind() < 20);
```

```
        display.set_cars_on_left(sensorsAndCameras.isObjectLeft());
        display.set_cars_on_right(sensorsAndCameras.isObjectRight());
    }

    void wipersOn() {
        display.set_wipers(vehicleControl.windshieldWipersOn());
    }

    void headlightLevel() {
       display.set_headlights(vehicleControl.getHeadLightLevel());
    }

    void currentSpeed() {
        display.set_speed((int)imu.getCurrentVelocity());
    }

    void automaticRearCamera() {
        display.set_rearview(vehicleControl.getGear() == 1 &&
imu.getCurrentVelocity() <= 0);
    }

    void checkLanes() {
        display.set_lane(gps.getLaneNumber());
    }

    void checkTurn() {
        display.set_left_turn(vehicleControl.getTurn() == -1);
        display.set_right_turn(vehicleControl.getTurn() == 1);

    }

    void detectLaneDeparture() {
        if (imu.getCurrentVelocity() != 0 &&
!gps.isOnUnregisteredRoad()) {
            if (scanners.distanceFromLineLeft() <= 0) {
                display.set_lane_warning(0); // changing lane to the
left
            }
            if (scanners.distanceFromLineRight() <= 0) {
                display.set_lane_warning(1); // changing lane to the
right
            }
        }
    }

    void checkGear() {
        display.set_gear(vehicleControl.getGear());
    }

    void checkCC() {

display.set_cruise_control_active(vehicleControl.getccActive());
    }

    void checkWarnings() {
        if (vehicleControl.getTurn() == 0
        || (vehicleControl.getTurn() == -1 &&
!sensorsAndCameras.isObjectLeft())
        || (vehicleControl.getTurn() == 1 &&
!sensorsAndCameras.isObjectRight())) {
            display.set_lane_warning(-1);
        } else if (vehicleControl.getTurn() == -1 &&
sensorsAndCameras.isObjectLeft()) {
            display.set_lane_warning(0);
        } else if (vehicleControl.getTurn() == 1 &&
sensorsAndCameras.isObjectRight()) {
            display.set_lane_warning(1);
        }
    }

    void updateDisplay() {
        checkGear();
        checkTurn();
        checkLanes();
        automaticObjectDetection();
        wipersOn();
        headlightLevel();
        currentSpeed();
        detectLaneDeparture();
        automaticRearCamera();
        checkWarnings();
        checkCC();
    }


    /* Run system */

    void run_systems() {

        display.print_display();

        signal(SIGINT, environment_handler);
        signal(SIGTSTP, vehicle_handler);

        while(true) {

            check_all();
            updateDisplay();
            display.print_display();

            if (wantsEnvironmentInput) {

                wantsEnvironmentInput = false;

                int input;
                std::cout << "\n\n                  0: default, 1:
car in front, 2: car behind, 3: car to the side, 4: light level, 5:
toggle rain\n";
                std::cout << "\n                   Change
Environment: ";
                std::cin >> input;
                int val;

                switch(input) {
                    case -1:
                        exit(0);
                    case 0:
                        sensorsAndCameras = SensorsAndCameras();
                        break;
                    case 1:
                        std::cout << "                  Distance in 
front: ";
                        std::cin >> val;
                        sensorsAndCameras.setDistanceInFront(val);
                        break;
                    case 2:
                        std::cout << "                  Distance 
behind: ";
                        std::cin >> val;
                        sensorsAndCameras.setDistanceBehind(val);
                        break;
                    case 3:
                        std::cout << "                  Object left 
(-1) or right (1): ";
                        std::cin >> val;
                        if(val < 0) 
sensorsAndCameras.setObjectLeft(true);
                        else if(val > 0) 
sensorsAndCameras.setObjectRight(true);
                        else {
                            sensorsAndCameras.setObjectLeft(false);
                            sensorsAndCameras.setObjectRight(false);
                        }
                        break;
                    case 4:
                        std::cout << "                  Light Level: 
";
                        std::cin >> val;
                        if(val < 0) val = 0;
                        sensorsAndCameras.setLightLevel(val);
                        break;
                    case 5:
                        std::cout << "                  Rain on (1) 
or off (0): ";
                        std::cin >> val;
                        sensorsAndCameras.setRain(val > 0);
                        break;
                    default:
                        break;
                    }

                updateDisplay();
                display.print_display();
            }

            if (wantsVehicleInput) {

                wantsVehicleInput = false;

                int input;
                std::cout << "\n\n                          0: 
default, 1: apply brake, 2: accelerate, 3: change gear, 4: turn 
signal\n";
                std::cout << "\n                           Vehicle 
Input: ";
                std::cin >> input;
                int val;

                switch(input) {
                    case -1:
                        exit(0);
                    case 0:
                        imu = IMU(60);
                        gps = GPS(true, false, 4, 2);
                        break;
                    case 1:
                        std::cout << "
Brake to what speed? ";
                        std::cin >> val;
                        if( (vehicleControl.getGear() == 3 && val < 0
                            || val > imu.getCurrentVelocity()
                            ) ||
                            (vehicleControl.getGear() == 1 && val > 0
) ||
                            vehicleControl.getGear() == 0 ) break;
                        wantsToBrk = true;
                        wantsToAcc = false;
                        speed_wanted = val;
                        break;
                    case 2:
                        std::cout << "
Accelerate to what speed? ";
                        std::cin >> val;
                        if( (vehicleControl.getGear() == 3 && val <
0) ||
                            (vehicleControl.getGear() == 1 && val > 0
) ||
                            vehicleControl.getGear() == 0 ) break;
                        wantsToAcc = true;
                        wantsToBrk = false;
                        speed_wanted = val;
                        break;
                    case 3:
                        if(imu.getCurrentVelocity() < -5 ||
imu.getCurrentVelocity() > 5) {
                            std::cout << "
Can only change gear at low speeds\n";

std::this_thread::sleep_for(std::chrono::milliseconds(1500));
                            break;
                        }
                        std::cout << "
Change gear to park (0), reverse (1), drive (3)? ";
                        std::cin >> val;
                        if(val == 0 || val == 1 || val == 3)
vehicleControl.setGear(val);
                        // note that neutral is not fully implemented
so it is not included
                        break;
                    case 4:
                        std::cout << "
Turn signal left (-1) or right (1): ";
                        std::cin >> val;
                        if(val < 0) vehicleControl.leftTurnSignal();
                        else if(val > 0)
vehicleControl.rightTurnSignal();
                        break;
                    default:
                        break;
                    }

                updateDisplay();
                display.print_display();
            }

            for (int i = 0; i < 40; i++) {
                if (!wantsEnvironmentInput && !wantsVehicleInput)
std::this_thread::sleep_for(std::chrono::milliseconds(50));
            }
        }

    }

};
```

Vehicle.cpp all of the code for the running vehicle. Planning takes an instance of VehicleControl, Display, and the Sensor Fusion classes (SensorsAndCameras, Scanners, GPS, IMU) and constantly updates all of the objects so the vehicle can be tested.

6.3 vehicle.hpp

```
/* struct to store system/vehicle status */

struct status_struct {
  int speed;              // in mph
  int gear;               // 0 -> park, 1 -> reverse, 2 -> neutral, 3 -> drive
  bool cruise_control_active;
  bool wipers_on;
  bool cars_in_front;
```

```
   bool cars_in_back;
   bool cars_on_left;
   bool cars_on_right;
   int lane_warning;            // 0 = warning on left, 1 = on right
   int headlights;              // 0 = off, 1 = on, 2 = highbeams
   bool rear_view;
   int lane;
   int num_lanes;
   bool leftTurn;
   bool rightTurn;
 };
```

Vehicle.hpp holds the status struct that is used by the Display class in vehicle.cpp.

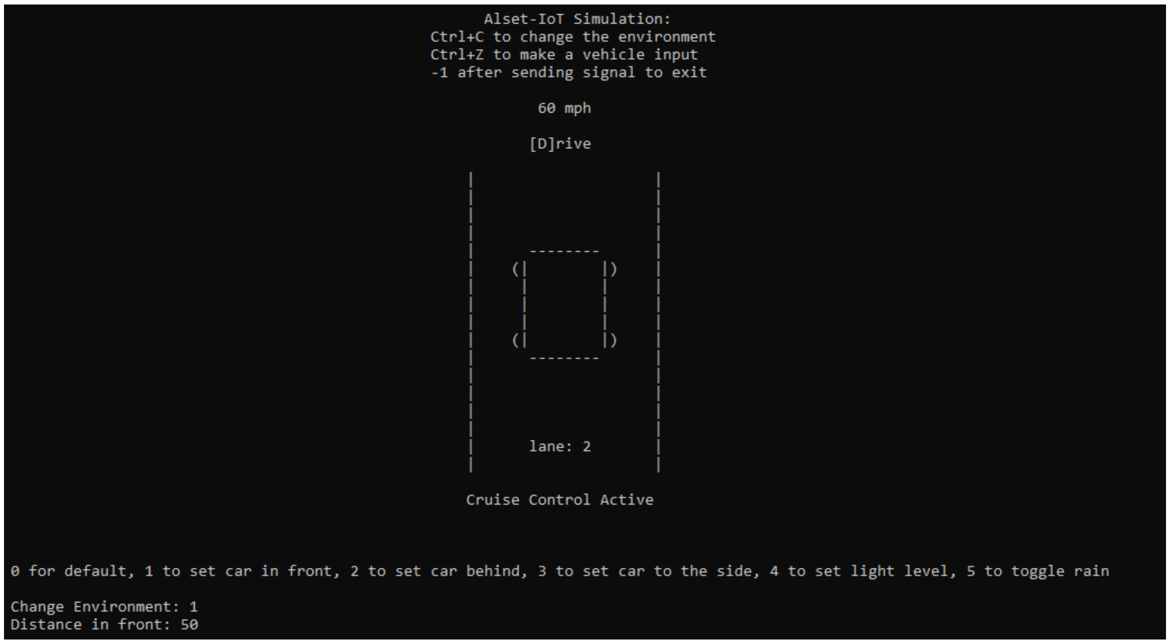
7. Testing

7.1 Use Case 1 – Vehicle brakes to avoid cars detected in front

Steps to replicate:

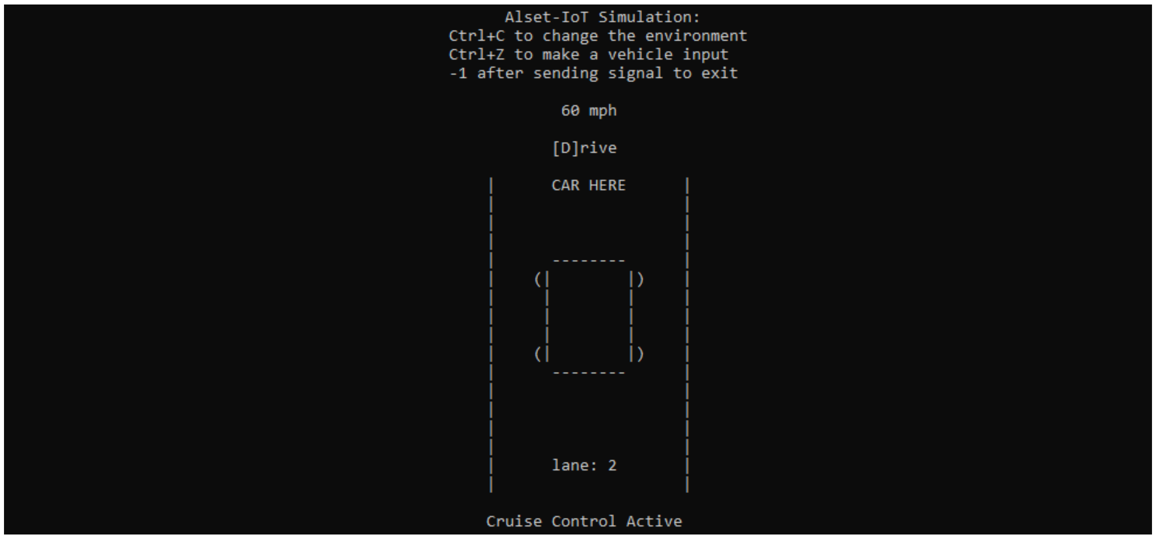
- Run the default state of the car
- Ctrl+C to signal to change the environment
- Enter a 1 to set car in front
- Enter the distance of the car in front

Note: A car must be less than 100 feet in front to be detected.



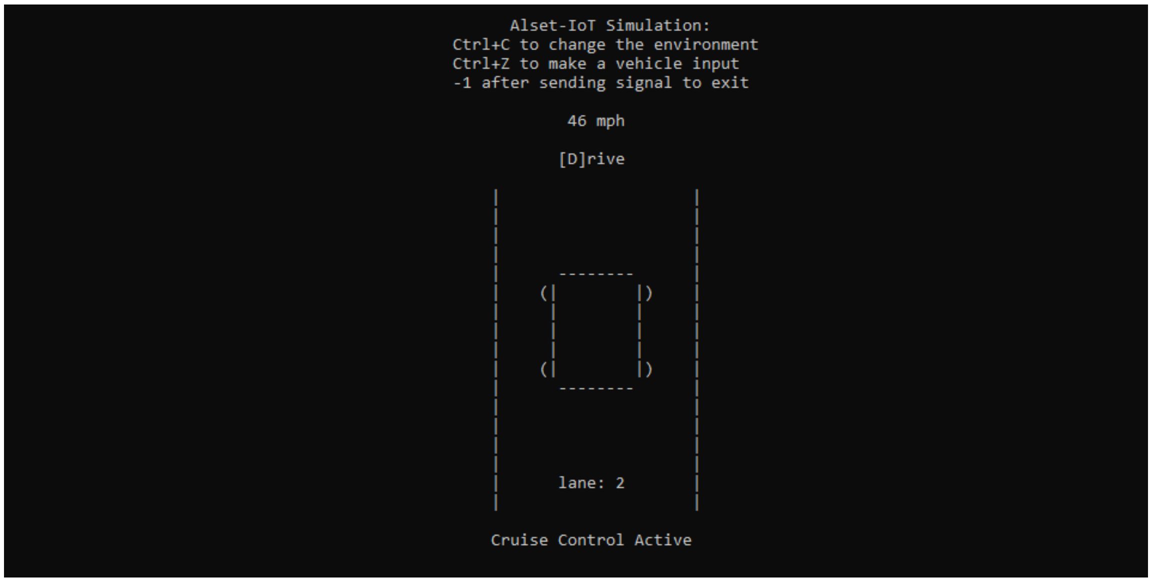
```
                    Alset-IoT Simulation:
                 Ctrl+C to change the environment
                 Ctrl+Z to make a vehicle input
                 -1 after sending signal to exit

                       60 mph

                      [D]rive

                 |                    |
                 |                    |
                 |                    |
                 |                    |
                 |      --------      |
                 |    (|        |)    |
                 |     |        |     |
                 |     |        |     |
                 |     |        |     |
                 |    (|        |)    |
                 |      --------      |
                 |                    |
                 |                    |
                 |                    |
                 |                    |
                 |      lane: 2       |
                 |                    |

               Cruise Control Active


0 for default, 1 to set car in front, 2 to set car behind, 3 to set car to the side, 4 to set light level, 5 to toggle rain

Change Environment: 1
Distance in front: 50
```

Changing the environment to have a car 50 ft in front.



```
Alset-IoT Simulation:
Ctrl+C to change the environment
Ctrl+Z to make a vehicle input
-1 after sending signal to exit

60 mph

[D]rive

|        CAR HERE        |
|                        |
|                        |
|                        |
|       --------         |
|     (|        |)       |
|      |        |        |
|      |        |        |
|      |        |        |
|     (|        |)       |
|       --------         |
|                        |
|                        |
|                        |
|                        |
|       lane: 2          |
|                        |

Cruise Control Active
```

System detects the car in front and displays the information.



```
Alset-IoT Simulation:
Ctrl+C to change the environment
Ctrl+Z to make a vehicle input
-1 after sending signal to exit

46 mph

[D]rive

|                        |
|                        |
|                        |
|                        |
|       --------         |
|     (|        |)       |
|      |        |        |
|      |        |        |
|      |        |        |
|     (|        |)       |
|       --------         |
|                        |
|                        |
|                        |
|                        |
|       lane: 2          |
|                        |

Cruise Control Active
```

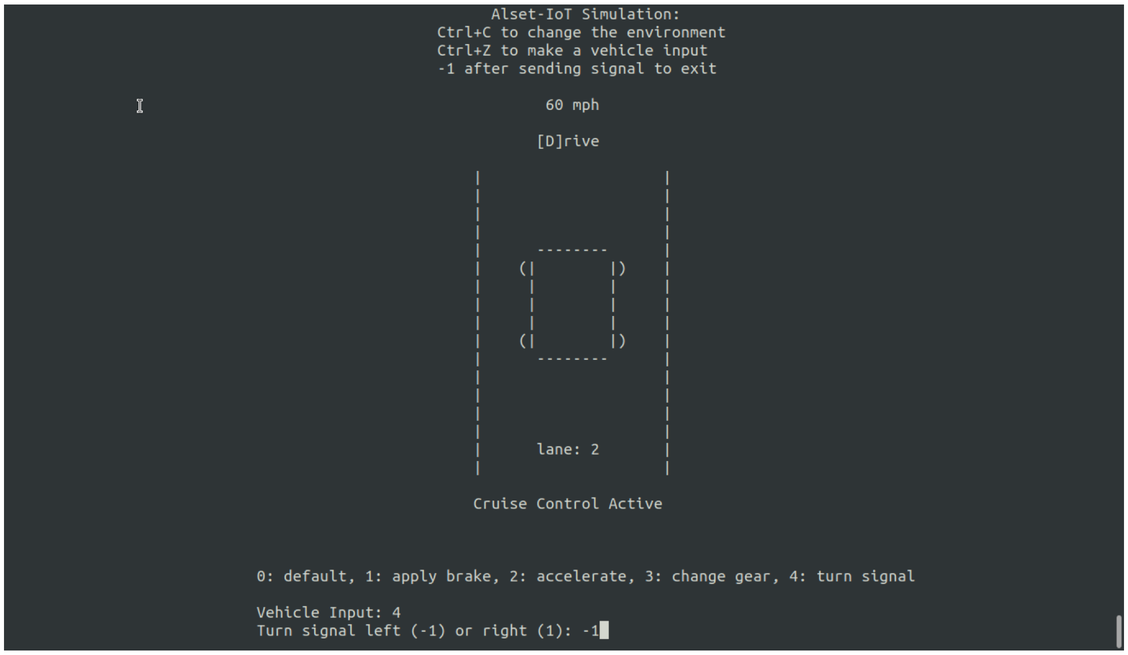
Vehicle brakes to 46 mph and car is no longer detected in front

7.2 Use Case 2 – Driver requests automatic lane change

Steps to Replicate:

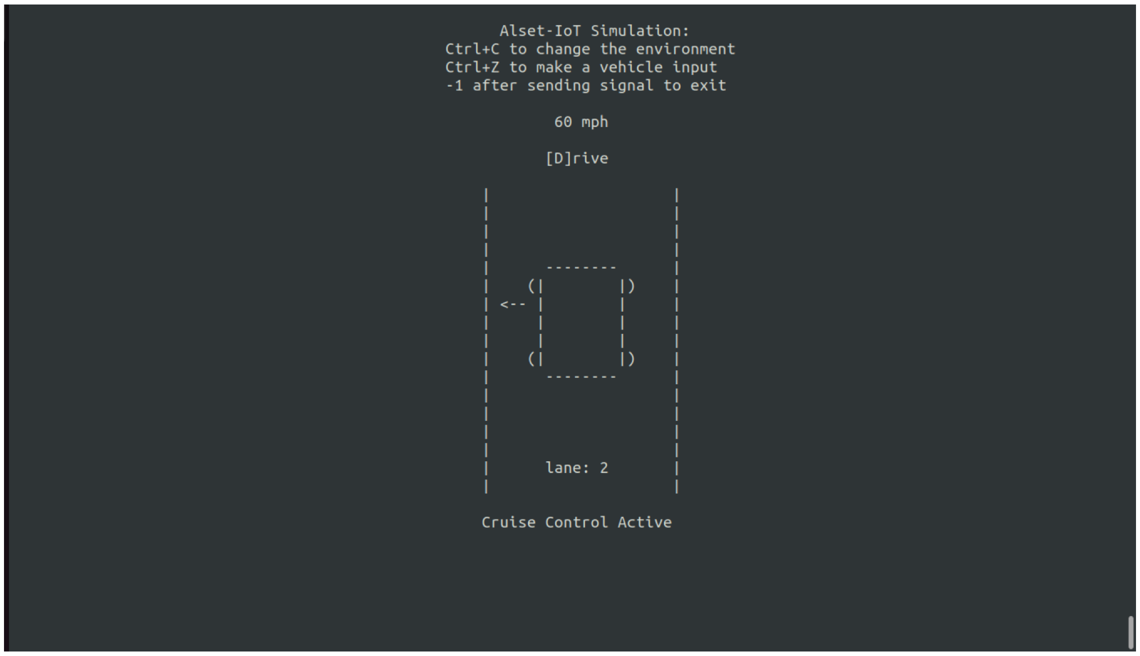
- Run the default state of the car
- Ctrl+Z to make a vehicle input

- Input 4 to turn on a turn signal
- Set the signal to either left or right with -1 or 1



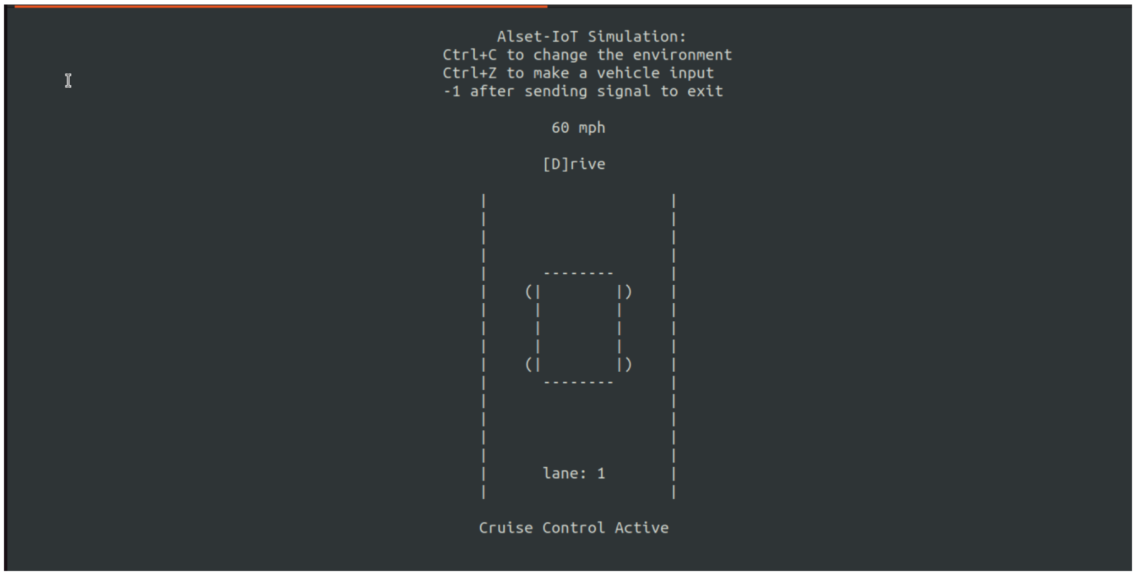
```
                Alset-IoT Simulation:
            Ctrl+C to change the environment
            Ctrl+Z to make a vehicle input
            -1 after sending signal to exit

                       60 mph

                      [D]rive

            |                    |
            |                    |
            |                    |
            |                    |
            |      --------      |
            |    (|        |)    |
            |     |        |     |
            |     |        |     |
            |     |        |     |
            |    (|        |)    |
            |      --------      |
            |                    |
            |                    |
            |                    |
            |                    |
            |      lane: 2       |
            |                    |

             Cruise Control Active


0: default, 1: apply brake, 2: accelerate, 3: change gear, 4: turn signal

Vehicle Input: 4
Turn signal left (-1) or right (1): -1
```

Inputting a left turn signal while in lane 2.



```
                Alset-IoT Simulation:
            Ctrl+C to change the environment
            Ctrl+Z to make a vehicle input
            -1 after sending signal to exit

                       60 mph

                      [D]rive

            |                    |
            |                    |
            |                    |
            |                    |
            |      --------      |
            |    (|        |)    |
            | <-- |        |     |
            |     |        |     |
            |     |        |     |
            |    (|        |)    |
            |      --------      |
            |                    |
            |                    |
            |                    |
            |                    |
            |      lane: 2       |
            |                    |

             Cruise Control Active
```

Left turn signal displayed.

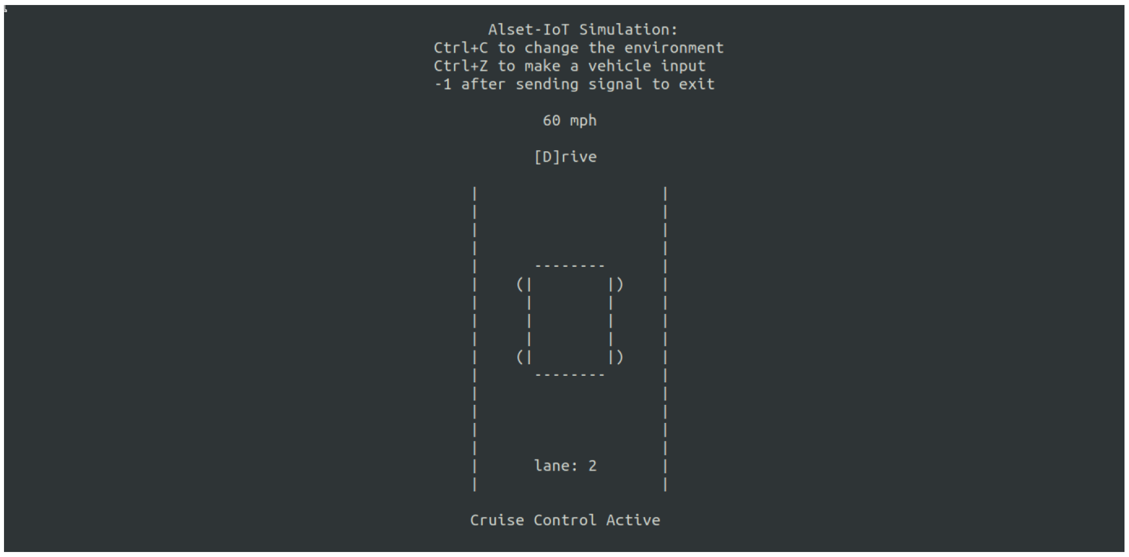


```
Alset-IoT Simulation:
Ctrl+C to change the environment
Ctrl+Z to make a vehicle input
-1 after sending signal to exit

60 mph

[D]rive

|                    |
|                    |
|                    |
|                    |
|      --------      |
|    (|        |)    |
|     |        |     |
|     |        |     |
|     |        |     |
|    (|        |)    |
|      --------      |
|                    |
|                    |
|                    |
|                    |
|      lane: 1       |
|                    |

Cruise Control Active
```

Vehicle successfully changes to lane 1.

7.3 Use Case 3 – Vehicle stays in lane in Cruise Control

By default, the lane position and speed will be maintained during Cruise Control. There are no steps to this test; as long as the vehicle remains in cruise control, the lanes will be displayed and only changed when input is provided.

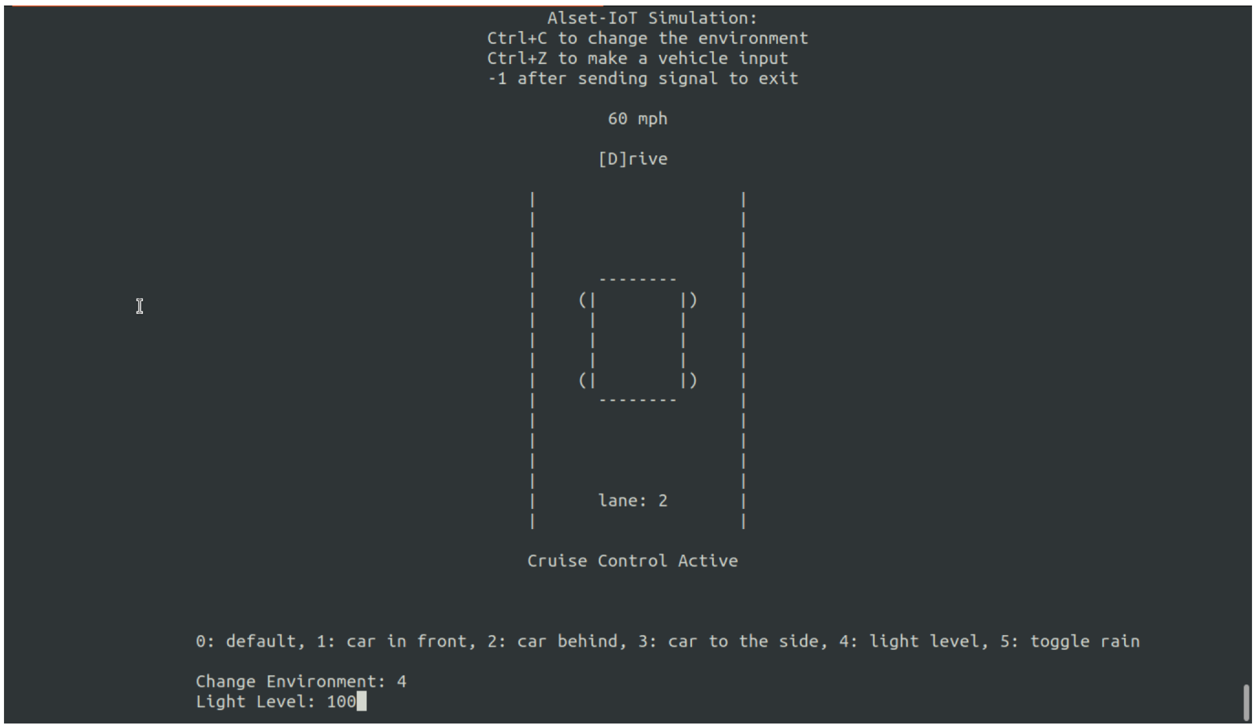


```
Alset-IoT Simulation:
Ctrl+C to change the environment
Ctrl+Z to make a vehicle input
-1 after sending signal to exit

60 mph

[D]rive

|                    |
|                    |
|                    |
|                    |
|      --------      |
|    (|        |)    |
|     |        |     |
|     |        |     |
|     |        |     |
|    (|        |)    |
|      --------      |
|                    |
|                    |
|                    |
|                    |
|      lane: 2       |
|                    |

Cruise Control Active
```

7.4 Use Case 4 – Headlights turn on in low-light conditions

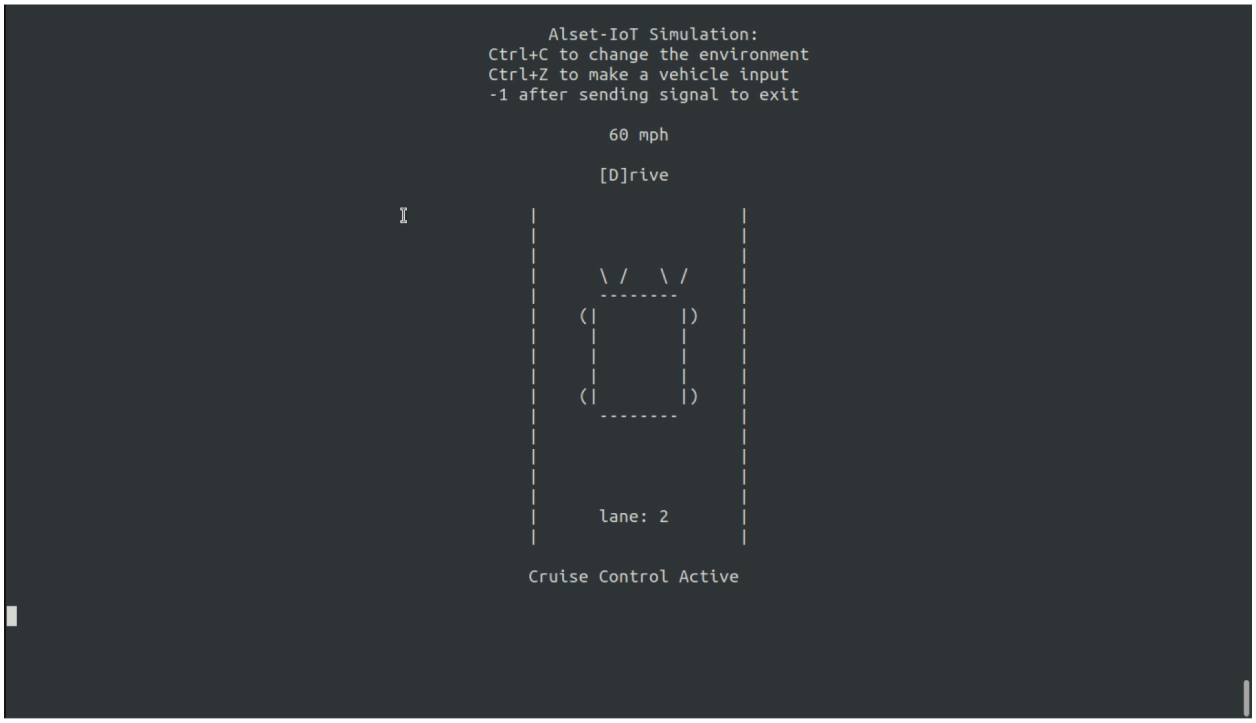
Steps to replicate:

- Run the default state of the car
- Ctrl+C to set the light level (by pressing 4)
- Input a light level under 200 lux and above 50 lux to activate the headlights (but not the high beams)



```
                    Alset-IoT Simulation:
                 Ctrl+C to change the environment
                 Ctrl+Z to make a vehicle input
                 -1 after sending signal to exit

                         60 mph

                         [D]rive

               |                     |
               |                     |
               |                     |
               |                     |
               |       --------      |
               |     (|        |)    |
               |      |        |     |
               |      |        |     |
               |     (|        |)    |
               |       --------      |
               |                     |
               |                     |
               |                     |
               |                     |
               |       lane: 2       |
               |                     |

                   Cruise Control Active


0: default, 1: car in front, 2: car behind, 3: car to the side, 4: light level, 5: toggle rain

Change Environment: 4
Light Level: 100
```

Changing the light level to 100 lux.

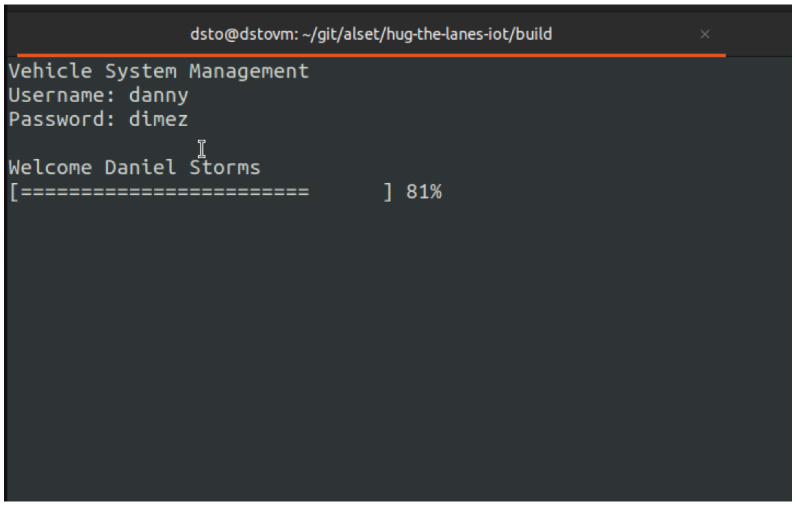


```
      Alset-IoT Simulation:
Ctrl+C to change the environment
Ctrl+Z to make a vehicle input
-1 after sending signal to exit

            60 mph

           [D]rive

    |                    |
    |                    |
    |                    |
    |      \ /   \ /     |
    |      --------      |
    |    (|        |)    |
    |     |        |     |
    |     |        |     |
    |     |        |     |
    |    (|        |)    |
    |      --------      |
    |                    |
    |                    |
    |                    |
    |                    |
    |      lane: 2       |
    |                    |

    Cruise Control Active
```

Low light level detected and headlights are turned on.

7.5 Use Case 5 – Technician logs in to monitor system status

This case takes place when first running the software. It allows for one of the technicians to log-in after entering a username and password. The following screenshot shows the log-in screen.



```
dsto@dstovm: ~/git/alset/hug-the-lanes-iot/build

Vehicle System Management
Username: danny
Password: dimez

Welcome Daniel Storms
[========================      ] 81%
```

The history of the vehicle’s state can be viewed by scrolling up in the terminal.

7.6 Other Numbered Requirements

All other numbered requirements are satisfied by one or more of the previous test cases or additional cases that will be presented here.

1. **Vehicle adaptively brakes when an object is detected**

Satisfied in Use Case 1 as the speed decreases based on current velocity and distance to the object.

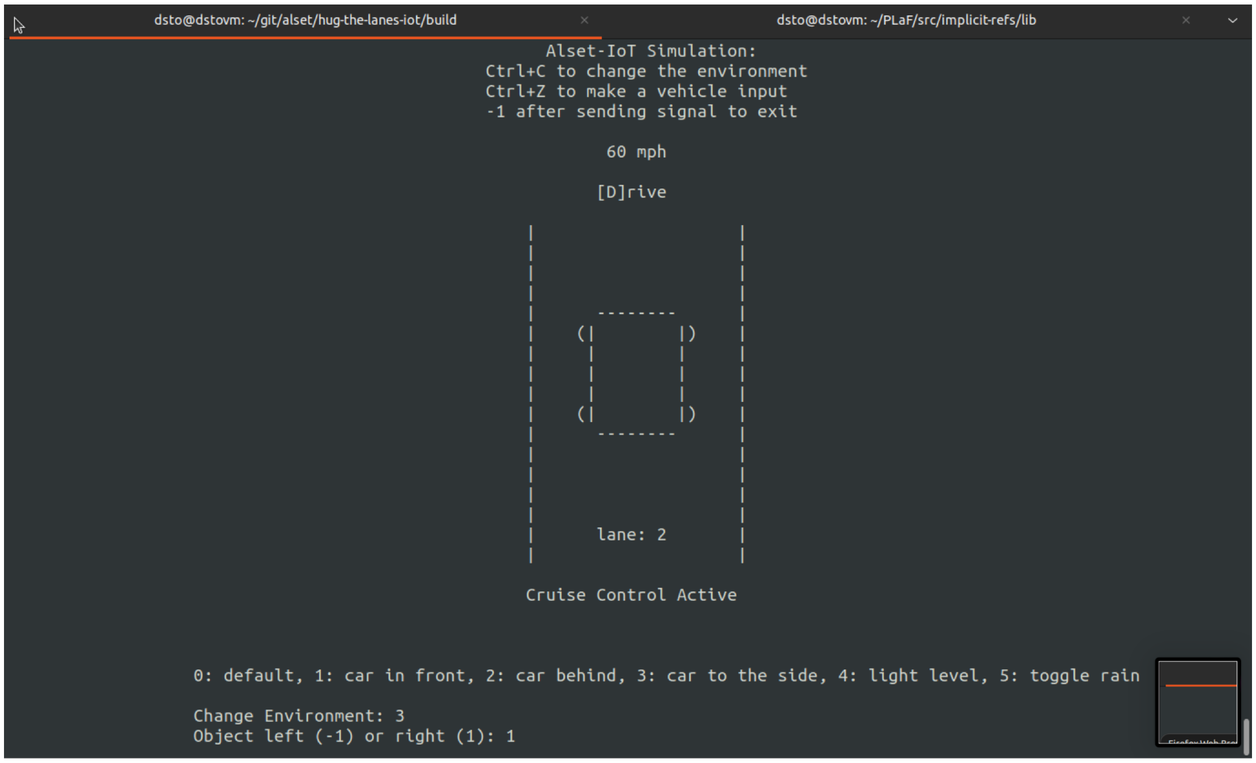
2. **Vehicle automatically detects lanes in real-time**

Satisfied in every Use Case, as the lanes are constantly displayed to the user.

3. **Vehicle Prevents User From Merging Into Object**

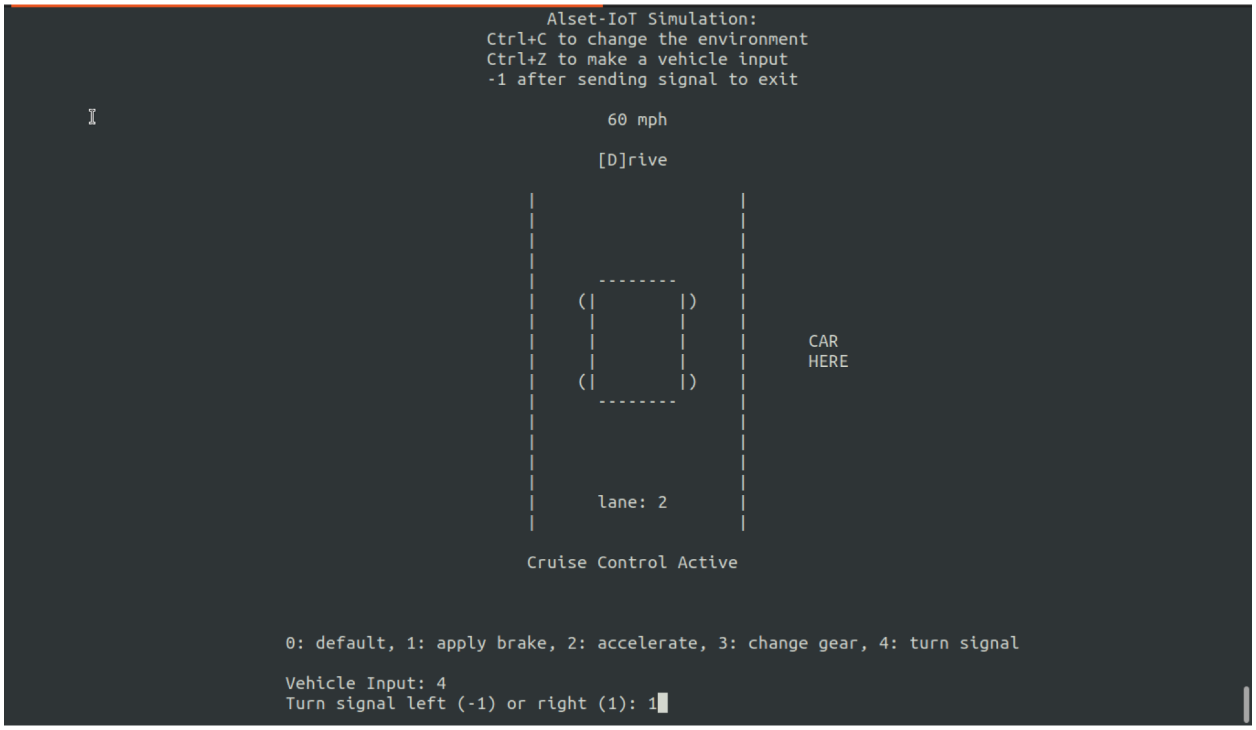
Satisfied in the following test:

- Run the default state of the car
- Ctrl+Z then 3 to set a car to the side (-1 or 1)
- Ctrl+C then 4 to set the turn signal to the same side as the car (-1 or 1)

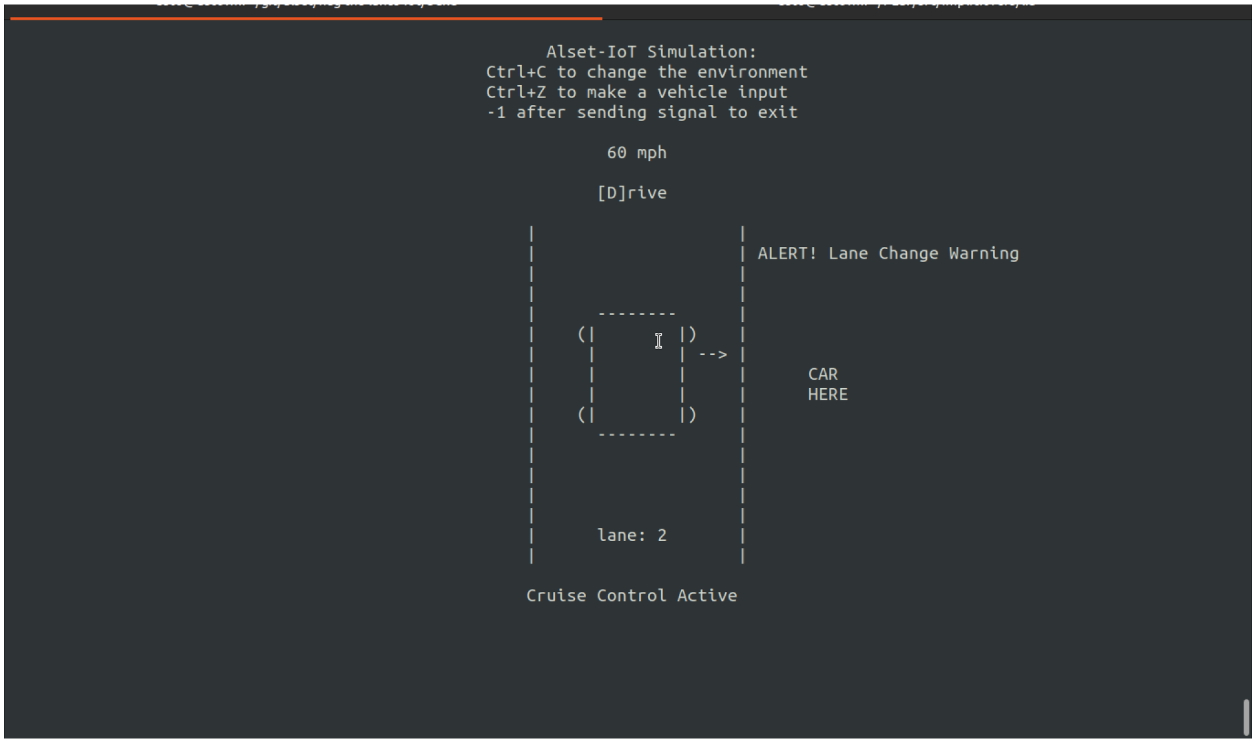


dsto@dstovm: ~/git/alset/hug-the-lanes-iot/build   dsto@dstovm: ~/PLaF/src/implicit-refs/lib

```
                    Alset-IoT Simulation:
                  Ctrl+C to change the environment
                  Ctrl+Z to make a vehicle input
                  -1 after sending signal to exit

                              60 mph

                             [D]rive

                     |                    |
                     |                    |
                     |                    |
                     |                    |
                     |      --------      |
                     |    (|        |)    |
                     |     |        |     |
                     |     |        |     |
                     |     |        |     |
                     |    (|        |)    |
                     |      --------      |
                     |                    |
                     |                    |
                     |                    |
                     |                    |
                     |      lane: 2       |
                     |                    |

                       Cruise Control Active


0: default, 1: car in front, 2: car behind, 3: car to the side, 4: light level, 5: toggle rain

Change Environment: 3
Object left (-1) or right (1): 1
```



```
                    Alset-IoT Simulation:
                  Ctrl+C to change the environment
                  Ctrl+Z to make a vehicle input
                  -1 after sending signal to exit

                              60 mph

                             [D]rive

                     |                    |
                     |                    |
                     |                    |
                     |                    |
                     |      --------      |
                     |    (|        |)    |
                     |     |        |     |
                     |     |        |     |      CAR
                     |     |        |     |      HERE
                     |    (|        |)    |
                     |      --------      |
                     |                    |
                     |                    |
                     |                    |
                     |                    |
                     |      lane: 2       |
                     |                    |

                       Cruise Control Active


0: default, 1: apply brake, 2: accelerate, 3: change gear, 4: turn signal

Vehicle Input: 4
Turn signal left (-1) or right (1): 1
```



```
      Alset-IoT Simulation:
Ctrl+C to change the environment
Ctrl+Z to make a vehicle input
-1 after sending signal to exit

            60 mph

           [D]rive

     |                    |
     |                    | ALERT! Lane Change Warning
     |                    |
     |                    |
     |     --------       |
     |   (|        |)     |
     |    |        | -->  |
     |    |        |      |     CAR
     |    |        |      |     HERE
     |   (|        |)     |
     |     --------       |
     |                    |
     |                    |
     |                    |
     |                    |
     |     lane: 2        |
     |                    |

    Cruise Control Active
```

The vehicle prevents the user from merging by alerting them with a warning and staying in the same lane, then turning the signal off.

4. **Vehicle maintains lane position and speed**

Satisfied in Use Case 3, in which the vehicle is in default cruise control, in which lane position and speed remain constant.

5. **Vehicle changes lanes when instructed**

Satisfied in Use Case 2, in which the vehicle changes lane when the driver requests, changing the lane number and turning off the signal when completed.

6. **Vehicle automatically activates lights**

Satisfied in Use Case 4, in which the vehicle turns on the headlights when a light level of under 200 lux is detected.

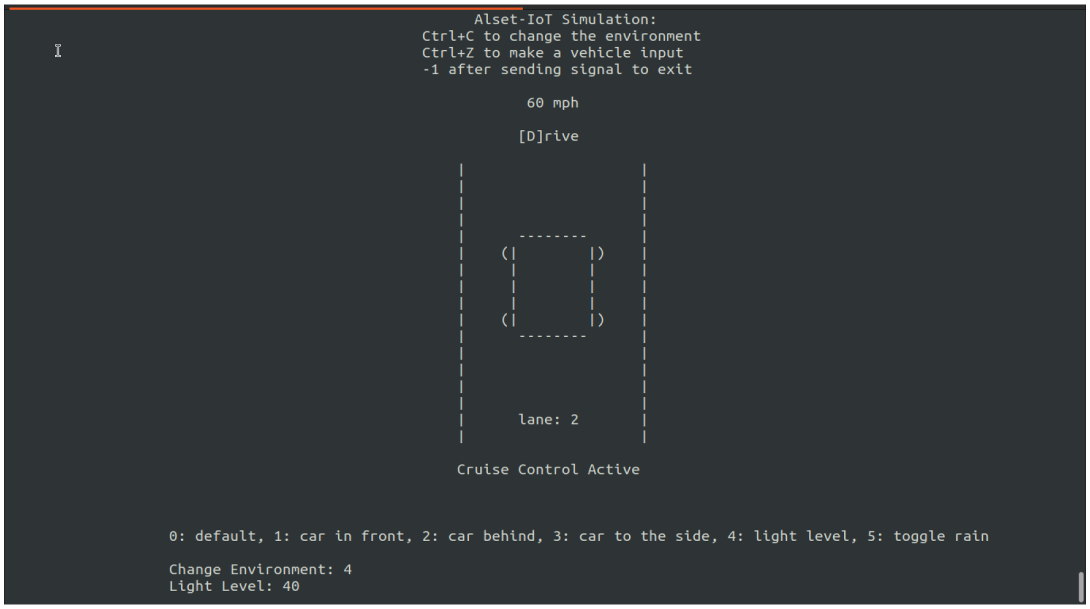
7. **Vehicle automatically activates high beams**

Satisfied in the following test:

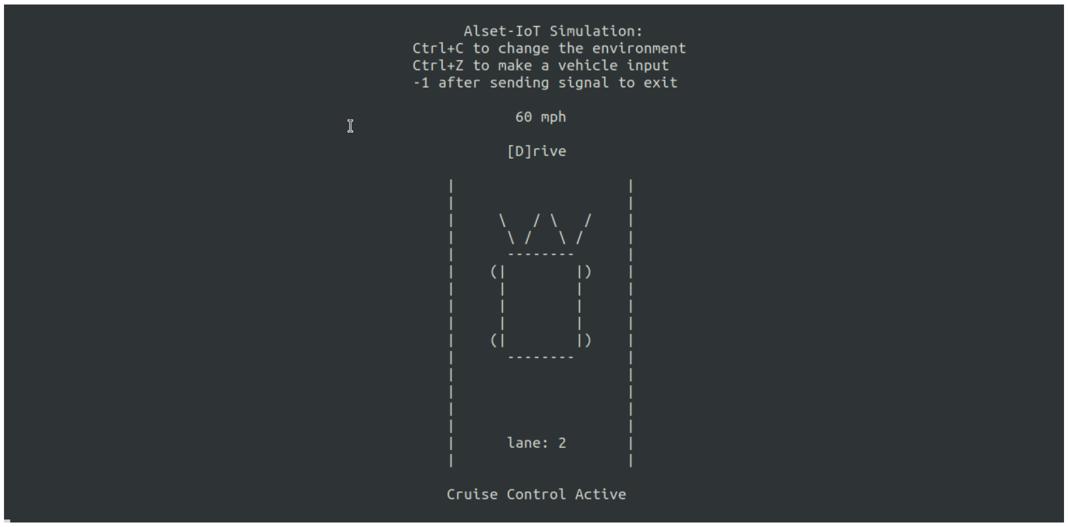
- Run the default state of the car
- Ctrl+C to change the environment
- Set the light level by entering 4
- Input a light level under 50 lux to activate the high beams

  High beams successfully activated.

Note: Other criteria for high beam activation is that there are no cars within 100 feet in front of the vehicle, the car is traveling over 25 mph, and it is not raining.



```
                    Alset-IoT Simulation:
                Ctrl+C to change the environment
                Ctrl+Z to make a vehicle input
                -1 after sending signal to exit

                        60 mph

                       [D]rive

              |                      |
              |                      |
              |                      |
              |                      |
              |       --------       |
              |    (|        |)      |
              |     |        |       |
              |     |        |       |
              |     |        |       |
              |    (|        |)      |
              |       --------       |
              |                      |
              |                      |
              |                      |
              |                      |
              |       lane: 2        |
              |                      |

                 Cruise Control Active


0: default, 1: car in front, 2: car behind, 3: car to the side, 4: light level, 5: toggle rain

Change Environment: 4
Light Level: 40
```

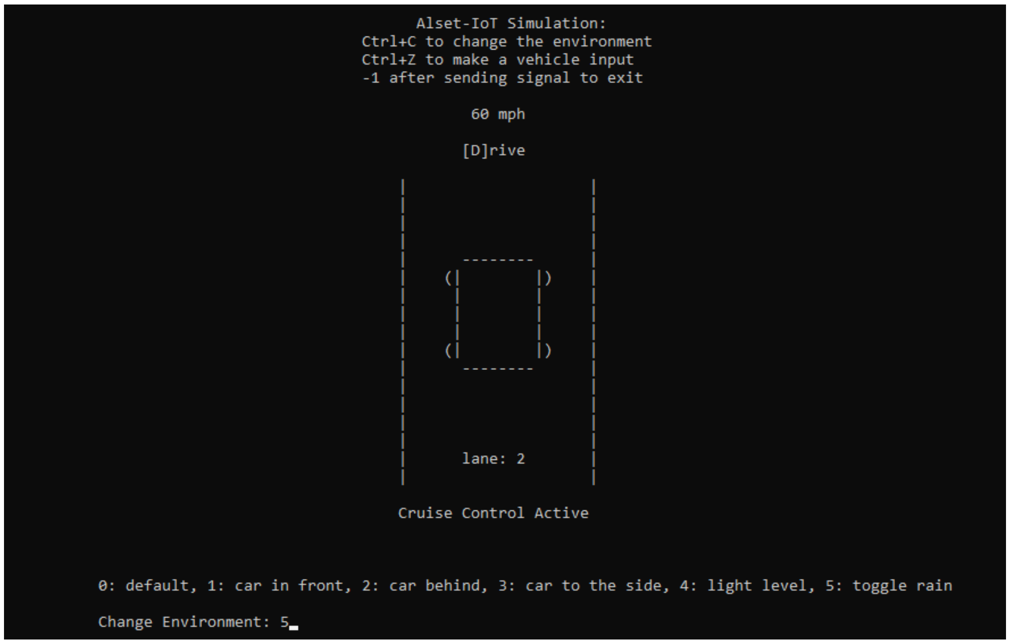


```
        Alset-IoT Simulation:
  Ctrl+C to change the environment
  Ctrl+Z to make a vehicle input
  -1 after sending signal to exit

               60 mph

              [D]rive

     |                    |
     |                    |
     |     \   / \   /    |
     |      \ /   \ /     |
     |      --------      |
     |    (|        |)    |
     |     |        |     |
     |     |        |     |
     |     |        |     |
     |    (|        |)    |
     |      --------      |
     |                    |
     |                    |
     |                    |
     |                    |
     |      lane: 2       |
     |                    |

        Cruise Control Active
```

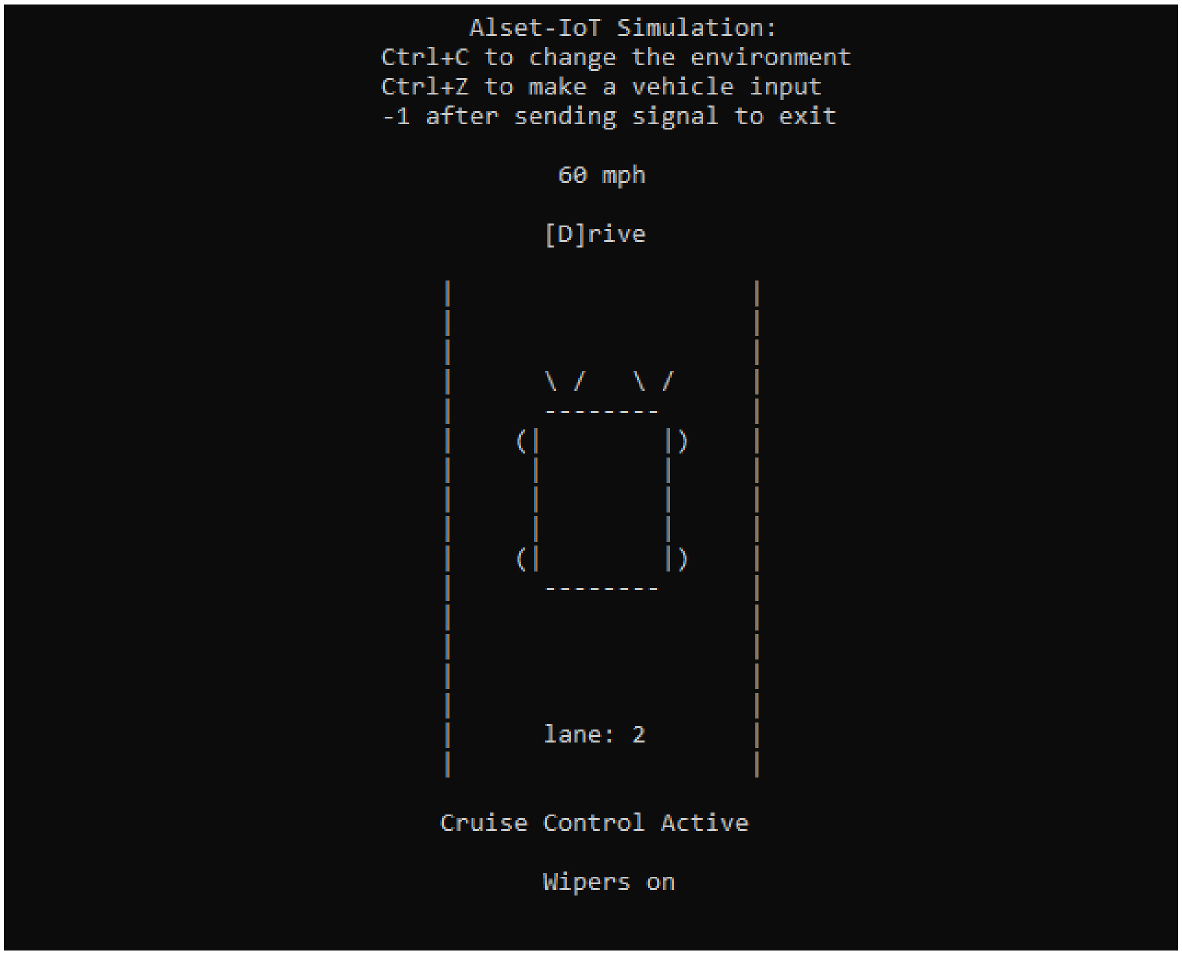
8. **Windshield wipers turn on in rain**

Satisfied in the following test:

- Run the default state of the car
- Ctrl+C to set the environment
- Enter 5 to toggle rain
- Enter 1 to turn the rain on



```
                                      Alset-IoT Simulation:
                                Ctrl+C to change the environment
                                Ctrl+Z to make a vehicle input
                                -1 after sending signal to exit

                                             60 mph

                                            [D]rive

                                   |                    |
                                   |                    |
                                   |                    |
                                   |                    |
                                   |      --------      |
                                   |    (|        |)    |
                                   |     |        |     |
                                   |     |        |     |
                                   |     |        |     |
                                   |    (|        |)    |
                                   |      --------      |
                                   |                    |
                                   |                    |
                                   |                    |
                                   |                    |
                                   |      lane: 2       |
                                   |                    |

                                      Cruise Control Active


0: default, 1: car in front, 2: car behind, 3: car to the side, 4: light level, 5: toggle rain

Change Environment: 5
```



```
         Alset-IoT Simulation:
      Ctrl+C to change the environment
      Ctrl+Z to make a vehicle input
      -1 after sending signal to exit

              60 mph

             [D]rive

       |                    |
       |                    |
       |                    |
       |      \ /   \ /     |
       |      --------      |
       |    (|        |)    |
       |     |        |     |
       |     |        |     |
       |     |        |     |
       |    (|        |)    |
       |      --------      |
       |                    |
       |                    |
       |                    |
       |                    |
       |      lane: 2       |
       |                    |

       Cruise Control Active

             Wipers on
```

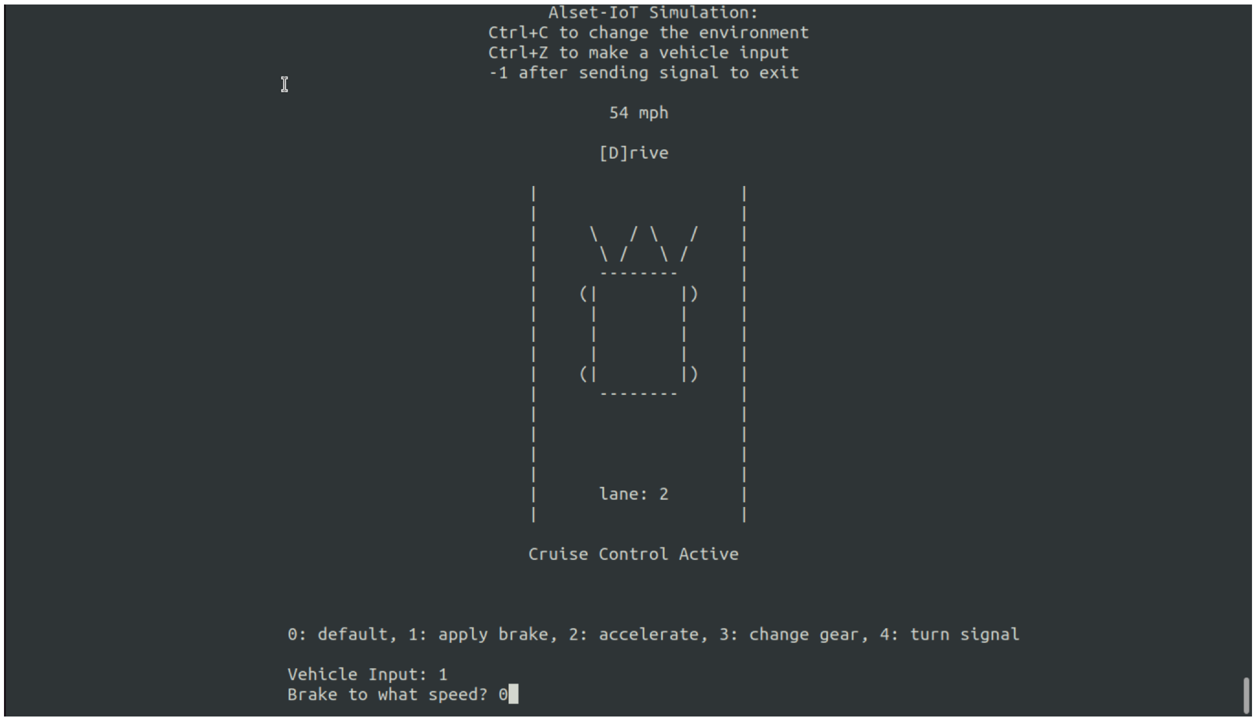
Once it is raining, wipers and headlights are turned on.

9. **Rear View Camera**

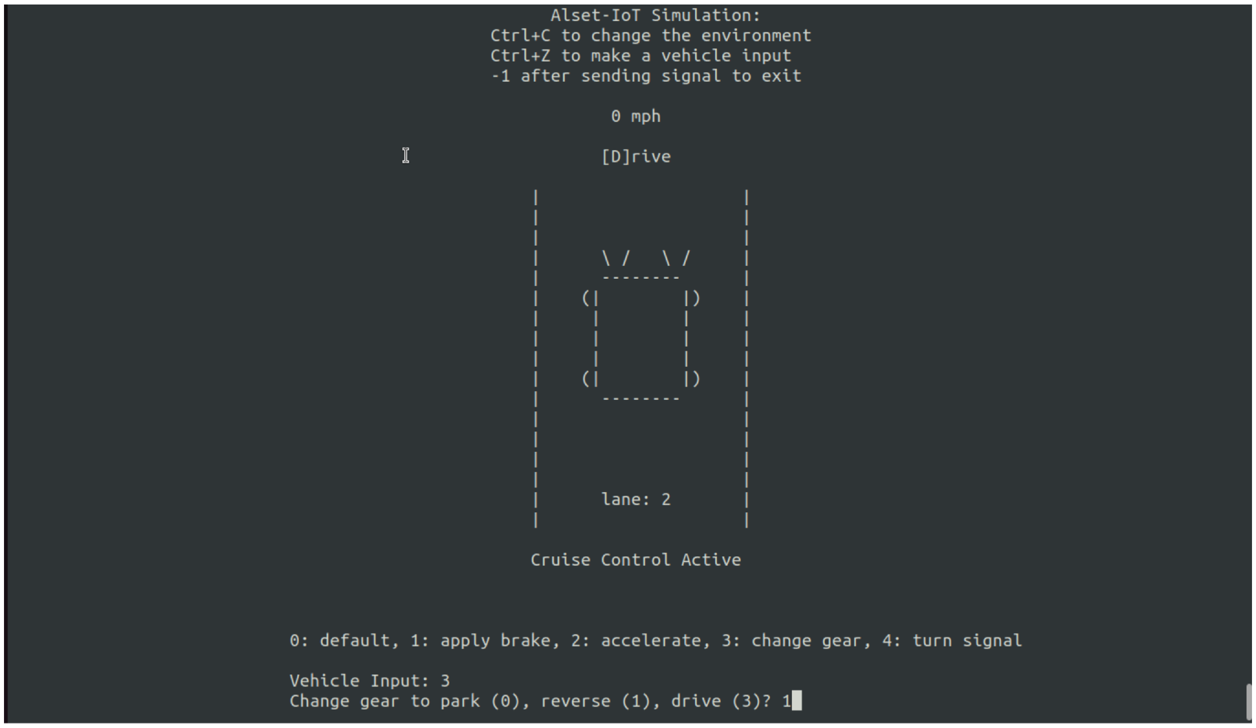
Satisfied in the following test:

- Run the default state of the car
- Ctrl+Z and 1 to apply the brake
- Set to a low velocity (<3 mph)
- Once at 0mph, Ctrl+Z and 3 to change gears
- Input 1 for reverse

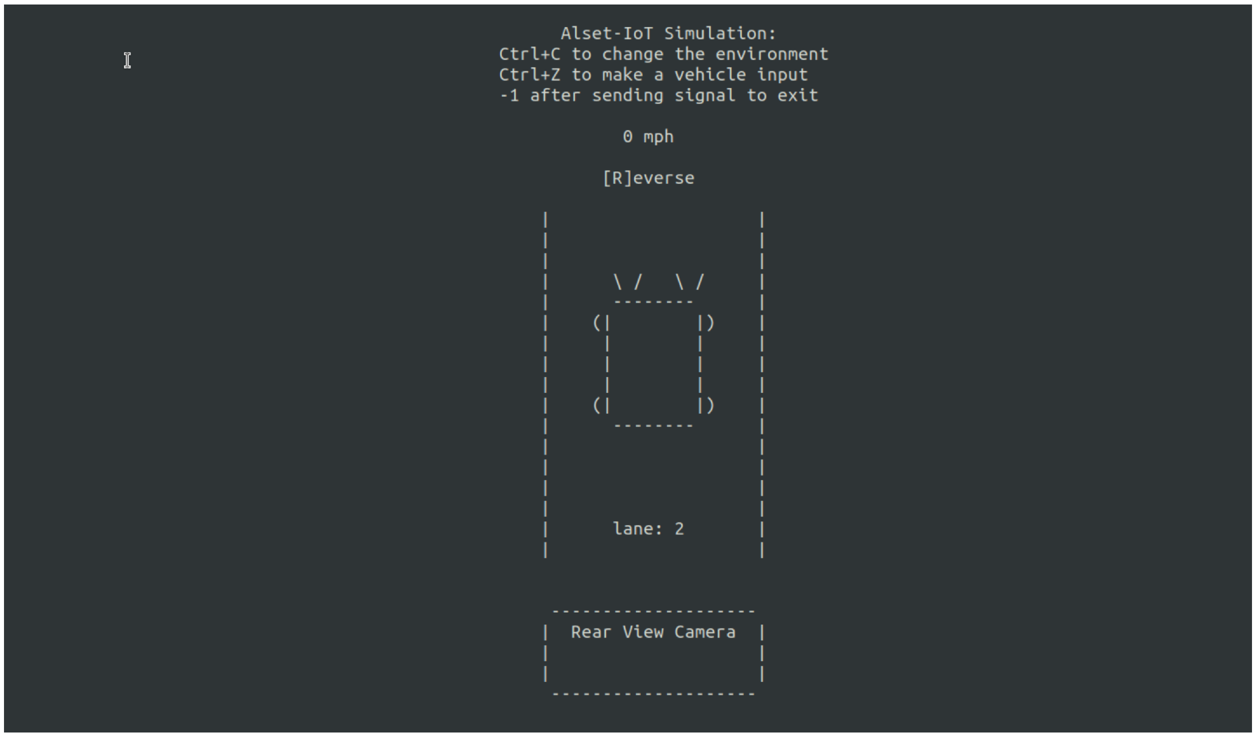
- The rear view camera will be automatically displayed



```
                     Alset-IoT Simulation:
                 Ctrl+C to change the environment
                 Ctrl+Z to make a vehicle input
                 -1 after sending signal to exit

                            54 mph

                           [D]rive

              |                    |
              |                    |
              |     \   / \   /    |
              |      \ /   \ /     |
              |      --------      |
              |    (|        |)    |
              |     |        |     |
              |     |        |     |
              |     |        |     |
              |    (|        |)    |
              |      --------      |
              |                    |
              |                    |
              |                    |
              |                    |
              |      lane: 2       |
              |                    |

               Cruise Control Active


0: default, 1: apply brake, 2: accelerate, 3: change gear, 4: turn signal

Vehicle Input: 1
Brake to what speed? 0
```

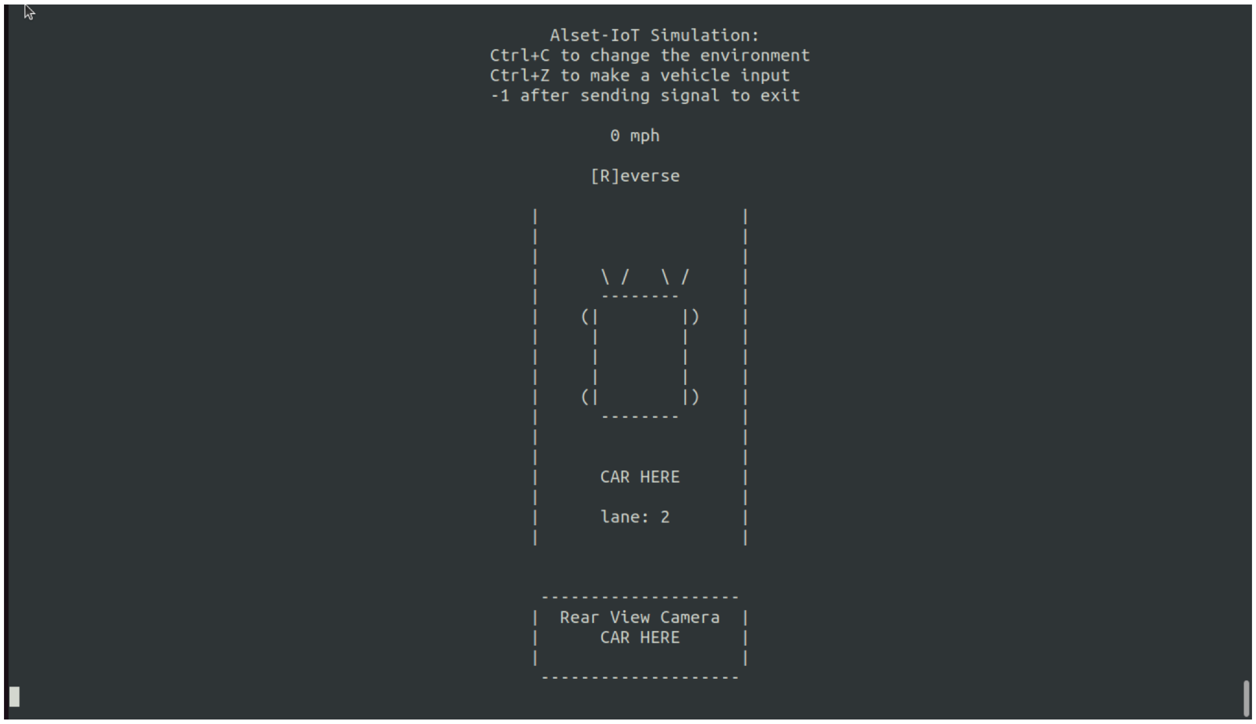


```
                     Alset-IoT Simulation:
                 Ctrl+C to change the environment
                 Ctrl+Z to make a vehicle input
                 -1 after sending signal to exit

                            0 mph

                           [D]rive

              |                    |
              |                    |
              |                    |
              |      \ /   \ /     |
              |      --------      |
              |    (|        |)    |
              |     |        |     |
              |     |        |     |
              |     |        |     |
              |    (|        |)    |
              |      --------      |
              |                    |
              |                    |
              |                    |
              |                    |
              |      lane: 2       |
              |                    |

               Cruise Control Active


0: default, 1: apply brake, 2: accelerate, 3: change gear, 4: turn signal

Vehicle Input: 3
Change gear to park (0), reverse (1), drive (3)? 1
```



```
          Alset-IoT Simulation:
     Ctrl+C to change the environment
     Ctrl+Z to make a vehicle input
     -1 after sending signal to exit

               0 mph

             [R]everse

     |                     |
     |                     |
     |                     |
     |      \ /   \ /      |
     |      --------       |
     |    (|        |)     |
     |     |        |      |
     |     |        |      |
     |     |        |      |
     |    (|        |)     |
     |      --------       |
     |                     |
     |                     |
     |                     |
     |                     |
     |      lane: 2        |
     |                     |


      -------------------
     |  Rear View Camera  |
     |                    |
     |                    |
      -------------------
```

The Vehicle will turn on the rear view camera when in reverse. Additionally, because there is a Car behind, it shows a Car in the rear view camera. This is shown below:



```
          Alset-IoT Simulation:
     Ctrl+C to change the environment
     Ctrl+Z to make a vehicle input
     -1 after sending signal to exit

               0 mph

             [R]everse

     |                     |
     |                     |
     |                     |
     |      \ /   \ /      |
     |      --------       |
     |    (|        |)     |
     |     |        |      |
     |     |        |      |
     |     |        |      |
     |    (|        |)     |
     |      --------       |
     |                     |
     |                     |
     |      CAR HERE       |
     |                     |
     |      lane: 2        |
     |                     |


      -------------------
     |  Rear View Camera  |
     |      CAR HERE      |
     |                    |
      -------------------
```

The vehicle will also not accelerate into a car behind it and will adaptively brake if a car is less than 20 feet behind, similar to when driving with a car closer than 100 feet.

10. **Parking sensors**

Satisfied in all Use Cases, as in a variety of cases there are cars displayed around the cars, regardless of the gear it is in.

References

Guo, Qiaochu. “Software System of Autonomous Vehicles: Architecture, Network and OS.” University of Pennsylvania, School of Engineering and Applied Science, 29 April 2022. Accessed 19 February 2023.

Pressman, Roger S. and Bruce Maxim. “Software Engineering Chapter 8: Requirements Modeling - A Recommended Approach.” *McGraw Hill*. Accessed 26 March 2023.