Integrated and Responsive Indoor/Outdoor Navigation - Final Reporting

Zhiyang (Kilam) Lin, Maximilian Keller, Mitko Aleksandrov, Jack Barton, Binghao Li, Johnson Xuesong Shen and Sisi Zlatanova

School of Built Environment, School of Minerals and Energy Resources Engineering, School of Civil and Environmental Engineering

Table of Contents

Abstract

This document summarises the investigations, developments and tests performed within the DGFI 2021 cross faculty seed project 'Integrated and Responsive Indoor/Outdoor Navigation'. The project was executed in collaboration between the School of Built Environment, the School of Minerals and Energy Resources Engineering, the School of Civil and Environmental Engineering and PAM (pam.co). The project investigated machine learning approaches for detecting moving pedestrians, using low-cost portable laser scanners. The experiments have clearly demonstrated that the developed approach is promising, and the proposed systems architecture has the potential to provide real-time results.

Introduction

To date, many approaches for indoor and outdoor and seamless indoor/outdoor navigation have been investigated. Many research groups have proposed methods for considering the dimensions of the user, path obstructing objects, or locomotion modes (walking, driving, flying). However, a little attention has been paid on methods to reflect dynamic environmental changes or moving objects such as pedestrians or crowds. Some research considering dynamic obstacles have been reported in the context of disaster management (Wang and Zlatanova 2019) or autonomous driving (Maurelli et al 2009, Rozenzweig et al 2015, Mozaffari et al 2020). Tracking of pedestrians has been developed as a separate area of research mostly for studying pedestrian behaviour and mobility patterns (Scheunert et al 2004, Li et al 2013, Xiao et al 2016, Qian et al 2021). There are many emerging solutions for tracking moving objects but they have been not considered in routing or evacuation applications for pedestrians and takin into account the real-world environmental context.

Existing solutions for tracking are strongly device-driven, leaving the corporate customer to maintain their own loosely-connected spatial documentation for the building's life-span. Whenever a physical change occurs, whether a floor layout, or unexpected crowd obstruction, these documents become unfit-for-purpose. Many technologies exist for scanning interior spaces, but the resulting datasets require expertise/time to process the data and extract the dynamic modifications. Our solution for monitoring/modelling aims for detecting pedestrians and recording crowd changes, which obstruct the free navigable area in built environments to include them in the path finding algorithms.

This project concentrated on investigating LiDAR sensors applied in autonomous driving, machine learning (ML), and advanced 3D modelling and simulation using a game engine for the purpose of estimating free navigable area. To compensate for large amounts of unstructured point cloud measurements, ML algorithms are used to identify the pedestrians, compute their density and delineate the free/secure navigation areas. The developed algorithms are tested in lab environment. Some of them have been also tested in the lower campus around the Red Centre building, UNSW Kensington campus.

This report is organised as follows. The next section presents the overall system architecture. Section two elaborates on the Lidar sensors and the data collection. The training of the machine learning model as well as the prediction of pedestrians is discussed in section three. Section four elaborates on the density computation. Section five provides further information on possibilities for visualisation and linking the different algorithms for a real-time experiment.

Workflow

The problem we have focussed on, is detecting pedestrians in a specific area and computing the density for a given period of time. If the density is higher than a given threshold, the area is considered too crowded and is excluded from the overall available navigation area. The shape of the area can be delineated (not considered in this report) and given as a polygon geometry (i.e. obstacle) to the routing application. The routing applications can make sure that that the area is avoided either by deactivating the network edges passing through the obstacle (Wang and Zlatanova 2020) or by re-computing the network (Aleksandrov et al 2021). In this project we investigated the machine learning technique to detect the pedestrians. As well-known, ML requires training with a large number of data sets to be able to provide reasonable predictions. In our experiments we have used KITTI data set (Geiger et al 2012) for the training. Since the duration of the project was during Covid restriction, no real experiments were completed. Instead, synthetic data sets was created by mimicking the scanner parameters and running an agent-based simulation in a game engine.

We have selected RoboSense scanners, VoxelNET and Unity3D for collecting data, detecting pedestrians and creatin synthetic data for the training of VoxelNET. Figure 1 presents the overall system architecture of the developments. The first block incorporates the data collection and the point cloud processing. The second block produces the pedestrian prediction result for further processing. The next component is visualisation. Rviz (GUI in ROS) is used as an example in the workflow, which takes the output (detected pedestrians) for visualisation. Robosense has its graphical tool which is called "RS Viewer" which can also be used in the workflow. Finally, the "API service" takes the detected pedestrians and incorporates it into a service for an API call.

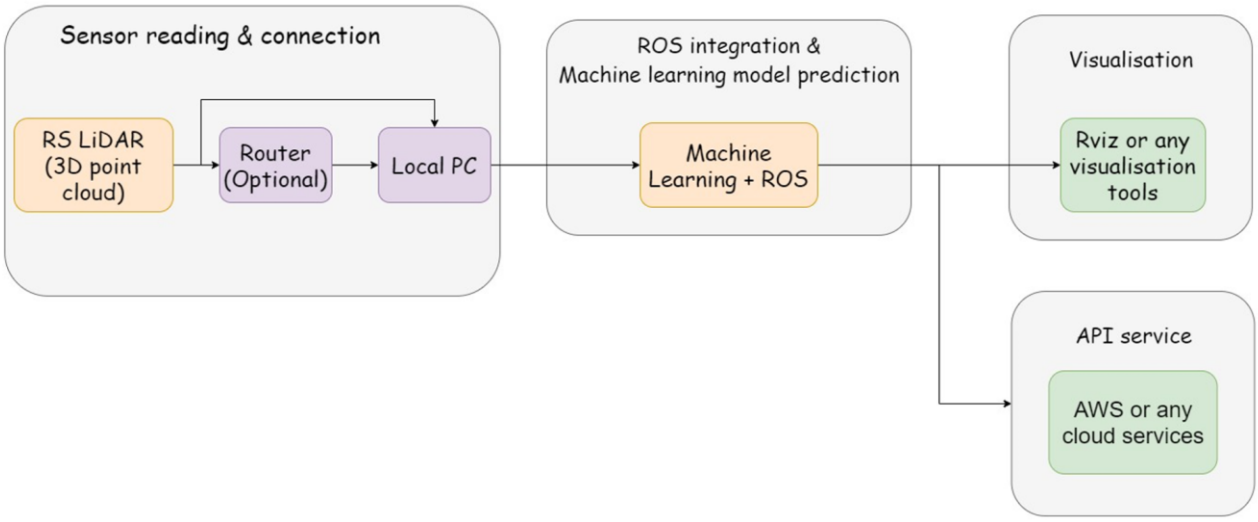


Figure 1: Overall system architecture

ROS stands for Robot Operating System, which is an open-source framework for programming a robotic system or sensors that have ROS compatible driver available (such as Robosense 3D LiDAR). ROS should be installed in Linux distributions such as Ubuntu 16.04, 18.04 and 20.04. The recommended version is 18.04 and Melodic as they are compatible with most of the existing packages for ROS and have been tested for obtaining the real scan from the RoboSense 3D LiDAR.

Robosense Black Pearl LIDAR Scanner Remarks

We have concentrated on LiDAR sensors because they can provide an accurate location and at the same time, they does not reveal any personal details. Other techniques for localisation such as Wifi (Verbree et al 2013) or CCTV cameras (Quing et al 2021) require strict data anonymising procedures for privacy protection.

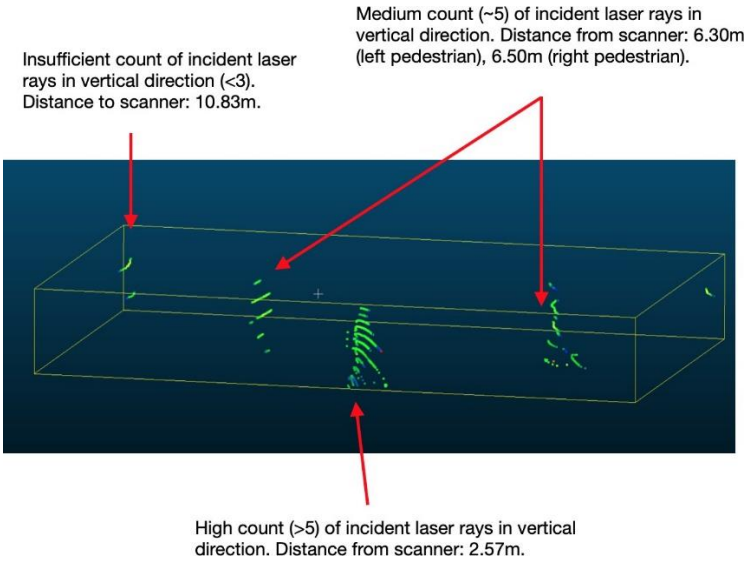


Figure 2:Example of a scan obtained with RoboSense Black Perl (RS-Bpearl)

We have selected the RoboSense Black Pearl (RS-Bpearl) 360˚x 90˚ Super Wide Field of View (FOV), Short-range LIDAR scanner which is designed specifically for blind spotdetection in autonomous vehicles. As such, this sensor can capture an almost complete hemispherical scan, ideal for capturing complete scenes from wall or ceiling mounted locations (similar to CCTV camera positioning). The trade-off to the substantial FOV of this sensor is a comparatively low resolution of these scans which are comprised of only 57,600 points per scan. As a result, the vertical angle between lasers is fixed at a relatively large ~2.81˚. Figure 2 and Figure 3 illustrates a scan, illustrating the number of points per a pedestrian with respect to the distance to the scanner.

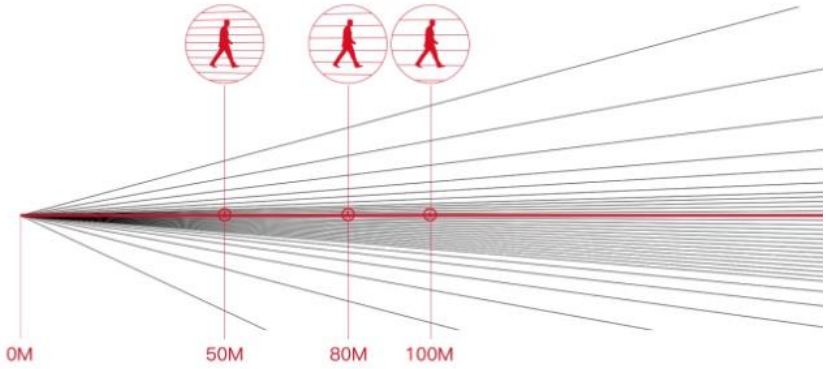


Figure 3: Manufacturer estimates for the number is scans with respect to distance to the scanner.

Tests have shown that the currently trained version of the VoxelNET detection network requires at least a set of three distinct vertical lasers to be incident on a pedestrian for a successful prediction. The figure below depicts a visualisation of an example scan conducted with five pedestrians at various distances to the scanner. This example clearly shows the range limitation of the RS-Bpearl scanner, with a pedestrian who is less than 11m far from the scanner with only two incident vertical laser rays.

Clearly, to increase the detectable range, several scanners should be used, or the maximum desirable range of the detectable pedestrians should be known, and at this distance the vertical separation of laser rays should be such that at least three laser rays are incident on the average person's height (Figure 4).

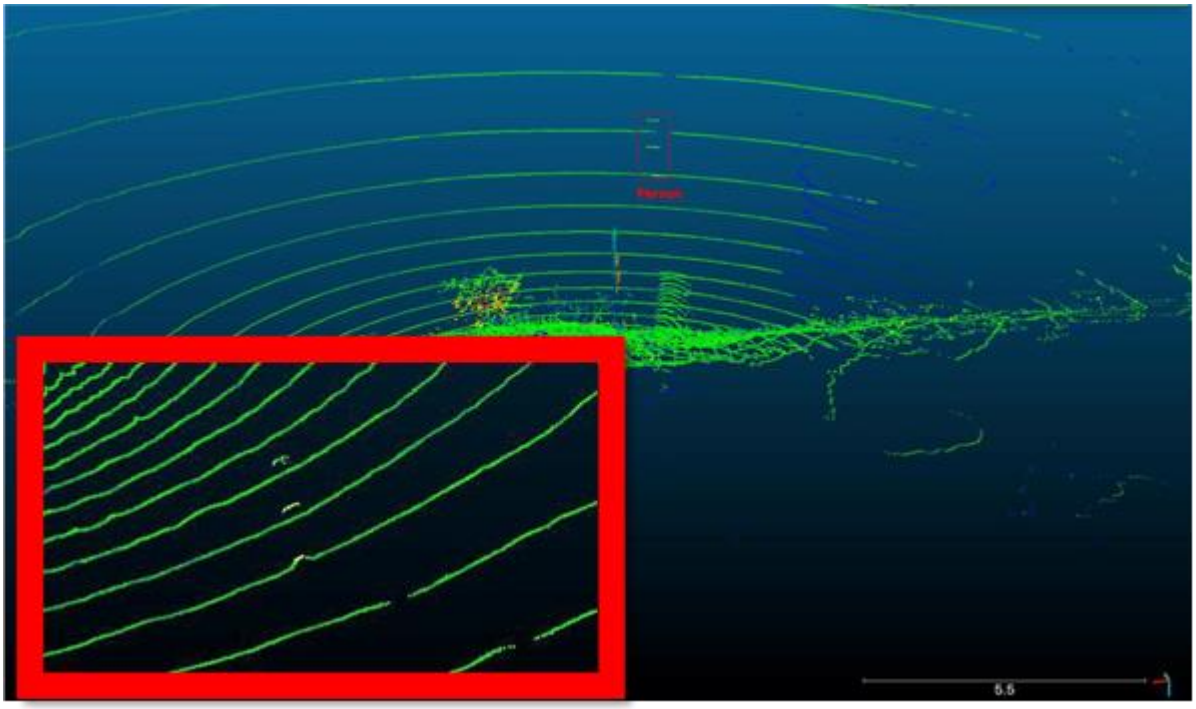


Figure 4: Illustration of pedestrian at the edge of the detectable range of the scanner (14m from scanner).

For example, the scan in Figure 4 was taken at the edge of a grass hill, which presents one benefit of the wide FOV of the Black Pearl scanner. It can be used in non-planar areas such that the scanning plane does not need to be aligned with the ground plane, as is the case with high resolution,lower FOV scanners (e.g. RS-LIDAR-16).

Obtaining data from the RS-Bpearl scanner

The RS-Bpearl sensor communicates using the IP/UDP protocol via an Ethernet connection. The UDP protocol packet is 1290 bytes long and consists of a 1248-byte valid payload and a 42-byte header. The IP address and port number of RS-Bpearl is set in the factory and the sensor will direct data to a defined computer IP address which is also set in factory. These addresses can be changed by the user. The sensor can communicate with a computer using the following protocols:

- MSOP (Main Data Stream Output Protocol). Distance, azimuth and reflectivity data collected by the sensor are packed and output to computer.
- DIFOP (Device Information Output Protocol). Monitor the current configurationinformation of the sensor.
- UCWP (User Configuration Write Protocol). User can modify some parameters of thesensor as needed.

RoboSense manufacturer maintains two GitHub repositories for obtaining data from the RS-Bpearl sensor. They provide a Software Development Kit (SDK), which includes main driver software and support for the Robot Operating System (ROS1 & ROS2) and protobuf-UDP communication (running on Ubuntu). Additionally, they provide the cross-platform driver kernel for RoboSense LiDARitself.

Within the project, we have tested the sensor in the ROS (noetic version) environment (Appendix ROS/RS-Lidar Driver setup guide document) successfully. Additionally, a python script waswritten to directly intercept the UDP data from the sensor through a socket connecting to the Ethernet port of the computer (Figure 5). The data packet is then parsed and decoded according to thepacket structure given in the RS-Bpearl user manual. This method was chosen as a more versatile and lightweight approach compared to the rs-lidar driver.

Figure 5: MSOP Packet structure

Based off the data packet structure outlined in the RoboSense manual, the following figures show screenshots of a simplified version of the python script used to decode these packets. The program could be further improved for efficiency, and currently uses two nested loops to iterate through the 12 data blocks containing 32 channels of data each (every channel is one of the 32 fixed lasers on the scanner). Comments are included throughout this code (Figure 6).

```python
#Omega are the vertical angles of lasers (fixed and given by laser ID - 0-32)
omega =
    [89.5,81.0625,78.25,72.625,67,61.375,55.75,50.125,86.6875,83.875,75.4375,69.8125,64.1875,58.5625,52.9375,47
    .3125,44.5,38.875,33.25,27.625,22,16.375,10.75,5.125,41.6875,36.0625,30.4375,24.8125,19.1875,13.5625,7
    .9375,2.3125]
#Actual value = readout * resolution
distanceResolution = 0.005
azimuthResolution = 0.01

# A function to convert the spherical data array to cartesian data array
def spherical2cartesian(frameData):
    distance = frameData[:,0]
    azimuth = frameData[:,1]
    omega = frameData[:,2]
    intensity = frameData[:,3]

    dist_cos = distance * numpy.cos(omega)
    pointX = dist_cos * numpy.sin(azimuth)
    pointY = dist_cos * numpy.cos(azimuth)
    pointZ = distance * numpy.sin(omega)
    frameCartesian = numpy.asarray([pointX,pointY,pointZ,intensity]).transpose()

    return frameCartesian

# Open a connection using a python socket to the port associated with the sensor
    sock = socket.socket(socket.AF_INET, # Internet
           socket.SOCK_DGRAM) # UDP
    sock.bind((UDP_IP, UDP_PORT))

# Main loop
while True:
    # Receive a data packet
    data, addr = sock.recvfrom(1248) # Max packet size in bytes (buffer)
    # Header is the first 8 bytes
    header = data[0:8]
    # Timestamp for each packet is bytes 20-30
    timeStamp = data[20:30]

    # Loop through the 12 datablocks in one packet
    for BlockNumber in range(0,12):
        blockStartIndex = 42+BlockNumber*100
        blockEndIndex = 142+BlockNumber*100

        # Obtain the pointcloud data (each data block has its own header too)
        dataBlock = data[blockStartIndex:blockEndIndex]

        # Obtain Azimuth for the data Block
        azimuthbyte1 = dataBlock[2]
        azimuthbyte2 = dataBlock[3]
        # Convert the raw value to angle
        azimuth = (azimuthbyte1*256+azimuthbyte2)*azimuthResolution

        # This condition checks if a new scan has begun
        # The data should be saved and sent as a complete frame here
        if (prevAzimuth>azimuth):
            # Convert to cartesian coordinates
            frameCartesian = spherical2cartesian(frameData)
            # Save an ASCII file with the frame contents
            numpy.savetxt('CaptureFrame{}.csv'.format(frameCounter), frameCartesian, delimiter=",")
            # Clear the data array for the next frame
            frameData = []
            # Increment a counter to keep track of frames
            frameCounter += 1

        for channelNum in range(0,32):
            # Calulate the offsets of the distance/intensity data for each channel
            channelStartIndex = 4 + channelNum*3
            channelEndIndex = channelStartIndex + 3
            # Copy data for one individual channel
            channelData = dataBlock[channelStartIndex:channelEndIndex]
            # Copy distance high and low byte
            distancebyte1 = int(channelData[0])
            distancebyte2 = int(channelData[1])
            distance = (distancebyte1*256 + distancebyte2)*distanceResolution
            intensity = int(channelData[2])

            # Validate each individual point (intensity and distance)
            if (cycle == 0 and distance < MAX_DISTANCE and intensity > MIN_INTENSITY):
                frameData.append([distance, numpy.deg2rad(azimuth+delta), numpy.deg2rad(omega[channelNum]),
                    intensity])
        # Keep Track of previous azimuth to determine transition to beginning of new frame
        prevAzimuth = azimuth
```

Figure 6: Simplified version of python script to decode UDP packets.

Improvements to the efficiency of the python code could be made by compiling the script using *Numba* to translate the python and *NumPy* functions into fast machine code. One version of this script was implemented entirely using *NumPy* functions to make this possible. Alternatively, the script could be re-written in a faster language such as C++.

Background Removal

The next step in processing the point clouds is extracting the moving objects by removing all static background. The most straightforward removal technique was cropping off the background points. This could be easily done since the sensor is stationary and the approximate measurements of the scene were testing data is obtained from is known. A simple python script was also used to crop, transform and re-orient the axes of point cloud frames (Figure 7).

```python
import os
import struct
import numpy as np
import csv
import math

os.chdir("SensorData/")
print(os.getcwd())

rotation_degrees = 45
rotation_radians = np.radians(rotation_degrees)
rotation_axis = np.array([1, 0, 0])
A =
    [[1,0,0],[0,np.cos(rotation_radians),-np.sin(rotation_radians)],
    [0,np.sin(rotation_radians),np.cos(rotation_radians)]]

for root, dirs, files in os.walk("."):
    for file in files:
        data = np.loadtxt(file, dtype=np.float32, delimiter=",")

        #Rotation
        data[:,:3] =  np.matmul(data[:,:3], A)
        data[:,2] = -1 * data[:,2]

        #Axes switching
        x = data[:,0].copy()
        y = data[:,1].copy()
        data[:,1] = x
        data[:,0] = y

        #Cropping
        rowIndex = 0
        array = []
        for row in data:
            #Crop if x > 10.62m or z > 0.5m or z < -2.3m or |y| > x - (90 degree FOV)
            if (row[0] > 10.62 or row[2] > 0.5 or row[2] < -2.3 or abs(row[1]) >
                row[0]):
                array.append(rowIndex)
            rowIndex += 1
        data = np.delete(data, array, axis=0)

        #File renaming
        Number = file.replace("CaptureFrame","")
        Number = Number.replace(".csv","")
        newFileName = Number.zfill(6) + ".txt"

        #File Saving
        np.savetxt(newFileName, data, delimiter=",")
        print("output:", newFileName)
```

Figure 7: Background removal and axes orientation

The data loaded from the files for the input of this script are in the form [x,y,z,intensity]. Rotation is applied because the field data was obtained using the sensor at a 45˚ angle from horizontal. The transformation rotation matrix A is used to rotate the point cloud. Furthermore, the point cloud is cropped into a 90 degree FOV by removing all points which satisfy the condition abs(y) > x in order to obtain a similar FOV as the VoxelNET model (Figure 8).

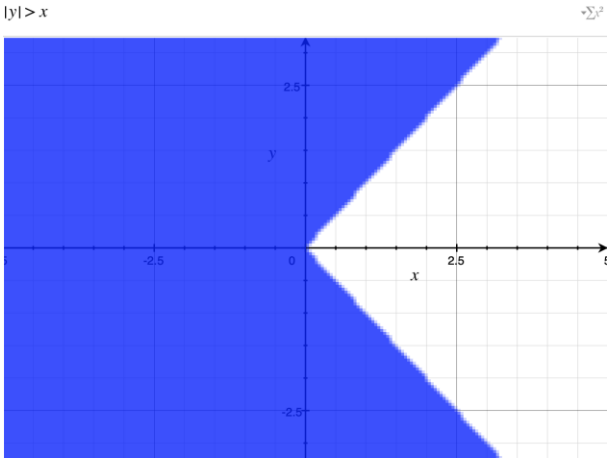


Figure 8: Blue region showing cropped data where |y| > x.forward direction is positive x-axis.

Another approach to remove the background which was initially attempted included capturing a "background" frame with the scene and no pedestrians. Every scanned frame was then compared to this background frame by comparing every point in the background to every point in the newly scanned frame. If a point already existed in the background, it is deleted (Figure 9). This method is not very efficient and hence, the processing time took too long for it to be feasible for real-time applications. Unfortunately, the RS-BPearl sensor starts recording each frame at inconsistent azimuth values (starting at slightly different values close to zero degrees every scan) which means that any background comparison must account for this shift and any missing points which could again shift the data. A sorting algorithm could be used to align the azimuth values of the background data and incoming frames, such that an efficient matrix comparison can be done instead.

```
#Check frameData against backgroundData
def removeBackground(frameData):
    #Array to store final result
    foregroundData = []
    #Array to store indices of background points to delete
    backgroundIndexList = []
    #Numerical tolerance values for distance, azimuth and omega
    filter = numpy.asarray([distanceTolerance, azimuthTolerance, omegaTolerance])
    #Compare every point in frame to every point in background
    for point1 in frameData:
        for point2 in background:
            #Compare the absolute difference of all three quantities
            if ((abs((point1-point2)) < [distanceTolerance, azimuthTolerance,
                omegaTolerance, 100]).all()):
                #If the point is a backgroud point, save its inedex
                backgroundIndexList.append(point1)
    #Delete all background points using their indices
    foregroundData = numpy.delete(frameData, backgroundIndexList, 0)
    #Return the array of foreground data
    return numpy.asarray(foregroundData)
```

Figure 9: Simple point comparison background removal function.

Error! Reference source not found. illustrates the result after applying the background removal and axes orientation and Figure 11 contains pictures of the experimental set up. Due to lockdown travel restrictions, the experiments were performed on a farm.

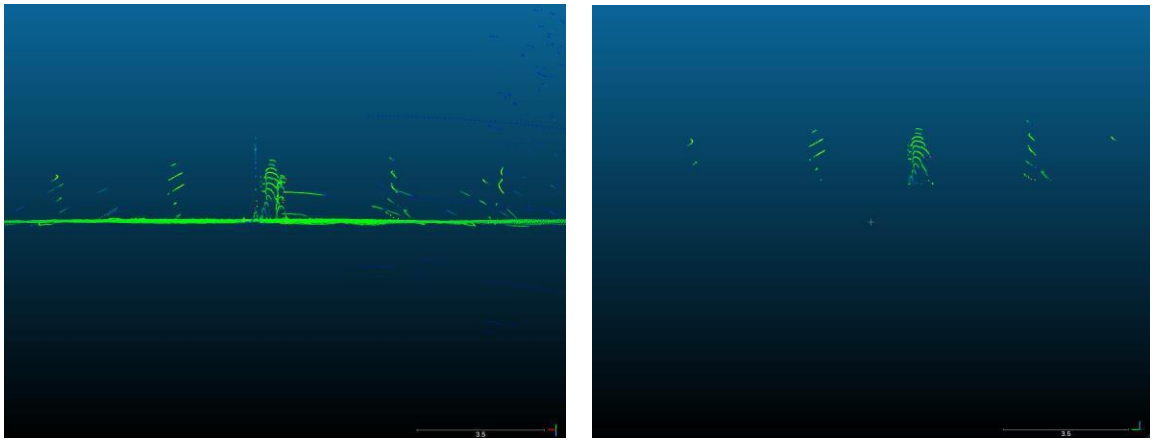


Figure 10: Point cloud frame (Capture 2 -> Frame 69) before and after cropping to VoxelNET FOV and background removal.

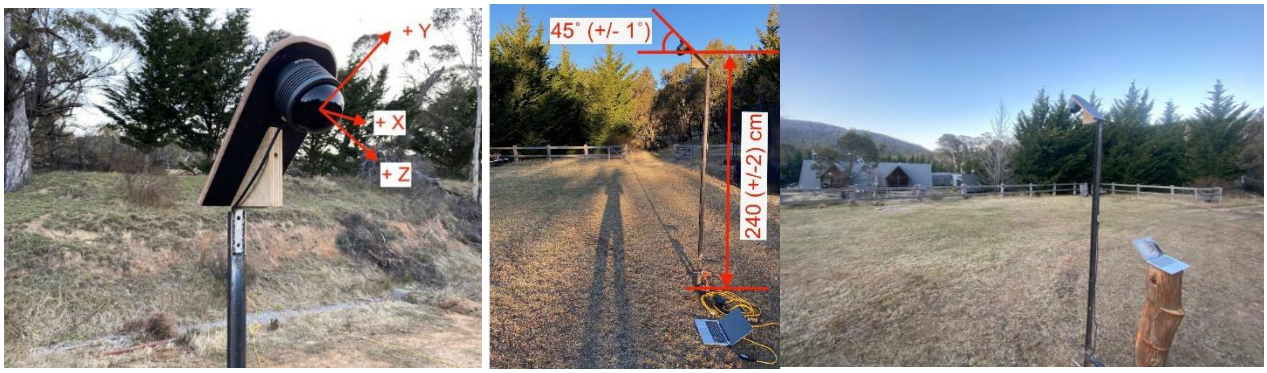


Figure 11: Paddock experiment set-up

Voxelnet

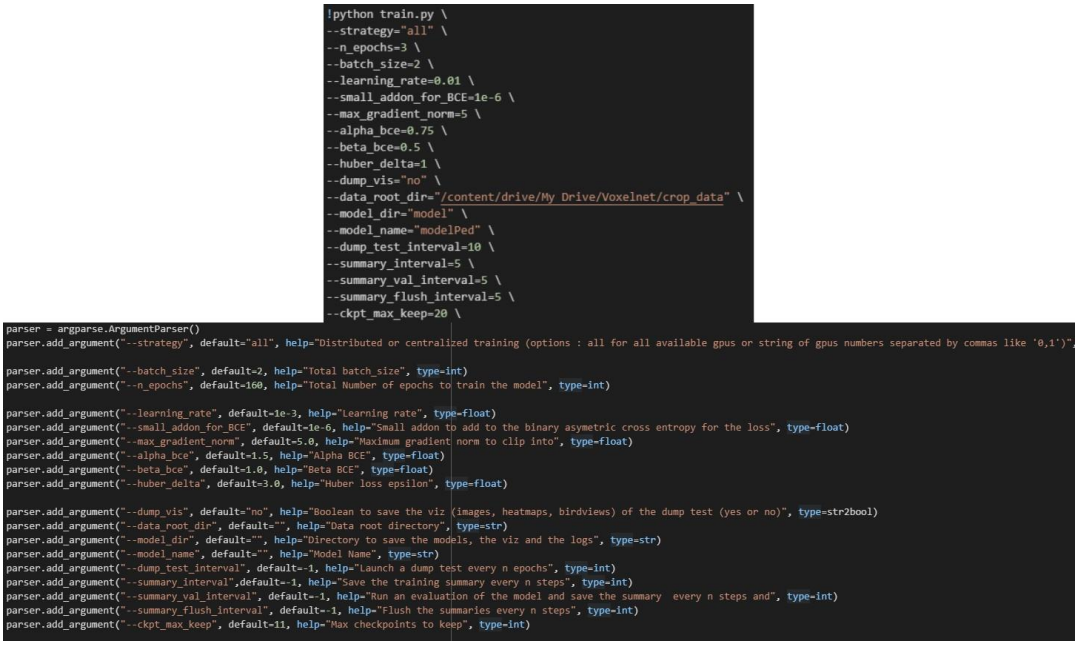
For the Machine learning several models were investigated. The selection of VoxelNET (https://www.mining3.com/research/voxelnet/) is the result of trial and error of different point cloud processing techniques, machine learning methods and deep learning models. Since the project is about the detection of pedestrians for the purpose of computing density, the ability to classify whether a cluster of the point cloud is a pedestrian or not is important. There is a wide range of existing methods, some of them are purely based on point cloud processing and statisticalanalysis (Scheunert et al 2004, Zhao et al 2005, Xiao et al 2016,), whilst some are based on the supervised training of a machine learning or deep neural network (Li et al 2009, Mozaffari et al 2020).

We initially experimented with a variety of traditional machine learning methods, such as Support Vector Machines (SVM) and K-nearest neighbour clustering, however, the results were not satisfactory due to the lack of features from the training data and required further feature engineering to better improve the result, due to the time limit, we instead change to a machine learning approach.

We then concentrated on two machine learning approaches which are Voxelnet and PointPillars. Several attempts were made to adapt the two Github repository codes to our application, however, only VoxelNET was successfully tailored. PointPillar is still an option but requires some more effort to train the network compared to that of Voxelnet.

Scripts and the hyper-parameters for training

The script for running prediction and training has already been documented in the workstation, as well as which folder the training and prediction files should be put into alongwith the instructions. The parameters for the training are shown in Figure 12.



```
!python train.py \
--strategy="all" \
--n_epochs=3 \
--batch_size=2 \
--learning_rate=0.01 \
--small_addon_for_BCE=1e-6 \
--max_gradient_norm=5 \
--alpha_bce=0.75 \
--beta_bce=0.5 \
--huber_delta=1 \
--dump_vis="no" \
--data_root_dir="/content/drive/My Drive/Voxelnet/crop_data" \
--model_dir="model" \
--model_name="modelPed" \
--dump_test_interval=10 \
--summary_interval=5 \
--summary_val_interval=5 \
--summary_flush_interval=5 \
--ckpt_max_keep=20 \
```

```
parser = argparse.ArgumentParser()
parser.add_argument("--strategy", default="all", help="Distributed or centralized training (options : all for all available gpus or string of gpus numbers separated by commas like '0,1')"

parser.add_argument("--batch_size", default=2, help="Total batch_size", type=int)
parser.add_argument("--n_epochs", default=160, help="Total Number of epochs to train the model", type=int)

parser.add_argument("--learning_rate", default=1e-3, help="Learning rate", type=float)
parser.add_argument("--small_addon_for_BCE", default=1e-6, help="Small addon to add to the binary asymetric cross entropy for the loss", type=float)
parser.add_argument("--max_gradient_norm", default=5.0, help="Maximum gradient norm to clip into", type=float)
parser.add_argument("--alpha_bce", default=1.5, help="Alpha BCE", type=float)
parser.add_argument("--beta_bce", default=1.0, help="Beta BCE", type=float)
parser.add_argument("--huber_delta", default=3.0, help="Huber loss epsilon", type=float)

parser.add_argument("--dump_vis", default="no", help="Boolean to save the viz (images, heatmaps, birdviews) of the dump test (yes or no)", type=str2bool)
parser.add_argument("--data_root_dir", default="", help="Data root directory", type=str)
parser.add_argument("--model_dir", default="", help="Directory to save the models, the viz and the logs", type=str)
parser.add_argument("--model_name", default="", help="Model Name", type=str)
parser.add_argument("--dump_test_interval", default=-1, help="Launch a dump test every n epochs", type=int)
parser.add_argument("--summary_interval",default=-1, help="Save the training summary every n steps", type=int)
parser.add_argument("--summary_val_interval", default=-1, help="Run an evaluation of the model and save the summary  every n steps and", type=int)
parser.add_argument("--summary_flush_interval", default=-1, help="Flush the summaries every n steps", type=int)
parser.add_argument("--ckpt_max_keep", default=11, help="Max checkpoints to keep", type=int)
```

Figure 12: Training parameters for VoxelNet

The “Strategy” parameter describes how the dataset will be distributed with the available GPUs in the PC or workstation. The distribution can be parallelised, meaning using multiple GPUstraining simultaneously, or centralised, meaning only use single GPU will be applied to perform training. The default is considered sufficient for the training.

The “Batch_size” parameter is defined as the number of training samples in each batch. In machine learning lingo, “an epoch” means when all the training samples have been consumed, and “Batch_size” means the number of training samples on the GPU for one forward and backward pass of the training model. The equation for calculating the number of required steps to consume all the training samples is defined in Eq. (2).

$$Number\ of\ steps = \frac{Total\ number\ of\ training\ samples}{Batch\ size} \tag{2}$$

The “n_epochs” parameter, as mentioned above, is the number of times for all the training samples are passed through the model, for instance, 1000 training samples, 3 epochs. A batch size of 2 means that the number of required steps for 1 epoch is 1000/2 = 500 steps, which means, after 500 steps, all the training samples will have already been consumed and 1 epochis completed, and 2 more epochs are required as we set the number of epochs to be 3.

The “learning_rate” parameter is defined as the rate at which the weights of the model is updated during the training, as the model learns by updating the weights to maximise the probability of producing the correct output. As mentioned in the VoxelNET paper (Zhou 2017),Stochastic Gradient Descent(SGD) optimiser is used for pedestrian training and it is also applied in this project. The learning rate is defined by the symbol η, the weight of the model is defined as w, and the loss function is defined as E, the weights updating rule is defined in Eq. (3), the SGD works by taking the number of training samples defined by the batch size asone update of the weight of the model, this process is repeated until the number of epochs is reached or training is interrupted.

$$w = w - \eta \frac{\delta E}{\delta w} \tag{3}$$

Typically, different loss functions will have different effects on the model during the training.The “small_addon_for_BCE” parameter is a small value that is added to the BCE (Binary Cross- Entropy) loss function. The original paper did not mention anything related to this addon value, for this project, the default value is more than enough.

The “max_gradient_norm” parameter defines the maximum norm for the vector gradient calculated from the loss function which can not be exceeded during the training of the model. Normally this value has to do with the gradient optimiser and is decided by user choice. he “alpha_bce”(α) and “beta_bce”(β) parameters refer to the coefficient in the loss functiondesigned by the author of VoxelNet. The loss function E defined by the Voxlent is described in Eq. (4).

$$E = \alpha \frac{1}{N_{pos}} \sum_i E_{cls}(p_i^{pos}, 1) + \beta \frac{1}{N_{neg}} \sum_j E_{cls}(p_j^{neg}, 0) + \frac{1}{N_{pos}} \sum_i E_{reg}(u_i, u_i^*) \quad (4)$$

Equation (3) defined the loss function for training the Voxelnetl model. The object of interest (OOI), such as pedestrian, has the ground-truth value of 1 and 0 for objects that are not interested. The loss function E consists of normalised classification loss function E_{cls} and regression loss function E_{reg} which are trained simultaneously. E_{cls} is for classifying the 3D bounding boxes. p_i^{pos} and p_j^{neg} are the softmax output, which is the output from an activation function in Machine Learning lingo, for positive and negative anchors respectively. The α and β are constants for the relative importance of the classified object, which are application dependent, from the sample values shown in Figure 9, α is higher and more important than β as we focused more on the pedestrians. N_{pos} and N_{neg} are the number of positive and negative anchors for Region Proposal Network(RPN).

The “huber_delta” is a value that is used to determine the loss for the regression loss function E_{reg}, in the implementation, the Huber loss function is used as the regression loss function, which is defined in Eq. (5).

$$E_{reg}(u_i, u_i^*) = \begin{cases} (u_i - u_i^*)^2 \; if \; |u_i - u_i^*| \leq huber\_delta \\ |u_i - u_i^*| \; if \; |u_i - u_i^*| > huber\_delta \end{cases} \quad (5)$$

u_i and u^* are the residual vector and its ground truth for training, from the original paper, $u_i, u^* \in \mathbb{R}^7$ contained 7 regression target values corresponding to the difference for x, y, z, l, w, h, θ, which are the centre location of the 3D bounding boxes (x, y, z), the length, width and the height of the 3D bounding boxes (l, w, h) and the yaw rotation angle (θ) around Z-axis. Note that most of these values are already defined in the implementation. For training, only parameters that are exposed to the user, as shown in Figure 12, can be adjusted.

The “dump_vis” parameter is defined as a Boolean value to save the visualisation for the dump test, the provided Github codebase is designed such that the model is validated at a certain interval and visualisation of the validated output will be produced and the user can choose to save depending on this parameter.

The “data_root_dir” parameter is the directory in which all the training and validating samples reside. However, the training script “train.py” never used this parameter for some reason.

The “model_dir” parameter is the directory into which the trained model will be saved. This directory contains the visualisation for the model prediction and the training logs for experimenting with different training methods and training related hyper-parameters.

The “model_name” parameter is the name of the model that is trained by executing the command “!train.py”, this parameter is used to distinguish different models and comparisonsamongst different models can be made.

The “dump_test_interval” parameter defines the interval for which the validation is performed, according to the instruction, validation of the model is performed at the numberof epochs defined by this parameter.

The “summary_interval” parameters define the interval for saving the training log into the “model_dir” directory, this is useful as later on training log can be used for experimenting with different configurations for the training of the model. According to the script “train.py”,the interval refers to the number of steps, and the training summary is logged after a certainnumber of predefined values for the parameter.

The “summary_val_interval” parameter refers to the interval for validation and logged summary of the training. According to the script “train.py”, the interval is defined as a numberof steps, similar to “summary_interval”.

The “summary_flush_interval” parameter refers to the interval for flushing out the training summary of the training. The word “Flush” here refers to saving the summery into the folder.

As the python library which the script uses is Tensorflow 2.0 and it is capable of flushing out the history of training summary at a prefined interval to any buffer for storage. Theoretically, “summary_interval” defines how frequent the summary is stored into the buffer, and “summary_flush_interval defines how frequent the buffer is flushed out for storage.

The “ckpt_max_keep” parameter is defined as the maximum number of checkpoints for the training of the model should keep during the training. Tensorflow is capable of continuing thetraining from the last checkpoint it is left over.

Note that the “model_dir” is the directory in which the model is stored, the “model_name” is also important and it is used to distinguish which model to use for the prediction.

Additionally, to be able to detect more pedestrians, a hyper-parameter called “RPN_NMS_POST_TOPK” in “config.py” is adjusted. This parameter is set to be 20 initially, which we changed to a higher value (e.g., 50, 60) for detecting more pedestrians. The name of the parameter came from Region Proposal Network(RPN)_Non-Maximum Suppression(NMS)_POST_TOPK(top k 3D bounding boxes).

Overall, the adjustable parameters are shown in Figure 12 and the explanation for each parameter is provided above. We have to adapt the parameters “n_epochs”, “batch_size”, “learning_rate”, “max_gradient_norm”, “alpha_bce”, “beta_bce” and “huber_delta”, these are training related, the rest are for file storage. The suggested values are shown in Figure 12.

Creating synthetic data

There are several disadvantages of using labelled data based on real scanning to create a ML model that would give you realistic results in different scenarios, which are as follows:

- Costly process – manual work
- Imperfectly labelled data
- Slow process
- Limited scenarios diversity
- Not able to detect pedestrians from a few points

Using synthetic data gives you the freedom to model scenarios and data (i.e., point clouds) collection process in the way you would like. Moreover, we can resolve all issues that are presented above. This allows us to improve results in the detection of objects, in our case pedestrians.

To collect point clouds data, we use Unity game engines. To collect synthetic data in Unity, it is important to have pedestrian simulation and point clouds scanning process.

For the pedestrian simulation, we use the most prominent models are based on social forces (Helbing & Molnar, 1995). This model can recreate certain phenomena such as queue formation and arching, shock waves and bottleneck effect. The model integrates into pedestrians’ movements the self-driven force $\vec{f}_i$, the forces by other pedestrians $\vec{f}_{ij}$, and $\vec{f}_{io}$ represents the force by obstacles such walls, pillars, furniture, etc. The total force $\vec{f}_i$ applied on a pedestrian can be formulated as Eq. (1).

$$\vec{f}_i = \frac{\vec{v}_i^{des} - \vec{v}_i}{\tau} + \sum_{i \neq j} \vec{f}_{ij} + \sum_{io} \vec{f}_o \quad (6)$$

Regarding the data collection process, we use the raycasting function that Unity provides to detect objects in a specific direction. We cast uniformly rays in the same way as a scanner would do (Figure 13).

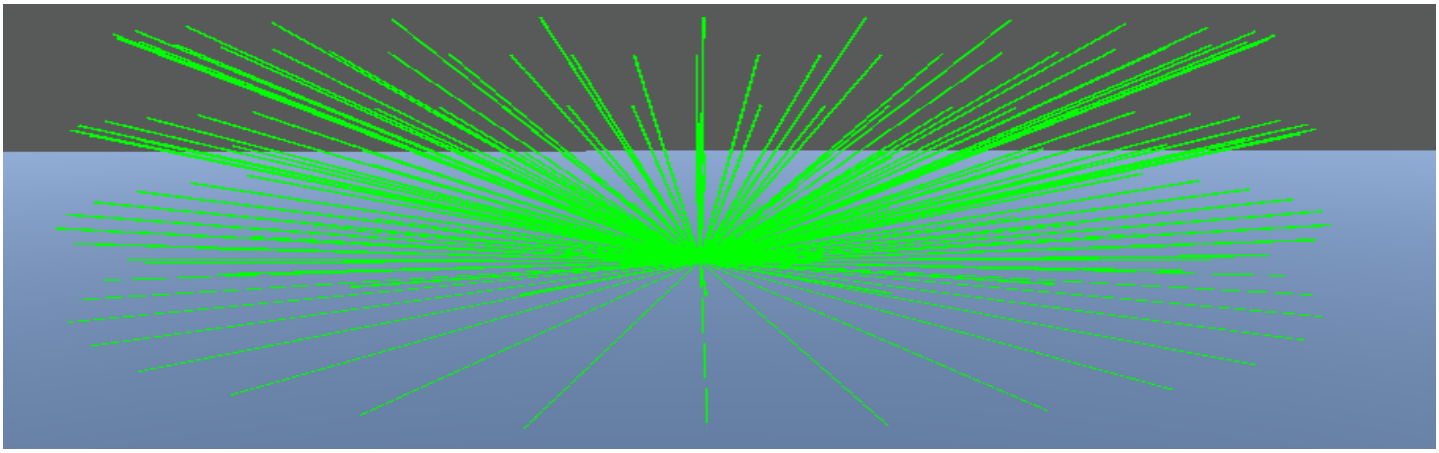


Figure 13. Scanning process in Unity from one point

We can place the scanner at any given place in the scene and start detecting points that hit animated pedestrians (Figure 14). The important part is that pedestrians can be in different postures, and we can detect diverse point clouds of them. Point clouds can clearly resemble the pedestrian body shapes, but not more than this (Figure 15).

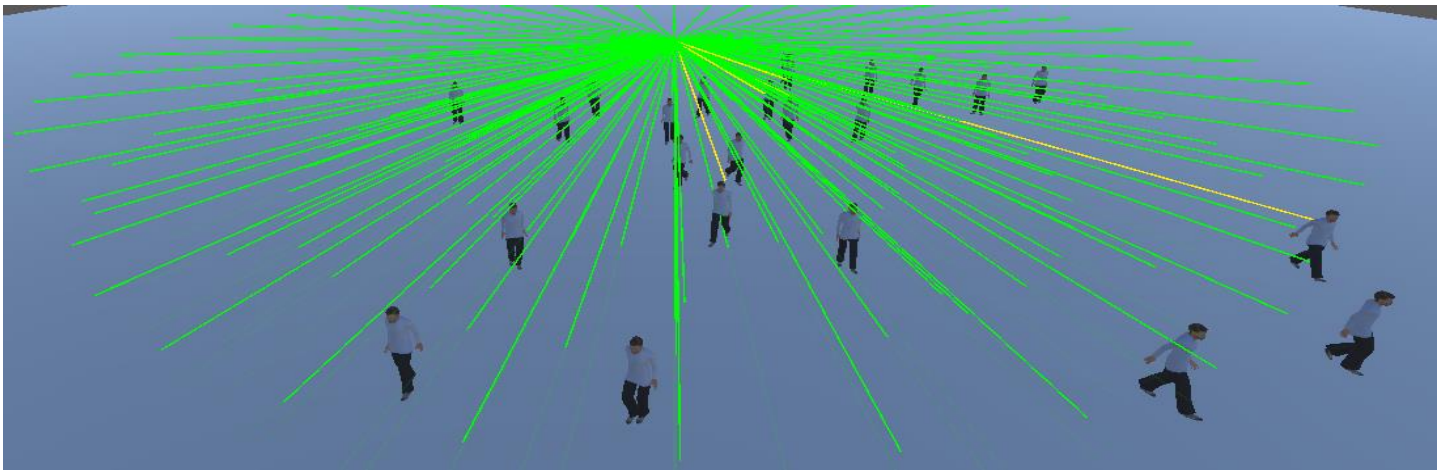


Figure 14. Point clouds detection using pedestrian simulations

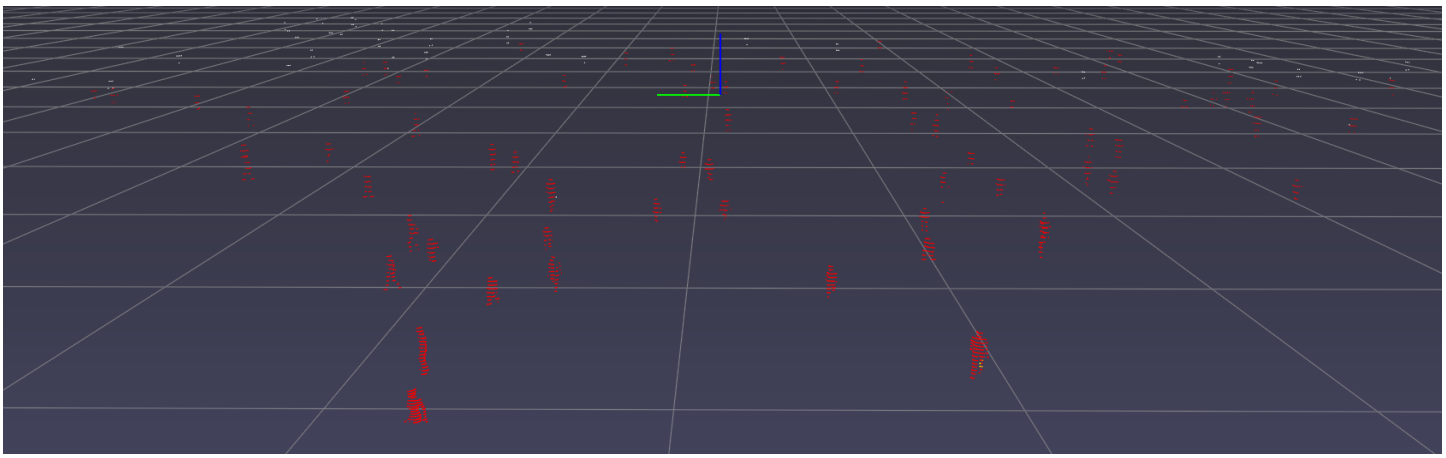


Figure 15. Point clouds of pedestrians

Apart from the point clouds of pedestrians, we need to calculate the 7 parameters which VoxelNet requires. Thus, we need to identify a centroid and bounding box for each pedestrian as well as the rotation of the box. We use all vertices representing an agent in the scene to identify better the bounding box, which will also give us the centroid. For the rotation, we use the direction of movement of pedestrians (Figure 16). In this way, we can almost perfectly identify bounding boxes and their properties representing pedestrians, which is impossible if the work is done manually.

Figure 16. Rotated boxes and rays hitting pedestrians

We need to mention that we can collect 1000 datasets in 1 hour which is way quicker than what is possible manually.

Setup, timing and training results

Setup

There are 5 synthetic datasets and 1 KITTI dataset with all the pedestrians extracted from theoriginal dataset for training, some of them need to have the axis swapped. As shown in Figure 17, the Z-axis in the point cloud coordinate is the negative Y-axis in the camera coordinate, the Y-axis in the point cloud coordinate is the negative X-axis in camera coordinate etc. The input data follows the KITTI data format. The imported 3D point cloud is unstructured, which is an N-by-4 array with each point havingx,y,z, intensity, the array is packed in ".bin"(binary format) and in little-endian.

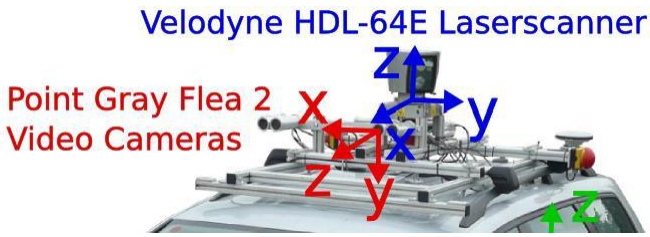


Figure 17: Axis for point cloud and camera coordinate frame

The output isthe predicted number of pedestrians given the 3D point cloud in KITTI format. Sample predicted output and the definition of each column from left to right is shown in Figure 18.

```
Pedestrian 0.0000 0.0000 0.0000 221.0000 162.0000 245.0000 219.0000 1.8986 0.4407 0.7711 -12.6334 1.4942 24.1367 1.4806 1.0000
Pedestrian 0.0000 0.0000 0.0000 283.0000 169.0000 311.0000 232.0000 1.7923 0.6610 0.9094 -9.1211 1.6455 21.0160 1.2275 0.9993
Pedestrian 0.0000 0.0000 0.0000 228.0000 171.0000 259.0000 222.0000 1.6711 0.6162 0.8490 -12.1851 1.5623 23.9570 1.5516 0.9988
Pedestrian 0.0000 0.0000 0.0000 549.0000 167.0000 582.0000 226.0000 1.7748 0.6879 0.9288 -1.3619 1.5693 22.2030 1.1054 0.9978
Pedestrian 0.0000 0.0000 0.0000 383.0000 169.0000 413.0000 228.0000 1.7796 0.6856 0.9134 -6.5448 1.6182 22.2617 1.0722 0.9974
Pedestrian 0.0000 0.0000 0.0000 206.0000 164.0000 236.0000 222.0000 1.8566 0.6106 0.9405 -12.6398 1.5291 23.4308 1.3744 0.9962
Pedestrian 0.0000 0.0000 0.0000 404.0000 163.0000 437.0000 232.0000 1.8349 0.6868 0.9260 -5.1405 1.5531 19.6257 1.5130 0.9943
```

Num elements	Parameter name	Description	Type	Range	Example
1	Class names	The class to which the object belongs.	String	N/A	Person, car, Road_Sign
1	Truncation	How much of the object has left image boundaries.	Float	0.0, 0.1	0.0
1	Occlusion	Occlusion state [0 = fully visible, 1 = partly visible, 2 = largely occluded, 3 = unknown].	Integer	[0,3]	2
1	Alpha	Observation Angle of object	Float	[-pi, pi]	0.146
4	Bounding box coordinates: [xmin, ymin, xmax, ymax]	Location of the object in the image	Float(0 based index)	[0 to image width],[0 to image_height], [top_left, image_width], [bottom_right, image_height]	100 120 180 160
3	3-D dimension	Height, width, length of the object (in meters)	Float	N/A	1.65, 1.67, 3.64
3	Location	3-D object location x, y, z in camera coordinates (in meters)	Float	N/A	-0.65,1.71, 46.7
1	Rotation_y	Rotation ry around the Y-axis in camera coordinates	Float	[-pi, pi]	-1.59

Figure 18: Sample predicted output from Voxelnet and the KITTI format (Preparing the Input Data Structure — Transfer Learning Toolkit 2.0 documentation 2011)

Four folders are important for the "training" and "validation" subfolder separately, which are "calib", "image_2", "label_2" and "velodyne".

The "calib" folder contains all the calibration files for each training sample as later this file can be used for visualisation purposes. The calibration file needs to be carefully constructed. A sample calibration file for the synthetic data is shown in Figure 199. The relevant parameters are "P2", "R0_rect" and "Tr_velo_to_cam", "P2" is a 3-by-4 projection matrix that projects a point in the rectified reference camera coordinate(camera-0) onto the camera-2 image coordinate, "R0_rect" is the rotation matrix to rectify the rotation for the referenced camera coordinate such that multiple images lie on the same plane, and "Tr_velo_to_cam" is a projection matrix which maps a point in the 3D point cloud coordinate to reference coordinate (camera-0).

```
P0: 1.046457 0 0 0 0 3.48819 0 0 0 0 1.0050125 0.60150375
P1: 1.046457 0 0 0 0 3.48819 0 0 0 0 1.0050125 0.60150375
P2: 1.046457 0 0 0 0 3.48819 0 0 0 0 1.0050125 0.60150375
P3: 1.046457 0 0 0 0 3.48819 0 0 0 0 1.0050125 0.60150375
R0_rect: 1 0 0 0 1 0 0 0 1
Tr_velo_to_cam: 0 -1 0 0 0 0 -1 0 1 0 0 0
Tr_imu_to_velo: 1 0 0 0 0 1 0 0 0 0
```

Figure 19: Sample calibration file

The "image_2" folder contains all the images captured by camera number 2, according to theKITTI setup. Later on, the 2D boxes can be projected onto the image, however, this is not applicable as 3D boxes are predicted from the network so we can only visualise the labelled training data but not predicted data.

The "label_2" folder contains all the labels for the Object of interest (OOI) in one frame or training sample which defines what are the positive and negative objects with respect to thenetwork, this has to follow the KITTI format as mentioned above.

The “velodyne” folder contains all the 3D point clouds which are taken as the input to the VoxelNet, note that all the 3D point clouds should be converted to .bin (binary) and in little- endian format. Note that the number of files in each of the four folders should be identical, meaning the number of files in “calib” should be equal to “label_2”, “image_2” and “velodyne”.

The setup instructions for training has been documented in a Jupyter notebook on the workstation, the timing for training the model depends on the number of epochs passed in when running the “!python train.py” command.

Timing

There are 5 synthetic datasets and 1 KITTI dataset with all the pedestrians extracted from theoriginal dataset. The split into training and validation is 80% and 20%. That is, roughly 932 training samples and 233 validation samples. This ratio could be differentdepending on the quality of the training sample. As a rule of thumb, a ratio of 9:1, 8:2, 7:3, 6:4 is good for training the model. The more the training samples, the longer the training willtake in each epoch.

Normally, 8 - 14 hours of training time will be the minimum required for the training to be successful for 1165 samples. These estimates are for the KITTI dataset with pedestrians extracted from thetraining samples. For the synthetic data created with Unity, 6 – 12 hours of training is required for the training to be successful for 1165 samples, because the synthetic data has fewer points than the KITTI dataset. However, this is just empirically determined, the actual timing will depend on the number of training samples, number of epochs, number of batches and number of GPUs that is currentlyavailable.

The timing of training the model also depends on the tolerance range of a specific application,for instance, in this project. A classification error of 0.1378 and a regression error of 0.6867is reached after 700 steps, which can be considered as a good indicator of the model is trainedsuccessfully.

Training results and tests with the real scans

The synthetic datasets have difference in the number of points to be considered as one pedestrian with similar environmental complexity. This is related to the automated labelling process. In the last synthetic dataset, the number of labelled pedestrians are significantly increased compared to the previous datasets to test whether the model can detect more pedestrians. KITTI dataset is also used for training and performing testing on the synthetic dataset to test the robustness of the training.

The training result of the VoxelNet model on the synthetic dataset is shown in Figure 19Figure 20. Note that the curves are smoothed out for better visualisation. Two important metrics are “train/cls_loss” and “train/reg_loss”, the former is the weighted average classification loss from the first two terms in Eq. (3). The classification loss gradually dropped after 250 steps from around 0.865 to 0.845, the regression loss has some fluctuations and a little bit of rising at the beginning of the training, subsequently slowly decreased from 0.18 to 0.14.

However, the “cls_neg_loss” has an unstable tendency, which increases from 0.7 to 0.8 indicating the network may be erroneous in classifying the background due to crowded scenes. This behaviour is yet to be explored further. Even if the unexpected behaviour of “cls_neg_loss”, the overall loss “train/loss” is still decreased from 1.03 to 0.99.

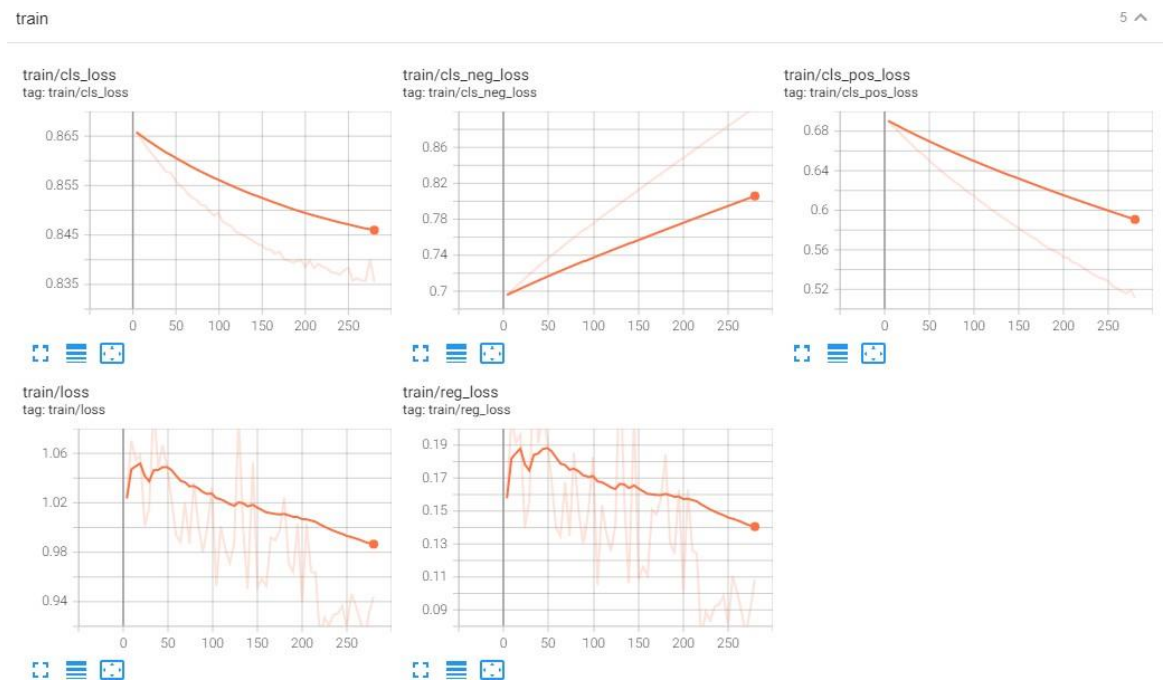


Figure 20: Training results for the synthetic data set

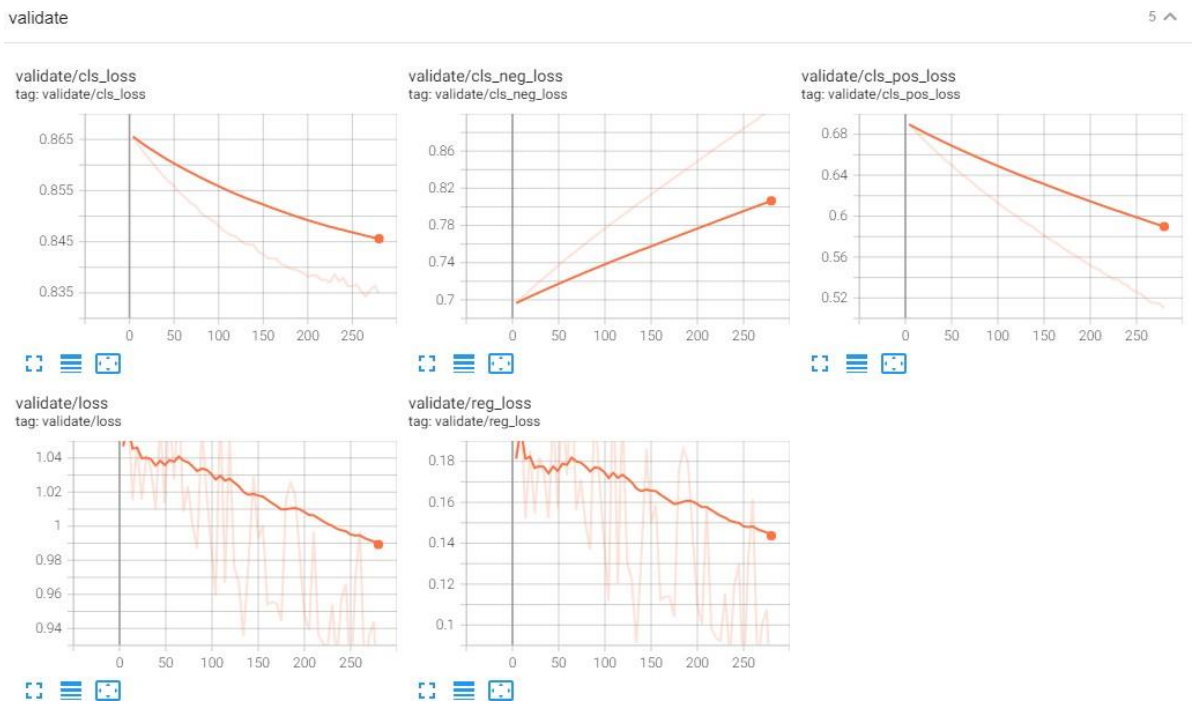


Figure 21: Validation result for synthetic dataset

The validation result of the VoxelNet model on the synthetic dataset is shown in Figure 21. Similar to the training result, the two important metrics are "validate/cls_loss" and "validate/reg_loss". As expected, they both have a similar response to that of the corresponding "train/cls_loss" and "train/reg_loss", as they are evaluated in every 5 steps, according to Figure 12. The "cls_neg_loss" is also unstable, which increases from 0.7 to 0.8, this phenomenon is similar to "train/cls_neg_loss" due to the crowded scenes in the dataset,meaning the network has difficulties classifying the background. The overall loss "validate/loss" is similar to "train/loss" which drops from 1.04 to 0.98, note that the loss wasstill decreasing after 250 steps.

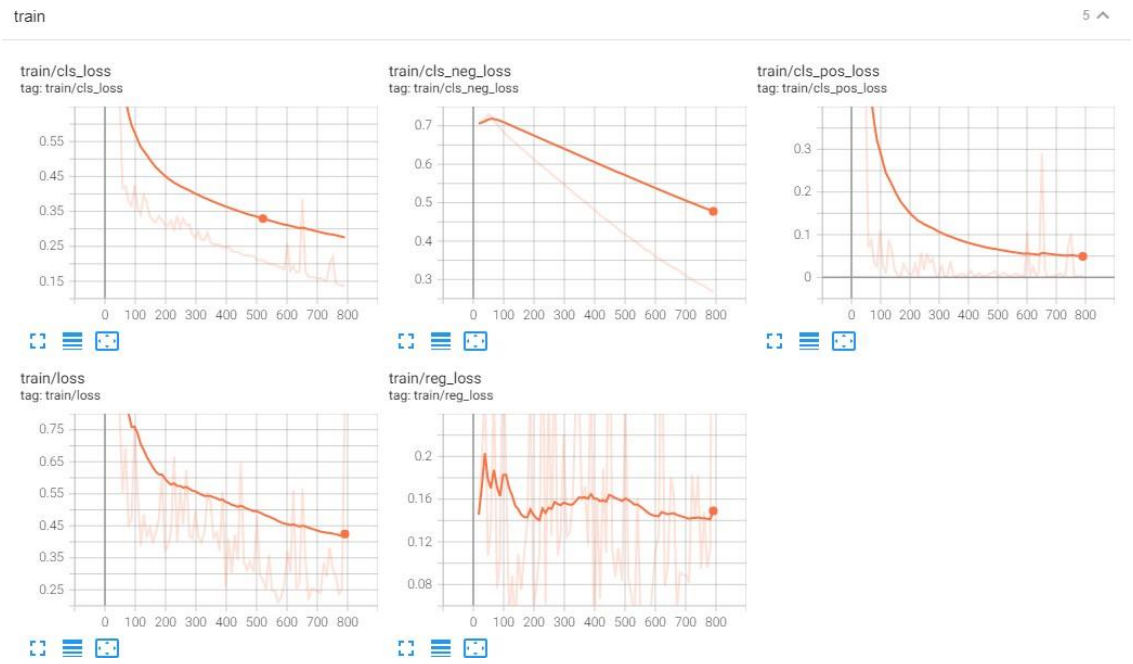


Figure 22: Training result for KITTI dataset

The VoxleNET model is also trained using the KITTI dataset with all thepedestrians extracted. The training result for the KITTI data set is shown in Figure 22. Similarto the training result for the synthetic dataset, “train/class_loss” drops from 0.85 to 0.26 after 700 steps. Note that “train/cls_neg_loss” also drops as opposed to Figure 20, in which the loss is increased and is unstable. However, “train/reg_loss seemed to struggle to crossover the 0.14 baseline and have oscillatory behaviour. The probable reason is the limit of the error for the dataset or the configurations of the hyper-parameters shown in Figure 12. Overall, “train/loss” presents a healthy behaviour of a normal machine learning training curveand the results are expected and reasonable.

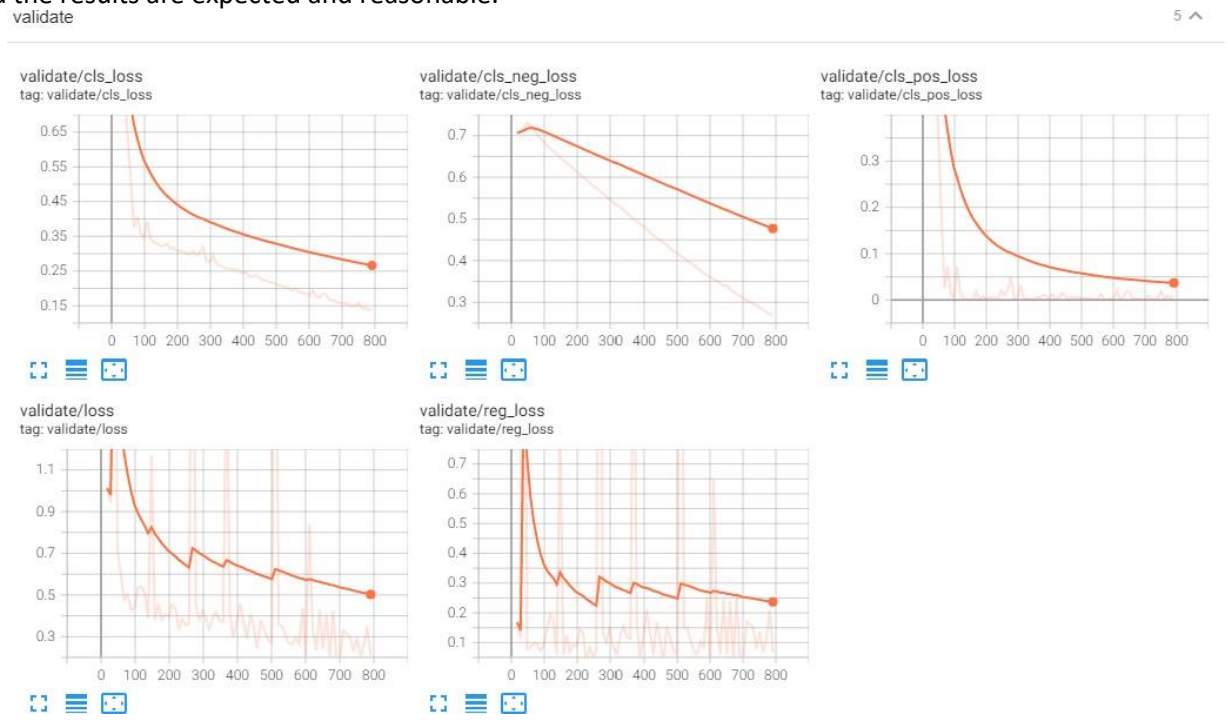


Figure 23: Validation result for KITTI dataset

The validation result for the KITTI dataset is shown in Figure 23. The validation curves are similar to that of the training

curves, except the “validation/reg_loss” which is a jigsaw-like shape. This is typically due to the biased in the validation dataset, as each time the network is trained with 932 samples but only evaluated on 2 or 3 validation samples, as indicated by thebatch_size. This is somewhat implied that the small portion of the validation sample is properly labelled and trusted by the network. Note that the visualisation tool is called tensorboard.

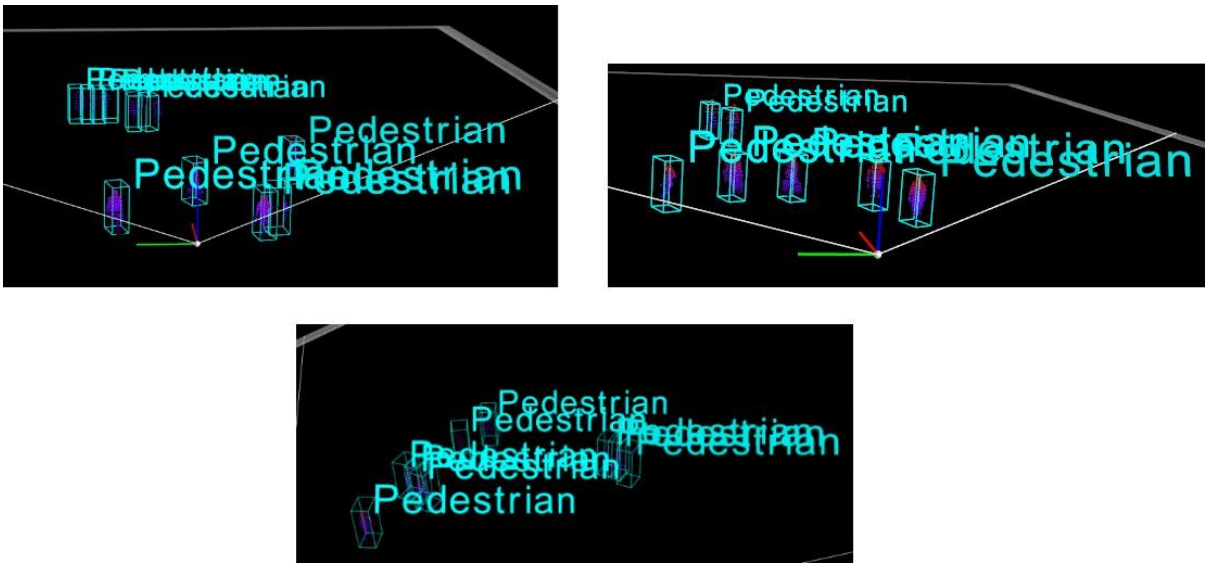


Figure 24: Sample predicted result from the trained VoxelNET model

Several tests against the synthetic and KITTI dataset have been performed after the training of the VoxelNET model. The results are shown in Figure 24 along with the 3D bounding boxes and the label. Some false positively detected pedestrians included in the result due to the occlusion and how the network is trained. Furthermore, the threshold “RPN_NMS_POST_TOPK” and training parameters also play an important when training the VoxelNet. As visible on Figure 24, occasionally more than one box is needed to cover one pedestrian. This effect comes from the value of “RPN_NMS_POST_TOPK” because the network tries to produce the top k number of 3D bounding boxes.

The input 3D point cloud for the prediction is from KITTI dataset and synthesised dataset. No tests have been completed with real data directly from RS-Bpearl.

Density computation

The density of people can be calculated using the following formula,

$$\text{Crowdedness} \propto \eta \frac{n}{A} \tag{7}$$

Where, n – number of pedestrians (boxes), η – normalisation coefficient, which depends on the area and A represents the specific area.

The number of “boxes” that are required in the visible area to be able to compute the density of people has to be greater than or equal to 0. This number is predicted through the Voxelnet,however, some checks on the predicted number should be performed before the density calculation such as an abnormal increase or decrease in a short period of time. However, this is considered to be application dependent. The software which is used for the density computation is written in Matlab script “densityCalculation.m”, some sample demonstrationsof the results are included in Figure 25.

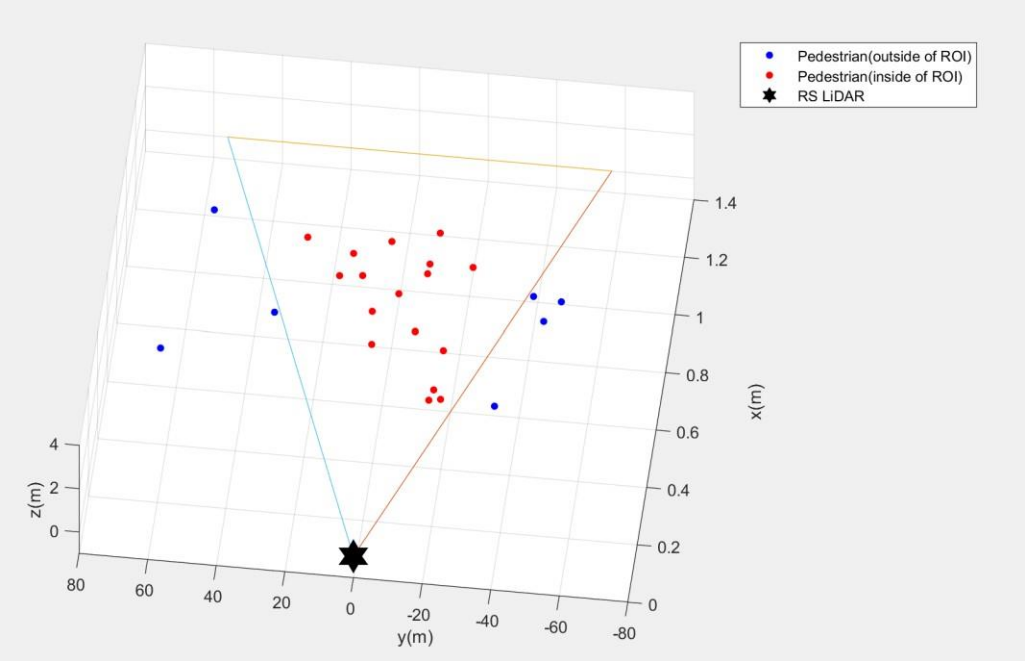


Figure 25: Visualisation of labelled training data

The labelled training data is visible in Figure 25. The FOV in this case is the triangular-like shape and is referred to as "Region Of Interest"(ROI), the red dots represent pedestrians that are currently within the FOV and included in the density calculation, whereas the blue dots represent those excluded. The heatmap that corresponds to Figure 25 is shown in Figure 26. The axes are in metres, the crowed the area is, more reddish colour it is indicated.

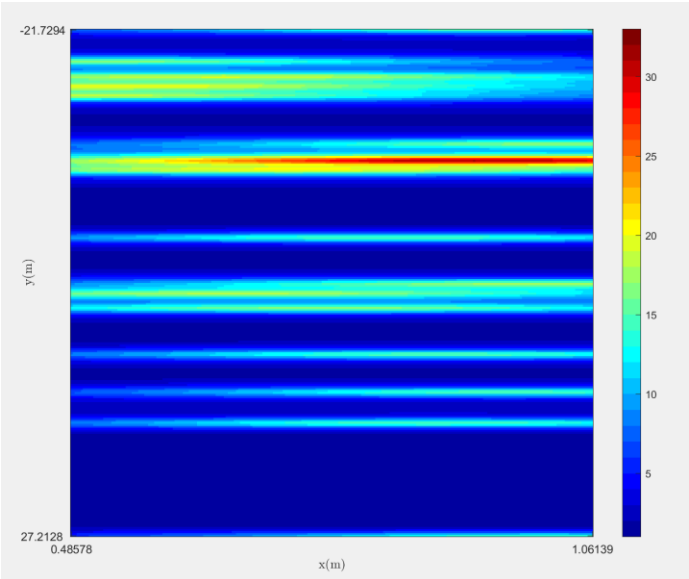


Figure 26: Heatmap of the crowdedness

A plot of the computed density over time is shown in Figure 27. A running average is maintained, and all historical data were plotted.

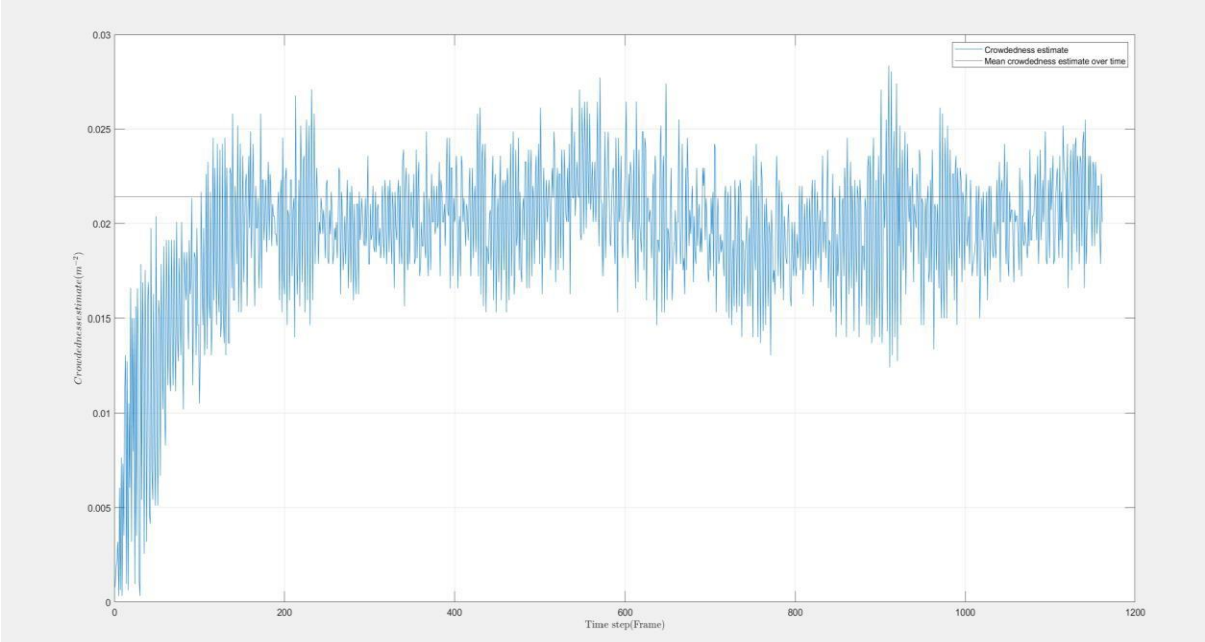


Figure 27: Density of crowd vs. time plot

Conclusions and recommendations

This project has experimented with several algorithms for detecting pedestrians using RS-Bpearl. Due to COVID-19 restrictions, the experiments are completed with existing training data sets and synthetic data (simulating the RS-Bpearl scanner). Nevertheless, we believe performed tests and investigations confirmed the proposed system architecture (Figure 1) is viable.

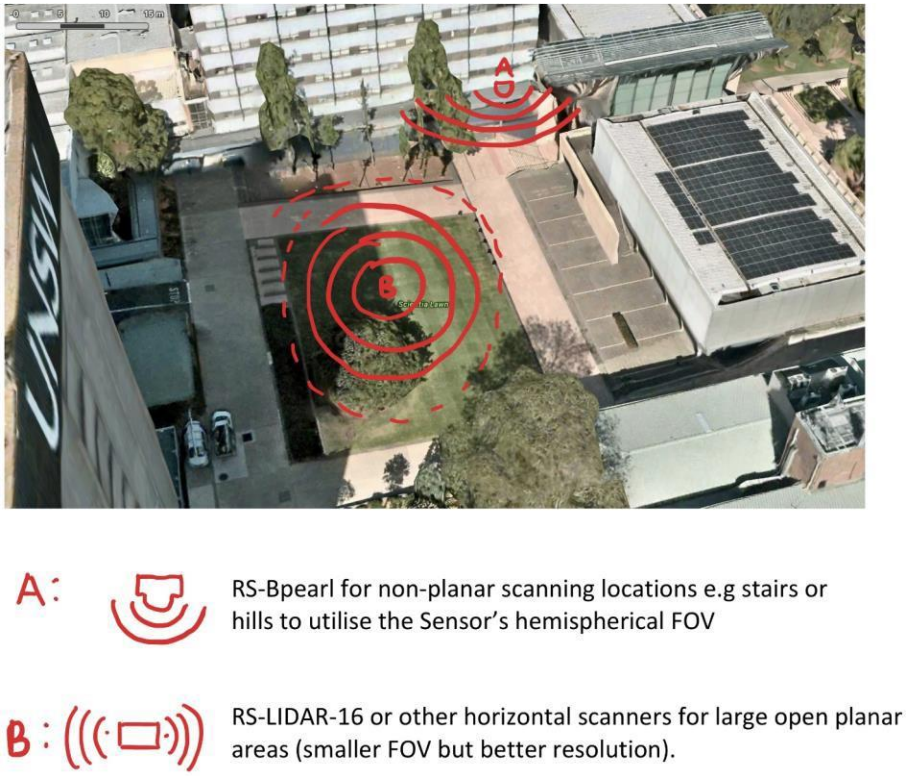


Figure 28: Example of using two sensors (A and B) to cover the open area at front of Red Centre.

System Architecture

The RS-Bpearl LiDAR sensor is currently the only sensor that we used and consider as input to the entire workflow. As mentioned previously the sensor has a limited range and therefore a better option will be to use several sensors to cover a larger space and produce more dense point cloud. It is also recommendable to investigate other scanners. For example, The RS-LIDAR-16 (Vertical FOV +/- 15˚) and/or RS-LIDAR-M1 (Vertical FOV +/- 12.5˚) could both be mounted a couple meters off the ground (above head height of people) in large open areas (such as the UNSW walkway) with their FOV in line with the ground plane. Figure 288 illustrates an configuration of two sensors for the area at from of the Red Centre The RS-Bpearl is more suited to walkways, stairs, hills or other obscure areas which make use of its wide FOV.

If more than one scanner is used, multiple RoboSense sensors could be connected using hard-wired ethernet cables or using internet routers for wireless data transfer from each sensor to a common computer. We have tested the WLAN communication by using an internet router to directly transmit the MSOP data from the B-Pearl sensor via the UDP (User Datagram Protocol) protocol which is utilised by the scanner. Figure 29 illustrates the LAN and WLAN connection that was tested with the sensor.

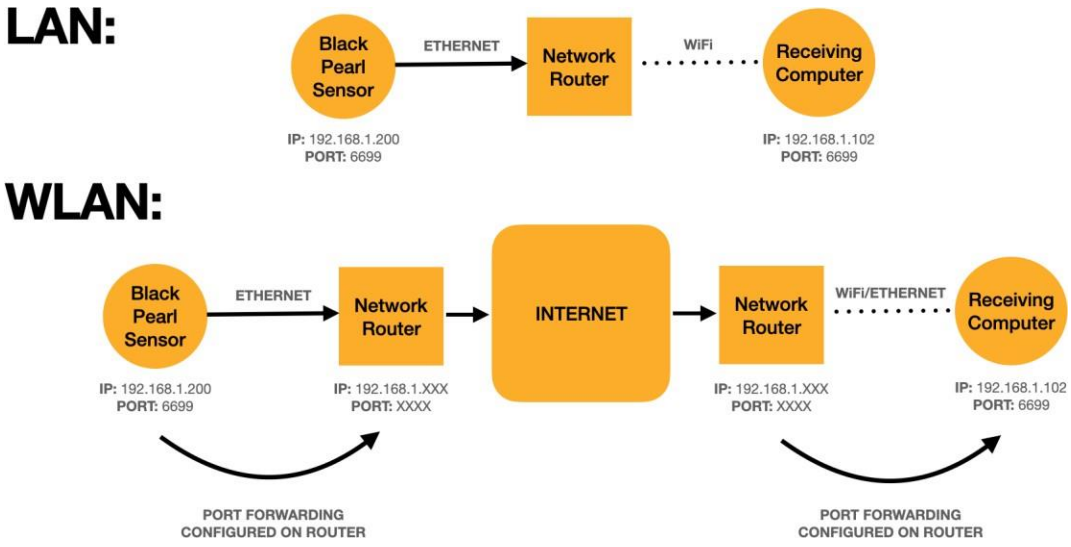


Figure 29: LAN and WLAN connection schematic

Although the tests are quite reliable for short distances, there was noticeable gaps in the received live point cloud feed. Lab tests have showed substantial packet loss when the receiving computer was in a different room (through multiple walls) or at a large distance to the internet router (>20m). Hence, for larger geographical areas, preprocessing of the point cloud data may be required to decrease the latency and increase the reliability of transmitted data. To achieve this, there are two main methods which could be realised to decrease the overall point cloud size, including downsampling and background removal. Downsampling can be achieved using a voxelisation of predefined size. Then only the centroid of the of the contained points is used instead of the original point cloud points. Alternatively, the background removal which has been discussed earlier in this report, could also be performed on the sensor side of the network before the point cloud data is transmitted. In this way, the overall point cloud size would be decreased from 2-3 Mb (for a full 57,600pts/frame with x,y,zand intensity in float32 format in ASCII) to only several kB (depending on how full the frame).

The two standard network protocols which could be utilised to transmit data over a network include TCP (Transmission control Protocol) and UDP (User Datagram Protocol). UDP is used by the scanner from factory because it is the faster and more lightweight of the two. UDP is a connectionless transport protocol. Each packet is only sent once and the receiver does not acknowledge a received message, meaning there is no gurantee that packets are received. On the contrary, TCP is a reliable and ordered transport protocol, which guarantees that the packets are transmitted in the correct order and without any loss. However, a lossy connection will significantly increase transmisson times, as many packets are resent if they are not properly received. If the point cloud is reduced in size prior to being transmitted,

messages may be sent with TCP more reliably over larger networks.

The second block in the systems architecture, i.e. ROS integration & Machine learning model and prediction, requires the integration of the trained VoxelNET model in to ROS. We have currently not been able to achieve this due to complications arising from the VoxelNET model being trained on GPU hardware whereas published implementations of the network in ROS require CPU hardware. The third component can be easily performed if the ROS the ML learning is running under ROS. The API is not realised with the time reference of this project, but can be easily implemented, given more tome.

It is strongly recommended to use ROS. The main benefit of using ROS is the direct use of the existing SDK for rslidar, the 3D point cloud from the 3D LiDAR is published to a configurable ROS topic and can be treated as an input to the proposed workflow or any other existing ROS packages or nodes. However, in case of being unfamiliar with ROS, since the 3D LiDAR utilises the UDP to transmit data, a python script or C++ file can be written and combined with the Point Cloud library to process all the incoming data packets by developing a network connection to the specified port number of the 3D LiDAR.

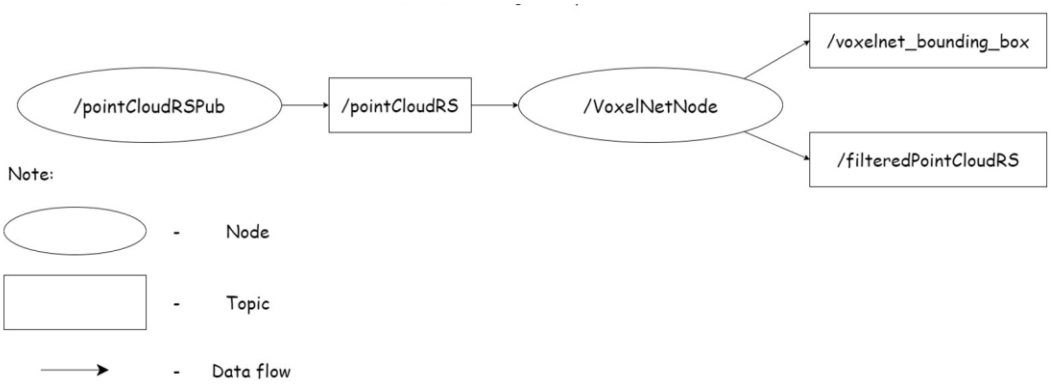


Figure 30: ROS integration and Machine Learning

Figure 30 represents the internal implementation in ROS. A node in ROS is a piece of code that will be continuously sending and receiving data to or from other nodes. The topic is similar to that of a channel where the data is "published" to and to receive the data, the node will need to "subscribe" to the topic and further process the data. The node "/pointCloudRSPub" sends 3D point cloudto the topic "/pointCloudRS" and the node "/VoxelNetNode" subscribes to the same topic and publish the detected pedestrians to the two topics which are "/voxelnetbounding_box" and "/filteredPointCloudRS", these can then be visualised in Rviz.

Future research

Based on the performed experiments, the following steps are envisaged for near future research and developments:

- Further investigation on ROS and possibilities to run VoxelNET on ROS
- Connecting the point cloud processing with the VoxelNET
- Improving the performance of the background removal using voxels.
- Performing tests with real data from RS-Bpearl and improving the VoxelNET model for real data
- Developing API to provide number of pedestrians in the FOV
- Integrating density computation in the workflow
- Setting up an experiment at front of the Red Centre,
- Further tuning of algorithms to be able to process laser scanes in a relatively short (near real) time.
- Introducing more scanners for obtaining a denser point cloud
- Experimenting with different groups of pedestrians

References

Alexandrov, M., D. J. Heslop and S. Zlatanova, 2021, 3D Indoor Environment Abstraction for Crowd Simulations in Complex Buildings, Buildings, 2021, 11(10), 445

Geiger, A., P. Lenz and R. Urtasun, 2012, Are we ready for autonomous driving? The KITTI vision benchmark suite, *2012 IEEE Conference on Computer Vision and Pattern Recognition*, 2012, pp. 3354-3361, doi: 10.1109/CVPR.2012.6248074.

Helbing, D., & Molnar, P. (1995). Social force model for pedestrian dynamics. *Physical Review E*, *51*(5), 4282.

Li, Ch-Ch, P-Ch. Wu and Ch. H. Lin, 2013, Pedestrian detection using heuristic statistics and machine learning,*2013 9th International Conference on Information, Communications & Signal Processing*, 2013, pp. 1-5, doi: 10.1109/ICICS.2013.6782960.

Maurelli, F., D. Droeschel, T. Wisspeintner, S. May and H. Surmann, 2009, A 3D laser scanner system for autonomous vehicle navigation, *2009 International Conference on Advanced Robotics*, 2009, pp. 1-6.
Mozaffari, S., O. Y. Al-Jarrah, M. Dianati, P. Jennings and A. Mouzakitis, 2020, Deep Learning-Based Vehicle Behavior Prediction for Autonomous Driving Applications: A Review, in *IEEE Transactions on Intelligent Transportation Systems*, doi: 10.1109/TITS.2020.3012034.

Qian, Y., J. Barthelemy, and P. Perez, 2021, Urban vehicle localization in public LoRaWan network, Lecture Notes on Computer Science, International Workshop on Multi-Agent Systems and Agent-Based Simulation, MABS 2020: Multi-Agent-Based Simulation XXI, pp 28-40

Preparing the Input Data Structure — Transfer Learning Toolkit 2.0 documentation 2011, Nvidia.com, viewed 9 December 2021, <https://docs.nvidia.com/tao/archive/tlt-20/tlt-user-guide/text/preparing_data_input.html>.

Rosenzweig, J. and M. Bartl, 2015, A Review and Analysis of Literature on Autonomous Driving, THE MAKING-OF INNOVATION, E-JOURNAL makingofinnovation.com, OCTOBER 2015, 57p.

Scheunert, U., H. Cramer, B. Fardi and G. Wanielik, 2004, Multi sensor based tracking of pedestrians: a survey of suitable movement models, *IEEE Intelligent Vehicles Symposium, 2004*, 2004, pp. 774-778, doi: 10.1109/IVS.2004.1336482.

Verbree, E., S. Zlatanova, K. B. A. van Winden, E. B. van der Laan, A. Makri, L. Taizhou, and A. Haojun, 2013 To localise or to be localised with *WiFi* in the Hubei museum? The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences, Volume XL-4/W4, 2013. 31-35 December 2013, Cape Town, South Africa

Wang, Z. and S. Zlatanova, 2019, Safe Route Determination for First Responders in the Presence of Moving Obstacles, IEEE Transactions on Intelligent Transportations Systems, pp 1-19

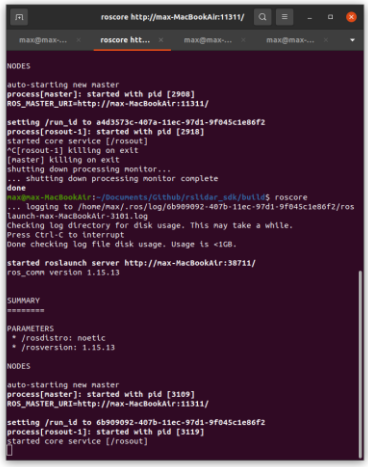
Xiao,W , B. Vallet , K. Schindler, N. Paparoditis, 2016,Simulataneous detection and tracking of pedestrians from panoramic laser scanning data, ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences, Volume III-3, 2016 XXIII ISPRS Congress, 12–19 July 2016, Prague, Czech Republic, pp 295-302

Zhao, H and R. Shibasaki, 2005, A novel system for tracking pedestrians using multiple single-row laser-range scanners, in *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, vol. 35, no. 2, pp. 283-291, March 2005, doi: 10.1109/TSMCA.2005.84

Zhou, Y 2017, VoxelNet: End-to-End Learning for Point Cloud Based 3D Object Detection,17 November.

Appendix: Running version of ROS (tested with ROS Noetic on Ubuntu 20.4)

Roscore node running as a process in the background.



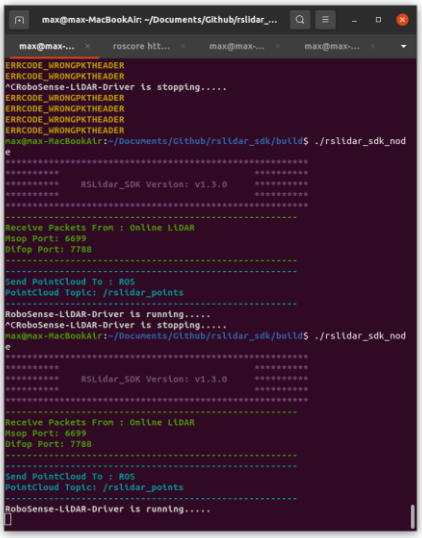
RoboSense LiDAR SDK downloaded from the Robosense GitHub repository.

Refer to: https://github.com/RoboSense-LiDAR/rslidar_sdk

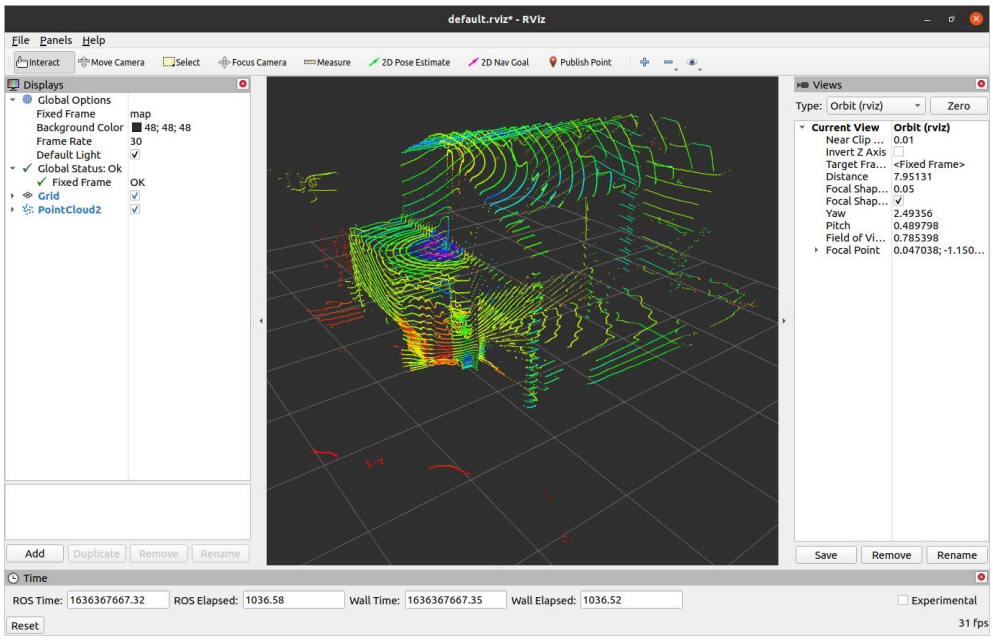
Run the following code to compile the driver package using CMake and start the executable:

```
cd rslidar_sdk
mkdir build && cd build # Assuming build folder does not existcmake ..
&& make -j4 # Compile the C++ code
./rslidar_sdk_node # Run the sensor node Note: make
sure the roscore is run in advance.
```

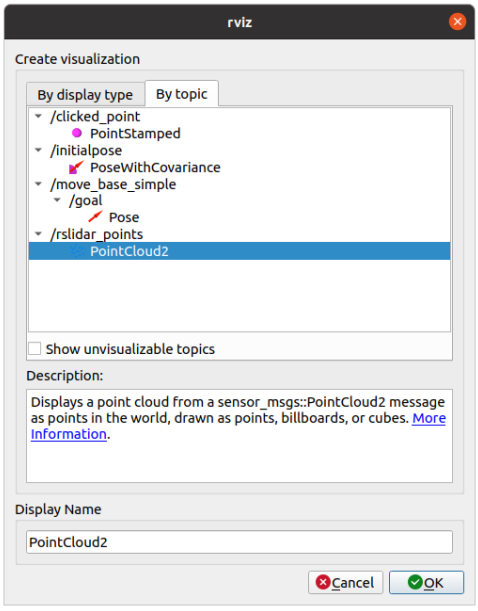
RS_driver running in background:



The visualisation tool for the 3D point cloud data on rviz (visualization tool for ROS which allows you to visualise) from RS B-Peral.



Rviz subscribes to the topic “/rslidar_points” to receive data:



Later on, we can implement/utilize pedestrian detection algorithms and subscribe to the /rslidar_points and visualize the result in the same way as we did for 3D point clouds by stacking them to each other.

Other Configurations: