CSCI5302 Final Project: Team Online Section

Andreas Brecl, Benjamin Spicer, Daniel Mathews,
Hannah Quirk, Micah Zhang, Sean Newman, and Xiaojun Yin

***Abstract*— Numerous companies, universities, and independent researchers have begun exploring implementation of self-driving vehicles. Currently, the state of the art autonomous vehicles that are available for consumers to purchase and drive still require human-in-the-loop interaction, where the driver is able to intervene in critical situations, and are not fully autonomous. Still, the prospect of full autonomy on cars has driven large tech companies to pour millions of dollars into research and development. To explore autonomous driving on a smaller scale, this paper explores the implementation of a 1/10th-scale autonomous vehicle that is able to drive without human intervention or assistance around a closed course. This car uses a combination of computer vision and onboard sensors to simultaneously track its position on the course and adjust its trajectory depending on its current position.**

I. INTRODUCTION

To better understand the functionality of autonomous vehicles and the various implementations that can be used for self-driving cars, a $\frac{1}{10}$th-scale RC car with autonomous navigation capabilities was developed. This car was designed to be able to complete laps around a closed track, which is a series of hallways, without any operator intervention. The vehicle uses a combination of computer vision and sensors that allows the onboard microprocessor to send control signals to the car's motors in order to adjust its trajectory and navigate the hallways.

An ODRIOD UX4 microprocessor and Intel RealSense Camera were mounted on to a stripped-down RC car body in order to implement the autonomous navigation. The camera, which is capable of taking depth information, feeds data to the ODROID, which then sends control inputs to motors and servos connected to an electronic speed controller that allow the car's wheels to drive and turn. The control has the car drive toward the deepest point detected by the RealSense camera, thus allowing it to navigate the long hallways on the racetrack. The car also uses an inertial measurement unit (IMU), which allows the ODROID to track the car's relative position. These hardware components also allowed for completion of various challenges for the competition. Input from the camera and the IMU and control signals to the system were implemented and processed using ROS running on Ubuntu 18.04.

While there were many implementation challenges, the team was able to successfully design a car capable of meeting the project criteria. The car consistently completes laps around the track at an average of 30 seconds, which is well within a good time for the competition guidelines. The car is also able to complete the challenges for the project, and has multiple safety measures in place to stop the car in case of software failure.

II. RELATED WORK

The team primarily looked at documentation and past submissions to the "Formula $\frac{1}{10}$th" or F1TENTH competition [1] for inspiration on this project. The F1TENTH competition, similar to this project, asks participants to develop an autonomous $\frac{1}{10}$th-scale RC car. While the actual racetrack and challenges the cars for the F1TENTH competition has to complete are different from this project, the design of the vehicle and the hardware used is very similar. The website has thorough documentation on methods used by teams in the past and recommended starting points, so it gave a good background on the project for the class. After gaining a general understanding of the project from this documentation, specific methods for implementation were explored by reading documentation for the Intel RealSense camera [2] and documentation for various ROS packages. A few research projects with thorough documentation were explored, such as a study done by Feng on visual SLAM for a unmanned ground vehicle [3]. While many of the methods from these sources were not directly implemented, they provided a better understanding of the project.

III. METHODOLOGY

The following section describes a detailed overview of the process the team took to designing, integrating, and testing a fully autonomous and functional race car. There are a number of components and subsystems which must be developed to validate the performance of the vehicle which are explained in detail in the following subsections.

A. Hardware Overview

- ODROID UX4 with 2GB memory
- Intel RealSense D435
- 1/10 AMP MT 2WD Monster Truck RTR ECX03028T2
- Mini Maestro 18-Channel USB Servo Controller
- PhidgetSpatial 3/3/3 Basic 1042.0
- 64GB MicroSD Card
- USB Hub
- 7.2 V Ni-MH Battery
- 14.2 V Li-Po Battery
- Brushed Electronic Speed Controller (ESC)

B. Software

Due to compatibility with software libraries, the team's preference in operating system, and overall heritage of software implemented for robotics, the team decided to use the

Linux distribution Ubuntu 18.04 in conjunction with ROS Melodic running on the ODRIOD UX4.

C. System Overview

The system is powered with two batteries. The first battery is a 7.2 V Ni-MH battery that provides power specifically to the ESC, DC motor, and servo controller. The ESC from this battery power, provides 5V regulated power to the servo controller and varying output to the DC motor to control it's speed. The second battery is a 14.2 V Li-Po battery that provides power for the ODROID. This system is directly plugged into the buck converted which provides a regulated 5 V output to the ODROID. From the ODROID, power is distributed to the rest of the system. The inertial measurement unit (IMU), and the Intel RealSense D435 camera are directly powered via USB. From the servo controller, the PWM signals also carry power to the steering servo. A top-down view and a side view of the completed car are shown in the figures below.

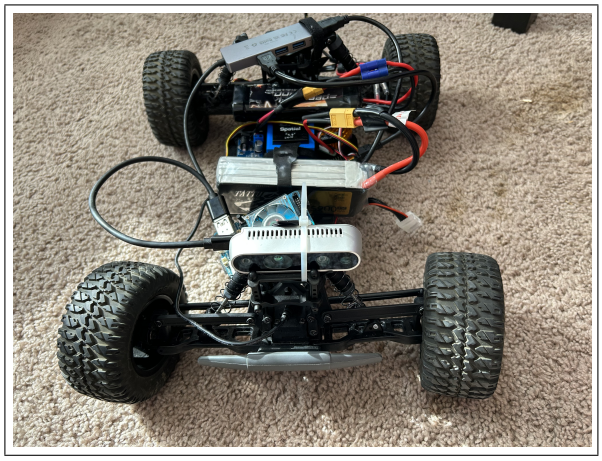

Fig. 1. Front View of Completed Car

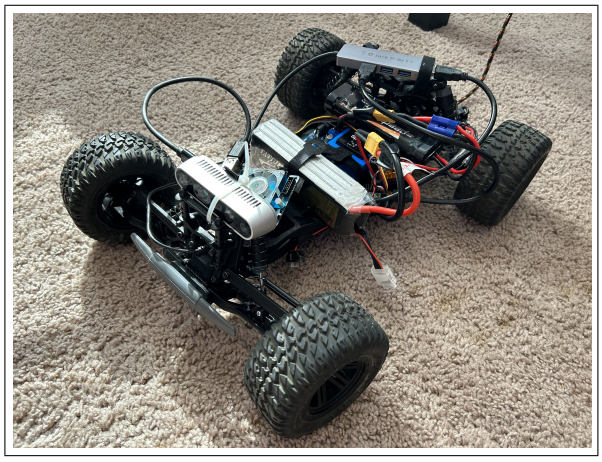

Fig. 2. Side View of Completed Car

The IMU, servo controller, and the RealSense camera all feed data to the ODROID via USB. The RealSense camera gives video data and depth measurements which are used for the autonomous driving. The IMU gives gyro, accelerometer, and magnetometer data, and the servo controller directly relays IR distance sensor data to the ODROID. The servo controller sends PWM signals to the servo and ESC to have the speed of the motor changes and the angle of the servo.

The ODROID is responsible taking all of the collected data to determine the next action of the vehicle. Once the ODROID determines each action to take, it sends this information via USB to the servo controller to change the servo and motor positions. Figure 3 below highlights how each component on the system is connected, and how power and data are distributed throughout the system.

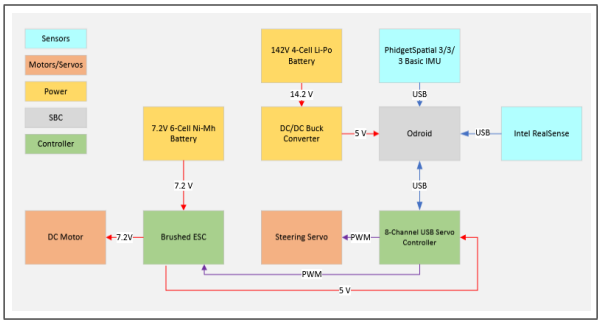

Fig. 3. Functional Block Diagram of the Full System

D. Servo and Motor Controls

Operation of the servo and motor is critical to the motion of the vehicle. The servo provides vehicle steering and the motor provides a means of thrust. To control both of these items, the Pololu Maestro servo controller was used to send PWM commands to the motor's ESC and the servo. The ESC provides control to the motor based on this PWM command. Different versions of Pololu boards have varying PWM values that correspond to different actions. For this particular board, the PWM range values were 4000 to 8000 with a mid point of 6000. This was useful for determining the expected output of the system. For the Polulu board to operate, it must receive a 5V power from some source. This power source can be provided from the ODROID or the ESC. This car uses the ESC 5V pin to power the Pololu, as this simplifies wiring and avoids the need for the use of extra wires.

For operation of the servo, the following results were observed. When the PWM value was set to 6000, the servo sat a straight forward facing value. This means the vehicle moves straight at this setting. The boundary values, 4000 and 8000, corresponded to a maximum left and right turn. These turns were approximately equivalent to 25 and -25 degree angles. When the PWM values linearly changed, the angles the wheels were at also linearly changed. This means that the turn values could be linearly fit between the PWM values and the expected turn angles. However, when the vehicle is on the ground, it cannot overcome the friction it is experiencing effectively when making small angle adjustments. In other words, the wheel cannot turn as expected due to the friction between the rubber and the

ground. This means that realistically, the vehicle will only see true turning changes on intervals of about 5 degrees.

For the operation of the motor, similar results were seen. When the PWM value was set to 6000, the motor sat at a neutral value so it would be stationary at this value. The boundary values, 4000 and 8000, were full reverse and full throttle. Though this is the case, this result is not as straight forward to achieve. The ESC waits for values received to calibrate it's neutral, full throttle, and full reverse values. This means that when the vehicle starts, it must receive a 6000 value, then 8000 value, and then a 4000 value to properly calibrate the vehicle. If this is not done, the vehicle will go at wildly unpredictable results when launching the system. For this reason, the ESC was calibrated before each test run of the vehicle.

E. Motion Planning with Computer Vision and IMU

The motion planning techniques used for navigating and guiding the vehicle through the race course relied on using depth imaging from the RealSense camera and data readings from the IMU. There were multiple packages and ROS nodes that had to be generated to parse and communicate this data across the system's hardware. The motion planning is done via a Python script which generates a node in itself. This script follows a publish/subscriber messaging pattern to relay data. It handles motion planning based on the output from the depth information of the D435 camera. It subscribes to the depth image topic from the D435 and an Inertial Measurement Unit (IMU) topic, and it publishes to two different topics: control command, and control difference.

The script utilizes several Python modules. The most important modules are rospy, cv2, Imu and CvBridge. These modules are used to facilitate communication between different ROS nodes, utilize OpenCV for handling image data and conversions, perform mathematical operations, and manipulate arrays. The raw depth data being published by the Intel RealSense camera was first analyzed in order to better understand the data being published while the camera was taking information. This was done by taking bag files from the car during operation. The bag files collected were able to be replayed, and provided a visual display of the camera data. This was used to debug the various Python scripts being used for motion planning. An example screenshot of the depth image and camera information is shown below, visualized and played-back utilizing a web based program called Webviz.

This image from a collected bag file gave the team confidence that depth information may be used for motion planning of the vehicle through the hallways of the engineering center. As can be seen from the image, there is a large white section displayed which represents the maximum distance down the length of the hallway. The team decided to use this information to generate a turning control law that would guide the car towards the center of this maximum depth area. This was done by first tuning the image above to generate a different type of resolution that would better suite the hallways and be more consistent for the operation of

Fig. 4. Initial visual image of the depth data.

the vehicle. This tuning was done by transforming the depth data to an OpenCV image and changing the scaling values to something that would be represented by the next image below. The image was cropped to the bottom half of the original image to eliminate any possibility that the camera would be reading unwanted information or being misguided by windows or classrooms.The image below shows the outline of the contours around the maximum depth contour. These contours are what guided the car as the middle point was taken as the point the car was attempting to go towards. The proved to be an effective strategy for the straight portions of the track as the contour would be very consistent and the car tended to stay close to the middle of the hallway at all times. Another key feature of this image below is the diagnostic information our team was outputting on the side of the Webviz interface. The *control_cmd* and *contour_w* were being published based on the values that we were calculating for the car's velocity, angle, and width of the contour. This information was crucial to see when tuning and debugging the performance of the vehicle during the training process.

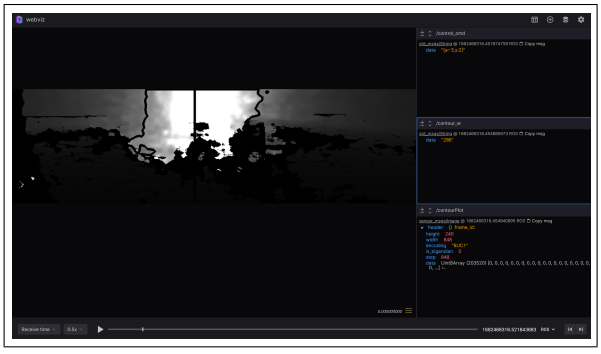

Fig. 5. Adjusted depth data image and diagnostic information.

The next challenge when implementing the motion planning is the detection and activation of turning when approaching the end of a hallway. Various ways of detecting when to turn were tested before the final implementation was decided. Depth data, contour data, and IR sensor integration were all tested. When looking at the depth images as the car approached walls, the contour values would drop significantly lower. The scaling of the depth image would eliminate the contour in most cases as the wall in front of the car was

close enough. This can be seen in the figure below. At each turn on the race track, the same thing would occur with the contour values. Therefore, the system was programmed to turn if a contour width of less than 100 would occur more than three times in a row. This method works very well for the racetrack used for this competition, as the racetrack is a series of straight halls connected by 90-degree turns. Because this implementation worked well, other methods, such as using the IR sensor, were taken out of consideration due to their increased complexity in implementation.

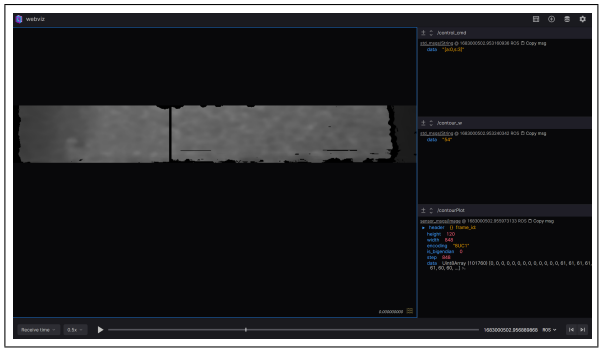

Fig. 6. Depth information when approaching the end of a hallway.

The scripts used for motion planning perform the following sequence. They first check if both IMU data and image data are available. Then, if either is missing, the method returns without doing anything. Otherwise, the script performs several operations on the depth image to modify and extract contour values. It find the largest contour, and calculate the angle and velocity of the contour's center point relative to the camera's field of view. Doing so allows the vehicle to perform a depth following algorithm where it follows to middle of the largest depth contour which works great for following through hallways on straight sections of the track. The script also checks if the vehicle is currently in a turn or a straight section of the track and adjusts the angle and velocity accordingly.

If the vehicle is in a straight section, the method sets the velocity to a much faster constant value to decrease the time. If the vehicle detects a wall infront of it, and enough time has passed since the previous turn, the method sets a boolean flag to switch to a turn state, calculates the current yaw angle using the most recent IMU data, sends a turn command to the servo controller with an angle of 25 and a slower constant velocity value. If the vehicle has turned enough to reach a target yaw angle, then the vehicle returns to the normal state of following the depth data through the hallway.

F. Operation of Vehicle

Once all of the capabilities needed for autonomous navigation were completed, the car was tested in the racetrack at low speeds to ensure functionality. A video of the car being tested at low speeds can be found here. It can be seen that the car is able to reliably take turns and navigate the course. After this trial was successful, the speed was slowly increased in order to test the maximum speed that the navigation algorithm could perform at.

In order to operate the vehicle, various shell scripts were created. First, a script was created that would calibrate the ESC to the proper values after a reboot. This script launches roscore, then sets up a server node that acts as the interface between ros and the servo controller which controls the servo and the motor. Finally, it sends a command for the neutral value of the controller, which is 6000, then the maximum forward value, which is 4000, back to a neutral value, then to the maximum reverse value, which is 8000, then finally back to the neutral value. This needs to be run after any restart, and if the battery has been unplugged from the ESC. The next script created was the launch script. This script launched the nodes for the camera, IMU, motion planning, coefficient of friction calculation, and servo server. Additionally, this script also changes the permissions on all the files so that they are executable as whenever a change were pulled from the git branch, it reset the file permissions to not be executable. Finally, a kill script was created that kills all of the nodes that the launch script launches using the pkill command in Linux.

IV. RESULTS

The performance of the car was measured based on how quickly it could complete laps on the racetrack, how reliable the system can perform, and how well it can complete various challenges. The final design of the car is able to complete laps reliably around the racetrack at about 30 seconds. A video showing the car driving around the course at its maximum speed can be found here. The challenges selected for this project explored various capabilities of the autonomy stack and hardware on the car. First, the car was developed to back out of a collision and continue driving. Image detection was used to track the car's position to an obstacle, and then control was implemented for the car to be able to reverse out of the obstacle and turn to avoid it when proceeding forward. A video of this challenge being completed can be found here. A sparse map of the racetrack was also developed in order to demonstrate mapping capabilities, which can be used for motion planning and simultaneous localization and mapping (SLAM). The sparse map was produced using the RTAB-MAP ROS package. A real-time estimate of the coefficient of friction was calculated by tracking inputs from the IMU during the drive, and computer vision was used to allow the car to stop at stop signs. A video of the car stopping at a stop sign can be found here.

On competition day, the course used was different from the original course in the project plan. The team's car was able to complete a lap on the new track at approximately ten seconds. A video of the car racing on the new course on competition day can be found here.

A. A-Type Challenge: Backing out of a Collision and Continuing Driving

For our A-type challenge, we decided to back out of a collision and continue driving. This challenge involved a

few main components. First, the car needed to be able to determine when a collision occurred. Next, once the car has determined a collision occurs, the car needs to back out of the collision, Finally, the car has to wait for the obstruction to be cleared before continuing with its lap. For this challenge, the speed of the car was lowered significantly, but the motion planning algorithms were kept the same. Since this challenge requires getting into a collision, we wanted to avoid getting into collisions at our top speed where the components could be damaged. In order to get data, a sample collision was performed by running the car into an empty recycle bin at 50 percent speed. These results can be seen in figure 7 below:

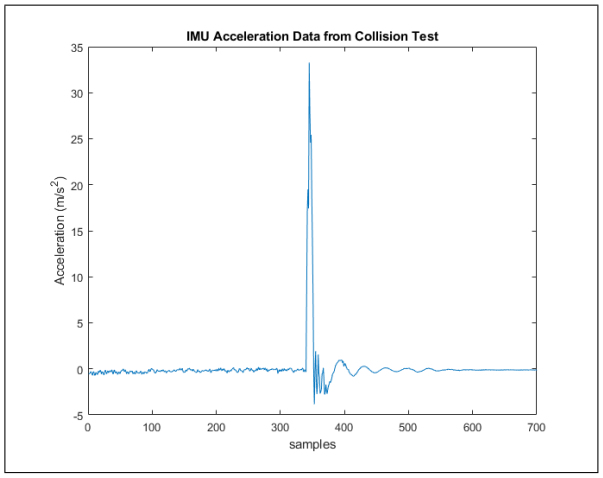

Fig. 7. IMU Acceleration Data from a Collision

During nominal operations, the motion planning system saves the most recent orientation of the car via the filtered IMU data. In order to determine when a collision occurs, the system is constantly monitoring acceleration data from the IMU. Once the IMU detects an acceleration of 4 m/s^2 or greater, a system flag is set that a collision occurs. Once the motion planning system sees the collision flag, it begins to back up. At this point, the motion planning software compares the current orientation of the car via the filtered IMU data, and compares it to the orientation of the car before the collision. The car will then back up at an angle and attempt to get the car oriented to within 5 degrees of the orientation before the crash. This method for backing up, rather than just backing up in a straight line, was chosen in case the collision significantly altered the orientation of the car (i.e. if the car only hit the corner of the obstacle and turned towards the wall). Once the car has completed backing up, it waits for the obstruction to clear before continuing. The flowchart for this challenge can be seen in figure 8.

B. B-Type Challenges: Developing an Accurate Sparse Map

For our sparse map, we used the RTAB-Map (Real-Time Appearance-Based Mapping) algorithm. RTAB-Map is a graph-based SLAM approach developed by Mathieu Labbe. It uses an incremental appearance-based loop closure detector with a bag-of-words approach to make a hypothesis on whether a given input image comes from a previously

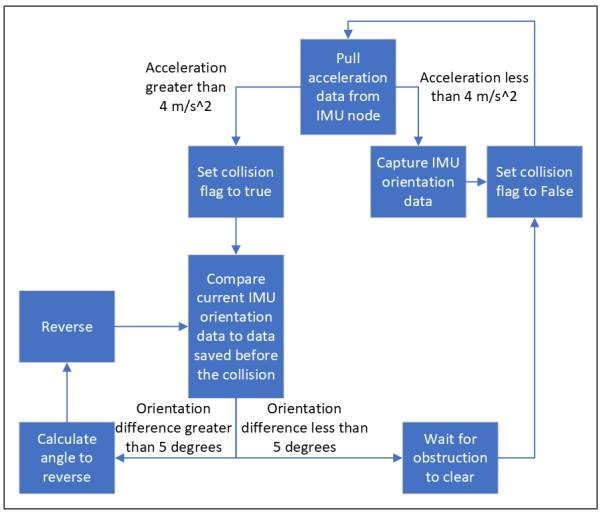

Fig. 8. Code Flowchart for Collision Detection and Continuing Driving

tracked location or from a new location. Each new loop closure hypothesis is used to add a constraint to a map graph, and then this map graph is filtered via a graph optimizer to ensure that an accurate sparse map is produced. RTAB-Map is capable of using RGB-D, stereo depth, and lidar as input, and is a widely used SLAM approach.

We chose to use RTAB-Map because it is the SLAM algorithm recommended by Intel to be used with the D435i in their realsense-ros wiki page. We had originally wanted to the use the OrbSlam 2 algorithm. However, we found that Orb-Slam performed poorly, losing tracking as soon as the camera moves by even a slight amount. Since the precompiled binaries for RTAB-Map only supports 64-bit machines and the Odroid uses a 32-bit architecture, we had to build RTAB-Map from source. Since the build process requires a large amount of RAM, we had to use a swap file to give the Odroid enough memory to complete the build. We also installed the ROS package for RTAB-Map. Our launch file for the sparse map, based on `opensource_tracking.launch`, consisted of 5 stages. The first stage is to launch our phidget IMU. The seconds stage is to launch the Intel RealSense D435 camera, specifying that both IR cameras are to be enabled. The third stage uses the `imu_filter_madgwick` ROS package to filter and fuse the raw data from the Phidget IMU. The fourth stage is to launch RTAB-Map. While the original launch file uses RGB-D based visual odometry. However, during our testing we found that RGB-D provided poor quality odometry data that was insufficient for adequate tracking. Consequently, we later switched to using the two IR cameras on the D435 to provide visual odometry data via stereo vision. To do this, we had to pass the rectified stereo image topics from the two IR cameras along with their info topics to RTAB-Map, and to tell it to use stereo vision. The fifth stage of the launch file is to use the `robot_localization` ROS package to fuse the IMU and visual odometry data. To configure `robot_localization`, we used the ukf template together with the default parameters recommended by the Intel RealSense documentation. With the 5 stages, our

launch file allows us to fuse the filtered data from an external IMU together with the visual odometry data provided by the stereo IR cameras on the D435 and to be fed to the RTAB-Map SLAM algorithm, which then uses this data together with the image data from the D435 camera to create a sparse map.

Figure 9 shows the computation node graph for our sparse map code. The top left section of the node graph corresponds to the D435 camera, and shows how the `realsense2_camera` node and its manager node communicate with one another. The bottom middle section of the node graph corresponds to the IMU, and shows how the Phidget IMU feeds data to the IMU filter node, which then sends the filtered IMU data to the IMU manager. The middle right section of the node graph corresponds to RTAB-Map, and shows how it receives the filtered IMU data from the IMU manager and rectified infrared images from the realsense camera to the rtabmap node, the stereo odometry node, and the ukf robot localization node. The stereo odometry node uses the images from the camera to produce odometry data, the ukf node fuses the IMU data with the odometry data, and the rtabmap node is ultimately what creates the sparse map.

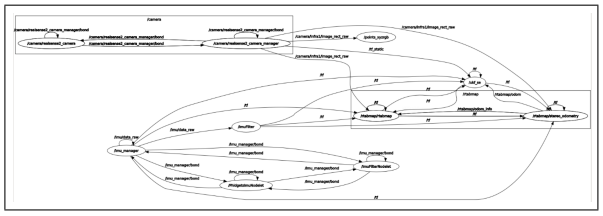

Fig. 9. Sparse Map Computation Node Graph

Figure 10 shows an visualization of the sparse map produced our RTAB-Map via our modified launch file. This visualization was produced in the RViz app, launched by the RTAB-Map package. On the bottom left is a view of the raw image received from the D435 camera. In this case it shows the table our project team worked at in the basement of the Engineering Center. The top left shows the configuration for our sparse map set-up. All nodes are using the `camera_link` fixed frame. The sparse map itself is a `voxel_cloud` topic produced by the rtabmap node. This voxel cloud is what is displayed in the center of the figure. Each pixel received from the realsense camera is assigned a position within the 3D grid. Comparing the raw image with the voxel cloud, we can see the laptop bag, TV, and whiteboard as three distinct objects within view. The locations and relative distances of these three objects within the sparse map correlate well with their real-world locations, indicating that the sparse map is highly accurate. There are two T-shaped markers near the center of the grid. These markers correspond to the left and right infrared cameras on the D435, hence why they are faced back-to-back to one another. The small cloud of points in the top right section of the grid corresponds to noisy data near the peripheral vision of the camera.

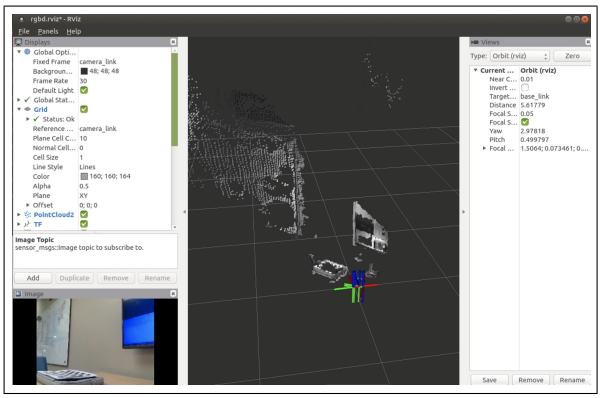

Fig. 10. RTABMAP Visualization in RViz

Figure 11 shows the sensor drift we experienced when using RTAB-Map, which we believe is caused by poor alignment of the IMU and visual odometry data, which resulted in loss of tracking with prolonged movement of the camera. To elaborate, unlike the previous two sets of T-shaped markers. The first set of markers corresponds to the initial location of the camera, which is treated as the origin of the sparse map. The second set of markers corresponds to what RTAB-Map believes is the current location of the camera. However, looking at the raw image from the camera, we can see that since the camera hasn't actually moved at all, this believed location produced by RTAB-Map is incorrect. During our testing we noticed that this phenomenon, known as sensor drift, tended to happen whenever camera was moved.

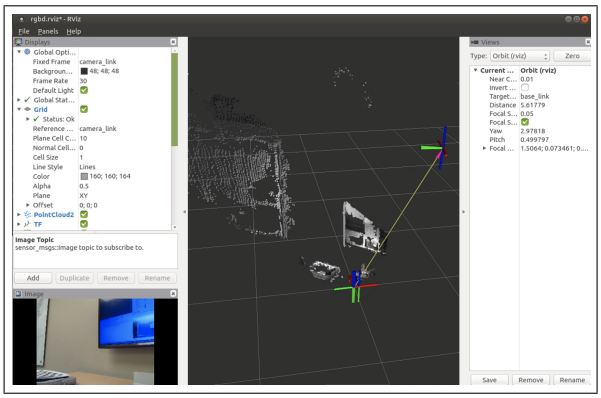

Fig. 11. Sensor Drift via Misaligned IMU and Odometry Data

During testing we found that RTAB-Map was extremely vulnerable to sensor drift and not sufficiently reliable to use for creating a sparse map of the racetrack. Neither walking with the camera nor running the car at its lowest speed were enough to avoid sensor drift and produce a usable sparse-map of the race track. We tried extensively to diagnose and resolve this problem but were ultimately unsuccessful. Consequently, we decided to pivot to stop-sign detection as our replacement B-objective. Since we were able to show that each component of the sparse map algorithm was able to function on its own (i.e. the stereo odometry produced high quality odometry measurements and the IMU produced

high quality filtered data, both a sufficiently high rates), we believe that the issue is with the robot localization node, which appears to be unable to adequately fuse the data from our external IMU with the odometry data from the stereo cameras, resulting in poor quality fused localization measurements that are published at a low intermittent rate. It is possible that with additional time, experience, and requisite knowledge that we could improve the fused localization quality, but it was a valuable learning experience regardless.

C. B-Type Challenges: Real-Time Estimation of the Coefficient of Friction

For one of our challenges, we decided to develop a real-time estimation of the coefficient of friction. This challenge involves building a basic dynamics model of the system, then paring this dynamics model with the filtered output of the IMU data to calculate the coefficient of friction. This challenge will allow us to see how the vehicle's traction varies throughout the track.

To model the friction, we first must model the direction of our system in a two-dimensional field. From this we can calculate the total force. Then we can substitute the coefficient of friction force equation in. Then finally solve for the μ value. This is was the derivation looks like.

$$F_x = m * a_x \tag{1}$$

$$F_y = m * a_y \tag{2}$$

$$F_t = \sqrt{F_x^2 + F_y^2} \tag{3}$$

$$F_n = m * g \tag{4}$$

$$\mu = \frac{F_t}{F_n} \tag{5}$$

The value for m or mass can be measured and g is our gravity term. The accelerations or a terms can be pulled from the /imu/data topic. With all of these values, an estimate can be formed for the coefficient of friction. For this value to be real time, it must be calculated at each received data point outputted by the IMU. With this implemented into a ROS node, the following results are achieved.

As can be seen in the figure above, the estimated coefficient of friction is non constant. An initial flat period can be seen. This is when the vehicle is stationary waiting to start. Once it starts, an initial peak can be seen. This peak corresponds to the initial acceleration of the vehicle getting to it's operating speed. Once it reaches the operating speed, the vehicle reaches a more constant value. This is due to the vehicle coasting and having minimum acceleration during this period. After this, another spike in the data value occurs. This is due to the turning around the corner action of the vehicle. As the vehicle approaches a corner, it decelerates and starts turning. This interaction causes an increase in the coefficient of friction. This leveling then occurs again as the vehicle reaches it's straight down the hallway speed. Once again after this, a turn occurs causing the spike in friction data. To get an accurate estimate for the coefficient of friction, only the data points from where the car was actually driving were used. In addition, the data was smoothed in order to filter out the spikes caused by the turns, as these coefficients were likely high due to wheel sliding when turning and an imbalance in the direction of the normal force. The smoothed data is shown below. The average coefficient of friction was found to be around 0.2.

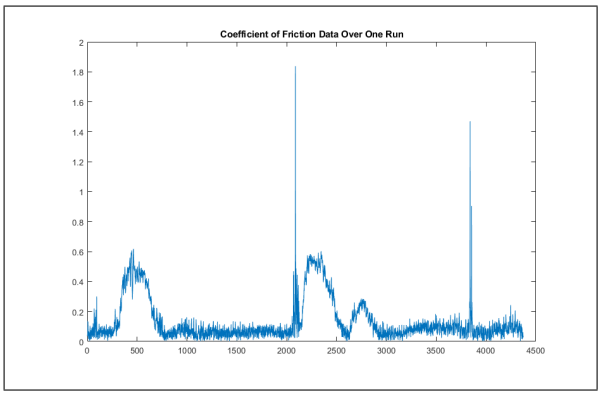

Fig. 12. Live Estimate of Coefficient of Friction

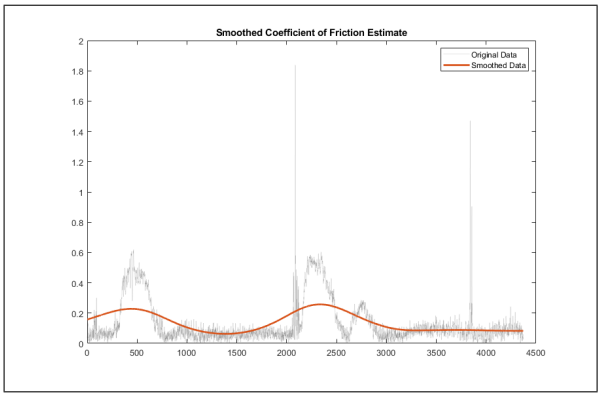

Fig. 13. Smoothed Coefficient of Friction Data

D. B-Type Challenges: Stop Sign Detection

The Stop Sign Detection challenge involved detecting the image of a stop sign using the onboard sensors, briefly bringing the car to a stop, and then continuing onward. This is done on a frame-by-frame basis using solely the RGB output of the RealSense camera and the OpenCV library, where processing every frame consists of three steps.

First, the image is downscaled to 35% of its original size to reduce the amount of processing required while still retaining enough information to have accurate results. Next, the image is filtered to only where there are bright red pixels. Lastly a series of erosions and dilations are applied using a 4 by 4 circular kernel to clean up the image and OpenCV's Simple Blob Detector is run on the results. If a blob is detected it is assumed to be a stop sign and the car comes to a stop for a

moment, and then continues, ignoring all detected stop signs for 5 seconds while it passes by the current sign.

The blob detector itself is controlled by a number of parameters that allow for further specification of which blobs are to be accepted. The most useful ones for the stop sign detection were the Filter by Area, which eliminated any stray pixels that may have made it through the preprocessing, and Filter by Inertia, which eliminated any blobs that were too non-circular. Parameters were tuned by hand in a number of different lighting conditions to provide the most accurate results.

Fig. 14. Original Image with Detected Blob Shown

Fig. 15. Image After all Postprocessing

The results of the detection were imperfect, but good enough to be usable. In particular, as shown in Figure 14, the detection location may be slightly off from the true location of the stop sign. When using a higher resolution image, this issue appears to be fixed, but it would make the algorithm too slow. The detection is also not particularly robust against lighting changes, as the color mask was only tested in a few lighting conditions. Lastly, the detection can also be confused by large red objects that may, to the naive algorithm, appear like stop signs. This may be improved by using contour approximation to check if the detected blob is an octagon.

E. C-Type Challenges: Reports

See VI for the completion of type-C challenge reports on human-robot interaction for autonomous vehicles and how deep-learning could be used to improve performance.

V. DISCUSSION

The designed scaled autonomous car is able to race around the race day competition track with an average time of around 10 seconds. The car was able to do this by using a depth-following algorithm in which the car follows the deepest point in the images detected by the onboard camera. This algorithm took image outputs from the Intel RealSense camera, and used OpenCV to mask it in order to trace out the deepest point in the track. This depth-following algorithm works incredibly well for the given track, which has many straight sections and only four turns. However, it the car was being raced on a more complex track, a different approach may have been taken.

Originally, the team had planned to use a sparse map generated by the Intel RealSense camera coupled with a mapping package, such as Orbslam or RTAB-MAP, in order to do simultaneous localization and mapping (SLAM). However, it was quickly realized that this is both very complicated to implement with the given hardware stack, and incredibly slow for a competition in which speed and reliability of the car are the most important factors. When running the packages with the given camera, the updates on the map occurred very slowly even after tweaking various parameters in the mapping packages. As the map generation itself took far too long for the competition constraints, doing motion planning on top of that was out of the question. This led the team to pivot to the depth-following algorithm mentioned above, which is incredibly fast and well-suited specifically for this project and this racetrack.

In general, setting up the systems and packages needed for the project proved to be more difficult than expected. The hardware originally received for the project ended up being slightly different than what was needed, and it took setting up the car to realize this. While this was resolved simply by reaching out to the TA for hardware components, it was definitely a setback as realizing the part was missing in the first place took time. Next, setting up the ODROID with Ubuntu 18.04 and the packages needed for the project was difficult. There were many packages that did not launch either on the ODROID or on the selected OS. For example, some Intel-developed RealSense packages could not launch. This was worked around by using other packages that performed similar tasks, or by feeding the camera data into a separate laptop to check functionality and to calibrate the camera before mounting it on the car.

Finally, testing the functionality of the car came with challenges as well. First, the car was developed while on the school's internet. This meant that the IP address that teammates would use for remote connection changed, and had to be tracked. Then, when the car was initially tested, it was soon discovered that the school's WiFi network did not extend along the entire length of the hallway racetrack.

This was an issue, as the car crashed into walls when it lost connection with the internet. To resolve this, a mobile hotspot was setup, and a team member ran around the track with the car to ensure it maintained connectivity the entire time. There were also many tests in which the speed was set high and the image detection or another script running failed, and the car crashed into the wall. This was something that had to be prevented at all costs, as it could catastrophically damage the onboard hardware. To protect against this, a rope was tied to the car. A teammate would run with the car while holding the rope in order to stop it in case of a crash. This was coupled with the ability to kill the car's driving process via remote connection on a computer.

Ultimately, the development of the autonomous car came with numerous setbacks that were incredibly frustrating. Many of these setbacks, including the ones mentioned in this section, felt trivial but very time consuming. While the solutions were very clear in hindsight, at the time, they were not so obvious. Later into the project, the most important factor that led to the successful completion of the project was repeated testing under different conditions, and tweaking various scripts for the autonomous driving to make sure the driving was reliable.

VI. CONCLUSION

In conclusion, the objective of the project was successfully achieved by developing a 1/10th scale autonomous driving car that was able to complete a lap around a hallway without human control. The car's autonomous driving algorithm, in which the car navigated toward the deepest point the camera detected using OpenCV, was successful. Some challenges included hardware and network problems, and software incompatibilities, were overcome by either pivoting approaches or finding workarounds for the purposes of the race. Future iterations of the car may implement a more complex autonomy stack on a microprocessor that is capable of processing larger amounts of data more quickly. Ideally, a sparse map would be developed and motion planning would be done on that sparse map, while onboard sensors account for local changes. While this is currently not possible on the current hardware setup, it would be an exciting next step in the project's development.

APPENDIX

VII. C-TYPE REPORT: HUMAN-ROBOT INTERACTION

With autonomous vehicles such as self-driving cars being at the forefront of modern technology, the need for developing an understanding of human-in-the-loop robotic systems has been increasingly important. In many ways, formal verification of systems that allow for human intervention is more difficult than verification of fully autonomous systems. This is because humans are inherently unpredictable, and having a computer understand what a driver might do in a given situation is extremely difficult. In addition to this, many autonomous cars have multiple systems that need to be tracked so that their goals and tasks do not conflict. There are many systems available now which require operator-robot interaction, such as autonomous aerial vehicles, autonomous cars, and consumer products with autonomous systems. This paper will primarily focus on autonomous cars, where the driver is able to intervene at certain points while the car is driving. In these systems, human behavior has to be quantified so that the system is able to account for it.

The U.S. National Highway Traffic Safety Administration (NHTSA) defines four levels of automation, with level three out of four being "limited self driving automation." [4] Limited self driving automation is considered to be semi-autonomous capabilities in which the driver is still present and attentive at the wheel. More formally, the four requirements common to all level 3 systems are effective monitoring, minimal intervention, prescient switching, and conditional correctness. Effective monitoring and minimal intervention means that the system is able to identify whether human intervention is needed or not, and requests driver intervention when there is a high chance of system failure. Then, the system must allow for sufficient response times from the driver, and the system must be able to maintain operation until the driver intervenes. On the operator side, the human must maintain some sort of awareness and be able to take control of the vehicle within an acceptable time. This level of automation is what is available on the market for consumer use. From these requirements, it is clear that both the car's automation system and human behavior must be formally defined to some extent in order to ensure safety.

To be able to model human-robot interaction in this context, formal methods are needed. Formal methods define the controller as a stochastic system in which there are probabilistic transitions from discrete states in the system; in practice, this is defined as a Markov chain, which can sometimes be augmented with costs and rewards depending on the behavior being observed. Then, properties such as the probability of system failure, long-run behavior, and time-dependent system operation can be determined using linear temporal logic operators on the previously defined Markov chain. In the case of human-robot interaction, interactions are sometimes modeled as two-player stochastic games in which the human is playing against the autonomous system. In this case, two separate Markov decision processes are defined, in which strategies for both systems are selected

based on desired performance. In general, the model has the driver competing against the autonomy stack in order to identify the worst-case scenario for the car. In other words, the driver is employing strategies that work against the car's goal, such as maintaining a level of safety, at every choice, and the autonomy stack is employing strategies to reach that goal. Modeling the system in this way allows for different goals of the system, such as time to reach a goal or costs spent on tolls/gasoline, to be compared in order to reach an optimal strategy while ensuring safety. These strategies can be compared using Pareto curves, which generate a maximum bound for the strategies along linearly scaling values for each goal.

Using Markov processes for controller synthesis has shown to be necessary for automated driving. One often cited example of this is 2007 DARPA Urban Challenge [5]. For this challenge, teams were asked to build an autonomous vehicle that can complete complex maneuvers in an urban environment and drive in traffic for a $2m prize. These systems were not human-in-the-loop systems, meaning that they should have been more simple than even some autonomy stacks currently on the consumer market. One primary example from this challenge that highlights the need for formal verification is Caltech's submission "Alice," which was a modified Ford van with over 200 concurrent threads and 25 processors. While navigating the driving track, this submission had an unwanted interaction between the obstacle avoidance system and the path planner. The obstacle avoidance algorithm was declaring the car too close to a wall, while the path planner was repeatedly repathing the car too close to the wall. The two systems were unaware of each other's actions and therefore could not synchronize to form a strategy in the car was able to successfully navigate the turn. Using formal verification, the prioritization of each system was able to be defined and systematic strategies using linear temporal logic on the Markov chain were generated. Once formal verification was implemented, the car was able to resolve the issues it originally faced. Again, this example highlights how formal verification can prevent unwanted and unexpected interactions between onboard systems on an autonomous vehicle.

While there are many examples that have highlighted the need for formal verification, there are also limitations in defining human-robot interactions as stochastic processes. The most glaring limitation in formal verification is that human behavior can be extremely difficult to model. For example, many have attempted to model characteristics such as operator fatigue, awareness, and workload as probabilistic distributions. For many of these human conditions that can greatly affect, for example, the ability to drive, the best that can be done is to take averages of datasets that have captured human interaction with machines and come up with a best guess at how a human will respond in a given situation. Researchers that have explored this, such as Feng et. al [10], have suggested that the workaround for this can be to form specialized databases for humans interacting with a particular machine. This can be very impractical, as a large amount of data would have to be collected in order for this to be effective, and selecting parameters to observe in the first place already begins defining human behavior as rigid when it is often unpredictable. One may be able to say that the average human reaction time is a certain value, or that the average human gets fatigues after a certain number of hours but, in practice, this can vary greatly from person to person and the worst-case scenario must always be considered as it is a human safety issue. While strategies can be implemented so that the same mistakes are not made twice, the system cannot identify if it does not know something. Meaning, if something occurs that is unexpected to both the human and the car, the verification system is unable to make a best guess as to what to do. Because of the strict and bounded nature of stochastic modelling, alternatives need to be considered to design a truly failsafe system.

One alternative to formal verification is the use of machine learning to train the onboard autonomy stack. Data from a large number of vehicles can be analyzed to identify patterns in operator behavior, which then improves the autonomy stack. Autonomous cars rely on many sensors, and the data acquired from these sensors cannot always detect and identify what they need to. Using machine learning algorithms, they can be trained to better analyze data. Based on sensor data, the car needs to be able to make decisions on navigating the road and maintaining safety. Using data collected from real drivers, machine learning algorithms can be trained to make appropriate decisions both in regular navigation and in safety-critical situations. Doing so can also allow the autonomy stack to use real-time data in order to adjust its behavior. For example, it can use current traffic conditions in order to adapt its planned route and adjust to more safe behavior while it is in a congested area. This sort of adaptability is something that formal verification struggles with. Adding more information to a formal verification system can lead to state explosion and intractability, as the state size grows exponentially. Using machine learning onboard the autonomous vehicle makes it very adaptable to many different situations because the algorithm learns and adapts based on real datasets.

However, using machine learning instead of formal verification has several downsides as well. One major security issue is the ability to feed incorrect or malicious data to machine learning algorithms. When this happens, the algorithm can misinterpret safety-critical situations. One very simplified example of this would be if the autonomy stack is trained to stop at red lights, and an attack occurs that feeds the algorithm data to instead drive through red lights. These security concerns, called adversarial attacks, can cost human lives in the context of autonomous vehicles. In the same vein, machine learning algorithms are highly dependent on the data they are trained on. Even if the data is not malicious as mentioned above, if the data is not comprehensive or it is biased, it can lead to unexpected behavior from the vehicle. One example of this would be if data on driver behavior for use on an autonomous car is only collected in a certain region, but the car is deployed across the country. This may

cause issues, as driving patterns in the region where the data was collected may not be representative of patterns in the entire country. For safety-critical situations, which must work in emergencies, there needs to be an assurance that either the autonomy stack will work to keep the driver safe, or the stack can identify far enough in advance that the driver needs to intervene. In these situations, formal verification may be a better solution as it guarentees property satisfaction.

In practice, machine learning algorithms should be used in conjunction with formal verification methods in order to ensure safety and performance of the autonomy stack onboard self-driving vehicles. Formal verification can allow for fixed behavior on certain safety-critical tasks, such as stopping at stop signs and traffic lights. It is also necessary for synchronization between multiple systems onboard a vehicle to ensure unwanted interactions or conflicts between systems are avoided. Machine learning algorithms can help the car continue moving autonomously in unexpected situations that the formal verification model may not have accounted for, and excels in making the system more robust and able to adapt to many different situations. The ultimate goal of autonomous vehicles is ensuring safety while taking the driver off of the wheel as much as possible, but the safety of the driver is the most important consideration in the design of the car.

VIII. C-Type Report: Deep Learning

Prior to the invention of autonomous controllers for vehicles, cars have been operated entirely by humans. In order to be licensed to drive vehicles, humans have to undergo extensive periods of training to learn not only how to operate their vehicles, but also how the vehicles around them are likely to move. Additionally, humans are able to draw on their life's worth of experiences to easily detect and recognize other vehicles and obstacles. However, autonomous vehicles don't have the breadth of experience or the processing power of a human brain, so detecting and tracking obstacles on the road is a much larger challenge. With the introduction of deep learning, systems are able to be trained to more quickly and accurately detect obstacles on the road, without having to perform major adjustments to the system architecture. Additionally, well-trained deep learning algorithms have the ability to rapidly track, and even predict, the movement of other vehicles or objects in the area. Through the implementation of deep learning, autonomous vehicles can improve their performance and achieve better, safer results than previously achievable.

Deep learning algorithms in general can be used to take sensors data and make decisions based on features learned from sensor data. On an autonomous car, some of the sensors used include LIDAR, radar, and camera sensors. Lidar and radar sensors can give the system information about the distance and velocity of objects in the field of view. Cameras can give the system information about moving objects and what those objects are. The deep learning algorithm then takes this data and learns to predict, for example, the next location of an obstacle based on its past movement and velocity. It can also improve its image detection system based on the camera data, thus improving its ability to recognize camera information as the objects they are supposed to be. The applications for this with autonomous vehicles is clear, but some examples include predicting the movement of other cars, identifying and avoiding obstacles, recognizing traffic signs, and avoiding pedestrians.

The object detection system onboard an autonomous car is one of the most important systems for ensuring safety and smooth functionality. The detection algorithms essentially takes a data feed from a camera, and identifies and localizes objects in an image or video frame. Therefore, the algorithm has to be able to distinguish different distinct objects in the frame, and classify them according to what they are. For example, on a road, the camera onboard the car should be identifying other vehicles and classifying them as cars. An untrained algorithm might incorrectly identify car-shaped objects as cars, but a trained algorithm should be able to parse between, for example, a dumpster and a car. Numerous deep learning methods for object detection exist, each with their positives and negatives.

Some deep learning methods include region proposals (R-CNN, Fast R-CNN, Faster R-CNN) and you only look once (YOLO) algorithms [11]. In R-CNN methods, first, a general regions in which objects might exist are defined. These regions are extracted from the image and passed through a deep convolutional neural network in order to pull relevant features from the image region. Then, a bounding box around the object in the image, if there is one, is repeatedly refined to match the object. This method works extremely well but suffers from slowness because each region that is identified needs to be analyzed. Fast and Faster R-CNN both improve the speed of R-CNN by using various processes to improve the region detection and extraction. YOLO methods instead take the entire image and define probabilities on areas in the image that have a higher chance of having an object. This method is very fast, especially when compared to the base R-CNN. The use of deep learning for object detection means that objects can be identified in various conditions,. such as at night and in different weather patterns.

In the context of the competition for this paper, deep learning for object detection could be used to better identify obstacles. One example is, on the final racetrack, there were a few chairs and tables on the side of the track. With the depth following algorithm used for this project, it was possible that the car could incorrectly drive toward underneath a chair or table. With object detection, the car would be able to identify that these objects are obstacles that should not be on the course, and avoid them accordingly. Additionally, functionality for some of the challenges could have been expanded using object detection algorithms.

Deep learning can also be used for object tracking. Object tracking follows an object's location over time in a video sequence [12]. This means that it estimates and predicts the positions, velocities, and other information of moving objects in a video. The applications for autonomous cars are clear, as video can track and predict the position of

other cars, pedestrians, animals, and other moving objects around the car. The use of deep learning allows the objects to be tracked accurately even in difficult scenarios and video capturing conditions. Because the algorithm is predicting the movement of an object, it can continue to estimate where the object is even if it is no longer in the video frame. This is different from object detection, which can only identify current objects in the image frame.

Objects tracking algorithms have four basic processes. First, similar to object detection, objects of interest are identified and bounding boxes around those objects are drawn. Then, the object image data is represented in a way that it can be tracked in varying image conditions. For example, once a car is identified, it might drive through a dark area in which it still needs to be able to be identified. The motion of the object is then estimated in order to predict the next position of the object in time. This motion estimation is used to create a best estimate of the final position of the object.

Object tracking algorithms have both single and multi-object tracking, but multi-object tracking is primarily used for autonomous cars. In multi-object tracking, every object in the video is tracked and has its position estimated. In some algorithms, the objects are tracked relative to each other. Multi-object tracking also classifies all of the objects in the camera view. For example, a car's image detection might classify pedestrians, other cars, and obstacles.

Object tracking has many challenges which need to be resolved in order to have reliable performance on vehicles. First, distinguishing various objects from each other can be very difficult when the objects are close together or overlapping. For example, when pedestrians are crossing the road, they may be overlapping in the video captured of them. The algorithm needs to ensure that each person is identified correctly, and not as one blob that inaccurately represents the size or class of objects. This issue is called occlusion, and can be resolved through parameter tuning. Next, videos may have background clutter that interferes with the ability to distinguish objects. Training the algorithm on a good dataset can help the object tracking better differentiate objects of interest from the background. The deep learning algorithm must also be able to do object tracking and movement prediction extremely quickly, as autonomous cars need to be able to react quickly in situations where the driver is in danger.

Two specific deep learning methods for object tracking are Multi-Domain Net (MDNet) and GOTURN. MDNet uses large datasets to build a comprehensive understanding of objects, their visual variations, and spatial relationships. It takes annotated videos of different classes of objects. In this way, it pre-trains on the annotated videos and then does object tracking by creating layers for different domains of objects. GOTURN forms relationships between an object's appearance and motion to track objects both in the training set and not included in the training set. It trains offline on a video data set to handle different orientations of objects and various lighting and video conditions. GOTURN specifically uses a regression method to track objects based on a search region in the current data and target objects from prior video frames. These frames are then compared against each other to identify and track the target.

Object tracking could be used on the competition car to better perform in a number of challenges, and to drive in more complicated environments. For example, the rolling ball challenge could be completed more robustly, accounting for varying ball speeds and spinning of the ball, using an object tracking algorithm. The car may also be able to drive in complicated environments, such as hallways with moving people or multiple moving obstacles. The current car is programmed to only drive toward the deepest part of the image captured. This method would crumble in complex environments with multiple moving objects. While implementing object tracking onboard the car would be very complicated, for the development of a significantly more robust and smart car, this would be an interesting next step.

In conclusion, the implementation of deep learning algorithms can be used to increase the performance of vehicles using the gathered sensor input. Utilizing sensor fusion and deep learning can allow for systems to be adaptable. Deep learning can allow for object avoidance of irregular tasking while more traditional methods will fail to meet changing goals. Along with this, deep learning allows for vehicles to be taught on a wide range of scenarios that can be used to help with decision-making on new events encountered. The utilization of deep learning could have improved the performance of our vehicle by creating less structured approaches to turning. Specifically, turns taken by our vehicle had inconsistent conditions causing varying results. Deep learning would allow for the vehicle to be better at generalizing results and prevent the breakdown of logic in new situations. Overall, deep learning can drastically improve the performance of vehicles and increase the consistency of autonomous control results.

ACKNOWLEDGMENT

We would like to acknowledge Professor Hayes and Professor Heckman, along with the course teaching assistants for CSCI5302, for their help on this project as well as their guidance on Piazza. We would also like to acknowledge those who answered questions on Piazza and helped troubleshoot issues on the system.

REFERENCES

[1] "F1TENTH." F1TENTH - Build Documentation, 9 Apr. 2020, f1tenth.org/.

[2] IntelRealSense. "IntelRealSense/Realsense-Ros: Intel(r) RealSense(Tm) Ros Wrapper for Depth Camera." GitHub, 1 May 2023, github.com/IntelRealSense/realsense-ros.

[3] Duzhen, Feng. VSLAM and Navigation System of Unmanned Ground Vehicle, Dec. 2018, https://www.utupub.fi/handle/10024/147661

[4] "Automated Vehicles for Safety." NHTSA, https://www.nhtsa.gov/technology-innovation/automated-vehicles-safety.

[5] Buehler, Martin, et al., editors. "The DARPA Urban Challenge." Springer Tracts in Advanced Robotics, 26 Nov. 2009, https://doi.org/10.1007/978-3-642-03991-1.

[6] IntelRealSense. (n.d.). Slam with d435i. GitHub. https://github.com/IntelRealSense/realsense-ros/wiki/SLAM-with-D435i

[7] RTAB-Map. (n.d.). Real-Time Appearance-Based Mapping. http://introlab.github.io/rtabmap/
[8] M. Labbé and F. Michaud, "RTAB-Map as an Open-Source Lidar and Visual SLAM Library for Large-Scale and Long-Term Online Operation," in Journal of Field Robotics, vol. 36, no. 2, pp. 416–446, 2019.
[9] Mur-Artal, R., & Tardós, J. D. (2017). Orb-slam2: An open-source slam system for monocular, stereo, and rgb-d cameras. IEEE transactions on robotics, 33(5), 1255-1262.
[10] Feng, Lu, et al. "Controller Synthesis for Autonomous Systems Interacting with Human Operators." Proceedings of the ACM/IEEE Sixth International Conference on Cyber-Physical Systems, 2015, https://doi.org/10.1145/2735960.2735973.
[11] Vadapalli, Pavan. "Ultimate Guide to Object Detection Using Deep Learning [2023]." Ultimate Guide to Object Detection Using Deep Learning [2023], 21 Nov. 2022, www.upgrad.com/blog/ultimate-guide-to-object-detection-using-deep-learning/.
[12] "The Complete Guide to Object Tracking [+V7 Tutorial]." V7, www.v7labs.com/blog/object-tracking-guide. Accessed 9 May 2023.