

Semantic Modelling of an Indoor Parking Garage Using Hand-held GeoSLAM LiDAR Point Clouds

by

Jingyi (Kristie) Hu

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Science
in
Geography

Waterloo, Ontario, Canada, 2024

© Jingyi (Kristie) Hu 2024

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The development of high-definition (HD) digital twin models for underground parking lots presents significant challenges due to the absence of signals from satellite navigation systems, fluctuating lighting conditions, and obstruction-rich environments. These complexities hinder applications that rely on accurate spatial awareness, such as emergency rescue, navigation assistance, and autonomous parking. This thesis presents an elaborate methodology for generating an HD digital model of an indoor parking lot. A LiDAR-based Simultaneous Localization and Mapping (SLAM) system was used for point cloud acquisition and colorization. The methodology encompasses the application of leading-edge algorithms, including line feature extraction, semantic segmentation, and surface reconstruction. The effectiveness of the proposed methodology is underscored by parallel comparisons of ground truth with visual output (e.g., line segmentation, and reconstructed models). Notably, segmentation via DCTNet achieves high-performance metrics in the average class IoU of the model (90.74%) and average F_1 score (98.65%). Overall, these demonstrate the efficiency of the proposed methodology in developing a detailed indoor parking garage model using advanced LiDAR-based SLAM technology, addressing challenges in GPS and lighting, and providing crucial insights for future advancements in 3D indoor modelling through comprehensive accuracy assessments and semantic enhancements.

Acknowledgements

I am eternally grateful to my parents, Liangya and Xiaoling, whose love and sacrifices have been the bedrock of my perseverance. You have all been a constant source of love and encouragement during the ups and downs of my academic pursuit.

I extend my thanks to my supervisor, Dr. Jonathan Li, for his unwavering guidance, patience, and support throughout this research. Your insights and wisdom have been invaluable to my work.

I would like to express my profound gratitude to all those who have been instrumental in the completion of this thesis. Special thanks go to Dening Lu from the Department of Systems Design Engineering, University of Waterloo, for your constructive advice and support in experiments.

I would like to thank Jing Du and Willow Liu's accompanying and encouragement during my research. I would also like to thank Kyle Gao and Hongjie He for their support during my graduate study.

Lastly, I would like to acknowledge the financial support provided by the Caivan Future Cities Graduate Scholarship and the IMAE award which enabled me to commit to my research fully.

Completing this thesis was a formidable task, and it would not have been possible without the generous help of many individuals. To all who have contributed in one way or another, thank you.

A handwritten signature in Chinese characters, likely '刘静' (Liu Jing), written in black ink. The signature is stylized and includes a long horizontal stroke extending to the left.

Dec. 18th, 2023

Table of Contents

Author's Declaration	ii
Abstract	iii
Acknowledgements	iv
List of Figures	ix
List of Tables	xii
List of Abbreviations	xiii
1 Introduction	1
1.1 Motivation	1
1.2 Research Gaps	2
1.2.1 Benchmark Establishment	2
1.2.2 Data Acquisition	2
1.2.3 Data Quality Control	3
1.3 Objectives of the Study	3
1.4 Structure of the Thesis	4

2	Related Works	5
2.1	Line Detection	5
2.2	Line Framework Optimization	8
2.3	Surface Reconstruction	9
2.4	Semantic Segmentation	14
3	Data Collection	19
3.1	Study Area	19
3.2	Data Collection	21
3.2.1	Hand-held Laser Scanner	21
3.2.2	Setup and Trajectory	25
3.3	Data Preprocessing	27
3.3.1	“.GeoSLAM” to “_filter_colored.laz”	27
4	Semantic Segmentation and Surface Regularization	30
4.1	Data Pre-processing	31
4.1.1	Semantic Labelling	31
4.1.2	Ceiling Removal	33
4.1.3	Subsampling	33
4.2	Semantic Segmentation	35
4.2.1	Input: Preparing Train and Test Sets	35
4.2.2	Sample Generation	38
4.2.3	Weight	39
4.2.4	Model Training	39
4.3	Surface Sampling and Reconstruction	40
4.3.1	Vertical and Horizontal Plane Extraction	40
4.3.2	HDBSCAN Clustering	42
4.3.3	BPA	43

4.3.4	pyRANSAC-3D (RANSAC Multi-Plane Fitting)	43
4.4	Line Feature Reconstruction	45
4.4.1	3D Line Detection	46
5	Results and Discussion	47
5.1	Semantic Segmentation Result	47
5.1.1	Evaluation Metric	47
5.1.2	Results	48
5.1.3	Visualization of Segmented Results	50
5.2	Surface Sampling and Reconstruction Results	52
5.2.1	HDBSCAN Clustering	52
5.2.2	BPA Surface Reconstruction	53
5.2.3	pyRANSAC-3D (multi-plane fitting)	55
5.2.4	Random Sample Consensus with multi-plane fitting	56
5.2.5	Surface Output Comparison	60
5.3	Line Reconstruction	60
5.3.1	3D Line Detection	60
5.3.2	Output Optimization	61
5.3.3	Reconstruction Visualization Output	63
5.4	Limitations	67
5.4.1	Human/ Operator Error	67
5.4.2	Insufficient/missing Data	67
5.4.3	Equipment/ Algorithm Limitation	68
6	Conclusions and Recommendations for Future Studies	71
6.1	Conclusions	71
6.2	Recommendations for Future Studies	71
6.2.1	Point Cloud Completion	71
6.2.2	Object Detection	72

References	74
Appendices	78
Appendix A	78
Appendix B	81
Appendix C	86
Appendix D	87
Appendix E	88
Appendix F	89
Appendix G	90

List of Figures

1.1	Information loss due to occlusions (Source: Balado et al., 2020)	3
2.1	A comparison of outputs from KNNs algorithm for region growing and merging (Source: Lu et al., 2019)	6
2.2	3D line segment extraction result (Source: Lu et al., 2019)	7
2.3	Examples of imperfect line structure extraction (Source: Wang et al., 2018)	8
2.4	Space Partitioning & Reconstructed Floor Plan (Source: Babacan et al., 2016)	11
2.5	Floor plan Reconstruction with (a) indicates raw point clouds, and (b) refers reconstructed floor plan of the indoor scene (Source: Fang et al., 2021)	12
2.6	Segments \tilde{e}_i, \tilde{e}_j on Candidate Edges e_i, e_j (Source: Fang et al., 2021)	13
2.7	pyRANSAC-3D plane fitting (Source: Leonardo, 2022)	14
2.8	Architecture of PointNet network (Source: Qi et al., 2017a)	15
2.9	Architecture of PointNet++ network (Source: Qi et al., 2017b)	15
2.10	Architecture of encoder-to-decoder structure in DCTNet (Source: Lu et al., 2023)	16
3.1	General workflow	19
3.2	Location of the study area (Source: Google OpenStreetMap)	20
3.3	Overview of the raw point clouds of the parking garage (filtered.laz)	23
3.4	Bird view of the raw data at level 4 in outlet parking garage	24
3.5	Devices setup (a): ZEB Horizon and hollow-out cross; (b): Control point alignment testing	25

3.6	Trajectory and moving direction (star: start and endpoint; arrow: direction)	26
3.7	Workflow of “Process SLAM” on GeoSLAM Connect	27
3.8	Image capture by ZEB Vision camera: (a) BS image at Spot ∂ (BS_{∂}); (b) FS image at Spot ∂ (FS_{∂})	28
3.9	Example of an Image Position file	29
4.1	Stage 2 of the overall workflow	30
4.2	Ceiling Removal: (a) before ceiling removal; (b) after ceiling removal	33
4.3	A comparison between the original and the sub-sampled data: (a) Original Density; (b) Sub-sampled Density	34
4.4	Examples of raw test sample (size=4096 points) from sample generation	38
4.5	Calculate class weights	39
4.6	Detailed workflow of surface sampling and reconstruction	40
4.7	RANSAC plane segmentation and plane fitting equation	41
4.8	3D plot of the plane (based on the computed plane fitting equation)	41
4.9	Horizontal Plane Extraction: (a) Raw point cloud cell; (b) detectHorizon output: Horizontal plane in red [1, 0, 0] and all the other inlier points (vertical planes) in light grey [0.8, 0.8, 0.8]	42
4.10	pyRANSAC-3D (a) line fitting, (b) plane fitting, and (c) circle fitting example (Source: Leonardo, 2022)	44
4.11	Detailed workflow of edge or line segments reconstruction	45
4.12	Sample outputs using LineDecton3D (Source: Lu et al., 2019)	46
5.1	Confusion matrix for semantic segmentation task	49
5.2	Ground truth (left) vs. segmented results (right) on test dataset	50
5.3	Cluster output with HDBSCAN clustering approach	52
5.4	BPA mesh visualization	53
5.5	pyRANSAC-3D multi-plane fitting visualization	55
5.6	Algorithm for RANSAC Plane Fit	57
5.7	Part A: Revised RANSAC multi-plane fitting clusters 00 - 11	58

5.8	Part B: Revised RANSAC multi-plane fitting clusters 12 - 16	58
5.9	Merged RANSAC multi-plane fitting visualization	59
5.10	Line segment output using LineDetection3D ($k = 20$)	60
5.11	Optimized line segment reconstruction output visualization	61
5.12	Optimized line segment reconstruction output visualization detail views . .	62
5.13	Optimized line segment reconstruction output visualization detail views . .	65
5.14	Raw point clouds vs. final 3D model	66
5.15	GeoSLAM laser scanner operation during data collection	67
5.16	Insufficient points and miss detected line segments	68
5.17	Data Occlusion	69
5.18	Optimized enclosed loop trajectory 01 (vehicle and pillar focused)	70
5.19	Optimized enclosed loop trajectory 02 (wall structure focused)	70
6.1	Data loss during collection: (a) inconsistent walking speed;(b) occlusions .	72
6.2	Vehicle in bounding box	73
6.3	Safety bollards clusters by HDBSCAN	73
6.4	Safety bollards in bounding box	73

List of Tables

3.1	GeoSLAM ZEB Horizon technical specification	21
3.2	GeoSLAM ZEB Horizon configuration	22
3.3	GeoSLAM ZEB Vison camera accessory	22
3.4	Parameter of the indoor parking garage scene	23
4.1	Example sub-sets for each class	32
4.2	Data size before vs. after sub-sampling for each class	34
4.3	Data size before vs. after sub-sampling for classes in semantic segmentation	34
4.4	Training sets	36
4.5	Testing sets	37
5.1	Machine configuration	47
5.2	Testing sets breakdown	48
5.3	GT vs. good prediction example	51
5.4	GT vs. Unsatisfactory predictions example	51
5.5	BPA Output Comparison	54

List of Abbreviations

BPA Ball-Pivoting Algorithm [40](#), [43](#), [53](#), [54](#)

BS Backsight [28](#)

BSP Binary Space Partitioning [6](#), [9](#), [10](#)

cGAN conditional Generative Adversarial Network [8](#), [9](#), [68](#)

DCTNet Dynamic Clustering Transformer-based Network [16–18](#), [31](#), [35](#), [39](#)

DL Description Length [10](#), [11](#)

FPS Farthest Point Sampling [38](#)

FS Foresight [28](#)

GFL Global Feature Learning [17](#), [35](#)

GPS Global Positioning System [22](#), [23](#)

GT Ground Truth [50](#)

GTA Great Toronto Area [20](#)

HDBSCAN Hierarchical Density-based Spatial Clustering [31](#), [40](#), [42](#), [44](#), [55](#), [56](#)

IMU Inertial Measurement Unit [25](#), [26](#), [69](#)

IoU Intersection Over Union [49](#)

KNN K-Nearest Neighbours [6](#), [7](#), [17](#), [38](#)

LED Light-emitting diode [25](#)

LFA Local Feature Aggregating [17](#)

LiDAR Light Detection and Ranging [21](#), [22](#), [71](#)

LSD Line Segment Detector [5](#)

LSHP Line-Segment-Half-Planes [6](#)

MCMLSD Markov Chain Marginal Line Segment Detector [7](#)

MDL Minimum Description Length [10](#)

MLP Multilayer Perceptrons [14](#), [17](#), [35](#)

MVCNN Multi-view Convolutional Neural Networks [16](#)

NumPy Numerical Python [43](#), [44](#)

OpenCV Open Source Computer Vision Library [46](#)

OpenMP Open Multi-processing [46](#)

PCA Principal Component Analysis [6](#)

RANSAC Random Sample Consensus [9](#), [13](#), [31](#), [40](#), [41](#), [43](#), [44](#), [56](#), [60](#)

ReLU Rectified Linear Unit [17](#)

SDC Semantic Feature-based Dynamic Clustering [38](#)

SDS Semantic Feature-based Dynamic Sampling [17](#)

SGD Stochastic Gradient Descent [39](#)

SLAM Simultaneous Localization and Mapping [2](#), [3](#), [13](#), [21](#), [22](#), [69](#), [71](#)

TLS Terrestrial Laser Scanning [8](#)

WHU Wuhan University [8](#)

Chapter 1

Introduction

1.1 Motivation

The advent of digital twin technology has opened new horizons for modelling and managing complex infrastructures (Lehtola et al., 2022). In the modern landscape of urban infrastructure management, navigation, and navigational assistance, the development of high-definition (HD) maps and digital twin models, especially for indoor environments, is becoming increasingly significant, as a comprehensive 3D model of the surrounding environment is crucial for navigation purposes (Elghazaly et al., 2023). Recently, laser scanning technology has been commonly used for indoor mapping (Cui et al., 2019). However, the reliability and application of laser scanning techniques are often undermined by the inherent limitations in indoor scenes. These can be fluctuating lighting conditions, obstructions occlusions in limited spaces, etc (Yurtsever et al., 2020)., which would reflect a series of issues in data quality like poor cloud density, inconsistencies, spatial discreteness, and incompleteness Döllner (2020). The deficiencies will further exacerbate the negative effects on the overall model leading to poor spatial accuracy and unsatisfactory segmentation output, such setbacks can significantly undermine the feasibility and dependability of 3D modeling. Consequently, there is an increasing demand to refine and standardize these techniques (Laoudias et al., 2018), given that the sophistication of indoor models does not yet match that of their outdoor counterparts, and there were very few previous studies that attempted to develop and evaluate point cloud model performances within indoor underground scenes (Wróblewski et al., 2022).

This thesis addresses these challenges accordingly, introducing a comprehensive methodology that delivers precise reconstructed models of underground parking facilities. The

process starts with the acquisition of raw point cloud data, which are then colored using a LiDAR-based [Simultaneous Localization and Mapping \(SLAM\)](#) system in GeoSLAM ([Sammartano and Spanò, 2018](#)). This system can be specifically refined to tackle the complexities of indoor infrastructure mapping, ensuring robustness and reliability in the model reconstruction ([Fathi et al., 2015](#)).

The growing demand for efficient use of urban space is pushing the development of underground parking lots ([Broere, 2016](#)). However, the navigation within these environments is often fraught with inaccuracies due to the aforementioned challenges. There is an urgent need to improve the spatial accuracy and semantic understanding of these scenes to support the development of autonomous assistance systems like a last-kilometre autonomous parking assistance solution ([Mounce and Nelson, 2019](#)). This study is dedicated to addressing these needs by developing a highly accurate underground parking lot model that can be fundamental for the deployment of navigation assistance systems, providing them with the critical environmental data necessary for successful operation.

1.2 Research Gaps

1.2.1 Benchmark Establishment

2D geospatial information nowadays is developed very well with complete structures for both proprietary GIS and most authorized open-source GIS ([Orengo, 2015](#)), but the situation for 3D point cloud data is different. Especially for an indoor scene, existing indoor mobile laser scanning (iMLS) point clouds and labelled datasets are not rich enough to stimulate large-scale public participation in geographic information system (PPGIS) collaborations like 2D geospatial information ([Sieber, 2006](#)). There is only a handful of globally recognized open-access indoor point cloud datasets available for the community to test and validate 3D geospatial information model performances ([Guo et al., 2020](#)).

1.2.2 Data Acquisition

Unlike 2D geospatial information, 3D geospatial information incorporates a third coordinate into each data entry, which means we shall consider not only the X and Y coordinates but also the Z axis ([Orengo, 2015](#)) which is perpendicular to the horizontal X-Y plane ([Lee, 2004](#)). This introduces another challenge and complexity during data collection. As shown in [Figure 1.1](#), the presence of obstacles (e.g. a car) can impede the line of sight between

the laser scanner and the target structure (e.g. a wall on the façade line) (Balado Frías et al., 2020). This results in significant information loss due to occlusions, where the areas behind obstacles are not scanned and thus remain undetected during the data collection phase (Balado Frías et al., 2020).

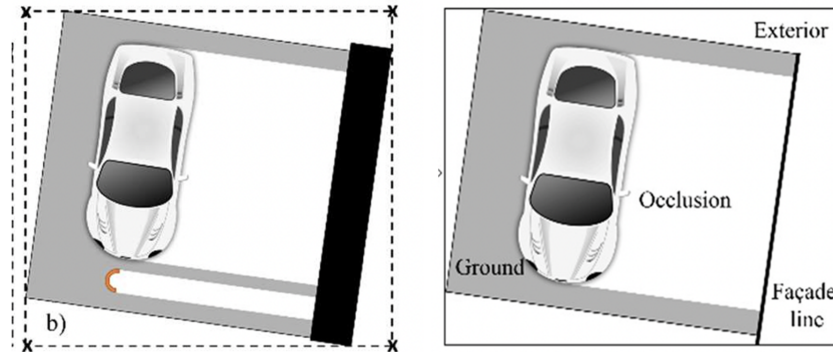


Figure 1.1: Information loss due to occlusions (Source: Balado et al., 2020)

1.2.3 Data Quality Control

The intrinsic characteristics of 3D point cloud data include high density, irregular distribution, and varying scales, which can intensify data quality issues (Kodors, 2017). For example, these features may contribute to occlusions, incomplete data capture, inconsistent point densities, and spatial discontinuities, all of which could hinder and complicate the production and utilization of 3D point clouds. In this case, there is another gap steering our future contributions toward addressing low-level taskings, such as point cloud correction and completion (Fei et al., 2022). The enhancement of point cloud quality is urgently needed.

1.3 Objectives of the Study

The objectives of this study are three-fold:

- To develop a high-quality point cloud dataset using a LiDAR-based SLAM system.

- To construct an annotated indoor parking lot model achieved by semantic segmentation, line extraction, and surface reconstruction, respectively.
- To assess and validate the model’s accuracy and utility using globally recognized metrics. The outcomes will then be compared to derive insights for subsequent enhancements.

1.4 Structure of the Thesis

This thesis is organized into five chapters:

Chapter 1 introduces the motivations, research gaps, and objectives of the study with challenges in generating 3D models in an underground environment.

Chapter 2 presents the related works in indoor mapping and semantic modelling using laser scanning point clouds.

Chapter 3 describes the proposed methodology and overall workflow. Three tasks (semantic segmentation, surface reconstruction, and line segment extraction and optimization) are detailed.

Chapter 4 presents the experimental results with the model performance evaluation and visualization. It also reiterates the contribution made to the field of indoor mapping and discusses the limitations and impact of this thesis.

Chapter 5 proposes directions for future work to advance the capabilities of semantic modelling of indoor environments.

Chapter 2

Related Works

Over the years, indoor modelling and environmental reconstruction have made substantial progress, rooted in continuous research that has enhanced our methods for analyzing, visualizing, and reconstructing spaces (Berger et al., 2014). Contemporary algorithms have significantly broadened and refined methodologies for performing a variety of tasks, whether applied to sets of 2D images or directly onto 3D point clouds (Xu et al., 2021), marking a significant evolution in the field. Processing 3D data has now become a popular research domain that enables users to extract and analyze spatial information, recognize patterns, and understand the complex interrelationships between global structures and local features within the voxels of a specific scene in different environments (Ioannidou et al., 2017). Work related to line detection (Section 2.1), line optimization (Section 2.2), surface reconstruction (Section 2.3) and semantic segmentation (Section 2.4) have been reviewed, respectively.

2.1 Line Detection

Lin et al. (2015) deviated from conventional methods by emphasizing the direct extraction of line segments from unstructured 3D point clouds in real-world environments. This approach combines earlier research that prioritized 2D line detection (Von Gioi et al., 2008) and 3D reconstruction from multiple images (Jain et al., 2010). In their study, Lin et al. transformed point clouds into rendered images from various viewpoints and then applied the **Line Segment Detector (LSD)** algorithm (Von Gioi et al., 2008). to identify 2D line support regions. Subsequently, these regions were re-projected into 3D space. To improve

accuracy, these regions were represented using a [Line-Segment-Half-Planes \(LSHP\)](#) structure. The result is a structured final output, which applies a space partitioning algorithm similar to the work of [Babacan et al. \(2016\)](#) in Section 2.3, which utilized the [Binary Space Partitioning \(BSP\)](#) for feature extraction.

[Lu et al. \(2019\)](#) proposed an efficient framework for rapid line detection and segmentation within point cloud data . The process begins with the segmentation of input point clouds into distinct 3D planes using a combination of techniques for region growing and merging, as depicted in Figure 2.1.

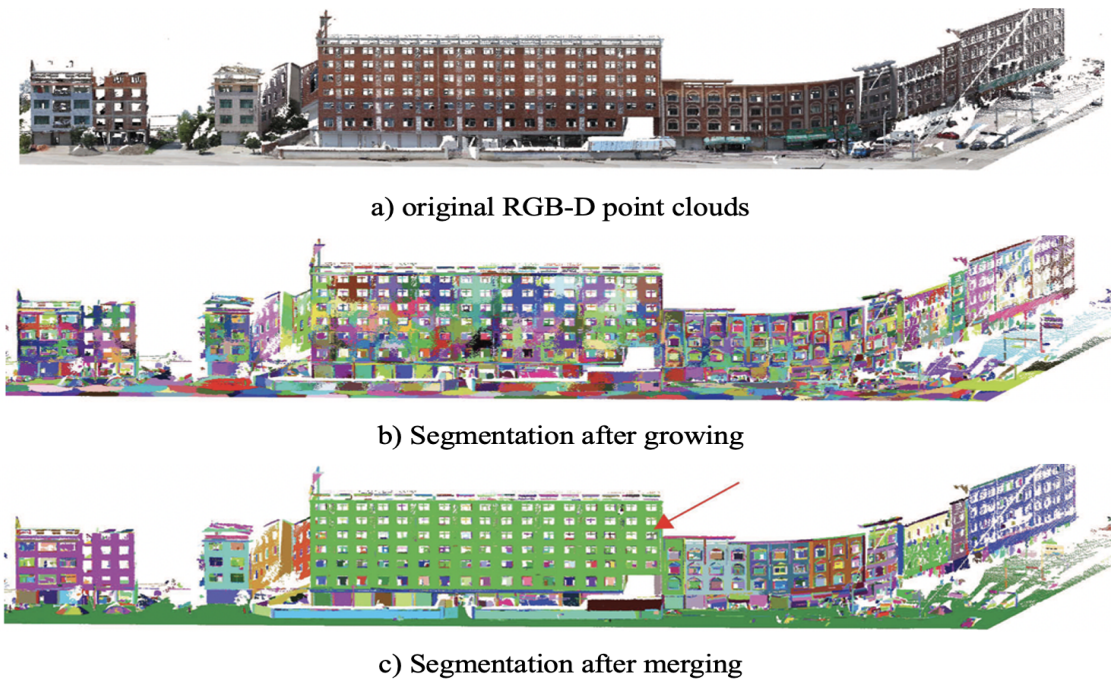


Figure 2.1: A comparison of outputs from KNNs algorithm for region growing and merging (Source: Lu et al., 2019)

The algorithm [K-Nearest Neighbours \(KNN\)](#) is used to identify neighbouring points, and the normal of each plane is estimated through the [Principal Component Analysis \(PCA\)](#). The equation of the covariance matrix for [PCA](#) can be expressed as follows:

$$\sum = \frac{1}{k} \sum_{i=1}^k (P_i - \bar{P})(P_i - \bar{P})^T \quad (2.1)$$

where P_i is a neighbouring point, and \bar{P} is the mean vector of the [KNNs](#). The region growing and merging approach is applied to identify planes within a point cloud by grouping points on the same surface.

Subsequently, for each identified plane, the corresponding points are orthogonally projected onto it, creating a 2D representation. These 2D line segments are then reprojected back onto their respective 3D planes to reconstruct the original 3D segments. A post-processing step is implemented to remove outliers and merge 3D line segments that are close to each other. The final output, which should resemble the illustration in [Figure 2.2](#), comprises cleanly defined 3D edges boundaries and surfaces.



Figure 2.2: 3D line segment extraction result (Source: [Lu et al., 2019](#))

[Hu et al. \(2022\)](#) developed an approach that combines a geometric feature-enhanced line extraction technique with a hierarchical topological optimization process. Initially, line segments are extracted from 2D edge maps using the [Markov Chain Marginal Line Segment Detector \(MCMLSD\)](#) algorithm and then re-projected into 3D to form initial candidates. Subsequently, these candidates are optimized for topological accuracy through merging and refinement operations. The process begins with plane extraction via region growing and merging. Segmented regions are then mapped onto 2D grids through a geometric feature-enhanced 3D-to-2D projection, followed by the extraction of 3D line segments through the integration of multiscale edge features. Finally, line segments undergo a two-stage refinement by first merging perceptually similar line segments to form precise line sets and then applying hierarchical optimization to these merged line segments to enhance the topology and accuracy of the overall line set.

This method outperformed current leading algorithms, achieving an average of 86% completeness and correctness in line extraction and running at a speed of 25,000 points per second when tested on the Semantic 3D and the [Wuhan University \(WHU\)-Terrestrial Laser Scanning \(TLS\)](#) datasets.

2.2 Line Framework Optimization

The output of line segment extraction often contains many occlusions and elements from cluttered backgrounds. [Wang et al. \(2018\)](#) categorized these occlusions into three main types: 1) a parallel or orthogonal relationship between some adjacent edges or lines, as shown in Figure 2.3 (a), 2) incomplete structures and disconnected line segments or edges, illustrated in Figure 2.3 (b), and 3) extrusions of line segments or edges, as depicted in Figure 2.3 (c).

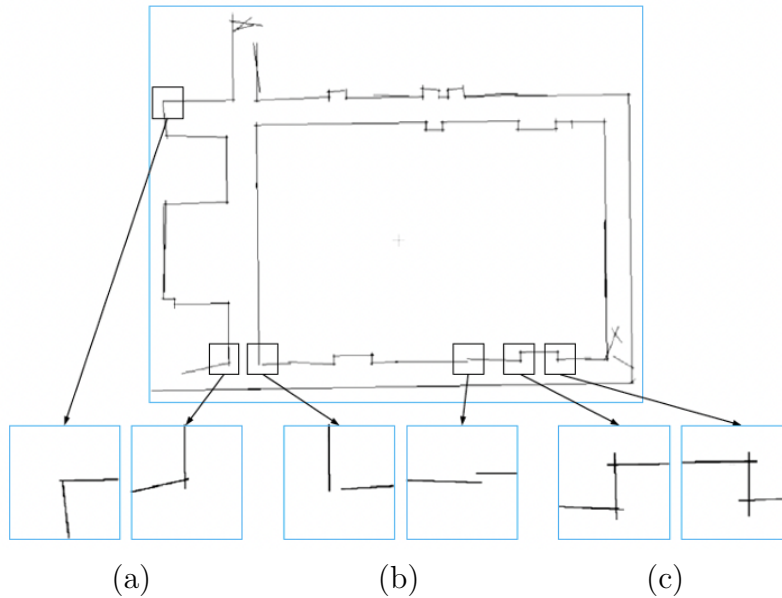


Figure 2.3: Examples of imperfect line structure extraction (Source: Wang et al., 2018)

To optimize the generation of imperfect semantic lines, [Wang et al. \(2018\)](#) proposed an optimization model based on a [conditional Generative Adversarial Network \(cGAN\)](#) framework, which begins by fitting the line structures onto their corresponding 2D planes. To transform the points in the new coordinate system, the z coordinate of each point is

set to zero. Two orthogonal unit vectors $u_x = (u_{x1}, u_{x2}, u_{x3})$ and $u_y = (u_{y1}, u_{y2}, u_{y3})$ are selected as the new x and y axes, and a new z axis will be denoted as $u_z = (u_{z1}, u_{z2}, u_{z3})$.

A new coordinate for each point will be shown as follows:

$$(x', y', z', 1) = (x, y, z, 1) \cdot T \cdot R \quad (2.2)$$

where the translation matrix (T) and the rotation matrix (R) are represented as follows:

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -x_0 & -y_0 & -z_0 & 1 \end{bmatrix} \quad (2.3)$$

$$R = \begin{bmatrix} u_{x1} & u_{x2} & u_{x3} & 0 \\ u_{y1} & u_{y2} & u_{y3} & 0 \\ u_{z1} & u_{z2} & u_{z3} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (2.4)$$

The x and y coordinates are mapped to rows and columns in a 2D image, which is then subdivided into 256×256 sub-images. These sub-images are processed by a convolutional neural network (specifically, the VGG16 CNN architecture) to extract features. Three fully connected layers are then employed to classify these features. Once the 2D line segments have been optimized using the [cGAN](#) models, they are reconstituted into 3D point representations using Equation 2.2.

2.3 Surface Reconstruction

Reconstructing floor plans typically involves connecting fragmented or missing lines to create a surface that is geometrically and topologically accurate. [Babacan et al. \(2016\)](#) introduced a space-partitioning approach to extract geometric details in three steps: 1) selecting the optimal horizontal slice based on volume, 2) applying 3D plane fitting to points chosen by [Random Sample Consensus \(RANSAC\)](#), and 3) regularizing by converting 3D data into 2D and performing line quantization. Additionally, doors and other structural building elements were identified as semantic data to refine the geometric details further. Using the developed framework, the model can be extensively segmented with the [BSP](#)

method, which is informed by these refined primitives. The **BSP** process generates a tree (known as a **BSP** tree), where each node contains a partitioning element (a line in 2D or a plane in 3D). Each leaf node represents a convex subspace of the original area. Mathematically, a plane used in the **BSP** and defined by the general plane equation can be expressed as follows:

$$Ax + By + Cz + D = 0 \quad (2.5)$$

where A, B, C are the normal vectors to the plane (denoted as \vec{n} or $N = (A, B, C)$); x, y, z are the 3D coordinates; D is the distance from the origin to the plane along the normal direction. For any point $P(x, y, z)$: if $Ax + By + Cz + D > 0$, P is in front of the plane (front-half); if $Ax + By + Cz + D < 0$, P is behind the plane (back-half); and if $Ax + By + Cz + D = 0$, P is on the plane.

The **BSP** process in 3D employs the plane equation (Eq. 2.5) to determine the position of a point relative to the partitioning plane, effectively organizing the space into distinct regions. To complete the floor plan, adjacent sections from the **BSP** tree that share common attributes are merged using the **Minimum Description Length (MDL)** principle (Babacan et al., 2016). In essence, the **MDL** principle seeks to identify the model that offers the most concise representation of the data, considering both the complexity of the model and the data it encodes. While the **MDL** principle can be applied to a variety of statistical models and comes in several forms, the underlying concept can be mathematically represented by:

$$DL = \lambda(L(D) + (1 - \lambda)(L(H)) \quad (2.6)$$

$$L(D) = \frac{\Omega}{2 \ln 2} \quad (2.7)$$

$$L(H) = L'_p \log_2 L_p \quad (2.8)$$

where H is the hypothesis or model, D is the data, and λ is the weighting value. In (Eq. 2.7), Ω is the sum of the squared residuals between the data D and a model H : $[D - H]^T \times [D - H]$. In (Eq. 2.8), $L(D | H)$ is the cost to the data D given the model H (the goodness of fit), $L(H)$ is the length or cost that describes the model H , L_p is the **Description Length (DL)** or complexity of the initial model, and L'_p refers to the **DL** or complexity of a hypothesis (a potential new model).

The optimal model is a model that has a minimum DL value. With Babacan’s method, fragmented floating 3D data can be transformed into a coherent surface representation illustrated in Figure 2.4.

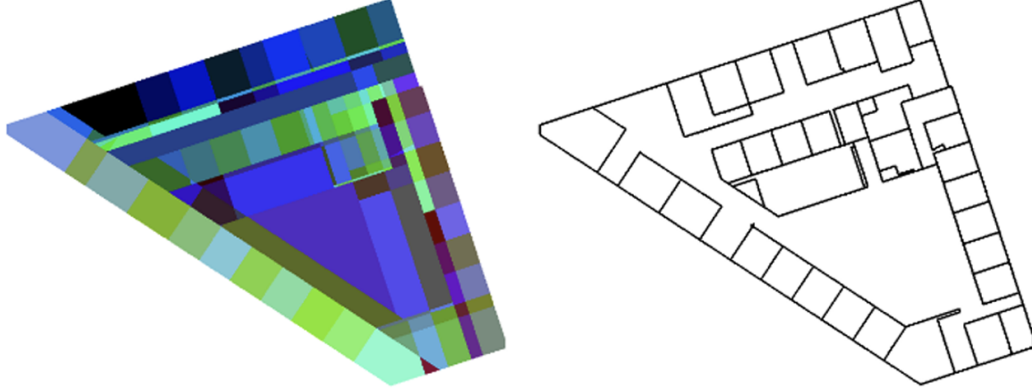


Figure 2.4: Space Partitioning & Reconstructed Floor Plan (Source: Babacan et al., 2016)

The significance of detecting wall planes is highlighted due to their status as some of the most prominent structures found indoors. Fang et al. (2021) proposed a space partitioning method designed to produce surfaces with a focus on identifying walls as key boundaries within an indoor environment. The method relies on a region-growing approach to detect planes while retaining the edges and facets formed by the critical inliers that correspond to these planes (see Figure 2.5). To isolate floor and ceiling planes, the algorithm identifies planes which normally are parallel to the z-axis (in an upward direction) and are situated near the top and bottom of the 3D bounding box that encapsulates the scene. The remaining planes are classified as walls (vertical planes), delineating the space boundary.

$$U(x) = (1 - \lambda) U_{fidelity}(x) + \lambda U_{complexity}(x) \quad (2.9)$$

where $U_{fidelity}(x)$ is the degree of agreement between a state configuration x and the input data, and $U_{complexity}(x)$ indicates the complexity of the output boundary.

$$U_{fidelity}(x) = \beta U_{points}(x) + (1 - \beta) U_{walls}(x) \quad (2.10)$$

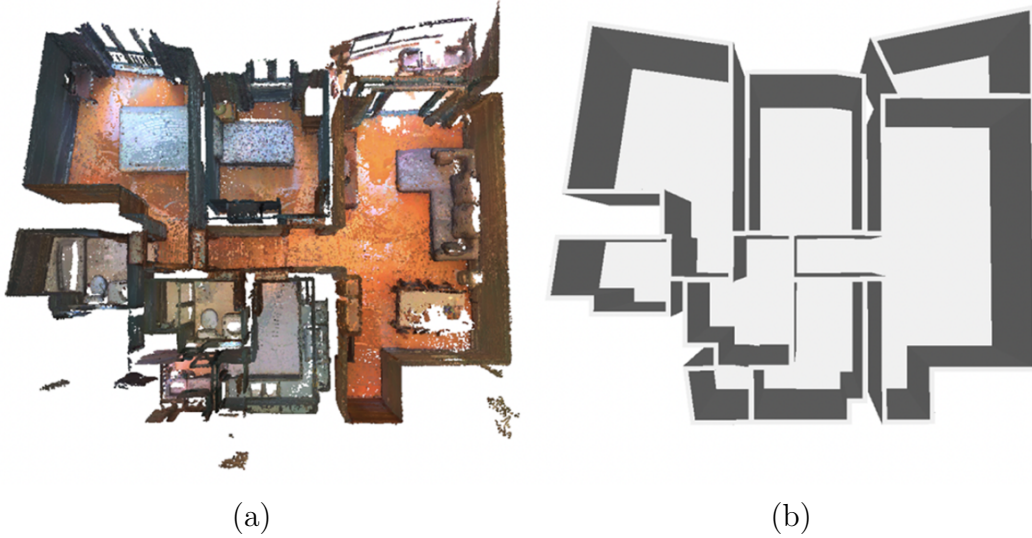


Figure 2.5: Floor plan Reconstruction with (a) indicates raw point clouds, and (b) refers reconstructed floor plan of the indoor scene (Source: Fang et al., 2021)

The first term $U_{points}(x)$ quantifies the proportion of input points that are enclosed by the boundary edges, which can be expressed mathematically as follows:

$$U_{points}(x) = \sum_{i=0}^n -|P(f_i^1) - P(f_i^2)| \cdot \frac{|e_i|}{\hat{E}} \cdot x_i \quad (2.11)$$

The term $U_{points}(x)$ is measured to calculate the extent to which boundary edges enclose the input data. It operates on the premise that each edge (e_i) is shared by two facets (denoted as f_i^1 and f_i^2). The length of each edge is represented by $|e_i|$, and \hat{E} signifies the cumulative length of all edges under consideration (Fang et al., 2021). The function $P()$ in Equation 2.11 is used to measure the probability that a given facet lies within the confines of the boundary edge. Consequently, $U_{points}(x)$ is designed to preferentially select edges in which incident facets exhibit a varied proportion of internal cells (contains input points), ensuring that a precise boundary accurately reflects the spatial distribution of the input point clouds (Fang et al., 2021).

A second term $U_{walls}(x)$ is designed to mitigate the influence of such noisy edges that might become erroneously active during the edge detection process, whose mathematical equation is represented as follows:

$$U_{walls}(x) = \sum_{i=1}^n \left(1 - \frac{|\tilde{e}_i|}{|e_i|}\right) \cdot \frac{|e_i|}{\hat{E}} \cdot x_i \quad (2.12)$$

where $|\tilde{e}_i|$ is the length of the overlapping segment between the edge and the corresponding wall plane. Essentially, $U_{walls}(x)$ imposes a penalty on the edges based on how little they overlap with the relevant wall structures.

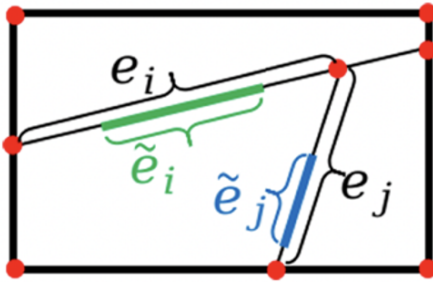


Figure 2.6: Segments \tilde{e}_i, \tilde{e}_j on Candidate Edges e_i, e_j (Source: Fang et al., 2021)

In addition, the **RANSAC** algorithm, originally introduced by **Fischler and Bolles (1981)**, is widely used in model fitting, particularly renowned for its robustness in handling data with outliers. Over the years, its utility has expanded, making it especially effective for plane fitting within sets of 3D points (**Mariga, 2022**). The core idea of **RANSAC** is to iteratively select a random subset of the original data, fit a model to this subset, and then determine how many of the other data points fit this model. This process repeats multiple times, and the best-fitting model is chosen as the final model.

PyRANSAC-3D (see Figure 2.7) is a novel implementation of the **RANSAC** algorithm designed by **Mariga (2022)** specifically tailored for 3D point cloud data. It is capable of fitting primitive 3D shapes such as planes, cuboids, and cylinders. This makes it particularly valuable in tasks that involve the identification and extraction of these shapes from a 3D point cloud, such as 3D **SLAM**, 3D reconstruction, object tracking, and many others, including but are not limited to plane, line, and point fitting.

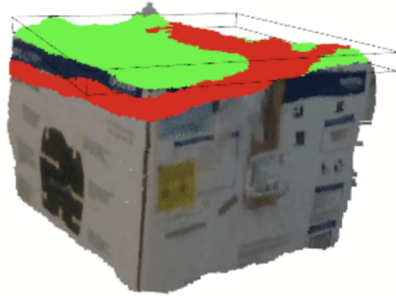


Figure 2.7: pyRANSAC-3D plane fitting (Source: Leonardo, 2022)

2.4 Semantic Segmentation

Nowadays deep learning strategies primarily aim to establish a comprehensive global embedding by conducting initial point-wise embedding, which are subsequently integrated through a sophisticated aggregation mechanism. The following summary reviews several prevailing literature on semantic segmentation techniques for point clouds, which provides valuable perspectives on how neural network architectures in semantic segmentation can function as benchmarks for evaluation and quality assurance during the generation of indoor 3D models. Semantic segmentation is a complex task that involves assigning each 2D pixel or 3D voxel to a category from a predefined set of classes. While it has been a subject of extensive research in the context of 2D imagery over the past decades, this research pivots towards the semantic segmentation of 3D point clouds. Accordingly, the discussion that follows will predominantly address recent advancements and methodologies contributed to the high-level tasking of 3D point cloud data.

PointNet (Qi et al., 2017a) was designed to process dispersed point-wise information and aggregate global features, utilizing shared multilayer perceptron (Multilayer Perceptrons (MLP)s) and symmetric pooling operations to ensure effective feature learning. The full network of PointNet is shown in Figure 2.8.

The key highlights in PointNet for semantic segmentation include 1) a symmetric function to make the network invariant to the order of the points (PointNet uses a maximum-pooling layer to aggregate the features of all points), 2) Shared MLP that combines local and global information to learn the spatial encoding of each point, and 3) Segmentation Network designed as an extension of the original classification network by adding local and global feature aggregation to assign a class to each point.

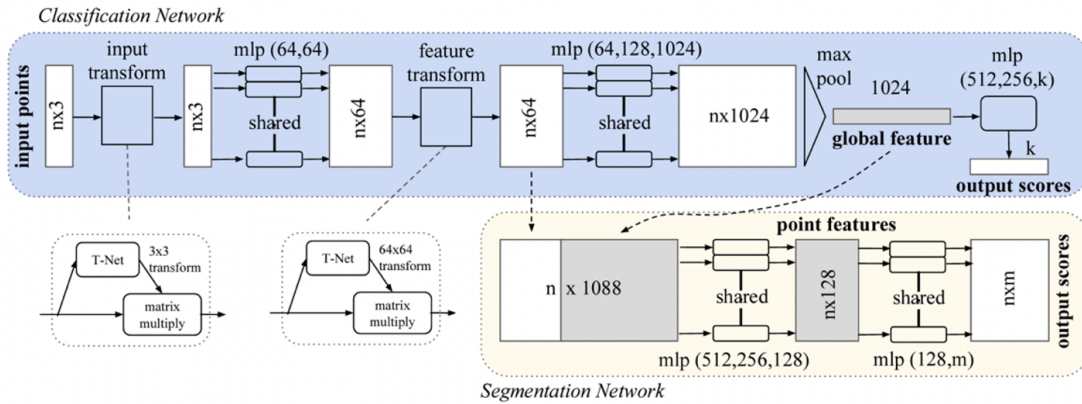


Figure 2.8: Architecture of PointNet network (Source: Qi et al., 2017a)

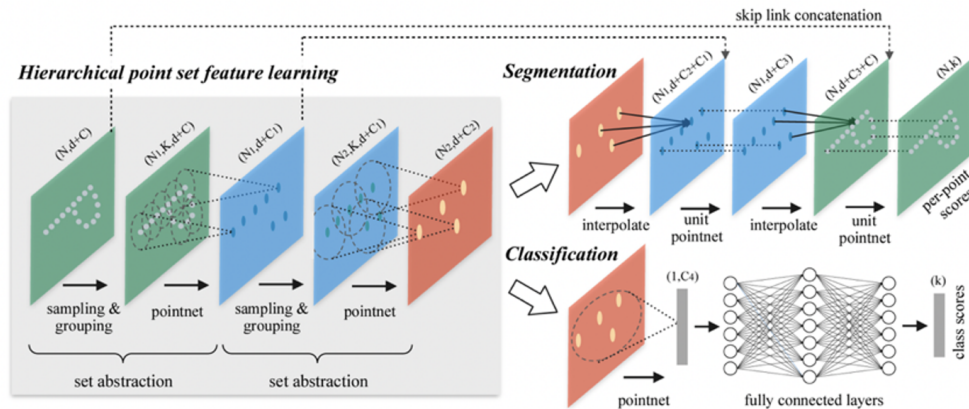


Figure 2.9: Architecture of PointNet++ network (Source: Qi et al., 2017b)

PointNet is one of the pioneering deep learning algorithms in directly processing point clouds. While there were previous attempts to apply deep learning to 3D data, PointNet was the first ever to operate directly on unstructured point clouds without requiring conversion to 3D voxel grids or multi-view images (Qi et al., 2017a). This represented a significant advancement in this field that allowed the processing and analyzing of raw point clouds while preserving the fine-grained spatial locality of the data, which may be lost in other representations such as voxel grids and some 3D-2D approaches such as Multi-view Convolutional Neural Networks (MVCNN) (Jiang et al., 2019) and SnapNet (Boulch et al., 2018).

PointNet++ (Qi et al., 2017b) is proposed as an advancement of PointNet (Qi et al., 2017a) later in the same year and addresses one of its main limitations: detecting local details within point clouds. It employs a hierarchical neural network that applies PointNet recursively on nested subsets of point clouds to capture local geometric patterns based on neighbouring feature pooling, a general workflow is illustrated in Figure 2.9.

Lu et al. (2023) introduce Dynamic Clustering Transformer-based Network (DCTNet), a transformer-based network specifically engineered for semantic segmentation tasks. The model’s architecture (see Figure 2.10) is specifically tailored to address the inherent challenges in semantic segmentation, emphasizing the enhancement of local semantic homogeneity.

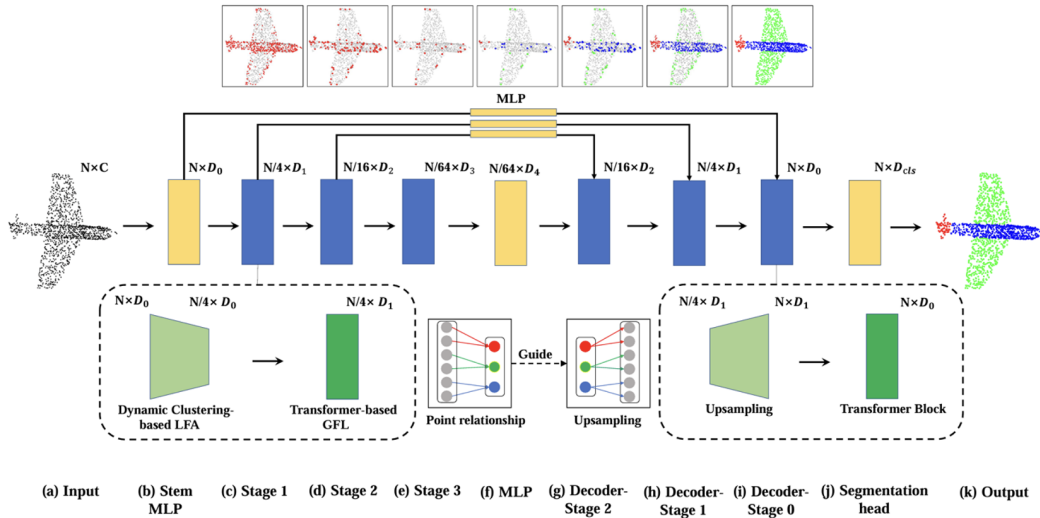


Figure 2.10: Architecture of encoder-to-decoder structure in DCTNet (Source: Lu et al., 2023)

In **DCTNet**, each encoder stage comprises two pivotal blocks: a dynamic clustering-based **Local Feature Aggregating (LFA)** and a transformer-based **Global Feature Learning (GFL)** Blocks. The **LFA** block is instrumental in ensuring discriminative local feature extraction, while the **GFL** block, with its dual attention mechanism, is well designed to capture long-range contextual relationships within the data. Similar to the encoder, the decoder follows a U-Net design ([Ronneberger et al., 2015](#)) and includes stages with a semantic feature-guided upsampling block and a transformer-based **GFL** block, paralleling those in the encoder. This symmetric design is crucial for reconstructing detailed semantic information from compressed feature representations. The output of this process is the generation of a final point prediction through a **MLP** head layer, which consists of two linear layers enhanced with batch normalization and **Rectified Linear Unit (ReLU)**. This layer not only provides the final output but also ensures the preservation of critical semantic details.

The integration of the **LFA** and the **GFL** blocks within **DCTNet** represents a significant advancement in addressing the demands in semantic segmentation by effectively harmonizing local and global feature representations ([Lu et al., 2023](#)).

The model employs Semantic feature-based Dynamic Sampling **Semantic Feature-based Dynamic Sampling (SDS)**, a process involving local density computation, distance indicator calculation, and an innovative scoring and sampling technique. This method involves calculating the local density d_i for each point p_i based on its **KNN** (Φ_i) in the feature space. With given an input point set P :

$$P = \{p_i\}_{i=1}^N \in R^{N \times D} \quad (2.13)$$

where D is the dimension of the input. The local density d_i can be calculated as:

$$d_i = \exp \left(-\frac{1}{k} \sum_{p_j \in \Phi_i} \|p_i - p_j\|^2 \right) \quad (2.14)$$

Next, a distance indicator δ_i for each p_i can be computed and defined as:

$$\begin{cases} \min_{d_j \in \Omega_i} \|p_i - p_j\|^2, & \text{if } \Omega_i = \emptyset \\ \max_{d_j \in \mathbb{R}} \|p_i - p_j\|^2, & \text{otherwise} \end{cases} \quad (2.15)$$

where $\Omega_i = \{d_i \in \Gamma \mid \forall d_j > d_i\}$. δ_i represents the minimal feature distance between a point p_i and any other points with higher local density, facilitating the scoring of each

point by combining d_i and δ_i . This scoring system prioritizes points with higher scores for sampling, dynamically updating this selection based on semantic features.

DCTNet's local feature aggregation addresses the limitations of conventional averaging methods by implementing a set of learnable attention scores $A = \{a_i\}_{i=1}^N$. This set allows for the calculation of the significance of each point within a cluster, enhancing the representation of individual points' importance and mitigating information loss. Specifically, for a point cluster $C = \{C_{si}\}_{i=1}^S$, the significance of each point si can be calculated as follows:

$$s_i = \frac{\sum_{j \in C_{si}} \exp(a_j) C_{si_j}}{\sum_{j \in C_{si}} \exp(a_j)} \quad (2.16)$$

where a_j is the learnable attention score for C_{si_j} within the cluster. Meanwhile, the relationship between the sampling points and cluster points in the previous step is also stored for the point cloud upsampling in the decoder. Furthermore, the model integrates a cross-attention Transformer to enhance the features of sampling points and reduce information loss during aggregation. This is achieved through the generation of Q_{query} , K_{key} , and V_{value} with learnable weight matrices W_Q , W_K , and W_V :

$$Q_{query} = SW_Q \quad (2.17)$$

$$K_{key} = PW_K \quad (2.18)$$

$$V_{value} = PW_V \quad (2.19)$$

An attention map M is then calculated (see Eq. 2.20), integrating feature similarities and point importance, to produce an enhanced set of sampling points.

$$M = softmax \left(\frac{QK^T}{\sqrt{D}} + A \right) \quad (2.20)$$

Additionally, **DCTNet** includes a Semantic Feature-guided Upsampling Block, which effectively upscales point clouds by transferring semantic features from sampling points to corresponding cluster points (Lu et al., 2023).

Chapter 3

Data Collection

The data collection in the research can be summarized into Stage 1 of the general workflow illustrated in Figure 3.1 below. Overall, Stage 1 covers Sections 3.1 to 3.3, introduces the study area, outlines the data collection process, and describes the preprocessing steps required to convert raw data into labelled point clouds. This phase is foundational for the subsequent analysis and findings presented in later chapters of the thesis.

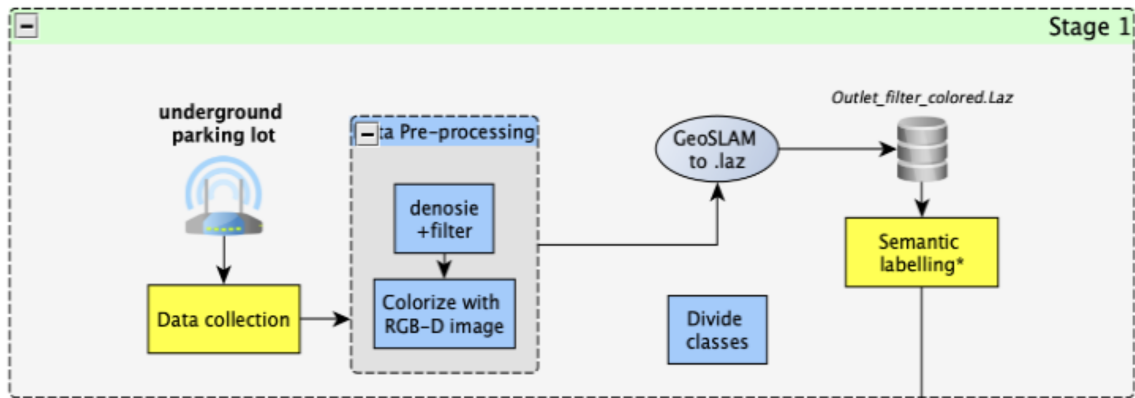


Figure 3.1: General workflow

3.1 Study Area

The study area encompasses an indoor parking garage located near Toronto Premium Outlets, located at 13850 Steeles Avenue, Halton Hills, ON L7G 0J1, Ontario, Canada.

The Outlet located near a key junction of major highways (Hwy 401), on the outskirts of the [Great Toronto Area \(GTA\)](#), provides easy access while escaping from downtown congestion, making it an attractive spot for consumers in search of a stress-free shopping experience. This parking facility spans four levels: L1, L2, L3, and L4. L4 has been designated as the targeted area for data collection. The location of the parking garage and its aerial perspective are illustrated in Figure 3.2.



Figure 3.2: Location of the study area (Source: [Google OpenStreetMap](#))

3.2 Data Collection

The raw 3D [Light Detection and Ranging \(LiDAR\)](#) point clouds were collected using GeoSLAM ZEB Horizon laser scanner (with datalogger) installed with the ZEB Vision camera (to generate corresponding RGB-D images).

3.2.1 Hand-held Laser Scanner

The GeoSLAM ZEB Horizon is known for its portability and versatility in capturing 3D spatial data. The details about hardware and accessories are demonstrated in [Tables 3.1, 3.2, and 3.3](#).

Table 3.1: GeoSLAM ZEB Horizon technical specification

Parameter	Specification
Range	100 m
Laser	Class 1 eye-safe / λ 903 nm
Field of view (FOV)	$360^\circ \times 270^\circ$
Processing	Post
Datalogger carrier	Backpack / Shoulder strap
Scanner weight	1.45 kg
Datalogger weight	1.4 kg
Colourised point cloud	✓
Real-time processing	✓
Data collection speed	300,000 points per second
Number of sensors	16
Vertical angular resolution	2°
Horizontal angular	0.2°
Relative accuracy	up to 6 mm (1-3 mm)
Raw data file size	25-50 MB/ minute
Battery life	Up to 2 hours of continuous scanning

The scanner reaches up to 100 meters and it is capable of operating under varying lighting conditions. The datalogger utilizes the [SLAM](#) algorithm in GeoSLAM for real-time data processing and point cloud generation; however, this feature is only for the ZEB Horizon RT, which is an advanced version of the standard ZEB Horizon. For the ZEB Horizon, post-processing is often performed with GeoSLAM-related software such as

Table 3.2: GeoSLAM ZEB Horizon configuration




Scanner	Datalogger
	
(Source: GeoSLAM)	

Table 3.3: GeoSLAM ZEB Vision camera accessory

Resolution	4K Panoramic
Capture Rate	2FPS
Colour	RGB
ZEB Vision Camera (Source: GeoSLAM)	

GeoSLAM Hub, Connect, Draw, etc., which efficiently processes the SLAM algorithm to produce detailed 3D data and maps.

Post-processing aims to transform raw, unstructured points into valuable, precise, and meaningful information. The processed data can be outputted in common point cloud formats, ensuring compatibility with a range of third-party post-processing tools such as CloudCompare, MeshLab, etc. The ZEB Horizon operates independently of the [Global Positioning System \(GPS\)](#), enabling the mapping of multi-level structures and seamless indoor-to-outdoor transitions with multi-scans. It is specifically designed for complex industrial settings and can navigate challenging terrains with ease. The battery life lasts up to 2 hours of continuous scanning, with an alternative option for field-swappable batteries.

The GeoSLAM ZEB Horizon uses [LiDAR](#) sensors to measure distances by illuminating a target with laser light and measuring the reflection with a sensor. The time it takes for the laser to reflect to the sensor is used to calculate the distance, and thus, create a 3D representation of the scanned environment. The [LiDAR](#) technology enables the ZEB Horizon to quickly capture numerous amounts of spatial data by emitting thousands of laser pulses every second. The data collected is then processed using [SLAM](#) technology, which allows the ZEB Horizon to produce 3D maps of indoor, underground, and difficult-

to-navigate spaces in real-time, without the need for the [GPS](#).

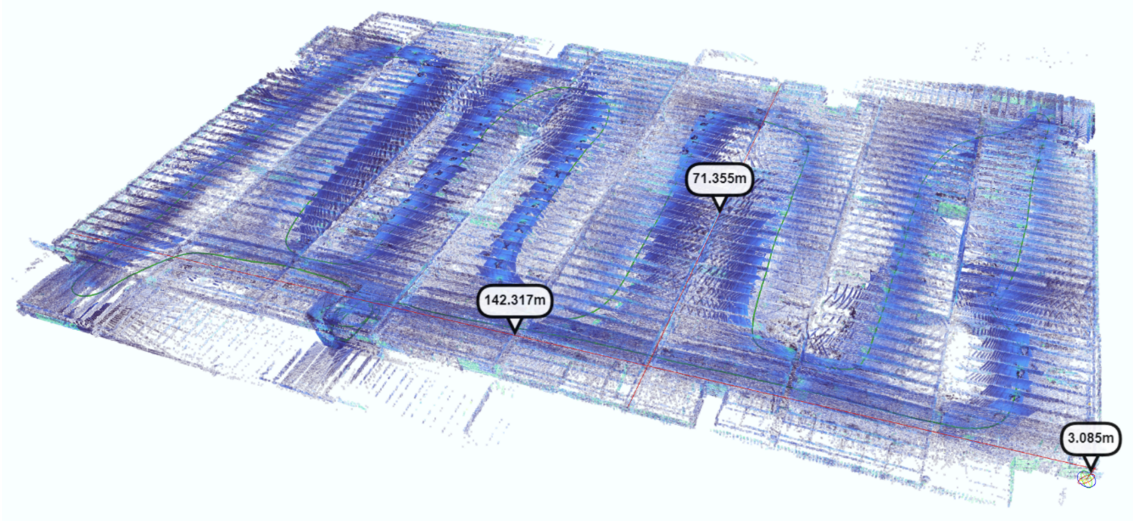


Figure 3.3: Overview of the raw point clouds of the parking garage (filtered.laz)

$$Density = \frac{Mass}{Volume} \tag{3.1}$$

Table 3.4: Parameter of the indoor parking garage scene

Parameter	Specification
Sensor	GeoSLAM ZEB Horizon Laser Scanner (LiDAR)
Area	10,155.03 m ²
Volume	433,702.84 cubic units
Density	223.6 mass units/cubic volume
Size	1.0 GB
Collected Date	April.29 th , 2023

The point clouds we collected at outlet level 4 are illustrated as a bird view image in Figure 3.4. Based on Equation 3.1, the calculated volume is approximately 433,703 cubic units. Subsequently, using the given mass, the density is determined to be roughly 223.60 mass units per cubic volume unit. These values are recorded in Table 3.4.

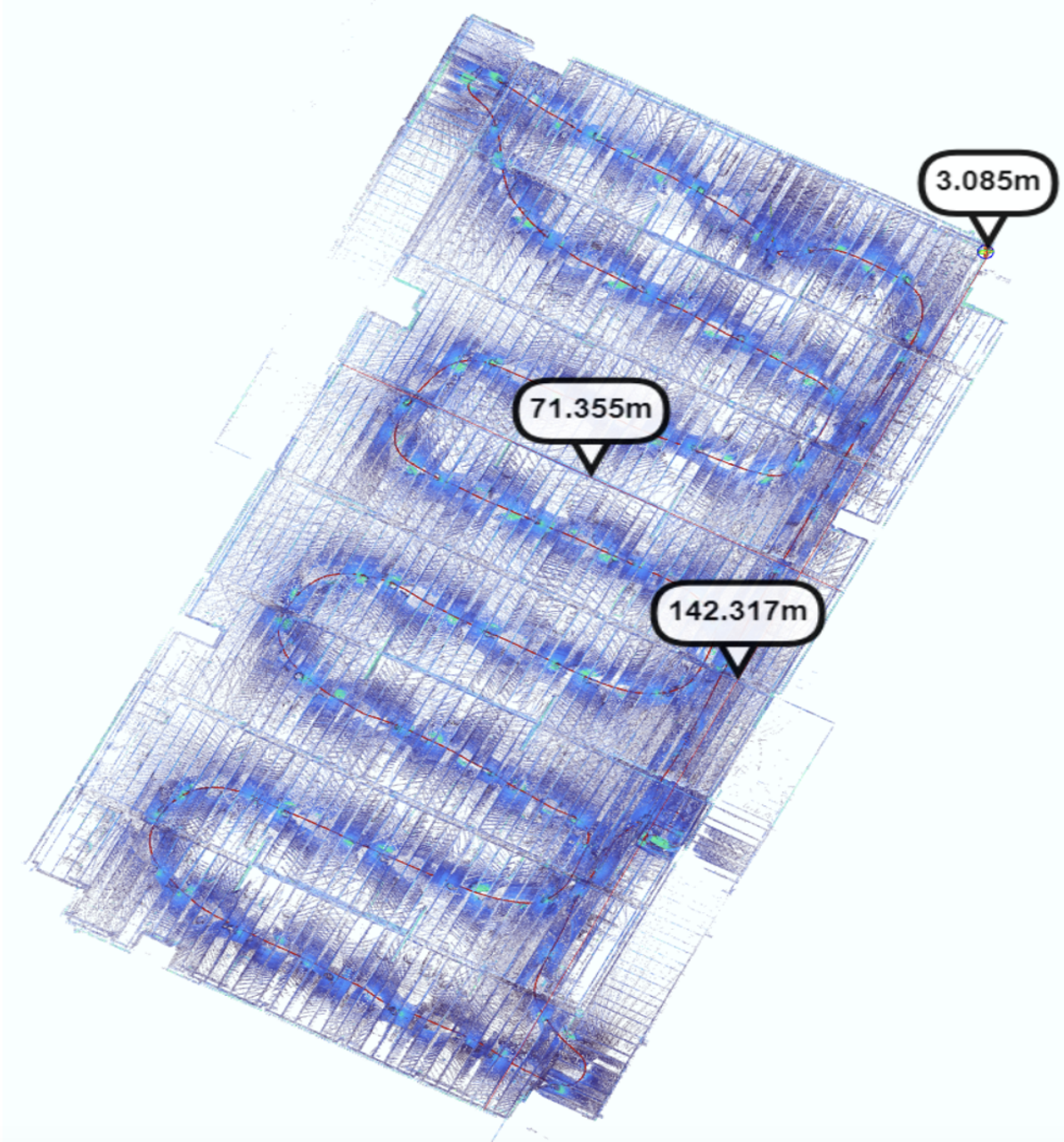


Figure 3.4: Bird view of the raw data at level 4 in outlet parking garage

3.2.2 Setup and Trajectory

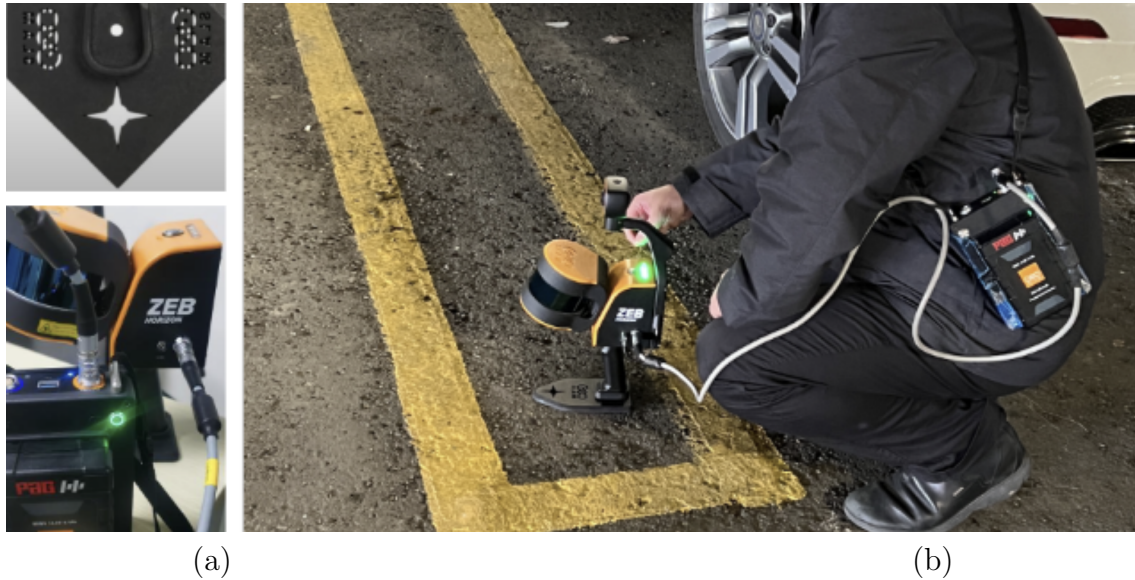


Figure 3.5: Devices setup (a): ZEB Horizon and hollow-out cross; (b): Control point alignment testing

The start and stop controls for the scanning operation are managed via the button located on the scanner head, as depicted in Figure 3.5. Additionally, the scanner head features a rectangular [Light-emitting diode \(LED\)](#) light, synchronized with the processor light, which collectively serves as operating indicators. These lights show different statuses: a red sequential light signifies that the processor is ready for operation; a flashing yellow light indicates that the scanner is still in the initiation phase; and a solid green light denotes active data collection.

To initiate data collection, after assembling and connecting the ZEB Vision camera, laser scanner, and datalogger as demonstrated in Figure 3.5, activate the set of scanners by pressing the button on either the scanner head or the datalogger. Wait for about 40 seconds until the red sequential light on the scanner head switches to a flashing yellow light (or the blue light on the datalogger processor changes to a red flashing light), indicating that the scanner head is in the preparation stage for scanning. Both lights will flash approximately 11 times before the scanner head's [LED](#) remains continuously lit. During this interval, the scanner must be kept stationary, as any detected movements by the [Inertial Measurement Unit \(IMU\)](#) will prevent the initiation of the scanning operation. Data collection can begin once the laser light [LED](#) turns green and starts to rotate.

During the scanning procedure, upon encountering a control point, the operator should gradually squat and precisely align the center of the hollowed-out cross on top of the control point. This position should be maintained stationary for 5 seconds to ensure successful capture of the control point. In terms of movement, it is essential to turn or rotate with caution, avoiding rapid adjustments in the scanning direction. Where feasible, scanning should be performed in a closed loop, as depicted in Figure 3.6, with the same start and end point connected by a circular trajectory for enhancing the IMU accuracy. It is recommended to limit the duration of each scanning session to around 25 minutes. During scanning, the movement should be consistent and at a slower pace akin to a leisurely walk.

Upon completion of data collection, the operator must remain immobile for 5 seconds before activating the termination process. This involves pressing the button located on the laser head or the corresponding button on the processor. The button should be held down until the laser head stops rotating. At this time, the processor's data indicator will illuminate yellow, signalling the initiation of data storage. Once the light goes off, the data storage is completed. Only after the indicator light turns off can the laser head be restarted to collect data for the next operation stage.

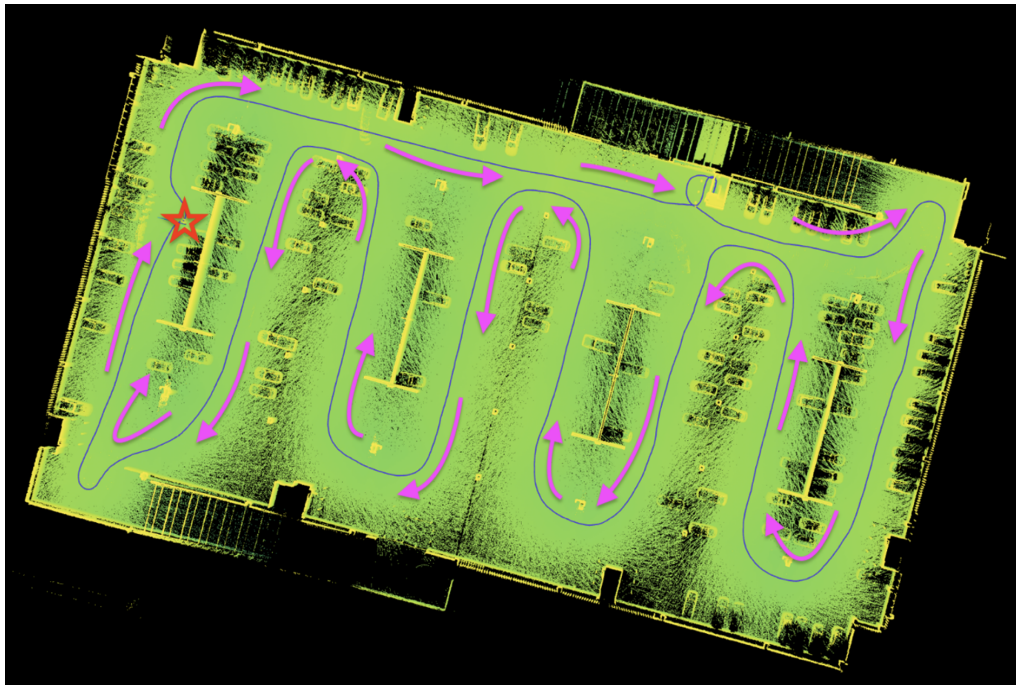


Figure 3.6: Trajectory and moving direction (star: start and endpoint; arrow: direction)

3.3 Data Preprocessing

Subsequently, the colourized and refined point clouds will be segmented into distinct classes based on the pre-defined labels, such as walls, floors, pillars, etc.

3.3.1 “.GeoSLAM” to “_filter_colored.laz”

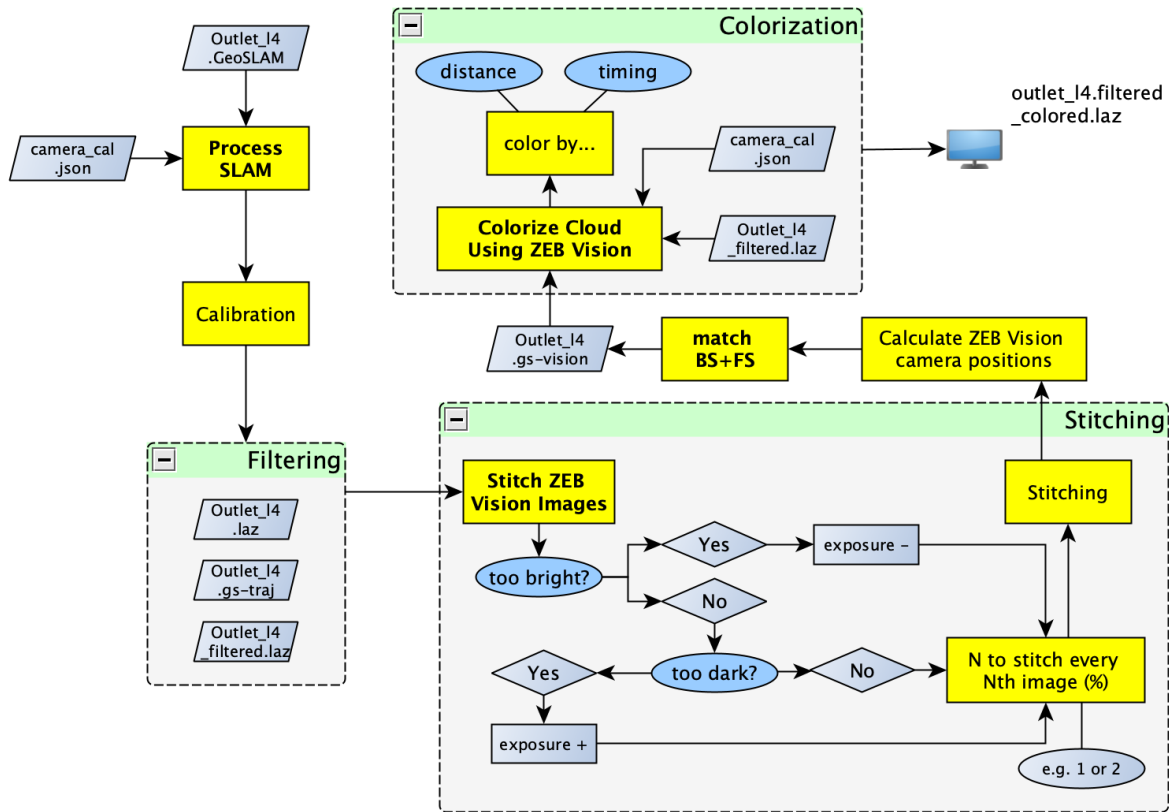


Figure 3.7: Workflow of “Process SLAM” on GeoSLAM Connect

Upon completion of the data collection phase, the raw point clouds are imported into software for preprocessing. During this stage, the raw data will be filtered, colourized with panoramic RGB images, and exported as a .laz file, which is compatible with many third-party processing tools, including CloudCompare. Figure 3.7 illustrates a clear and detailed

workflow for the process preceding colorization. Initially, the “Outlet_14.GeoSLAM” must undergo filtering and calibration. During this stage, the ‘Point Resolution’ option was maintained at its default setting, and the ‘Smooth and Clean Data’ option was checked. This procedure results in the generation of three distinct files: a standard LAZ file, a ‘*gs – traj*’ file (which outputs a trajectory similar to the one shown in Figure 3.6), and a filtered LAZ file.

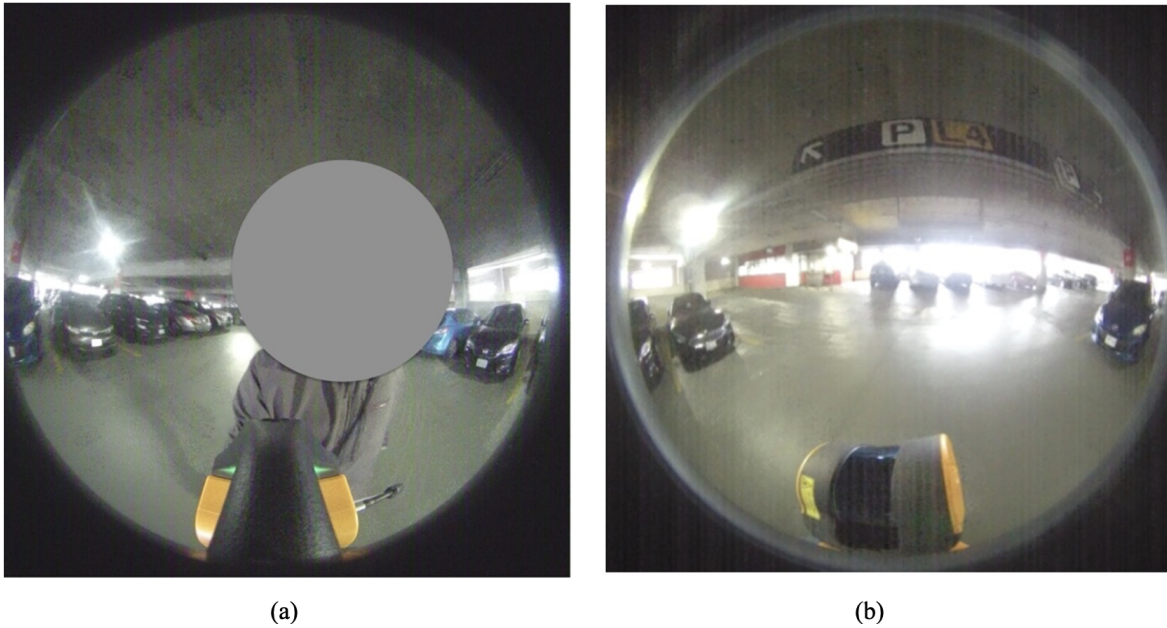


Figure 3.8: Image capture by ZEB Vision camera: (a) BS image at Spot ∂ (BS_{∂}); (b) FS image at Spot ∂ (FS_{∂})

Before moving to the colorization stage, there is a stitching process that aligns **Backsight (BS)** and **Foresight (FS)** images for each site. This alignment is achieved by pairing **BS** and **FS** images of the same site, as demonstrated in Figure 3.8, to create panoramic sets. The process involves inputting the corresponding GeoSLAM dataset, trajectory file (*gs-traj*), timing file (*imageTiming.json*), and calibration file (*camera_cal.json*). Subsequently, an image pose file was generated, as shown in Figure 3.9. Each point in Figure 3.9 represents a site which contains a set of **BS + FS** to form an individual panoramic. For each **BS+FS** site ($FS_{\partial} + BS_{\partial}$), a panoramic view can be accessed by clicking on the designated point located on top.

The next step is to utilize the ‘colourize cloud using the ZEB Vision tool. This requires

the input of the filtered. laz file, the image pose file, and the camera calibration file. The operator has the option to mask certain areas of the image, depending on the scanning method used, hand-held or backpacked. This masking aims to exclude irrelevant information, such as the operator’s face, and the person’s outline or silhouette demonstrated in Figure 3.8. In our study, since the scanner was handheld during data collection, the ‘hand-held’ option will be selected for this phase. In addition, the method of cloud colorization is chosen between timing and distance options. The timing option typically processes at a faster pace, whereas the distance option, though slower, yields better results. For low-resolution mode, checking the box accelerates the process, whereas leaving it unchecked enhances the quality of the result. Furthermore, another option is provided to remove uncolored points, resulting in cleaner data output. However, it is important to note that this may also lead to the loss of crucial data, leaving the box unchecked can preserve more detail in the point clouds. For data quality preservation, we selected the “by distance” method, unchecked low-resolution mode during the pre-processing stage.

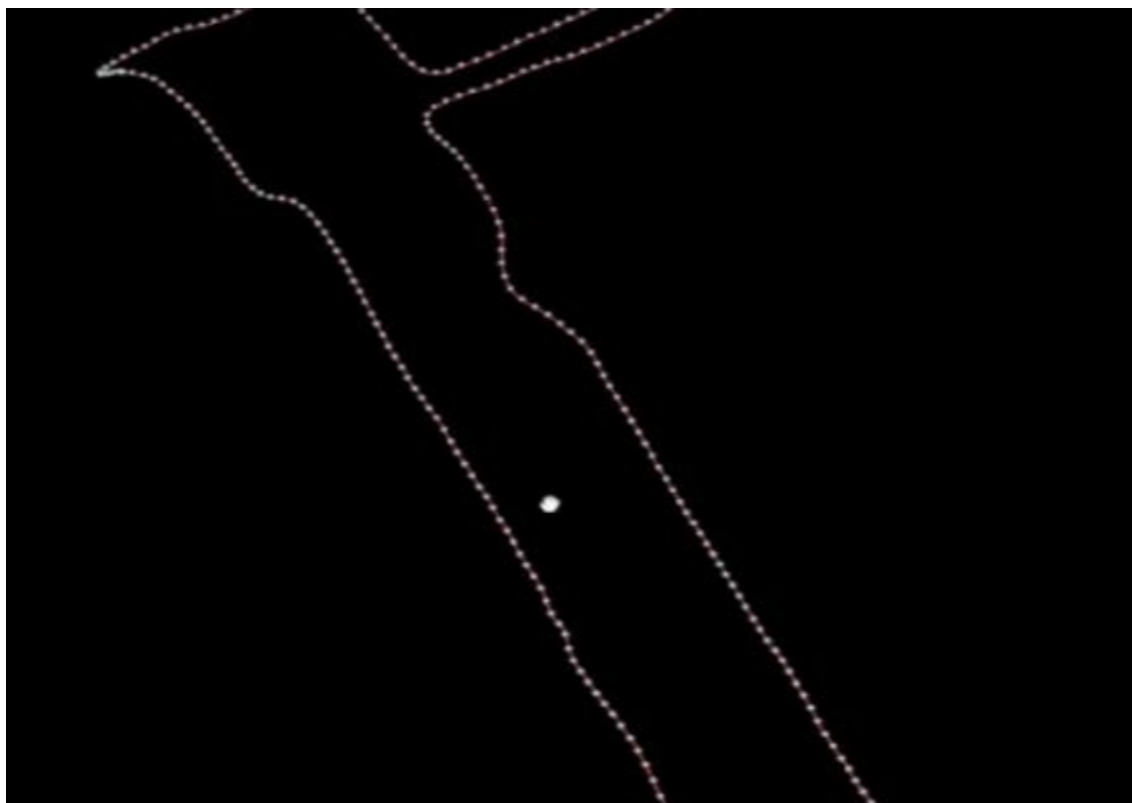


Figure 3.9: Example of an Image Position file

Chapter 4

Semantic Segmentation and Surface Regularization

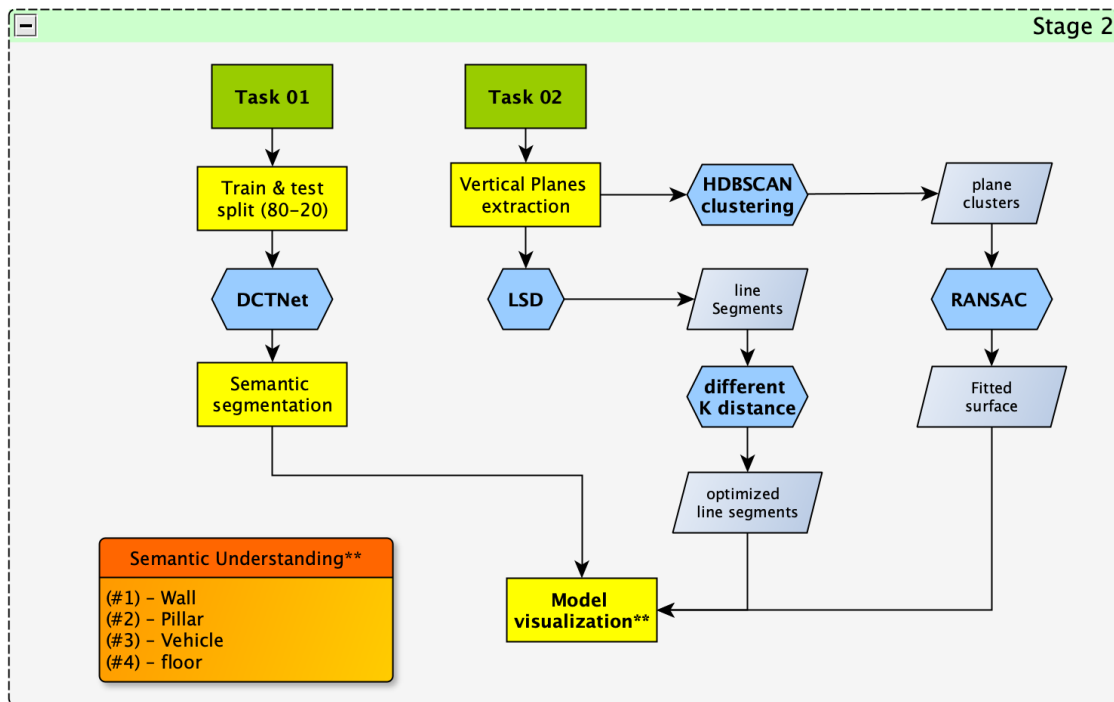


Figure 4.1: Stage 2 of the overall workflow

This chapter comprises Sections 4.1 to 4.4, which delves into analytical and optimization techniques for semantic understanding using a transformer-based network called [DCTNet](#) (Lu et al., 2023) for label understanding. Followed by a model reconstruction method that combines a [Hierarchical Density-based Spatial Clustering \(HDBSCAN\)](#) clustering, a [pyRANSAC-3D](#) plane fitting (Mariga, 2022), a revised [RANSAC](#) plane fitting, and a 3D line detection (Lu et al., 2019) algorithm.

4.1 Data Pre-processing



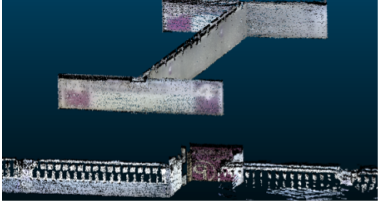


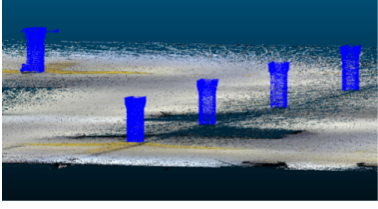

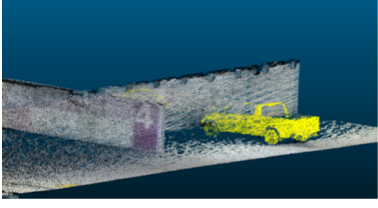

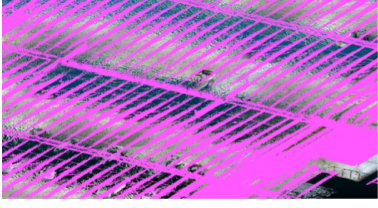

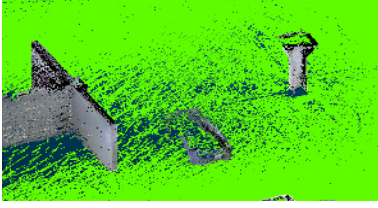
4.1.1 Semantic Labelling

Seven distinct classes have been delineated, each representing a unique semantic entity within the dataset. These classes, along with their corresponding labels and detailed definitions, are as follows:

- **Bollard** (Class 01): small vertical structures designed for safety and security purposes, serving as physical barriers between pedestrians and vehicles.
- **Wall** (Class 02): vertical structures enclose spaces, serving as boundaries or edges.
- **Pillar** (Class 03): vertical and cylindrical structural support element designed to bear the load of the parking garage’s upper floors.
- **Vehicle** (Class 04): motorized transportation (e.g. sedans, trucks, SUVs, etc.)
- **Ceiling** (Class 05): upper horizontal surface engineered for structural support.
- **Floor** (Class 06): lower horizontal surface marked for vehicular movement/ parking.
- **Others/ Unclassified** (Class 07): other unclassified object points.

Subsets of each class have been generated in both RGB and scalar field formats, as presented in Table 4.1. This table also delineates the point count for each label, revealing a relatively smaller number of points in the bollard (127,625) and pillar (469,885) classes. The wall class comprises 5,842,108 points, and the vehicle class contains 1,439,990 points. A higher point count is observed in the floor class (25,252,755), while the ceiling class possesses the most points within a single category, 35,640,963 in total. The unclassified category holds 4970 points. In sum, the entire point cloud dataset encompasses 68,778,296 points post-filtering and smoothing, before any segmentation processes.

Table 4.1: Example sub-sets for each class

Label	RGB	Scalar Field	Points
0 - Bollard			127,625
1 - Wall			5,842,108
2 - Pillar			469,885
3 - Vehicle			1,439,990
4 - Ceiling			35,640,963
5 - Ground			25,252,755
6 - Other	\	\	4,970
Total			68,778,296

4.1.2 Ceiling Removal

In the context of developing and reconstructing an indoor model, the significance of the ceiling class is diminished due to its lack of interaction with navigable surfaces. Furthermore, ceiling points can obscure other crucial features when viewed from above. Therefore, segmenting and removing ceiling points could enhance the visibility of other important classes, which visualization and evaluation can be facilitated more easily in subsequent stages. The differences in before vs. after ceiling removal are presented in Figure 4.2. Following the successful segmentation of the ceiling class, a similar approach can be applied to remove the lower horizontal planes and the floor. This technique, detailed in Section 4.3.1 on surface reconstruction, helps to distinguish the surface of the parking lot from other structures and objects, further refining the accuracy of the model for autonomous navigation.

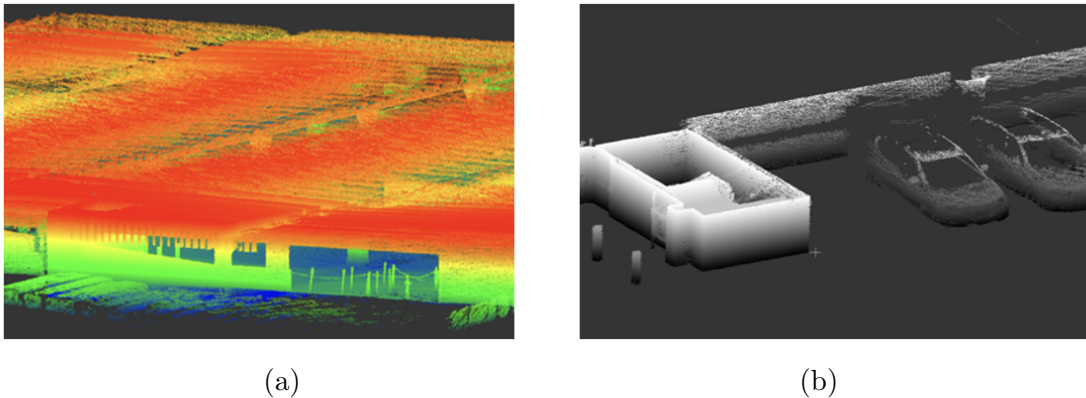


Figure 4.2: Ceiling Removal: (a) before ceiling removal; (b) after ceiling removal

4.1.3 Subsampling

For better performance efficiency and timesaving for data processing and analysis, the entire dataset has been downsampled by specifying a minimum spacing between points in the resulting subsampled point cloud using the ‘space’ subsample method in CloudCompare. This minimum spacing parameter is set by the user and determines how densely or sparsely the point cloud will be sampled. In our study, the minimum space parameter was set to be 0.05, which is relatively small compared to our large dataset. This can significantly reduce the overall data size while still retaining essential features of the data. (less data but remains fidelity to the original point clouds). A comparison between the original point cloud and sub-sampled point cloud is illustrated in Figure 4.3 and Table 4.2 respectively.

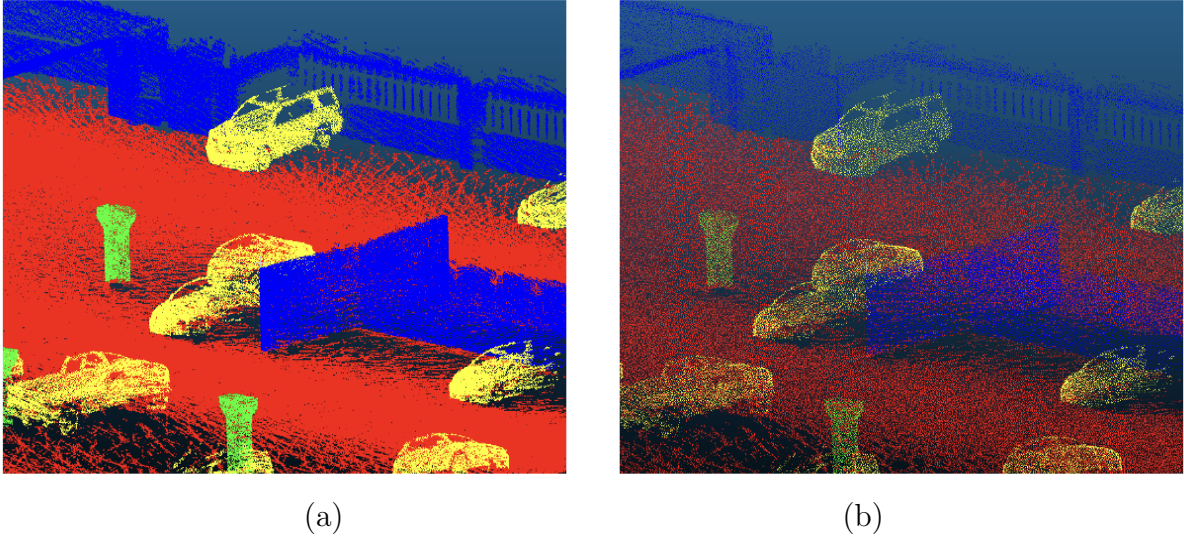


Figure 4.3: A comparison between the original and the sub-sampled data: (a) Original Density; (b) Sub-sampled Density

Table 4.2: Data size before vs. after sub-sampling for each class

Num.	Class	Before (# of points)	After (# of points)
1	Safety Bollard (label 0)	127,625	7,505
2	Wall (label 1)	5,842,108	429,729
3	Pillar (label 2)	469,885	34,128
4	Vehicle (label 3)	1,439,990	141,424
5	Ground (label 4)	25,252,755	1,573,007

Table 4.3: Data size before vs. after sub-sampling for classes in semantic segmentation

Num.	Class	Before (# of points)	After (# of points)
1	Wall (label 0)	5,842,108	429,729
2	Pillar (label 1)	469,885	34,128
3	Vehicle (label 2)	1,439,990	141,424
4	Ground (label 3)	25,252,755	1,573,007

The safety Bollard class in Table 4.2 has been removed for the semantic segmentation task due to the small size of the targets in this class. These targets are significantly smaller than those of other classes, making it difficult to achieve reliable segmentation results.

4.2 Semantic Segmentation

A [DCTNet](#) model was utilized to conduct a semantic segmentation task. A general hierarchical structure can refer to Figure 2.10 from Chapter 2. In the initial stage of processing, the original point cloud is fed into the encoder, where a stem [MLP](#) block projects the input data into a higher-dimensional feature space. Subsequently, these enhanced features undergo several stages of refinement, aiming to extract both local and global features effectively.

Compared with PointNet ([Qi et al., 2017a](#)) and PointNet++ ([Qi et al., 2017b](#)), the incorporation of a dual-attention Transformer-based [GFL](#) block in [DCTNet](#), utilizing Point-wise Self-Attention and Channel-wise Self-Attention mechanisms, enables a thorough capture of global features. This feature significantly improves the modelling of long-range context dependency and channel interaction analysis ([Lu et al., 2023](#)). Besides, this technique outperforms traditional point cloud interpolation methods, ensuring the preservation of detailed semantic information in the upsampling process, thereby enhancing the overall quality and accuracy of the semantic segmentation.

4.2.1 Input: Preparing Train and Test Sets

To prepare the training and testing dataset for the semantic segmentation task, a Pareto Principle (80-20 rule) has been adapted into the study. 80% of the dataset will be split into training sets while the remaining 20% will be put into a testing/validation set. Since spatial data are locational-based and spatial sensitive, meaning features across different regions of the dataset can vary significantly. Randomizing the splitting of the dataset might not capture the diversity effectively in both train and test sets. Therefore, for spatial data, manual sampling can ensure that both the training and testing sets represent the full variety of the data, which allows us to control for potential biases that a random split might introduce, especially in a spatial dataset where patterns are typically geographically clustered. An overview of the manually pre-sliced training and testing sets have been illustrated in Tables 4.4 and 4.5 below.

Table 4.4: Training sets

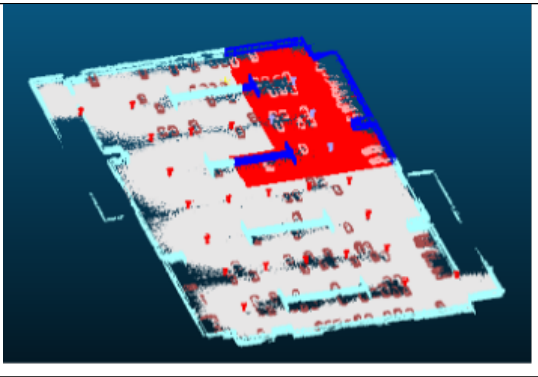
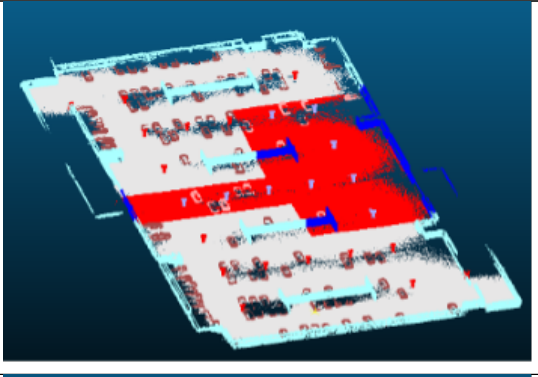
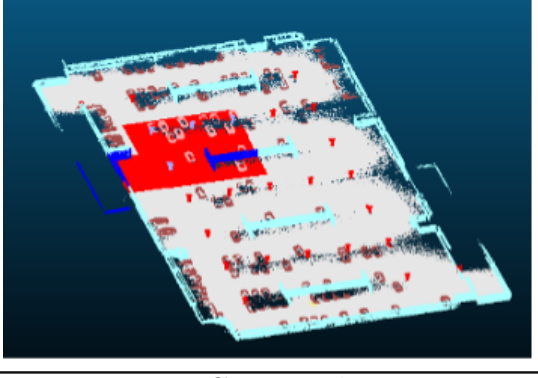
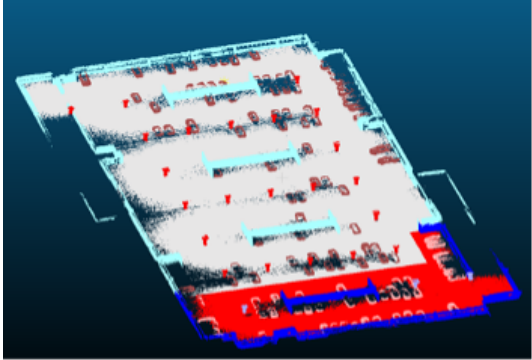
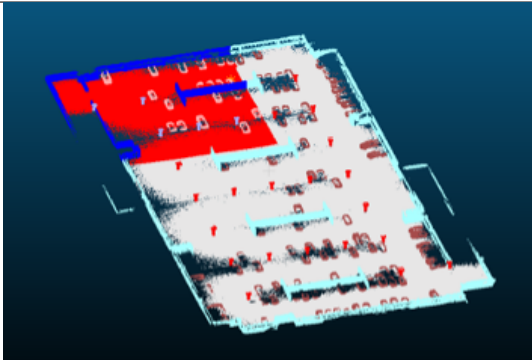
Sliced Set #	Point Size	Overview
01.txt	455,896	
02.txt	526,430	
03.txt	314,182	
Continued on next page		

Table 4.4 – continued from previous page

Sliced Set #	Point Size	Overview
04.txt	489,057	
All train (01-04.txt)	2,285,565	1,785,565 / 2,278,288 (78.4%)

* Account for about **80% (78.4%)** of the entire subsampled dataset.

Table 4.5: Testing sets

Sliced Set #	Point Size	Overview
00.txt	492,723	
All test (00.txt)	492,723	492,723/2,278,288 (21.6%)

* Account for about **20% (21.6%)** of the entire subsampled dataset.

4.2.2 Sample Generation

The sample generation script was tailored for preprocessing point clouds for segmentation tasks, where processing and reducing the size of point clouds while retaining their structural properties. *'Coordinate_normalize'* utilizes a standard approach to preprocess the point clouds to a normalized scale by calculating the mean and max distances and then normalizing the XYZ of each point by subtracting the mean and dividing by the max value. Then, a *'square_dist'* function computes the squared Euclidean distance between two sets of points while a *'knn_point'* function implements the **KNN** algorithm to find a specific number (*'n_sample'* = 4096) of nearest neighbours from a set of points (*'new_xyz'*) from a larger set (*'xyz'*). In addition, functions *'idx_cut_off'* and *'farthest_pt'* have been used for identifying the farthest point from a given seed point and cutting off point beyond a certain distance respectively.

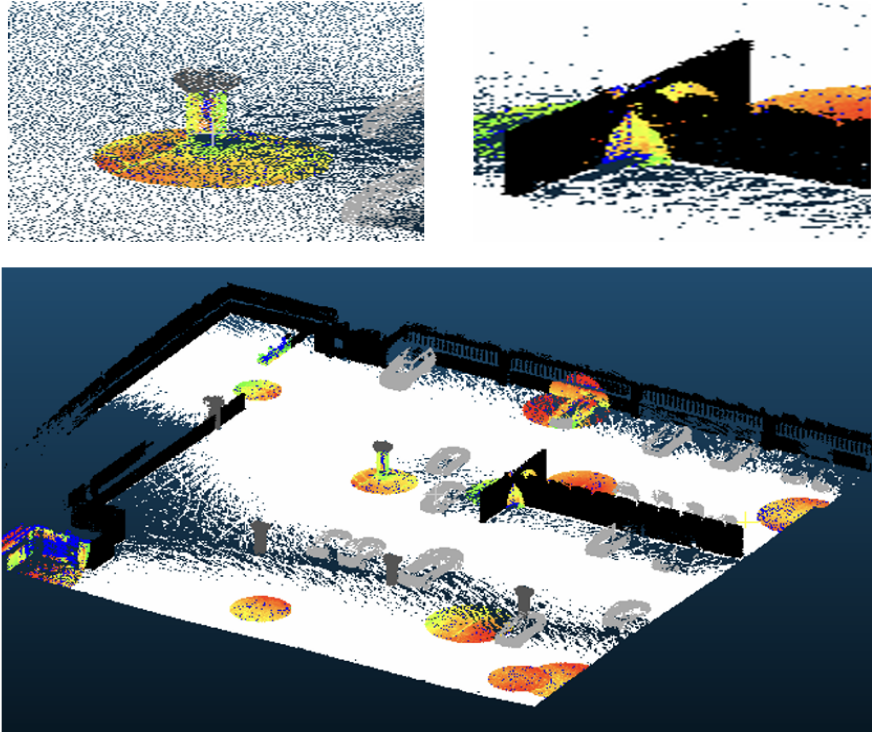


Figure 4.4: Examples of raw test sample (size=4096 points) from sample generation

The subsequent **Semantic Feature-based Dynamic Clustering (SDC)** method provides efficient and focused sampling output, which is approximately 4 times faster than traditional **Farthest Point Sampling (FPS)**. Apart from that, the method strategically retains fewer points in flat

areas while focusing on key areas such as the contours of pillars, vehicles, and other vertical features; thereby providing more informative data for network learning. Sampled clusters in test samples utilizing the above method have been demonstrated in Figure 4.4.

4.2.3 Weight

Weights are assigned in the loss function of the model to handle class imbalance. As Figure 4.5 indicates, a self-defined weight for each class will be generated and input into the [DCT-Net](#) model for later training purposes. Based on the calculation, the weights for classes 1 to 4 are 1.2672451707936863 (wall), 15.95675105485232 (pillar), 3.8506335558321076 (vehicle), and 0.34619807794879487 (ground).

```
def calculate_class_weights(num_samples_class_1, num_samples_class_2,
                           num_samples_class_3, num_samples_class_4):

    total_samples = num_samples_class_1 + num_samples_class_2 + \
                    num_samples_class_3 + num_samples_class_4

    weights = [total_samples / (4.0 * n) for n in
               [num_samples_class_1, num_samples_class_2,
                num_samples_class_3, num_samples_class_4]]

    return weights

weights = calculate_class_weights(429729, 34128, 141424, 1573007)
print(weights)
```

Figure 4.5: Calculate class weights

4.2.4 Model Training

To achieve global feature learning, a dual-attention transformer was integrated into the model for long-range context dependency modelling. The model was trained for 150 epochs with a batch size of 2. The [Stochastic Gradient Descent \(SGD\)](#) optimizer was used with a learning rate of 0.01, with a decay of 0.01% of every 5 epochs (step sizes).

4.3 Surface Sampling and Reconstruction

In Section 4.3, a comprehensive approach for 3D surface plane reconstruction has been presented (illustrated in Figure 4.6). The process begins by clustering the entire point cloud dataset using the **HDBSCAN** algorithm. Subsequently, the methodology incorporates the use of the **Ball-Pivoting Algorithm (BPA)** for plane mesh reconstruction. This algorithm is chosen for its effectiveness in generating meshes from 3D point clouds. Following the mesh reconstruction, a **RANSAC**-based approach is employed for plane fitting. This method is particularly adept at accurately identifying and fitting planes in the presence of noisy data or outliers. The combination of the **BPA** for mesh generation and the **RANSAC** for plane fitting provides a robust framework for precise and efficient 3D surface reconstruction. Finally, the generated models would be visually presented alongside the original, unprocessed point clouds, providing a clear comparison between the processed and raw point clouds.

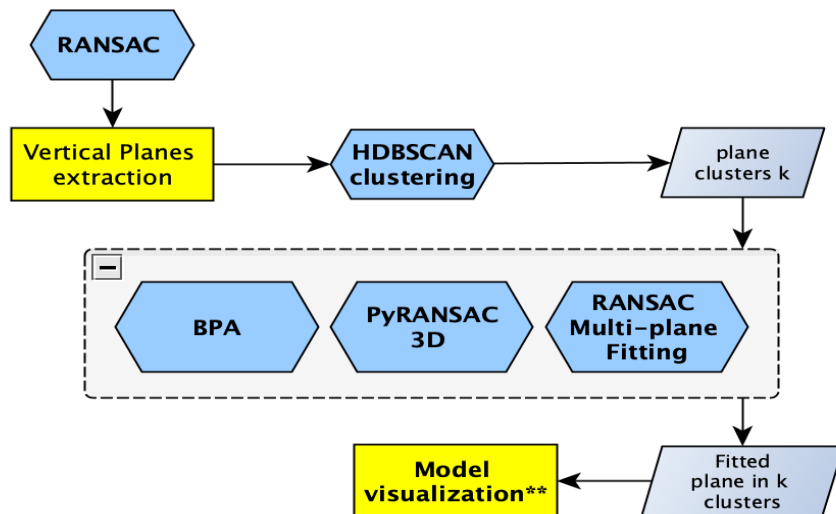


Figure 4.6: Detailed workflow of surface sampling and reconstruction

4.3.1 Vertical and Horizontal Plane Extraction

To extract vertical features (vertical planes like walls, pillars, etc.) from the horizontal surfaces for model reconstruction, a Python script `'detectHorizon'` has been created using `Open3D` (a library for processing 3D point cloud data) to perform plane segmentation on point cloud and visualize the results. The script is well-structured, with functions to create a mesh from plane

coefficients, display inliers and outliers, and the main function to load, process, visualize, and output the point cloud data. In the function `create_mesh_from_plane` (in Appendix A1), the coefficients of a plane (from the plane segmentation) are extracted to generate a mesh to represent the plane. The mesh is created using a grid of points in the X and Y dimensions, with the Z dimension calculated from the plane equation. This helper function can be used for visualizing the detected plane in the point cloud. The function `display_inlier_outlier` on the other hand visualizes the point cloud with inliers and outliers differentiated by colours (inliers as red [1, 0, 0]; outliers as grey [0.8, 0.8, 0.8]). Additionally, it shows the plane detected by the RANSAC as a green mesh.

To demonstrate the capability of detecting horizontal plane, a RANSAC plane segmentation has been performed in a small portion of the point clouds. Parameters have been tested and evaluated in the code chunk below (see Figure 4.7): distance threshold (`distance_threshold`) = 0.235, number of points randomly sampled in each iteration (`ransac_n`) = 50, and the number of iterations (`n_iterations`) = 1000. A plane equation is also generated in Figure 3.15a. by computing the a , b , c , d of the plane model where $ax + by + cz + d = 0$.

```
# Perform RANSAC plane segmentation
plane_model, inliers = pcd.segment_plane(distance_threshold=0.235,
                                         ransac_n=50,
                                         n_iterations=1000)

[a, b, c, d] = plane_model
print(f"Plane equation: {a}x + {b}y + {c}z + {d} = 0")
```

Figure 4.7: RANSAC plane segmentation and plane fitting equation

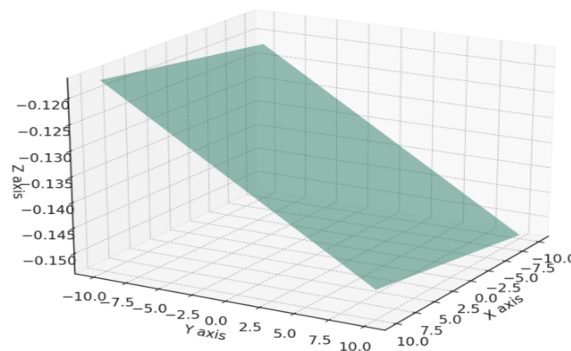


Figure 4.8: 3D plot of the plane (based on the computed plane fitting equation)

A 3D representation of the plane, derived from the given equation, is successfully visualized in Figure 4.8. This graphical depiction aids in understanding the spatial orientation and distribution

of the plane within a 3D coordinate system. Subsequently, to differentiate inliers and outliers within a sample data set, the point cloud is loaded and processed using the ‘*display_inlier_outlier*’ function. This procedure effectively segregates the inliers and outliers, facilitating a clear extraction. The outcome of this process is demonstrated in Figure 4.9, showing the distinct categorization in the original point cloud.

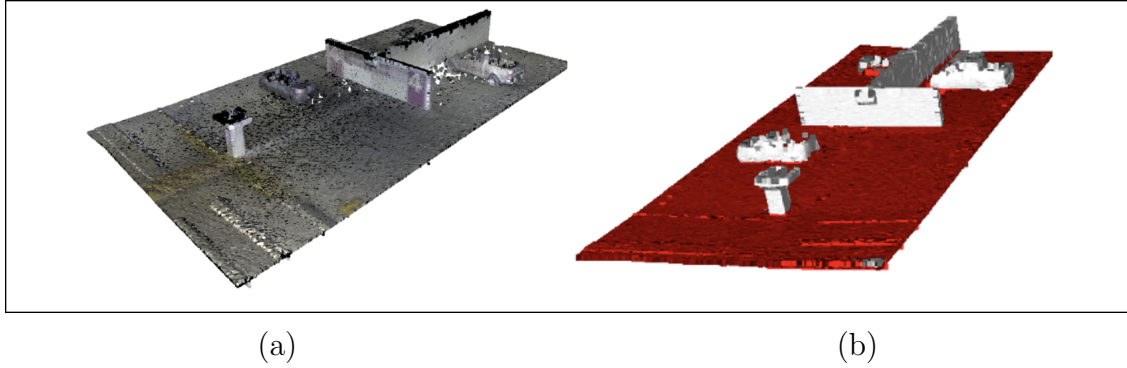


Figure 4.9: Horizontal Plane Extraction: (a) Raw point cloud cell; (b) detectHorizon output: Horizontal plane in red $[1, 0, 0]$ and all the other inlier points (vertical planes) in light grey $[0.8, 0.8, 0.8]$

The horizontal and vertical planes, classified as inliers and outliers, respectively, are efficiently extracted using the ‘*o3d.io.write_point_cloud*’ function. This process allows for the specification of a custom output path to save the extracted planes. Through this method, the two distinct sets of planes are segregated and stored for subsequent processing.

4.3.2 HDBSCAN Clustering

Before conducting pyRANSAC for plane fitting, the extracted vertical planes from Section 4.3.1 will be sampled and clustered using the HDBSCAN, which is an advanced clustering algorithm that works well with spatial data and it is capable of identifying clusters of varying densities. Parameters ‘*min_cluster_size*’ and ‘*min_sample*’ are tested and set for the clustering process as shown in the two-line code fragment below (for more details, see Appendix A2). After clustering, the script printed out the unique labels(*cluster_IDs*) and the estimated number of clusters of the input data. Then, it creates an output directory if it does not exist, where the clustered data will be saved. For each cluster, it saves the points belonging to that cluster in a separate text file in the output directory, while ignoring the noise points denoted as ‘-1’.


```
clusterer = hdbscan.HDBSCAN(min_cluster_size=900, min_samples=30)
labels = clusterer.fit_predict(xyz_points)
```

4.3.3 BPA

The **BPA** algorithm was implemented as one of the solutions for surface reconstruction as the Point clouds have been converted into mesh models using the Ball Pivoting Algorithm ([Bernardini et al., 1999](#)) provided by Open3D. The script iterates over files in a specified directory, checking for files that end with a `.txt` extension, indicating the ones that contain a point cloud. For each file, the script reads the point clouds in a [Numerical Python \(NumPy\)](#) array using the `load_points` function. The [NumPy](#) array which contains XYZ dimensions is then converted into an Open3D `PointCloud` object.

```
distances = pcd.compute_nearest_neighbor_distance()
avg_dist = np.mean(distances)
radii = [avg_dist, avg_dist * 2, avg_dist * 4]

# Ball Pivoting algorithm
bpa_mesh = o3d.geometry.TriangleMesh.create_from_point_cloud_ball_pivoting(
    pcd, o3d.utility.DoubleVector(radii))

all_meshes.append(bpa_mesh)
```

The average nearest neighbour distance among points has been computed and used to define a set of radii for the **BPA** (See the script chunk above). The radii of the **BPA** are chosen based on the average distance, with multiples of this distance to ensure robust meshing across different scales. Lastly, the generated mesh for each file is added respectively to a list called `all_meshes`, which can be saved as `.ply` to a specific directory.

4.3.4 pyRANSAC-3D (RANSAC Multi-Plane Fitting)

Alternatively, another encapsulated open-source tool, `pyRANSAC-3D`, proposed by Leonardo (2022) has been utilized for sampling and plane fitting. The `pyRANSAC-3D` package, a specialized tool for implementing the [RANSAC](#) algorithm ([Fischler and Bolles, 1981](#)) in 3D point cloud data, can be installed using standard Python package managers, either via Conda or pip3.

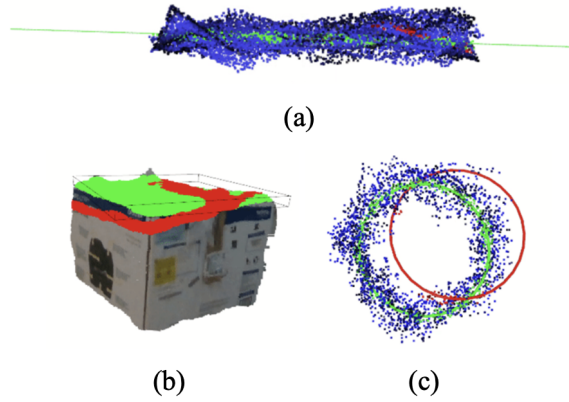


Figure 4.10: pyRANSAC-3D (a) line fitting, (b) plane fitting, and (c) circle fitting example (Source: Leonardo, 2022)

After loading the points from the given directory (*path*), a pre-defined class *plane* object (see the defined class in the script below), which finds the equation of an infinite plane using the **RANSAC** algorithm has been applied. The `.fit()` function then selects three random points from the point cloud (formatted as a **NumPy** array `pts` with shape `[N,3]`), identifies inliers within a defined threshold (*thresh*), and iterates up to *maxIteration* times to fit the optimized plane.

```

    Class Plane():
    | Plane()

plane1 = pyrsc.Plane()
best_eq, best_inliers = plane1.fit
    (pts, thresh=0.05, minPoints=100, maxIteration=1000)

```

The algorithm returns the parameters of the detected plane in the standard linear equation form: $Ax + By + Cz + D = 0$. These parameters are provided as a **NumPy** array of shape `[1, 4]`, representing the coefficients *A*, *B*, *C*, and *D* respectively. Besides, the method also identifies and returns the inliers from the input data, which are the points that are considered to be part of the plane within the specified threshold distance.

Given the presence of multiple clusters derived from the **HDBSCAN** clustering method, an integration approach has been applied by involving a for loop structure. This loop will iterate

through a designated folder that contains all clusters. The iteration mechanism will allow systematic processing of each cluster. By applying pyRANSAC-3D (Mariga, 2022) in the loop, each cluster’s planar features can be individually assessed and extracted, facilitating a comprehensive analysis of the 3D point cloud data. This structured approach ensures that each cluster is accurately processed, and the planar characteristics inherent within the spatial data are effectively captured.

4.4 Line Feature Reconstruction

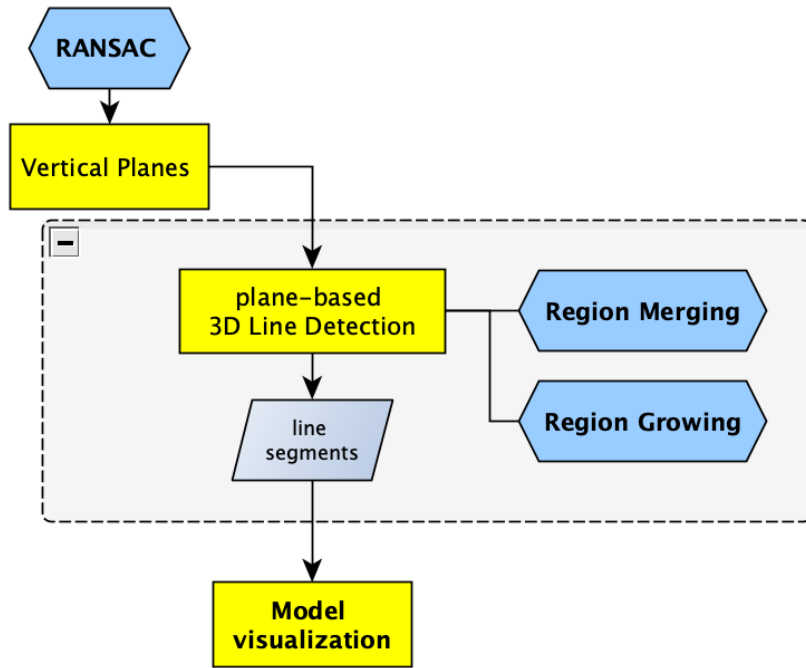


Figure 4.11: Detailed workflow of edge or line segments reconstruction

Section 4.4 elaborates on a methodical approach for reconstructing line segments in 3D point clouds, as shown in Figure 4.11. The procedure applied a 3D line detection algorithm on the vertical plane data, which was derived from Section 4.3.1. A region-growing and region-merging approach will be applied to the 3D line segments directly for plane-based 3D line feature detection.

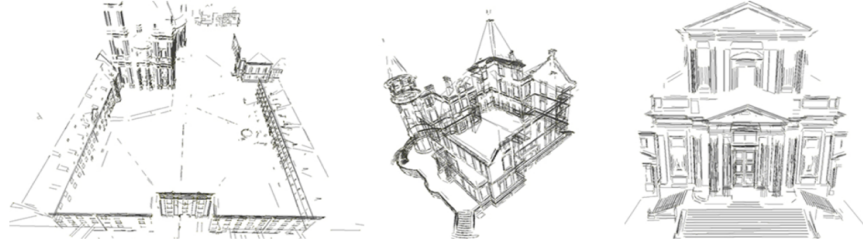


Figure 4.12: Sample outputs using LineDetection3D (Source: Lu et al., 2019)

4.4.1 3D Line Detection

For 3D line segment detection and generation, a C++ script was utilized. (demonstrated in Appendix B1). The general workflow was a three-step algorithm proposed as part of Lu et al’s work in 2019. Firstly, the unorganized point clouds were segmented into 3D planes using region growing. Then, in the second step, for each segmented plane, the points would be projected to form a 2D image, which would later be utilized for 2D contour extraction and Least Square Fitting to get the 2D line segments. Those 2D line segments would then be re-projected onto the 3D plane to get the corresponding 3D line segments. Lastly, a post-processing procedure is proposed to eliminate outliers and merge adjacent 3D line segments as the final output. The generated line segments should be similar to the output in Figure 4.12.

According to Appendix B1, the ‘*LineDetection3D*’ class is used to detect lines and planes in the raw point clouds; and the ‘*detector.run*’ method performs the core operation of extracting geometric features (lines and planes) from the data. The ‘*writeOutPlane*’ and ‘*writeOutLines*’ functions write the detected planes and lines to text files, which includes the coordinates and coded colours for visualization purpose. To initiate the process, defined functions would be called in the ‘main’ function including setting file paths, reading data, running the 3D line detection algorithm, and writing the output.

The script’s execution requires the installation of two essential packages: [Open Source Computer Vision Library \(OpenCV\)](#) (version higher than 2.4x) and [Open Multi-processing \(OpenMP\)](#). Additionally, CMake version 3.27.7 is employed for constructing the project environment. This includes the tasks of configuring and generating the solutions necessary for the script’s operation. For a visual guide on the setup process, refer to Appendix B2, which illustrates the steps for setting up and configuring the project environment using CMake. Once the environment is successfully configured, the corresponding files, outlined in Appendix B3 will be generated in the designated directory. Among these files, the ‘*LineFromPointCloud.sln*’ would be the CMake solution specifically designed to handle the 3D line detection task.

Chapter 5

Results and Discussion

The methods presented in Chapter 3 were tested and accessed on two distinct machines. The technical specifications for these machines are comprehensively detailed in Table 5.1. Specifically, Machine A was employed for specific tasks including data pre-processing, semantic segmentation, and line detection and reconstruction, as elaborated in Sections 3.3, 4.1, and 4.3, respectively. On the other hand, Machine B was utilized for the processing of surface sampling and reconstruction (Section 4.2). This strategic allocation of tasks across two machines facilitated a more efficient and efficient workflow utilizing the methods presented in Chapter 3.

Table 5.1: Machine configuration

	Machine A	Machine B
CPU	AMD Ryzen 7 5800x 8-Core (16 CPUs)	Apple M1 Pro
GPU	NVIDIA GeForce RTX 3080	Apple M1 Pro
RAM	64 GB	16 GB

5.1 Semantic Segmentation Result

5.1.1 Evaluation Metric

4 evaluation metrics are adopted in the segmentation task: Precision, Recall, Specificity, and F1 Score. They are interrelated and can provide a relatively balanced view of the model’s effectiveness. Precision measures the accuracy of the positive prediction. The ratio of correctly

predicted positive observations (True Positive) to the total predicted positives (True Positive + False Positive) can be represented as:

$$Precision = \frac{TP}{TP + FP} \quad (5.1)$$

, where TP refers to the number of positive cases correctly identified as positive; FP refers to the number of negative cases incorrectly identified as positive. Recall (a.k.a. sensitivity) measures how many actual positive cases were correctly identified, which can be referred as:

$$Recall = \frac{TP}{TP + FN} \quad (5.2)$$

, where FN refers to the number of positive cases incorrectly identified as negative.

Specificity measures the proportion of actual negative cases that were correctly identified. It's the ratio of correctly identified negatives to the total actual negatives, which can be written as:

$$Specificity = \frac{TN}{TN + FP} \quad (5.3)$$

, where TN refers to the number of negative cases correctly identified as negative.

The F1 Score is the harmonic mean of Precision and Recall. It is a balance between Precision and Recall, providing a single metric that accounts for both False Positives and False Negatives:

$$F1\ Score = 2 \times \frac{Precision \times Recall}{Precision + Recall} \quad (5.4)$$

Specifically, precision and recall compensate for each other, which means improving one might result in a trade-off with the other; F1 score on the other hand combines both metrics (precision and recall), offering a comprehensive evaluation of the entire model.

5.1.2 Results

Table 5.2: Testing sets breakdown

Class	Point Num
Wall (0)	101,143
Pillar (1)	6,057
Vehicle (2)	30,903
Ground (3)	354,620
All test (00.txt)	492,723

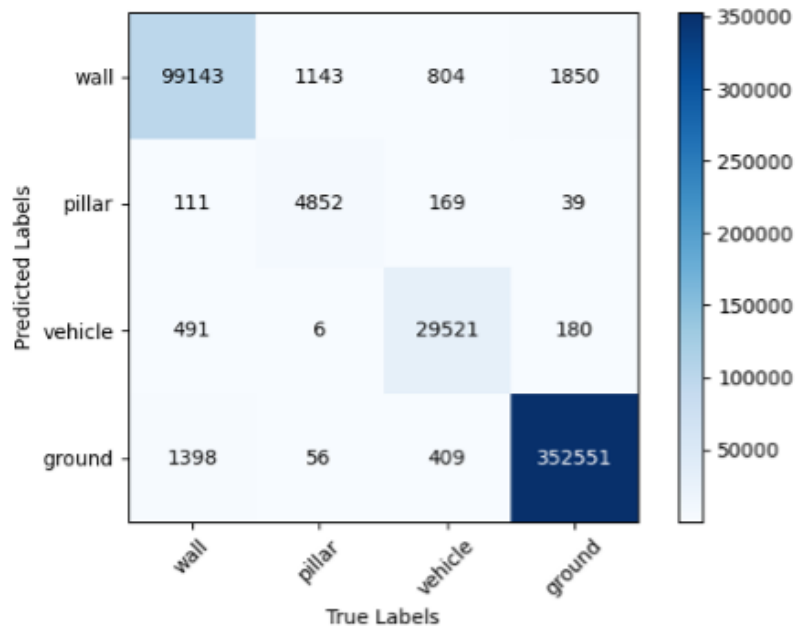


Figure 5.1: Confusion matrix for semantic segmentation task

According to Appendix C1, the overall accuracy and average F1 score of the model are around 98.65% and 94.88%. The average class Intersection Over Union (IoU) of the model is 90.74%, with the highest IoU (99.4%) in ‘ground’ and the lowest IoU (86.4%) in ‘pillar’. Similar patterns are also reflected in the Recall percentage, with the highest recall in ‘ground’ (99.4%) and the lowest recall in ‘pillar’ (80.1%) A confusion matrix illustrated in Figure 5.1 has been generated along with the evaluation process. The ‘TrueLabels’ on the horizontal axis at the bottom of the matrix refers to the labels that are true for the test dataset, and each Column’s true label’s sum should be equal to its corresponding class point size indicated in Table 5.2; The ‘PredictedLabels’ refers to the labels that model predicted are on the vertical axis on the left side of the matrix. The number within the matrix represents the count of instances for each combination of predicted and true labels. The diagonal from the top left to the bottom right represents the number of correct predictions for each class: 99143 correct predictions for ‘wall’; 4852 correct predictions for ‘pillar’; 29521 correct predictions for ‘vehicle’; and 352551 correct predictions for ‘ground’. The off-diagonal cells represent misclassifications. Take the ‘wall’ true labels in Figure 5.1 as examples (the first column from the left), the first row [wall, wall] refers to the model correctly predicting the ‘wall’ 99143 times; Moving downwards, [pillar, wall] means the model incorrectly predicts the wall into ‘pillar’ 111 times; in [vehicle, wall] and [ground, wall], the model incorrectly predict the wall into ‘vehicle’ 491 times and ‘ground’ 1398 times. For the ‘wall’ predicted labels (first low), [wall, pillar] refers to the model incorrectly predicting ‘pillar’ into ‘wall’ 1143 times. In

$[wall, vehicle]$ and $[wall, ground]$, the model incorrectly predicts 'vehicle' and 'ground' into 'wall' 804 and 1850 times.

In summary, the model exhibits proficient predictive capabilities, notably for the 'ground' class, while showing areas for improvement in distinguishing between classes with fewer correct predictions and higher misclassification rates, such as 'pillar' (which is mainly due to the fact of insufficient input points).

5.1.3 Visualization of Segmented Results

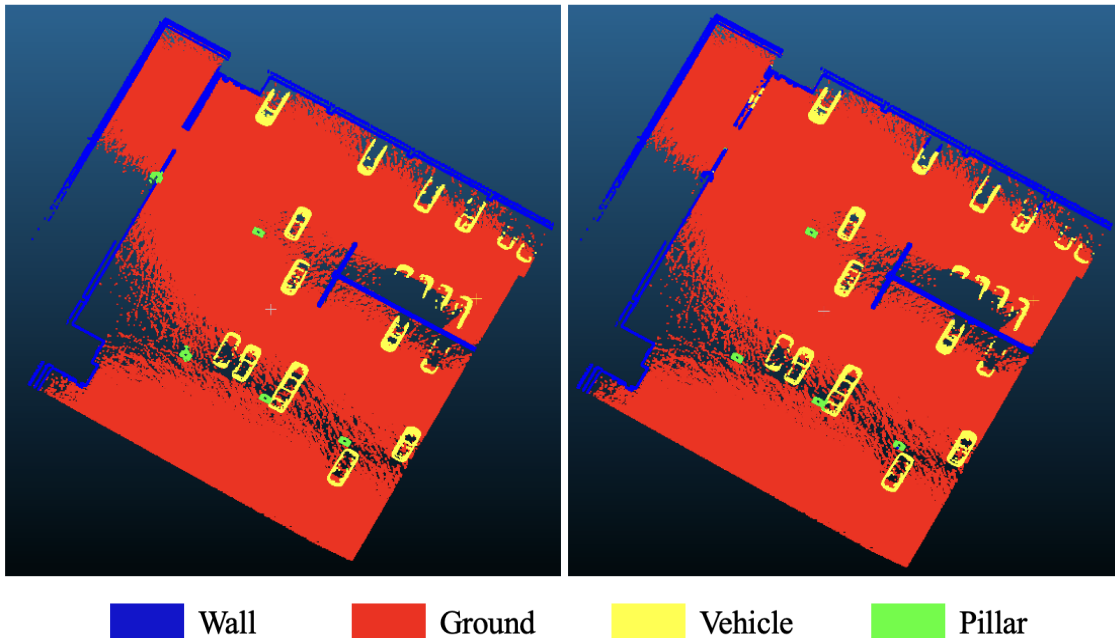


Figure 5.2: Ground truth (left) vs. segmented results (right) on test dataset

Based on the visual output presented in Figure 5.2, the predictive model generally aligns well with the ground truth, apart from misclassification where 'pillar' has been incorrectly segmented as 'wall' in the top left of the dataset. This specific error is consistent with the quantitative data provided in the confusion matrix in Figure 5.1, which suggests that around twenty percent of instances labelled as 'pillar' have been predicted as 'wall'. For a more granular examination of the model output, detailed visual comparisons between the **Ground Truth (GT)** and the model's predictions are illustrated in Tables 5.3, and 5.4 respectively.

Table 5.3: GT vs. good prediction example

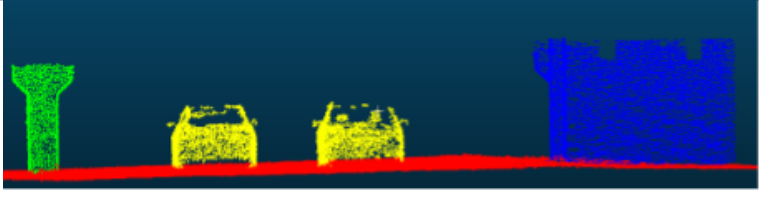
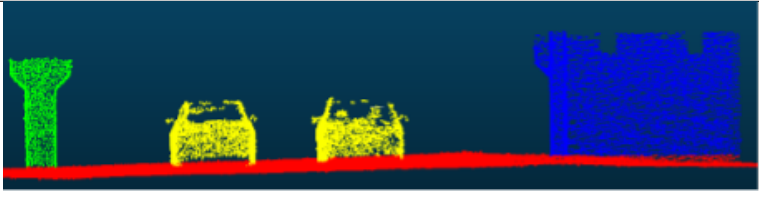

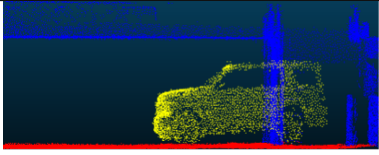
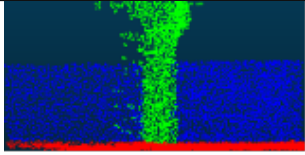
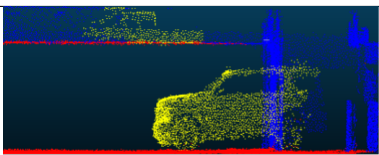
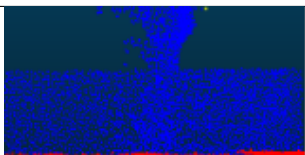

GT (Raw Input)	
Prediction	
	

Table 5.4: GT vs. Unsatisfactory predictions example

GT (Raw Input)		
Prediction		
Result	[vehicle, wall]	[wall, pillar]
		

5.2 Surface Sampling and Reconstruction Results

5.2.1 HDBSCAN Clustering

By running the script proposed in Section 4.3.2, clusters with designated '*min_cluster_size*' and '*min_sample*' are generated shown in Figure 5.3 below. The process results in a total of 17 distinct clusters. Notably, only the '*xyz*' coordinates from the original dataset are retained for projection, altering the data shape from $[N,12]$ in the original input point clouds to $[N,3]$ in the output point cloud clusters (refer to appendix D1 for HDBSCAN clustering console).

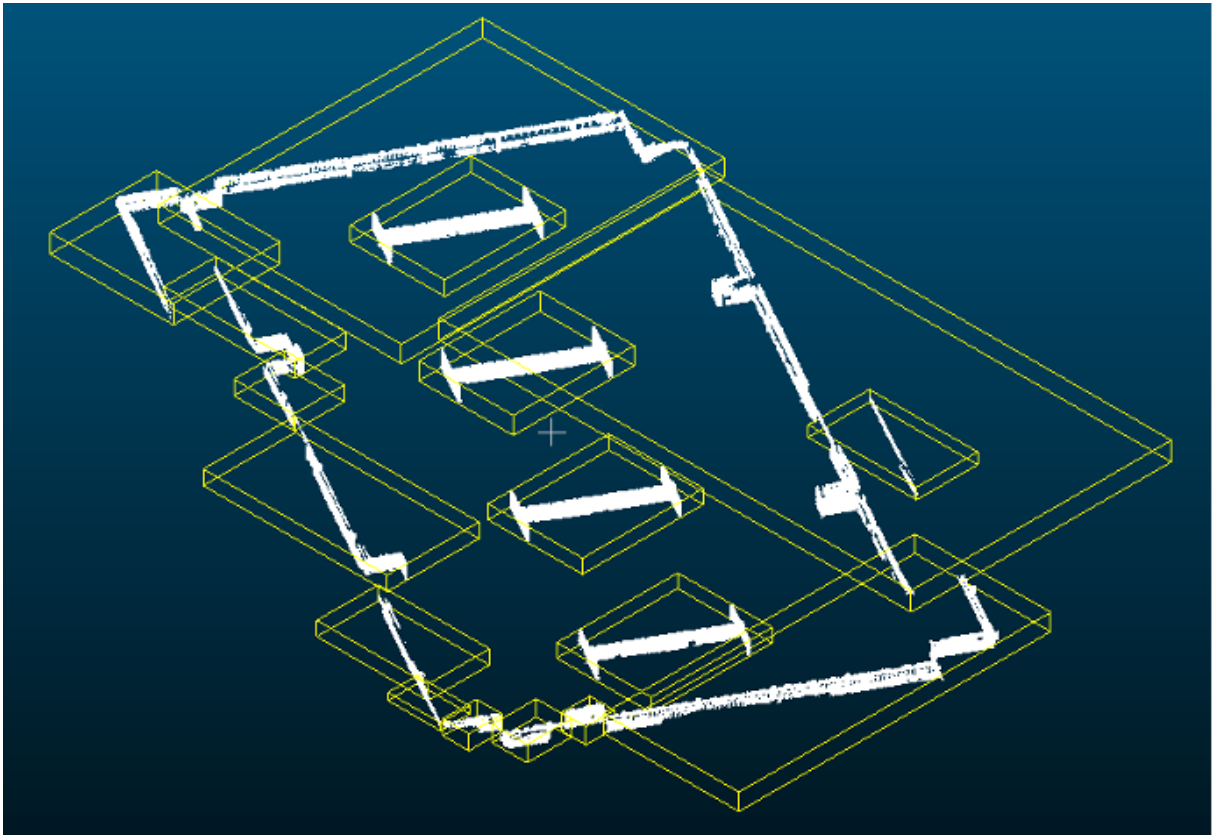
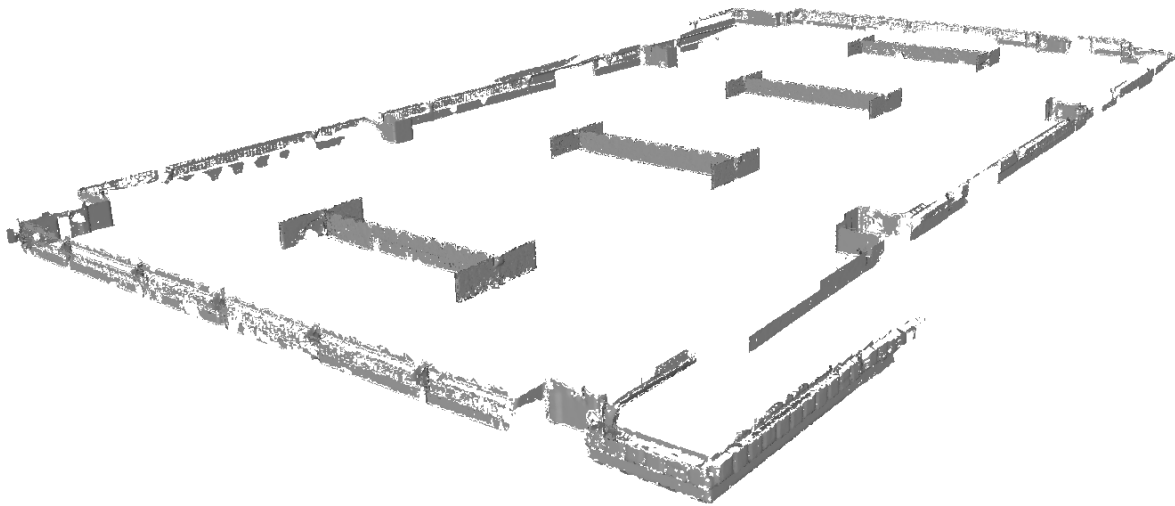


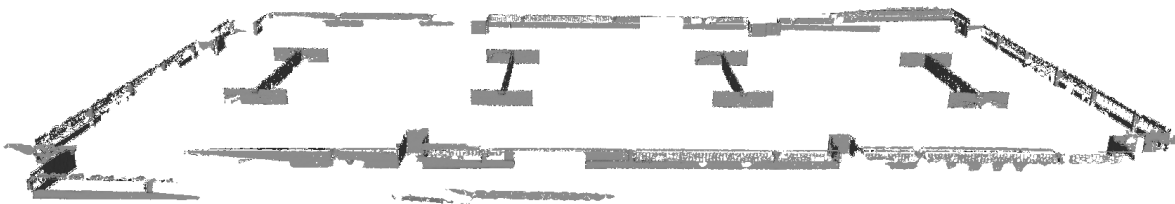
Figure 5.3: Cluster output with HDBSCAN clustering approach

5.2.2 BPA Surface Reconstruction

Adopting the Open3D [BPA](#) as demonstrated in Section [4.3.3](#), a mesh surface has been generated in Figure [5.4](#) below. Since the applicability of [BPA](#) is heavily dependent on the parameter tuning (ball radius), an inappropriate radius could lead to poor mesh quality and incomplete surface reconstruction.



(i)







(ii)

Figure 5.4: BPA mesh visualization

According to the result, it appears that the [BPA](#) may not be ideal for addressing holes and gaps in data. This is evidenced by its tendency to produce unrealistic shapes and leave unfilled holes in the mesh output.

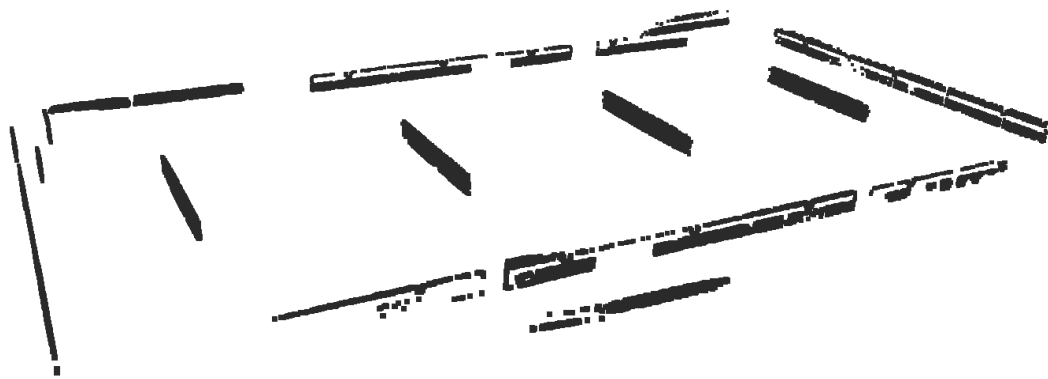
Furthermore, while the [BPA](#) demonstrates efficiency in computing simple shapes, such as flat walls and edges (refer to '*FairOutput*' in [Table 5.3](#)), its performance diminishes when dealing with complex geometries. This is particularly noticeable in sharp features with intricate details, as exemplified in the bottom right output of [Table 5.5](#). In such cases, the [BPA](#) is prone to either missing these features entirely or excessively smoothing them, thereby compromising the accuracy of the representation.

Table 5.5: BPA Output Comparison

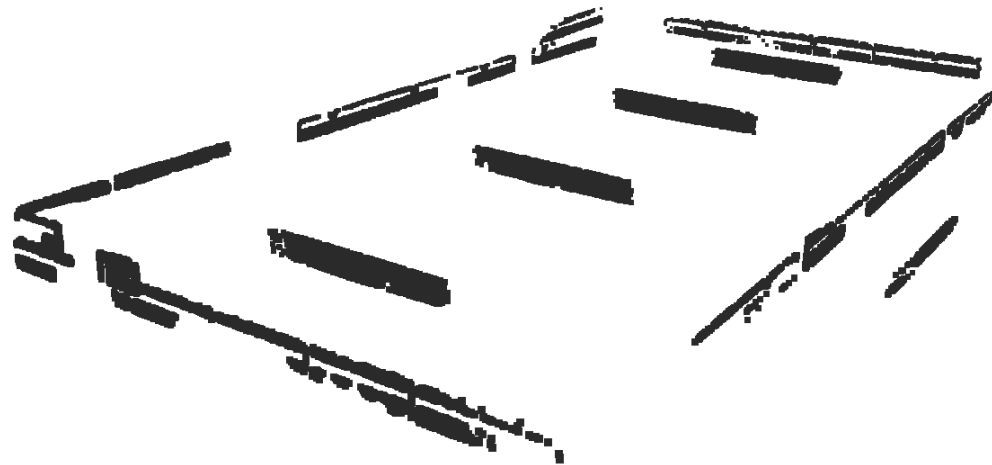
Fair	Bad
	
	

5.2.3 pyRANSAC-3D (multi-plane fitting)

By applying pyRANSAC-3D in the for loop demonstrated in Section 4.3.4, each cluster generated from the HDBSCAN clustering approach would be assessed and extracted for plane fitting correspondingly, and all the fitted planes would be merged and generated a complete output with multiple fitted planes illustrated in Figure 5.5 below.



(i)



(ii)

Figure 5.5: pyRANSAC-3D multi-plane fitting visualization

5.2.4 Random Sample Consensus with multi-plane fitting

Since the result of pyRANSAC-3D is not ideal, a refined RANSAC algorithm has been adopted in this section. This advancement involves iterating the RANSAC plane-fitting functions (refer to Figure 5.6) across various clusters as .txt files within a specific directory. By aggregating results from these clusters, the method evolves into a comprehensive multi-plane fitting RANSAC approach. This innovation not only preserves the algorithm’s inherent robustness but also extends its applicability to more complex and segmented 3D datasets, resulting in a more holistic and accurate plane-fitting output.

In Figure 5.6, the *'ransac_plane_fit'* function is detailed with its key parameters, each serving a specific role in the plane fitting process using the RANSAC algorithm. The parameter *'points'* denotes the set of 3D points that serve as input for the plane fitting procedure. The *'ransac_plane_fit'* function iteratively seeks the best plane fit for a set of 3D points. Each iteration randomly selects *'ransac_n'* points to estimate a plane using the *'estimate_parameter'* function from the *'plane_model'* class (see Table Appendix E1). It calculates the distance of all points to this plane, identifying inliers within the *'max_dst'* threshold. The function updates the best model parameters and inliers if the current iteration’s inlier ratio exceeds the previous ones. The algorithm terminates when the inlier ratio surpasses *'stop_inliers_ratio'* or after reaching the maximum number of iterations, *'max_trials'*. The final output will return the best plane parameters, the inlier points for the plane, and the remaining points (outliers).

Lastly, in the *'__main__'* function (see ‘Appendix F1 for more detail), the script iterates through all *'txt'* files in a specified directory *'path'*, where each file is expected to contain the 3D point clouds clustered using HDBSCAN clustering, return multiple fitted surfaces in the scene as one single output, illustrated in Figures 5.7, 5.8, and 5.9 respectively.

```

Def ransac_plane_fit(points, ransac_n, max_dst,
max_trials=1000, stop_inliers_ratio=1.0):

    # RANSAC plane fitting
    pts = points.copy()
    num = pts.shape[0]
    iter_max = max_trials
    best_inliers_ratio = 0
    best_plane_params = None
    best_inliers = None
    best_remains = None

    for i in range(iter_max):
        sample_index = random.sample(range(num), ransac_n)
        sample_points = pts[sample_index, :]
        plane = plane_model()
        plane_params = plane.estimate_parameters(sample_points)

        index = plane.calc_inliers(points, max_dst)
        inliers_ratio = pts[index].shape[0] / num

        if inliers_ratio > best_inliers_ratio:
            best_inliers_ratio = inliers_ratio
            best_plane_params = plane_params
            bset_inliers = pts[index]
            bset_remains = pts[index == False]

        if best_inliers_ratio > stop_inliers_ratio:

            print("iter: %d\n" % i)
            print("best_inliers_ratio: %f\n" % best_inliers_ratio)
            break

    return best_plane_params, bset_inliers, bset_remains

```

Figure 5.6: Algorithm for RANSAC Plane Fit

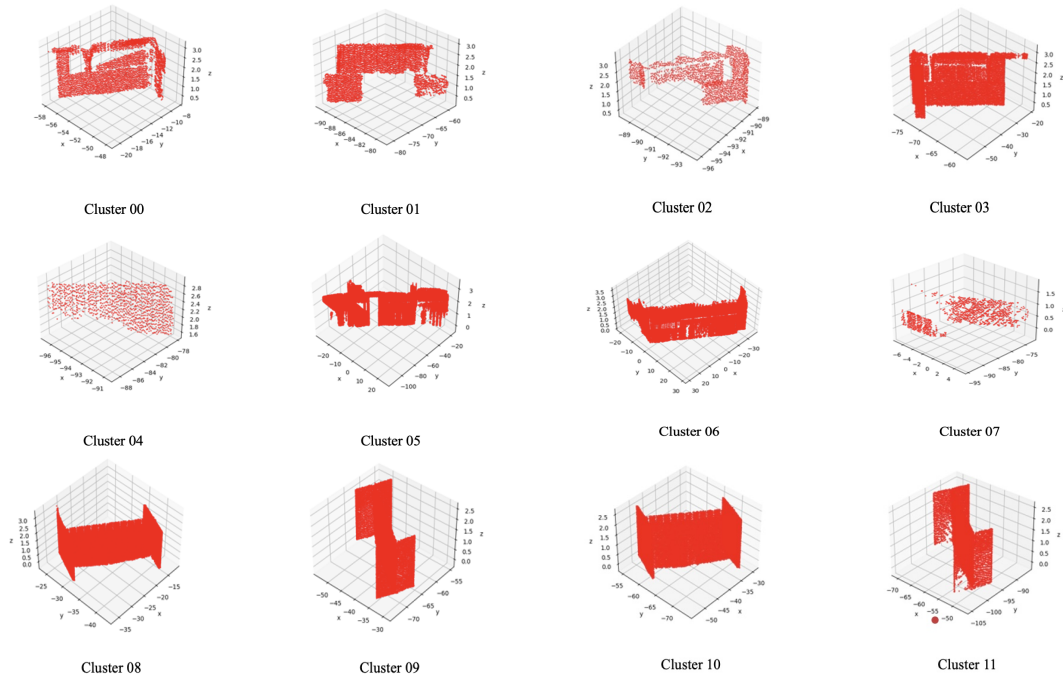


Figure 5.7: Part A: Revised RANSAC multi-plane fitting clusters 00 - 11

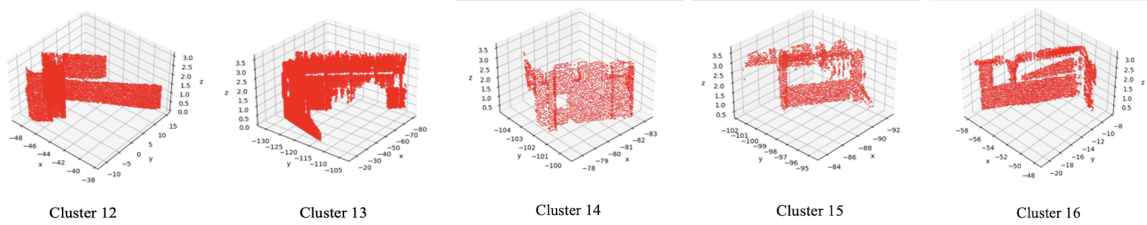
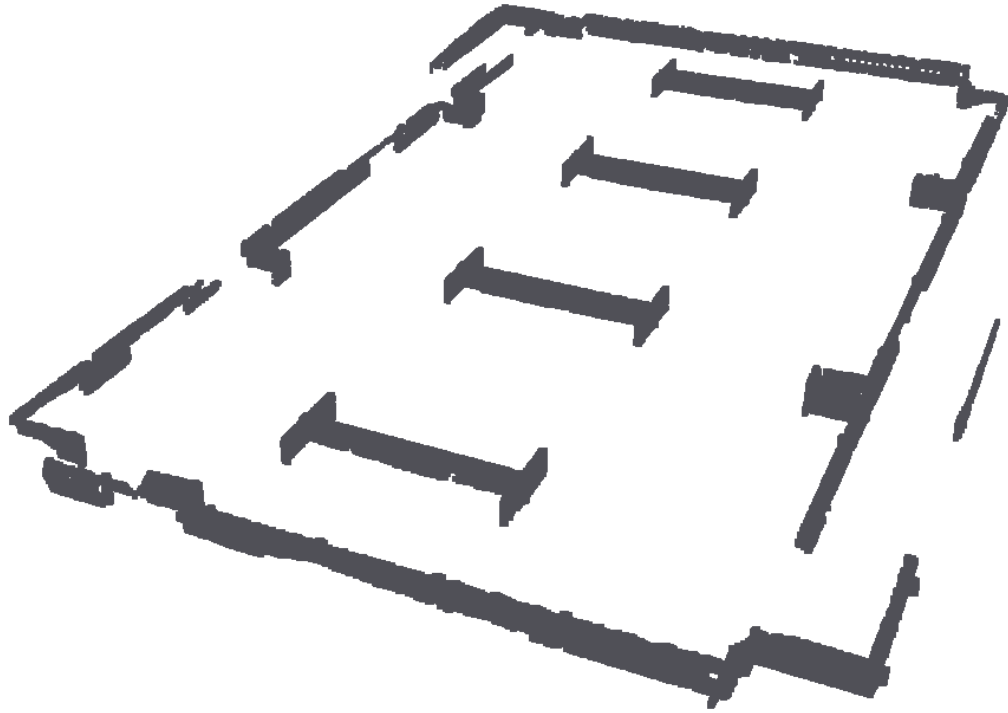
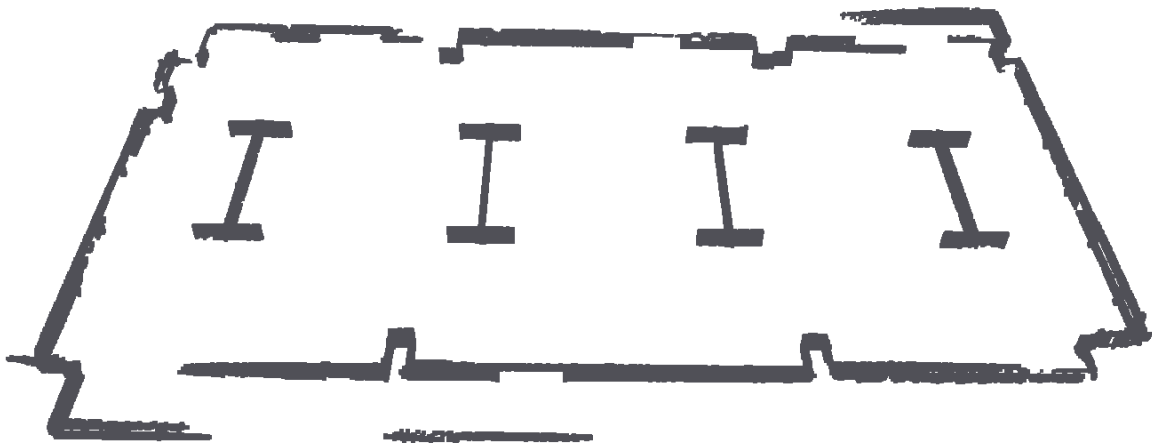


Figure 5.8: Part B: Revised RANSAC multi-plane fitting clusters 12 - 16



(i)



(ii)

Figure 5.9: Merged RANSAC multi-plane fitting visualization

5.2.5 Surface Output Comparison

The comparative analysis of three surface visualization outputs reveals that the revised [RANSAC](#) multi-plane fitting method outperforms the others in generating the most accurate output in this study. This is particularly evident in the characteristics and quality of the original point cloud, which is characterized by numerous unfilled holes and gaps. These deficiencies lead to a less effective surface reconstruction through mesh generation.

The output derived from the standard pyRANSAC-3D tool demonstrates certain limitations, notably in its failure to detect and fit plenty of surfaces accurately. This shortfall can result in a significant loss in information and data fidelity. In contrast, the output from the enhanced [RANSAC](#) multi-plane fitting approach performs well in retaining intricate details. It detects and fits most of the general structure of all vertical planes present in the original point clouds, thereby ensuring a more detailed and accurate representation of the data.

5.3 Line Reconstruction

5.3.1 3D Line Detection

The 'LineDetection3D' algorithm was employed to identify line segments within the dataset. Initially, a global parameter of $k=20$ was set, as detailed in [Appendix B1](#). This parameter choice facilitated the detection of line segments. The outcomes of this process are shown in [Figure 5.10](#), which illustrates the line segments as identified by the algorithm.

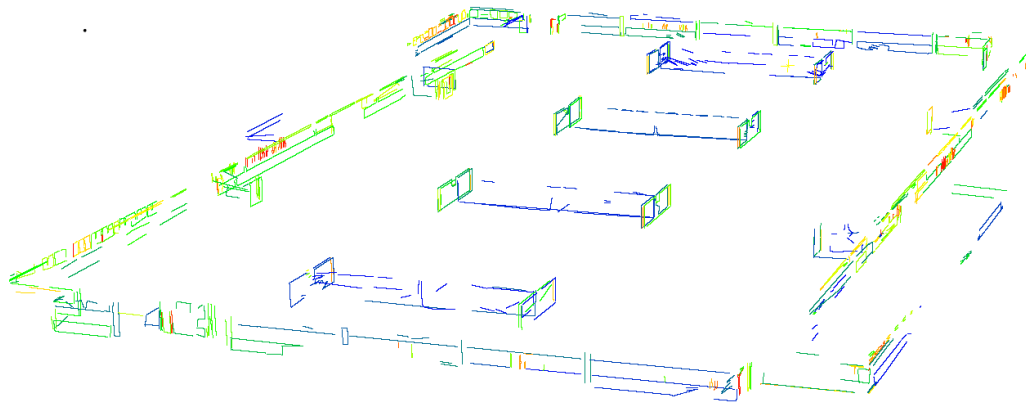


Figure 5.10: Line segment output using LineDetection3D ($k = 20$)

5.3.2 Output Optimization

In the initial implementation, the k parameter value in the `'detector.run()'` function, as outlined in Appendix B1, was preset to be 20. However, a single global k value proved inadequate due to the uneven distribution of points in the dataset, leading to unsatisfactory line segment output in terms of fidelity and precision. To enhance the quality of the output, a more tailored approach was adopted. The dataset was segmented into smaller clusters based on point features, considering variations in point density and size. This method facilitated the tailoring of different k values to clusters, thereby enhancing the efficacy of the line detection process. For instance, larger clusters, which typically represent expansive surfaces such as flat walls with long extended edges, were allocated a higher k value of 150. Conversely, smaller feature clusters, exemplified by short edge walls, were assigned a lower k value of 10. This adaptive strategy in the k value assignment supports the precision and dependability of true line detection across disparate cluster sizes.

Following the segmentation and line detection steps, a subsequent phase involved the meticulous removal of outliers and noise. This refinement was pivotal in producing a cleaner and more visually appealing output, which is illustrated in Figures 5.11 and 5.12, indicating the enhanced clarity and improved aesthetic quality of the final dataset.

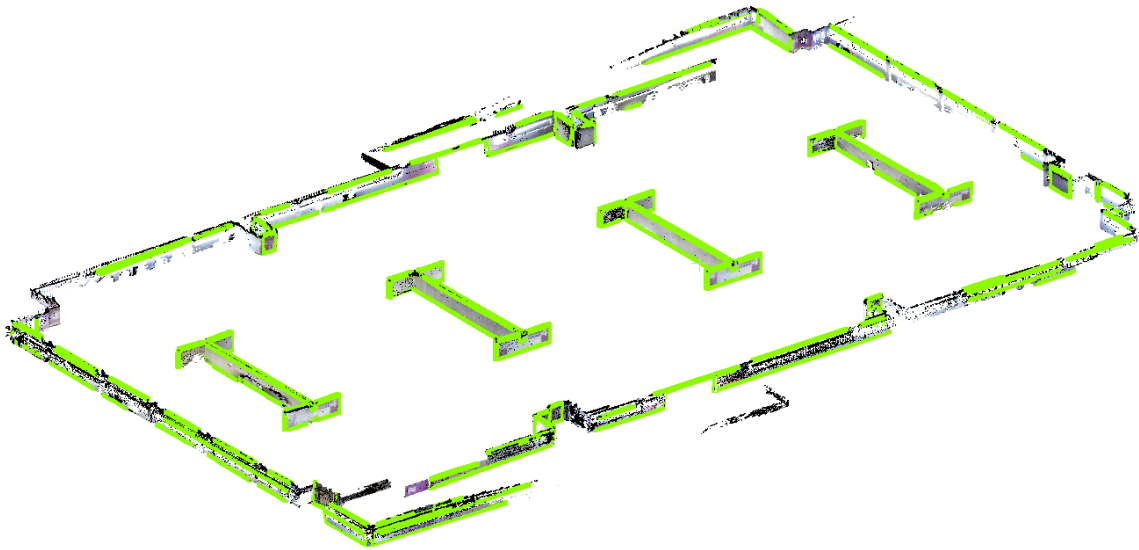
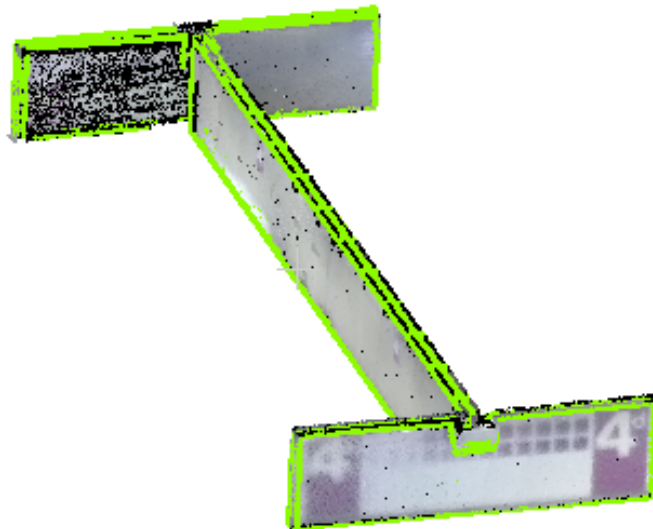
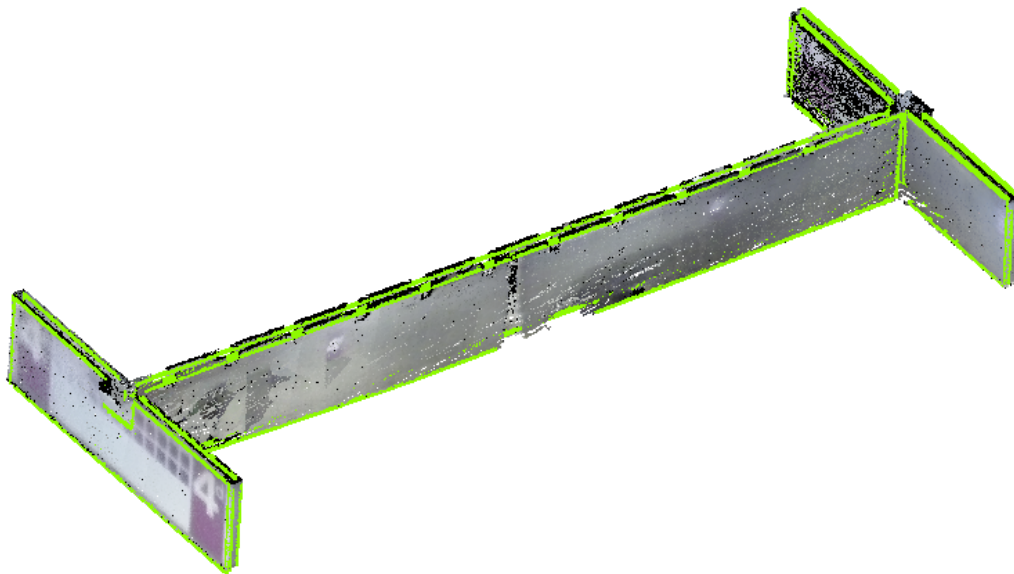


Figure 5.11: Optimized line segment reconstruction output visualization



(i)

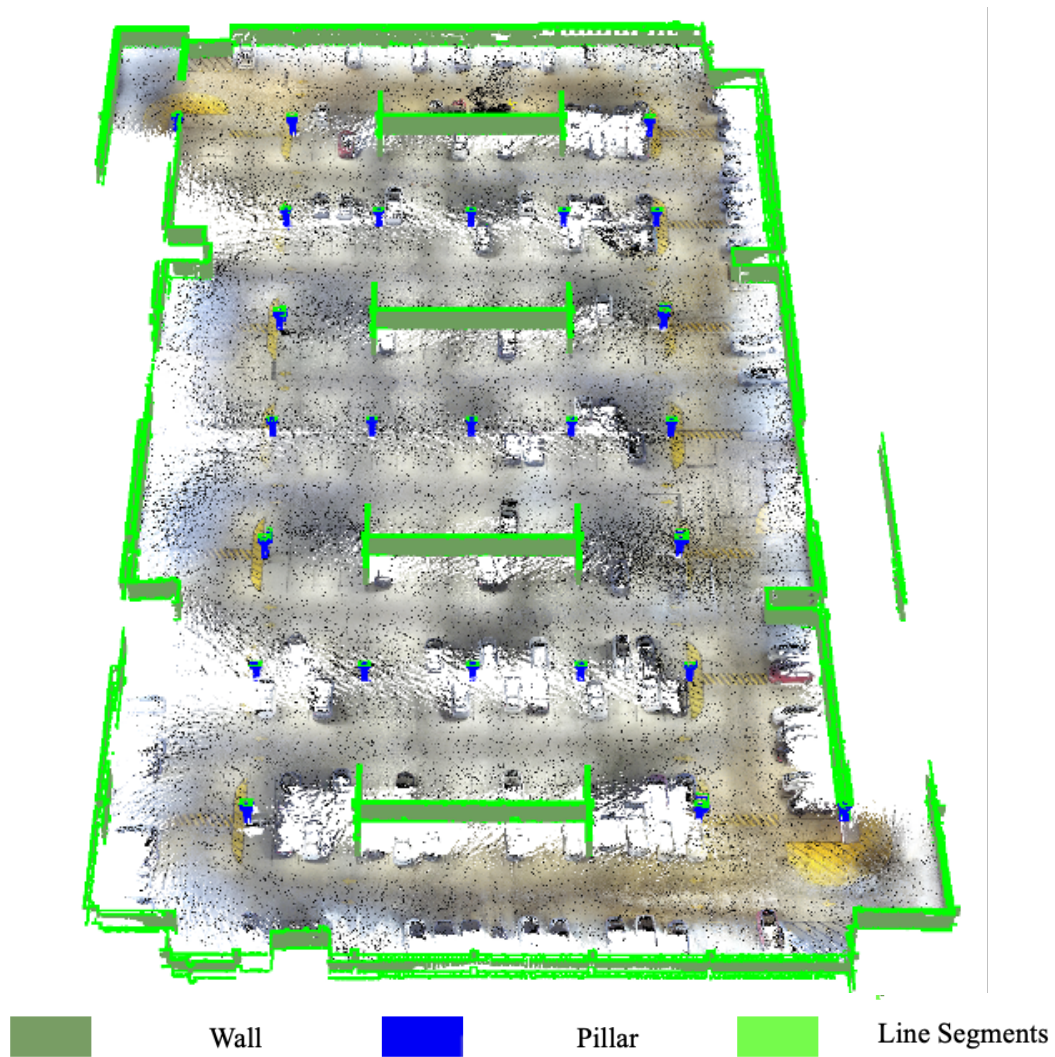


(ii)

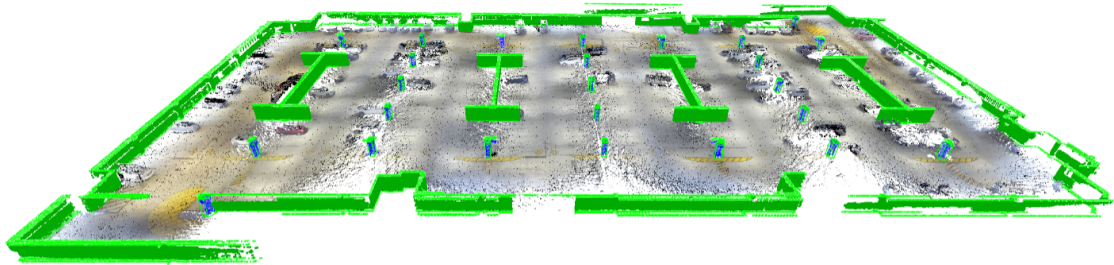
Figure 5.12: Optimized line segment reconstruction output visualization detail views


5.3.3 Reconstruction Visualization Output

Integrating the results from the preceding phases, a detailed reconstruction of the indoor parking structure is presented in Figures 5.13 and 5.14. In these visualizations, edges and line segments are highlighted in lime green, pillars are depicted in navy blue, and walls, representing vertical planes, are rendered in olive green. This colour-coded scheme facilitates a visual distinction among the various structural elements within the model.

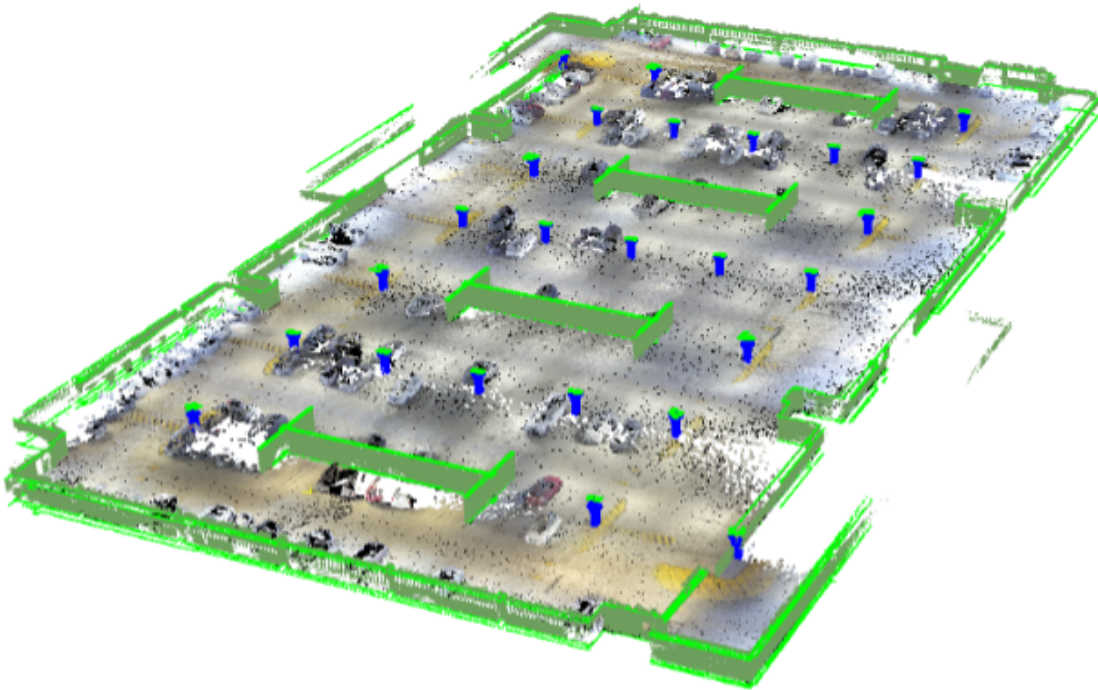


(a)



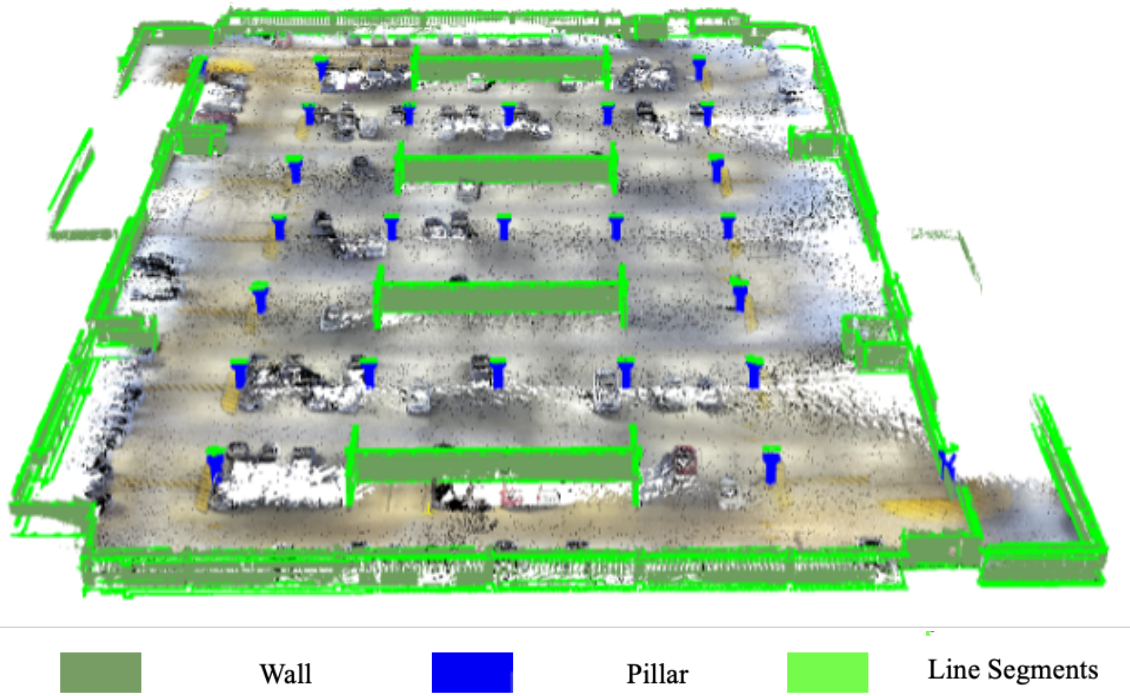
 Wall  Pillar  Line Segments

(b)

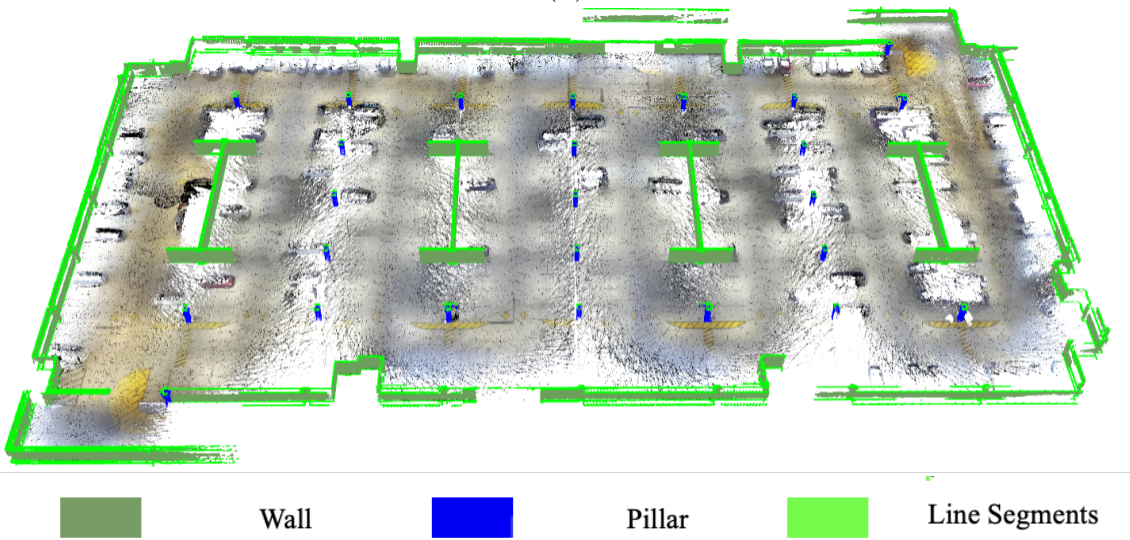


 Wall  Pillar  Line Segments

(c)

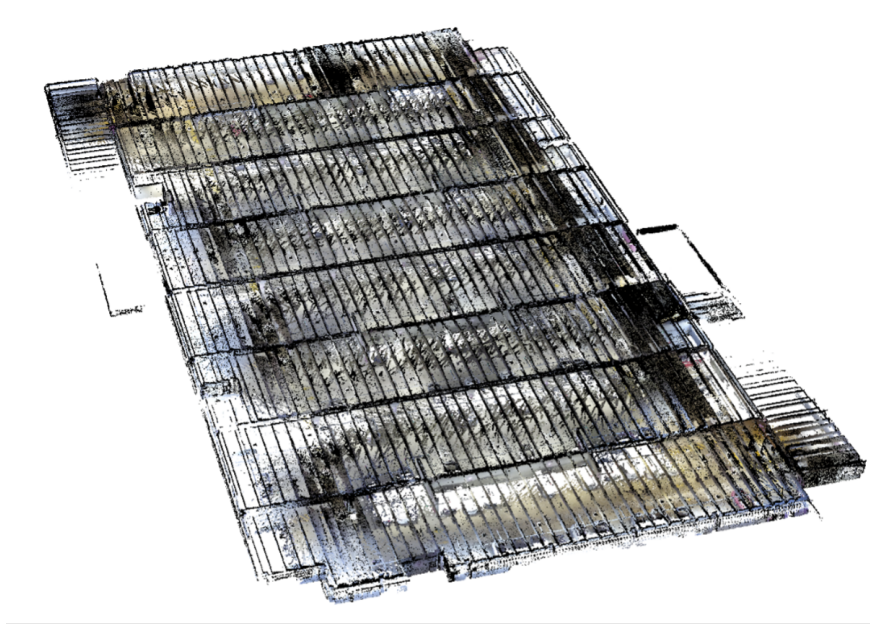


(d)

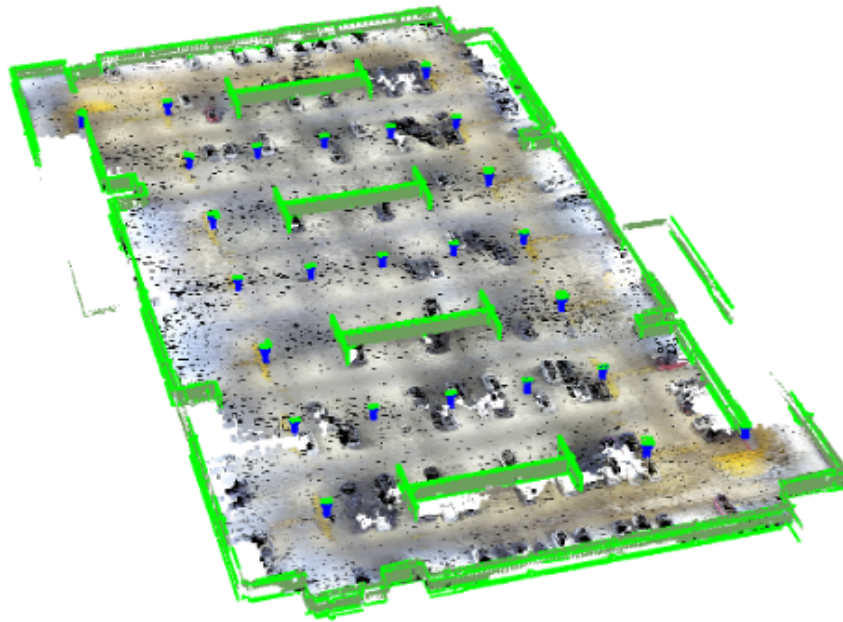


(e)

Figure 5.13: Optimized line segment reconstruction output visualization detail views



(a)



(b)

Figure 5.14: Raw point clouds vs. final 3D model

5.4 Limitations

5.4.1 Human/ Operator Error

Human or operator error during the *data collection* (see Figure 5.15) phase can result from improper operation or handling of equipment, which can introduce inaccuracies into the dataset. For instance, if an operator misinterprets the protocol or fails to follow standard operating procedures, the collected data could be compromised. Such errors might include incorrect calibration of instruments, leading to skewed measurements; inconsistent walking pace during collection or even incomplete collecting loop, resulting in data contamination or information loss.



Figure 5.15: GeoSLAM laser scanner operation during data collection

5.4.2 Insufficient/missing Data

The dataset might be insufficient in size if it does not include enough data points to capture the underlying patterns or relationships. In such cases, the model may overfit, learning the noise and specificities of the small dataset rather than the general trends. This overfitting results in

poor performance when the model encounters new, unseen data. This issue is evident in the line detection and reconstruction phase, where missing data in the vertical plane adversely impacts subsequent stages. As illustrated in Figure 5.16, the insufficient input points on the vertical surface correlate with the misidentified edges, highlighted in lime green.

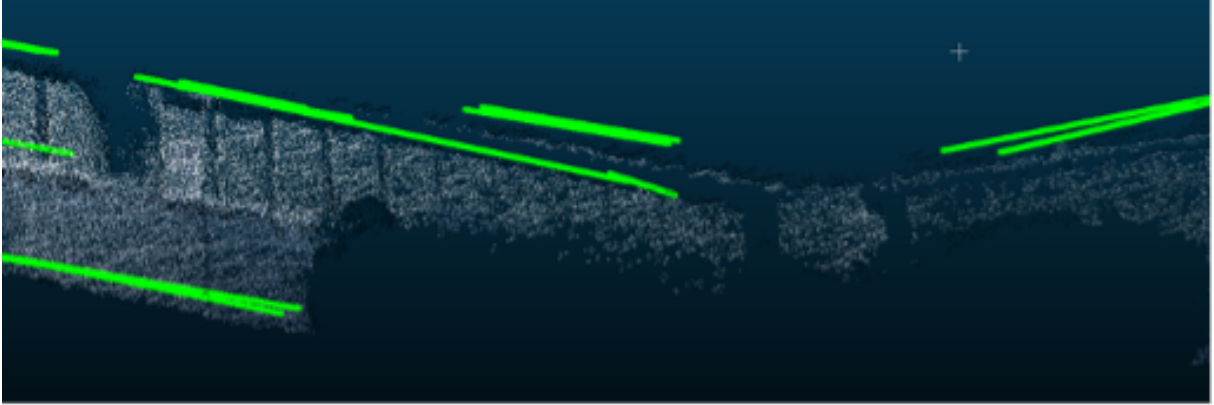


Figure 5.16: Insufficient points and miss detected line segments

In addition, during the method design stage for line segment optimization, a **cGAN** model was initially employed, as detailed in Appendices G1 to G4, to enhance edge reconstruction. This process involved converting 3D data into multiple 2D images for input into the discriminator and creating the self-defined training and testing sets for the generator. However, only 6400 training and 3200 testing sets were generated, which proved insufficient not only in quantity but also in structural complexity. The training/testing sets considered only two variables: disconnectivity and non-orthogonality. This limitation resulted in significant generator (G) and discriminator (D) losses, indicating that the **cGAN** model was not optimally utilized for refining the line framework. Additionally, the subsequent task of interpolating these 2D images back into 3D point clouds presented its own set of complexities and challenges.

5.4.3 Equipment/ Algorithm Limitation

Laser scanners operate by emitting a laser beam towards a target and recording the time it takes for the light to reflect to the sensor. This method is highly effective for capturing the geometry of visible surfaces. However, if a part of the target area is obscured by another object, the laser beam cannot penetrate through or bend around the obstructing object to reach hidden surfaces. Consequently, the scanner only records the information on surfaces that are directly visible to it, missing any details that are behind or obscured by other objects. Referring to Figure 5.17, the presence of vehicles in the scanning environment serves as an obstruction, blocking the target

area which includes the wall and ground. This obstruction results in the laser scanner's inability to detect and record the target area, leading to information loss.

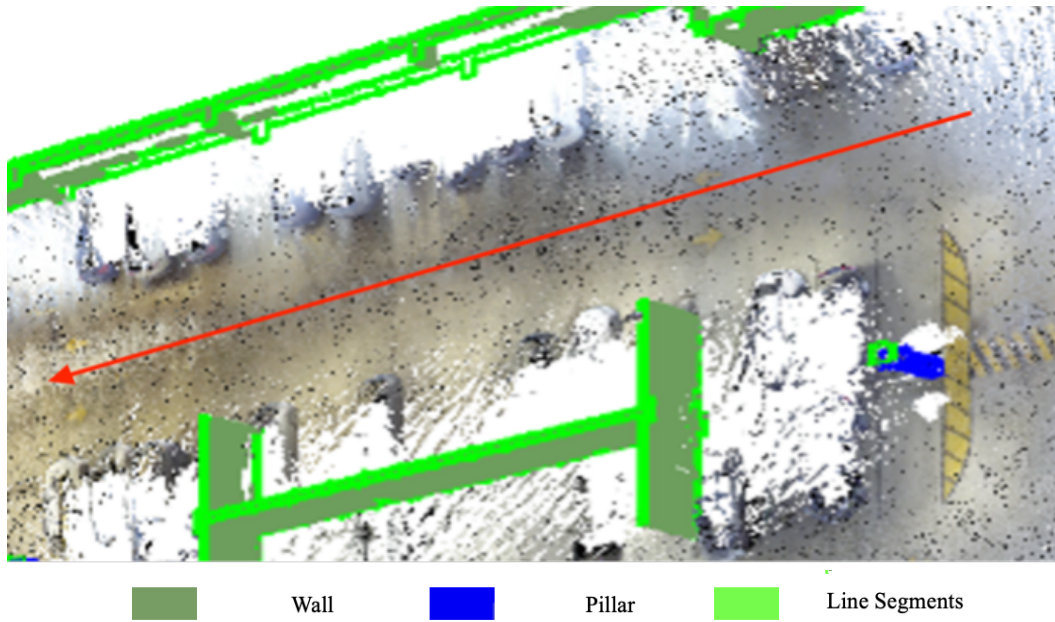


Figure 5.17: Data Occlusion

Besides, enclosed loops in [SLAM](#) algorithms play a pivotal role in ensuring the accuracy, consistency, and efficiency of the map creation and localization process, which are essential for the reliable operation of autonomous robots and vehicles in dynamic environments. By completing a closed scan, people can comprehensively analyze related errors that occur in data resolution and adjust parameters to resolve these errors. Although the big loop in our data collection was enclosed, partial loops were not enclosed (refer to [Figure 3.6](#)), this may cause a series of misclosure. To minimize the error caused by [IMU](#) drift, optimized routes are proposed in [Figures 5.18](#) and [5.19](#) as alternatives to walk the trajectory in enclosed loops. The pink start refers to the start/finish spot, the green arrows indicate the moving direction, and the purple/ blue line represents the refined trajectory.

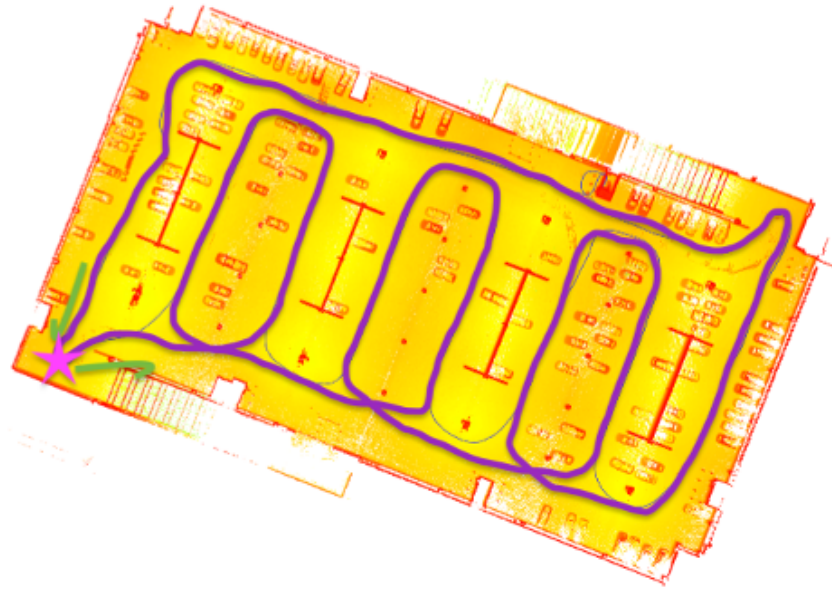


Figure 5.18: Optimized enclosed loop trajectory 01 (vehicle and pillar focused)

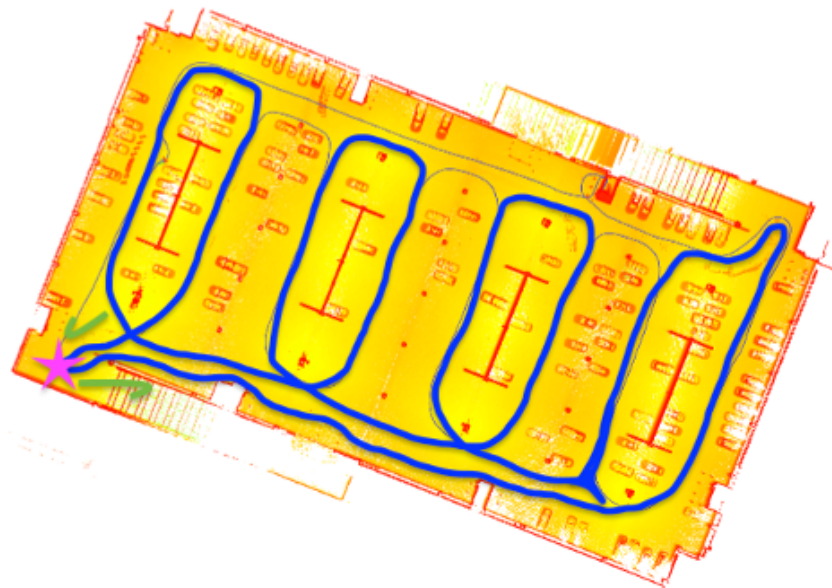


Figure 5.19: Optimized enclosed loop trajectory 02 (wall structure focused)

Chapter 6

Conclusions and Recommendations for Future Studies

6.1 Conclusions

In this thesis, we have applied a groundbreaking approach to indoor spatial modelling, with a particular focus on the development of precise digital twin models for indoor parking garages. The methodology is founded on a [LiDAR-based SLAM](#) system, serving as the backbone for the overall approach, which enables us to capture detailed spatial data for indoor scenes. In addition, the data model has been enhanced with advanced algorithms dedicated to semantic segmentation, surface reconstruction, and line feature extraction. These algorithms process the raw data from the [LiDAR-based SLAM](#) systems, transforming it into a meaningful and highly accurate digital representation of the physical environment.

This work paves the way for applying digital twin technology in complex indoor settings, highlighting the crucial role of point cloud data in creating precise 3D representations of physical objects or environments. Overall, this thesis has effectively showcased the generation of highly precise digital models for indoor parking structures.

6.2 Recommendations for Future Studies

6.2.1 Point Cloud Completion

This approach refines the 3D map, resulting in a more visually coherent and complete output. As illustrated in [Figure 6.1](#), significant portions of the area in the existing dataset are missing.

This incompleteness is primarily attributed to two factors: an inconsistent walking pace and occlusions. These issues cause the collected data to be incomplete, failing to accurately represent the entire scene. By employing point cloud completion in the pre-processing stage, we can address these gaps and ensure a more thorough representation of the indoor environment.

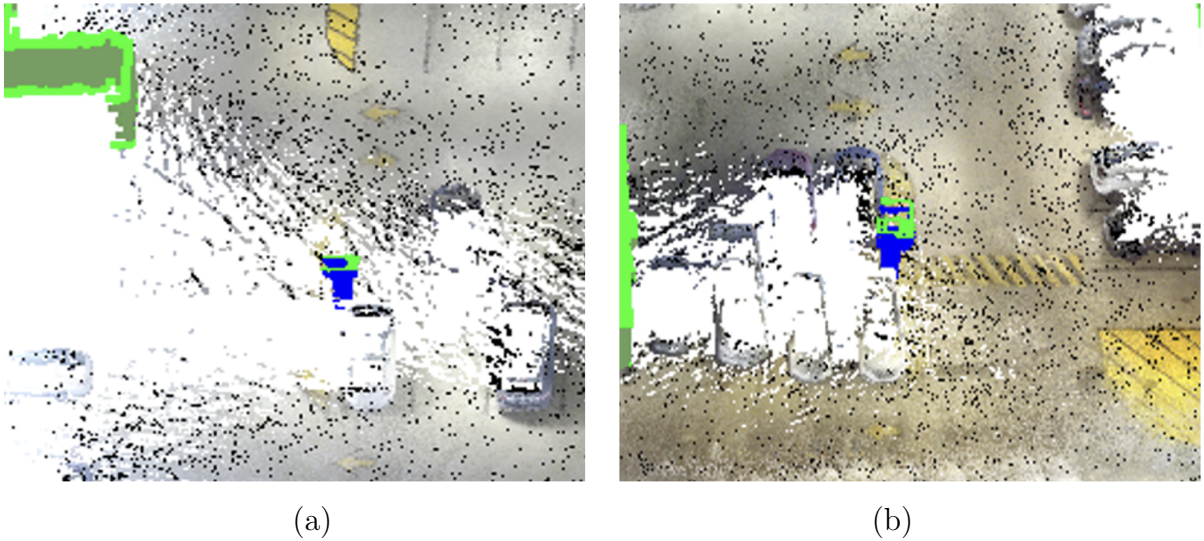


Figure 6.1: Data loss during collection: (a) inconsistent walking speed;(b) occlusions

6.2.2 Object Detection

To ascertain the accuracy and practicality of the model, it can be integrated into a prototype for an autonomous last 1-km parking system, followed by a thorough evaluation of its performance. A key aspect of this evaluation is the successful detection of safety bollards, which were previously omitted in Section 4.1.3. Ensuring that the system can identify and avoid these bollards is crucial to avoid any potential interference with them. Similarly, the detection of other vehicles is required for the system's functionality. To enhance the model's capability to recognize such obstacles, the implementation of an object detection algorithm as a subsequent step is recommended.

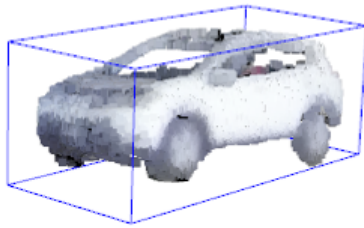


Figure 6.2: Vehicle in bounding box

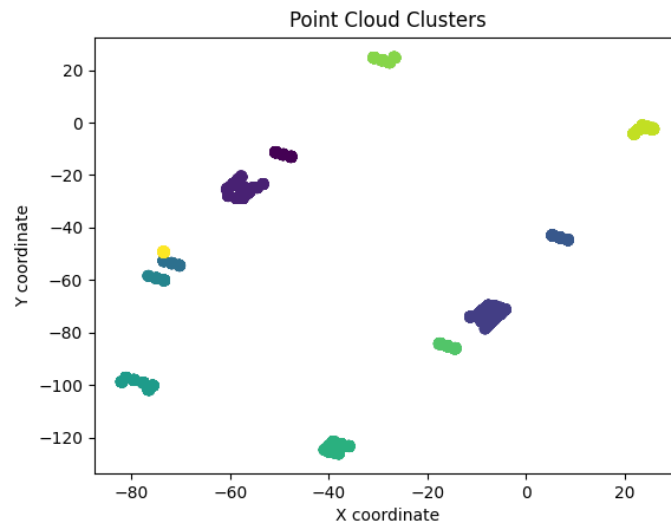


Figure 6.3: Safety bollards clusters by HDBSCAN

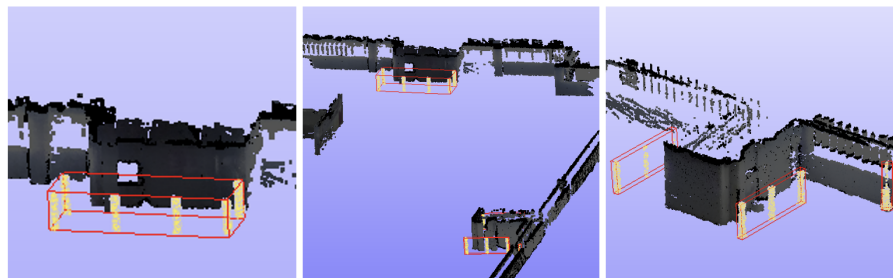


Figure 6.4: Safety bollards in bounding box

References

- Babacan, K., Jung, J., Wichmann, A., Jahromi, B., Shahbazi, M., Sohn, G., and Kada, M. (2016). Towards object driven floor plan extraction from laser point cloud. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 41:3–10.
- Balado Frías, J., González Rodríguez, M. E., Verbree, E., Díaz Vilariño, L., Lorenzo Cimadevila, H. R., et al. (2020). Automatic detection and characterization of ground occlusions in urban point clouds from mobile laser scanning data. *ISPRS Annals of Photogrammetry Remote Sensing and Spatial Information Sciences*.
- Berger, M., Tagliasacchi, A., Seversky, L. M., Alliez, P., Levine, J. A., Sharf, A., and Silva, C. T. (2014). State of the art in surface reconstruction from point clouds. In *35th Annual Conference of the European Association for Computer Graphics, Eurographics 2014-State of the Art Reports*, number CONF. The Eurographics Association.
- Bernardini, F., Mittleman, J., Rushmeier, H., Silva, C., and Taubin, G. (1999). The ball-pivoting algorithm for surface reconstruction. *IEEE transactions on visualization and computer graphics*, 5(4):349–359.
- Boulch, A., Guerry, J., Le Saux, B., and Audebert, N. (2018). Snapnet: 3d point cloud semantic labeling with 2d deep segmentation networks. *Computers & Graphics*, 71:189–198.
- Broere, W. (2016). Urban underground space: Solving the problems of today’s cities. *Tunnelling and Underground Space Technology*, 55:245–248.
- Cui, Y., Li, Q., Yang, B., Xiao, W., Chen, C., and Dong, Z. (2019). Automatic 3-d reconstruction of indoor environment with mobile laser scanning point clouds. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 12(8):3117–3130.
- Döllner, J. (2020). Geospatial artificial intelligence: potentials of machine learning for 3d point clouds and geospatial digital twins. *PFG–Journal of Photogrammetry, Remote Sensing and Geoinformation Science*, 88:15–24.

- Elghazaly, G., Frank, R., Harvey, S., and Safko, S. (2023). High-definition maps: Comprehensive survey, challenges and future perspectives. *IEEE Open Journal of Intelligent Transportation Systems*.
- Fang, H., Lafarge, F., Pan, C., and Huang, H. (2021). Floorplan generation from 3d point clouds: A space partitioning approach. *Isprs Journal of Photogrammetry and Remote Sensing*, 175:44–55.
- Fathi, H., Dai, F., and Lourakis, M. (2015). Automated as-built 3d reconstruction of civil infrastructure using computer vision: Achievements, opportunities, and challenges. *Advanced Engineering Informatics*, 29(2):149–161.
- Fei, B., Yang, W., Chen, W.-M., Li, Z., Li, Y., Ma, T., Hu, X., and Ma, L. (2022). Comprehensive review of deep learning-based 3d point cloud completion processing and analysis. *IEEE Transactions on Intelligent Transportation Systems*.
- Fischler, M. A. and Bolles, R. C. (1981). Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395.
- GeoSLAM (Accessed 2023). Homepage. Retrieved from <https://geoslam.com/>.
- Guo, Y., Wang, H., Hu, Q., Liu, H., Liu, L., and Bennamoun, M. (2020). Deep learning for 3d point clouds: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 43(12):4338–4364.
- Hu, Z., Chen, C., Yang, B., Wang, Z., Ma, R., Wu, W., and Sun, W. (2022). Geometric feature enhanced line segment extraction from large-scale point clouds with hierarchical topological optimization. *International Journal of Applied Earth Observation and Geoinformation*, 112:102858.
- Ioannidou, A., Chatzilari, E., Nikolopoulos, S., and Kompatsiaris, I. (2017). Deep learning advances in computer vision with 3d data: A survey. *ACM computing surveys (CSUR)*, 50(2):1–38.
- Jain, A., Kurz, C., Thormählen, T., and Seidel, H.-P. (2010). Exploiting global connectivity constraints for reconstruction of 3d line segments from images. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 1586–1593. IEEE.
- Jiang, J., Bao, D., Chen, Z., Zhao, X., and Gao, Y. (2019). Mlvcnn: Multi-loop-view convolutional neural network for 3d shape retrieval. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, pages 8513–8520.

- Kazhdan, M., Bolitho, M., and Hoppe, H. (2006). Poisson surface reconstruction. In *Proceedings of the fourth Eurographics symposium on Geometry processing*, volume 7, page 0.
- Kodors, S. (2017). Point distribution as true quality of lidar point cloud. *Baltic Journal of Modern Computing*, 5(4).
- Laoudias, C., Moreira, A., Kim, S., Lee, S., Wirola, L., and Fischione, C. (2018). A survey of enabling technologies for network localization, tracking, and navigation. *IEEE Communications Surveys Tutorials*, 20(4):3607–3644.
- Lee, J. (2004). A spatial access-oriented implementation of a 3-d gis topological data model for urban entities. *GeoInformatica*, 8:237–264.
- Lehtola, V. V., Koeva, M., Elberink, S. O., Raposo, P., Virtanen, J.-P., Vahdatikhaki, F., and Borsci, S. (2022). Digital twin of a city: Review of technology serving city needs. *International Journal of Applied Earth Observation and Geoinformation*, page 102915.
- Lin, Y., Wang, C., Cheng, J., Chen, B., Jia, F., Chen, Z., and Li, J. (2015). Line segment extraction for large scale unorganized point clouds. *ISPRS Journal of Photogrammetry and Remote Sensing*, 102:172–183.
- Lu, D., Zhou, J., Gao, K. Y., Li, D., Du, J., Xu, L., and Li, J. (2023). Dynamic clustering transformer network for point cloud segmentation. *arXiv preprint arXiv:2306.08073*.
- Lu, X., Liu, Y., and Li, K. (2019). Fast 3d line segment detection from unorganized point cloud. *arXiv preprint arXiv:1901.02532*.
- Mariga, L. (2022). pyRANSAC-3D.
- Mounce, R. and Nelson, J. D. (2019). On the potential for one-way electric vehicle car-sharing in future mobility systems. *Transportation Research Part A: Policy and Practice*, 120:17–30.
- Orengo, H. (2015). Open source gis and geospatial software in archaeology: towards their integration into everyday archaeological practice.
- Pilouk, M. (1996). Integrated modelling for 3d gis. ITC.
- Qi, C. R., Su, H., Mo, K., and Guibas, L. J. (2017a). Pointnet: Deep learning on point sets for 3d classification and segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 652–660.
- Qi, C. R., Yi, L., Su, H., and Guibas, L. J. (2017b). Pointnet++: Deep hierarchical feature learning on point sets in a metric space. *Advances in neural information processing systems*, 30.

- Ronneberger, O., Fischer, P., and Brox, T. (2015). U-net: Convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2015: 18th International Conference, Munich, Germany, October 5-9, 2015, Proceedings, Part III 18*, pages 234–241. Springer.
- Sammartano, G. and Spanò, A. (2018). Point clouds by slam-based mobile mapping systems: accuracy and geometric content validation in multisensor survey and stand-alone acquisition. *Applied Geomatics*, 10:317–339.
- Sieber, R. (2006). Public participation geographic information systems: A literature review and framework. *Annals of the association of American Geographers*, 96(3):491–507.
- Von Gioi, R. G., Jakubowicz, J., Morel, J.-M., and Randall, G. (2008). Lsd: A fast line segment detector with a false detection control. *IEEE transactions on pattern analysis and machine intelligence*, 32(4):722–732.
- Wang, C., Hou, S., Wen, C., Gong, Z., Li, Q., Sun, X., and Li, J. (2018). Semantic line framework-based indoor building modeling using backpacked laser scanning point cloud. *ISPRS journal of photogrammetry and remote sensing*, 143:150–166.
- Wróblewski, A., Wodecki, J., Trybała, P., and Zimroz, R. (2022). A method for large underground structures geometry evaluation based on multivariate parameterization and multidimensional analysis of point cloud data. *Energies*, 15(17):6302.
- Xu, C., Yang, S., Galanti, T., Wu, B., Yue, X., Zhai, B., Zhan, W., Vajda, P., Keutzer, K., and Tomizuka, M. (2021). Image2point: 3d point-cloud understanding with 2d image pretrained models. *arXiv preprint arXiv:2106.04180*.
- Yurtsever, E., Lambert, J., Carballo, A., and Takeda, K. (2020). A survey of autonomous driving: Common practices and emerging technologies. *IEEE access*, 8:58443–58469.

APPENDICES

Appendix A

A1. Functions for Plane Extraction [4.3.1]

```
def create_mesh_from_plane(plane_model, width=1, height=1):
    a, b, c, d = plane_model
    # Create a mesh grid
    x = np.linspace(-width / 2, width / 2, 10)
    y = np.linspace(-height / 2, height / 2, 10)
    xx, yy = np.meshgrid(x, y)
    zz = (-d - a * xx - b * yy) / c
    # Create vertices and triangles for the mesh
    vertices = np.vstack((xx.flatten(), yy.flatten(), zz.flatten())).T
    triangles = []
    for i in range(9):
        for j in range(9):
            idx = i * 10 + j
            triangles.append([idx, idx + 1, idx + 10])
            triangles.append([idx + 1, idx + 11, idx + 10])
    mesh = o3d.geometry.TriangleMesh()
    mesh.vertices = o3d.utility.Vector3dVector(vertices)
    mesh.triangles = o3d.utility.Vector3iVector(triangles)
    mesh.compute_vertex_normals()
    return mesh

def display_inlier_outlier(cloud, ind, plane_model):
    inlier_cloud = cloud.select_by_index(ind)
```

```
outlier_cloud = cloud.select_by_index(ind, invert=True)
# Create a mesh plane to visualize the detected plane
mesh_plane = create_mesh_from_plane(plane_model, width=10, height=10)
mesh_plane.paint_uniform_color([0.1, 0.9, 0.1]) # Green plane

inlier_cloud.paint_uniform_color([1, 0, 0])
outlier_cloud.paint_uniform_color([0.8, 0.8, 0.8])
o3d.visualization.draw_geometries([inlier_cloud, outlier_cloud, mesh_plane])
```

A2. HDBSCAN cluster analysis [4.3.2]

```
def load_points(file_path):

    return np.loadtxt(file_path)

points = load_points("/Users/kristiehu/Desktop/thesis/segments.txt")

xyz_points = points[:, :3] # Extracting x, y, z coordinates
print(f"XYZ points shape: {xyz_points.shape}")

# Clustering with HDBSCAN
clusterer = hdbscan.HDBSCAN(min_cluster_size=900, min_samples=30)
labels = clusterer.fit_predict(xyz_points)
print("Clustering completed.")

unique_labels = set(labels)
print("Unique labels (clusters):", unique_labels)

n_clusters = len(unique_labels) - (1 if -1 in unique_labels else 0)
print(f"Estimated number of clusters: {n_clusters}")
...
if not os.path.exists(output_directory):
    os.makedirs(output_directory)
    print(f"Created directory: {output_directory}")
else:
    print(f"Directory already exists: {output_directory}")

# Count the number of points in each cluster and save them
for cluster_id in unique_labels:
    if cluster_id != -1: # Ignore noise points
        cluster_points = xyz_points[labels == cluster_id]
        number_of_points = len(cluster_points)
        print(f"Cluster {cluster_id} has {number_of_points} points.")

        file_path = os.path.join(output_directory, f"cluster_{cluster_id}.txt")
        np.savetxt(file_path, cluster_points)
        print(f"Saved file: {file_path}")
```

Appendix B

B1. 3D Line Detection using C++ [[4.4.1](#)& [5.3.1](#)]

```
#include <stdio.h>
#include <fstream>
#include "LineDetection3D.h"
#include "nanoflann.hpp"
#include "utils.h"
#include "Timer.h"
using namespace cv;
using namespace std;
using namespace nanoflann;

void readDataFromFile( std::string filepath, PointCloud<double> &cloud )

{
    cloud.pts.reserve(10000000);
    cout<<"Reading data ..."<<endl;

    // 1. read in point data
    std::ifstream ptReader( filepath );
    std::vector<cv::Point3d> lidarPoints;
    double x = 0, y = 0, z = 0, color = 0;
    double nx, ny, nz;
    int a = 0, b = 0, c = 0;
    int labelIdx = 0;
    int count = 0;
    int countTotal = 0;
    if( ptReader.is_open() )
    {
        while ( !ptReader.eof() )
        {
            //ptReader >> x >> y >> z >> a >> b >> c >> labelIdx;
            //ptReader >> x >> y >> z >> a >> b >> c >> color;
            //ptReader >> x >> y >> z >> color >> a >> b >> c;
            //ptReader >> x >> y >> z >> a >> b >> c ;
            ptReader >> x >> y >> z;
            //ptReader >> x >> y >> z >> color;
            //ptReader >> x >> y >> z >> nx >> ny >> nz;
            cloud.pts.push_back(PointCloud<double>::PtData(x,y,z));}
        ptReader.close();}
}
```

```

std::cout << "Total num of points: " << cloud.pts.size() << "\n";}

void writeOutPlanes( string filePath, std::vector<PLANE>)
{
    // write out bounding polygon result
    string fileEdgePoints = filePath + "planes.txt";
    FILE *fp2 = fopen( fileEdgePoints.c_str(), "w");
    for (int p=0; p<planes.size(); ++p)
    {
        int R = rand()%255;
        int G = rand()%255;
        int B = rand()%255;

        for (int i=0; i<planes[p].lines3d.size(); ++i)
        {
            for (int j=0; j<planes[p].lines3d[i].size(); ++j)
            {
                cv::Point3d dev =
                planes[p].lines3d[i][j][1] - planes[p].lines3d[i][j][0];
                double L = sqrt(dev.x*dev.x + dev.y*dev.y + dev.z*dev.z);
                int k = L/(scale/10);
                double x = planes[p].lines3d[i][j][0].x,
                y = planes[p].lines3d[i][j][0].y,
                z = planes[p].lines3d[i][j][0].z;
                double dx = dev.x/k, dy = dev.y/k, dz = dev.z/k;
                for ( int j=0; j<k; ++j)
                {
                    x += dx;
                    y += dy;
                    z += dz;
                    fprintf( fp2, "%.6lf  %.6lf  %.6lf  ", x, y, z );
                    fprintf( fp2, "%d  %d  %d  %d\n", R, G, B, p );}}}}
                fclose( fp2 );}

void writeOutLines ( string filePath, std::vector
<std::vector<cv::Point3d> > &lines, double scale )
{
    // write out bounding polygon result
    string fileEdgePoints = filePath + "lines.txt";
    FILE *fp2 = fopen( fileEdgePoints.c_str(), "w");
    for (int p=0; p<lines.size(); ++p)
    {
        int R = rand()%255;
        int G = rand()%255;
        int B = rand()%255;
        cv::Point3d dev = lines[p][1] - lines[p][0];

```



```

double L = sqrt(dev.x*dev.x + dev.y*dev.y + dev.z*dev.z);
int k = L/(scale/10);
double x = lines[p][0].x, y = lines[p][0].y, z = lines[p][0].z;
double dx = dev.x/k, dy = dev.y/k, dz = dev.z/k;
for ( int j=0; j<k; ++j)
{ x += dx;
  y += dy;
  z += dz;
  fprintf( fp2, "%.6lf  %.6lf  %.6lf  ", x, y, z );
  fprintf( fp2, "%d  %d  %d  %d\n", R, G, B, p );}}
fclose( fp2 );}

```

```

void main()

```

```

{
    string fileData = "E://khu//wall_sub.txt";
    string fileOut = "E://khu//detectedline";
    // read in data
    PointCloud<double> pointData;
    readDataFromFile( fileData, pointData );

```

```

    int k = 20;
    LineDetection3D detector;
    std::vector<PLANE> planes;
    std::vector<std::vector<cv::Point3d> > lines;
    std::vector<double> ts;
    detector.run( pointData, k, planes, lines, ts );
    cout<<"lines number: "<<lines.size()<<endl;
    cout<<"planes number: "<<planes.size()<<endl;

```

```

    writeOutPlanes( fileOut, planes, detector.scale );
    writeOutLines( fileOut, lines, detector.scale );}

```

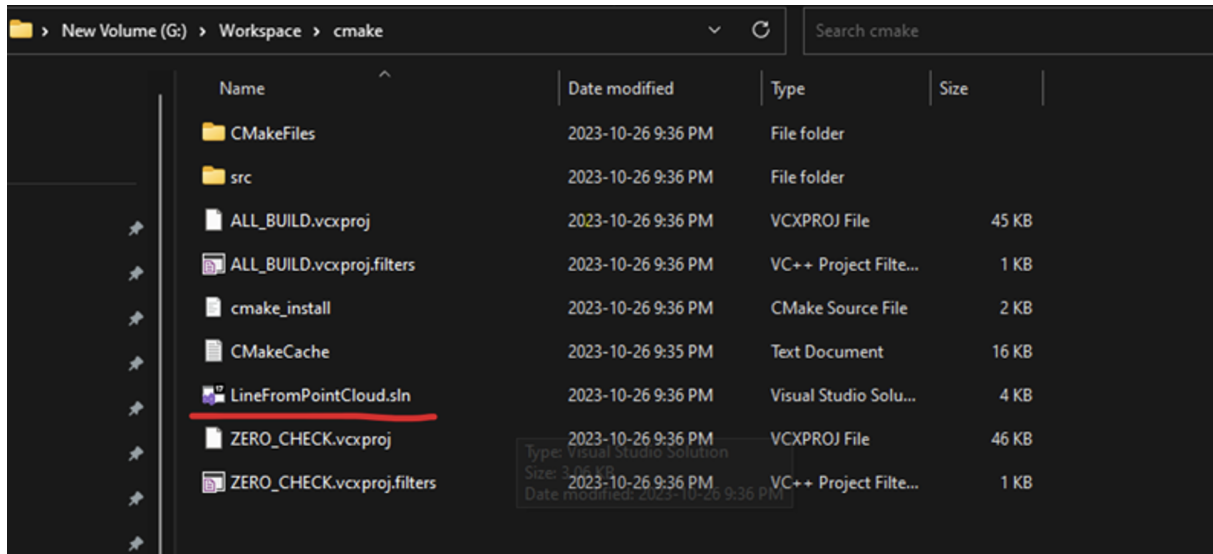
B2. CMake Configuration Console [4.4.1]

```
CMake Warning (dev) at CMakeLists.txt:2 (PROJECT):
cmake_minimum_required() should be called prior to this top-level project()
call. Please see the cmake-commands(7) manual for usage documentation of
both commands.
This warning is for project developers. Use -Wno-dev to suppress it.

The C compiler identification is MSVC 19.37.32825.0
The CXX compiler identification is MSVC 19.37.32825.0
Detecting C compiler ABI info
Detecting C compiler ABI info - done
Check for working C compiler: C:/Program Files/Microsoft Visual
Studio/2022/Community/VC/Tools/MSVC/14.37.32822/bin/Hostx64/x64/cl.exe - skipped
Detecting C compile features
Detecting C compile features - done
Detecting CXX compiler ABI info
Detecting CXX compiler ABI info - done
Check for working CXX compiler: C:/Program Files/Microsoft Visual
Studio/2022/Community/VC/Tools/MSVC/14.37.32822/bin/Hostx64/x64/cl.exe - skipped
Detecting CXX compile features
Detecting CXX compile features - done
CMake Deprecation Warning at CMakeLists.txt:6 (CMAKE_MINIMUM_REQUIRED):
Compatibility with CMake < 3.5 will be removed from a future version of
CMake.

Found OpenMP_C: -openmp (found version "2.0")
Found OpenMP_CXX: -openmp (found version "2.0")
Found OpenMP: TRUE (found version "2.0")
With OpenMP
OpenCV ARCH: x64
OpenCV RUNTIME: vc16
OpenCV STATIC: OFF
Found OpenCV: G:/Workspace/opencv/build (found version "4.8.0")
Found OpenCV 4.8.0 in G:/Workspace/opencv/build/x64/vc16/lib
You might need to add G:\Workspace\opencv\build\x64\vc16\bin to your PATH to be able to run
your applications.
OpenCV ARCH: x64
OpenCV RUNTIME: vc16
OpenCV STATIC: OFF
Found OpenCV 4.8.0 in G:/Workspace/opencv/build/x64/vc16/lib
You might need to add G:\Workspace\opencv\build\x64\vc16\bin to your PATH to be able to run
your applications.
OpenCV Libraries:
opencv_calib3d;opencv_core;opencv_dnn;opencv_features2d;opencv_flann;opencv_gapi;opencv_high
gui;opencv_imgcodecs;opencv_imgproc;opencv_ml;opencv_objdetect;opencv_photo;opencv_stitching
;opencv_video;opencv_videoio;opencv_world;opencv_calib3d;opencv_core;opencv_dnn;opencv_featu
res2d;opencv_flann;opencv_gapi;opencv_highgui;opencv_imgcodecs;opencv_imgproc;opencv_ml;open
cv_objdetect;opencv_photo;opencv_stitching;opencv_video;opencv_videoio;opencv_world
Configuring done (13.5s)
Generating done (0.0s)
```

B3. Generated CMake Solution Files (.sln) [4.4.1]



Appendix C

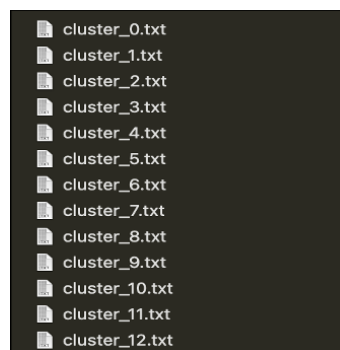
C1. Metrics in Semantic Segmentation [5.1.2]

```
(G:\env\pytorch_env)
PS G:\transformer\test_output_4096> python metrics.py
[[9.91430e+04 1.14300e+03 8.04000e+02 1.85000e+03]
 [1.11000e+02 4.85200e+03 1.69000e+02 3.90000e+01]
 [4.91000e+02 6.00000e+00 2.95210e+04 1.80000e+02]
 [1.39800e+03 5.60000e+01 4.09000e+02 3.52551e+05]]
The model overall accuracy is 0.9864913957740962
+-----+-----+-----+-----+-----+
|          | Precision | Recall | Specificity | F1 Score |
+-----+-----+-----+-----+-----+
| wall    | 0.963    | 0.98   | 0.99        | 0.971    |
| pillar  | 0.938    | 0.801  | 0.999       | 0.864    |
| vehicle | 0.978    | 0.955  | 0.999       | 0.966    |
| ground  | 0.995    | 0.994  | 0.987       | 0.994    |
+-----+-----+-----+-----+-----+
eval point avg class IoU: 0.907377
eval point accuracy: 0.986491
ave_F1_score: 0.948750
```

Appendix D

D1. HDBSCAN Clustering Console [5.2.1]

```
/Users/kristiehu/opt/anaconda3/envs/pyRANSAC3D/bin/python
/Users/kristiehu/Desktop/Surface/pyRANSAC-3D/slices_HDBSCAN.py
Data loaded successfully.
Data shape: (435480, 10)
XYZ points shape: (435480, 3)
Clustering completed.
Unique labels (clusters):
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16}
Estimated number of clusters: 17
Created directory: /Users/kristiehu/desktop/surface/12cluster/
Cluster 0 has 42664 points.
Saved file: /Users/kristiehu/desktop/surface/12cluster/cluster_0.txt
Cluster 1 has 43757 points.
...
Cluster 16 has 18057 points.
Saved file: /Users/kristiehu/desktop/surface/12cluster/cluster_16.txt
Saved file: /Users/kristiehu/desktop/surface/12cluster/cluster_16.txt
```



Appendix E

E1. Defined Class 'Plane_Model' [5.2.4]

```
class plane_model(object):

    def __init__(self):
        self.parameters = None
    def calc_inliers(self, points, dst_threshold):
        c = self.parameters[0:3]
        n = self.parameters[3:6]
        dst = abs(np.dot(points - c, n))
        ind = dst < dst_threshold
        return ind
    def estimate_parameters(self, pts):
        num = pts.shape[0]
        if num == 3:
            c = np.mean(pts, axis=0)
            l1 = pts[1] - pts[0]
            l2 = pts[2] - pts[0]
            n = np.cross(l1, l2)
            scale = [n[i] ** 2 for i in range(n.shape[0])]
            n = n / np.sqrt(np.sum(scale))
        else:
            _, _, c, n = SVD(pts)
        params = np.hstack((c.reshape(1, -1), n.reshape(1, -1)))[0, :]
        self.parameters = params
        return params
    def set_parameters(self, parameters):
        self.parameters = parameters
```

Appendix F

F1. Re. RANSAC Multi-Plane Fitting [5.2.4]

```
if __name__ == "__main__":
    path = "/Users/kristiehu/desktop/surface/1206_cluster/"

    for filename in os.listdir(path):
        if filename.endswith('.txt'):
            file_path = os.path.join(path, filename)
            xyz_points = load_points(file_path)[: , :3]
            if xyz_points.size == 0:
                print(f"No data in file: {filename}")
                continue

            plane_set, plane_inliers_set, data_remains =
ransac_plane_detection( xyz_points, 3, 5, max_trials=1000,
stop_inliers_ratio=1.0, initial_inliers=None,
out_layer_inliers_threshold=230,
out_layer_remains_threshold=230)
            if len(plane_set) == 0:
                print(f"No planes detected in file: {filename}")
                continue

            plane_set = np.array(plane_set)
            all_inliers.append(plane_inliers_set)
            print("==== Plane Parameters =====")
            print(plane_set)
            show_3dpoints(plane_inliers_set)
            output_path = "/Users/kristiehu/Desktop/line/m_wall.txt"
            merge_and_save_planes(all_inliers, output_path)
```

Appendix G

G1. cGAN - Generator [5.4.2]

```
class Generator(nn.Module):

    def __init__(self):
        super(Generator, self).__init__()
        # Encoder
        self.enc1 = nn.Conv2d(in_channels=1, out_channels=64,
                               kernel_size=4, stride=2, padding=1)
        self.enc2 = nn.Conv2d(64, 128,
                               kernel_size=4, stride=2, padding=1)

        # Decoder
        self.dec1 = nn.ConvTranspose2d(128, 64,
                                        kernel_size=4, stride=2, padding=1)
        self.out = nn.ConvTranspose2d(64, 1,
                                       kernel_size=4, stride=2, padding=1)

    def forward(self, x):
        # Encoder
        x = F.leaky_relu(self.enc1(x), 0.2)
        x = F.leaky_relu(self.enc2(x), 0.2)

        # Decoder
        x = F.relu(self.dec1(x))
        x = torch.tanh(self.out(x)) # tanh for output
        return x
```


G2. cGAN - Discriminator [5.4.2]

```
class Discriminator(nn.Module):

    def __init__(self):
        super(Discriminator, self).__init__()
        self.layer1 = nn.Conv2d(in_channels=2, out_channels=64,
                                kernel_size=4, stride=2, padding=1)
        self.layer2 = nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1)

        # Final layer
        self.final = nn.Conv2d(128, 1,
                                kernel_size=4, stride=1, padding=0)

    def forward(self, input, target):
        x = torch.cat([input, target], dim=1)
        x = F.leaky_relu(self.layer1(x), 0.2)
        x = F.leaky_relu(self.layer2(x), 0.2)

        # Additional layers would be processed here

        x = self.final(x)
        # Flatten the output and return a single value per image pair
        return torch.sigmoid(x.view(-1, 1).squeeze(1))
```

G3. cGAN - Class 'LineA' [5.4.2]

```
class LineA(Dataset):

    def __init__(self, input_dir, target_dir, transform=None):
        self.input_dir = input_dir
        self.target_dir = target_dir
        self.transform = transform
        self filenames = os.listdir(self.input_dir)

    def __len__(self):
        return len(self.filenames)

    def __getitem__(self, idx):
        input_img_path = os.path.join(
            self.input_dir, self.filenames[idx])
        target_img_path = os.path.join(self.target_dir,
            self.filenames[idx])
        input_img = Image.open(input_img_path).convert('L')
        target_img = Image.open(target_img_path).convert('L')
        if self.transform:
            input_img = self.transform(input_img)
            target_img = self.transform(target_img)

        return input_img, target_img
```

G4. cGAN - Class 'Real' [5.4.2]

```
class Real(VisionDataset):

    def __init__(self, root, transform=None):
        super(Real, self).__init__(root, transform=transform)
        self.directory = 'G://lineA//real//' # Set the root directory
        self.folders = ['1_fixed', '2_fixed', '3_fixed', '4_fixed']
        self filenames = []

        for folder in self.folders:
            folder_path = os.path.join(self.directory, folder)
            for filename in os.listdir(folder_path):
                self.filenames.append(os.path.join
                    (folder_path, filename))

    def __len__(self):
        return len(self.filenames)

    def __getitem__(self, idx):
        img_path = self.filenames[idx]
        image = Image.open(img_path).convert('L')
        if self.transform:
            image = self.transform(image)

        return image
```