[MQP-AW1-GCIM]

Vision System Prototype for Goat cart:

An Autonomous Golf Cart

Major Qualifying Project Report

completed in partial fulfillment of the

Bachelor of Science degree

at *Worcester Polytechnic Institute*, Worcester, MA

Submitted by:

Muhaimin Islam

Advised by:

Alexander Wyglinski Ph.D

Abstract

This MQP attempts to prototype a depth camera-based vision system for the Goat Cart the Autonomous Golf Cart, with Microsoft Kinect. This vision system works by polling depth data using Microsoft Kinect and giving inputs to other modules *i.e.* brakes, steering module so that the vehicle can avoid obstacles. Based on a simple occupancy grid algorithm the vision system was able to give correct inputs to other modules to avoid obstacles in a lab test where movement decision results were printed on to the console. This MQP also explores the logistics of a LIDAR based vision for the Goat Cart.

Acknowledgments:

I would like to thank Professor Wyglinski for this opportunity to work on the Golf Cart Project, this project introduced me to some important concepts that will help me in my Engineering career. Without help and guidance from the following people this project would not be possible.

Professor Alexander Wyglinski

Leah Morales (ECE Shop)

Mathew Addileita

Jade A Pierce,

Alexander Westfield Briskman

Camila Di Fino Napolitan

Table of Contents

List of Figures:

List of Tables:

List of Tutorials:

Chapter 1: Introduction

1.1 Motivation:

In 2010, there were 32,999 people killed, 3.9 million were injured, and 24 million vehicles were damaged in motor vehicle crashes in the United States [1]. The economic costs of these crashes totaled $242 billion. Included in these losses are lost productivity, medical costs, legal and court costs, emergency service costs (EMS), insurance administration costs, congestion costs, property damage, and workplace losses. [1]. It is estimated that ninety percent of motor vehicle crashes are caused by human error [2]. Figure 1 shows the summary of accidents and human error, roadway issues and vehicle issues.

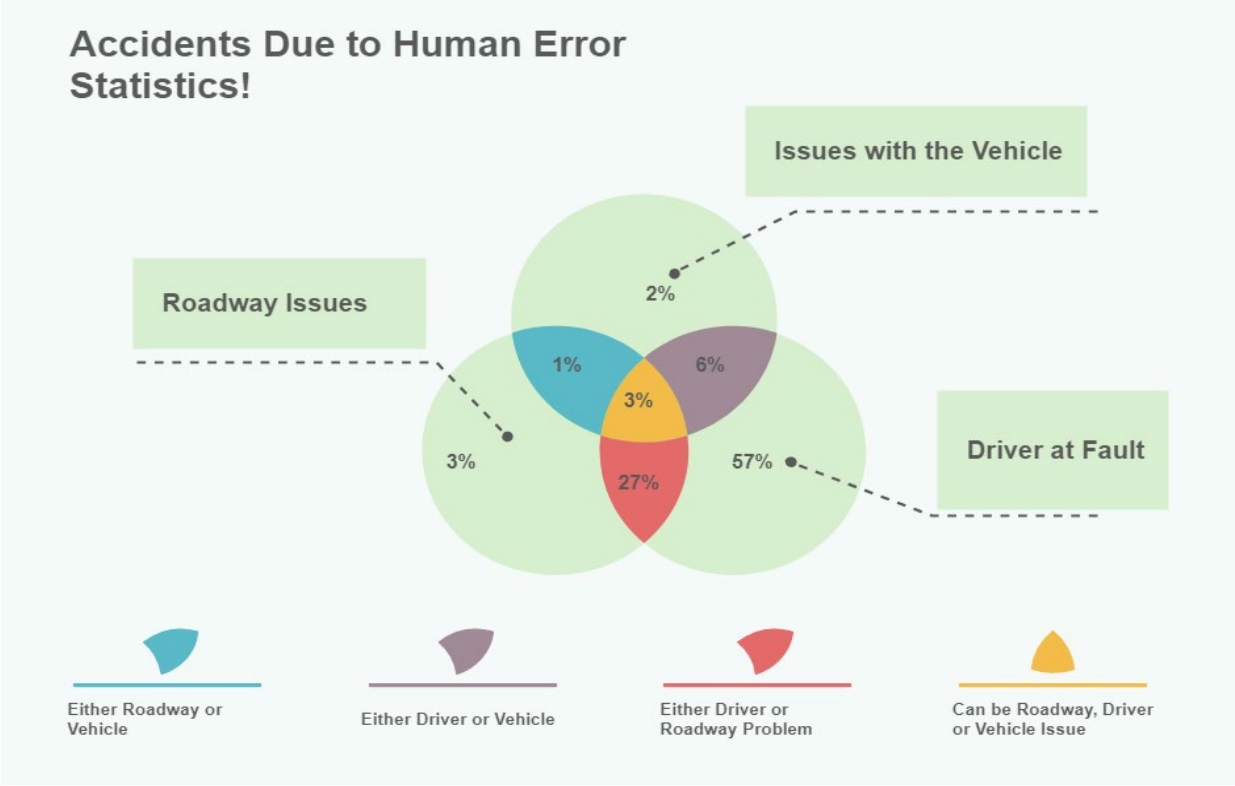


Figure 1: Accidents due to Human Errors statistics adapted from [3]

Not only accidents can be reduced by implementing some sort of automation in driving but also fuel economy can be improved as well. Semiautonomous vehicles can reduce the mean fuel

consumption can reduce mean fuel consumption up to 5.3% depending on driving situation and speed. In the United States, a large number of conventional vehicle manufacturers are investing heavily in autonomous vehicle technologies [4]. According to the European Environmental agency's data shown in Figure 2, road transport is one of the main polluting factors in cities with emissions from transportation having risen 20% in the last 2 decades. Electrical vehicles can dramatically reduce local greenhouse emissions significantly [5].

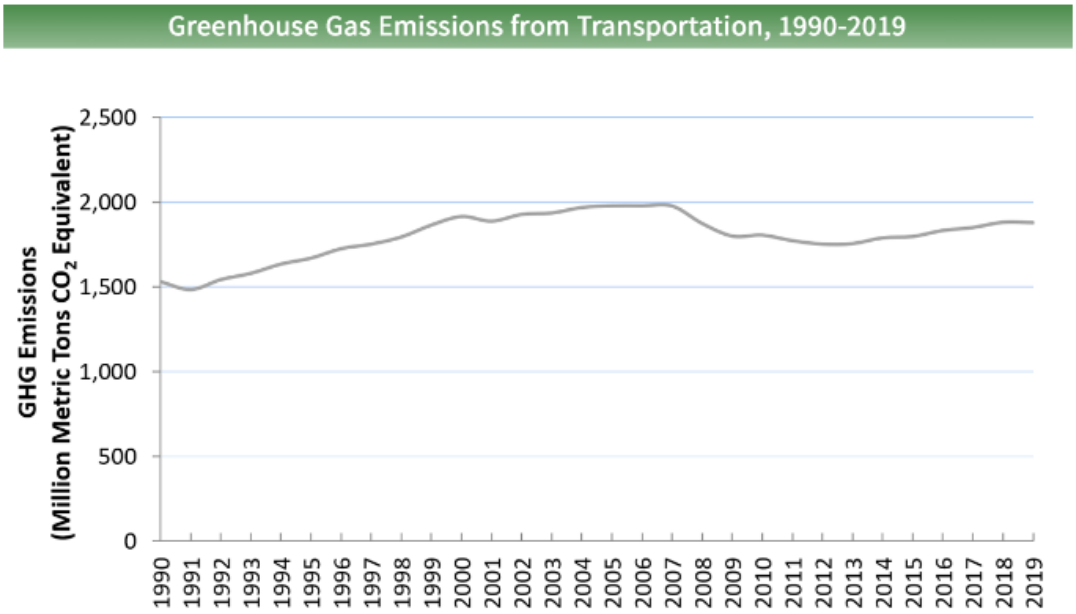


Figure 2: Greenhouse Gas Emissions by transportation relative to 1990s level. Data Source [5]

Advances in autonomous vehicles technology and mass deployment of this technology depends on the political and economic will towards a sustainable environment, as well as the integration of those systems with information and communications technologies that are rapidly being introduced into modern passenger vehicles. Most of the advances are led by vehicle manufacturers, in addition to the defense and academic research communities [6]. Google claimed that in 2015 their cars experienced 272 failures and would have crashed more if their human drivers had not intervened, driving between 30,000 and 40,000 miles per month [6]. Additional failures are documented by

all the manufacturers working in this sector, including a casualty during a Uber test in Arizona in 2018 [7]. Public transportation can significantly benefit from the introduction of intelligent vehicles since they have the potential to improve safety in urban areas as well decrease the cost of transportation, decrease congestion, and improve service for the user overall.

1.2 State of the Art Automated Vehicles:

To understand the current state of autonomous vehicles it is important to understand a formal taxonomy of different autonomous systems [8] that have been laid out by the NSHTA in 2017. They released a guideline of automation ranging from no automation to full automation as well as where the current industry stands based on this taxonomy. It was referred to as the 6 Levels of Automation defined which is by SAE International as the following [6]:

1. No Automation (Level 0) – The human driver must complete all driving tasks even with warnings from vehicles.
2. Driver Assistance (Level 1) – The automated system shares steering and acceleration/deceleration responsibility with the human driver under limited driving conditions (*e.g.*, high speed cruising), and the driver handles the remaining driving tasks (*e.g.*, lane change).
3. Partial Automation (Level 2) – The automated system fully controls the steering and acceleration/deceleration of vehicles under limited driving conditions, and the human driver performs remaining driving tasks.
4. Conditional Automation (Level 3) – The automated system handles all driving tasks under limited driving conditions, and expects that the human driver will respond to requests to intervene (*i.e.*, resume driving).

5. High Automation (Level 4) – The automated system handles all driving tasks under limited driving conditions even if the human driver does not respond to requests to intervene.
6. Full Automation (Level 5) – The automated system takes full control of all driving tasks under all driving conditions that can be managed by a human driver.

1.3 Previous MQPs:

The primary of objective of the work in the 2017-2018 Goat Cart's MQP was to come up with a solid electromechanical base of the platform so vision can be integrated. The goal of this project was to integrate a vision system with this prototype platform. Unfortunately, it was not possible to integrate the contributions of this MQP with the results of the other MQP team due to timing of the two projects. Therefore, the outcomes of this work were never tested by the Goat Cart. Consequently, this MQP focused on developing a functional vision system that could detect obstacles and make decisions, which could eventually be integrated into such a platform.

The Golf Cart MQP team of 2017-18 had made progress in formulating a modular design, but little to no integration. The following modular blocks were functional in the Golf Cart the way I received it: A CAN bus system, a steering system, a power supply system. Other systems such as the throttle and brakes were not functional at the time. As a result, it was difficult to perform comprehensive integration with this platform.

1.4 MQP Goals:

Motivated by the fact that automating safety features of a vehicle can reduce accidents significantly, a basic vision system for the Goat Cart was prototyped such that it can be improved in the future with better camera, Lidar and Vision Systems. This was a standalone system that should be modular to be integrated in an Automated Vehicle with little work.

These are this project's contribution to the Goat Cart System:

1) Integration of Microsoft Xbox Kinect to Linux Server
2) Use of Kinect successfully as a Lidar
3) Produced Example Python Scripts for Obstacle Avoidance with Kinect
4) Successful Communication to CAN bus from Server

1.5 Report Organization and Summary:

The report is structured as follows: Chapter 2 describes a background tutorial about the Goat Cart and Vision System for the Golf Cart in general. A Kinect based vision system and a proposed approach to use Lidar as tool for vision system of the Goat Cart is described in Chapter 3. Chapter 4 describes conclusions and future works of the project.

Chapter 2: Background of the Golf Cart:

The Goat Cart is supposed to have the following sub systems that will interact with each other with a CAN bus.

1. Steering System
2. Brake System
3. CAN (Control Area Network) Bus System
4. Vision System
5. Server Decision System
6. Sensors

The ultimate goal of these subsystems is to provide an automation framework for the Golf Cart so the cart can achieve different levels of automation with the control of the server. A high level system diagram can be seen from Figure 3.

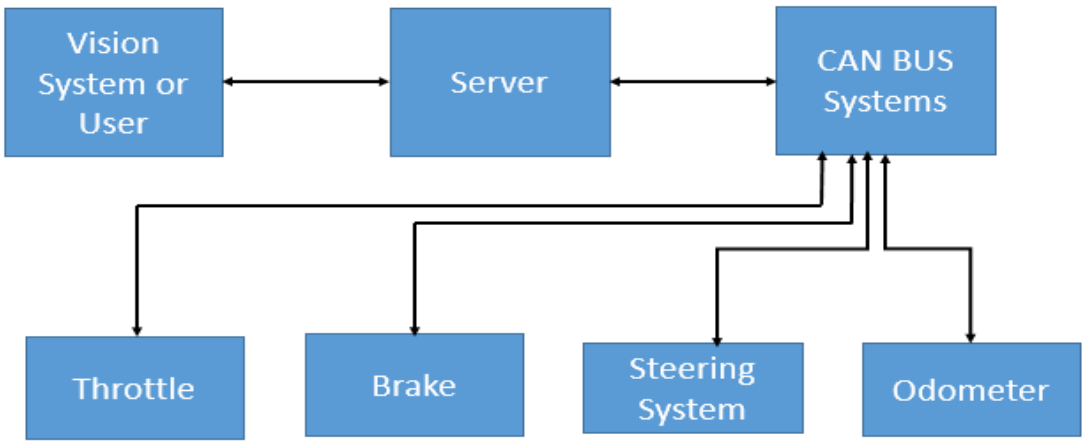


Figure 3: High Level System Diagram of Goat Cart. Vision system or User input interfaces with the Server and the server then communicates with the CAN bus system to relay the message to different subsystem.

All the subsystems shown in the diagram are supposed to be connected by a CAN bus. One of the CAN bus boards is the master board that is controlled by the server and the other boards are the

slaves. The master board communicates with the server in order to give the decision engine useful information about the state of the vehicle. Based on the information that the decision engine get different decisions are made.

2.1 Steering System:

The steering system of the Goat Cart is steered by using a CIM motor. The motor is controlled by a Sabretooth 2x60 motor controller. As shown in Figure 4, these are following parts of the Steering System.

1) Sabretooth 2x60 Motor Controller [9]
2) Teensy 3.5 Controller [10]
3) 10 k Ohm Resistor
4) 2 Limit Switches
5) Encoder Chip [11] for the Steering

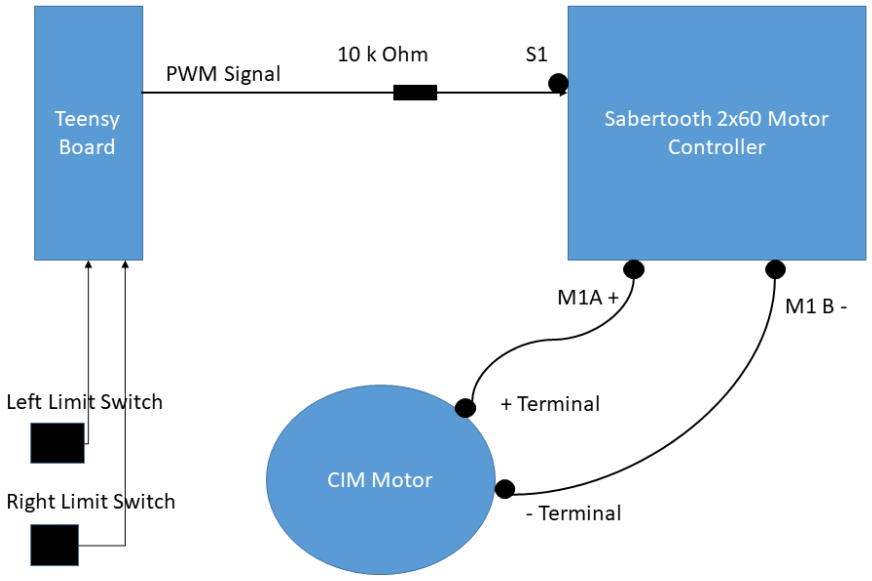


Figure 4: Steering System Diagram

In order to control the CIM motor, a Teensy 3.5 Board is used. The firmware code for the steering system is written in Arduino environment. In this instance, the teensy board talks to the motor controller. Motor controller is treated as a servo object in the Arduino environment. To control the motor controller servo.write is used.

In order to make the steering go left, a value of 80 is written in the servo while to go right a value of 110 is written in the servo. To calibrate the steering system right and left limit switches along with an encoder is used to have a feedback system to understand where the steering is at the moment. Left and right limit switches are used to get feedback whether the steering has reached the left or right most corner. The encoder is used to figure out where the steering is at any given moment.

However, it is necessary to calibrate the encoder to calculate correct encoder value based on the position of the steering. The following process is used to calibrate the steering encoder; go right all the way and get the encoder value, go left come all the way get encoder value, come half way of the left and right most position take the encoder value when it is in middle.

The steering system is controlled by software only and not integrated with the CAN bus yet. Further calibration of the system needs to be done with another encoder to ensure the control software has some sort of feedback.

Several suggestions on integrating the steering wheel include the following:

1. Add manual override mechanism for a user to control the steering wheel.
2. Adding another encoder to measure the turns the new wheel takes.
3. Calibrate the turns the new steering is taking and integrate the wheel with the current Steering CAN bus.

2.2 Brake System:

Braking is needed to slow the vehicle down and stop to avoid collisions. The braking in the Goat Cart is done with a motor that is controlled by a Sabretooth [9] motor controller. The system has the following parts as shown in Figure 5.

1. Brake Motor
2. Sabretooth 2x60 Motor Controller
3. Limit Switch (for User Induced Braking)

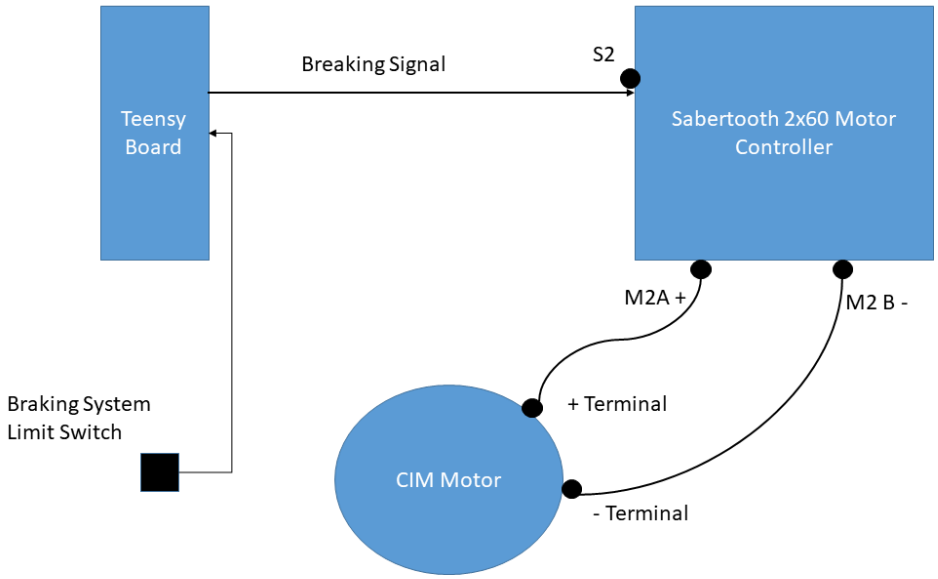


Figure 5: Brake System Diagram

The brake system consists of 2 types of braking mechanisms that are available for the Goat Cart: Software induced braking and user induced braking. Software braking occurs when the vision system detects an obstacle in front of the vehicle and sends a command to brake in the CAN bus. Once the firmware receives the signal, the braking motor gets started and uses the brake for the vehicle. The driver / user can also brake traditionally with a manual brake. Once the user presses

the brake it gets pressed with the limit switch once the limit switch gets pressed it signals the brake motor start and brake.

2.3 Vision System:

In order to implement a vision system for the Goat Cart, Microsoft Xbox Kinect is used. With the Kinect, depth image is taken and a very simple occupancy grid is created. With the occupancy gird, some basic autonomous decision making is done. Some examples of the autonomous decision making include the following:

1) Brake/ Slow Down Stop,
2) Turn Left,
3) Turn Right,
4) Go Forward, and
5) Go Around the Obstacle.

It is to be noted that these decisions are not full proofed yet it needs more testing to drive the cart autonomously.

2.4 CAN (Control Area Network) Bus System:

CAN (Control Area Network) is a real-time communication protocol that provides fast, simple, efficient, and robust communication among different subsystems [13]. CAN bus protocol is mandated by the Environmental Protection Agency [12], therefore it is widely used in the automobile industry. The 2017-18 Goat Cart MQP team implemented custom CAN bus interface boards. After implementing the boards, the team daisy chained the boards to communicate between different boards. The intended architecture of the CAN bus system in the Goat Cart is shown in Figure 3. One of the CAN bus boards is the master board that is controlled by the server and the other boards are the slaves. The master board communicates with the server in order to give the

decision engine useful information about the state of the vehicle. Based on the information that the decision engine get different decisions are made.

The 2017-18 Goat Cart team had a few daisy chained CAN interface boards that this project integrated with the server. Connection of the server with the CAN bus system was proved and demonstrated with a Python script in March 2018. Figure 6 shows an example script was used to demonstrate communication with the CAN bus. The script was a simulation to connect to different components in the CAN bus system that was used to light up leds; it demonstrated successful CAN bus was from the server. However, this achievement was modular no system eve integration was achieved.

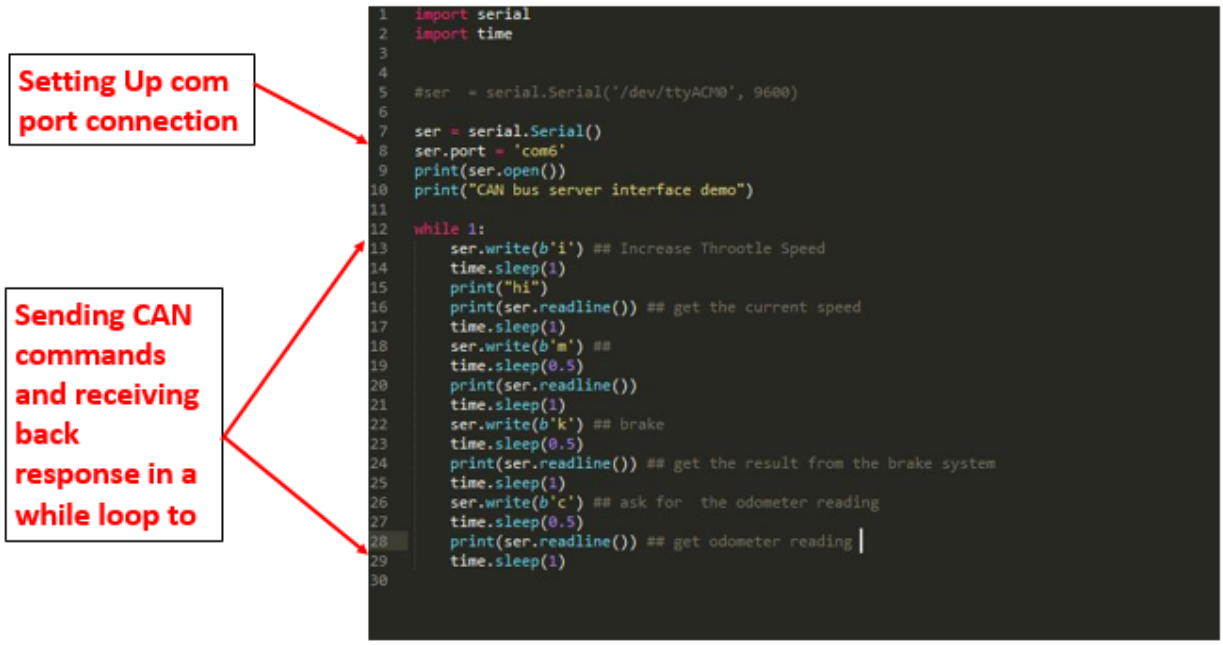


Figure 6:CAN bus Control Example. This example Script that was used to demonstrate successful communication with the CAN bus. Script sets up serial connection and sends and receives CAN messages inside a while loop as annotated in the figure.

2.6 Power Supply for the Peripherals:

The 2017-18 Goat Cart MQP Team designed the power supply for the Goat Cart [13]. A brief description of their design is shown below. The power supply harness is divided into two parts.

1) Power Throttle Supply 48V battery system
2) 12 V system for Peripherals

The 48V power system supplies power to the throttle and braking system. The 12 V power system supplies power to the peripheral systems. A diagram of the harness for the 12V system is shown in Figure 8.

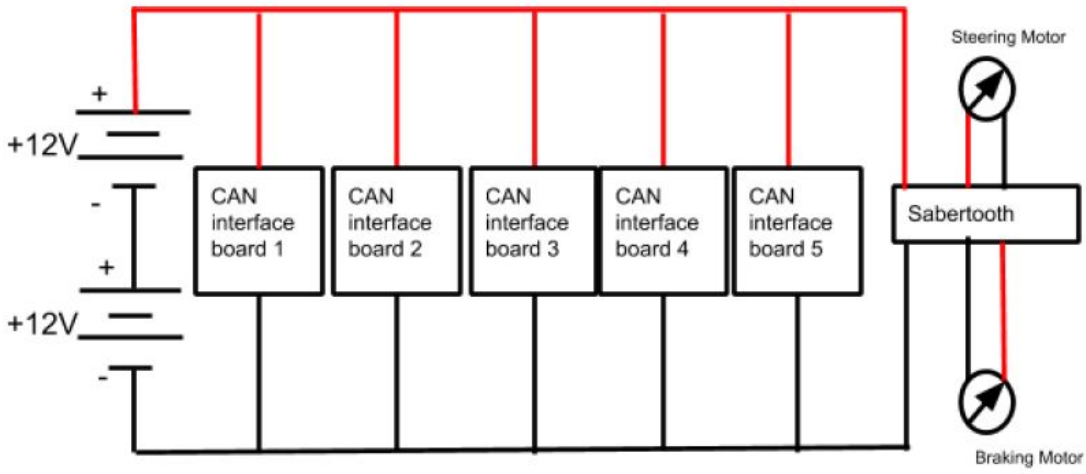


Figure 7 : Power Supply Harness for Peripherals [14]

The charging of the batteries is handled by a float battery charger, the Automatic 1.5-Amp Battery Charger/Maintainer. This new charger can be continually plugged into and outlet and left in the power system. The only maintenance is needed at this point is checking the water levels of all the batteries every few months.

2.7: Goat Cart System Diagram and with Kinect Vision System:

The goal of the project is to automate the functionalities of the Goat Cart platform. As it is seen in the system diagram in Figure 3 the objective of the platform is to automate functionalities while driving. A Vision system and user Input system is designed to communicate with the server.

From user input or the inputs from the vision system, the server will be able to send and receive different commands and information to different parts of the Goat Cart system through a CAN bus. The automation system has a server which takes necessary decisions based on the different sensor inputs and sends decisions to the CAN bus. In this project, as the diagram shows there are three subsystems that are designed to work together to provide automation to the cart. The subsystems are vision system (Kinect), server (Decision Making Module), and CAN Bus. The CAN bus connects to the sub modules necessary that are to drive the cart around. In this implementation, the sub modules consist of the following:

1) Throttle (Increases speed),
2) Brakes,
3) Odometer, and
4) Steering System

The server is able to send and receive back commands to the CAN bus. The commands that are sent are either user input based or obstacle avoidance algorithm of the Goat Cart.

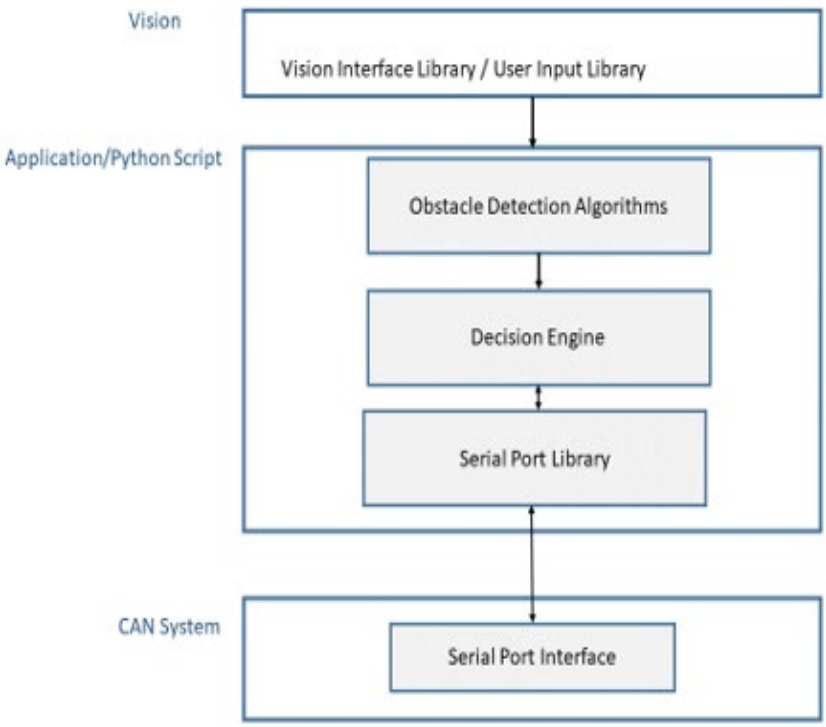


Figure 8: Intended Software Architecture of the Goat Cart platform is seen in the diagram here. From the Vision System or Input the application Layer/Obstacle Avoidance Library id intended to communicate with the CAN bus via serial port interface to send and receive back CAN messages.

Referring to Figure 8, the different components of the software architecture are as follows:

1) Vision Interface Library / User Input Library
2) Application / Decision Making Library
3) Serial Library

2.8: Chapter Summary

The vision interface library interfaces with Microsoft Xbox Kinect with the server. Depth data and vision data are brought back to the server by vision interface library. The decision making library is supposed to take inputs from different sub systems, sensors and depth data from Kinect, and make decisions so the cart can drive safely. A python serial library is used to enable communication with the Server and the CAN Bus network; UART protocol is used to send commands and receive commands from the CAN bus.

Chapter 3: Prototype of Vision System:

This chapter describes Kinect-based approach to develop an obstacle avoidance algorithm for the Goat Cart. This chapter also discusses the logistics necessary in order to prototype a Lidar based vision system. The fundamental concepts used in this project were: Kinect, Occupancy Grid Algorithm, Depth Image, and Lidar.

3.1 What is Kinect:

The Kinect [15] is an intelligent hardware platform manufactured by Microsoft. It has a RGB (Red, Green, Blue) camera, an IR (Infrared Ray) laser projector, an IR CMOS sensor, a servo to adjust the tilt of the device and a microphone array. The RGB camera is basically a webcam, however IR emitter and depth sensor helps Kinect to view the world in 3D and see obstacles. Budget concerns, excellent community support and high depth image sample rate were the factors that led to the using of the Kinect as a vision system for the Goat Cart.

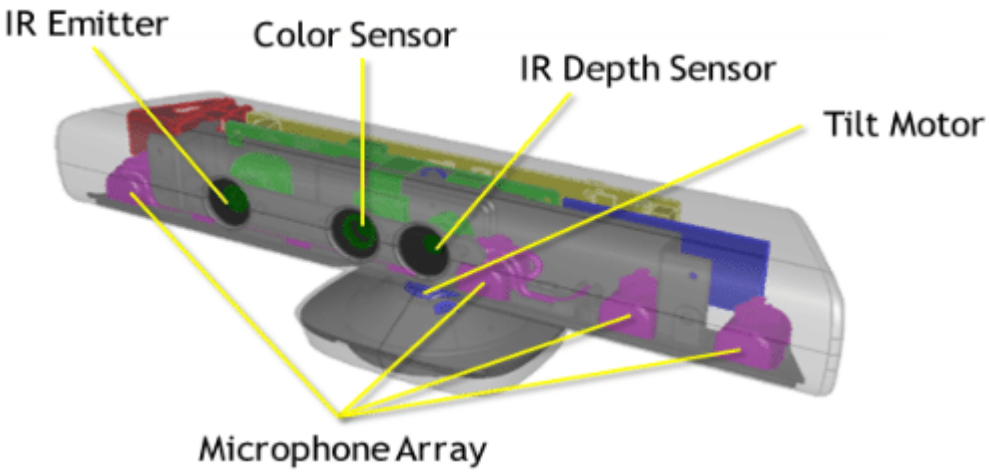


Figure 9: Different components of Kinect [16]

3.2 Vision Implementation and Integration:

The proof of concept vision system was implemented in Python with OpenCV and Open Kinect. OpenCV (Open Source Computer Vision Library) is an open-source computer vision and machine learning software library. Open Kinect is the open-source software used to interface with Kinect and the Linux Server.

In order to implement a vision system with a Kinect depth sensor first, an interface with Linux and Kinect must be configured. The Kinect depth sensor is basically an IR sensor by which depth information can be collected at 30 frames/sec. The procedure to interface the Kinect with a Linux server described in Appendix 1 assumes that the user has Ubuntu or Debian-based Linux distribution with open cv installed in the system. Open a terminal and run the following command:

```
sudo apt –get update & upgrade
```

The reason for issuing this command is that it will obtain the latest Linux package information and upgrade the packages based on the information. In fact, Open CV and other Kinect packages might not work properly if we do not have the correct package information. At this point it is necessary to install all the necessary dependencies for the Microsoft Kinect driver for Linux. To install the dependencies, we need to issue the following command:

```
sudo apt-get install git-core cmake freeglut3-dev pkg-config build-essential libxmu-dev libxi-dev libusb-1.0-0-dev
```

Table 1: Libraries installed for Kinect Interfacing with Linux

Library Installed	Reason for Installing
Libusb-01	Libusb-01 is a generic USB device library. It is installed so that, Kinect can send and receive data/commands to and from the server.
Build Essentials	GNU compiler collection, make, g++, dpk-gdev etc that are used to install build system for Kinect drivers.
Cmake	Build system tool for installing Microsoft Kinect driver for Linux.
git core	Version control and getting source code for libfreenect.
Freeglut-dev	Graphics libraries to view depth data from Kinect.

A description of the libraries installed and the rationale behind installing the libraries are provided in Table 1. After installing the libraries described in Table 1 the Linux system has all the tools to install Libfreenect driver that will interface with Kinect and Linux server. First, it is needed to clone the libfreenect library in our local repository. Using the command:

```
git clone git://github.com/OpenKinect/libfreenect.git
```

After cloning the repository, the next step is to navigate to the repository with the terminal and configure the build system for installing libfreenect. This can be achieved by typing the command:

```
cd libfreenect
```

After navigating to the repository, it is imperative to configure the build system for installing libfreenect Kinect Linux driver. Thus, the following commands need to be entered in the terminal:

```
mkdir build
cd build
cmake -L ..
make
sudo make install
sudo ldconfig /usr/local/lib64/
```

The commands will configure cmake build system and install libfreenect driver in in the location /usr/local/lib64/. At this point, it is important to add Kinect as a non –privileged user such that Python scripts can access the Kinect properly. Therefore, the following commands needs to be entered in the terminal:

```
sudo adduser $USER video
sudo adduser $USER plugdev
```

At this point, a file needs to be generated with the rules for the Linux device manager. In order to accomplish this the following command needs to be typed:

```
sudo nano /etc/udev/rules.d/51-kinect.rules
```

Following this step, the following code found in Appendix 2 will need to be entered.

Following this step, it is necessary to log out and log back in, for the installation take effect. To test the installation, the command "`freenect-glview`" needs to be entered in the terminal. This would cause a window to pop up showing depth and RGB images. More details are provided in Appendix 1.

3.3 Kinect Hardware and Software Interfacing challenges:

While connecting the Kinect to the Linux server, there were several software and hardware issues that needed to be addressed to achieve successful connection. The Kinect power cord was never connected to internal power supply of the Goat Cart; rather the power supply was connected to the wall power supply. As a result, some consideration needed to be taken to how Kinect would be powered in the event of an operational mobile vehicle.

It was observed that when the Kinect connected to the Linux server correctly, the Kinect connection light was typically green/red which indicated Kinect was interfacing. Most of the time when Kinect failed to see any data, the cases were traced back to connection light not being on. Another challenge that was observed while connecting Kinect USB to Linux Server was while going through a hub instead of connecting Kinect USB directly to Linux server our recommendation is that it is best not to use a hub to connect the Kinect with a Linux server as the Kinect USB device driver was not designed to support connections going through a USB hub. There were several software issues while connecting Kinect with the Linux server. For example, after installing the Kinect library, it is important to add the necessary links and cache for the installed Kinect driver using this command:

```
sudo ldconfig /usr/local/lib64/
```

This creates the necessary links and cache to the most recent shared libraries found in /usr/local/lib64/. The Kinect device driver rules need to be saved as a Root user since doing so without administrative privileges will result in the device driver settings not being saved effectively. To use Kinect as a non-privileged user, it is necessary to add $user to the plugdev group, as this group allows members to mount and unmount removable devices through pmount.

3.5 Calculating distance from depth map:

After connecting the Kinect properly with the Linux server, the next step is to process the depth information from Kinect depth sensor. Depth information needs to be translated to distance values from the camera. The formula to translate the Kinect depth data to distance was found from several Open Kinect blogs referenced in Appendix 3. The formula that was ultimately used to calculate the distance from camera (using depth data) is:

$$distance = 0.1236 * tan\left(\frac{depthInformaion}{2842.5} + 1.1863\right) \text{ ……..…….. Equation 1}$$

The Kinect views the world in a 640x480 grid. The grids are divided in to 3 simple grids. The average of the depth values of the grids are taken. Using the depth values, calculations of the distances of the obstacles from the camera are determined. Figure 10 (more details in Section 3.7) shows the depth map of the experiment in Room AK318 at 125 Salisbury St Worcester, MA, 01609, USA.

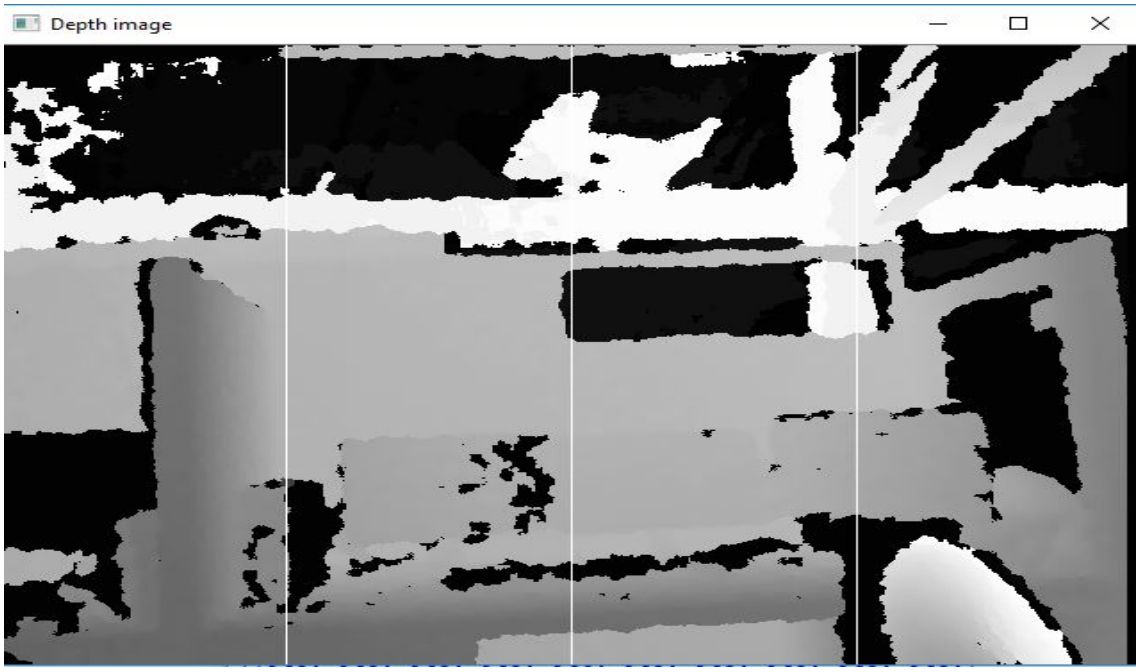


Figure 10: Depth Map with occupancy grid in WPI Wireless Innovation Lab Atwater Kent Lab Room on February 2, 2018

Observing the depth map shown in Figure 10, in order to avoid obstacles a basic occupancy grid was developed. The following algorithm detailed in Table 2 was designed to demonstrate simple obstruction avoidance by Microsoft Kinect.

Table 2: Vision system Scenarios and Decisions

Scenario	Decisions
Depth Value of whole Frame > 5 meters	Move Forward
Depth Value of whole Frame < 1.5	Brake
Middle Grid is less than 2.5 meters and greater than 3. 4 and right grid greater than 3 meters	Turn Right
Middle Grid is less than 2.5 meters and greater than 3. 4 and left grid greater than 3 meters	Turn Left

Since the infrastructure was not available to automate the cart, the movement decisions results from the vision system was printed to the console. An example of how this algorithm was implemented in can be found in Appendix 4. The Vision system's integration with the CAN Bus was implemented modularly, where a Python serial library was used to communicate with the CAN Bus from the server. The data was sent from the server to the CAN Bus and the signal went to different buses as it can be seen in the diagram of Figure 3.

3.6 CAN bus and Server interfacing procedure:

The Server and Master CAN Bus are connected at a baud rate of 9600 with a `/dev/ttyACM0` interface with Teensy 3.5 Interface with Master CAN Bus and Server were made with pyserial library. To connect to the master CAN Bus implemented using Teensy 3.5 that is connected to the USB port of the server, it is necessary to instantiate a Python serial object with the correct port number and serial baud rate. For example, in the Linux server the serial object was instantiated by the following:

```
ser  = serial.Serial('/dev/ttyACM0', 9600)
```

Here `/dev/ttyACM0` is the serial port that we are connecting to and 9600 is the value of baud rate. In the Arduino IDE, the serial port can be found by going to Tools -> Port ->. However, if another Arduino is connected to the server, it will be difficult to know which is the correct port to connect to. In this situation, the following process was defined:

1. Disconnect the master CAN Bus USB connection from Server
2. Open up a terminal and type the command: `ls /dev >file1`
3. Connect the master CAN Bus USB to the server
4. In terminal type the command `ls /dev >file2`
5. Then type the command: `diff file1 file2`
6. The resulting diff or output will look something like:

```
82a83
>serial
155a157
>ttyACMO
```

In this case, ttyACM0 is the correct port that we need to connect to. This approach is capable of finding the correct port because /dev is the location of special device files in Linux. Once a USB

device is connected, it shows up in /dev. Therefore, forming an Arduino connection and taking the diff of the two files gives us the correct port. From the Teensy 3.5 documentation, it was indicated that the serial baud rate of Teensy 3.5 was 9600. If the following command "`stty /dev/Port`" is issued in terminal, the baud rate can be extracted as well. In this case, the port will be the one that was disconnected. An example output of the command is the following:

```
"speed 9600 baud; line = 0; -brkint -imaxbel"
```

where the speed value represents the baud rate. To demonstrate communication with the CAN Bus, the script shown in Figure 11 was used to attempt communication. After some testing, the script shown in Figure 12 was chosen:

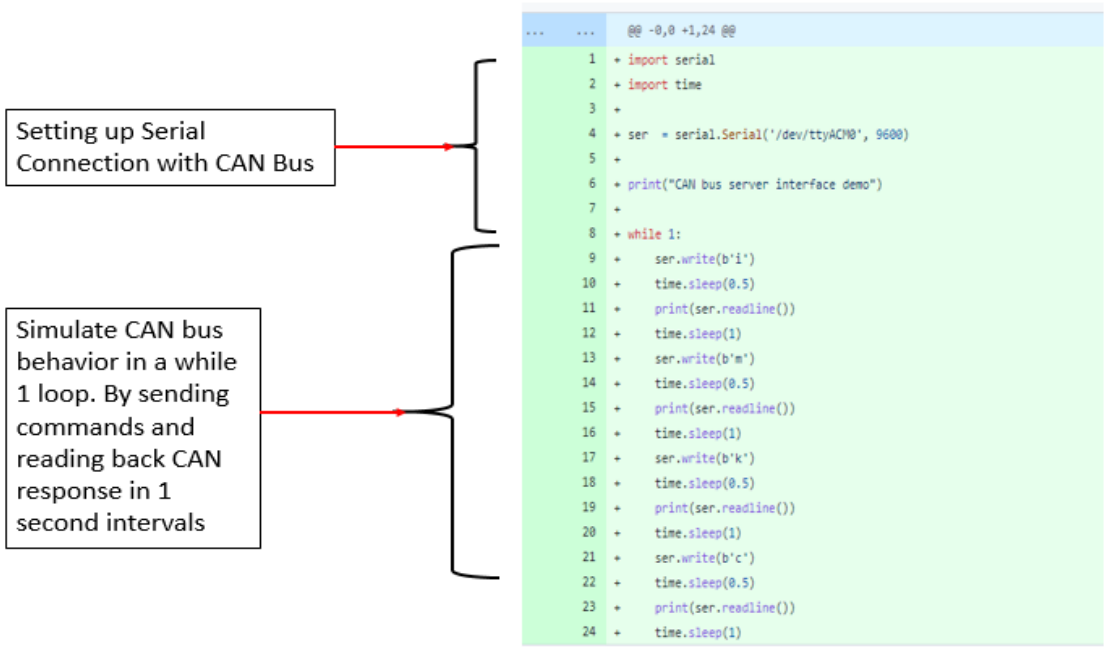


Figure 11: CAN Bus Interface demonstration 1. Demonstrates how Server and CAN bus communication was prototyped for the first time it in the server. This example script in the server was used to communicate with the CAN bus. It is seen that inside a while loop CAN bus commands are sent at half second interval to simulate real time driving situation

Figure 12 is a git diff between the script shown in Figure 11 and the next iteration of script that was used to prototype communications with the server and CAN bus. This script was used to stress test the CAN bus and the server interface.

Figure 12: CAN bus Interface Demonstration 2. Demonstrates second iteration of how Server and CAN bus communication was prototyped in the server. This example script in the server was used to communicate with the CAN bus. It was used to stress test the server and CAN bus interface. As the diff shows the time between sending commands and receiving response back was decreased significantly.

As observed in Figure 11, the delay between sending a command to CAN Bus and waiting for the response was 1 second in the first try. Since this was performed in a while loop, this process appeared to function properly in a simulated environment. The simulation was blinking leds when can bus commands were sent to different CAN Bus. To stress test the interface, the wait time between sending and receiving CAN Bus messages was decreased by a factor of 10 to 0.1 second. As a result, the blinking leds were still observed. The purpose of this experiment was to stress the

scenario when the CAN Bus will be functional; based on the results it can be safely concluded CAN interface worked correctly.

3.7 Kinect Vision System Results:

Due to not having sufficient infrastructure, the test results of the vision system were obtained via console outputs indicating what the vehicle should do based on Table 2. Scenarios of decisions for the vehicle to move forward, brake, turn right, and turn left were simulated by modifying the depth map data shown in Figure 10. As shown in Figure 10, the depth map that was converted to get different scenarios for obtaining outputs from the vision system it is supposed to be taken with a grain of salt.There are several inconsistencies with the data that was collected in from Room AK318 including card board walls, chairs, tables, open area and other obstacles in the room that does not at all represent the situation of a road where the vehicle will be driving around. These are annotated in Figure 13

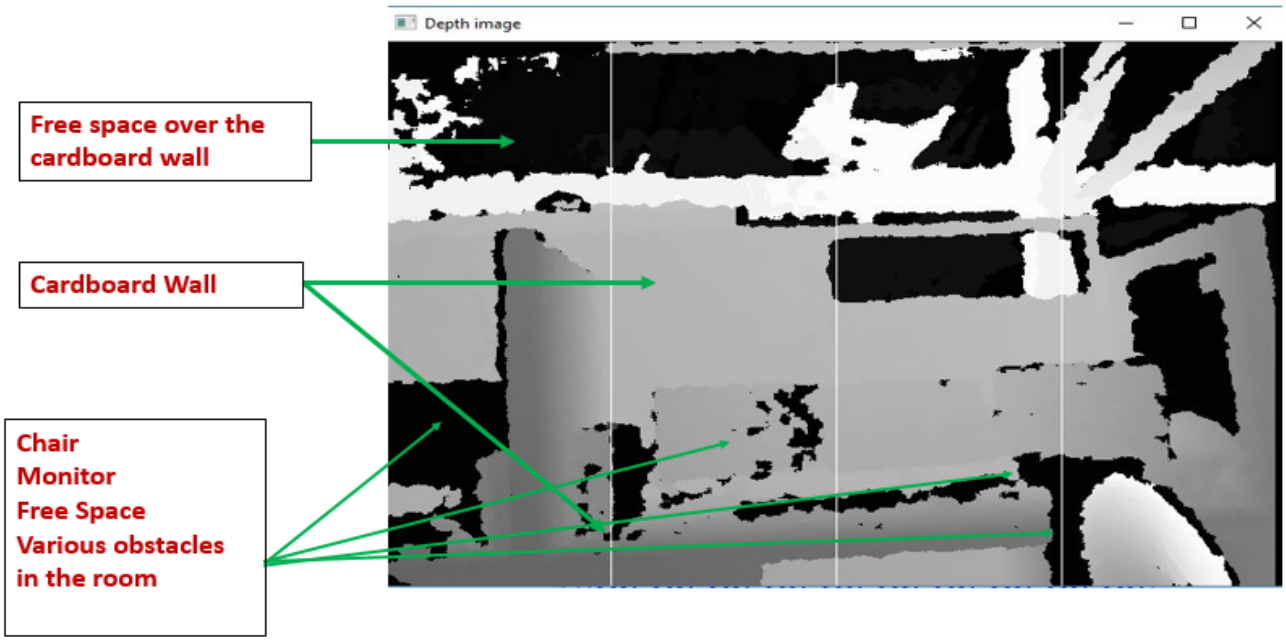


Figure 13: Annotated inconsistencies in Room AK318 taken in 18th February 2018. It is seen that there are different inconsistencies in the depth image i.e.: chair, Monitor, Cardboard wall, free space over the wall, etc. do not necessarily represent the situation in a road.

3.7.1 Scenario 1 Move forward:

In this scenario obstacle avoidance algorithm sends the signal to move forward as the average depth of the whole frame was greater than 5 meters. In Figure 14, the depth image of Room AK318 taken in February 2018 is shown. There were several inconsistencies, i.e, Cardboard wall, chair, monitor, various obstacles in the room, free space over the wall are annotated in Figure 14.

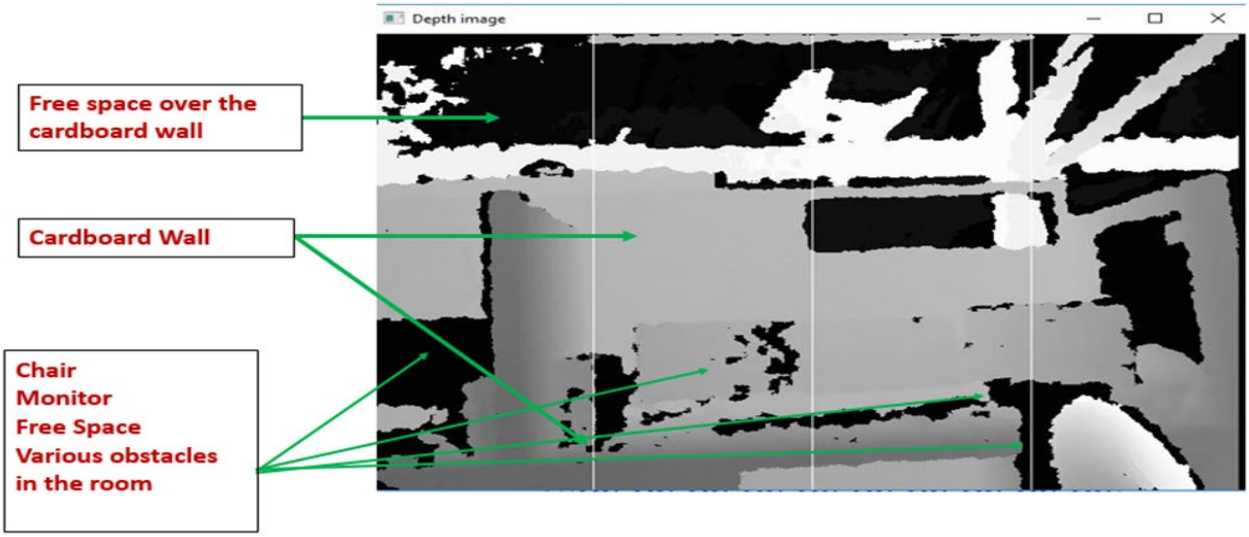
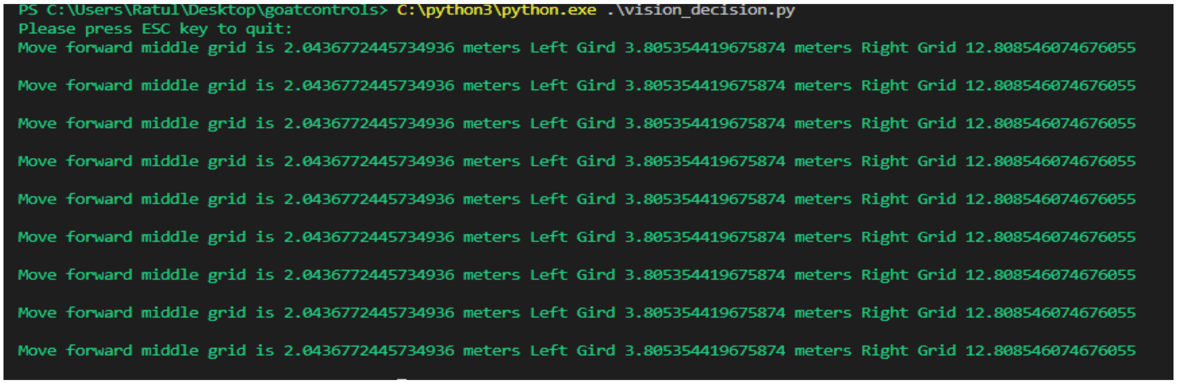


Figure 14: Depth Map of Room AK318 on February 19th, 2018 it as annotated with free space over the cardboard wall, Cardboard wall, Chair, monitor, various obstacles. The laboratory is located at Atwater Kent Building 3rd Floor the conditions at the lab do not represent an area where the Goat Cart will be driven around.

Figure 15 shows the movement decision and the average depth value printed on the console. It can be deduced from the average depth value printed on to the console that average depth in this scenario was three meters. As a result the vision system printed move forward on the console based on *Table 2*. However, the results needs to be taken with a grain of salt as the lighting of the room and inconsistencies as depicted in Figure 14 did have an effect on the depth value.



```
PS C:\Users\Ratul\Desktop\goatcontrols> C:\python3\python.exe .\vision_decision.py
Please press ESC key to quit:
Move forward middle grid is 2.0436772445734936 meters Left Gird 3.805354419675874 meters Right Grid 12.808546074676055

Move forward middle grid is 2.0436772445734936 meters Left Gird 3.805354419675874 meters Right Grid 12.808546074676055

Move forward middle grid is 2.0436772445734936 meters Left Gird 3.805354419675874 meters Right Grid 12.808546074676055

Move forward middle grid is 2.0436772445734936 meters Left Gird 3.805354419675874 meters Right Grid 12.808546074676055

Move forward middle grid is 2.0436772445734936 meters Left Gird 3.805354419675874 meters Right Grid 12.808546074676055

Move forward middle grid is 2.0436772445734936 meters Left Gird 3.805354419675874 meters Right Grid 12.808546074676055

Move forward middle grid is 2.0436772445734936 meters Left Gird 3.805354419675874 meters Right Grid 12.808546074676055

Move forward middle grid is 2.0436772445734936 meters Left Gird 3.805354419675874 meters Right Grid 12.808546074676055

Move forward middle grid is 2.0436772445734936 meters Left Gird 3.805354419675874 meters Right Grid 12.808546074676055
```

Figure 15: Console output from Vision System to move forward and log of average depth in different grids. The result is movement decision output that the vision decision system produces when the vision system is feed with the depth map shown in Figure 14.

3.7.2 Scenario 2 Turn Around Left:

In this scenario, the obstacle avoidance algorithm sends the signal to turn around left in accordance to scenarios listed in Table 2. Depth of the left grid was 13 meters so the obstacle avoidance algorithm sends signal to go around left. Figure 16 and Figure 17 shows the depth map and console out from vision system respectively. This depth image was manipulated to have the left grid pixels annotated here to have the values so that vision system will give the vision decision to go left.

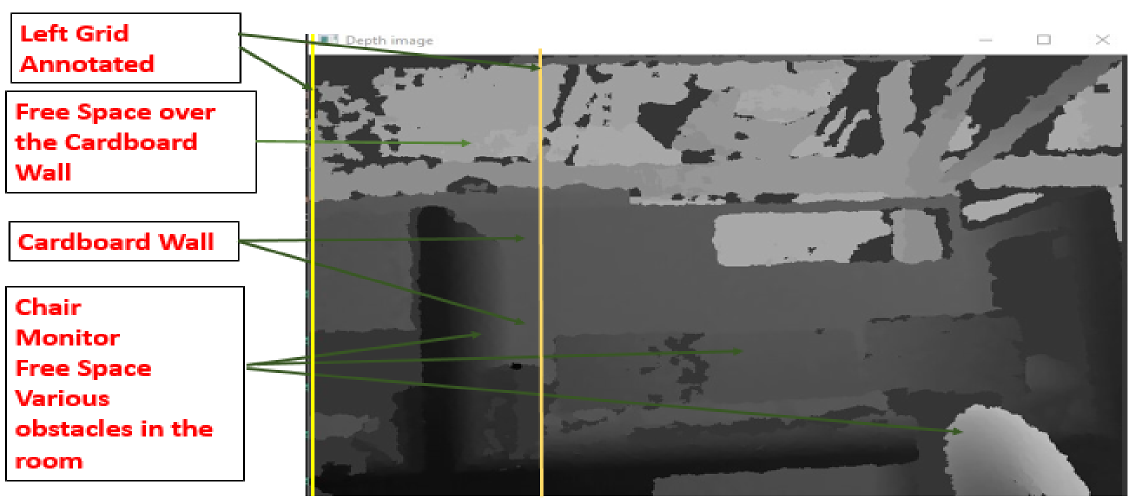


Figure 16: Depth Map of Room AK318 adopted from manipulating depth image found in Figure13. Cardboard wall, chair, monitors, free space is there as well the pixel values in the left grid are manipulated as well. Therefore, movement decision from this depth image was to turn left.

As annotated in Figure 16, the image is manipulated to have certain depth value, so that the movement decision to turn left is printed on to the console in accordance with the scenarios listed in Table 2. This result is depicted in Figure 17 as the image is manipulated, and all the inconsistencies of the depth image are annotated in Figure 16.

```
middle grid is 7.765386354841163 meters Left Gird 13.394491255309168 meters Right Grid 2.6203580941781515

Go around Left
middle grid is 7.765386354841163 meters Left Gird 13.394491255309168 meters Right Grid 2.6203580941781515

Go around Left
middle grid is 7.765386354841163 meters Left Gird 13.394491255309168 meters Right Grid 2.6203580941781515

Go around Left
middle grid is 7.765386354841163 meters Left Gird 13.394491255309168 meters Right Grid 2.6203580941781515

Go around Left
middle grid is 7.765386354841163 meters Left Gird 13.394491255309168 meters Right Grid 2.6203580941781515

Go around Left
middle grid is 7.765386354841163 meters Left Gird 13.394491255309168 meters Right Grid 2.6203580941781515
```

Figure 17: Console output from Vision System to turn left and log of average depth in different grids. The result is movement decision output that the vision decision system produces when the vision system is feed with the depth map shown in Figure 16.

3.7.3 Scenario 3 Turn Around Right:

In this scenario, depicted by the depth image in Figure 18, the vision system recommends to turn around right in accordance to scenarios listed in Table 2. In this scenario, the depth for the left grid is 5 meters so obstacle avoidance algorithm sends signal to turn around right. Figure 18 and Figure 19 shows the depth map and the console output from vision system, respectively. This depth image was manipulated to have the right grid pixels annotated here to have the values so that vision system recommended to turn right.

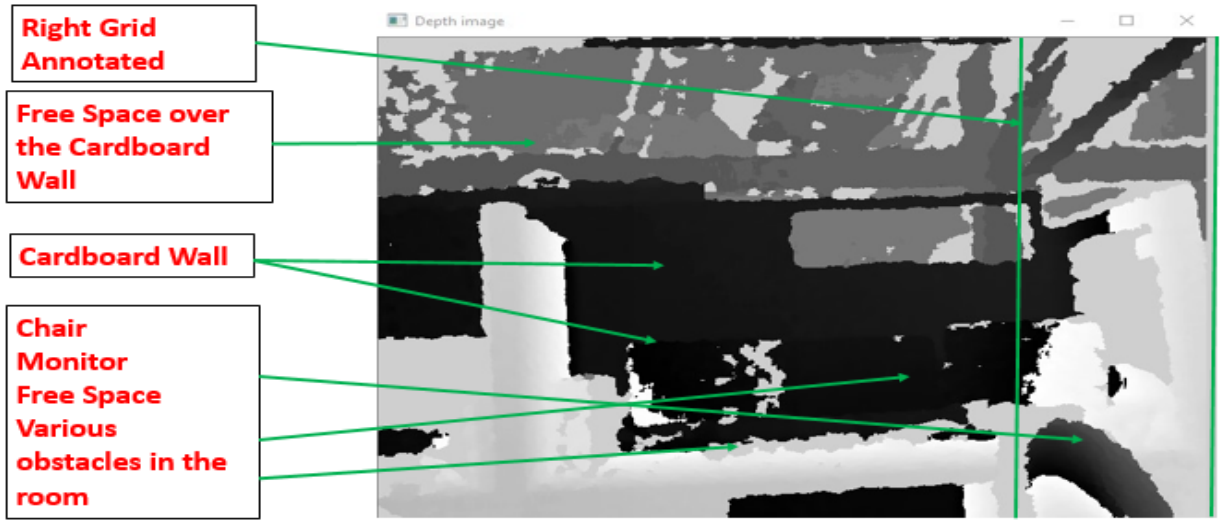


Figure 18: : Depth Map of Room AK318 adopted from manipulating depth image found in Figure13. Cardboard wall, chair, monitors, free space is there as well the pixel values in the right grid are manipulated as well. Therefore, movement decision from this depth image was to turn right.

As annotated in Figure 18, the image is manipulated to have certain depth value, so that the movement decision to turn right is printed onto the console in accordance with the scenarios listed in Table 2. This result is depicted in Figure 19 and needs to be examined as the image is manipulated, and the inconsistencies of the depth image that are annotated in Figure 18.

```
middle grid is 1.1739869023215814 meters Left Gird 1.6634822788900432 meters Right Grid 4.439278004563455

Go around Right
middle grid is 1.1739869023215814 meters Left Gird 1.6634822788900432 meters Right Grid 4.439278004563455

Go around Right
middle grid is 1.1739869023215814 meters Left Gird 1.6634822788900432 meters Right Grid 4.439278004563455

Go around Right
middle grid is 1.1739869023215814 meters Left Gird 1.6634822788900432 meters Right Grid 4.439278004563455

Go around Right
middle grid is 1.1739869023215814 meters Left Gird 1.6634822788900432 meters Right Grid 4.439278004563455

Go around Right
middle grid is 1.1739869023215814 meters Left Gird 1.6634822788900432 meters Right Grid 4.439278004563455
```

Figure 19: Console output from Vision System to turn Right and log of average depth in different grids. The result is movement decision output that the vision decision system produces when the vision system is feed with the depth map shown in Figure 18

3.7.4 Scenario 4 Brake:

In this scenario, obstacle avoidance algorithm sends the signal to move brake as the average depth of the whole frame was lesser than 1 meter. Figure 20 and Figure 21 shows depth map and console output for braking respectively. . In this scenario, depth value for the annotated grid is less than 2 meters so obstacle avoidance algorithm recommends to brake. This depth image was manipulated to have the grid pixels annotated here to have the values so that vision system recommended to brake.

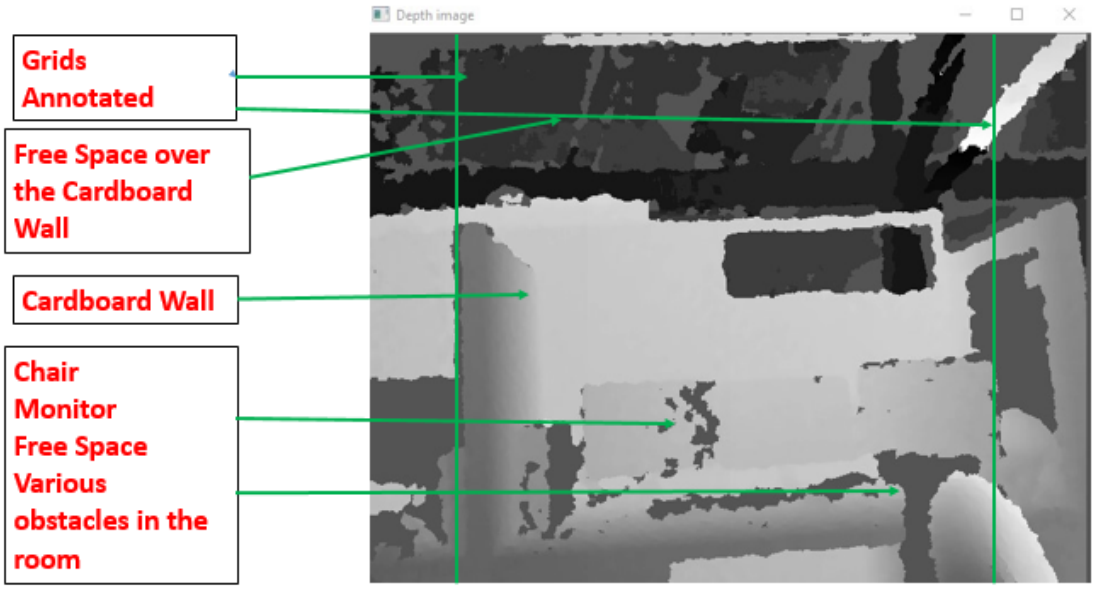


Figure 20: : Depth Map of Room AK318 adopted from manipulating depth image found in Figure13. Cardboard wall, chair, monitors, free space is there as well the pixel values in the annotated grids are manipulated as well. Therefore, movement decision from this depth image was to turn right.

As annotated in Figure 20 the image is manipulated to have certain depth value, so that the movement decision to brake is printed on to the console in accordance with the scenarios listed in Table 2. This result is depicted in Figure 21 and needs to be examined as the image is manipulated, and all the inconsistencies of the depth image are annotated in Figure 18.

```
Hit Brakes!!
middle grid is 0.6446702430436027 meters Left Gird 0.7994642345138886 meters Right Grid 1.2436943718449638

Hit Brakes!!
middle grid is 0.6446702430436027 meters Left Gird 0.7994642345138886 meters Right Grid 1.2436943718449638

Hit Brakes!!
middle grid is 0.6446702430436027 meters Left Gird 0.7994642345138886 meters Right Grid 1.2436943718449638
```

Figure 21: Console output from Vision System to brake and log of average depth in different grids. The result is movement decision output that the vision decision system produces when the vision system is feed with the depth map shown in Figure 20

3.8 Lidar Selection for Goat Cart:

For the application of obstacle avoiding, a vision system it is recommended to use a RP Lidar. To come up with the design choice, the following Lidar specifications listed in Table 3 were evaluated in an evaluation matrix.

Table 3: Lidar Evaluation Matrix

Lidar / **Requirements**	**Lidar Lite** [17]	**Slamtec RPLidar A1** [18]	**Tf Mini Lidar** [19]	**Score Lidar Lite**	**Score Slamtec RPLidar A1**	**Score Tf Mini Lidar**
Cost	$129.99	$114.95	$39.90	2	2	3
Resolution	1 cm	0.2cm	5 mm	2	3	3
Accuracy	+/- 2.5 cm for distance > 1 meter	0.2%	1 % less than 6 m, 2 % 6 – 12 m			
Range	5 Cm to 40 meters	12 meters	0.3m co– 12 m	4	3	2
Operating Voltage	4.75 – 5 V	5 V	4.5 – 6 V	2	2	2
Current Consumption	105mA idle; 130mA continuous	100 mA	800 mA (peak current)	2	2	2
Interface	I2C or PWM	UART	UART	1	3	3
Total Score	N/A	N/A	N/A			

As shown in Table 3, different Lidars were selected for evaluation based on cost, resolution, accuracy, range, operating voltage, current consumption and interface. From the data shown in Table 3; the RP Lidar is within budget and has the most accuracy and range; furthermore; it has the capability to interface with Teensy 3.5 Boards. Therefore, RP Lidar is recommended to use for a Lidar based vision system for the Goat Cart.

In the case of Lidar, the prototyped vision uses system similar kind of vision system compared to the Kinect vision system. It can use similar algorithms to make decisions such as go forward, turn left and right, or to stop. The Lidar will be used to measure distance how far the object is from the vehicle.

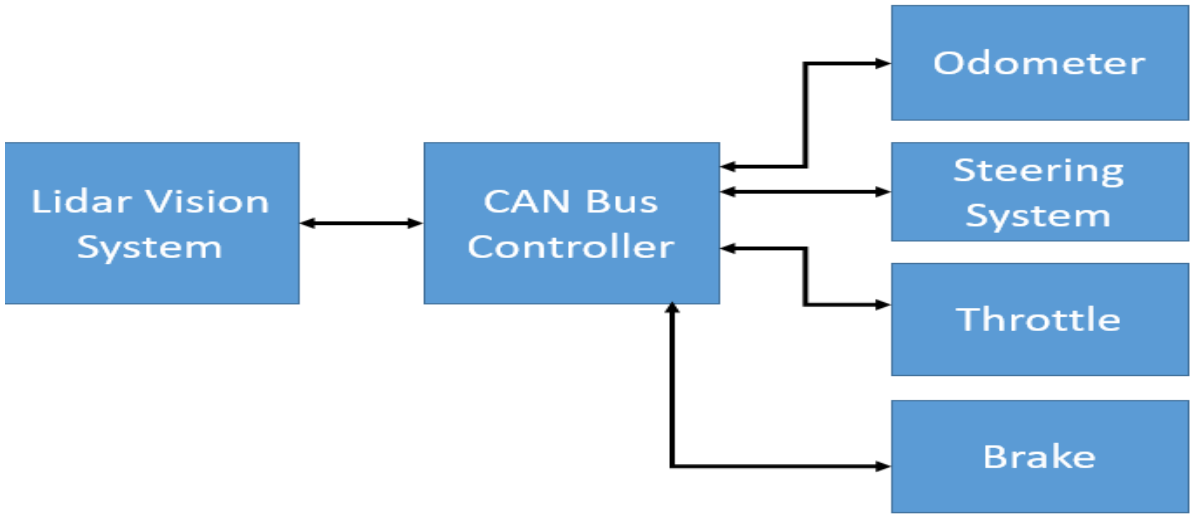


Figure 22 : LIDAR Vision System as seen here will be interfacing with the CAN bus controller to send messages to throttle, Steering System, Brake and the Odometer System

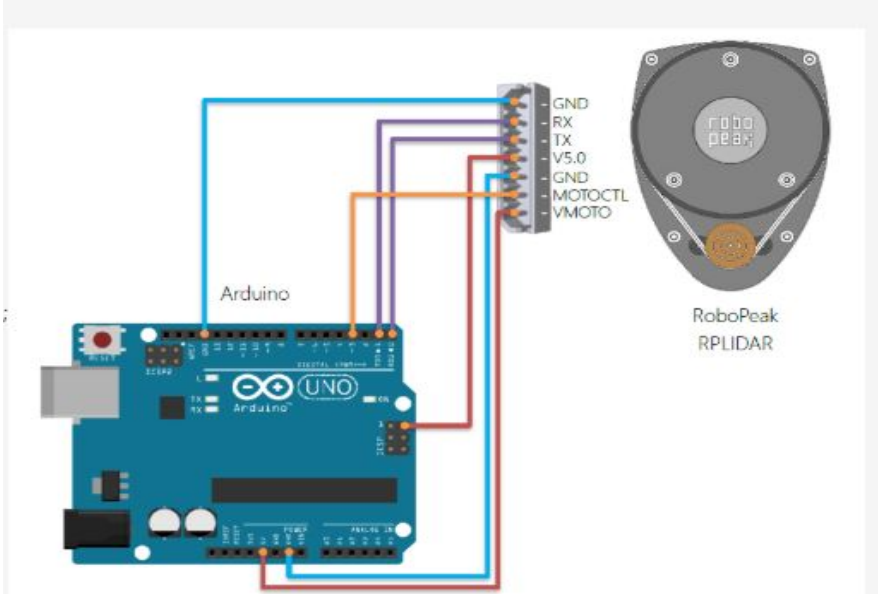


Figure 23: LIDAR Circuit Diagram. Here the Arduino represents the Master CAN bus that will interface with the existing CAN bus infrastructure that is available in the Goat Cart Platform. In the case of a LIDAR based vision system

A controller Arduino (see Figure 23) to interface with the Lidar so the Arduino can perform necessary calculations to make decisions that will be necessary for a Lidar based vision system.

The RP Lidar has capabilities to provide substantial amount of data so any project working with RP Lidar should be able to process the data timely in order to use it. In order to be able to use RP Lidar, it is essential to interface with the Lidar and Teensy 3.5. After interfacing with the Lidar and Teensy 3.5 it will be possible to send messages to different components of the CAN Bus system. An example to connect with RP Lidar with Arduino/ Teensy 3.5 can be found in Appendix 5. Lidar with a controller Teensy 3.5/ Arduino will make efforts to communicate with the CAN Bus if any obstacle is seen. The actions can be similar to the actions that were taken in the Kinect based vision system. Table 2 shows some of the decisions that can be taken once Lidar will be detecting objects that are near the Goat Cart.

3.11 Chapter Summary:

Kinect views the world in 640x360 grid. An obstacle avoidance algorithm was developed by dividing the 640x360 grid into 3 grids (front, left, right). The average depth values of the grids mapped the obstacles in front of the camera. One suggested improvement is adding more grids and calibrate the camera. After more testing when the system is ready to be integrated more grids can be added to make the decision making more robust.

Chapter 4 Conclusion and Future Works

After this project, a vision system (with Microsoft Kinect) Linux server, and CAN Bus communication, CAN Bus control (with Teensy 3.5) were successfully prototyped. This project successfully achieved an operational vision system as described in Chapter 3. The vision system detected obstacles and was able to take different decisions for the Goat Cart shown in Chapter 3.3. Interface with the server and CAN Bus prototyped as described in Chapter 2.4.

Once the hardware is integrated, the vision system along with other driving functionalities can be tested and improved upon. Due to resource constraints hardware integration was not performed. The issues that could have been improved are:

1) Hardware integration;
2) Autonomous turning right and left;
3) Autonomous driving functions *i.e.* speed calculation, autonomous speed increase and decrease;
4) Machine Learning in Vision System.

Works Cited

[1] Administraton, National Highway Traffic Safety, "The Economic and Societal Impact of Motor Vehicle Crashes," May 2015.

[2] B. W. Smith, "HUMAN ERROR AS A CAUSE OF VEHICLE CRASHES," Standford Law School, 18 December 2013. [Online]. Available: http://cyberlaw.stanford.edu/blog/2013/12/human-error-cause-vehicle-crashes.

[3] K. Kumar, "The Role of Perceptual and Cognitive Filters in Observed Behavior," in *Human Behaviour and Traffic Safety*, 1985, pp. 151-170.

[4] Ching-YaoChan, "Advancements, prospects, and impacts of automated driving systems," *Science Direct,* vol. 6, no. 3, p. 2, 2017.

[5] European Environmental Agency, "Total Greenhouse Gas Emission Trends and Projections; Technical Report,," Copenhagen, Denmark, 2017.

[6] "SAE International AUTOMATED DRIVING, Levels of driving," SAE International , [Online]. Available: https://www.sae.org/news/press-room/2018/12/sae-international-releases-updated-visual-chart-for-its-%E2%80%9Clevels-of-driving-automation%E2%80%9D-standard-for-self-driving-vehicles.

[7] B. Decker, "National Transportation Safety Board (NTSB) Preliminary Report Highway: HWY18MH010.," May 2018. [Online]. Available: : https://www.ntsb.gov/investigations/AccidentReports/.

[8] US Department of Transportation, "National Highway Traffic Safety Administration Federal Automated Vehicles Policy," [Online]. Available: https://www.transportation.gov/AV.

[9] "Sabertooth 2x60 User's Guide - Dimension Engineering," 20 March 2018. [Online]. Available: https://www.dimensionengineering.com/datasheets/Sabertooth2x60.pdf. [Accessed 21 March 2018].

[10] "Teensy USB Development Board," 01 04 2018. [Online]. Available: https://www.pjrc.com/teensy/. [Accessed 01 04 2018].

[11] "Magnetic Encoder Chip User's Guide," 10 April 2018. [Online]. Available: https://www.ctr-electronics.com/downloads/pdf/Magnetic%20Encoder%20User's%20Guide.pdf.

[12] R. I. Davis, A. Burns, R. J. Bril and J. J. Lukkien,, "Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised," Real-Time Systems, vol. 35, no. 3, pp. 239-272,, 2007.

[13] Team, 2017-18 Golf Cart MQP, "Goat Cart; An Autonomous Golf Cart," WPI, Worcester,MA, 2018.

[14] David J Baker, Alexander Briskman, Camila Di Fino Napolitano, Matthew A Mahan,Jared Greene Perkins,Jade A Pierce, "Goat Cart; An Autonomous Golf Cart," WPI, Worcester,MA, 2018.

[15] "What is Kinect?," Search Health IT, 21 March 2018. [Online]. Available: https://searchhealthit.techtarget.com/definition/Kinect.

[16] R. Kabrawala, "Improvement of an indoor blind navigation system based upon depth sensing," 2016. [Online]. Available: https://www.researchgate.net/figure/Components-in-Microsoft-Kinect-v10-3-The-working-protocol-of-the-system-is-based-on_fig1_299513399.

[17] "Lidar Lite v3 Operational Manual and Technical Specifications," [Online]. Available: http://static.garmin.com/pumac/LIDAR_Lite_v3_Operation_Manual_and_Technical_Specifications.pdf.

[18] "RP LIdar A1 Low Cost 360 Laser Range Sensor," 12 02 2019. [Online]. Available: http://bucket.download.slamtec.com/b90ae0a89feba3756bc5aaa0654c296dc76ba3ff/LD108_SLAMTEC_rplidar_datasheet_A1M8_v2.2_en.pdf.

[19] "SeedStudio Grove- TF Mini Lidar Datasheet," Seed Development Limited (seedstudo.com) .

[20] World Health Organization, "World Health Organization," January 2018. [Online]. Available: http://www.who.int/mediacentre/factsheets/fs358/en/. [Accessed 20 February 2018].

[21] D. J. Fagnant and K. Kockelman, "Preparing a nation for autonomous vehicles: opportunities, barriers and policy recommendations," *Elsevier Ltd,* vol. A, no. 77, pp. 167-181, 2015.

[22] B. D. Lawrence, "A vision of our transport future," *Macmillan Publishers Limited,* vol. 497, pp. 182-182, 2013.

[23] W. Zhang, S. Guhathakurta, J. Fang and G. Zhang, "Exploring the impact of shared autonomous vehicles on urban parking demand: An agent-based simulation approach," *Sustainable Cities and Society,* vol. 19, pp. 34-35, 2015.

[24] K. Heineke , P. Kampshoff, A. Mkrtchyan and E. Shao, "Self-driving car technology: When will the robots hit the road?," McKinsey&Company, May 2017. [Online]. Available: https://www.mckinsey.com/industries/automotive-and-assembly/our-insights/self-driving-car-technology-when-will-the-robots-hit-the-road. [Accessed 20 February 2018].

[25] "National Ocean Service," National Oceanic and Atmospheric Administration , 23 06 2018. [Online]. Available: https://oceanservice.noaa.gov/facts/lidar.html.

[26] "Depth Map Metadata," Google , 17 March 2018. [Online]. Available: https://developers.google.com/depthmap-metadata.

[27] Comerón A, Muñoz-Porcar C, Rocadenbosch F, Rodríguez-Gómez A, Sicard M., "Current Research in Lidar Technology Used for the Remote Sensing of Atmospheric Aerosols," *Sensors (Basel).,* vol. 17, 2017.

[28] Thrun, S., Burgard, W. and Fox,D, Probabilistic Robotics., Cambridge, Mass: MIT Press, 2005.

Appendix 1 : Configure Kinect and Linux Interface.

Here are the steps to get started with using the kinect :-

Note :- This tutorial assumes that you have Linux(Ubuntu or Ubuntu based Linux distro) with opencv installed on your system.

1) Open a terminal and run the following commands

   sudo apt –get update & upgrade

2) Install the necessary dependencies.

   sudo apt-get install git-core cmake freeglut3-dev pkg-config build-essential libxmu-dev libxi-dev libusb-1.0-0-dev

3) Clone the libfreenect repository to your file system.

```
git clone git://github.com/OpenKinect/libfreenect.git
```

4) Go to the libfreenect repository and install libfreebect.

```
cd libfreenect

mkdir build

cd build

cmake -L ..
```

```
make

sudo make install

sudo ldconfig /usr/local/lib64/
```

5) To use Kinect as a non-root user do the following

```
sudo adduser $USER video

sudo adduser $USER plugdev
```

6) Also make a file with rules for the Linux device manager

```
1

sudo nano /etc/udev/rules.d/51-kinect.rules
```

Then paste the following and save

Appendix 2: Kinect Rules

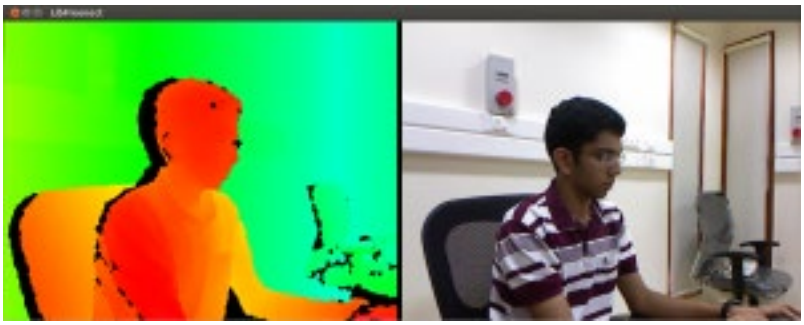
```
# ATTR{product}=="Xbox NUI Motor"

SUBSYSTEM=="usb", ATTR{idVendor}=="045e", ATTR{idProduct}=="02b0",
MODE="0666"

# ATTR{product}=="Xbox NUI Audio"

SUBSYSTEM=="usb", ATTR{idVendor}=="045e", ATTR{idProduct}=="02ad",
MODE="0666"
```

```
# ATTR{product}=="Xbox NUI Camera"

SUBSYSTEM=="usb", ATTR{idVendor}=="045e", ATTR{idProduct}=="02ae",
MODE="0666"

# ATTR{product}=="Xbox NUI Motor"

SUBSYSTEM=="usb", ATTR{idVendor}=="045e", ATTR{idProduct}=="02c2",
MODE="0666"

# ATTR{product}=="Xbox NUI Motor"

SUBSYSTEM=="usb", ATTR{idVendor}=="045e", ATTR{idProduct}=="02be",
MODE="0666"

# ATTR{product}=="Xbox NUI Motor"

SUBSYSTEM=="usb", ATTR{idVendor}=="045e", ATTR{idProduct}=="02bf",
MODE="0666"
```

7) Log out and back in. Run the following command in a terminal to test if libfreenect is correctly installed

```
1

   freenect-glview
```

This should cause a window to pop up showing the depth and RGB images. Pressing ‘w’ on the keyboard causes the kinect to tilt up and pressing ‘x’ causes the kinect to tilt down. There are several other control options that are listed in the terminal when “freenect-glview” is run



8) In order to use the Kinect with opencv and python, the python wrappers for libfreenct need to be installed. Before doing that, install the necessary dependencies

```
sudo apt-get install cython
```

```
sudo apt-get install python-dev
```

```
sudo apt-get install python-numpy
```

9) Go to the directory ……./libfreenect/wrappers/python and run the following command

```
sudo python setup.py install
```

10) Save the code given below as a (.py) file say (kinect_test.py)

#import the necessary modules

import freenect

import cv2

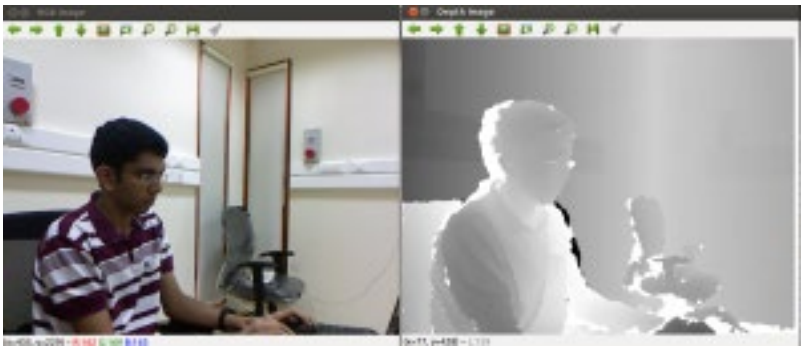
import numpy as np

```
#function to get RGB image from kinect

def get_video():

    array,_ = freenect.sync_get_video()

    array = cv2.cvtColor(array,cv2.COLOR_RGB2BGR)

    return array


#function to get depth image from kinect

def get_depth():

    array,_ = freenect.sync_get_depth()

    array = array.astype(np.uint8)

    return array


if __name__ == "__main__":

    while 1:

        #get a frame from RGB camera

        frame = get_video()

        #get a frame from depth sensor

        depth = get_depth()

        #display RGB image

        cv2.imshow('RGB image',frame)
```

```
        #display depth image
        cv2.imshow('Depth image',depth)


        # quit program when 'esc' key is pressed
        k = cv2.waitKey(5) & 0xFF
        if k == 27:
            break
    cv2.destroyAllWindows()
```

11) Run the above program

```
python kinect_test.py
```



Hit the ‘esc’ key to quit the program. For further information, check

1) Open kinect page :- http://openkinect.org/wiki/Main_Page

2) libfreenect github page :- https://github.com/OpenKinect/libfreenect

Appendix 3 : Depth to distance conversion

From their data, a basic first order approximation for converting the raw 11-bit disparity value to a depth value in centimeters is: 100/(-0.00307 * rawDisparity + 3.33). This approximation is approximately 10 cm off at 4 m away, and less than 2 cm off within 2.5 m.

A better approximation is given by Stéphane Magnenat in this post: distance = 0.1236 * tan(rawDisparity / 2842.5 + 1.1863) in meters. Adding a final offset term of -0.037 centers the original ROS data. The tan approximation has a sum squared difference of .33 cm while the 1/x approximation is about 1.7 cm.

Appendix 4: Vision System Tutorial Python Example Script

```
#import the necessary modules

import freenect

import cv2

import numpy as np

import math


#function to get RGB image from kinect

def get_video():

    array,_ = freenect.sync_get_video()

    array = cv2.cvtColor(array,cv2.COLOR_RGB2BGR)

    return array


#get raw depth data from kinect

def get_depth():

    array,_ = freenect.sync_get_depth()

    array = array.astype(np.uint16)

    return array


#function to get depth image from kinect

def get_depth_image():

    array,_ = freenect.sync_get_depth()

    array = array.astype(np.uint8)

    return array
```

```
def func(csv):
    #print(type(csv))
    #print(csv.shape)
    a = np.mean(csv)
    distance = 0.1236*math.tan(a/2842.5+1.1863)
    return distance




if __name__ == "__main__":
    print("Please press ESC key to quit:")
    while 1:
        #get a frame from RGB camera
        frame = get_video()
        #get a frame from depth sensor
        depth = get_depth()
        depth_image = get_depth_image()
        #display RGB image
        cv2.imshow('RGB image',frame)
        #display depth image
        cv2.imshow('Depth image',depth_image)
        if(func(depth)<1.5):
            print("BOOOM")
```

```
        else:
            print("\n")
            pass


        # quit program when 'esc' key is pressed
        k = cv2.waitKey(5) & 0xFF
        if k == 27:
            print("ESC Key pressed quiting")
            break
    cv2.destroyAllWindows()
```

Appendix 5: RP LIDAR Arduino

```
// This sketch code is based on the RPLIDAR driver library provided by RoboPeak
#include <RPLidar.h>

// You need to create an driver instance
RPLidar Lidar;

// Change the pin mapping based on your needs.
////////////////////////////////////////////////////////////////////////////////
#define LED_ENABLE  12 // The GPIO pin for the RGB led's common lead.
                       // assumes a common positive type LED is used
```

```
#define LED_R      9  // The PWM pin for drive the Red LED
#define LED_G      11 // The PWM pin for drive the Green LED
#define LED_B      10 // The PWM pin for drive the Blue LED

#define RPLIDAR_MOTOR 3 // The PWM pin for control the speed of RPLIDAR's motor.
                    // This pin should connected with the RPLIDAR's MOTOCTRL signal
//////////////////////////////////////////////////////////////////////////////




#define NEO_RGBSPACE_MAX (byte)(200L*255/360)
int _r, _g, _b;

//Set current RGB with the hue: HSV(hue, x, x)
void hue_to_rgb( _u8 hue)
{
/*`
  Hue(in Degree):  0 (RED) ----> 60 (Yello) ----> 120 (Green) --...... ----> 360
  Hue'(fit to _u8):0      ----> 60/360*255 ----> 120/260*255 --...... ----> 255
*/

  //convert HSV(hue,1,1) color space to RGB space
  if (hue < 120L*255/360)   //R->G
  {
    _g = hue;
    _r = NEO_RGBSPACE_MAX - hue;
    _b = 0;
  }else if (hue < 240L*255/360)  //G->B
  {
```

```
      hue -= 120L*255/360;
      _b = hue;
      _g = NEO_RGBSPACE_MAX - hue;
      _r = 0;
   }else //B->R
   {
      hue -= 240L*255/360;
      _r = hue;
      _b = NEO_RGBSPACE_MAX - _r;
      _g = 0;
   }
}

void displayColor(float angle, float distance)
{
   //1. map 0-350 deg to 0-255
   byte hue = angle*255/360;
   hue_to_rgb(hue);

   //2. control the light

   int lightFactor = (distance>500.0)?0:(255-distance*255/500);
   _r *=lightFactor;
   _g *=lightFactor;
   _b *=lightFactor;

   _r /= 255;
   _g /= 255;
   _b /= 255;
```

```
   analogWrite(LED_R, 255-_r);
   analogWrite(LED_G, 255-_g);
   analogWrite(LED_B, 255-_b);
}

void setup() {
 // bind the RPLIDAR driver to the arduino hardware serial
 Lidar.begin(Serial);

 // set pin modes
 pinMode(RPLIDAR_MOTOR, OUTPUT);

 pinMode(LED_ENABLE, OUTPUT);
 pinMode(LED_R, OUTPUT);
 pinMode(LED_G, OUTPUT);
 pinMode(LED_B, OUTPUT);

 digitalWrite(LED_ENABLE, HIGH);

 analogWrite(LED_R,255);
 analogWrite(LED_G,255);
 analogWrite(LED_B,255);
}

float minDistance = 100000;
float angleAtMinDist = 0;

void loop() {
```

```
 if (IS_OK(Lidar.waitPoint())) {
  //perform data processing here...
  float distance = Lidar.getCurrentPoint().distance;
  float angle = Lidar.getCurrentPoint().angle;

  if (Lidar.getCurrentPoint().startBit) {
   // a new scan, display the previous data...
    displayColor(angleAtMinDist, minDistance);
    minDistance = 100000;
    angleAtMinDist = 0;
  } else {
    if ( distance > 0 &&  distance < minDistance) {
      minDistance = distance;
      angleAtMinDist = angle;
    }
  }
 } else {
  analogWrite(RPLIDAR_MOTOR, 0); //stop the rpLidar motor

  // try to detect RPLIDAR...
  rpLidar_response_device_info_t info;
  if (IS_OK(Lidar.getDeviceInfo(info, 100))) {
    //detected...
    Lidar.startScan();
    analogWrite(RPLIDAR_MOTOR, 255);
    delay(1000);
  }
 }
}
```