Tracking Traffic With Machine Vision

CSCE A470 Final Report

James Flemings and Jon Rippe

12/13/2021

Table of Contents

Abstract

We are developing a Portable Data Acquisition System (PDAQS) to study road user behavior. A previous group had started working on this project, but were unable to complete the project's goals. Our work addresses the issues they encountered as well as made fruitful progress towards the creation of a robust PDAQS. Specifically, we developed a modularized framework that is capable of efficiently processing Light Detection and Ranging (LiDAR) and video data and detecting objects in 2-D and 3-D. Our main results are an alignment tool that adjusts and tests point transformation parameters and LiDAR/camera timing synchronization in real time, and the integration of point cloud detection into our framework. Though lots of progress has been made, there is plenty of future work that needs to be done.

1. Introduction

The clients for this project are Dr. Shawn Butler and Dr. Vinod Vasudevan.They proposed developing a Portable Data Acquisition System (PDAQS) to study road user behavior. To accomplish this, they split the development of the PDAQS into two steps: setting up the hardware and developing software programs to extract useful data. The hardware setup consists of an instrumented vehicle with sensors such as LiDAR, four video cameras, a GPS, and Onboard Diagnostics (OBD) sensors attached to it. This step is completed and is being used to generate data.

The goal of this project is to develop software programs to extract valuable data received from the PDAQS. This goal can be divided into two tasks: objection detection and object tracking. Previous project members focused on the objection detection task by using both LiDAR data and frames from the video cameras. However, problems encountered with light-points clustering from LiDAR, which will be further discussed below, produced a fruitless outcome. Our work hopes to address these problems.

1.1 LiDAR Data

A LiDAR sensor can compute the distance of an object by emitting laser beams and measuring the time it takes for the reflected laser beams to be detected by the sensor. The horizontal range of our LiDAR sensor is 200 meters and the vertical field of view is 40 degrees. The frequency of data collection is 5 to 20 Hz. LiDAR sensors have a full 360 degree field of view that can generate a 3-D image of the surrounding environment. Each image contains light-points where point clouds are a function of the distance between it and the LiDAR sensor. These point cloud can be used to render a 3D representation of an environment (Figure 2.1).

The LiDAR data is in a binary pcap file format. Its general structure is shown in Figure 2.2. This file format must be read in and extracted using existing modules. Once read in, the structure needed to be parsed. Jack Sexauer used the Python module scapy to accomplish this.
After parsing the LiDAR data, the light-points are clustered using Density-Based Spatial Clustering of Applications with Noise (DBSCAN), which is a density-based clustering algorithm

that groups together 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 the LiDAR processing and clustering is computational complexity; the runtime for a small sample of the data is upwards to several hours.

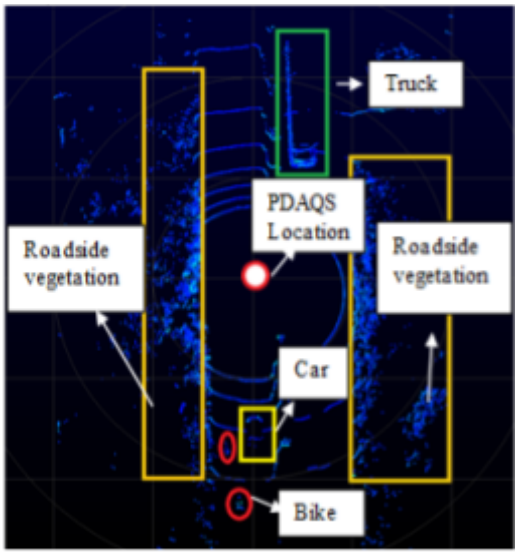


Figure 2.1: A sample LiDAR scan image.

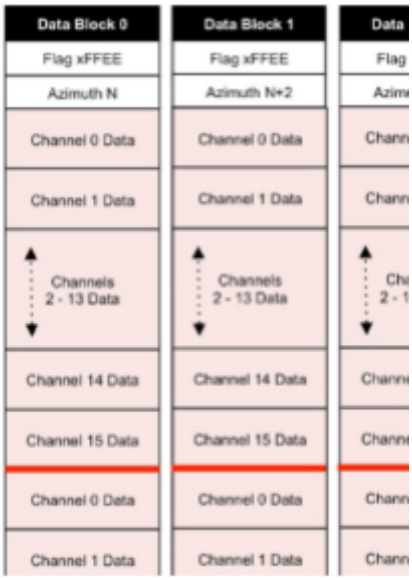


Figure 2.2: The General Structure of pcap files.

1.2 Object detection

With the clusters formed from DBSCAN, Ty Bergstrom, who previously worked on this project, overlaid the clusters on top of a sample of video. Figure 2.3 contains the final product.

As you can see, the clusters look arbitrary and don't correlate with the objects in the video. Thus, there is an error on the LiDAR side of the project. Naturally, the obvious diagnostic could be that DBSCAN is not performing well for this task. But other potential sources of error could be how the LiDAR data is being read in or processed.

Figure 2.3: LiDAR clusters overlaid on video.

2. Requirements

The end product of this project can be summarized from the original project proposal:

- “The main objective of this proposal is to develop a portable data acquisition system (PDAQS) that can capture trajectories of various road users with high accuracy.”
- “The primary end product will be an efficient PDAQS system capable of collecting detailed vehicle trajectories.”

To meet these requirements, we will be focusing on the LiDAR side of the project. Hence, we hope by the end of this project the LiDAR clusters overlaid on top of a video stream will be correctly correlated with the objects in the video. Consequently, we will need to debug and or rewrite the existing LiDAR framework in our project. This includes exploring new modules to read and process LiDAR data, and accurately detecting LiDAR data. Another focus will be to reduce the runtime of processing and detection of LiDAR and video data. This can be indirectly resolved from the exploration of processing modules and detection algorithms. We could also explore optimizations that utilize a GPU for processing and detection.

3. Design

The program was built to be as modular as possible, but with pieces that still fit well together. This is so future teams can modify aspects of the programs as they need without having to navigate spaghetti code or surgically remove functions and classes. Figure 3.1 shows a high-level, partial diagram outlining the classes the program uses.

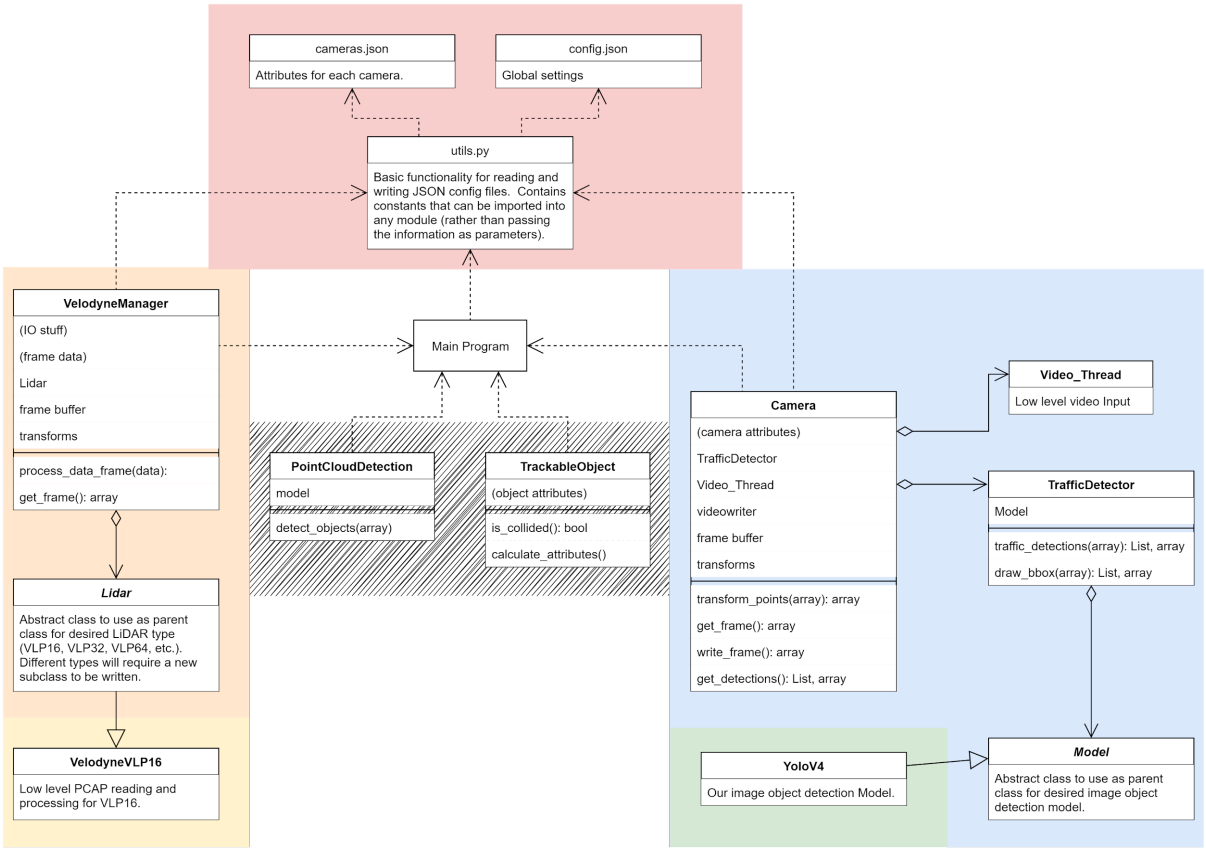


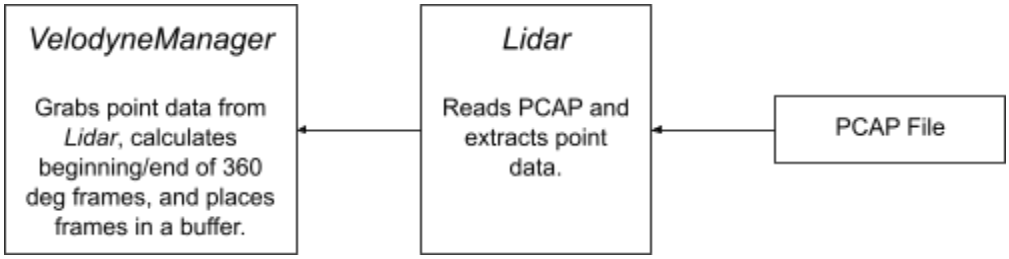
Figure 3.1: Class Diagram.

3.1 Data

LiDAR data is saved as a PCAP file. It is a stream of TCP packets. Camera data is 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 save files by default. The file names are used to determine timestamp data for video frames.

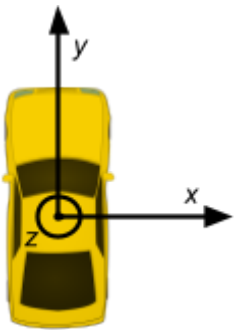
3.2 LiDAR

The LiDAR data is recorded as a PCAP file, which is a stream of TCP packets. 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 can be supported by writing new *Lidar* class implementations. LiDAR data is processed as follows:



Each point in a LiDAR frame includes the following attributes:

0. ID: arbitrary number assigned to point
1. Timestamp: time since epoch
2. X position: cartesian X coordinate*
3. Y position: cartesian Y coordinate*
4. Z position: cartesian Z coordinate*
5. Distance: distance from device in meters
6. Intensity: amount of light reflected back to device
7. Latitude/omega: vertical angular offset
8. 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 points out the sides of the vehicle, Y axis points front and back, Z axis is up and down (altitude).

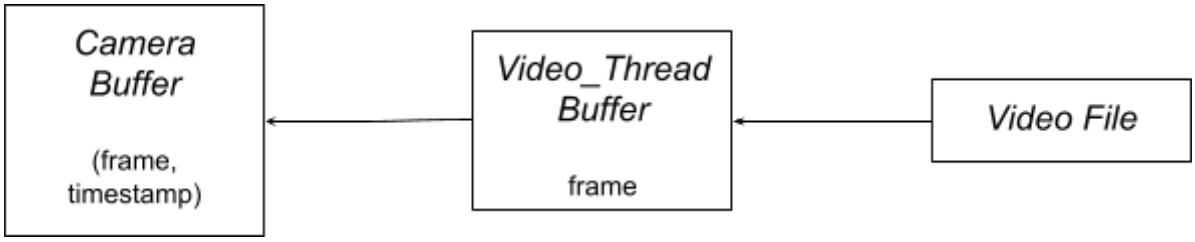
The *VelodyneManager* class takes a path to a PCAP file and fills its buffer with point cloud frames. LiDAR frames are placed onto the *VelodyneManager* framebuffer as a Tuple (frame, timestamp), with the frame being a 2D array of points and point data and the timestamp being the frame's timestamp.

3.3 Cameras

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

3.3.1 Reading Video Files

Each *Camera* class contains a *Video_Thread* object. This object is responsible for low level video input. It reads a single video file and places video frames on its own frame buffer. The *Camera* class then uses the *Video_Thread* frame buffer to fill its own 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 next video file, then continues the process. This allows the *Camera* class to work uninterrupted on its own frame buffer while transitioning from one video file to the next.



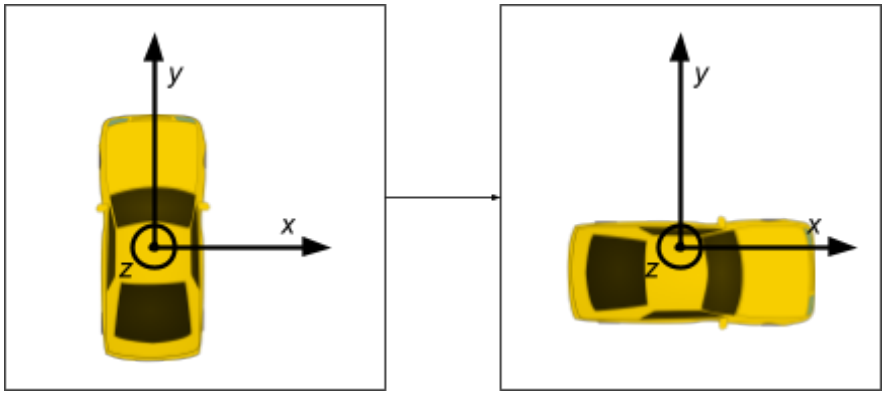
3.3.2 Retrieving a Frame

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 passed in time than the next frame's timestamp. With this setup, the LiDAR frame's timestamp can be used in practice as a master timestamp to retrieve desired camera frames.

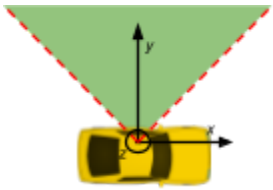
3.3.3 Transforming Points

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 is done 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:

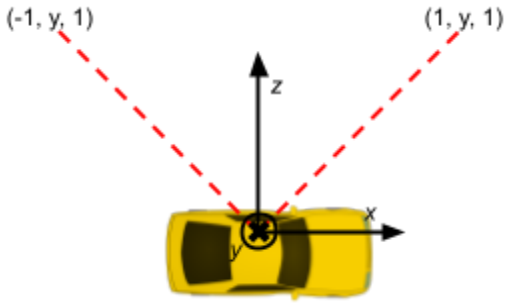
1. Linear transformation: change the origin from LiDAR device to camera device with the camera at (0, 0, 0), Y axis pointing in front of the camera and X axis pointing to the sides of the camera:



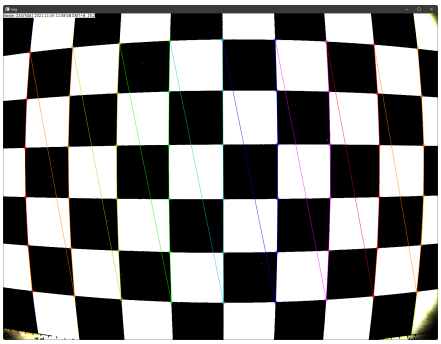
2. Point filter: remove any points not in the camera's view. This reduces the number of points to make calculations on and prevents overflow of floating point values when the coordinates are projected (overflow will occur for points close to the X axis but far from the Y axis):

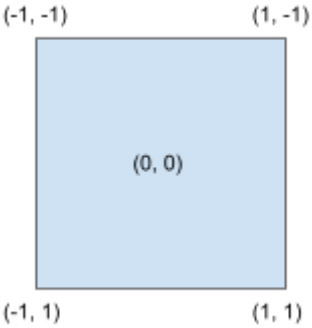


3. Linear transformation: Change the axes so the XY plane is in the point of view of the camera and normalize all point values to [-1, 1]. The Y axis is inverted because pixel coordinates along the Y axis are also inverted (get smaller as you go up):

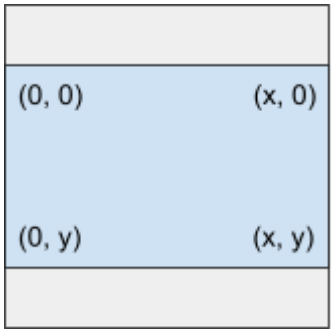


4. Non-linear transformation: Project all points onto the far plane (at distance 1) and apply the distortion formula to account for camera lens distortion. At this point, the Z coordinate information (distance from the camera) is no longer needed:





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



3.3.4 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, which in turn is 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 coords
- 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)*

3.4 Object Detection Models

Our framework contains two object detection tasks: 2D using video and 3D using LiDAR.

3.4.1 2D Image Object Detection

Currently, 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 (compared to a typical TensorFlow Keras model). By sacrificing a small amount of accuracy, we get an increase in inference speed and a significant decrease in VRAM usage: about 4GB down to about 300MB. This allows us to maintain separate neural network models for each camera, which permits parallel inference while leaving plenty of VRAM left for other uses (e.g., LiDAR detection models).

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

The previous group used MobileNet, a lightweight, small neural network that has decent performance. The rationale for selecting this model was because of its low memory usage. However, its performance isn't optimal for our needs. For example, in Figure 3, there are vehicles in clear view of the camera, yet the model doesn't detect it.

3.4.1 3D Detection

Getting point cloud detection integrated into our framework is a very important requirement because having 3D detected objects gives us their centroid location, which is critical for object tracking. Currently, we integrated the Open3D-ML framework into our framework and have the

PointPillars model implemented. Currently, getting Open3d-ML set up is 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, there is a runtime error on the Open3D-ML side that prevents us from using the PointRCNN model. Furthermore, Open3D-ML supports the use of TensorFlow and Pytorch for PointRCNN; currently, we are using the Pytorch implementation.

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

In terms of memory usage, the PointPillars model isn’t relatively large; it takes up roughly 50 MB of memory. The programmer can choose to use a CPU or GPU for computation, though GPU runs faster. Since a point cloud frame has less data points than a video frame, and the PointPillars model has less parameters than the scaled YoloV4, inference on LiDAR is faster.

3.5 Global Configuration Files

The project utilizes 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* (you’ll see this done at the beginning of every file). The JSON files store data in the same format as Python Lists and Dictionaries.

3.5.1 config.json

Contains global settings as a Dictionary:

- *from*: (int) starting timestamp. The *VelodyneManager* class and *Camera* class will use this to seek to the desired starting location within their respective files.
- *to*: (int) ending timestamp. The classes mentioned above will shutdown 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 VLP device used. *VelodyneManager* will use 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 is exported to CSV files instead of buffer.
- *ply*: (int) boolean value. If set to 1, point data is exported to PointCloud files instead of buffer.
- *max distance*: (int) maximum desired point distance in meters. *VelodyneManager* will delete 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* will remove 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* will remove anything outside of this range (points that are too low or too high).

3.5.2 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 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).

4. Software Development Process

For this project, we were given high level requirements for the development of this project. Thus, we had lots of discretion in choosing the implementation to fulfill those requirements. We roughly followed the agile method, having weekly development sprints followed by updates with project implementation goals during Friday meetings.

4.1 Testing and Debugging

Since most of our deliverable software isn't the final product for this project, and due to time constraints, we didn't spend too much time thoroughly testing the framework other than running short test programs to ensure each module is operating as intended. However, as we have stated above, the framework is modularable, so it would be very easy to run unit tests as well as integration tests for future teams that want to use these modules in their final product.

4.2 Environment Challenges

It was challenging to find an existing point cloud detection framework to work with our data. Starting from scratch and training a 3D object detection model ourselves is infeasible because this requires countless hours of labeling data and deep learning accelerated hardware for training. So we decided to scour the internet in hopes of finding a 3D detection framework that

we could integrate into our own framework. There are two main issues with existing 3D detection frameworks: documentation and tight integration with KITTI. The first issue is that many of the frameworks are undocumented because they're created by researchers. The main purpose those frameworks serve is to achieve state-of-the-art performance, which will then be used for a publication. The researchers don't intend for it to be used by other people other than to reproduce their results. Thus, it's difficult for other users to figure out how to use the framework for their own needs.

The second main issue is that the frameworks are tightly integrated with KITTI. KITTI is a 3D vision benchmark suite for programmers to design models for 3D tasks, such as object detection and semantic segmentation, on the KITTI dataset. This dataset contains many requirements that our dataset just doesn't meet [3]. Thus, we are unable to use these models on our data.

Fortunately, we were able to find and successfully use Open3D-ML in our framework. Open3D-ML contains a healthy amount of documentation and doesn't have strict requirements for the KITTI dataset. We did encounter some issues with getting an environment set up; specifically, Open3D-ML only supports certain versions of Python as well as Pytorch and Cuda [2]. Managing the installation of the correct versions of these libraries was a challenge.

5. Results

There are two demo-able results from the project. One is an alignment tool that can be used by users to read in video and LiDAR data and correctly overlay LiDAR points on top of detected objects. The second is a LiDAR point cloud frame with detected 3-D objects from the PointPillars model using Open3d-ML.

5.1 Alignment Tool

This tool was designed as a way to streamline manual calibration of the raw data. It allows a developer to adjust and test point transformation parameters and LiDAR/camera timing synchronization in real time and save their changes to the camera's configuration. The following image contains 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 is 96% confident that the object within the blue box is a truck. Each object could possibly contain LiDAR points, depending on how far the object is from the LiDAR sensor.

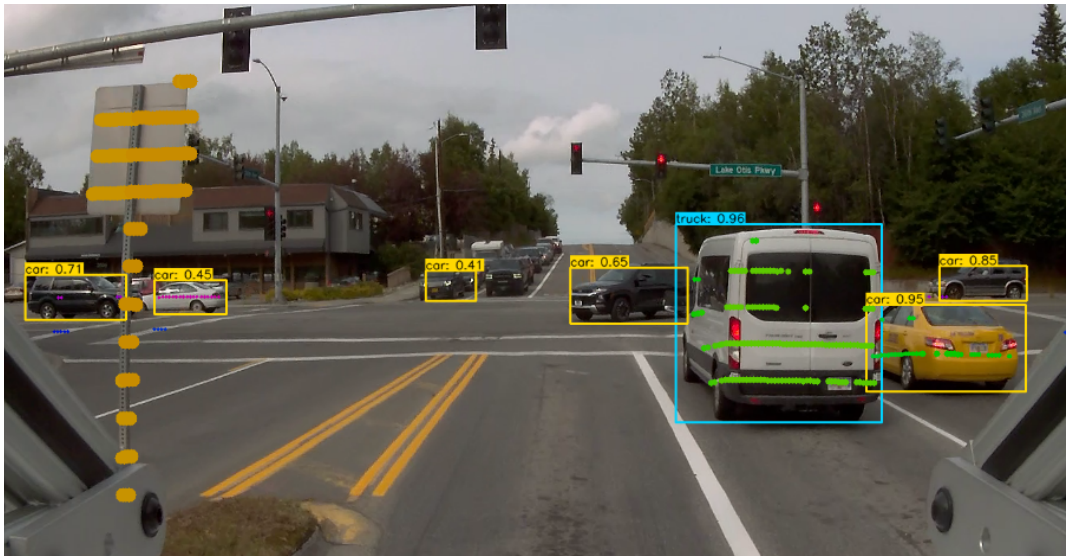


Figure 5.1: Alignment tool showing point transformations and object detections.

5.2 Point Cloud Detection

The points pillar model reads in a point cloud frame where each point contains the x, y, z coordinates and an intensity value measured by the LiDAR sensor. The output contains 3-D detected objects with grey bounding boxes as well as the object type (pedestrian, car, etc.). Figure 5.2 illustrates the output.

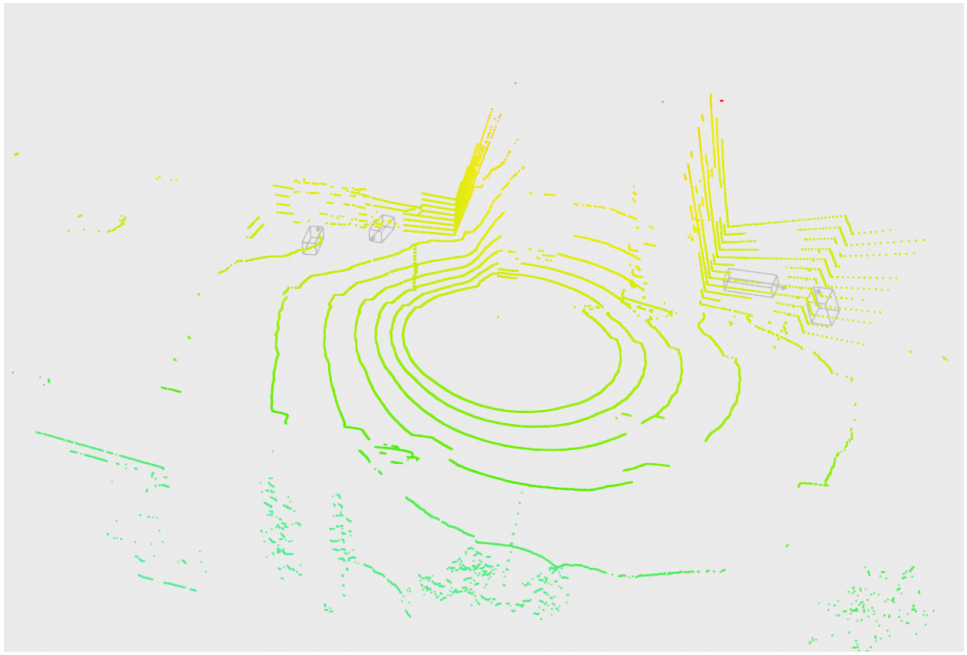


Figure 5.2: PointPillar object detection.

6. Code Inspection Considerations

There are some issues that were logged during our code inspection, and here are the actions that were taken:

- Initialized declared variables.
- Included function type hints.
- Removed ‘params’ argument (initially to pass configurations to objects).
    - Realized it was unnecessary
    - Motivation behind using global configuration settings for better modularity
- Changed some variable names.
- Tons of documentation.
- Removed some magic numbers.
    - They disappeared with the creation of the Model abstract class and YoloV4 subclass.

7. Summary and Conclusion

For this project, we resolved many of the issues that the previous group encountered for this project. Our *VelodyneManager* and *Camera* modules properly decodes the PCAP file and extracts 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 the user to jump around different timestamps within the data, and see detected objects with their corresponding LiDAR points overlaid on top. Furthermore, the user can manually change the distortion coefficients to reduce the fish eye effect from the camera.

We also made tremendous progress in the usability of point cloud detection models. Though the performance isn’t the best, it’s a good starting point for a future team to use and further improve upon. It can also save the future group time from problems that we encountered since we documented them. The use of point cloud detection is essential for this project as we need to track objects, so this brings us closer to our goal.

Overall, we found this project challenging, but we enjoyed the work. We learned a lot about many topics from different fields to help us with our project. This included object detection from machine learning, transformations from linear algebra, multithreading and optimization, and many more. The skills that we gained from this project will be useful for us in our careers.

8. Future Work

The project currently does *not* 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 on doing this algorithmically,

but it may also be possible to do with machine learning models. We'll leave that decision to the next team.

Our detection methods could likely be improved upon. For example, we originally wanted to use a frustum-based method for our 3D object detection, but we were unable to find adequate pretrained models to integrate into the project. Perhaps the KITTI dataset could be used to develop and train a custom model.

There ought to be a better (or standardized) process for data collection. As of right now, the LiDAR sensor and the video camera are booted asynchronously, which causes misalignment between both data sets. Because of this, we needed to engineer a hacky solution to work around this. This is further exacerbated by prematurely starting up the instruments, which generates a plethora of non useful data. A more robust solution would be to remove this discrepancy in the first place by creating a system that synchronizes the startup of both instruments, and also only start the instruments once the vehicle is being operated.

A graphical user interface that is intuitive enough to be used by civil engineers will eventually be needed. As of right now, the project mostly consists of logically separated classes and modules that are waiting to be tied together by a main program. The runnable tools we've created are executed via the command line and strong software/programming knowledge is needed to properly run them.

The project has important hardware requirements. We were lucky enough to be developing in similar environments with similar enough GPU hardware. Ideally a dedicated server should be built for developing and running the software. The software could then be tailored to use specific GPU models/drivers. Additionally, a docker image could make it easier for future developers to get set up with the various frameworks used throughout the program.

The groundwork we laid in our project should help fulfil these future work.

9. Acknowledgements

We want to thank our clients Dr. Shawn Butler and Dr. Vinod Vasudevan for providing tremendous guidance and support for this project. We also like to thank our instructor Dr. Mock for assisting us whenever we needed it. Finally, we would like to extend our appreciation to the Pacific Northwest Transportation Consortium for funding this project through the instrumentation used to collect the data.

10. References

[1] https://towardsdatascience.com/how-dbscan-works-and-why-should-i-use-it-443b4a191c80
[2] https://github.com/isl-org/Open3D-ML
[3] http://www.cvlibs.net/datasets/kitti/eval_object.php?obj_benchmark=3d
[4] https://github.com/hunglc007/tensorflow-yolov4-tflite
[5] https://github.com/ArashJavan/veloparser
[6] https://github.com/AlexeyAB/darknet
[7] https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html

Appendix A: User Guide

Collecting/Processing Raw Data

At the moment, raw data is 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: It is assumed that because there is only one LiDAR device to many cameras, 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 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 (Figure #). For distortion coefficients, take an image of the chessboard (Figure #) and use *tools/video/distortion_tool.py* to obtain the distortion coefficients.

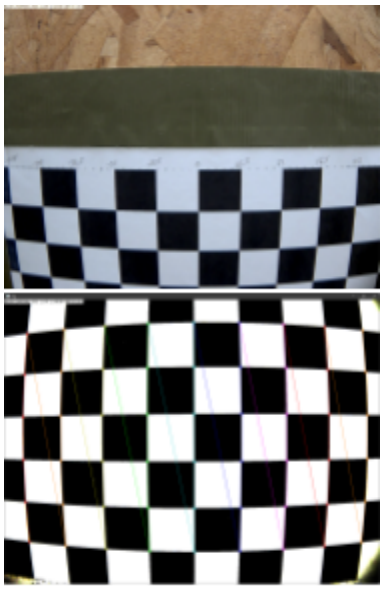


Figure #: FOV calibration image (top) and distortion calibration image (bottom).

Alignment Tool (*alignment_tool.py*)

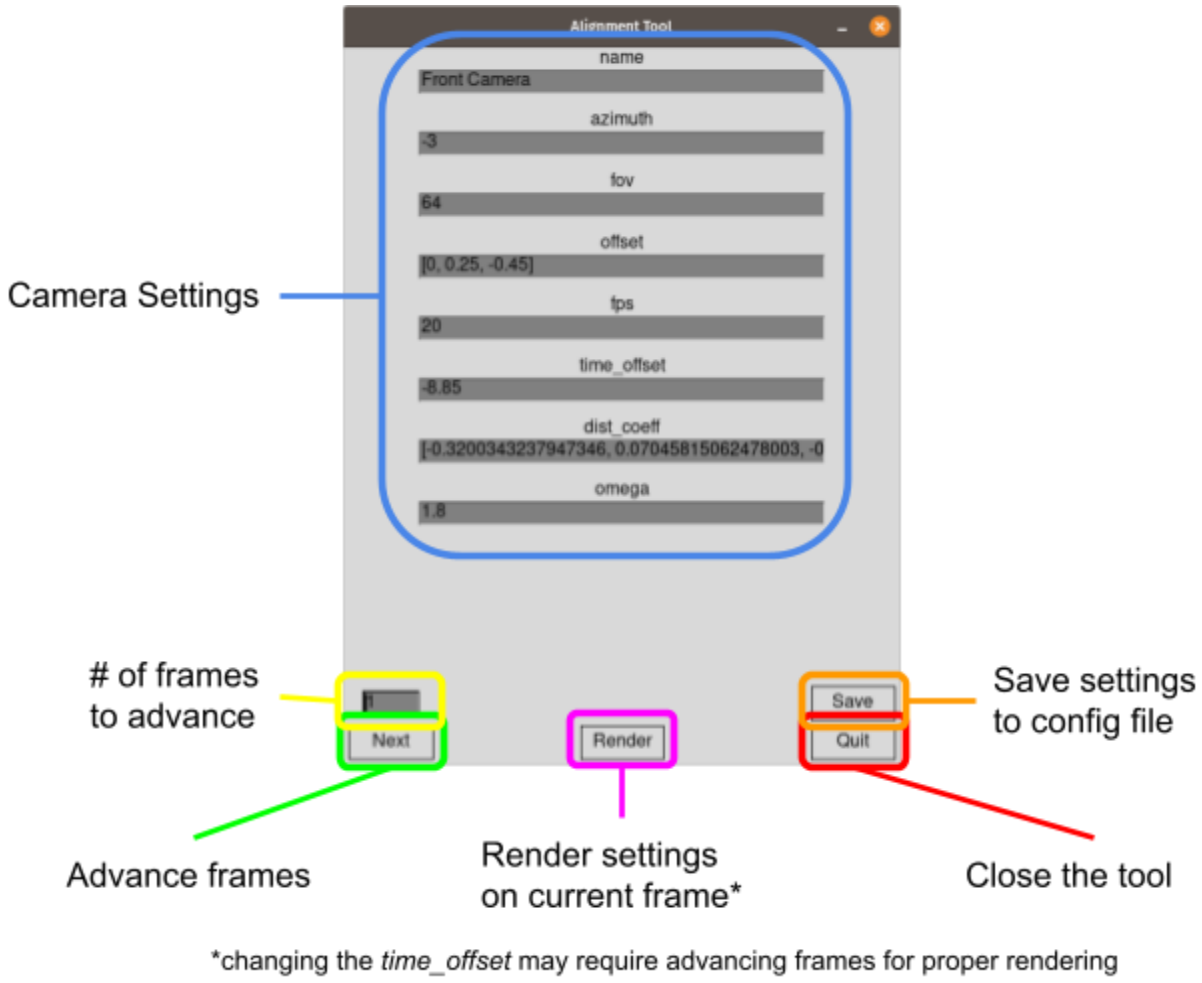
This tool allows a developer to verify LiDAR/camera transformations and timing and make real time adjustments to camera settings. The constants at the top of the Python program allow you to select your desired camera settings and paths to video and PCAP files. It is not a robust tool and will crash if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

1. The *cameras.json* file exists.
2. The camera index exists.
3. The PCAP path exists.

4. The video path exists.
   Note: these requirements are not exhaustive.

Upon running the tool, the data will be read in and three windows will 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



*changing the *time_offset* may require advancing frames for proper rendering

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 transforming LiDAR points to overlay on the camera image. The constants at the top of the Python program allow you to specify the image you took 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 – as seen in Figure #.