Raspberry Pi Controlled Autonomous Vehicle

Ked Bastakoti, Mitchell Terry, and Tim Dorny
https://eng.utah.edu/~bastakot/seniorproject

Abstract— **Autonomous vehicles are a rapidly developing technology, and for this project we focused on designing a small land drone capable of navigating a marked course while avoiding obstacles. The drone used an ultrasonic sensor to detect obstacles and image processing to process and interpret the desired path. A Raspberry Pi was the heart of the project, handling signals, processing images, providing PWM signals to the motor driver bridge, and running the integrated program and its main logical functions.**

I. Introduction

A. Motivation

Autonomous vehicles are an up-and-coming technology that could have a major impact on the society. The technology behind these self-driving vehicles is still in the early phase of development. This technology utilizes many concepts covered in our courses as computer engineering students. In an autonomous drone, hardware components must work together with software to create a system that is capable of navigating the complex driving environment. It is very exciting to work on something similar to this new technology that is expected to be one of the next major achievements.

Previous researchers have studied image processing and developed many algorithms for lane detection and locating the vehicle in the lane, as far back as 1990 [1]. There are different approaches and algorithms depending upon hardware limitations (micro-controller vs server processing, CPU, GPU, memory, etc.). Cho et. al [2] developed a lane following robot using a single camera as input. It was able to recognize the left and right sides of a lane and was able to maintain the center line of a track. Fig. 1 and Fig. 2 demonstrate the original and the detected lane using Hough Transformation. Hough Transformation was used to detect a lane, and a PID controller was used to control the direction of a mobile robot.

Lane detection in the real world is difficult due to a number of factors. These factors include problems such as shadows, saturation, lane wear, and poor visibility conditions. [3]. These factors require advanced image processing algorithms to be used by real-world examples to correctly detect lane lines and other road markings.

Another key component of autonomous vehicles is obstacle detection. There are two approaches to obstacle detection that are commonly utilized. Active methods use sensors to actively detect obstacles as they appear. Passive methods try to detect obstacles through passive measurements of the vehicle's surroundings, such as camera images [4].

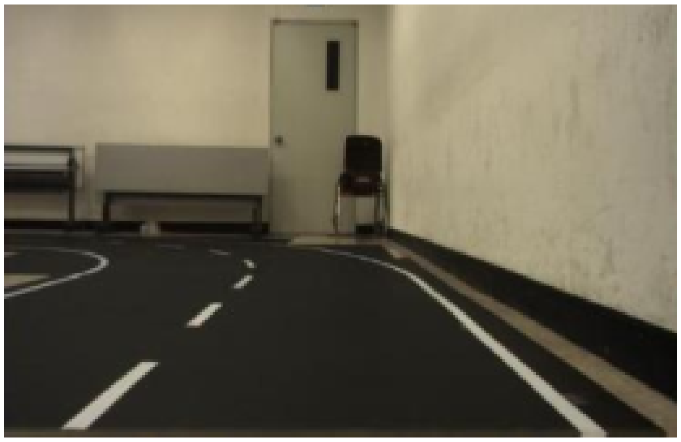
Fig. 1. Original lane (adapted from [2]).

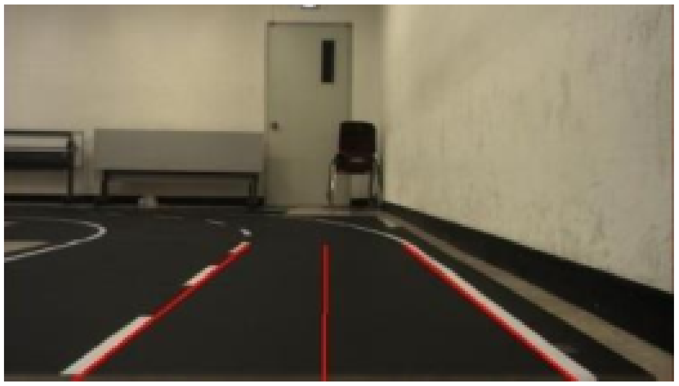
Fig. 2. Detected lane using Hough Transformation (adapted from [2]).

For our project, we implemented our own version of this autonomous technology on a much smaller scale. We utilized sensors to allow our drone to read and react to the environment. A camera was also used along with image processing to allow our drone to follow a marked path. A Raspberry Pi 3 Model B, the latest version as of early 2017, was the main computational engine of this system. Fig. 3 shows a high-level overview of the Raspberry Pi board. Detecting distances of objects and obstacles in the surrounding environment was also explored using the inverse perspective mapping algorithm in OpenCV [5]. However, it was solely based upon the Ultrasound Sensor in the final implementation.

Our finished product was a small land drone that was

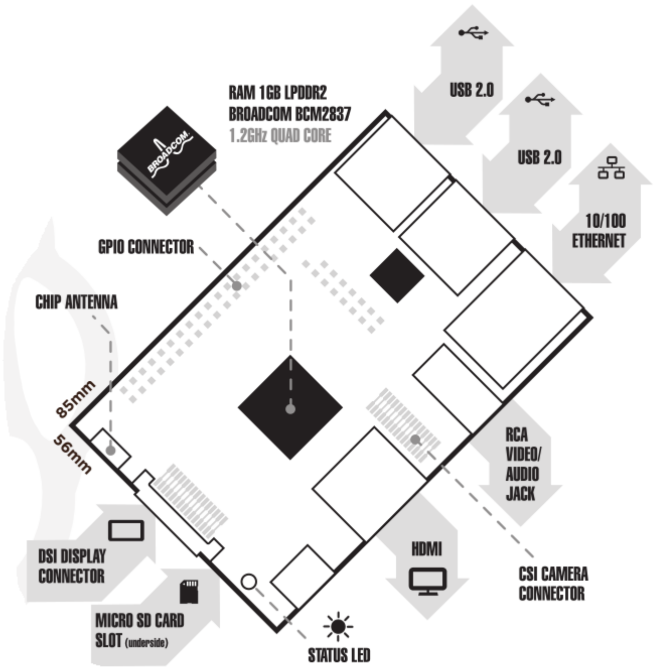

Fig. 3. The Raspberry Pi 3 Model B has 1 GB DDR2 RAM and 1.2 GHz quad-core processor. It also supports image processing libraries such as OpenCV (adapted from [6]).

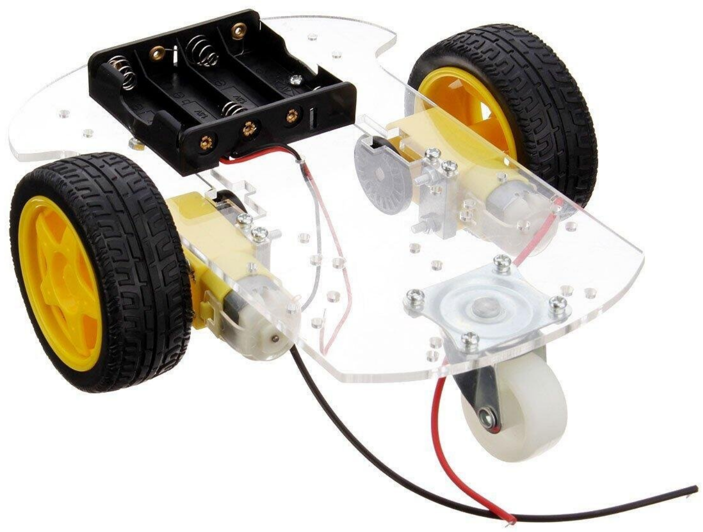

Fig. 4. Chassis, motors, and battery pack of the Gowoops kit.

capable of navigating a marked course while reacting and adapting to obstacles it encounters. The physical sensors detected obstacles, while the camera read and followed lane markings. Software within the Raspberry Pi controlled the system.

We demonstrated our project by setting up a circular marked course using tape. Our vehicle navigated the course without receiving extra human input. The vehicle reacted to changes such as the addition or removal of obstacles while continuing to navigate the course.

II. Hardware

For hardware components we wanted to find components that were cheap but could still handle the required tasks for this project. We purchased a chassis kit that consisted of two motors, a plastic chassis plate, and a battery holder. The Gowoops car chassis kit that was purchased from Amazon is shown in Fig. 4. The kit included the chassis, two motors, mounting hardware for the motors and a battery pack for the motors that could also be mounted. The chassis is a simple piece of plastic with a number of mounting options in a generally rectangular shape. Since our project involved a number of pieces that needed to be mounted on the chassis, we elected to use Velcro to mount the other pieces of hardware onto the chassis. This allowed us to easily add, remove, or adjust components while also effectively securing them in place.

The motors included in the kit we purchased are simple, cheap DC motors. Providing a voltage to one of their two terminals cause them to spin either forward or backward.

Since the raspberry PI board is unable to provide enough current to power the motors, we needed a separate motor bridge board. We used a L298N H-Bridge motor drive controller, shown in Fig. 5, which received signals from the Raspberry PI board and its output drove two motors.

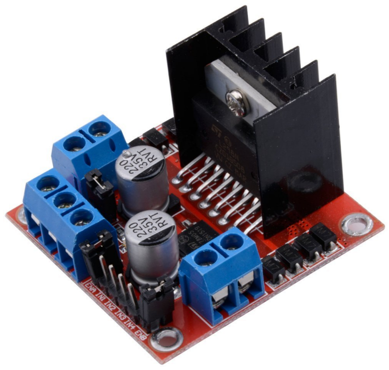

Fig. 5. L289N motor bridge

The chip uses a dual-channel H-bridge driver. A general pin-out diagram of this controller can be seen in Fig. 6. It takes a power supply of up to 12 volts. The board has six logic pins, three for each motor. Each motor has an enable pin along with two direction pins. The enable pin controls the operation of the motor while the two direction pins control the direction the motor will spin. The chip supports two DC

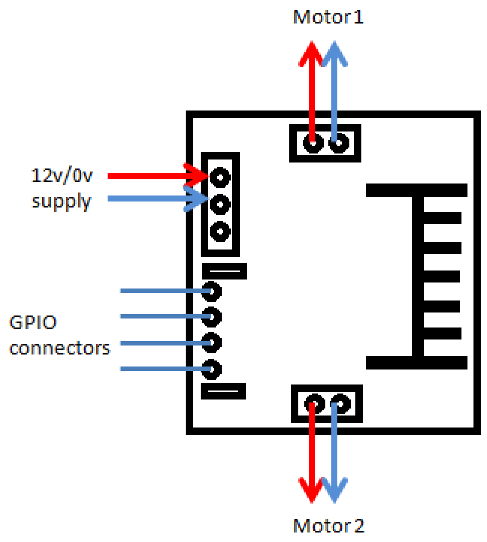

Fig. 6. General overview of the L298N motor drive controller pinout.

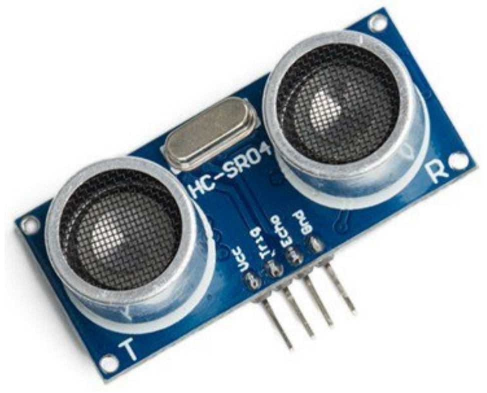

Fig. 7. Ultrasonic sensor with 4 I/O pins.

motors, providing two lines for each motor's terminals. One of the primary reasons we chose this motor controller was that it can support PWM (pulse width modulation) signals. This would allow us to control the speed at which each motor turns, enabling the ability to make turns of varying intensities.

Initially we had purchased a L9110S DC stepper motor driver board to use for this project, but had difficulties getting the PWM functionality to work properly. After going through two of these boards, we decided to switch to the L298N driver which ended up working extremely well.

An ultrasonic sensor, shown in Fig. 7, was used at the front of the chassis to detect obstacles while driving forward. The sensor sends sound waves at a specific frequency and waits for the echo to bounce back. The approximate distance to obstacles is calculated by multiplying speed of sound in air with the total time taken to receive the echo and dividing the product by 2.

Raspberry PI has a built-in interface for camera modules. However, due to the placement of other hardware components and other specific requirements, we had to use a separate USB camera mounted on top of the chassis. It captures images continuously as the drone drives around. The OpenCV library's built-in functions are used to further process the images.

We used the Raspberry Pi 3 Model B because it has the ability to process images without the help of external computing devices such as a laptop or server. With 1 GB of DDR2 RAM and a 1.2 GHz quad-core processor, it was able to meet the basic requirements of our project, and it handled the integrated program successfully. The GPIO pins, shown in Fig. 8, of the Pi were used to interface with other hardware devices.

The Raspberry Pi is powered through a portable battery pack using a micro-USB cable. For the motors, a separate power source consisting of 4 to 8 AA batteries is used. By using two different power sources, it allowed us to run programs on the PI and test the motors for a much longer time than what would have been possible by just a single source.

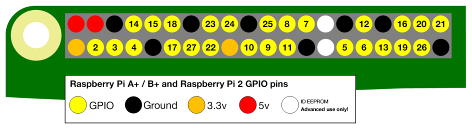

Fig. 8. Raspberry Pi GPIO pin diagram.

III. Software

This project consisted of three major components, motor control, obstacle detection and image processing. Each of these components were split into their own individual classes to follow an object oriented programming model. All of the code for this project was written in Python 2.

A. Motor, Powertrain Model

The software uses two classes to model control of the drone's motors. The classes are named Motor and Powertrain.

The Motor class is assigned on construction GPIO pins that map to the motor controller board pins forward, backward, and enable. The class implements functions that set signals for those GPIO pins to drive a motor to do an action. The forward function sets the forward pin high and the backward pin low, while the backward function does the inverse. Both of these driving functions also take a duty cyle and configure the enable pin to a PWM signal at the given duty cycle. The brake function sets all pins high to lock the motor, and the off function sets all pins low to stop powering the motor.

The Powertrain class models the use of two motors, left and right, to act as the drone's drive system. The class is assigned on construction GPIO pins that map to the motor controller board pins forward, backward, and enable for the left and right motors. The class implements functions that manipulate the motors to do movement and steering actions. The forward and reverse functions call the motor's forward and backward functions, respectively. Forward and reverse take a duty cycle and pass it to the motor functions. The turn function takes in duty cycles and forward-moving booleans for left and right motors as direct control of the motors. The functions turn left, turn right, turn intensity, and pivot build on the turn function, modifying inputs to fit their respective descriptions. The wrapper turn functions take a maximum duty cycle and forward-moving boolean, and take an intensity parameter that is used to calculate the duty cycle for the slower moving motor. The behavior of intensity is that at zero, the motors both recieve the same duty cycle, and as the intensity increases the slower motor recieves a lower duty cycle, until the slower motor reaches a duty cycle of zero. The turn intensity function can take a signed intensity to detect a turn direction, left for negative, right for positive. The pivot function takes a boolean for turning the drone clockwise. The stop function calls the motor's stop function.

B. *Image Processing*

The image processing implementation is contained within a single class, ImageProcessor. The constructor for this class takes two arguments: a camera port and lane type value. The camera port is used to determine which camera the program will be pulling images from. Since only a single web cam is connected to the Raspberry Pi, a value of zero is always used. The lane type argument tells the program if it will be looking for light lanes on a dark background with a value of one, or looking for dark lanes on a light background with a value of zero.

The goal of this class was to take what the camera sees, extract the location of any lane lines in the image, and tell the vehicle how it should adjust its trajectory. To accomplish this, binary thresholding was used. To perform the image processing within the class, the python library OpenCV was used. First, a color image is taken using the attached web cam. This image is then converted to gray scale using OpenCV's cv2.cvtColor function. Finally, we used OpenCV's cv2.threshold function to perform binary thresholding on the gray scale image. Depending on the lane type that was specified, either the standard binary thresholding or inverted binary thresholding is performed.

A gray scale image is essentially a two-dimensional array of pixel values ranging from zero to 255, with zero representing pure black, and 255 representing pure white. Binary thresholding works by taking a value from zero to 255 to use as the threshold. Every pixel in the gray scale image that has a value greater than this threshold is set to a value of 255, while every pixel with a value less than the threshold is set to a value of zero. The resulting image has only two possible color values. Ideally, we want the resulting image to have the higher value where lane lines are detected and the lower value everywhere else. To maximize performance, the environment that the vehicle will be driving in must be considered. Bright or dark areas that are not supposed to be detected as lanes could cause problems. Motion blur and lighting can cause noise within the image that might make it difficult to accurately determine lane locations. These are factors that must be considered when designing the algorithm. Fig. 9 provides an example of how an image is changed when processed using binary thresholding.

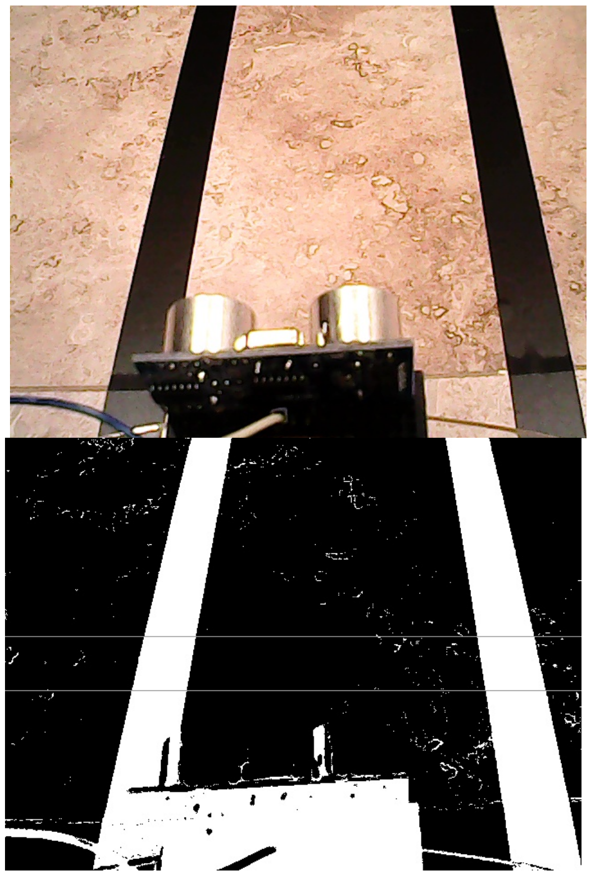
Fig. 9. Example of how an image is altered through the process of binary thresholding. The horizontal gray lines in the center of the thresholded image represent the rows that are checked for lanes.

Once we have a properly thresholded image, we must determine exactly where the detected lanes are and how to adjust based on what we see. To determine where the lanes are, we can simply iterate over the image, which is a two-dimensional array containing values of zero or 255, and see where the values of 255 are found. In the interest of efficiency, we selected two row values that would be iterated over to find the lanes. An example of these rows can be seen in Fig. 9. We check two rows to be able to tell where the lanes are as well as which direction they are slanting. This information can help when determining how to adjust the vehicle's path. To reduce the negative effect of noise in the image, a minimum lane value is specified. This value specifies how many light pixels must be consecutively found to count as the detection of a lane. Once a range of pixels is determined to be a lane line, the midpoint is calculated and stored in a tuple. The program creates one tuple for each row that is scanned. Ideally, when the vehicle is heading straight down the lanes, there are two values in each tuple at the resolution of this stage.

Finally, the locations of the detected lanes are used to determine if and how the vehicle needs to adjust. First, the number of lanes that were detected in each row is checked.

For example, if there are two points in each tuple, we know that the vehicle is between the two lanes and small adjustments could be made. If one lane is detected in each row, we know that the vehicle is veering off course and requires more significant adjustments. Using the direction the single lane is slanting, we could determine which direction the vehicle needs to turn to get back on course. If more than two lanes are detected in either row, we know there is some sort of interference present in the image, and if no lanes are detected we assume that the vehicle is off course.

Initially, this class returned a value between -1 and 1 that represented the direction and intensity of the determined adjustment. So a value of 0.1 would tell the vehicle to make a slight turn right while a value of -0.9 would tell the vehicle to make a sharp turn left. Due to issues encountered as the project neared conclusion, the return values of the class were changed. The final algorithm utilized the same cases as before, such as how many lanes were detected in each row, but the return values and what they represent were changed. The case of one lane being detected in each row was also adjusted. Rather than using the slant of the lane to determine which direction to turn, the location of the lane in relation to the midpoint of the image was used. If the lane is seen in the right-hand side of the image, the vehicle is told to turn right, and vice versa. Instead of returning a value telling the vehicle how to adjust, the class now returns a value simply telling it which direction to turn, with some additional signals. A value of 1 tells the vehicle to turn right, -1 means turn left, and zero means continue in the same direction. A value of 2 tells the vehicle to switch directions, and a value of 3 signals that the system has completely lost track of the lanes.

IV. Debugging Tools

Since the Pi is running a basic command-line version of the Linux operating system, we had to develop some tools in order to communicate with the Pi and debug our code during the development phase.

A. OnPiBoot Module

The first of such tools is the OnPiBoot Module which performs two important functions automatically when the PI is turned on. First, it emails the IP address of the PI to pre-configured emails which can then be used to ssh into the machine. Second, it starts a local web server on the PI which allowed us to view the processes images and their raw counterparts.

B. Server Module

The Server Module is an extremely helpful tool for debugging and checking the status of PI. As shown in Fig. 10, it takes user input via a client program, forwards that to the main control logic, and updates the client program with the updated status information. It also allows us to control the drone with the arrow keys and 'z' and 'x' for acceleration and deceleration. It shows the complete status of the motors and the state of different parameters such as current duty cycles of each motor, turn direction, distance to obstacles, and minimum and maximum allowed PWM duty cycle values.

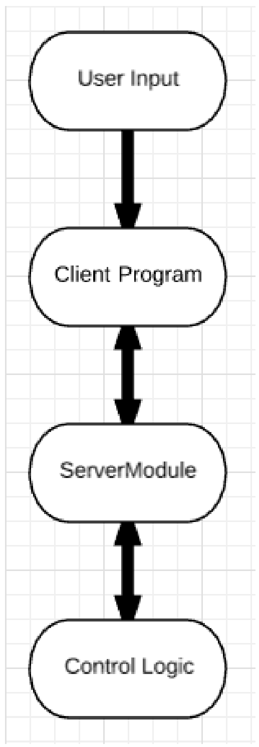

Fig. 10. Diagram showing interaction of ServerModule with other components.

To manually control the drone, first the server is started on the PI. Then, a client can connect to the server from any browser and start interacting with the drone. On the client side, a simple HTML file and a JavaScript file with AJAX logic is served. When the user preses any key, the key code is sent to the server. The server, on receiving this key code, compares it with pre-defined constants at the ServerConstants module and executes the appropriate handler such as accelerating, decelerating, or changing direction.

Furthermore, the server also sends the current status of the drone to the client, and the client displays the the status as a debugging aid. To save the bandwidth on the PI and minimize the workload, the status is only updated after a key press.

C. Motors Calibration Logic

The logic to calibrate motors works in connection with the Server Module. First, the server needs to be running to start the calibration. Then, using the navigation keys on the client browser, the drone can be driven around for a test. While driving straight, if the drone drifts more towards the right, this means the right motor is running slower than left motor and the right motor speed can be increased by pressing the '$>$' key. Similarly, if it drifts more towards the left, the left motor can be calibrated by pressing the '$<$' key until the drone starts going straight.

D. Obstacle Detection Logic

Multiple ideas were explored to detect obstacles and to avoid collision. The ultrasonic sensor located at the front of the drone is responsible for detecting large obstacles. Alternative means of detecting obstacles such as using machine learning to detect stop signs were also tested, but did not prove to be useful due to the inability of the PI to handle a large amount of data and processing requirements.

1) ObstacleDetector Module: The ObstacleDetector module is used to instantiate an instance of the ultrasonic sensor in software. It takes three inputs: trigger pin, echo pin, and distance threshold. The distance threshold parameter is helpful to set in order to run a specific handler when the distance to obstacles becomes less than the threshold. The trigger pin and echo pin directly correspond to the hardware trigger and echo pin of the sensor. To measure the distance to an obstacle, first the trigger pin is set to high. This starts sending the sound waves. After 10 microseconds the pin is set to low and a while loop is used to wait to read the input value in trigger pin. The trigger pin automatically goes high when the echo is received by the sensor. The time difference between the two events multiplied by the speed of sound in air, 34,300 cm/s, gives the two way distance. Hence, to get the one way distance the product is divided by 2. Fig. 11 demonstrates the operation of an ultrasonic sensor device.

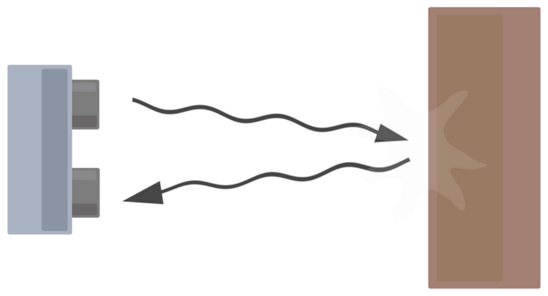

Fig. 11. Diagram illustrating basic operation of the ultrasonic sensor.

To facilitate easier integration, ObstacleDetector runs on a separate thread once started and continuously measures distance and updates the status. It also allows registration of event handlers to be executed when an obstacle is detected or is within a specified range.

2) Stop Sign Detector Module: An attempt was made to detect printed stop signs that would be placed on the side of the lane markings using the Haar Cascades algorithm for rapid object detection [7]. Haar Cascades internally uses machine learning to recognize the features and patterns of objects. First, a stop sign was printed and several images of it were taken to create the positive images. Then, several background images of different objects were taken which acted as negative images. The images were then used to train using the Haar Cascade and generate an XML file. However, the generated XML file when used to test in the real environment was not able to detect the stop signs. Furthermore, the Pi was not able to properly handle the extra computational load incurred by this algorithm, so the integration with the rest of the program logic could have significantly slowed down the process. An alternative idea was to use a real desktop or laptop computer to perform the heavy computing and communicate over the network with the Pi, but as it was a little beyond the scope of this project we didn't focus on that part.

V. Implementation

In many cases, integrating numerous parts into a cohesive whole is a difficult and taxing process. Fortunately for us, our project's individual components came together without a lot of difficulty.

The first piece that was integrated was the motors. Using the L298N chip, integrating the motors with the Raspberry Pi was as simple as hooking up six jumper wires and controlling their signals through software. Getting this up and running was very quick and painless. As time went on, we experienced a number of issues relating to the motors and their performance, but each of those issues was caused by hardware and beyond our control. A power source was also required for the motors. We started by using an array of 4 AA batteries (6V) and eventually added a second array for a total of 8 AA batteries (12V).

The second part of the system that was added was the ultrasonic sensor. This sensor's purpose was to tell the system when it should yield to an obstacle that is in the vehicle's path. Once again, integration of this piece of hardware with the pi was very straightforward. It takes a 3 volt input which the pi can supply, trigger and echo pins that are connected using the the pi's GPIO pins, and a ground pin. A voltage divider was added to limit the incoming 5 volt supply to better conform to the sensor's requirements. Once the sensor was wired up, a simple block of code was all it took to start getting live updates of the sensor's detected distance.

The final hardware component that needed to be added was the camera. It was a simple process of installing the necessary drivers on the pi and plugging the webcam into one of the USB slots. Pulling images from the camera into our program was also quite straightforward. A folded piece of cardboard was initially used to mount the camera on the chassis. While visually unimpressive, it performed its job remarkably well and provided additional space underneath for the batteries and wires.

The main concern when integrating our project together was space. As the kit we purchased is relatively small, there was a limited amount of space to place components. As previously mentioned, Velcro was used as our primary means of mounting, which allowed for easy adjustments to our layout. We needed to fit all of the hardware onto the chassis while also accounting for the position and angle of the camera. We eventually settled on the layout seen in Fig. 12 with the camera mounted on the front half of the portable USB power supply and the pi on the back half. The

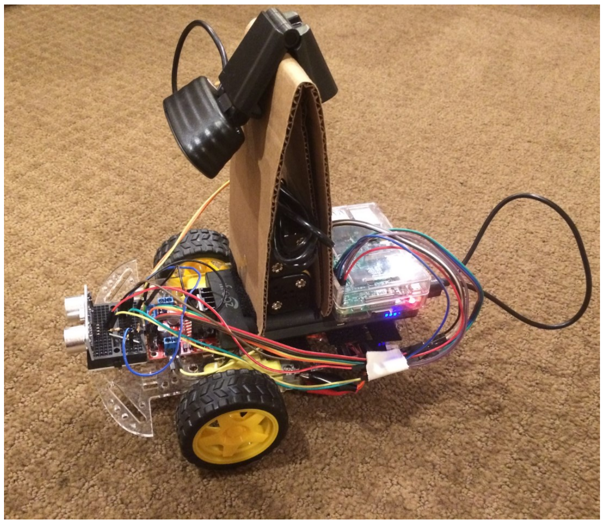

Fig. 12. Final build of the project as demoed on December 8th.

ultrasonic sensor of course is mounted on the front end of the vehicle with the L298N chip directly behind it. While the final product certainly lacks refinement, it performs as desired and clearly displays how each component of the system interacts with the rest.

VI. Difficulties

Our choice of using cheaper parts came at a cost. Specifically, the hardware used was generally fragile and in many cases did not function completely properly.

A. Problems with the Motor Bridge

The first Motor Bridge we purchased did not properly handle the PWM signals generated by the Raspberry Pi. It took a long time to debug the code and hardware and to come to this conclusion. After purchasing another motor bridge, it worked flawlessly.

B. Problems with the Motors

Without extra load, initially, the motors performed fine when individually tested under different power and PWM settings. However, soon after placing both motors on the chassis, and testing both of them together, they started demonstrating problems. Even if the same power and PWM signals were provided to both motors they would randomly show different behaviors. To overcome this problem, we tested the calibration logic as mentioned in the section above. However, even after calibration, the drone was not fully controllable. Either one wheel would randomly stop spinning or spinning with less speed than requested. A large amount of time was invested in figuring out the remedy of this problem, but the problem was not deterministic and calibration failed to address the issue. Finally we decided to control or drive a single motor at a time to minimize the effect of this issue. This ultimately allowed us to create a working demo.

One difference between the initial plan and the final implementation is that we implemented calibration functionality in our project whereas initially we planned to use PID controller logic. The purpose of either of these is to correct the behavior of the motors. However, since the motors we used were extremely primitive and lacked a feedback mechanism, it was better to implement the calibration logic which allowed manual control of each motor.

VII. Conclusion

The world is moving towards autonomous vehicles. Even though our project is not necessarily scalable to a full size autonomous vehicle, it exposed us to a few of the issues that we might encounter in a real life situation. The project was interesting in itself, and it required us to incorporate knowledge from both hardware and software engineering fields. From this project we learned to integrate hardware and software together to create a working product. Even though we got caught up in number of unexpected situations and difficulties with hardware and computing limitations, the project was successful in meeting the proposed requirements. In addition, the knowledge and experience we gained by working on this project will serve as a foundation for further academic and professional advancement.

References

[1] S. K. Kenue, “Lanelok: Detection of lane boundaries and vehicle tracking using image-processing techniques-part i: Hough-transform, region-tracing and correlation algorithms,” in *1989 Symposium on Visual Communications, Image Processing, and Intelligent Robotics Systems*. International Society for Optics and Photonics, 1990, pp. 221–233.

[2] Y. Cho, S. Kim, and S. Park, “A lane following mobile robot navigation system using mono camera,” *Journal of Physics: Conference Series*, vol. 806, no. 1, p. 012003, 2017. [Online]. Available: http://stacks.iop.org/1742-6596/806/i=1/a=012003

[3] X. Du and K. K. Tan, “Comprehensive and Practical Vision System for Self-Driving Vehicle Lane-Level Localization,” *IEEE Transactions on Image Processing*, vol. 25, no. 5, pp. 2075–2088, 2016.

[4] C. Hne, T. Sattler, and M. Pollefeys, “Obstacle Detection for Self-driving Cars using Only Monocular Cameras and Wheel Odometry,” *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*, pp. 33–55, 2015.

[5] S. Tuohy, D. O’Cualain, E. Jones, and M. Glavin, “Distance determination for an automobile environment using inverse perspective mapping in opencv,” in *IET Irish Signals and Systems Conference (ISSC 2010)*, June 2010, pp. 100–105.

[6] “Raspberry datasheet,” 2017. [Online]. Available: http://docs-europe.electrocomponents.com/webdocs/14ba/0900766b814ba5fd.pdf

[7] R. Lienhart and J. Maydt, “An extended set of haar-like features for rapid object detection,” in *Proceedings. International Conference on Image Processing*, vol. 1, 2002, pp. I–900–I–903 vol.1.