DEVELOPING A PORTABLE DATA ACQUISITION SYSTEM TO STUDY ROAD USER BEHAVIOR

DRAFT FINAL PROJECT REPORT

by

Vinod Vasudevan, Ph.D., P.E.
Shawn Butler, Ph.D.

University of Alaska Anchorage

Sponsorship
PacTrans and University of Alaska Anchorage

for

Pacific Northwest Transportation Consortium (PacTrans)
USDOT University Transportation Center for Federal Region 10
University of Washington
More Hall 112, Box 352700
Seattle, WA 98195-2700

In cooperation with U.S. Department of Transportation,
Research and Innovative Technology Administration (RITA)

DISCLAIMER

The contents of this report reflect the views of the authors, who are responsible for the facts and the accuracy of the information presented herein. This document is disseminated under the sponsorship of the U.S. Department of Transportation's University Transportation Centers Program, in the interest of information exchange. The Pacific Northwest Transportation Consortium, the U.S. Government and matching

TECHNICAL REPORT DOCUMENTATION PAGE

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
	01764463	

4. Title and Subtitle	5. Report Date
Developing a Portable Data Acquisition System to Study Road User Behavior	February 15, 2023
	6. Performing Organization Code

7. Author(s) and Affiliations	8. Performing Organization Report No.
Vinod Vasudevan, Ph.D., P.E.; 0000-0003-4078-9213 Shawn Butler, Ph.D.	2020-S-UAF-2

9. Performing Organization Name and Address	10. Work Unit No. (TRAIS)
PacTrans Pacific Northwest Transportation Consortium University Transportation Center for Federal Region 10 University of Washington More Hall 112 Seattle, WA 98195-2700	**11. Contract or Grant No.** 69A3551747110

12. Sponsoring Organization Name and Address	13. Type of Report and Period Covered
United States Department of Transportation Research and Innovative Technology Administration 1200 New Jersey Avenue, SE Washington, DC 20590	Draft Final Report **14. Sponsoring Agency Code**

15. Supplementary Notes

Report uploaded to: www.pactrans.org

16. Abstract

Data on driver behavior under various circumstances are essential for understanding traffic and quantifying traffic safety. Traditional data collection approaches (such as using driving simulators and video-graphic methods) possess various drawbacks. Recent advances in technology make instrumented vehicles (IV) affordable. In such an experimental setup, a portable data acquisition system (PDAQS) is equipped with various sensors, including light detection and ranging (LiDAR) sensors, to capture drivers' behavior when they are driving the vehicle as naturally as possible. A PDAQS helps to accurately observe instantaneous vehicle information and driver behavior for longer road sections and durations. A PDAQS can be used as a probe unit to capture drivers' driving behavior around the IV. The data that can be obtained accurately at high frequency include changes in headways and speeds and interactions among road users. Because the installation is portable, the same system can be used among different vehicles and locations (such as urban, rural, and different states).

In this project, we assembled a PDAQS and developed software to process the collected data. The tasks included developing the hardware and software programs to extract valuable data. On the software side, we developed a modularized framework capable of efficiently processing LiDAR and video data and detecting objects in two and three dimensions. Our main results were an alignment tool that adjusts and tests point transformation parameters, LiDAR/camera timing synchronization in real time, and the integration of point cloud detection into our framework. Although lots of progress was made, a additional future work must be done before the product can be applied to real-time projects.

17. Key Words	18. Distribution Statement
Instrumented vehicles, Probe vehicles, Detection and identification technologies,	

19. Security Classification (of this report)	20. Security Classification (of this page)	21. No. of Pages	22. Price
Unclassified.	Unclassified.	21	N/A

SI* (MODERN METRIC) CONVERSION FACTORS

APPROXIMATE CONVERSIONS TO SI UNITS				
Symbol	**When You Know**	**Multiply By**	**To Find**	**Symbol**
		LENGTH		
in	inches	25.4	millimeters	mm
ft	feet	0.305	meters	m
yd	yards	0.914	meters	m
mi	miles	1.61	kilometers	km
		AREA		
in^2	square inches	645.2	square millimeters	mm^2
ft^2	square feet	0.093	square meters	m^2
yd^2	square yard	0.836	square meters	m^2
ac	acres	0.405	hectares	ha
mi^2	square miles	2.59	square kilometers	km^2
		VOLUME		
fl oz	fluid ounces	29.57	milliliters	mL
gal	gallons	3.785	liters	L
ft^3	cubic feet	0.028	cubic meters	m^3
yd^3	cubic yards	0.765	cubic meters	m^3
	NOTE: volumes greater than 1000 L shall be shown in m^3			
		MASS		
oz	ounces	28.35	grams	g
lb	pounds	0.454	kilograms	kg
T	short tons (2000 lb)	0.907	megagrams (or "metric ton")	Mg (or "t")
		TEMPERATURE (exact degrees)		
°F	Fahrenheit	5 (F-32)/9 or (F-32)/1.8	Celsius	°C
		ILLUMINATION		
fc	foot-candles	10.76	lux	lx
fl	foot-Lamberts	3.426	$candela/m^2$	cd/m^2
		FORCE and PRESSURE or STRESS		
lbf	poundforce	4.45	newtons	N
lbf/in^2	poundforce per square inch	6.89	kilopascals	kPa

APPROXIMATE CONVERSIONS FROM SI UNITS				
Symbol	**When You Know**	**Multiply By**	**To Find**	**Symbol**
		LENGTH		
mm	millimeters	0.039	inches	in
m	meters	3.28	feet	ft
m	meters	1.09	yards	yd
km	kilometers	0.621	miles	mi
		AREA		
mm^2	square millimeters	0.0016	square inches	in^2
m^2	square meters	10.764	square feet	ft^2
m^2	square meters	1.195	square yards	yd^2
ha	hectares	2.47	acres	ac
km^2	square kilometers	0.386	square miles	mi^2
		VOLUME		
mL	milliliters	0.034	fluid ounces	fl oz
L	liters	0.264	gallons	gal
m^3	cubic meters	35.314	cubic feet	ft^3
m^3	cubic meters	1.307	cubic yards	yd^3
		MASS		
g	grams	0.035	ounces	oz
kg	kilograms	2.202	pounds	lb
Mg (or "t")	megagrams (or "metric ton")	1.103	short tons (2000 lb)	T
		TEMPERATURE (exact degrees)		
°C	Celsius	1.8C+32	Fahrenheit	°F
		ILLUMINATION		
lx	lux	0.0929	foot-candles	fc
cd/m^2	$candela/m^2$	0.2919	foot-Lamberts	fl
		FORCE and PRESSURE or STRESS		
N	newtons	0.225	poundforce	lbf
kPa	kilopascals	0.145	poundforce per square inch	lbf/in^2

*SI is the symbol for the International System of Units. Appropriate rounding should be made to comply with Section 4 of ASTM E380.
(Revised March 2003)

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

CAV:	Connected automated vehicle
CPU:	Central processing unit
CSV:	Comma-separated values
DBSCAN:	Density-Based Spatial Clustering of Applications with Noise
FOV:	Field of vision
IV:	Instrumented vehicles
GPS:	Global Positioning System
GPU:	Graphics processing unit
LiDAR:	Light detection and ranging
OBD:	On-board diagnostics
PacTrans:	Pacific Northwest Transportation Consortium
PCAP:	Packet capture
PDAQS:	Portable data acquisition system
TCP:	Transmission Control Protocol
VLP:	Visible light programming
VRAM:	Video random access memory
WSDOT:	Washington State Department of Transportation

ACKNOWLEDGMENTS

The work presented here is the result of four undergraduate students who worked on this project. This project also supported a couple of Capstone projects. We thank James Flemings, Jon Rippe, Ty Bergstrom, and Jack Sexauer from the University of Alaska Anchorage Department of Computer Science for their contributions.

EXECUTIVE SUMMARY

Data on driver behavior under various circumstances are essential for understanding traffic and quantifying traffic safety. Traditional data collection approaches (such as using driving simulators and video-graphic methods) possess various drawbacks. Therefore, innovative ways to gather continuous data on vehicle interactions with the driving environment need to be explored.

Recent advances in technology make instrumented vehicles (IV) affordable. In such an experimental setup, a portable data acquisition system (PDAQS) is equipped with various sensors, including light detection and ranging (LiDAR), to capture drivers' behavior when they drive the vehicle as naturally as possible. A PDAQS helps researchers accurately observe instantaneous vehicle information and driver behavior for longer road sections and durations. Also, a PDAQS can be used as a probe unit to capture drivers' driving behavior around the IV. The data that can be obtained accurately at high frequency include changes in headways and speeds and interactions among road users. Because the installation is portable, the same structure can be moved among different vehicles and locations (urban and rural) to capture driving and environment data for various conditions.

In this project, we assembled a PDAQS and developed software to process the collected data. The tasks included developing the hardware and software programs to extract valuable data. On the software side, we developed a modularized framework capable of efficiently processing LiDAR and video data and detecting objects in two and three dimensions. Our main results were an alignment tool that adjusts and tests point transformation parameters, LiDAR/camera timing synchronization in real time, and the integration of point cloud detection into our framework. Although lots of progress was made, some additional future work must be done before the product can be used for application

.

CHAPTER 1. INTRODUCTION

Capturing detailed road user behavior is critical for planning and designing a safe transportation infrastructure. Transportation researchers normally use data from sensors such as loop detectors, video detectors, Bluetooth sensors, and radar sensors to gather and analyze macroscopic traffic data. Probe vehicles also capture macroscopic data on traffic streams, such as speed, flow, and density. For microscopic data, researchers rely on video data and speed guns. With the improvements in image processing, video data can be used for various applications, including capturing properties of the stream and individual vehicles. However, the inability of video data to measure distances effectively is a significant drawback for different transportation applications that require micro-level, vehicle-by-vehicle trajectories. At the same time, because video cameras are stationary, they can only observe the traffic within their vicinity.

One way to address these limitations is to use a portable data acquisition system (PDAQS) assembled on a vehicle. Unlike roadside sensors, such a system provides flexibility in data collection because it can be moved as required. With the fast development of sensor technologies, most of the drawbacks of video data can be addressed by using portable sensor units. Light detection and ranging (LiDAR) sensors can obtain high-resolution traffic data in real time. LiDAR sensors can scan 360-degree, three-dimensional surrounding objects and collect the coordinates of each object in its scanned range at a high frequency with accuracy sufficient for (connected and automated vehicle (CAV) applications. Because LiDAR collects data at a high rate (typically in the range of 10 to 20 Hz), the coordinates can be processed to extract trajectories and estimate the speeds of vehicles and other objects.

This study was divided into two major activities: the development of hardware and the development of software. Each of these steps is discussed in the following chapters.

CHAPTER 2. DEVELOPMENT OF HARDWARE

2.1. PDAQS Sensors

The PDAQS was made up of various sensors. Although several sensors were available, a few were selected on the basis of previous experience and intended application. The critical sensors included LiDAR, video cameras, an on-board diagnostics (OBD) sensor, and a Global Positioning System (GPS) unit. Each of these sensors is briefly discussed in this section. These sensors were procured from with other sources and not with the PacTrans funding. However, the PacTrans funding was used to assemble them to make the PDAQS.

2.1.1. LiDAR

Because the purpose of the instrumentation was to capture the driving environment, the most critical sensor was LiDAR. LiDAR has increasingly been used in CAV applications. The sensor emits laser beams and measures the time it takes for the sensor to detect the reflected laser beams. This information is used to compute distances to objects. The main advantage of the LiDAR sensor is that it generates a 360-degree field of view (FOV) and reports the locations of objects within its range. UltraPuck, a commercially available LiDAR from a market-leading manufacturer, Velodyne, was procured. This sensor generates a 3-D picture of the surrounding environment by using 32 laser beams mounted in a compact housing. It provides positions of the objects relative to the LiDAR. The range of this LiDAR sensor in the horizontal plane is 200 meters, and the vertical field of view is 40 degrees. The frequency of data collection is 5 to 20 Hz.

2.1.2. Video Cameras

Although LiDAR captures the surroundings by using point clouds, the shape of the point cloud of an object may vary depending on the relative location of the point cloud from the LiDAR and the angle it makes with the LiDAR. This could pose severe challenges for efficient detection. Video cameras can help avoid some of these challenges. Also, as some other studies have shown, a fusion technique using both video and LiDAR images may help efficiently detect objects. We procured five Basler IP cameras for this project.

2.1.3. GPS

Because LiDAR and video cameras capture data from the surroundings, the data are all relative to the LiDAR. The exact location of the instrumented vehicle must be determined to estimate the actual positions of the captured objects. A GPS unit can help provide the location

information of the vehicle carrying the LiDAR. GPS coordinates can also be used to estimate the speed of the instrumented vehicle, which is vital in determining the speed of other vehicles within sight of the instrumented vehicle. We procured a Garmin 18xx GPS.

2.1.4. OBD Sensors

Onboard diagnostics (OBD) sensors capture various vehicle properties. In this case, they can be used to extract the speed of the instrumented vehicle. It is important to note that the speeds of various objects estimated with LiDAR data show the relative speed of the instrumented vehicle. Therefore, to calculate the actual speeds of the objects, the instrumented vehicle's speed must be accurately determined. GPS can also be used to determine speeds. However, as errors may be associated with the location data provided by the GPS, the estimates of speeds may amplify these errors. In addition, bad weather conditions, dense surroundings, and forested areas can further limit the use of GPS because in such conditions, satellites may fail to provide accurate location information. We procured a CSS CL2000 OBD sensor.

In addition, a laptop computer, a power source (a portable power bank), and a router were procured.

2.2. Assembling the Hardware

The procured hardware was assembled to create the PDAQS. The LiDAR was powered from the power bank and was directly connected to the laptop computer. The LiDAR interface enabled data to be directly stored in the laptop computer. The IP cameras were connected to the router, which was connected to the laptop computer. The router was powered by a power bank. The clocks of the video cameras were synchronized to the time on the laptop by using the camera interface. Hence, although the video data were captured on the SD card in the IP cameras, they had the same time stamp as the LiDAR data. GPS and OBD sensor units stored the data directly in the laptop computer. These steps ensured that data from various sources were synchronized. Figures 2.1 to 2.4 show the assembly.

Figure 2.1 Hardware assembly with LiDAR and video cameras

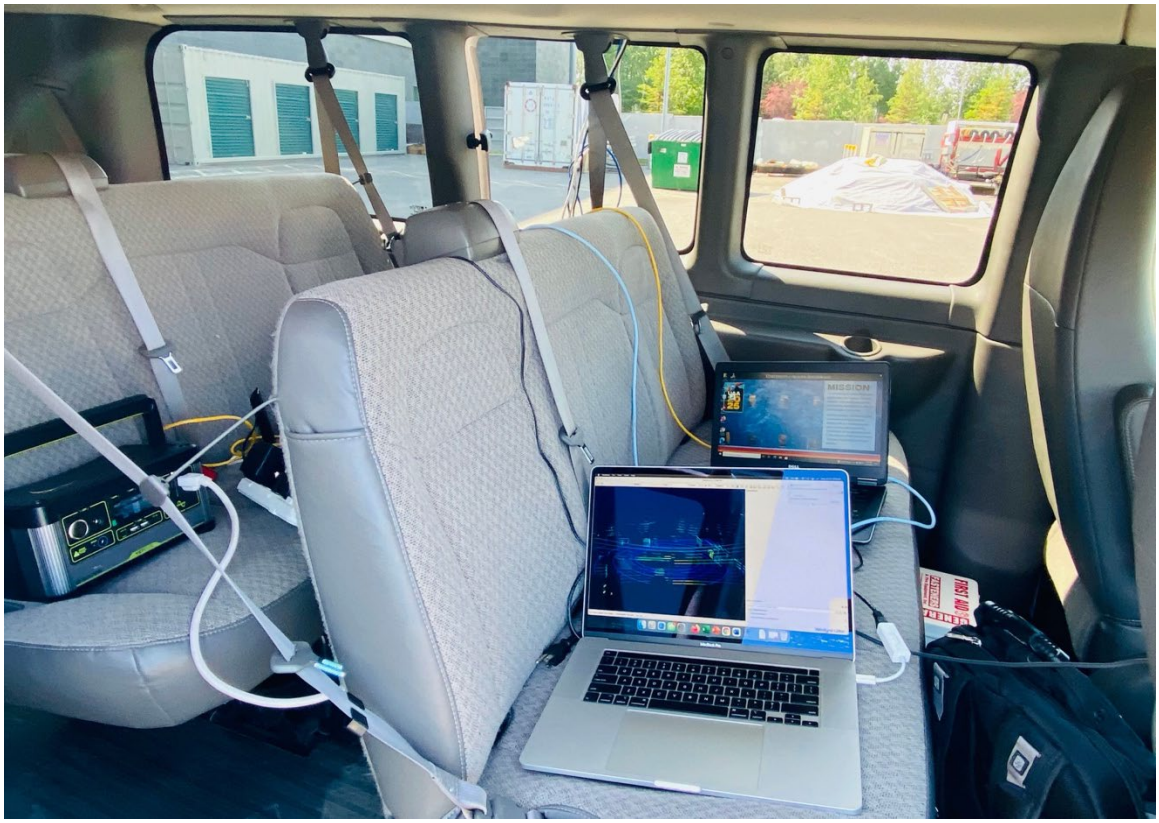


Figure 2.2 Hardware assembly inside the vehicle

Figure 2.3 Hardware assembly of the vehicle

Figure 2.4 The instrumented vehicle

CHAPTER 3. AN OVERVIEW OF THE SOFTWARE DEVELOPMENT

The goal of this task was to develop software programs to extract valuable data received from the PDAQS. This goal was divided into object detection and object tracking. Initially, we focused on the object detection task by using LiDAR data and frames from the video cameras. However, problems encountered with light-point clustering from the LiDAR, which is discussed below, produced a fruitless outcome. Therefore, the revised methodology is discussed in this chapter.

3.1. LiDAR Data

A LiDAR sensor can compute the distance of an object by emitting laser beams and measuring the time it takes for the sensor to detect the reflected laser beams. The horizontal range of our LiDAR sensor was 200 meters, and the vertical field of view was 40 degrees. The frequency of data collection was 5 to 20 Hz. LiDAR sensors have a 360-degree field of view that can generate a 3-D image of the surrounding environment. Each image contains light points in which point clouds are a function of the distance between the point clouds and the LiDAR sensor. These point clouds can be used to render a 3-D representation of an environment (Figure 3.1).

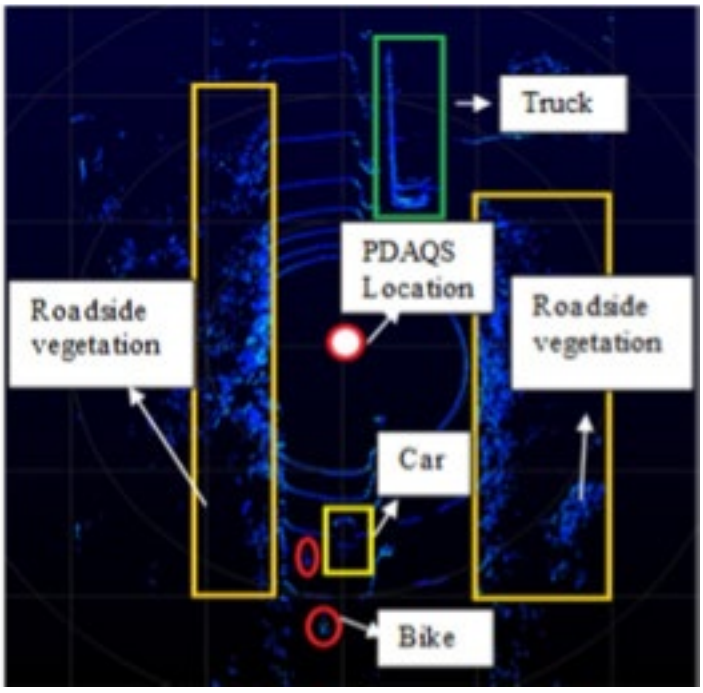


Figure 3.1 A sample LiDAR scan image

The LiDAR data were in a binary pcap file format. Their general structure is shown in Figure 3.2. This file format had to be read in and extracted with existing modules. Once read in, the structure needed to be parsed. We used the Python module scapy to accomplish this.

After the LiDAR data had been parsed, the light-points were clustered by using Density-Based Spatial Clustering of Applications with Noise (DBSCAN), a density-based clustering algorithm that groups points that are close to each other based on a distance measurement (usually Euclidean distance) and a minimum number of points [1]. A limitation of LiDAR processing and clustering is computational complexity; the runtime for a small sample of the data could be upwards of several hours.

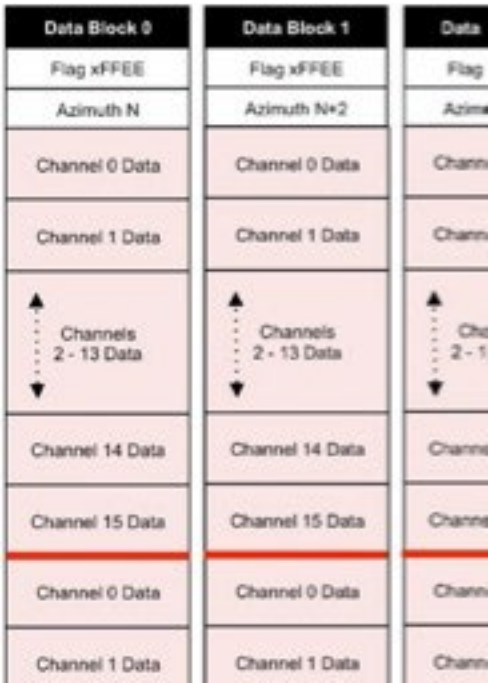


Figure 3.2 The general structure of *.pcap* files

3.2. Initial Object Detection Program

As discussed before, we started with using DBSCAN to detect objects. With the clusters formed with DBSCAN, we overlaid the clusters on top of a sample video. Figure 3.3 shows the final product. As can be seen, the clusters looked arbitrary and did not correlate with the objects in the video. Therefore, there was an error on the LiDAR side of the project. The obvious

diagnosis could have been that DBSCAN was not performing well for this task. But other potential sources of error could have been how the LiDAR data were being read or processed.

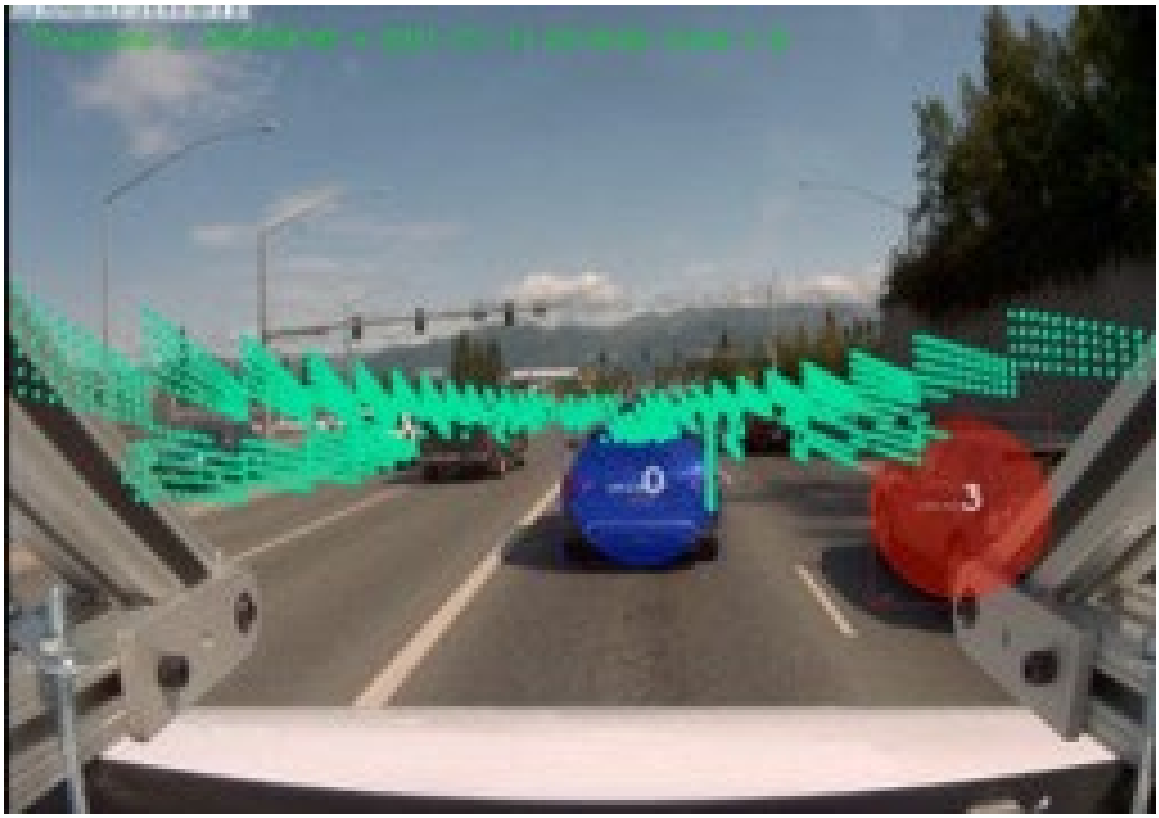


Figure 3.3 LiDAR clusters overlaid on video

3.3. The Revised Design

To address the issues in the preliminary program, we focused on the LiDAR side of the project. We rewrote the existing LiDAR framework. This step included exploring new modules to read and process LiDAR data and accurately detect LiDAR data. Another focus was reducing the runtime of processing and detecting LiDAR and video data. This approach indirectly resolved the exploration of processing modules and detection algorithms.

The revised program was built to be as modular as possible but with pieces that still fit well together. Such an approach provided options for efficient modification in the future. Figure 3.4 shows a high-level, partial diagram outlining the program's classes.

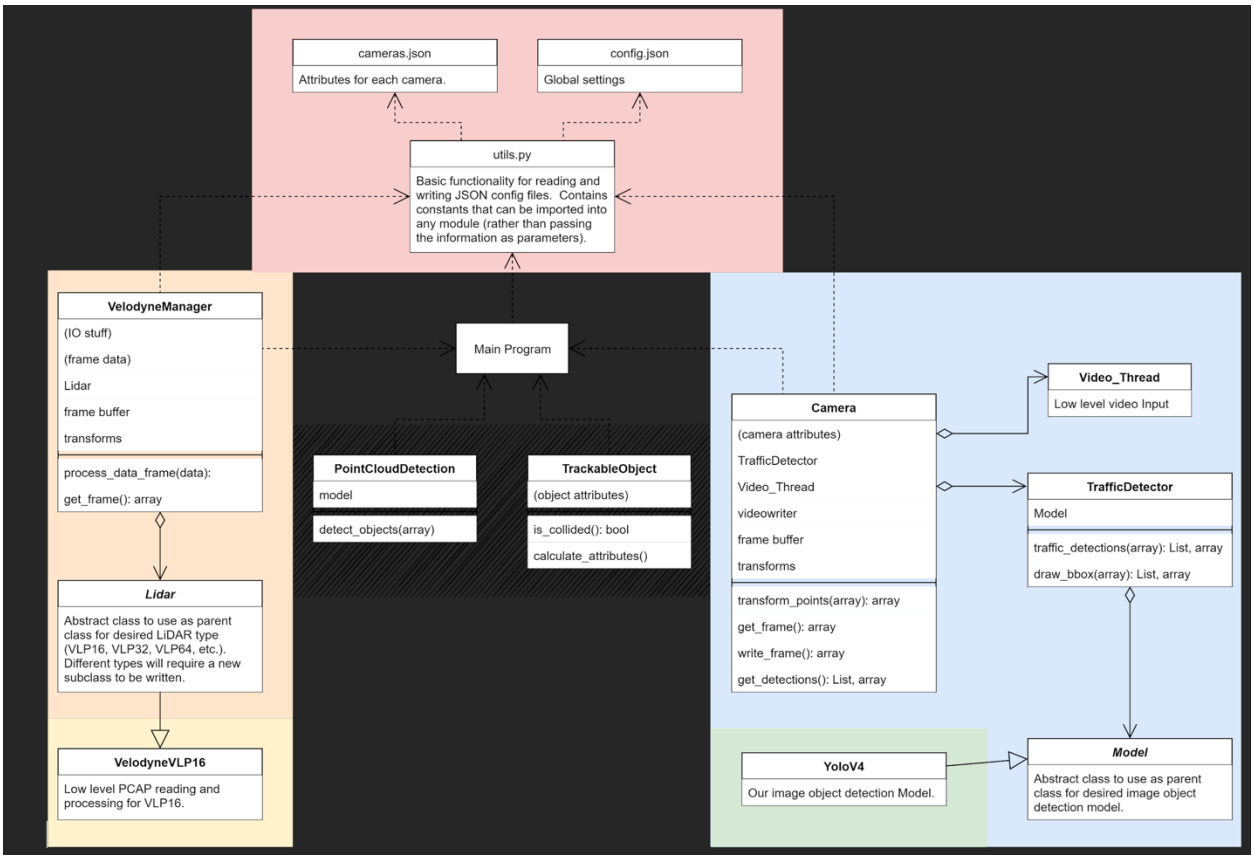


Figure 3.4 Class diagram of the revised program

3.4. Data Setup

LiDAR data are saved as a packet capture (PCAP) file. This is a stream of Transmission Control Protocol (TCP) packets. Camera data are saved as H264 video files. Each video file contains 60 seconds of content. Video files are named according to their start time; this is how the cameras saved files by default. The file names are used to determine timestamp data for video frames.

3.5. LiDAR Data

The LiDAR data are recorded as a PCAP file, a TCP packet stream. The LiDAR abstract class is used to create child classes that read specific types of Velodyne files—in our case VLP16. Other types include VLP32 and VLP64. Other LiDAR devices could be supported by writing new LiDAR class implementations. LiDAR data are processed as shown in Figure 3.5.

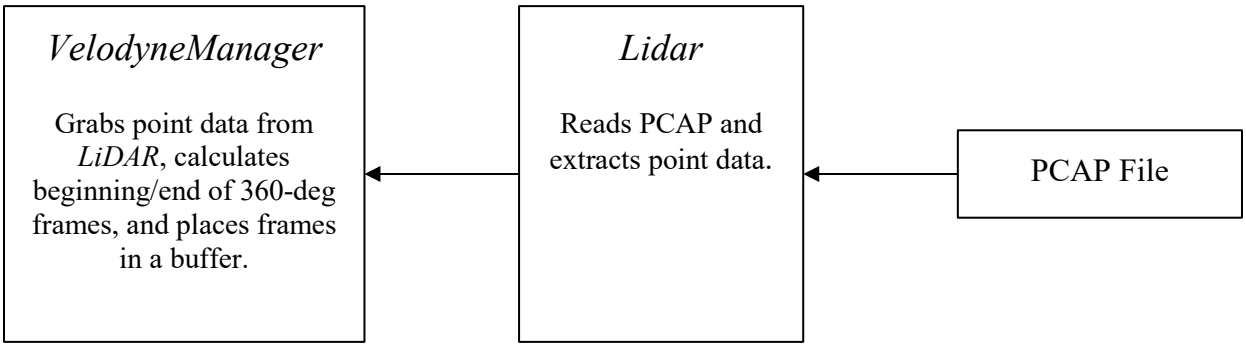


Figure 3.5 LiDAR data processing

Each point in a LiDAR frame includes the following attributes:

- ID: arbitrary number assigned to point
- Timestamp: time since epoch
- X position: Cartesian X coordinate*
- Y position: Cartesian Y coordinate*
- Z position: Cartesian Z coordinate*
- Distance: distance from device in meters
- Intensity: amount of light reflected back to device
- Latitude/omega: vertical angular offset
- Longitude/azimuth: horizontal angular offset.

*Cartesian coordinates are relative to the LiDAR device, with the origin (0, 0, 0) at the device. The X axis pointed out the sides of the vehicle, the Y axis pointed front and back, and the Z axis was up and down (altitude).

The VelodyneManager class takes a path to a PCAP file and fills its buffer with point cloud frames. LiDAR frames are placed onto the VelodyneManager framebuffer as a Tuple (frame, timestamp), with the frame being a 2-D array of points, and point data and the timestamp being the frame's timestamp.

3.6. Camera Data

Each instance of the Camera class corresponds to a particular camera. Camera attributes are stored in the cameras.json configuration file (see the Configuration section). The Camera class acts as a focal point for processing camera-related information.

3.6.1. Reading Video Files

Figure 3.6 shows the reading process. Each Camera class contains a Video_Thread object. This object is responsible for low-level video input. It read a single file and places video frames on its buffer. The Camera class then uses the Video_Thread frame buffer to fill its frame buffer. When the Video_Thread frame buffer is out of frames (i.e., end of file), the Camera class shuts it down and starts a new Video_Thread for the following video file, then continues the process. This process allows the Camera class to work uninterrupted on its frame buffer while transitioning from one video file to the next.

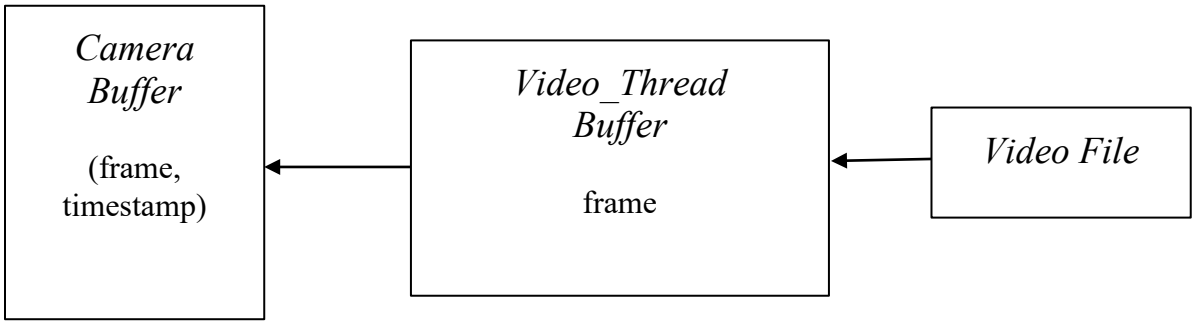


Figure 3.6 Reading video files

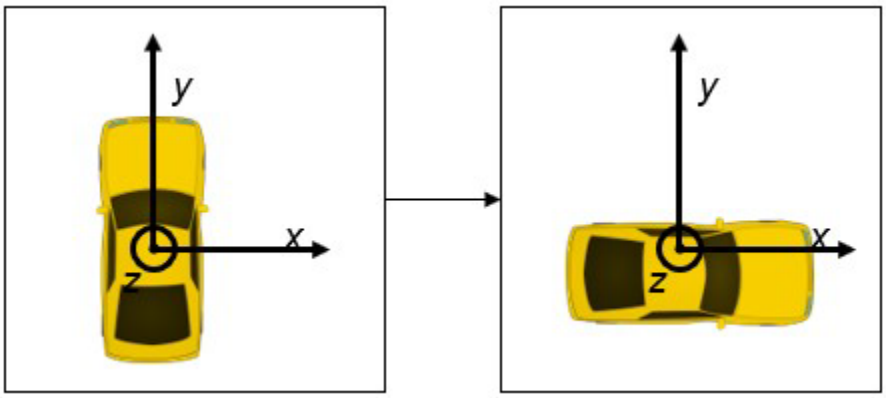
3.6.2. Retrieving a Frame

The get_frame(time) function in the Camera class takes a timestamp and returns the frame closest to that timestamp. It does this by popping frames from the buffer until the popped frame's timestamp is closer to the passing time than the next frame's timestamp. With this setup, the LiDAR frame's timestamp could be used in practice as a master timestamp to retrieve desired camera frames.

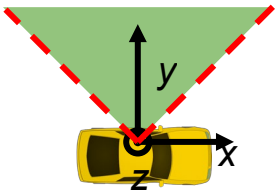
3.6.3. Transforming Points

To accurately overlay LiDAR points onto camera frames, the point coordinates must be changed from (x, y, z), where (0, 0, 0) is the LiDAR device, to (x, y), where (0, 0) is the top left pixel in the camera image. This step is done by using linear and non-linear transformations and without using Python loops (using numpy array calculations) to save time. The example images below show the transformation process for a camera pointing out the left side of the car:

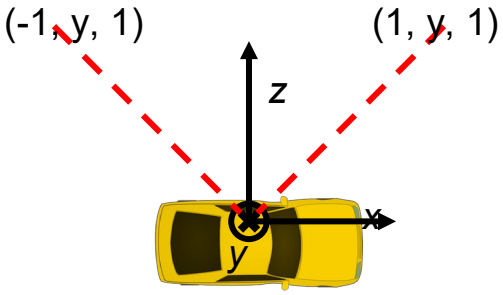
1. Linear transformation: Change the origin from the LiDAR device to the camera device with the camera at (0, 0, 0), the Y axis pointing in front of the camera, and the X axis pointing to the sides of the camera:



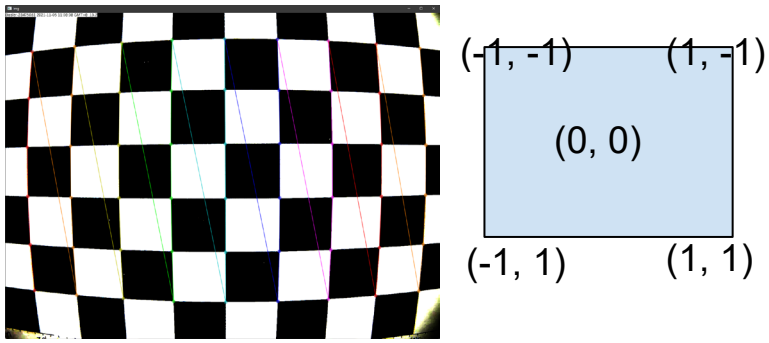
2. Point filter: Remove any points not in the camera's view. This reduces the number of points to make calculations with and prevents an overflow of floating point values when the coordinates are projected (overflow will occur for points close to the X axis but far from the Y axis):



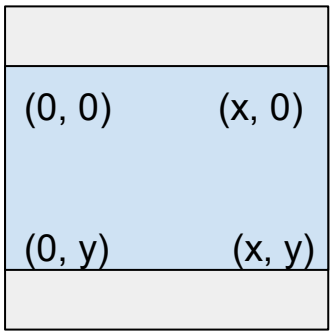
3. Linear transformation: Change the axes so the X,Y plane is in the point of view of the camera and normalize all point values to [-1, 1]. The Y axis is inverted because pixel coordinates along the Y axis are also inverted (get smaller as they go up):



4. Non-linear transformation: Project all points onto the far plane (at distance 1) and apply the distortion formula to account for camera lens distortion. At this point, the Z coordinate information (distance from the camera) is no longer needed:



5. Linear transformation: Normalize the points to the camera's resolution and set the origin (0, 0) to the upper-left corner:



3.6.4. Retrieving Detections

The Camera class uses its TrafficDetector member to get object detections within a frame. The TrafficDetector class is a wrapper class for the Model class, an abstract class used to create custom detection models.

The traffic_detections(frame) function takes a video frame (an image) and returns a video frame (for our purposes, the same frame with detection boxes drawn) and a list of objects detected in the following format:

[boxes, scores, classes, num]

where

- boxes: ndarray of shape(1, n, 4) is an array of bounding box coordinates
- scores: ndarray of shape(1, n) is an array of confidence scores
- classes: ndarray of shape(1, n) is an array of detected classes
- num: ndarray of shape(1,) has value n (*n is the number of detected objects)*

3.7. Object Detection Models

The framework developed contains two object detection tasks: 2-D using video and 3-D using LiDAR.

3.7.1. 2-D Image Object Detection

Our framework uses a scaled (256×256) YoloV4 TensorFlow Lite model because of its performance in real-time object detection [4][6]. TensorFlow Lite models are designed to be run on mobile devices, which makes them extremely resource friendly (in comparison to a typical TensorFlow Keras model). By sacrificing a small amount of accuracy, we achieved an increase in inference speed and a significant decrease in video random access memory (VRAM) usage: about 4 GB down to about 300 MB. This step allows us to maintain separate neural network models for each camera, which permits parallel inference while leaving plenty of VRAM for other uses (e.g., LiDAR detection models).

We designed our framework so that a developer can easily switch between different 2-D object detection models if the situation demands it. Other detection models can be implemented by creating new Model subclasses.

3.7.2. 3-D Image Object Detection

Getting the point cloud detection integrated into our framework was essential because having 3-D-detected objects would give us their centroid location, which was critical for object tracking. We integrated the Open3D-ML framework into our framework and implemented the PointPillars model. Getting Open3d-ML set up was a bit involved, requiring more steps than just a simple “pip install” command. More details about this are contained in the “Environment Challenges” section. Open3D-ML currently supports two models: PointPillars and PointRCNN. Unfortunately, a runtime error on the Open3D-ML side prevented us from using the PointRCNN model. Furthermore, Open3D-ML supports using TensorFlow and Pytorch for PointRCNN; we used the Pytorch implementation.

Open3D-ML also provides a visualization tool to see the predictions that the PointPillars model makes for a given point cloud frame. Our framework was designed so that the programmer can choose to enable this feature when making predictions with the point cloud data.

In terms of memory usage, the PointPillars model is relatively small; it takes up roughly 50 MB of memory. The programmer can use a central processing unit (CPU) or graphics

processing unit (GPU) for computation, although GPU runs faster. Because a point cloud frame has fewer data points than a video frame, and the PointPillars model has fewer parameters than the scaled YoloV4, inference with LiDAR is faster.

3.8. Global Configuration Files

The project utilized some global configurations that can be easily accessed and changed within the JSON configuration files found in the *config* directory. A *CONFIG* constant and other configuration tools can be imported from *tools.utils*. The JSON files store data in the same format as Python Lists and Dictionaries.

3.8.1. Config.json

This contains global settings as a Dictionary:

- *from*: (int) starting timestamp. The *VelodyneManager* class and *Camera* class use this to seek the desired starting location within their respective files.
- *To*: (int) ending timestamp. The classes mentioned above shut down operations when this timestamp is reached.
- *frame buffer size*: (int) The size in frames of the various frame buffers throughout the program.
- *classes*: (List: int) The desired classification indices from the *coco* names (e.g., 0 = car).
- *gps-port*: (int) Used by *VelodyneManager* to communicate with a GPS device (not used in this project).
- *data-port*: (int) Arbitrary ethernet port number used by *VelodyneManager* to read the PCAP TCP stream.
- *type*: (String) The type of visible light programming (VLP) device used. *VelodyneManager* uses this to create the correct *LiDAR* class.
- *gps*: (int) Boolean value indicating whether to process GPS data (not used in this project).
- *text*: (int) Boolean value. If set to 1, point data are exported to comma-separated value (CSV) files instead of a buffer.
- *ply*: (int) Boolean value. If set to 1, point data are exported to PointCloud files instead of a buffer.

- *max distance*: (int) Maximum desired point distance in meters. *VelodyneManager* deletes any points beyond this distance when creating frames. Set to negative to disable.
- *max points*: (int) Maximum number of points per frame. If the number of points in a frame exceeds this, *VelodyneManager* removes random points to decrease frame size. Set to negative to disable.
- *z-range*: (List: float) The range (altitude) of points to add to a frame in meters. *VelodyneManager* removes anything outside of this range (points that are too low or too high).

3.8.2. Cameras.json

Specific camera attributes are stored here and can be imported by a user program to create *Camera* objects. Camera attributes are stored as a List of Dictionaries, with each Dictionary corresponding to a different camera:

- *name*: (String) A plain text, human readable identifier for the camera.
- *azimuth*: (float) Horizontal rotation of the camera in degrees (e.g., 90 degrees faces to the right of the vehicle).
- *fov*: (float) The camera's field of view angle in degrees.
- *offset*: (List: float) The camera's XYZ position in relation to the LiDAR device in meters.
- *fps*: (int) Frames per second. Camera's frame rate.
- *time_offset*: (float) Timing difference in seconds between the camera and the LiDAR device.
- *dist_coeff*: (List: float) The distortion coefficients for the camera.
- *omega*: (float) The vertical rotation of the camera in degrees (up and down).

CHAPTER 4. RESULTS

One of the major outcomes of this project was a functional hardware setup for a PDAQS. For the software, there were a couple of results. The first was an alignment tool that users can use to read video and LiDAR data and correctly overlay LiDAR points on top of detected objects. The second was a LiDAR point cloud frame with detected 3-D objects from the PointPillars model using Open3D-ML.

4.1. Alignment Tool

This tool was designed to streamline the manual calibration of raw data. With it developers can adjust and test point transformation parameters and LiDAR/camera timing synchronization in real time and save their changes to the camera's configuration. Figure 4.1 shows a screenshot of the output of the alignment tool. It contains a video frame, with each detected object having a bounding box, object type, and confidence value. For example, the model was 96 percent confident that the object within the blue box was a truck. Each object could contain LiDAR points, depending on how far the object was from the LiDAR sensor.

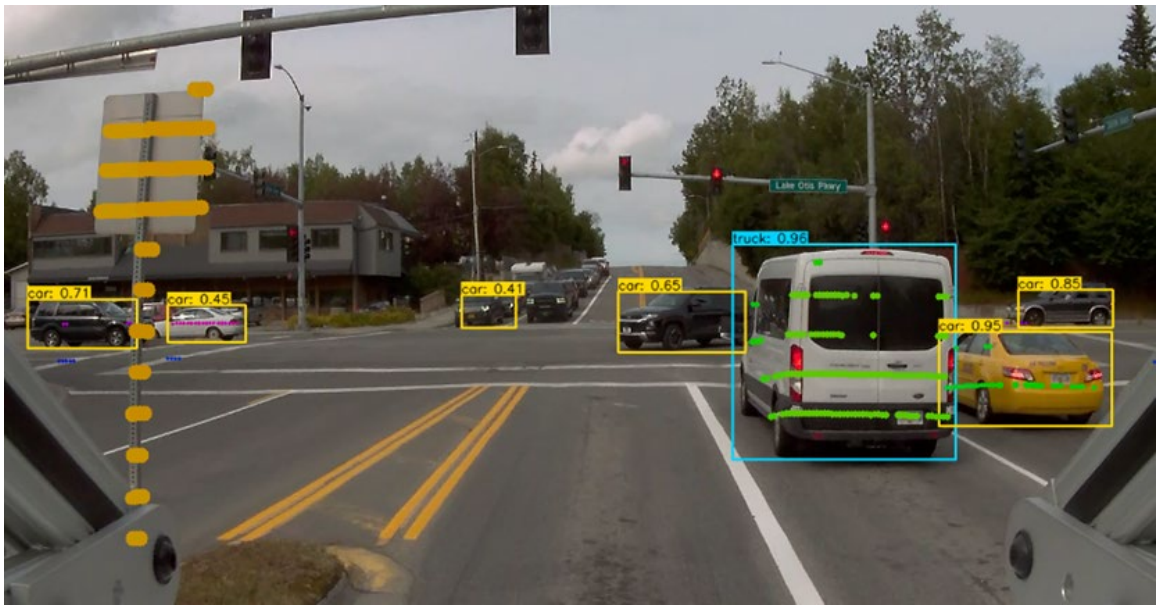


Figure 4.1 Alignment tool showing point transformations and object detections

4.2. Point Cloud Detection

The points pillar model read in a point cloud frame in which each point contains the x, y, and z coordinates and an intensity value measured by the LiDAR sensor. The output contains 3-

D detected objects with grey bounding boxes and the object type (pedestrian, car, etc.). Figure 4.2 illustrates the output.

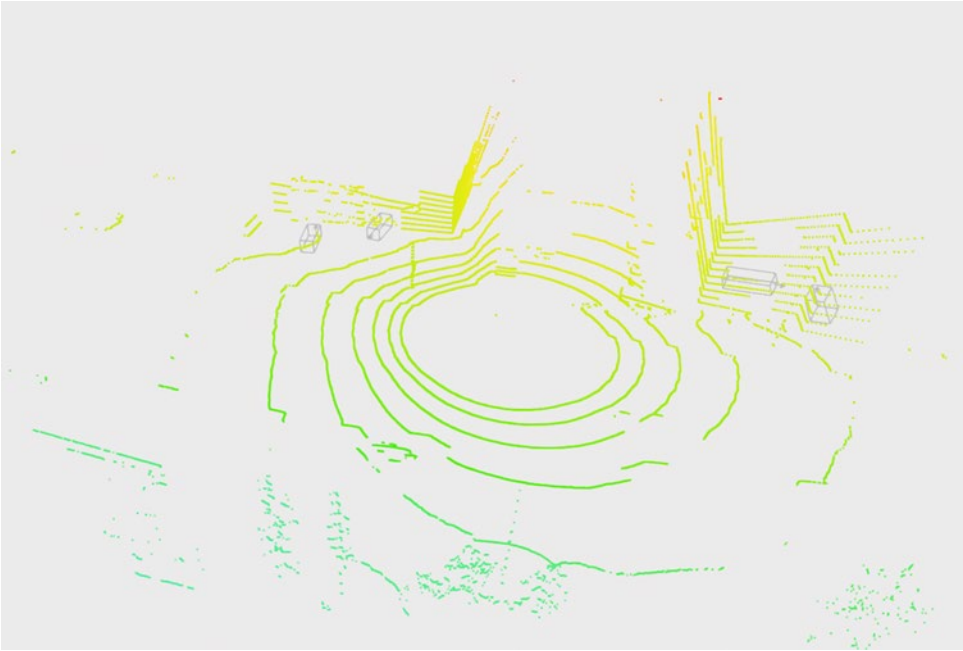


Figure 4.2 PointPillar object detection

Although these two results were excellent, these alone cannot be used to track objects efficiently. We need to put more efforts into this aspect. The tools developed here can be used for this purpose.

CHAPTER 5. SUMMARY AND CONCLUSIONS

5.1. Summary

We successfully assembled the hardware for a PDAQS. The PDAQS can be operated with minimal training. However, we faced a lot of issues with the software development. Object detection and tracking were challenging problems. Our VelodyneManager and Camera modules correctly decode the PCAP file and extract the point data to be used by the Camera to transform the points onto the camera video stream. The process successfully produces our intended output, with the LiDAR points correctly overlaid on top of detected objects. Our main product is a distortion tool that allows users to jump around different timestamps within the data and see detected objects with their corresponding LiDAR points overlaid. Furthermore, users can manually change the distortion coefficients to reduce the fisheye effect from the Camera.

We also made tremendous progress in the usability of point cloud detection models. Although the performance was not the best, it is a good starting point for further improvement. The use of point cloud detection was essential for this project, as we needws to track objects, bringing us closer to our goal.

5.2. Scope for Future Work

The project currently needs to meet its requirements. Logic needs to be implemented to tie video and LiDAR detections together to obtain valid trackable objects within each frame and to track these objects from one frame to the next. We laid out ideas for doing this algorithmically, but it may also be possible with machine learning models.

The current detection methods work well. However, they could likely be improved upon. For example, we originally wanted to use a frustum-based method for our 3-D object detection, but we needed help finding adequately trained models to integrate into the project. The KITTI dataset could be used to develop and train a custom model.

Also needed is a graphical user interface that is intuitive enough to be used by end users. Currently, the project consists of logically separated classes and modules waiting to be tied together by a main program. The runnable tools created are executed via the command line, and robust software/programming knowledge is needed to run them properly.

CHAPTER 6. REFERENCES

1) Github, 2020a, Open 3D ML. < https://github.com/isl-org/Open3D-ML>. Accessed May 13, 2020.
2) Github, 2020b, YOLO V4 in TensorFlow. < https://github.com/hunglc007/tensorflow-yolov4-tflite>. Accessed July 26, 2020.
3) Github, 2020c, Veloparser. < https://github.com/ArashJavan/veloparser>. Accessed November 16, 2020.
4) Github, 2021, YOLO4 for Windows and Linux. < https://github.com/AlexeyAB/darknet>. Accessed January 12, 2021.

5) Karlsruhe Institute of Technology, 2017, 3D Object Detection Evaluation 2017. < https://www.cv-libs.net/datasets/kitti/eval_object.php?obj_benchmark=3d>. Accessed July 14, 2020.

6) OpenCV, 2020, Camera Calibration. < https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.ht-ml>. Accessed November 25, 2020.

7) Towards Data Science, 2020, How DBSCAN works and why should we use it? < https://towardsdata-science.com/how-dbscan-works-and-why-should-i-use-it-443b4a191c80>. Accessed June 20, 2020.

APPENDIX A

Collecting/Processing Raw Data

The raw data are collected separately from all devices. There is no communication between devices during the collection process and no collection tools used to synchronize the streams, which means the data will need more processing before use.

Timing

Camera and LiDAR clocks should be set before collecting data. Setting the hardware clocks will synchronize the streams to within a few seconds (about a 9 second offset for our test data), which makes manual timing alignment much easier. If possible, synchronize clocks with an online service (e.g., NIST). The cameras used will save a series of 60 second video files with filenames in the format YYYY-MM-DD_hh-mm-ss_sss.0.h264, which corresponds to the start time of the video recording. The Camera object will use this filename as the video's base timestamp. For LiDAR data, timestamp information is included within the actual data.

NOTE: Because there is only one LiDAR device to many cameras, it is assumed that the LiDAR frame timestamps will be used as "master" timestamps. The Camera object's get_frame(time) function has been written with this assumption in mind.

Camera Distortion

The cameras use a variable zoom lens. If the zoom is changed, the cameras may need to be recalibrated to get the new field of view (FOV) and distortion coefficients. A calibration board was designed for this purpose. For FOV, use the inverse tangent function with the mm markings on the board and the board's distance from the camera. For distortion coefficients, take an image of the chessboard and use tools/video/distortion_tool.py to obtain the distortion coefficients.

Alignment Tool (alignment_tool.py)

This tool allows a developer to verify LiDAR/camera transformations and timing and to make real-time adjustments to camera settings. The constants at the top of the Python program allow users to select the desired camera settings and paths to video and PCAP files. It is not a robust tool and will crash if the following requirements are not met:

- The cameras.json file exists.
- The camera index exists.
- The PCAP path exists.

- The video path exists.

Note: these requirements are not exhaustive.

Upon running the tool, the data are read in, and three windows open: A bird's eye view of the LiDAR data, the camera frame with overlaid LiDAR points and optional detected objects, and a user interface where settings can be changed and updated.

The User Interface

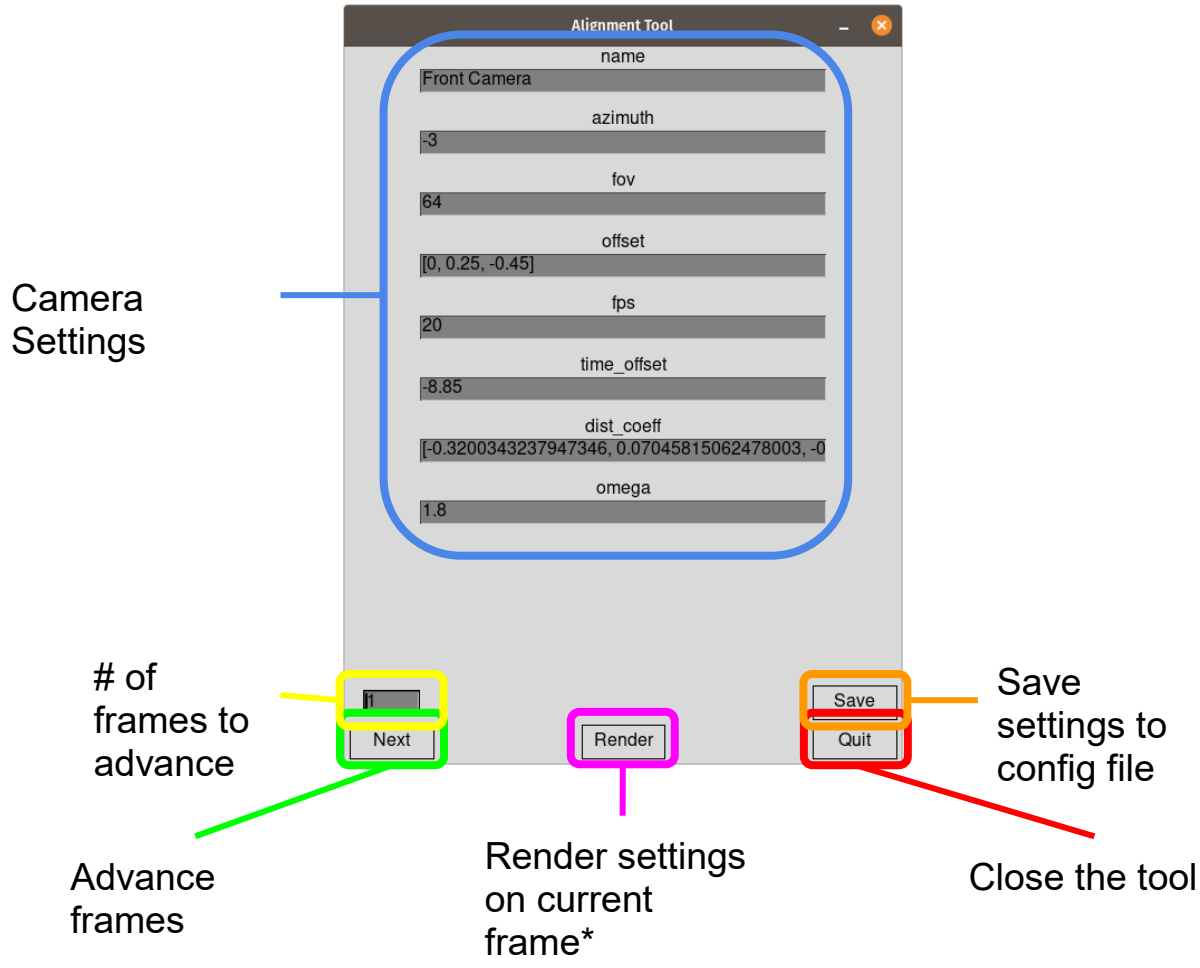


Figure A.1: The user interface for alignment

Distortion Tool (tools/video/distortion_tool.py)

This Python script uses OpenCV and an image of a chessboard to determine distortion coefficients of the camera. These coefficients are used when LiDAR points are transformed to

overlay on the camera image. The constants at the top of the Python program allow users to specify the image taken with the camera and the dimensions of the board. It is also not a robust tool. It may be necessary to increase the contrast and/or levels of the image so the tool can properly find the corners.