Contents

1 Introduction

The purpose of this document is to summarize my findings and experienced working with the Velodyne HDL-32E LiDAR. The objective is to provide the reader with an idea of how to gather data using the LiDAR and how to process the gathered data.

1.1 Reading Guide

The first part (Chapter 2) provides a quick start guide for using the Velodyne HDL-32E LiDAR. The main focus of this chapter is how to connect to the LiDAR, record data and playback recorded data.

The second and last part (Chapter 3) contains ideas for processing the gathered LiDAR data along with a brief explanation of the currently implemented software. This part are mainly geared toward persons with some technical insight.

2 Quick Start

The following provides a brief overview of how to get started using the Velodyne HDL32-E LiDAR. The focus being how to gather data and inspect it.

2.1 Hardware Setup

The hardware required to use the LiDAR consists of the following elements:

- The LiDAR, i.e. the Velodyne HDL-32E.
- The interface box.
- An AC to DC adapter. Voltages in the range $9 - 18V_{DC}$ are acceptable. (see the interface box manual[1] for more information).
- A computer with an ethernet port (RJ45).
- (Optional) GPS receiver. Only necessary if location and precise timestamps are required.
- (Optional) External harddrive. Only necessary if the internal harddrive of the computer is insufficient to store the anticipated amount of data.

How the different components are connected is shown in Figure 2.1. Note that the connection between the LiDAR and the interface box are hardwired. The AC to DC adapter and GPS are both connected to the interface box. The interface box and computer are connected using an ethernet cable (preferably CAT6).

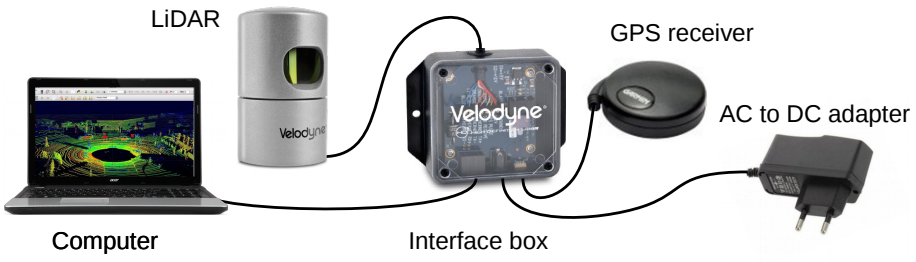

Figure 2.1: The different hardware components and how they are to be connected.

WARNING! Make sure that the LiDAR is properly mounted and that the upper part can move freely, as the LiDAR will start spinning once powered on.

WARNING! Do not modify the connection between the interface box and LiDAR before consulting the manual[1].

2.2 Configuring the LiDAR

The LiDAR contains a web interface in which it is possible to configure a variety of settings. The following will only cover the basic, i.e. connecting to the web interface and changing the main parameters. A more detailed guide is provided by Velodyne[2].

2.2.1 Connecting to the Web Interface

Apply the following steps to connect to the LiDAR's web interface:

1. Ensure that all cables are connected as outlined in Section 2.1. The LiDAR sensor should be powered on, i.e. the top of the sensor should be spinning.

2. Disable WiFi on your computer and configure your computer's IP adress to 192.168.1.77 (the last digits 77 can be anything, except 0, 255 or 201). A more in-depth guide on how to do this can be found in [2, page 4-10] for Windows 7 and in [2, page 11-12] for Mac OS.
   Note: This step can be skipped if the computer have already previously been configured in accordance with the above.

3. Open your preferred webbrowser (e.g. Google Chrome, Safari or Internet Explorer) and access the web adress 192.168.1.201. Your should now be greeted by the LiDAR's web interface as shown in Figure 2.2.

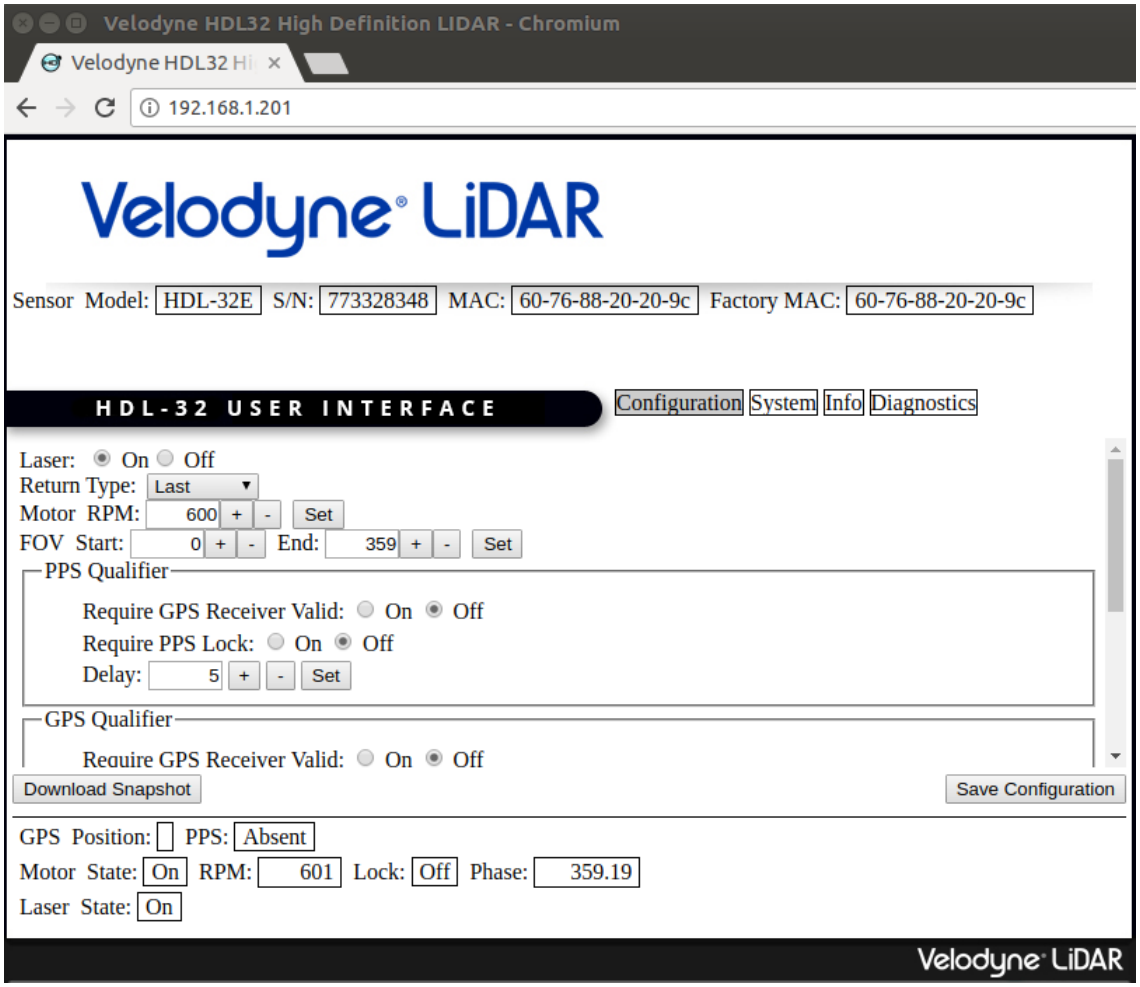

Figure 2.2: The LiDAR's web interface when accessed using Google Chrome.

2.2.2 LiDAR Settings

The following briefly highlights to two main settings of interest in the LiDAR webinterface.

- **FOV configuration** - Controls the field-of-view of the LiDAR. Any data outside the specified "start" and "end" angle will be discarded. The only advantage of limiting the field-of-view is a reduction in the amount of data gathered by the LiDAR and hence reduce the storage requirements. I.e. the precision of the measurements from the LiDAR will not increase when limiting the field-of-view.
- **RPM configuration** - Controls the rounds per minute of the LiDAR. The valid range for the RPM is 300 to 1200, which equates to 5-20 Hz. The configured RPM does not effect the amount of data gathered by the LiDAR, unlike the field-of-view setting. The effect of changing the RPM is as follows:
  - **Increase RPM** - It takes less time for the LiDAR to complete a full 360° rotation and it is hence better at capturing data of fast moving targets. The angular resolution is however reduced as the time between laser firings remains fixed.
  - **Decrease RPM** - It takes longer for the LiDAR to complete a full 360° rotation and the angular resolution is hence increased as the time between laser firings remains fixed. Fast moving targets may however be harder to capture.
- **Return Type** - It is possible for each of the LiDAR's laser beams to hit multiple objects due to the divergence of the beam, as illustrated in Figure 2.3. The LiDAR can hence be configured to report either; the **last return**, the **strongest return** or both (**dual mode**). The LiDAR may report the second-strongest return as well in dual mode if the latest and strongest returns are indentical. See [8, page 3-9] for more details.

  It is recommended to use dual mode if in doubt, as unnecessary data (i.e. returns) can be discarded at a later stage. The disadvantage of using dual mode is an increase in the amount of data received from the LiDAR and hence the amount of storage required during recording (see Section 2.4 for more information).

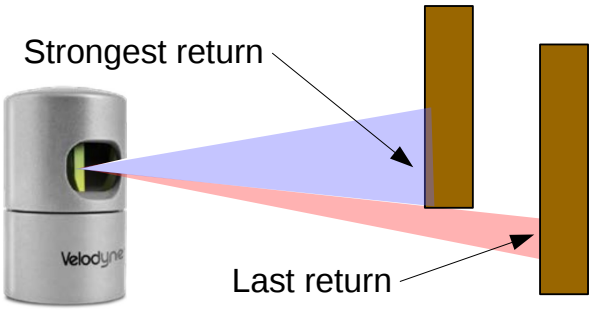

Figure 2.3: Example of how a single laser firing can result in multiple returns.

Tip: It is possible to view live data from the LiDAR (see Section 2.3) while configuring the above settings to see the effect of different configurations.

2.3 Inspecting Data

Data from the LiDAR can be inspected using the "VeloView" software provided by Velodyne Lidar and Kitware[3]. Versions of the software for both Linux, MacOS and Windows can be downloaded from Kitware's homepage[4]. The most recent version of the software have been included alongside this report as well. VeloView supports viewing both a live stream from the sensor and viewing a pre-recorded session.

2.3.1 Live Stream from Sensor

Perform the following steps to view live data from the LiDAR:

1. Ensure that the hardware is connected as outlined in Section 2.1.
2. Open VeloView and click on the icon in the upper left corner, as shown in Figure 2.4.

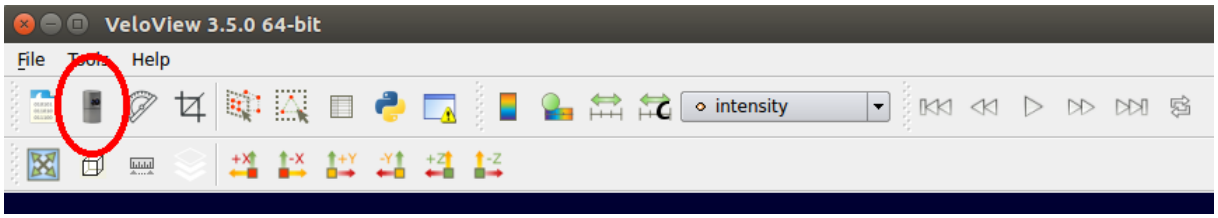

Figure 2.4: Open a live stream from the LiDAR by clicking on the marked icon.

3. Select the correct LiDAR model (i.e. "HDL-32E") in the "Sensor Configuration" dialog and click "OK".
4. Live data from the LiDAR should now be visible in the main window on VeloView.

Tip: Re-do the steps in Section 2.2.2 if no data shows up when using the live stream option. VeloView should be able to access the LiDAR's live stream if it is possible to connect to the LiDAR's web interface.

2.3.2 Pre-recorded Session

Perform the following steps to view a pre-recorded LiDAR session:

1. Open VeloView and click on the icon in the upper left corner, as shown in Figure 2.5.

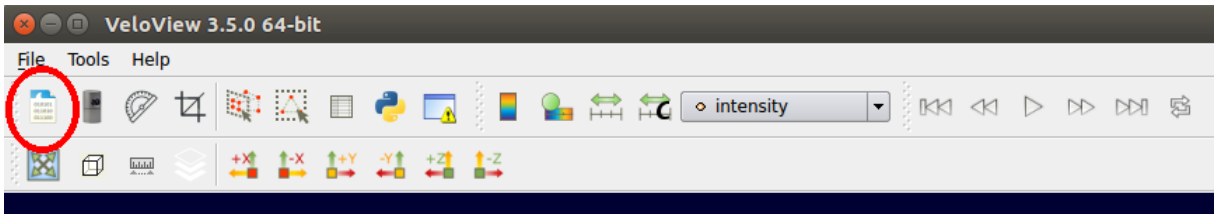

Figure 2.5: Open a pre-recorded LiDAR session by clicking on the marked icon.

2. Select the pre-recorded session to load by selecting the corresponding ".pcap"-file.

3. Select the correct LiDAR model (i.e. "HDL-32E") in the "Sensor Configuration" dialog and click "OK".

2.4 Recording Data

VeloView supports recording live data from the LiDAR to a single ".pcap"-file.

The steps to start recording live data are as follows:

1. Establish a connection to the LiDAR by following the steps in Section 2.3.1.

2. Click on the red "Record" button to start recording, as shown in Figure 2.6 and select where to save the ".pcap"-file in the dialog. The red "Record" button should now appear as being pushed/selected in order to indicate a recording session in progress.

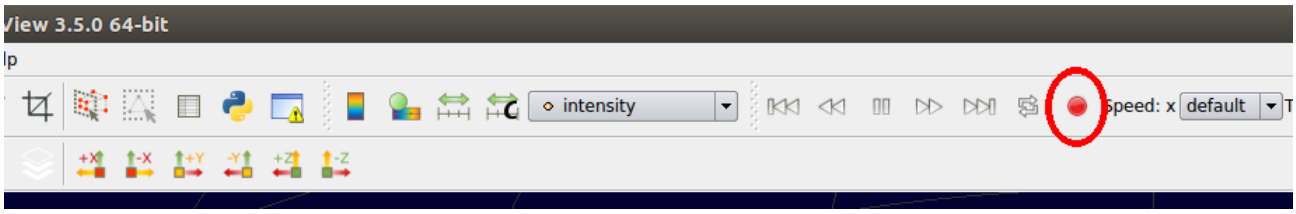

Figure 2.6: Start and stop recording live data from the LiDAR by clicking on the marked icon.

3. Click on the same "Record" button to stop recording.

The amount of storage required by the LiDAR when recording live data depends on the LiDAR's configuration, namely the selected field-of-view. A brief test was hence conducted in order to get some idea of what to expect in terms of storage usage when recording. The results are shown in Table 2.1.

FOV (Field-of-View)	**RPM** (Rounds per. minute)	**Used Storage** (Megabytes per. second)
$0 - 360°$	5 Hz	$\approx$ 2.4 MBps
$0 - 360°$	20 Hz	$\approx$ 2.4 MBps
$0 - 180°$	5 Hz	$\approx$ 1.2 MBps
$0 - 180°$	20 Hz	$\approx$ 1.2 MBps

Table 2.1: Storage usage of the LiDAR using different configurations.

3 LiDAR Processing

This chapter describes different ideas for the LiDAR data processing pipeline, along with a brief explanation of how the different stages are currently implemented. Some of the later stages in the pipeline are far from finished, as work have mainly been focused on the earlier stages.

The current pipeline are illustrated in Figure 3.1. The main idea is to identify objects of interest (i.e. pedestrians, bikes and cars) by identifying 3D points which does not belong to the background (i.e. the environment such as trees, roads, street signs and so on). The frames mentioned in the figure denotes the data gathered by the LiDAR during one 360° rotation.

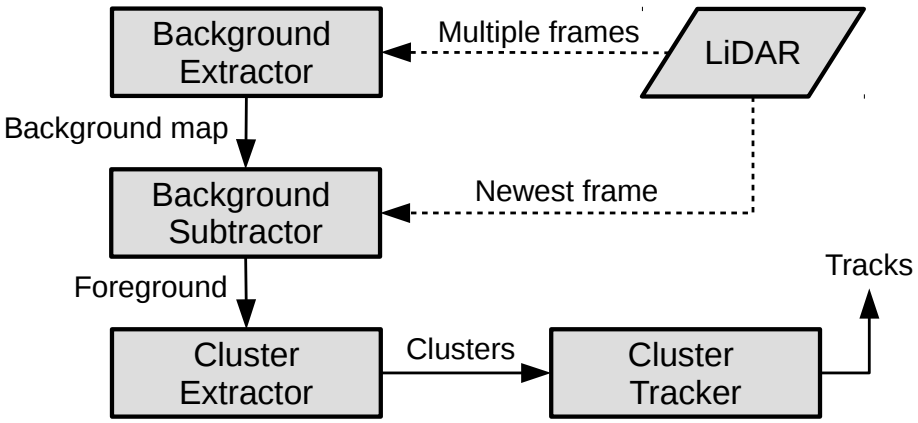

Figure 3.1: Outline of the proposed pipeline for processing LiDAR data.

The software is currently implemented in Python 3.5 and the following packets and libraries are used:

- PCL 1.8 (Point Cloud Library) (Processing of point cloud)
- Python-PCL (Python wrapper for the PCL library)
- PyQTGraph (Visualization of point cloud)
- PyOpenGL (Visualization of point cloud)

Details on how to install PCL 1.8 can be found in Appendix A. PyQTGraph and PyOpenGL can be installed using the "pip" installer. Instructions on how to install Python-PCL can be found on their github page[5].

3.1 Parsing Data

The first step of the pipeline is to parse the data from the LiDAR. Details of how to parse the data can be found in the manual of the Velodyne HDL-32[6] and VLP-16[7] LiDAR (the VLP-16 manual provides some additional details not found in the HDL-32 manual). The following mainly summarizes the parsing process by restating important figures and examples.

The data from the LiDAR is formatted as UDP (User Datagram Protocol) packets with a size of 1248 bytes. The format of the data packet is shown in Figure 3.2.

The data packet structure is as follows:

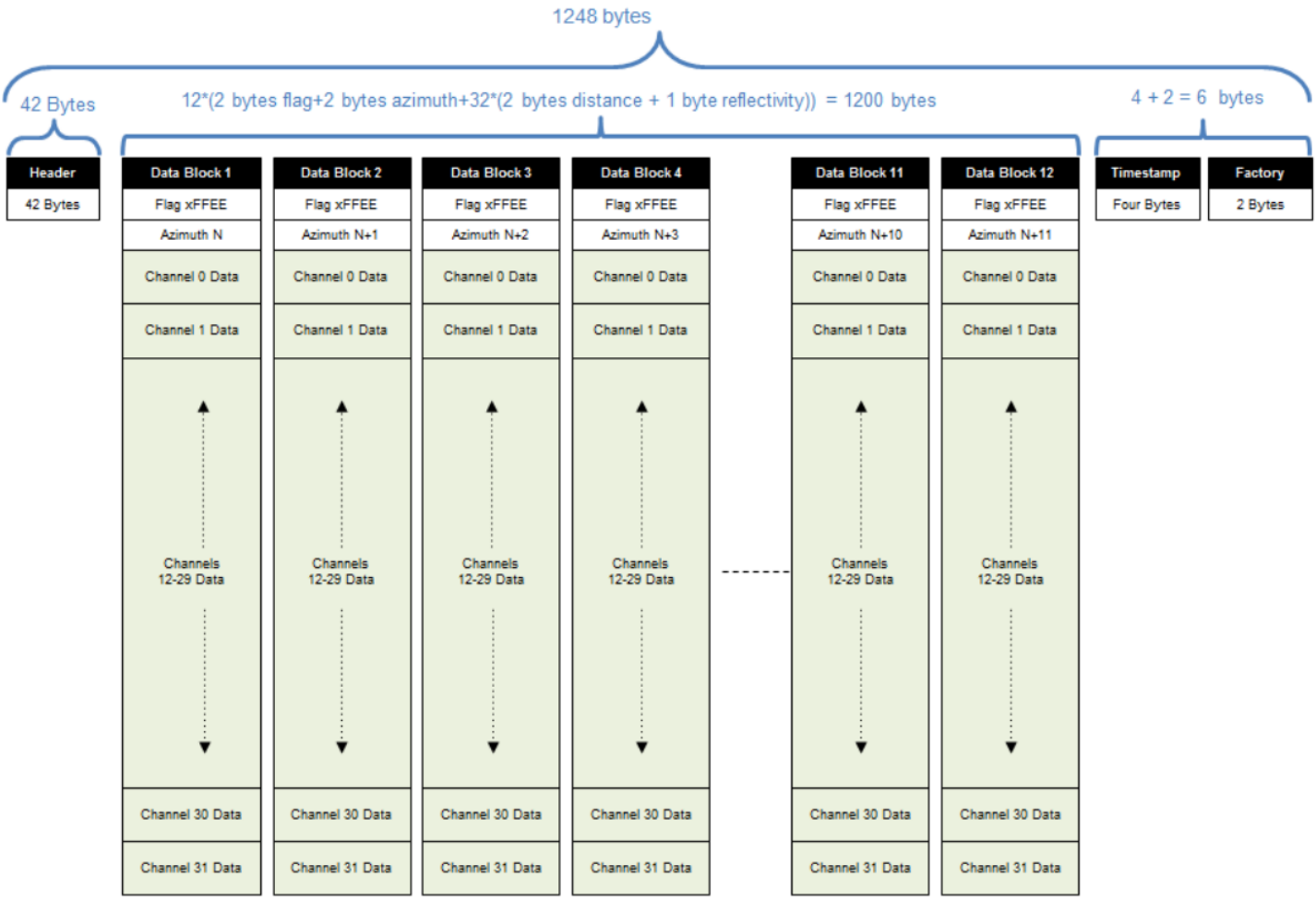

Figure 3.2: Data packet structure[8, page 14].

- Each **data packet** consists of 12 data blocks of 100 bytes each.
- Each **data block** begins with two bytes identicating a start identifier followed by another two bytes identicating the azimuth angle (i.e. rotation in horizontal plane) of the LiDAR. The two bytes identicating the azimuth value is followed by 32 data points.
- Each **data point** consists of three bytes. The first two bytes of the data point indicates range, i.e. the measured distance to an objectm and the last single byte indicates the reflectiveness of the object.

Azimuth Angle Parsing

Converting the two azimuth bytes to an angle (degrees) is done as follows:

- Read the azimuth bytes, e.g. “0x33” and “0x71”
- Reverse the bytes and combine them: “0x7133”
  (the data are transmitted least significant byte first)

- Cast to unsiged integer: "0x7133" $\rightarrow 28,979$
- Divide by 100 to get the final result: $289.79°$

Range Parsing

Converting the two range bytes to distance (meters) is done as follows:

- Read the angle bytes, e.g. "0xDA" and "0x52"
- Reverse the bytes and combine them: "0x52DA"
  (the data are transmitted least significant byte first)
- Cast to unsiged integer: "0x52DA" $\rightarrow 21,210$
- Multiple by 0.002 meters to get the final result: 42.42 meters
  (the LiDAR reports range in increments of 2 milimeters)

Polar Angle Parsing

The polar angle (i.e. rotation in vertical plane) associated with each data point can be found using the lookup table (LUT) shown in Table 3.1. The laser id needed for the lookup can be derived from the index of the current data point in the current block, as they are reported in order from lowest to highest. I.e. the first data point is laser id = 0 and the last data point is laser id = 31.

Converting to Cartesian Coordinates

The conversion from spherical to cartesian coordinates can be performed as shown in Figure 3.3, where R is the range, ω is the polar angle and α is the azimuth angle.

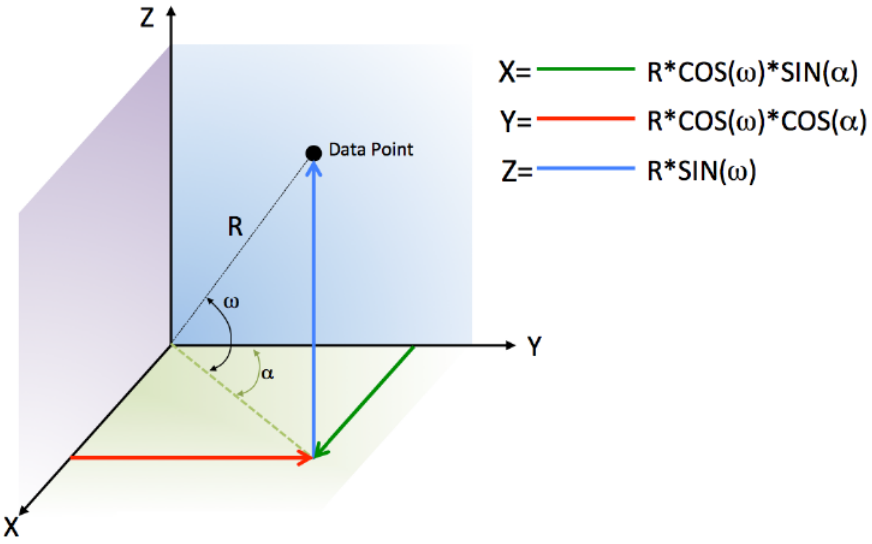

Figure 3.3: Conversion from spherical coordinates to cartesian coordinates[7, page 11].

Laser ID	Polar angle
0	-30.67
1	-9.33
2	-29.33
3	-8.00
4	-28.00
5	-6.66
6	-26.66
7	-5.33
8	-25.33
9	-4.00
10	-24.00
11	-2.67
12	-22.67
13	-1.33
14	-21.33
15	0.00
16	-20.00
17	1.33
18	-18.67
19	2.67
20	-17.33
21	4.00
22	-16.00
23	5.33
24	-14.67
25	6.67
26	-13.33
27	8.00
28	-12.00
29	9.33
30	-10.67
31	10.67

Table 3.1: Polar angle lookup table for the Velodyne HDL-32E.

3.1.1 Current Implementation

The current implementation supports parsing data directly from the sensor or from a ".pcap"-file, which is the file format used by VeloView (see Section 2.3) when recording data.

The following files pertains to data parsing:

- **"DataEntities.py"** - The logic for parsing the data packets.
- **"PcapParser.py"** - The logic for loading ".pcap"-files.
- **"OfflineDemo.py"** - Contains an example of parsing data from a ".pcap"-file.

- **"LiveDemo.py"** - Contains an example of parsing data directly from the LiDAR.

The following are currently not implemented and would be good ideas for future work:

- Interpolate azimuth angles while parsing packets. See [8, page 16-17] and [7, page 25].
- Implement parsing of timestamps from the LiDAR as the current implementation discards timestamp information. See See [8, page 19-22]
- Implement parsing of dual return packets as the current implementation always assume the data to be formatted as a single return packet. See [8, page 14-15].

3.2 Background Extraction

The purpose of this module is to generate a map of the background, i.e. static objects such as buildings, tree, lampposts and so on. The output of this module is hence a point cloud containing the XYZ positions of objects identified as belogning to the background.

3.2.1 Approach

The proposed approach relies on access to multiple LiDAR frames in order to extract the background. Note that the approach is based on the assumption that the position of the LiDAR is fixed and does not change.

The first step is to comtemplate each of the LiDAR frames as range images, i.e. an image where the x-axis is the azimuth angles, the y-axis is the polar angles (or laser IDs) and the intensity of each pixel is the range. An example of such is shown in Figure 3.4. The horizontal lines in the image are due to the lasers being fired and reported in a staggered pattern, which is also evident in Table 3.1.

Figure 3.4: A section of a range image from a LiDAR frame. The x-axis signifies the azimuth angle while y-axis signifies to polar angle.

All the LiDAR frames, after the conversion to range images, are than stacked on top of each other to to create a third axis. The p'th percentile is then calculate along this third axis, resulting in a range image of the background. I.e. the p'th percentile range is calculated for each pair of azimuth and polar angles.

The idea of using the p'th percentile of the range images is based on the observation that road users will often stop moving in the same spots in traffic. For instance, markings on the road at intersections will often cause cars to stop in the same locations when waiting for the traffic signal to change. Hence when looking at range readings from multiple LiDAR frames, but with the same azimuth and polar angles, it may look like the distribution in Figure 3.5.

It is quite clear that the distribution of the range readings appears to be bimodal, i.e. it contains two peaks. The peak at the lower ranges could be vehicles idling at a red light whereas the peak at

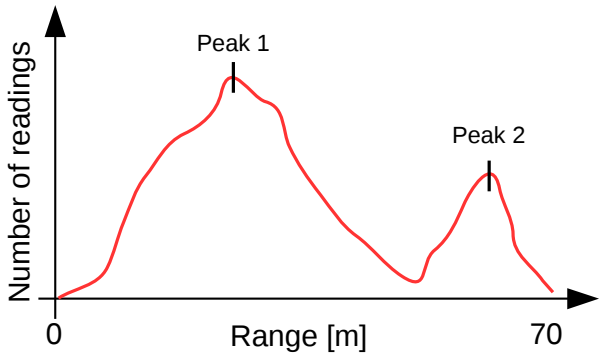

Figure 3.5: A possible distribution of range readings for a specific pair of azimuth and polar angles.

higher ranges is likely to be the actual background. Note that the lower range peak is much more frequent than the peak at higher ranges, signifying that more time is spent observing the vehicle than the actual background, such as the road or buildings.

The background peak, i.e. the peak with the highest range values, is found be setting the value of p relatively high when calculating the p'th percentile (values between 70 and 80 works well). The underlying base assumption of the entire background extraction module is hence that the peak with the highest range values must be the range to the actual background.

Identifying the maximum range value for each azimuth and polar angle should in theory work as well but this approach was found to be too susceptible to noise (for instance lampposts not being detected as part of the background).

The last step of the background extraction approach is to calculate XYZ positions from the newly found range values. This is done by applying the conversion from spherical to cartesian coordiantes using the equation mentioned in Section 3.1.

3.2.2 Current Implementation

The logic this module is implemented in the file: **"BackgroundExtractor.py"**. The current implementation can be executed on its own using the following command:

```
1    $ python BackgroundExtractor.py -f path/to/PcapFile -p 80 -z 70
```

When executed on its own it will extract the background from the specified ".pcap"-file and display it. The argument **-p** specifies the percentile to use whereas the argument **-z** specifies the minimum acceptable percentage of non-zero readings for each azimuth and polar angle pair.

Range readings for a specific pair of azimuth and polar angles are hence discarded if they contain a lot of non-zero readings (i.e. range = 0) as failure to do so was found to introduce a lot of noise in the extracted background. The background extraction module can also be imported and used as a class, as shown in **"OfflineDemo.py"**.

The following could be ideas for future work:

- Find a better way of aligning the range images in terms of the azimuth values. The current implementation only ensures that the range images being stacked have the same dimensions by cutting them to size. It is hence possible that the azimuth values are not perfectly aligned

in the current implementation. The impact of this does not appear servere in the tested scenarios and it is likely that the slight misalignment may just act as blurring filter.

3.3 Background Subtraction

The objective of this module is to accept a cloud of XYZ points and identify each of them as either foreground or background using the background map extracted in Section 3.2.

3.3.1 Approach

A straight forward approach would be to compare each XYZ point in the input cloud to each point in the background map and then mark the current point as background if some criteria is meet. A possible criteria could be the euclidean distance between the current point and a point in the background map being less than a pre-defined threshold. Such an implementation would in theory work but it would be highly inefficient as well.

The background subtraction module does however employ a more efficient approach than the above, as a octree data structure is used to represent the map of the background. The idea is to recursively divide the 3D search space into increasingly lower cubes, as shown in Figure 3.6.

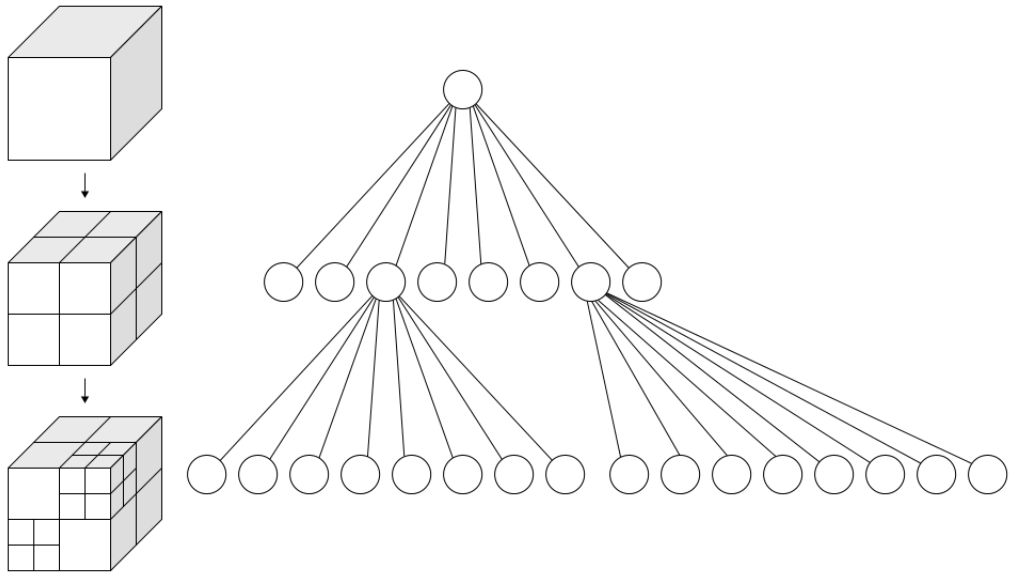

Figure 3.6: Example of dividing a 3D space into increasingly smaller cubes along with the corresponding octree[9].

The octree data is hence quite efficient data structure to work with, as each level of the tree reduces the search space by a factor of 8.

3.3.2 Current Implementation

The logic for this module can be found in the file: **"BackgroundSubtractor.py"** and it mainly consists of calls to PCL's Octree change detector. The **"resolution"** argument in the constructor of the class defines the minimum size of the cubes in the octree in meters. An example of using the background subtraction module can be found in **"OfflineDemo.py"**.

3.4 Cluster Extraction

The purpose of this module is to cluster the XYZ points identified as foreground in to clusters. Preferably one cluster for each real objects, i.e. bikes, pedestrians, cars and so on.

3.5 Approach

The currently implemented approach for clustering is very basic as it is purely done based on the euclidean distance in a 3D space. The main reason for doing was simply because it was supported out-of-the-box by PCL.

Using euclidean distance as a criteria for clustering is problematic as the resolution of the data differs based on the range to the LiDAR. The resolution meaning the number XYZ points per. cubic meters. A clustering paradigme based on euclidean distance may hence behave differently based on the range of the points to be clustered.

Another issue of using euclidean distance as the basic for clustering is the difference in resolution between the azimuth and polar angles. The later having a much lower resolution due to the design of the LiDAR.

3.5.1 Current Implementation

The logic of this module can be found in the file: **"BackgroundSubtractor.py"** and it mainly consists of calls to PCL's cluster extractor class. An example of using the background subtraction module can be found in **"OfflineDemo.py"**.

The following could be ideas for future work:

- Cluster based on point normals by appling a smoothness constraint where the normal between neighboring may only diverge a certain amount. An example of such is PCL's region growing segmentation [10].
- Cluster points in 2D by projection the XYZ points to a common ground plane. This could solve the issue of the azimuth and polar angles having different resolutions.

3.6 Cluster Tracking

Currently not implemented.

Bibliography

[1] *Interface Box User Manual*, a copy is included in the "appendix/manuals" directory.

[2] *Webserver User Guide*, a copy is included in the "appendix/manuals" directory.

[3] Kitware. github - VeloView. [Online]. Available: https://github.com/Kitware/VeloView

[4] Kitware. ParaView - VeloView. [Online]. Available: https://www.paraview.org/VeloView/

[5] Strawlab. github - python-pcl. [Online]. Available: https://github.com/strawlab/python-pcl

[6] *HDL-32E User Manual*, a copy is included in the "appendix/manuals" directory.

[7] *VLP-16 User Manual*, a copy is included in the "appendix/manuals" directory.

[8] *HDL-32E Application Note - Packet Structure Timing Definition*, a copy is included in the "appendix/manuals" directory.

[9] Wikipedia. Octree. [Online]. Available: https://en.wikipedia.org/wiki/Octree

[10] PCL. Region growing segmentation. [Online]. Available: http://pointclouds.org/documentation/tutorials/region_growing_segmentation.php#region-growing-segmentation

Appendices

A Installing PCL 1.8

The following instructions was tested on Ubuntu 16.04 LTS.

```
## Source:
## https://askubuntu.com/questions/916260/how-to-install-point-cloud-
   library-v1-8-pcl-1-8-0-on-ubuntu-16-04-2-lts-for

# 1) Install oracle-java8-jdk:
sudo add-apt-repository -y ppa:webupd8team/java && sudo apt update
sudo apt -y install oracle-java8-installer

# 2) Install universal pre-requisites:
sudo apt -y install g++ cmake cmake-gui doxygen mpi-default-dev
sudo apt -y openmpi-bin openmpi-common libusb-1.0-0-dev libqhull*
sudo apt -y libusb-dev libgtest-dev
sudo apt -y install git-core freeglut3-dev pkg-config build-essential
sudo apt -y libxmu-dev libxi-dev libphonon-dev libphonon-dev
sudo apt -y phonon-backend-gstreamer install phonon-backend-vlc
sudo apt -y graphviz mono-complete qt-sdk libflann-dev

# 3) Install PCL dependicies
sudo apt -y install libflann1.8 libboost1.58-all-dev

cd ~/Downloads
wget http://launchpadlibrarian.net/209530212/libeigen3-dev_3.2.5-4_all.
   deb
sudo dpkg -i libeigen3-dev_3.2.5-4_all.deb
sudo apt-mark hold libeigen3-dev

wget http://www.vtk.org/files/release/7.1/VTK-7.1.0.tar.gz
tar -xf VTK-7.1.0.tar.gz
cd VTK-7.1.0 && mkdir build && cd build
cmake ..
make -j7
sudo make install

# 4) Build and install PCL
cd ~/Downloads
wget https://github.com/PointCloudLibrary/pcl/archive/pcl-1.8.0.tar.gz
tar -xf pcl-1.8.0.tar.gz
cd pcl-pcl-1.8.0 && mkdir build && cd build
cmake ..
make -j7
sudo make install

# 5) Clean up
cd ~/Downloads
rm libeigen3-dev_3.2.5-4_all.deb VTK-7.1.0.tar.gz pcl-1.8.0.tar.gz
sudo rm -r VTK-7.1.0 pcl-pcl-1.8.0
```