Finding Specific Industrial Objects in Point Clouds using Machine Learning and Procedural Scene Generation

by

Daniel Felipe Lopez Morales

A thesis

presented to the University of Waterloo

in fulfillment of the

thesis requirement for the degree of

Doctor of Philosophy

in

Civil Engineering

Waterloo, Ontario, Canada, 2024

© Daniel Felipe Lopez Morales 2024

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Dr. Mani Golparvar-Fard Professor (University of Illinois at Urbana Champaign)
Supervisors:	Carl Haas Professor
	Sriram Narasimhan Professor
Internal Member:	Giovanni Cascante Professor
	Chul Min Yeum Professor
Internal-External Member:	David Clausi Professor (Department of Systems Design Engineering)

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the 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.

Statement of Contributions

Daniel Lopez was the sole author for chapters 1, 2, 6, and Appendices which were written under the supervision of Dr Carl Haas and Sriram Narasimhan and were not written for publication. This thesis consists in part of three manuscripts written for publication. Exceptions to sole authorship of material are as follows:

Research presented in Chapter 3:

Daniel Lopez was the primary author. This research was conducted at the University of Waterloo by Daniel Lopez with the supervision of Dr. Carl Haas and Sriram Narasimhan. Daniel Lopez contributed with the dataset generation methodology, point cloud balancing, formatting and implementation of the datasets, manually segmenting the validation point cloud, and training and validating the neural networks. Manolis Katsimpalis and Gabriel Earle provided the raw point cloud scan of the industrial training facility.

Citations:

- Lopez, D., Katsimpalis, M., Haas, C., & Narasimhan, S. (2024). Training networks to segment industrial scene point clouds: Dataset generation using TLS and synthetic data. Automation in Construction. Under review.

Research presented in Chapter 4:

Daniel Lopez was the primary author, and Harsh Patel was a secondary author. This research was conducted at the University of Waterloo by Daniel Lopez with the supervision of Dr. Carl Haas and Sriram Narasimhan. Daniel Lopez contributed with point cloud generation, point cloud balancing, formatting and implementation of the datasets, manually segmenting the validation point cloud, and training and validating the neural networks. Harsh Patel contributed with a subsection of the python implementation of the object spawning and the CHS object interaction with the scene. Manolis Katsimpalis and Gabriel Earle provided the raw point cloud scan of the industrial training facility.

Citations:

- Lopez, D., Patel, H., Katsimpalis, M., Haas, C., & Narasimhan, S. (2024). Realistic Procedural Generation of Point Cloud Datasets for Training Segmentation Networks in Industrial Environments. Automation in Construction. Under review.

Research presented in Chapter 5:

Daniel Lopez was the primary author, and Harsh Patel was a secondary author. This research was conducted at the University of Waterloo by Daniel Lopez with the supervision of Dr. Carl Haas and Sriram Narasimhan. Daniel Lopez contributed with the methodology to find specific object finding in point clouds, the implementation of the machine learning models used, the acquisition of the datasets used and the manual segmentation of the industrial scene dataset where specific object detection was tested.

Citation:

- Lopez, D., Haas, C., & Narasimhan, S. (2023). Specific object finding in point clouds based on semantic segmentation and iterative closest point. Automation in Construction, 156, 105116. https://doi.org/10.1016/j.autcon.2023.105116

Abstract

In the era of Industry 4.0 and the rise of Digital Twins (DT), the demand for enriched point cloud data has grown significantly. Point clouds allow seamless integration into Building Information Modeling (BIM) workflows, offering deeper insights into structures and enhancing the value of documentation, analysis, and asset management processes. However, several persistent challenges limit the current effectiveness of point cloud methods in industrial settings. The first major challenge is the difficulty in identifying specific objects within point clouds. Finding and labeling individual objects in a complex 3D environment is technically demanding and fraught with various issues. Manually processing these point clouds to locate specific objects is labor-intensive, time-consuming, and susceptible to human error. In large-scale industrial environments, the complexity of layouts and the volume of data make these manual methods impractical for efficient and accurate results. The second major challenge lies in the scarcity of industrial point cloud datasets necessary for training machine learning-based segmentation networks. Automating point cloud enrichment through machine learning relies heavily on the availability of high-quality datasets specific to industrial applications. Unfortunately, comprehensive datasets of this kind are either unavailable or proprietary, creating a significant barrier to developing effective segmentation networks.

Furthermore, the few current datasets often lack flexibility, being limited only by the areas that have been scanned. This rigidity, combined with the time-consuming process of manually segmenting data, slows down the development and deployment of scalable machine-learning solutions for point cloud segmentation. These limitations highlight the need for more flexible and adaptive solutions to efficiently address object detection, asset tracking, and inventory management in dynamic industrial scenarios.

This research addresses these challenges by developing open-access, weight-balanced class datasets specifically designed for 3D point cloud segmentation in industrial environments. The datasets integrate synthetic data with real-world industrial scans, offering a solution to the problem of imbalanced class distributions, which often hinder the accuracy of neural networks. Two methodologies for synthetic datasets were developed, one with random object placement and the second through a procedural generation pipeline, which includes rules for object placement and rules for generating tube structures for industrial elements, filling the scene with various objects of variable geometric features to understand the different effects that make a dataset realistic. This procedural generation technique provides a flexible method for

dataset creation that can be adapted for different objects, point cloud scales, point densities, and noise levels. The dataset improves the generalization capabilities of machine learning models, making them more robust in identifying and segmenting objects within industrial settings.

The second part of the research presents a methodology for efficiently and accurately identifying specific objects in point cloud scenes and two methodologies for creating open-access industrial datasets designed to train neural networks for segmentation. The first part of the research focuses on the object-finding methodology, which is crucial for multiple applications, including object detection, pose estimation, and asset tracking. Traditional methods struggle with generalization, often failing to differentiate between unique objects and general classes. The proposed methodology for specific object finding utilizes a point transformer network for point cloud segmentation and a fully convolutional geometric features network to enhance geometrical features using color. A key innovation in this process is using a color-based iterative closest point (ICP) algorithm on the output of the fully convolutional geometric features network. This enables precisely matching segmented objects with a point cloud template, ensuring accurate object identification.

Acknowledgements

This research receives the financial support of the Natural Sciences and Engineering Research Council of Canada (NSERC) and Ontario Power Generation (OPG) through the University Network of Excellence in Nuclear Engineering (UNENE)

I would like to express my profound gratitude to my advisors, Professors Carl Haas and Sriram Narasimhan, for their invaluable guidance and support throughout this journey. Their expertise, encouragement, and willingness to offer insightful feedback at every stage of my research have been essential in helping me grow both academically and personally.

I am also deeply thankful for the support of my colleagues and friends at Waterloo, who made this experience both rewarding and memorable. The time spent with you all has been filled with inspiration, collaboration, and friendship, making the challenges easier to overcome.

Lastly, I extend my deepest thanks and love to my family. Your unwavering support, boundless encouragement, and enduring belief in me have been the foundation of my strength. Without your constant presence and love, none of this would have been possible.

Dedication

To my father, whose memory is the driving force behind my achievements, I dedicate this work. Your wisdom, guidance and belief in me have always motivated me to reach a higher level of understanding of this world.

Table of Contents

Examining Committee Membership ii

Author's Declaration iii

Statement of Contributions iv

Abstract vi

Acknowledgements viii

Dedication ix

List of Figures xv

List of Tables xix

Chapter 1 Introduction 1

1.1 Background 1

1.2 Motivation 2

1.3 Overarching Contributions 4

1.4 Outline of the Thesis 4

Chapter 2 Literature Review 6

2.1 Procedural Generation of Training Data Set 6

2.2 Procedural Pipe Generation 10

2.3 Balancing objects in the dataset 10

2.4 Point Cloud Dataset Realism 12

2.5 3D model to Point Cloud 13

2.5.1 Scanned Object Approach 14

2.5.2 Topographic Approach 14

2.5.3 Scan-to-BIM change detection 15

2.6 Point Cloud Matching 15

2.7 Template Matching 16
2.8 Specific Point Cloud Matching 17
2.9 Neural Network Point Cloud Matching 18
2.10 Research Gaps and Contributions 20
Chapter 3 Training Networks to Segment Industrial Scene Point Clouds: Dataset Generation Using TLS and Synthetic Data 24
3.1 Introduction 24
3.2 Methodology 26
3.2.1 Object Generation 26
3.2.2 Object Standardization 27
3.2.3 Data Collection and Preparation 28
3.2.4 Point Cloud Generation 30
3.2.5 Hidden Point Removal 31
3.2.6 Weight-Balanced Class Dataset Generation 32
3.2.7 Dataset 33
3.3 Experiments 36
3.3.1 Experimentation Implementation 36
3.3.2 Real and Synthetic Datasets 36
3.4 Results 37
3.4.1 Dataset Balancing 38
3.4.2 Hidden Point Removal 39
3.4.3 Analysis 41
Chapter 4 Realistic Procedural Generation of Point Cloud Datasets for Training Segmentation Networks in Industrial Environments 45

4.1 Introduction 45
4.2 Methodology 46
4.2.1 Graph-Based Structure for Object Spawning 47
4.2.2 Integration with Synthetic Dataset Generation 48
4.2.3 Point Cloud Realism 51
4.2.4 Implementation Details 54
4.3 Experiments 56
4.3.1 Datasets 56
4.4 Results 58
4.4.1 Point Cloud Density and Noise Variability 59
4.4.2 Hidden point Removal 60
4.4.3 Per-Class Performance Analysis 61
4.4.4 Realism and Complexity 63
4.5 Analysis and Discussion 64
Chapter 5 Specific Object Finding in Point Clouds based on Semantic Segmentation and Iterative Closest Point 66
4.6 Introduction 66
4.7 Methodology 68
4.7.1 Target Point Cloud Resampling to Set Similarity Thresholds 69
4.7.2 Point Cloud Scene segmentation 70
4.7.3 Isolation and Clustering 71
4.7.4 Geometrical Colorization of the Point Clouds and Color ICP 72
4.8 Experiments 74
4.8.1 Dataset 76

4.8.2 Implementation Details 76
4.9 Results 78
4.10 Evaluation 80
4.11 Discussion 82
Chapter 5 Thesis Summary, Conclusions and Future Work 84
5.1 Summary 84
5.2 Conclusions 85
5.3 Future Work 87
References 90
Appendix A 122
Appendix B 124
B.1 Multi-Layer Perceptron 124
B.1.1 Activation Functions 126
B.1.2 Loss Functions 126
B.1.3 Backpropagation 127
B.1.4 Optimizers 127
Appendix C 128
C.1 Convolutional Neural Network (CNN) 128
C.1.1 Convolutions 129
C.1.2 Max Pooling 129
C.2 CNN Networks (VGG16) 130
C.3 Residual Networks 130
C.4 U-Net 131
C.5 Siamese Networks 132

C.6 Graph neural networks 132
C.7 T-Net 133
C.8 Other Neural Networks 133
C.8.1 Point Feature Analysis 134
C.8.2 Voxel Strategies 136
C.8.3 Hybrid Networks 138
Appendix D 143
Appendix E 173
Appendix F 188

List of Figures

Figure 1. Thesis Contributions per chapter 5
Figure 2. Procedurally generated Palladio 8
Figure 3. Procedurally generated room. 9
Figure 4. Synthetic Point cloud differences. 13
Figure 5. Thesis Methodologies 23
Figure 6. Methodology of Scene generation: 3D objects are transformed into point clouds through adjustments in point density, noise, and texture properties. These point clouds are then incorporated into the scene based on the balance of object count and prioritization index or weight over the dataset. 26
Figure 7. Object generation process entails: i) Stability AI Stable Diffusion 3 Medium creates high-quality images based on prompts; InstantMesh processing images by ii) removing backgrounds; iii) generating six views; and iv) constructing 3D models using instant-mesh-large model. 27
Figure 8. Object editor process entails: i) Simulate different lighting environments using predefined profiles, such as bright days or cloudy weather; ii) Modify point cloud rendering settings like scale, light intensity, and direction; iii) add class label of the object and save as GLB. 29
Figure 9. Real-world scan using FARO Focus Scanner that is used for validation. 29
Figure 10. Hidden point removal eliminates points occluded from the virtual scanner's point of view, leaving only the visible surface points in the scene. The hidden point removal on the I-Beam uses the scanner angle and a complete point cloud (left) to generate the occlusions that can be seen from different angles (right). 32
Figure 11. The image on the left presents a raw point cloud *Room* of the synthetically generated dataset, while the image on the right shows the corresponding segmentation. 34
Figure 12. The image on the left presents a section of real-world laser scan dataset, while the image on the right shows the ground truth of the manual segmentation of the real-world point cloud corresponding to the section on the left. 36
Figure 13. Application of the segmentation network on scene eight of the scan of an industrial point cloud using: i) Raw Scanned Point Cloud, ii) Segmentation's Ground Truth, iii) Unbalanced Dataset, iv) Unbalanced Dataset with HPR, v) Balanced Dataset, vi) Balanced Dataset with HPR. 43
Figure 14. Application of the segmentation network on scene five of the scan, showing the entrance section of an industrial point cloud using: i) Raw Scanned Point Cloud, ii) Segmentation's Ground Truth, iii)

Unbalanced Dataset, iv) Unbalanced Dataset with HPR, v) Balanced Dataset, vi) Balanced Dataset with HPR. 44
Figure 15. Overview of object hierarchy on spatial spawning. 48
Figure 16. The scene shows the different objects placed and the wireframe used for verification of placement and rotation. 49
Figure 17. Angle view of a room made with the procedural generation algorithm. 50
Figure 18. Top view of a room made with the procedural generation algorithm. 50
Figure 19 Point cloud noise from a floor of the real-world scan. 52
Figure 20 Section from a synthetic wall of the generated dataset, with 0 cm, 2 cm, and 5 cm added noise of the datasets. The outliers from the plane are shown in red. 53
Figure 21. Visualization of a point cloud room with 1000 point minimum density and 2cm noise objects. 56
Figure 22. Violin Plot of distribution of objects in the Datasets 57
Figure 23. Mean point segmentation accuracy performance of complete point cloud and HPR (hidden point removal) across varying point densities and noise levels. 59
Figure 24 mIoU point cloud segmentation performance of complete point cloud and HPR (hidden point removal) across varying point densities and noise levels. 60
Figure 25. Complexity of Area 10 ground-truth segmentation and segmentation by the dataset of 10000 points with 5cm noise. 65
Figure 26. Overview of the specific object-finding methodology. (1) Target resampling for candidate similarity thresholds. (2) Original scene is segmented, and (3) segmented objects with the same labels are clustered and become candidates. (4) Candidates are registered and compared to the thresholds. 69
Figure 27. Resampling methodology is used on chair 2 in office 3 from Area 4 from S3DIS point cloud dataset. a) Point cloud template b) meshed template c) Sampled template mesh with newly obtained points represented in blue. 70
Figure 28. Room segmentation using Stratified-Transformer network Lai (2022). Visualized with Cloud Compare 2.11 (2022) 71
Figure 29. Example of a) Isolating point clouds with same label from segmented scene and b) clustering point clouds to create candidates. 72
Figure 30. Geometric colorization output of FCGF network when taking template point cloud as input. 73

Figure 31. Colorization output of FCGF network of candidates. Candidate 1 should be accepted as it closely resembles the object. In contrast, candidates 2-4 display distinct geometries that deviate from template depicted in Figure 26. 73
Figure 32. Color ICP registration of colorized candidate point clouds to colorized template point cloud. The result of the ICP can be seen as candidates overlapped with template object. Candidate 1 has very similar points and can be seen as one object. Candidates 2-4 have resulting ICP transformations that do not show similar geometries. 73
Figure 33. Object found represented in red color. Visualized with Cloud Compare 2.11 (2022)..... 75
Figure 34. Synthetic point cloud industrial scene and corresponding scene with color labels. 75
Figure 35. Scene (on the left) and templates (on the right) for finding specific objects. a) b) and c) are slightly different chairs, but each has different point geometries that distinguish them. Large objects, such as a large table in d), are also found. 78
Figure 36. Four identical specific objects are found and visualized using template of a sofa..... 80
Figure 37. Finding specific object in construction environment. At right is the template of a large pipe. Left, shows the visualization of two identical specific pipes found using the framework 80
Figure 38. Confusion matrix of object finding framework. 81
Figure 39. Confusion Matrix 122
Figure 40. Machine Learning Neuron..... 124
Figure 41, Artificial Neural Network 125
Figure 42 Activation Functions..... 126
Figure 43. Back Propagation 127
Figure 44. Computer Vision Common Problems (Parmar, 2018). 128
Figure 45. Convolution..... 129
Figure 46. Max-pooling (Arnx, 2019) 129
Figure 47. VGG-16 Architecture (Hassan, 2018)..... 130
Figure 48. ResNet Module (He, 2015) 131
Figure 49. U-Net Architecture (Ronneberger, 2015) 131
Figure 50 Siamese Network Architecture (Dey, 2017) 132
Figure 51 Siamese Network results per convolution (Dey, 2017)..... 132
Figure 52. General Hierarchy of Networks..... 133
Figure 53 Point Net Network (Qi, 2017) 135

Figure 54 PointNet++ Network (Qi, 2017) 136
Figure 55 OctNet Voxelization (Riegler, 2017) 137
Figure 56 PVCNN architecture (Liu, 2019) 139
Figure 57. Hyper Dimensional Kernels for the Minkowski Engine (Choy, 2019) 141
Figure 58 Transformer Layer (Zhao, 2021) 141
Figure 59 Transformer Layer (Zhao, 2021) 142

List of Tables

Table 1. Frequency of objects in Agapaki's point cloud scan. 7
Table 2. Point cloud registration networks on 3DMatch Dataset. 19
Table 3. Performance metrics for evaluating point cloud generation and classification accuracy 38
Table 4. Mean accuracies (mAcc) for the objects in point cloud balance and HPR datasets. 40
Table 5. Point segmentation accuracy for different object classes across the eight real-life dataset subsections using the Balanced Dataset with HPR. The spaces mean that those objects were not in that scene 41
Table 6. Mean point segmentation accuracy per object class in each dataset 62
Table 7. Registration results from resampled point cloud and candidates using Colored ICP. Candidates with higher fitness value, lower Inlier RMSE and higher Correspondence Set than resampled threshold are accepted as found objects. 74
Table 8 Result of object finding tests using the S3DIS dataset for scenes and annotations 81

Chapter 1
Introduction

1.1 Background

Specific Point clouds have become a standard method for representing three-dimensional data in the construction industry, driven by the widespread availability of laser scanners and visual sensors. Beyond construction, sectors such as agriculture, oil and gas, and nuclear energy are increasingly adopting 3D point clouds to measure, survey, and visualize environments and objects within scans (Ogura, 2019; Huang, 2013; Agapaki, 2018). However, achieving this level of data abstraction requires manual annotation of point cloud data, a process that demands significant expertise and labor, thereby increasing extraction costs (Agapaki, 2020). To mitigate these challenges, automated scene segmentation and object detection within point clouds are essential for efficient annotation. This necessity underpins the motivation for this thesis, which proposes a methodology for specific object detection that employs a computationally efficient spatial query within large point cloud scenes.

Extracting valuable information from point clouds—such as tagging, counting, or verifying objects—often involves painstakingly isolating each element, a task that becomes even more cumbersome in dynamic environments where manual tracking is required, thereby escalating asset management costs. Furthermore, the levels of model maturity, as defined by the Level of Development (LoD) or Model Maturity Indices (MMI), vary significantly across engineering disciplines. For example, piping systems in industrial environments often require adherence to at least MMI 300 for basic engineering analyses, while structural components like steel frameworks demand the more detailed MMI 400 for advanced structural integrity assessments. This thesis addresses the challenges of meeting these diverse requirements by developing a flexible modeling approach capable of adapting to the specific LoD/MMI needs of varying engineering applications (Garcia, 2021). Fortunately, advancements in machine learning have significantly improved automated 2D and 3D computer vision tasks, including object detection and instance segmentation using point clouds. Modern computer vision and machine learning tools can effectively identify objects within a point cloud and extract pertinent data, such as type, location, orientation, and size (Cazorla, 2021). In industrial settings the level of detail (LoD) of data-rich model-driven approach to Scan-to-BIM can be measured using model maturity index (MMI). Nevertheless, in industrial settings, the scarcity of training datasets—stemming from poor documentation, privacy concerns, and a lack of diverse industrial scenes—

limits the ability of these algorithms to generalize across various applications. Consequently, the full potential of machine learning in this domain remains untapped.

To bridge this data gap, this thesis introduces a novel procedural generation approach for creating datasets tailored to industrial scenes. These synthetic datasets can train state-of-the-art machine learning algorithms for object detection in dynamic industrial environments. Labeled point clouds facilitate efficient object access through labels and provide a higher-level representation of the data. As automation in detection, segmentation, and asset management using point clouds advances, tasks such as tagging, automating maintenance, surveying, and conducting risk assessments (Agapaki, 2020) become more streamlined.

Secondly, this research focuses on the detection of specific objects within point clouds and their integration into facility management systems using cutting-edge machine learning techniques. The training is based on synthetic point clouds that accurately represent industrial plant environments. Particular emphasis is placed on industries managing legacy assets, such as the nuclear sector, where there is an urgent need for automation in object segmentation, detection, and asset management. This research aims to address these critical needs by enhancing the efficiency and reliability of point cloud data utilization.

1.2 Motivation

Object detection is a widely researched topic in computer science and engineering, both for 2D (Zaidi, 2021; Liu, 2019; Zou, 2019) and 3D (Jiao, 2019; Qian, 2021) objects. Early work in specific matches of 3D objects defined specific matches as candidate primitives C_j that match a prototype primitive P_i if the binary relation P_i is a subset of the binary relation C_j. Thus, every attribute-value pair (P, C) in the primitive P_i is also an element of the primitives C_j (Shapiro, 1981). Specific matching is more challenging with scanned point cloud's unstructured data since they suffer from noise, occlusions, and variable sparsity (Rao, 2021). Early approaches for specific matching would match the specific size of primitive surfaces and edges in a mesh to extract and match the features of an object (Shapiro, 1981). However, these approaches would only work for a small sample of objects, since each surface and possible variations would have to be calculated. Specific matching in point clouds requires a new definition and expansion towards efficiency in large point clouds. The closest problem to specific matching is template matching where the objective is to find potentially multiple occurrences of a small query point cloud in an extensive scene (Varga, 2021).

Template matching was first utilized to find specific pixel matches in images(Hemami, 1970; Cheng, 2016). In point clouds, template matching uses global and local approaches to utilize feature matching

methodologies. Global approaches use the entire point cloud to search for a template and are usually slow and limited to the classification and recognition of objects(Drost, 2010; Katsoulas, 2003; Zaharia, 2002; Greenspan, 1999). The local approach is used on smaller point clouds or point cloud subsets and is focused on finding local invariant features as a way of being more accurate but not specific(Gelfand, 2005; Mian, 2006; Rusu, 2009). The approaches used for template matching in point clouds rely on two main components, feature generation, and feature finding. Feature generation proposes geometrical attributes in subsections of the point clouds(Greenspan, 1999). Feature finding uses the generated attributes to match the features from a template to the modified point cloud scene(Sun, 2020).

Neural networks used in point cloud detection and segmentation use a similar template matching approach. The networks are taught in point cloud scenes or objects and learn the identifying geometric point cloud features of the training objects and their point labels. When querying a point cloud scene for identifying or segmenting, the networks use geometric feature's statistical similarity to identify the point's label(Simon, 2020; Buch, 2019). These networks rely on realistic training datasets to work well on real point cloud scans, however, many industrial areas are not available for scanning due to physical constraints or data confidentiality. Therefore, a feasible alternative involves generating bespoke datasets by combining synthetic data along with real-world scans. Real world scans that can be used for industrial point cloud scenes are found in Pipework by Yeo (2020), which focuses on pipe, valves and elbow elements. Sharif (2017) scans of multiple kinds of pipe scans are also useful since the addition of synthetic data to real scans in a dataset improves the network's ability to recognize objects(Danielczuk, 2019).

Point clouds are used for industrial asset management throughout the life cycle of a building. Asset management includes activities such as progress tracking, quality control, safety and asset monitoring(Liu, 2021; Xu, 2021; Wong, 2018). Salehi et al. (2017) demonstrates the use of point clouds to mirror factory machinery in the product lifecycle management (PLM). Salehi proposes simulations of hardware in the point cloud environment for equipment performance analysis. Another application by Qin et al. (2021) is the factory collision check on positional deviations in the factory. A building information model (BIM) is updated using the asset's point cloud position and rectifies the location of the assets in the BIM. The updated BIM checks for possible factory collisions in the process. An exhaustive literature review is provided in the next chapter.

1.3 Overarching Contributions

A research and industry needs survey revealed significant gaps in point cloud data generation, usage, and scope for industrial assets. Such bottlenecks currently limit asset operators and owners from fully harnessing the potential of point clouds. A summary of the main problems is:

1. Lack of efficient and accurate point cloud specific object finding algorithms.
2. Lack of industry-specific datasets.

This thesis seeks to address these gaps by developing new automated specific object detection algorithms trained with procedurally generated industry-specific datasets, where a target point cloud and a scene point cloud are provided as inputs, and a specific point cloud object can be found in the scene. Finding the specific object facilitates asset management in an industrial setting by tracking important objects' location and orientation. This procedure advances the latest point cloud segmentation, detection, and comparison methods, along with the possibility of achieving a broader scope in the types of objects that it can identify as constrained by a proposed procedurally generated dataset. The advances made possible in this thesis will allow for better point cloud segmentation, a benchmark for industrial point cloud scenes, and a useful temporal asset management tool. This work results in the following new contributions to the field:

1. An efficient specific object finding algorithm in a point cloud scene.
2. Comprehensive point cloud dataset generation framework with balanced classes for point cloud segmentation networks.
3. A framework for procedurally generating realistic point cloud datasets for industrial scenes.
4. Manual segmentation of a point cloud industrial scene for machine learning model validation.

These combined contributions significantly improve the efficiency and reliability of automated point cloud segmentation and object detection in industrial settings. The detailed contributions are illustrated in Figure 1. By offering robust solutions for data annotation, dataset generation, and model validation, this research represents a major advancement in integrating machine learning techniques into asset and facility management within complex industrial environments.

1.4 Outline of the Thesis

The thesis research was carried out in three main phases. In the first phase, focuses on building industrial datasets using a random placement algorithm and object balanced datasets. The third phase: (1) develops a

framework for realism in point cloud generation for machine learning segmentation networks, (2) tests whether realism in the dataset is truly necessary for achieving efficient point cloud segmentation on real-world data, and (3) establishes a methodology for generating procedurally generated realistic point cloud and object placement datasets. The second phase develops a framework to find specific objects in point clouds using machine learning segmentation, geometric feature point coloring, and iterative closest point algorithms.



Figure 1. Thesis Contributions per chapter

Chapter 2
Literature Review

This chapter provides the background literature and knowledge gaps related to point cloud matching, procedural generation, point cloud change analysis, and asset management. Point cloud matching section 2.1 provides background on template matching, specific point matching, and point cloud matching using machine learning techniques. The procedural generation section 2.2 section details the methodologies necessary to generate a balanced point cloud benchmark dataset using procedural generation, and object-based weight balance algorithms. Finally, the change analysis section 2.3 provides background on the tools and methodologies used in point cloud change visualization and asset management.

2.1 Procedural Generation of Training Data Set

This research focuses on industrial point cloud scenes. Such data is not widely available and some of it needs to be generated synthetically in order to apply point cloud segmentation and specific matching. For instance, a database applicable to industrial settings can be synthesized using these three options: TLS point clouds of real scenes, a synthetic dataset with realistic point cloud noise, or a combination of scanned and synthetic objects. Point cloud synthetic datasets have also been proven to work well with machine learning segmentation(Tremblay, 2018; Danielczuk, 2019; Danielczuk, 2019; Ma, 2020a; Ma, 2020b). The main objects for the network training and specific matching can be found in Agapaki (2018) study of the frequency of objects in British Petroleum plants. Agapaki's observation resulted in a total number of 1,308,936 objects. In essence: 22,143 objects from a sugar refinery, 240,687 objects from a petrochemical plant, 577,237 from a Gravity Based Structure (GBS) platform, 434,780 from a Tension-Leg Platform (TLP), and 34,089 objects from a fixed platform. The most frequent objects in the point cloud of a pipeline system are described in Agapaki et al Table 1 (Agapaki, 2018). Unfortunately, this dataset is not publicly available.

Table 1. Frequency of objects in Agapaki's point cloud scan.

		Frequency of appearance (average) (%)	Sample size (n)	Standard deviation (average)	Standard error (95 % Confidence Level)	Inter-project standard error
Piping object type	Straight pipe	52.1	192,081	303	1.0	$1.56*10^{-2}$
	Elbow	19.3	70,945	239	0.8	$1.25*10^{-2}$
	Flange	11.8	43,308	195	0.6	$1.09*10^{-2}$
	Tee & Olet	6.1	22,460	145	0.5	$0.81*10^{-2}$
	Valve	5.6	20,591	139	0.4	$0.76*10^{-2}$
	Other	2.2	8,137	89	0.3	$0.18*10^{-2}$
	Reducer	1.8	6,570	80	0.3	$0.36*10^{-2}$
	Cap	1.2	4,336	65	0.2	$0.17*10^{-2}$
Structural object type	CHS[a]	19.4	84,688	261	0.8	$1.77*10^{-2}$
	Channel	14.3	62,634	232	0.7	$1.72*10^{-2}$
	Solid bar	13.5	58,934	226	0.7	$0.34*10^{-2}$
	I-beam	13.1	57,314	223	0.7	$0.58*10^{-2}$
	Angle	11.9	51,886	214	0.6	$1.2*10^{-2}$
	Others	10.8	47,273	205	0.6	$0.98*10^{-2}$
	RHS[b]	9.2	40,439	192	0.6	$0.2*10^{-2}$
	PFT[c]	7.5	32,699	174	0.5	$0.28*10^{-2}$
	T-brace	0.4	1,663	41	0.1	$0.16*10^{-2}$
Electrical object type	Conduit	90.2	317,572	177	$29.8*10^{-2}$	
	Cable tray	6.1	21,585	142	$23.9*10^{-2}$	
	Electrical panel	2.1	7,397	85	$14.3*10^{-2}$	
	Lights	1.4	5,009	70	$11.8*10^{-2}$	
	Miscellaneous	0.07	250	16	$2.7*10^{-2}$	
	Alarm	0.05	190	14	$2.3*10^{-2}$	
	Speaker	0.03	120	11	$1.8*10^{-2}$	
	Others	0.01	39	6	$1.07*10^{-2}$	
	Power outlet	0.003	11	3	$0.53*10^{-2}$	

[a] Circular Hollow Section (CHS), [b] Rectangular Hollow Section (RHS), [c] Parallel Flanged Tee (PFT)

There exists one publicly available dataset for industrial scenes; e.g., Yeo et al.'s (2020) Pipework dataset for industrial pipe parts, which contains 15 objects with point cloud scans of each part focusing on elbows, flanges, and valves; the latest unreleased dataset is CLOI (Agapaki, 2020) which focuses on conduits, pipes, and elbows. Given the relatively scant publicly available datasets, new datasets need to be created with additional objects such as tanks, pumps, and machinery tailored to diverse industry types. Additional wall objects such as pipes and electric cables, grounded objects such as machinery, and ceiling objects such as crane rails also need to be generated. This can be achieved using procedural generation, which should be based on rules for the object placed in the scene while taking into account the balance of the objects as to provide a good benchmark of 3D point cloud industrial scenes to train neural networks (similar to S3DIS (Armeni, 2017) or KITTI, (Fritsch, 2013)which focus on the road and ground point cloud segmentation).

The scene design requires conceptual logic and rules such as objects not overlapping, continuous pipes and electrical wires, obstacles not being in the path of a door, and many others. The methodology can be set with Grammars, a system of formal rules that describe the structure of content generation. The grammars developed for 2D and 3D virtual scenes are the shape grammars. First exemplified in Figure 2 by Stiny et al. (1978) is Palladio's villas design. Their procedural hierarchy entails: 1) grid definition, 2) exterior-wall definition, 3) room layout, 4) interior-wall realignment, 5) principal entrances, 6) exterior ornamentation, 7) windows and doors, and 8) terminations where for every step the measures are based on previous villa measurements (Stiny, 1978).

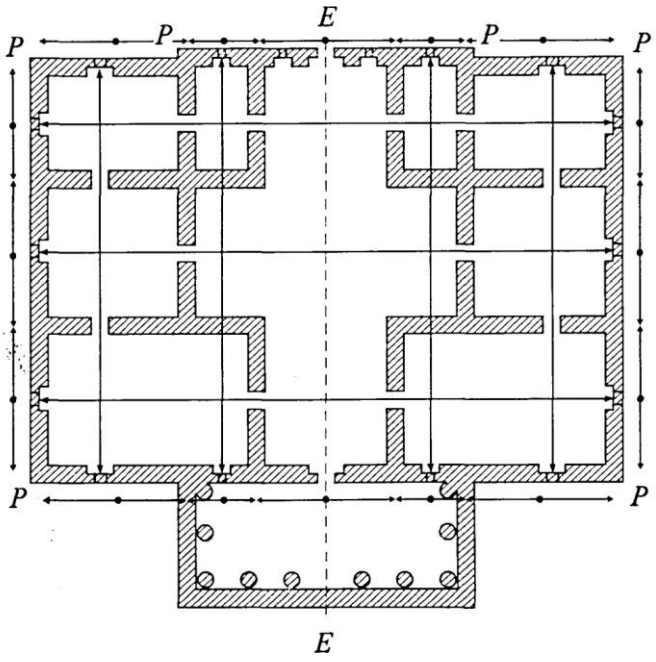


Figure 2. Procedurally generated Palladio

The industrial scenario requires room furnishing. Tobias Masters' Thesis (2017) explains a 3D generation of cities and room furnishing using expanding objects and random object placement. This method was made using rules related to: 1) placing an object along a random wall or a polygon side, 2) placing an object in the middle of the polygon, 3) orienting objects in rows inside the polygon, 4) randomly placing objects within the polygon, and 5) blocking polygons. Each rule is applied to polygons attached to a plane. For floor furnishing, rules 1, 2, and 3 are applied; for the roof, rules 3 and 4 are applied; for openings such as doors and hallways, rule 5 is applied. Tobias takes advantage of Unreal Engine 4, as their collision detection is preset not to have objects overlap.



Figure 3. Procedurally generated room.

While Tobias's method, seen in Figure 3, would be a fast approach, it does not consider a procedural logic such as connections or space relations, but predetermined polygon areas. Since the focus is on city generation, this approach works well when the number of objects to be added is limited to less than 20 objects. For a problem that requires more than 300 objects, this leads us to the approach taken in Ringqvist (2018) thesis , who created a procedural 2D layout and added the 3D objects according to the 2D plan while checking for obstructions. The hierarchy for the building is: 1) room dimensionality, 2) side constraints (i.e. doors and their dimensions), 3) unique object placement (i.e. a bed and a fridge should not be in the same room), 4) add wall objects with wall edge constraints, 5) register all furniture sides including wall attached objects, 6) set placement domain (all positions in room grid that fit), and 7) recursively test area placements until there is no overlap.

Ringqvist's process is more involved than Tobias'. While Ringqvist only used 43 different objects, it is helpful for hundreds of objects, since space constraints are in designated areas. It also takes into account wall constraints, which will be necessary for the use case of pipes. For wall objects, there is a need to procedurally generate pipe connections and electric cables. As previously mentioned, there are existing datasets for industrial pipes, but two additional industrial datasets remain inaccessible due to proprietary

licensing restrictions. Therefore, one of the outcomes of this research is the generation of an industrial scene point cloud benchmark.

2.2 Procedural Pipe Generation

As previously discussed, for a synthesized industrial dataset to provide structural reality, the walls of industrial settings require pipes and electrical cabling structures. Unfortunately, very little research has been performed on 3D procedural pipe structure generation. The only research to date on procedural tubes has been done by graph generation and independent Houdini software developers (Banks, 2014). Paris (2013) Master's thesis has developed a procedural tube generation using directed graphs for this problem. The graphs are generated using probabilities to generate branches, nodes, and end of branches. For the case of their research, significant pipe data exists through the work by Sharif (2021). The procedural connection of pipes made by Banks (2014) used user input to generate mesh intersections of various types of pipe objects. The direction angles are calculated to provide angle meshes while using modular pieces and repeating segments. The user input could be made procedurally using a modification of Hellinga (2019) Subway Map Generator, which uses Paris' thesis concepts. The pipe structure is similar to a subway continuity with stations representing the pipe's entrance and exit through a wall.

2.3 Balancing objects in the dataset

To properly train a network, a balanced input dataset is necessary. A class balance is achieved by modifying the class weights and inputs, removing over-sampled classes, or augmenting scenes of under-sampled classes (Zhu, 2019). Another effective approach, proposed by Boogaard (2022), involves dividing the original point clouds into chunks to address class imbalance. Chunks are generated by first selecting anchor points for each class and then incorporating neighboring points using a k-nearest-neighbor (k-NN) search. Boogaard further introduces a class-dependent strategy that enhances this process by increasing the number of anchor points for underrepresented classes, resulting in training datasets with higher entropy and improved class balance. At the time of this research the only class balance calculation while generating a scene, has been the approach by Griffiths (2019). The class balance calculation in equation 1 and 2 by Griffiths uses a class weighting approach which leads to balancing the dataset while it is being generated, which is explained more in depth in section 4.2. Given that the procedural layout generation and 3D spaces are configured, and there are class balance weight constraints, the next step is to generate the layout, which requires adding the objects to the scene, which is, in essence, the knapsack problem in 3D with irregular shapes. The knapsack problem(Hu, 2016) is a classical combinatorial optimization problem where given a set of objects, each with a weight and a value, determine the number of each item included in a limited

knapsack so that the total weight is less than or equal to a given limit and the total value is as large as possible.

The 3D knapsack problem with irregular shapes, or $3KP - (0/1)$, as studied by Ghomi et al. (2013), aims to optimize the coverage value R of a container C using a set of n 3D objects, denoted by $B = \{B1, B2, \ldots, Bn\}$. The coverage value R is calculated as:

$$R = \frac{\sum_{j=1}^{m} V(B_j)}{V(C)} \tag{1}$$

where $V\ (B_j)$ is the volume of the jth object in the subset m chosen from the total set of n 3D objects, and $V\ (C)$ is the volume of the container. Equation 1 assesses the efficiency of the selected objects' space within the container. Another term for this is "packing efficiency." Ghomi et al. (2013) developed a 3D packing algorithm that begins by randomly filling the volume, then employs a growth-based optimization algorithm focusing on the largest object's rotation and translation relative to others, utilizing the Chordal Axis Transform (CAT) for optimizing the coverage rate. This approach is compatible with class-balancing techniques, offering an efficient solution to the 3D knapsack problem within the context of dataset generation. Class imbalance, notably impactful in supervised learning tasks requiring varied class-labeled data, results in inadequate learning of underrepresented class features, potentially leading to incorrect predictions. Addressing class weights in point clouds involves modifying the cross-entropy loss to prioritize classes with fewer objects, considering the size variations within the scene for a more equitable training focus. Building upon this, Griffiths and Boehm (2019) adjust class weights based on point prevalence per class, thus favoring classes with fewer samples and smaller objects. Their class weight calculation, essential for creating balanced datasets and enhancing point cloud analysis, is encapsulated in the following equation:

$$w = (t_{max} - t_{min})\left(\frac{-\sum_{i=1}^{n}[P_i = x] - \min P}{\max P - \min P}\right) + t_{min} \tag{2}$$

where t_{max} and t_{min} denote the maximum and minimum weight thresholds respectively, P represents the entire point set, and x is the class for which the weight w is calculated. Equation 2 ensures neural networks trained on these datasets can deliver reliable and unbiased results, enhancing the understanding and analysis of industrial applications. However, none of these methodologies consider the object's size when comparing it with its class. Therefore, larger objects might get overrepresented in the point clouds in the same class, and smaller objects might not be detected. When the smaller objects are not properly trained, the accuracy and mIoU will not suffer due to the percentage point representation of larger objects of the same class. However,

this is an artificial outcome that cloaks true performance, because the network will not learn the geometrical features of the smaller objects of the same class. Small objects such as sensors, valves and instrumentation can sometimes be the highest value objects, depending on the real-world application. The methodology presented in this chapter addresses this gap.

2.4 Point Cloud Dataset Realism

Generating objects and procedural environments is essential for creating realistic point cloud datasets. Achieving realism includes addressing various factors that replicate the imperfections, variability, and spatial arrangements found in real-world scan data (Noichl, 2021; Triess, 2021). Realism in this context includes geometric accuracy, noise, incomplete data, color variations, and the precise placement of objects, all of which are important for developing robust and adaptable machine-learning models.

One key aspect of realism is the introduction of noise. In real-world scanning, noise results from sensor limitations, environmental factors, and surface reflectivity. To replicate these conditions, synthetic datasets must include noise (Triess, 2021). This enhances the ability of machine learning models to handle imperfect data and helps prevent overfitting to ideal, noise-free inputs. Training on noisy data improves model performance in real-world scenarios (Gao, 2022).Accurate placement and arrangement of objects are also essential for realism. In real environments, objects follow specific spatial constraints influenced by available space, regulations, and workflow requirements. Synthetic datasets must replicate these spatial relationships and object positioning variations to ensure that models generalize well across diverse industrial scenarios. Object placement must account for occlusions, shadows, and proximity effects, which increase complexity and challenge models in cluttered environments (Triess, 2021).

Another vital factor in realism is handling incompleteness and variations in point density. Real-world point clouds often contain missing data or irregular point spacing due to occlusions, sensor range limitations, or scanning angles (Triess, 2021). These characteristics must be included in synthetic datasets to reflect real-world conditions accurately. Incompleteness and density variations introduce uncertainty, requiring models to infer missing information and adapt to different levels of detail (Gao, 2022). This is particularly relevant in industrial settings, where sparse or incomplete data is common. These challenges have been explored in recent literature, such as the work of Garcia et al. (2021), which evaluates the productivity metrics in model-driven engineering for capital project delivery. Similarly, Dimitrov et al. (2016) emphasize the complexities in non-uniform B-spline surface fitting for 3D modeling, underscoring the need for accurate segmentation methodologies. Additionally, Perez-Perez et al. (2021) provide insights into the potential of deep learning

techniques like Scan2BIM-NET for semantic segmentation in industrial point clouds. By building on these foundational studies, this research further delves into resolving the current inefficiencies in automated modeling workflows.

Color variations and interactions between nearby objects also contribute to realism. Objects in close proximity may create occlusions, shadows, or subtle shape distortions that affect the appearance of the point cloud. These variations are necessary for training models to distinguish between overlapping or adjacent objects, a frequent challenge in industrial environments [49].Ultimately, synthetic point cloud datasets must mirror the complexity and imperfections found in real-world data to be truly realistic and effective[7]. Incorporating noise, incompleteness, density variations, color shifts, and the interactions between objects ensure that the generated point clouds capture the intricate details of actual industrial environments(Noichl, 2021; Nunez-Morales, 2024; Diniz, 2021). This level of realism is essential for improving the performance of segmentation algorithms and enabling their broader application in Industry 4.0 and Digital Twin technologies, where the accuracy and adaptability of machine learning models are paramount.

2.5 3D model to Point Cloud

Changing BIM and 3D models to point clouds to generate realistic synthetic data has been studied (Cook, 1986). Early works use ray casting on a 3D surface and generate random point sampling (Roth, 1982). An alternate point sampling method is Poisson-Disk on planar and mesh domains such as Dart Throwing (Cline, 2009). Dart throwing is a simple algorithm that generates samples and discards points that do not meet the minimum distance constraint. More complex and time-efficient methods have been created. Hierarchical Dart Throwing (HDT) by White (2007) generates quadtree subdivisions to sample a planar domain. Bowers et al. (2010) use a parallel generation of N-dimensional Poisson-disk samples by

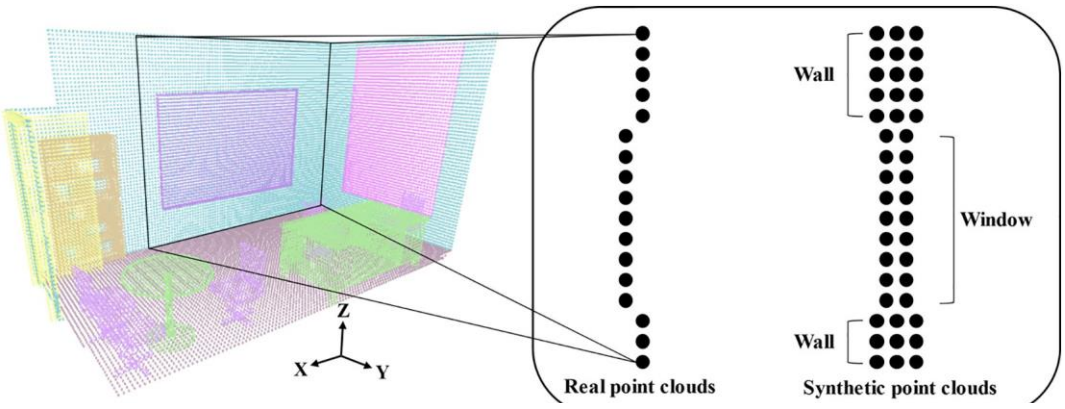


Figure 4. Synthetic Point cloud differences.

subdividing the domain and arranging the phase groups or element subsets to be processed in parallel. This last approach would be the starting point for the sample generation since the synthetic point cloud difference to a real point cloud scan is the noise and point cloud sparsity that is generated as shown in Figure 4 (Ma, 2020).

2.5.1 Scanned Object Approach

The topographic approach is separated into point-cloud-to-point-cloud (Beckley, 2018; Palma, 2015; Pătrăucean, 2015) point-cloud-to-mesh, and mesh-to-mesh change detection (Yotov, 2015; generation, 2016; Antova, 2019). The point cloud of an object can be compared by registration, computing the distances between the point clouds, finding the nearest point, and computing the euclidean distance. This is the approach of Palma et al. (2015), which initially samples the point cloud by removing outliers and computing the radius and point normals. Then, the Growing Least Square (GLS) method is used to compute multi-scale differences for each point to detect inconsistencies using point region analysis. Finally, it corrects inconsistencies such as occlusions using the surrounding points' information and their temporal proximity. Poullis (2013) developed a method for automatic modeling from point cloud data by preprocessing the point clouds into manageable datasets, which are then separated into clusters using a novel two-step, unsupervised clustering algorithm. Each cluster boundary is refined using a fast energy minimization process, and three-dimensional models are generated by outlining large flat surfaces.

2.5.2 Topographic Approach

The second method is having the point clouds transformed into meshes or surfaces, also called Digital Surface Models (DSMs), to detect changes by measuring the difference of the point cloud to the surface normal as shown by Lague et al. (2013). As described by Antova (2019), the third change analysis approach generates surfaces from the point clouds and compares the differences using surface normal differences. A similar approach by Braun et al. (2015) is where the point clouds are matched against a Building Information Model (BIM) to verify the component's existence. Other neural network approaches such as SiamGCN explained in Appendix C section C.5 are used by Ku et al. (2021), takes representative geometric features on point clouds and compares the changes. While all of these methods require point cloud registration, Shen et al. (2016) use the reference point cloud to generate baselines, which are connections of point features within a scan. These connections are compared to identify changes in the point clouds. A baseline is generated using principal component analysis (PCA) in the point cloud to fit a plane of the point

cloud. The baselines of the planes are compared and the normal vectors are matched to find the translation and rotation matrix transformation. This computationally fast approach could provide translation and rotation differences between point clouds and it can be implemented using the point cloud library (Ioannou, 2012).

2.5.3 Scan-to-BIM change detection

While knowing where the differences between point clouds are is essential, the knowledge of assets and their components in the organization's point clouds generates asset management value (Wang, 2015), knowing where the assets are and where they are moved when comparing the scan-to-BIM with the already generated BIM (Bosche, 2015). An early scan-to-BIM on an industrial point cloud in Pang et al. (2015) has approached the point cloud similarities to already made 3D models and replaces the models directly into the scan. However, it is only able to identify walls, pipes, and valves, and experiences difficulty with angles. This change detection data can be used for measuring, providing decision support functions, and providing a high level of information to an information model (Xu, 2021) such as BIM. Another approach "Scan-vs-BIM" by Bosche (2008) can automatically match each scan point to a 3D model object. Later, Bosche (2015) published focusing on identifying objects with generated BIM in point clouds for construction monitoring. Reinforcing "Scan-vs-BIM" using the Hough transform-based circular cross-section approach to handle partially occluded cylindrical objects, and metrics for inferring the object recognition and identification from proximity analysis. These applications can be applied to the Model Maturity Index (MMI) definitions and a Model Maturity Risk Index (Model MRI) toolkit to track the maturity of an engineering process, providing project teams with tools to benchmark progress and model completeness Garcia (2021). This approach enables integration of engineering workflows with BIM-driven processes, enhancing proactive project management.

2.6 Point Cloud Matching

Pattern matching and object recognition in point clouds enables the detection and estimation of an object's pose and location. Some of the earliest work by Dyer (1984) utilized model-based recognition based on three predominant and complimentary feature extractions: 1) the global scalar features such as size, perimeter, and area of the object, 2) the global geometrical features such as edges and curves of the object, 3) the local shape features that are used to describe the relationship between the features. The matching would compare the measured features of new objects with known features using a decision tree. Other early methods used Chin's methodology to find specific 3D representations by measuring surface, sweep, and volume and combining multi-view feature descriptions into a comparative model (Oshima, 1983). The

multi-view matching methods would store 2-D descriptions for every view and use intra-view (i.e., in one image) and inter-view (i.e., other images) feature extraction and matching (Oshima, 1983). Multi-view advancements and feature matching for object recognition led to point cloud registration. Integrated systems extract shape information from contours and the joint recognition of 3-D curved objects and the patterns on their surfaces (Wang, 1994).

Object recognition, template matching, and point registration are achieved within specific matching, where alterations of objects occur due to noise (Shapiro, 1981) or for purposefully flexible point cloud feature extraction (Zhou, 2021). Specific matching refers to a specific template matching where every attribute-value pair in the primitive template is also an element of the primitive point cloud scene(Shapiro, 1981). The closest approach to specific matching is template matching which aims to find all the possible occurrences of a small query point cloud in an extensive point cloud scene (Liu, 2020).

2.7 Template Matching

Template matching started in character recognition in images (Bowers, 1964; Fu, 1968; Hemami, 1970) using pixel feature matching. Then, template matching extended into higher dimensions such as video (Jurie, 2002; Sun, 2020), time dependent monitoring (Lin, 2011; Larsen, 2016), and point clouds (Greenspan, 1999). In point clouds, the template matching problem is defined as finding all occurrences of a template or query point cloud representing an object in a large-scale point cloud scene (Varga, 2021). Initial 3D template matching in 1983 by Oshima et al. (1983) focused on planar and curved surfaces by grouping the points into small surface elements, merging and classifying the surface elements into planar and curved. The classified surface elements are extended by merging adjacent surfaces with the same classification. Greenspan et al. (1999) developed point cloud template matching using a voxelized object model. The matching algorithm used a decision tree constructed using the object where each leaf node references possible object templates. Each voxel was measured and compared to traverse the decision, in case there is no specific match; the highest estimate is given by voxel neighborhood likelihoods.

In seminal work, Drost et al. (2010) created the point pair feature matching method. The point pair feature describes two oriented points' relative position and orientation and applies it to the scene. A fast-voting scheme is then used to optimize the points poses in a locally reduced search space parameterized in terms of points on the model and rotation around the surface normals. The voting scheme creates a global model description based on oriented point pair features. The matching takes an arbitrary point and its

neighbors to align the template to the scene. The normals are then matched for every possible point of the template in the scene. Multiple feature matches are accumulated in a vector accumulator space, and the votes are cast per local coordinate. This method considers possible occlusions but does not match the object specifically as the matching only covers parts of the object. Recent work by Vock et al. (2019) on template matching introduces pose estimation. Vock modified Drost's algorithm using a novel principal component analysis (PCA) edge detection approach to select salient points and reduce the validation computations by extensive point pair hypothesis testing. In addition, the point cloud is voxelized to reduce the point size while preserving the point features. This improvement increases the point coverage in the scene but adds match flexibility, accepting noise on each match.

Hinterstoisser (2011) created a multimodal-LINE algorithm (LINE-MOD), which combines gradient information and point normals to generate multiple modalities to perform template matching in small point clouds. The LINE approach, which uses only the image intensities, was modified to generate a 3D-LINE approach using depth maps for the LINE-MOD algorithm. However, this algorithm only solves pose estimation using the contours, and objects with similar contours would be misidentified. Miao et al. (2021) used the LINE-MOD and added iterative closest point (ICP) between the model point cloud and the scene point cloud after a LINE-MOD match to fix this problem.

Varga et al. (2021) proposed a point cloud pre-processing feature description and storage of described scenes in a database. Varga used point cloud transformations with outlier detection and vector computations and the Fast Point Feature Histogram (FPFH) method (Rusu, 2009) to generate point cloud vector descriptions. Afterward, principal component analysis was used to lower the size of the vectors and capture the five largest eigenvalues. The scene is transformed through this process, and the principal features are saved in the database. The template is then matched using a bipartite graph and nearest neighbor algorithms on the transformed template to match similar objects. Finally, Varga uses iterative closest point to generate a correspondence between template and object in the scene.

2.8 Specific Point Cloud Matching

Point cloud registration is the search of the rigid transformation that will align two point clouds. Much of the research in point cloud registration is made to align two point clouds of similar sizes (Huang, 2021; Gelfand, 2005; Agarwal, 2017; Guan, 2018; Sehgal, 2010; Lucas, 1981). Other authors mention specific matching (Guan, 2018; Sun, 2018) but only provide error minimization algorithms like iterative closest point without using specificness. Early papers generate a specific match between point clouds(Shapiro,

1981; Feng, 2009) using template matching. Specific point cloud matching is defined as a candidate primitive C_j matches a prototype primitive P_i if the binary relation Pi is a subset of the binary relation C_j. Thus, every attribute-value pair (a, v) in the primitive P_i is also an element of the primitive C_j (Guan, 2018).

Nuchter et al. (133) present a modification of Greenspan (1999) and Arya (1998) binary tree match, namely cached k-d trees for computing closest points iteratively. K-d trees are a generalization of binary search trees. Every node represents a partition of a point set to the two subsequent nodes. The root represents the entire point cloud, and the leaves or buckets provide a complete disjunct partition of the points. Nuchter's cached k-d Tree representation uses an iterative closest point variant that uses the specific closest point search. In the first iteration, a k-d tree search is used to compute the nearest points. The return function of the tree is modified such that the leaf containing the closest point is returned and stored in the cache vector of point pairs.

Another specific ICP match is Marani et al.'s (2016) modification of the iterative closest point algorithm by introducing the concept of a deletion mask. Deletion masks are introduced to remove the erroneous point correspondences extracted from ambiguous regions, where implicit differences can be raised due to the change of the sensor viewpoint. A virtual scan of the mesh re-samples the reconstructed surfaces starting from positions defined by the direction of a unit vector, having origin in a specific initial point. The iterative closest point starts with an empty deletion mask, taking the reference of the virtual scan. The point cloud is roto translated and superimposed. A pairwise comparison obtains the overestimation of the loss function and populates the deletion mask by sample comparison. The deletion mask is used to remove over estimated points and leave matched points at the end of the iterative closest point. It is important to note that these approaches do not define specific matching.

2.9 Neural Network Point Cloud Matching

Advancements in machine learning for point cloud feature detection have facilitated neural networks to generate instance segmentation and registration in point clouds (Zhang, 2020). Most machine learning point cloud registration methods focus on matching learned key points and features to minimize the differences between point clouds (Lu, 2019; Aoki, 2019; Sarode, 2019). A mixed approach is devised by Macron et al. (2020). Using RGBD data, the RGB image is classified using a VGG16 (Appendix C section C.2) network. The classification is queried from the database containing RGBD data from multiple views of the same object. The pose of the object is obtained using a local-descriptors-based approach. Finally, both RGBD

point clouds are registered and matched using colored iterative closest point. This approach is applicable only if there is adequate image data of the object, and all the RGBD views of the object are obtained.

Other machine learning approaches combine all the parts of the previous pipeline by Macron into one neural network architecture. Aoki et al. (2019) merged PointNet and ICP into PointNetLK, where PointNet is modified by removing the T-Net (Appendix section C.7) layers and instead is looped as a recurrent network using the Lucas and Kanade (LK) algorithm (Lucas, 1981). Machine Learning in Appendix B layers estimate the transformation and generate a descriptor of each point referencing the target mesh. The descriptors generate a global feature and align the point cloud using rigid exponential map transformations. This architecture can find the point cloud registration faster than ICP on large point clouds. A similar approach shown in Appendix C Figure 50 by Sarode et al. (2019) used a siamese network (Appendix section C.5) rather than the PointNet architecture to generate the descriptors. A comparison of these methods can be seen in Table 2 from Zheng et al. (2022).

Table 2. Point cloud registration networks on 3DMatch Dataset.

Model	Estimator	#Samples	RR(%)		Time(s)		
			3DM	3DLM	Model	Pose	Total
FCGF	RANSAC-*50k*	5000	85.1	40.1	0.052	3.326	3.378
D3Feat	RANSAC-*50k*	5000	81.6	37.2	0.024	3.088	3.112
SpinNet	RANSAC-*50k*	5000	88.6	59.8	60.248	0.388	60.636
Predator	RANSAC-*50k*	5000	89.0	59.8	0.032	5.120	5.152
CoFiNet	RANSAC-*50k*	5000	<u>89.3</u>	<u>67.5</u>	0.115	1.807	1.922
GeoTransformer	RANSAC-*50k*	5000	**92.0**	**75.0**	0.075	1.558	1.633
FCGF	weighted SVD	250	42.1	3.9	0.052	0.008	0.056
D3Feat	weighted SVD	250	37.4	2.8	0.024	0.008	0.032
SpinNet	weighted SVD	250	34.0	2.5	60.248	0.006	60.254
Predator	weighted SVD	250	50.0	6.4	0.032	0.009	0.041
CoFiNet	weighted SVD	250	<u>64.6</u>	<u>21.6</u>	0.115	0.003	0.118
GeoTransformer	weighted SVD	250	**86.5**	**59.9**	0.075	0.003	0.078
CoFiNet	LGR	all	87.6	64.8	0.115	0.028	0.143
GeoTransformer	LGR	all	**91.5**	**74.0**	0.075	0.013	0.088

The Dynamic Graph CNN (DGCNN) by Wang et al. (2019) uses the Edge Convolutional module to find common features in a point cloud. Symmetrical point clouds can encode similar features and therefore segment the point cloud faster by representing the points as nodes and edges in a graph. The graph connections of a region connected to a point become a feature if found again with the same number of edges and points. This work was used by Wang et al. (2019) in the Partial Registration Network (PRNet) by modifying the pre-trained network to work as a Siamese Graph Convolutional Networks (SiamGCN) and

learning the features from both the target point cloud and the source point cloud. However, the size of the point clouds was relatively small as the maximum test was with 4 million points.

Another advance is the Fully Convolutional Geometric Features (FCGF) network by Choy et al. (37). This network based on the Minkowski Engine focuses on fully-convolutional geometric features computed by a single pass through the fully convolutional network. Using a residual network (Appendix section C.3) and a U-net architecture (Appendix section C.4), this network can colorize the point cloud by comparing the geometric feature differences. The network inputs the point cloud through sparse convolutions and element-wise non-linearities, transforming the point cloud into a sparse tensor to get the sparse fully-convolutional features while using the unique loss function of hardest-contrastive and hardest-triplet.

The latest point registration neural network is the Sticky Pillars approach by Simon et al. (2020). Sticky Pillars use a pillar layer, a graph network layer, and an optimal transport layer to produce the point cloud feature matches. The inputs of the network are two point clouds to be registered. The pillar layer is composed of a sparse key point selection and two auto-encoders. The sparse key point selection uses LOAM (Zhang, 2014), a real-time edge and surface point selection method. The first auto-encoder is used for point features and the second for a positional encoder for location features. The point and positional features are passed to the graph neural network layer explained in Appendix section C.6. Finally, the optimal transport layer is a score matrix constructed by computing the unnormalized cosine similarity between each pair of features. A soft-assignment matrix is computed that contains matching probabilities for each pair of features, which lowers the accuracy when the size of the point clouds being compared varies.

The specific match in template matching is a non-trivial gap and still an open research problem (Cuturi, 2020). All these networks represent various non-specific match use cases. Template mat ching uses typical machine learning segmentation networks and ICP approaches to match the templates to the larger scene, including point closeness approximations and neural network generalizations of objects. The point cloud registration networks result in fast matches but with errors and approximations in the points selected as the part of a match, similar to ICP. Geometric feature matching provides specific point matches, but the point matches do not create a correspondence between multiple objects. The accuracy declines as both point clouds vary in size, which doesn't allow template matching.

2.10 Research Gaps and Contributions

Despite the growing adoption of point clouds in various industrial sectors, significant knowledge gaps persist that hinder the effective utilization of this technology. Early research in specific object matching,

such as Dyer's work (1984), has demonstrated inefficiencies when applied to large-scale data and point cloud inputs. Furthermore, existing literature on template matching and machine learning predominantly focuses on generalized features that are applicable to a broad set of similar objects, rather than targeting specific objects of interest. While there are studies addressing specific matching (Guan, 2018; Sun, 2018), these works often lack clear definitions and do not provide concrete matching approaches tailored to particular objects.

In the context of industrial scenes, the scarcity of comprehensive point cloud segmentation datasets is notable. Yeo et al.'s (2020) pipework dataset remains the only available resource, and it is exclusively concentrated on pipes. This limitation underscores the absence of extensive datasets for other industrial components, which is critical for advancing point cloud segmentation and object detection in diverse industrial environments.

To address these gaps, this thesis makes several key contributions that advance the state of the art in point cloud analysis and industrial data management. Firstly, it introduces a novel network-driven object-finding methodology specifically designed for detecting individual objects within point clouds. This methodology overcomes the inefficiencies identified in prior research by incorporating a resampling technique that establishes hard constraints for the Iterative Closest Point (ICP) algorithm, thereby enhancing the accuracy of object candidate acceptance. Additionally, the integration of a geometric colorization network enables a double color-based ICP registration process, effectively pruning non-conforming candidates and improving overall detection precision.

Another significant contribution is the development of an advanced synthetic dataset generation framework tailored for industrial scenes. Recognizing the limited availability of comprehensive point cloud segmentation datasets beyond pipework, this research proposes a procedural generation approach that utilizes tube structures and a variety of 3D objects to create realistic and diverse industrial environments. This approach not only facilitates the rapid synthesis of large-scale datasets that closely mimic real-world industrial complexity but also ensures that the generated data incorporates realistic challenges such as noise, incomplete data, and precise point placement. By doing so, the synthetic datasets provide a robust foundation for training machine learning models, enhancing their ability to generalize across different industrial applications.

The thesis also addresses the issue of class imbalance and representation in point cloud data. Implementing a weight-balancing strategy that accounts for the prevalence, size, and point density of

objects in typical industrial environments ensures a balanced representation across diverse object types. This balanced approach enhances the model's capability to discern subtle differences between various objects, thereby improving the accuracy and robustness of point cloud segmentation neural networks. Furthermore, this research introduces a comprehensive validation strategy by training models on synthetic datasets and validating them on separate real-world datasets. This method demonstrates the generalizability of the proposed models, effectively bridging the gap between synthetic and actual industrial applications. By validating on distinct real-world datasets, the research underscores the practical applicability and reliability of the synthetic datasets in real industrial scenarios.

In addition to methodological advancements, the thesis contributes an open and accessible dataset creation framework, which mitigates the scarcity of publicly available industrial point cloud datasets. By developing an open methodology for generating diverse industrial datasets, this research ensures that researchers and practitioners have access to high-quality, realistic data necessary for further advancements in industrial point cloud analysis and machine learning applications.

Lastly, the emphasis on realism in synthetic data generation and enhanced dataset tailoring for industrial use cases ensures that the synthetic data accurately reflects the complexities of real-world industrial environments. This targeted approach addresses the underrepresentation of specific industrial challenges, such as complex machinery configurations and dynamic workflows, thereby providing more relevant and accurate data for training and evaluating machine learning models.

Collectively, these contributions address the identified knowledge gaps by providing innovative methodologies for specific object matching, advanced synthetic dataset generation, comprehensive class balancing, robust validation techniques, and accessible dataset creation frameworks. This research significantly enhances the efficiency and reliability of automated point cloud segmentation and object detection in industrial settings, thereby facilitating improved asset and facility management through the integration of state-of-the-art machine learning techniques.

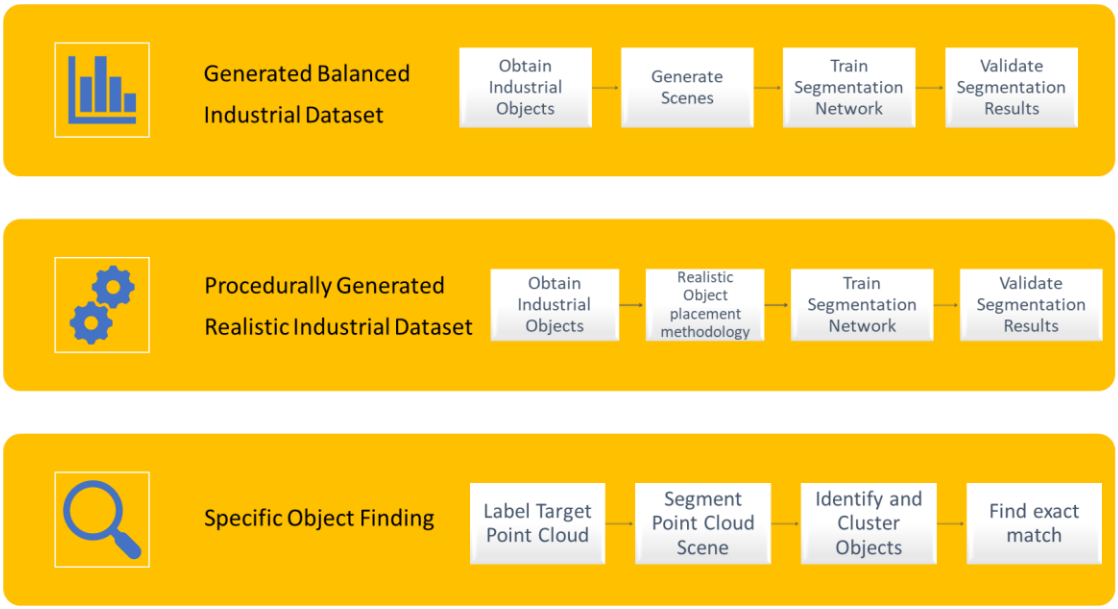


Figure 5. Thesis Methodologies

Chapter 3

Training Networks to Segment Industrial Scene Point Clouds: Dataset Generation Using TLS and Synthetic Data

Statement of Contributions

The research in this chapter was completed collaboratively with Gabriel Earle and Manolis Katsimpalis under the supervision of Dr. Carl Haas and Dr. Sriram Narasimhan at the University of Waterloo. Gabriel and Manolis were responsible for scanning the training facility from which the real-world scan was obtained. My primary contributions include the development of the random placement dataset generation methodology, implementing all Python code, training the network, and segmenting the real-world scan to generate the ground truth dataset for validation.

3.1 Introduction

In recent years, the automation and digitization of industrial processes have gained significant importance with the growth of Digital Twins and asset management. One pivotal aspect of this transformation is the efficient and accurate analysis of industrial scenes using point cloud data (Ma, 2020). Point clouds are data points in space, typically produced by 3D scanners. These data points represent the external surfaces of objects and are used in a myriad of applications, from creating 3D models of terrains to analyzing complex industrial environments. However, scanning or acquiring real-world point cloud data for industrial settings is often constrained by various factors, including privacy, security, and logistical issues. Furthermore, even when point cloud data is available, it may not be sufficient or representative for specific use cases, particularly in machine learning, where large amounts of data are often required for training robust models (Liu, 2019).

The synthesis of point cloud data through procedural generation provides an invaluable resource in generating scenes representative of industrial settings. Procedural generation refers to the creation of data algorithmically as opposed to manual creation. In the context of point cloud data, this can mean generating realistic 3D model environments. To effectively employ machine learning models, especially point cloud segmentation, it is vital to have access to extensive datasets that are representative of the variability and complexity of real-world industrial scenes.

However, synthetic objects alone are not sufficient. The neural networks' performance is closely linked to the quality and diversity of the data they are trained on. Good quality and diversity require generating synthetic scenes that can closely mimic the properties of real-world data. For instance, in an industrial setting, point clouds can be dense or sparse, clean or noisy, and have varying distributions of objects such as pipes, valves, and machinery. Therefore, a synthetic dataset for training neural networks must account for these variations to generalize to real-world data (Bosche, 2015).

Moreover, a critical aspect of training data, well-acknowledged in machine learning, is points cloud class balance (Griffiths, 2019). In segmentation tasks, handling imbalanced data, where some classes have substantially more examples than others, is imperative to avoid biasing the model towards the majority class. In the context of point cloud data of industrial scenes, this signifies the necessity of maintaining a balanced ratio between different objects. For instance, if the dataset contains many examples of pipes and insufficient examples of valves, the model needs to be appropriately calibrated to avoid compromised performance in recognizing valves in real-world data. Consequently, ensuring that synthetic datasets are realistic and well-balanced across different classes of objects found in industrial settings is a key priority. The generation of synthetic point cloud data is instrumental for the productive deployment of machine learning in industrial environments, and it is critical to ensure that such data is representative, realistic, point and class balanced, and accessible. Existing literature reveals a gap in the availability of diverse and complex datasets for industrial point cloud scenes, which are essential for training semantic segmentation networks.

This chapter aims to address the gap in existing 3D models with synthetic rendering techniques and class-aware sampling strategies to generate more representative datasets. It also introduces an algorithm tailored for optimizing the placement of irregular 3D objects and a real-world scan of an industrial scene as a real-world application example. Through these contributions, the chapter seeks to advance knowledge in dataset generation for industrial point clouds and foster the evolution of robust machine learning models.

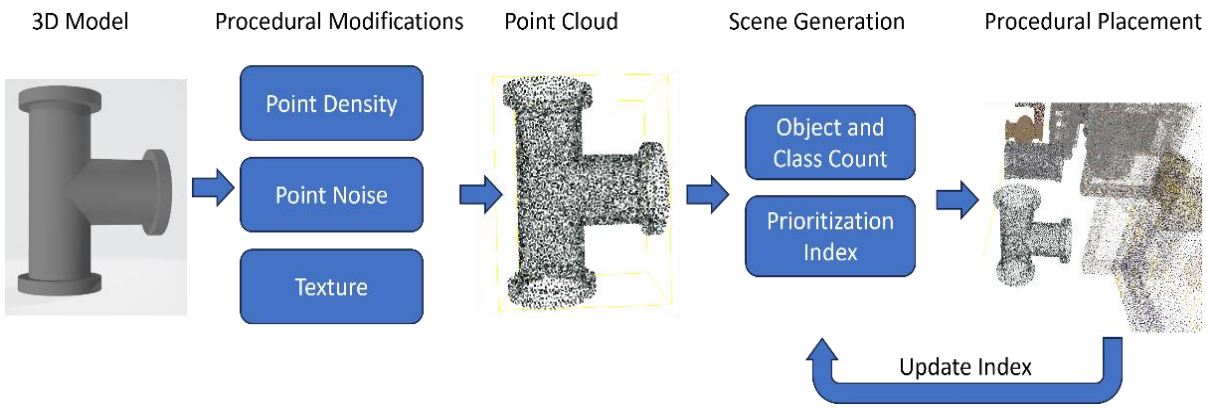


Figure 6. Methodology of Scene generation: 3D objects are transformed into point clouds through adjustments in point density, noise, and texture properties. These point clouds are then incorporated into the scene based on the balance of object count and prioritization index or weight over the dataset.

3.2 Methodology

The framework as seen in Figure 6, used in this work involves: i) Generating or importing the 3D objects found in industrial environments for which the network is being trained, ii) modifying and standardizing the 3D objects that will appear in the dataset, iii) point cloud sampling, and iv) generating a dataset using class balancing.

3.2.1 Object Generation

Our dataset generation methodology works with any GL Transmission Format Binary (GLB) file object. However, if no database of 3D objects is accessible, the object generation process can be used to build comprehensive industrial point cloud datasets. Stability AI Stable Diffusion 3 Medium (Esser, 2024) was employed. This tool generated detailed images from specific textual prompts, such as "short yellow railing in a construction scene with a white background and white floor, white ceiling" or "short one metal I-beam construction material with a white background and white floor, white ceiling." These prompts were designed to generate visual representations of key industrial objects. Once the images were generated, the most optimal views were manually selected based on clarity, completeness, and the angle of the object. Subsequently, InstantMesh (Xu, 2024) was used to process the selected images. InstantMesh uses the instant-mesh-large model variant and command-line scripts to remove the background of the image, generate six novel views imagined by Stability AI Stable Diffusion XL (Podell, 2023), and generate a mesh using the novel views created as seen in Figure 7. This approach provides flexibility, as any needed object can be generated by adjusting the text prompts, allowing the dataset to

cover a wide range of industrial requirements. This method leads to the creation of open-access objects maintaining accurate enough geometry and surface details.

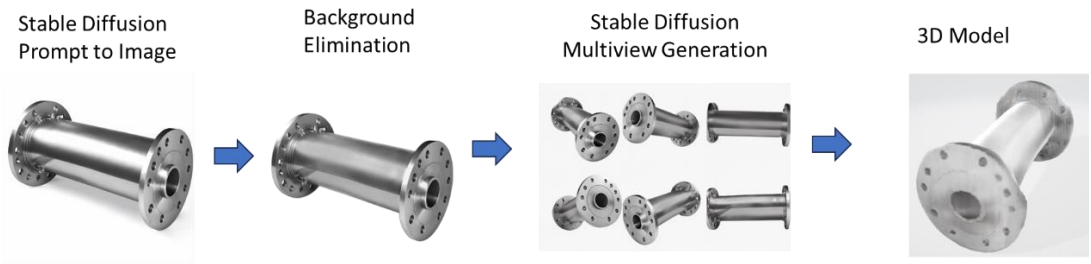


Figure 7. Object generation process entails: i) Stability AI Stable Diffusion 3 Medium creates high-quality images based on prompts; InstantMesh processing images by ii) removing backgrounds; iii) generating six views; and iv) constructing 3D models using instant-mesh-large model.

Additionally, custom scripts were developed to automate the creation of 3D models and their export to usable formats. One script generates random 3D splines and tube meshes. This process begins by defining random control points for a 3D spline, which are then interpolated to create a smooth curve. Tube, piping, and pipe spool meshes are created by extruding circular cross-sections along these splines. The resulting 3D models are exported as GLB files for further use. Another script facilitated the generation of varied pipe fittings and structural columns. For pipe fittings, the script generates components such as angles, crosses, and T-dividers by combining basic geometric shapes like cylinders and boxes. Structural columns, including circular, rectangular, and I-beam shapes, are also created with random dimensions. These models are further enhanced with random color variations using HSV color space adjustments to ensure visual diversity. The scripts automatically handle the exporting of these models as GLB files, ready for integration into the dataset. These automated processes ensure the efficient production of diverse and realistic models, enriching the dataset's applicability to a wide range of industrial scenarios.

3.2.2 Object Standardization

Integrating synthetic objects into the dataset generation process is a fundamental aspect of the methodology. This was achieved using 3D models, manually standardized for consistency using a purpose-built tool grounded in the open-source framework Open3D. This crucial step includes human verification of the objects and employs the tool's capabilities to modify 3D models in several ways to ensure uniformity and visual accuracy. An application for data preparation was created to support dataset development, offering

flexible 3D model manipulation, as shown in Figure 8. This application includes intuitive mouse-based interactions for rotation and translation, allowing thorough exploration of objects from diverse viewpoints. Additionally, the application enables the customization of material properties to promote dataset diversity and realism, supports switching between shaders such as Lit, Unlit, Normals, and Depth, and allows for direct color modification and scaling slider to modify the object size. These features help simulate different surface textures and lighting responses, preparing models for real-world conditions and improving material identification capabilities in downstream machine-learning applications. The application provides a detailed 3D inspection environment to support dataset curation. Users can adjust viewpoints, lighting conditions, and visual elements for a thorough examination of models. The ability to switch between shaders offers valuable insights into model geometry and surface characteristics.

3.2.3 Data Collection and Preparation

The dataset combines images generated using Stability AI Stable Diffusion and InstantMesh methodology with 3D modeling creations. Initially, 3D models showcasing common industrial objects such as I-Beams, wires, and structures were created. These models helped build synthetic scenes that mimic real-world situations. By blending synthetic data generated using Stability AI Stable Diffusion and InstantMesh methodology, a comprehensive set of objects for generating synthetic datasets was created. This method enhances the capabilities of machine learning models in industrial contexts and ensures practical and accurate simulations of industrial scenarios.

The objects in the dataset were consolidated into six classes: beam, circular hollow sections (CHS), column, door, equipment, and stairs. These classes represent the structural and architectural elements commonly found in industrial environments. The dataset contains a total of 259 point clouds distributed across these six classes. The most populous class is CHS with 116 point clouds, followed by column with 101 point clouds. The beam class contains 15 point clouds, equipment has 14, stairs have 9, and door has the fewest with 4 point clouds.

To verify that the dataset is capable of training a network to work on real scan data, a real-world industrial scene in Figure 9 was scanned using a FARO Focus Laser Scanner and manually segmented. This dataset can be used to validate that a network trained on the synthetic data can accurately identify objects such as I-beams, cables, chairs, pipes or circular hollow sections (CHS), column I-beams, desks, doors, and stairs.

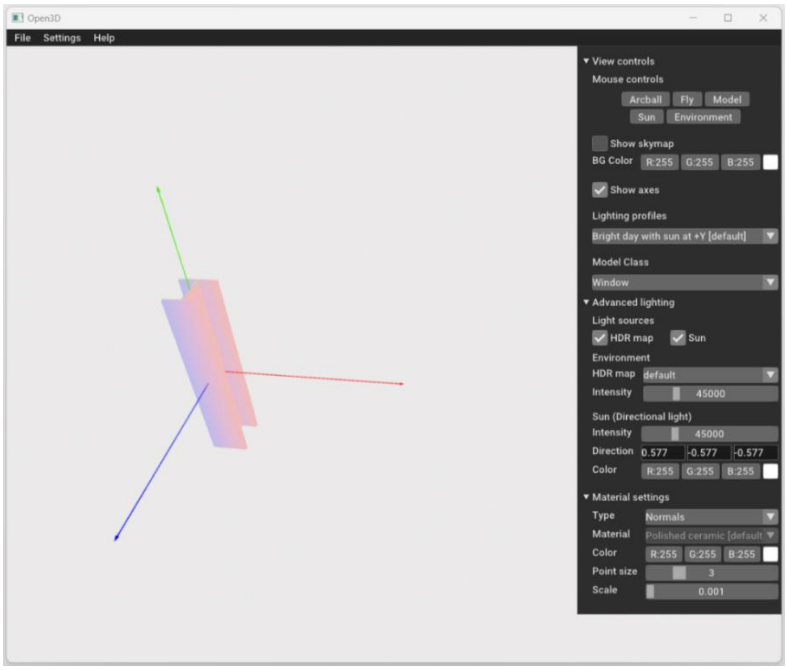


Figure 8. Object editor process entails: i) Simulate different lighting environments using predefined profiles, such as bright days or cloudy weather; ii) Modify point cloud rendering settings like scale, light intensity, and direction; iii) add class label of the object and save as GLB.

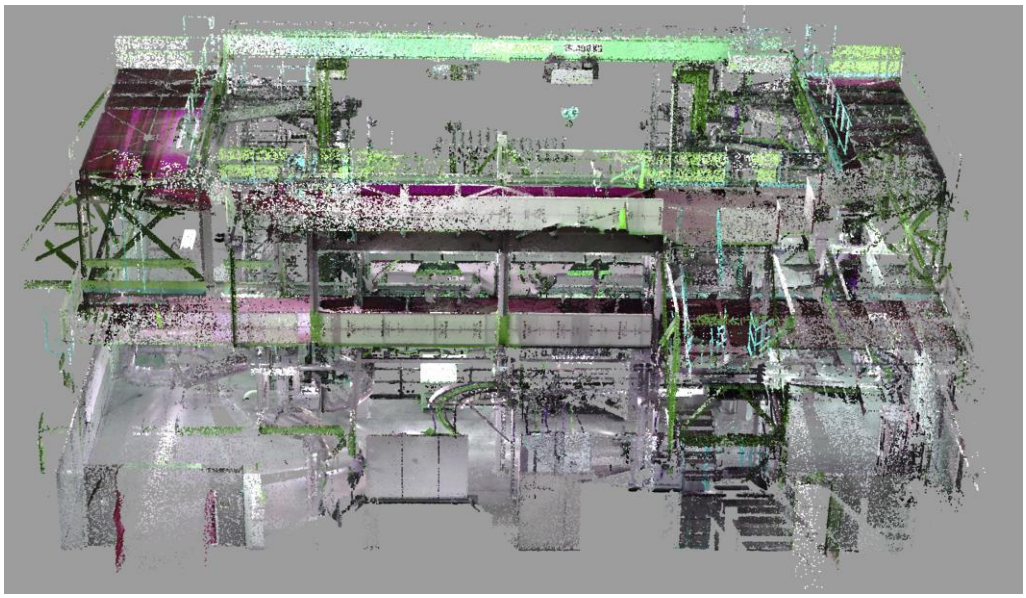


Figure 9. Real-world scan using FARO Focus Scanner that is used for validation.

3.2.4 Point Cloud Generation

Integrating point cloud sampling and noise introduction is crucial for enhancing machine learning models' realism and generalization capabilities in industrial contexts. This method prepares the models to handle incomplete or noisy data commonly encountered in real-world applications, thereby improving their reliability and effectiveness in various industrial tasks. This process involves sampling points from a mesh using techniques such as Poisson disk sampling, followed by the introduction of Gaussian noise to these sampled points. By systematically adding this noise, the point clouds become more representative of real-world conditions where data imperfections are common.

This approach also mimics the inherent inaccuracies of 3D scanners, which vary depending on the distance from the scanned objects. Different scanners exhibit distinct error ranges based on their operational principles and the distance of the object being scanned. For instance, scans taken at greater distances generally have higher noise levels due to the increased signal-to-noise ratio. Simulating these conditions during training ensures that the models are better equipped to handle the variability and imperfections typical of real-world data. This careful consideration of scanning errors enhances the models' robustness, making them more adaptable and accurate in practical industrial applications.

Generating point clouds to represent scenes is a crucial step in the methodology. Converting the 3D dataset into point cloud form involves specific techniques to ensure accuracy and realism in the generated data. The transformation from 3D models to point clouds requires techniques such as Dart Throwing and Poisson-Disk sampling. These methods distribute points within the 3D space, recreating the spatial characteristics of objects in the scene and capturing the essential features of the objects and structures found in industrial environments.

A significant consideration in this process is incorporating noise and sparsity factors. In real-world scenarios, point clouds are affected by sensor inaccuracies and environmental disturbances. To reflect this reality, levels of noise and sparsity are introduced into the synthetic point clouds, adding an element of authenticity and ensuring that the synthetic data closely emulates the imperfections inherent in actual industrial point cloud scans.

The script called GLB2PCD loads a 3D mesh model from a file and converts it into a point cloud representation. Poisson disk sampling ensures a uniform distribution of points across the mesh surface, accurately recreating the object's spatial features. Controlled levels of noise and sparsity are introduced to mimic real-world inaccuracies. After generating the point cloud, it is saved for further analysis, with

parameters related to noise, density, and distribution fine-tuned to match the characteristics of real-world industrial point clouds. This approach ensures that the point cloud not only represents the physical form of objects with high fidelity but also includes imperfections characteristic of real-world environments, making the synthetic point clouds valuable for various applications.

3.2.5 Hidden Point Removal

Hidden point removal (HPR) is a crucial technique in point cloud processing that simulates laser scan point cloud occlusion, enhancing the realism of 3D data representations. This method identifies and removes points obscured from a specific viewpoint, effectively mimicking how an object or scene would appear when observed from a particular position. In our implementation, we utilize this technique to generate varied perspectives of the same point cloud, enriching our dataset with diverse viewpoints.

Our python script called LidarSimulation.py implements hidden point removal using Open3D's built-in functionality. The process begins by calculating the center and diameter of the input point cloud, which are used to define potential virtual scanner positions. We establish four diagonal scan angles relative to the point cloud's center, positioned at a distance equal to the cloud's diameter. These virtual scanner positions represent different viewpoints from which the point cloud might be observed in a real-world scenario.

The hidden point removal operation is then performed using Open3D's hidden point removal function, as shown in Figure 10. This function inputs the point cloud, a selected scan position, and a radius parameter. The radius is 100 times the point cloud's diameter to ensure comprehensive coverage. The method returns a set of visible points from the chosen scanner's point of view. Our implementation randomly selects predefined scanner positions for each point cloud, applies the hidden point removal, and replaces the original point cloud with this new, partially occluded version. This approach introduces variability in the dataset, simulating different viewing angles and occlusion effects that might be encountered in real-world scanning scenarios.

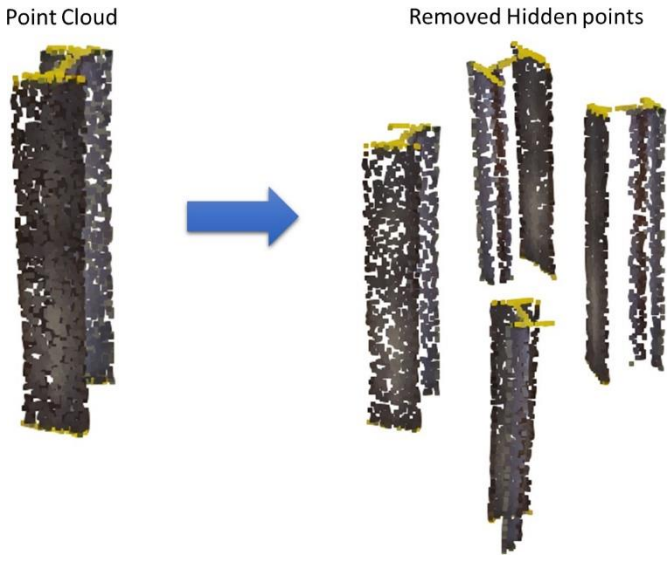


Figure 10. Hidden point removal eliminates points occluded from the virtual scanner's point of view, leaving only the visible surface points in the scene. The hidden point removal on the I-Beam uses the scanner angle and a complete point cloud (left) to generate the occlusions that can be seen from different angles (right).

3.2.6 Weight-Balanced Class Dataset Generation

Addressing class imbalance is crucial for training accurate and robust models in procedural generation, particularly in datasets consisting of point clouds. Class imbalance, when left untreated, can lead to models that are biased towards classes with a higher frequency of objects, diminishing the performance in recognizing and classifying less-represented classes. To mitigate this, a refined method for calculating class weights is proposed, which incorporates the density of points within each object and the total number of objects within each class. This approach ensures a balanced representation of all classes during model training.

The calculation of the weight for each object is governed by the following equation:

$$w_{ij} = \frac{P_{ij}}{P_{\text{class}_i}} \times \frac{N_{\text{class}_i}}{\sum_{i=1}^{C} N_{\text{class}_i}} \tag{7}$$

In equation 7, w_{ij} denotes the weight assigned to object j belonging to class i. The term P_{ij} represents the number of points associated with object j in class i, while $P_{\text{class}_i} = \sum_{j=1}^{N_{\text{class}_i}} P_{ij}$ corresponds to the total number

of points for all objects in class i. The parameter N_{class_i} is the total number of objects in class i, and $\sum_{i=1}^{C} N_{\text{class}_i}$ is the total number of objects across all classes, where C is the number of distinct classes in the dataset.

This method ensures that the weight of an object is proportional to its relative contribution to the total points in its class and inversely proportional to the number of objects in its class relative to the total dataset. Objects with fewer points, or objects belonging to smaller classes, are thus assigned higher weights. This approach promotes the representation of classes that might otherwise be under-represented in the dataset, especially when such classes contain objects with a relatively small number of points.

To ensure that the sum of all weights equals one, facilitating a balanced training process across the entire dataset, a normalized weight $\widehat{w}_{ij}$ is calculated as follows:

$$\widehat{w}_{ij} = \frac{w_{ij}}{\sum_{i=1}^{C} \sum_{j=1}^{N_{\text{class}_i}} w_{ij}} \tag{8}$$

The normalization step in equation 8 ensures that the sum of the weights across all objects remains consistent, regardless of class sizes or object densities, allowing the model to be trained on a dataset where the influence of each class is appropriately balanced.

In essence, this method assigns higher weights to objects with fewer points or to those belonging to classes with fewer objects, thus promoting their selection during the weight distribution phase. This strategy directly addresses the issue of class imbalance by ensuring that models trained on such datasets are exposed to a more representative sample of all classes, particularly the less frequent ones. Consequently, the trained models are better equipped to recognize and classify objects of varying complexities and sizes, leading to improved generalization and performance across diverse environments.

3.2.7 Dataset

In this section, the structure of the datasets, the types of datasets generated, and the corresponding output formats are discussed. This includes a detailed explanation of how the synthetic dataset is organized, the objects it contains, and the real-world dataset used for validation, which was collected using a laser scanner. Additionally, the section outlines how both datasets were split into training, validation, and testing sets to ensure a thorough evaluation of the models during experimentation.

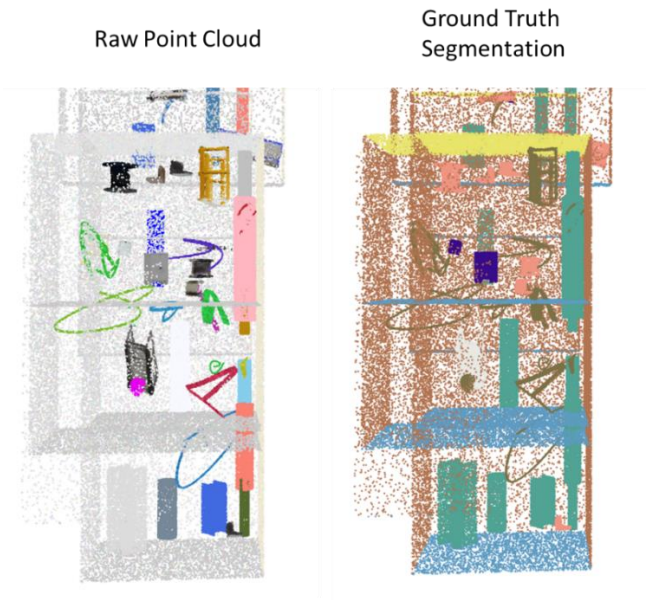


Figure 11. The image on the left presents a raw point cloud *Room* of the synthetically generated dataset, while the image on the right shows the corresponding segmentation.

3.2.7.1 Dataset Structure

The dataset used in this study is organized hierarchically to represent large indoor spaces, with particular emphasis on capturing the detailed structure of *Areas* and *Rooms*. Each *Area* represents a distinct section of a facility, such as a warehouse, factory, or office building. Within each *Area*, there are multiple *Rooms*, each representing a specific enclosed space as seen in Figure 11, such as a storage room, a pump room, or a control room. The data for each *Room* includes both a full room-level point cloud and individual object-level point clouds, allowing for a multi-scale analysis of the environment.

At the highest level, the dataset is divided into *Areas*, which can be thought of as disjoint sections of a building or environment. These *Areas* contain one or more *Rooms*, each of which represents a specific environment with distinct spatial and object characteristics. For example, in an industrial setting, *Area* 1 might represent "Factory Section A," while Area 2 might represent "Warehouse Section B." Each *Room* within these *Areas* contains objects typically found in such environments—pumps, valves, machinery, storage shelves, and other essential equipment.

Within each *Room*, the dataset stores two main types of data: the overall point cloud representing the entire room and separate point clouds for each individual object within the room. This is done to facilitate detailed object-level analysis while still preserving the context of the room as a whole. The room-level point cloud

files are stored in .txt format, with each file containing the 3D coordinates (X, Y, Z) and the corresponding color values (R, G, B) of each point in the scene. The XYZ values represent the spatial location of each point in 3D space, while the RGB values provide color information, making it possible to not only analyze the geometry of the room and objects but also their visual appearance.

Each *Room* in the dataset has a dedicated folder. Inside this folder, there is a .txt file containing the full point cloud data for the room (e.g., room_1.txt), as well as an Annotations subfolder where individual object point clouds are stored in separate .txt files (e.g., object_1.txt, object_2.txt). For instance, a *Room* named "Pump Room" might have a room-level file named room_1.txt and several object files such as valve_1.txt for a valve, pipe_1.txt for a pipe, and so on. Each of these files follows the XYZRGB format. In this format, the first three columns correspond to the spatial coordinates (X, Y, Z) of the point, while the next three columns represent the RGB color values, ranging from 0 to 255. This format allows for a detailed and accurate representation of both the geometric structure and the appearance of objects in the scene.

The dataset also includes information on the alignment of each *Room* within an *Area*. Each *Area* has an associated alignment file (Area_x_alignmentAngle.txt), which specifies the global rotation angle applied to the Rooms in that Area. The alignment angles are provided to ensure that the data from different *Rooms* can be properly oriented for consistent analysis across the dataset. The alignment angles are given in increments of 90 degrees, and each *Room* can be rotated by 0, 90, 180, or 270 degrees to ensure proper alignment. The alignment file begins with two header lines that describe the format, followed by the room names and their corresponding alignment angles. We designated specific real-world data as validation to assess the model’s performance.

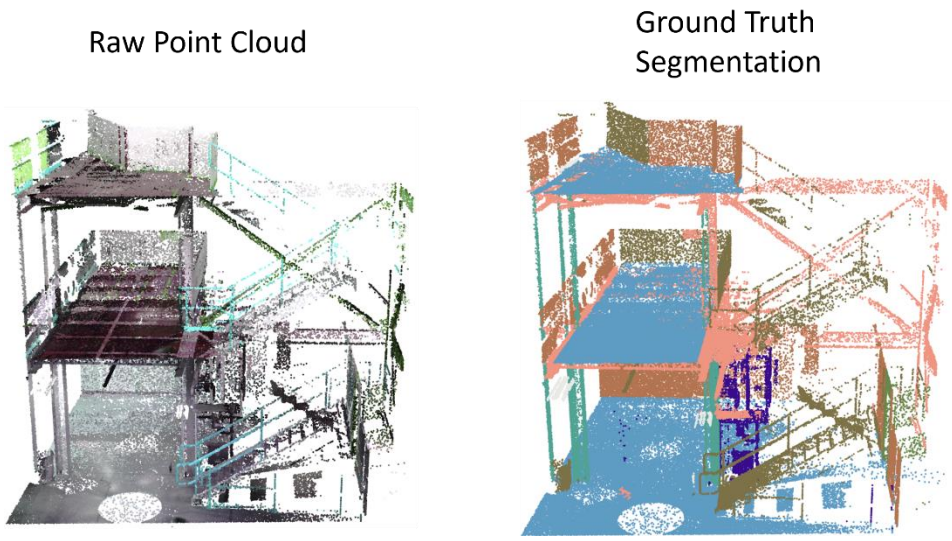


Figure 12. The image on the left presents a section of real-world laser scan dataset, while the image on the right shows the ground truth of the manual segmentation of the real-world point cloud corresponding to the section on the left.

3.3 Experiments

The following experiments were conducted to assess the impact of various dataset parameters—including class balance and hidden point removal (HPR)—on the performance of the trained network tested on a real-world point cloud scan, subdivided into eight sections. The goal was to evaluate how these factors influence segmentation accuracy and point classification performance in point cloud data. Effective object identification depends on the quality of point cloud segmentation.

3.3.1 Experimentation Implementation

The experimental evaluation was conducted on a desktop with 125.6 GiB of memory, an AMD® Ryzen Threadripper 3960X 24-core processor, NVIDIA RTX 3090 graphics, and a disk capacity of 21.0 TB. Operating on Ubuntu 20.04.6 LTS with GNOME version 3.36.8 and utilizing the X11 windowing system, this setup provided a robust platform for the dataset validation process. Key hyperparameters of the Superpoint Transformer (SPT) model include a voxel size of 0.03 for down sampling, a batch size of 1 for data loading, K-Nearest Neighbors (KNN) set to 45 for graph construction, and an anisotropic scaling factor of 0.2 for data augmentation.

3.3.2 Real and Synthetic Datasets

In this study, two distinct datasets were used: a real-world dataset collected through laser scanning and a synthetic dataset generated procedurally. Both datasets played crucial roles in training, testing, and validating the models developed for 3D scene understanding and object detection in industrial environments.

The real-world dataset comprises a comprehensive scan of an industrial scene. This dataset consists of 360 individual point clouds that capture various components, structures, and perspectives within the industrial environment. The point clouds span across two floors of the industrial space and represent a wide variety of objects and structures, including essential elements such as beams, cables, chairs, columns, desks, doors, electric boxes, floors, and stairs. The dataset is carefully segmented and labeled, making it possible to analyze both individual objects and the overall scene structure in detail.

The real-world dataset contains the following point clouds: 87 beams, 26 cables, 8 chairs, 53 circular hollow sections (CHS), 49 instances of clutter, 33 columns, 4 desks, 5 doors, 5 electric boxes, 6 floor sections, 23 railings, 4 stairs, 14 equipment units, and 4 wall sections. These point clouds reflect the diversity and complexity typically found in real-world industrial environments, where both structural elements and miscellaneous objects coexist in dynamic, often cluttered, settings. This dataset serves as an invaluable resource for validating the models developed using the synthetic data, providing a real-world benchmark to evaluate the accuracy and robustness of the models in practical applications.

The second dataset used in this study is a procedurally generated synthetic dataset created using custom-designed code. This dataset was specifically generated to simulate industrial environments, allowing for complete control over object placement, density, and variability within each scene. The synthetic dataset mirrors the complexity of real-world industrial environments but with greater flexibility to create different scenarios and conditions. The dataset is structured across 23 *Areas*, each containing 45 *Rooms*, with a maximum of 30 objects per *Room*. This design enables the generation of a wide variety of industrial scenes, with each *Room* containing a unique arrangement of objects such as beams, columns, ceilings, doors, and equipment. The dataset includes 4500 beams, 900 ceilings, 4491 CHS, 4512 columns, 4499 doors, 4499 pieces of equipment, 2241 floors, 900 rooms, 4499 stairs, and 3600 walls for a total of 20GB.

To ensure a robust evaluation of the Superpoint Transformer (SPT) point cloud segmentation model, the dataset was split into 14 *Areas* for training, with one synthetic *Area* (*Area* 10) initially designated for validation, and the remaining 5 *Areas* reserved for final testing. However, the real-world dataset, which corresponds to one *Area* in terms of scale and complexity, was swapped in place of synthetic *Area* 10 for validation and visualization purposes. This substitution allowed the real-world dataset to serve as a key resource for evaluating the model's generalization capabilities in real-world industrial environments, providing a direct benchmark against practical applications. The specifics of the experiments, including the training and validation procedures used to assess the model’s performance under different conditions, are further elaborated in Section 5. Finally, to do a fine-grain analysis, the real-world dataset was split into eight scenes to analyze each object in different sections of the scan.

3.4 Results

The primary goal of our experiments is to assess the effectiveness of the synthetic dataset we generated for training models in 3D point cloud segmentation, particularly in industrial environments. The two main experiments were conducted to analyze the impact of dataset balancing and hidden point removal (HPR) on model performance. The

model's performance was evaluated using several metrics, including point segmentation mean intersection over union (mIoU), which is defined by the overlap between the predicted segmentation and the ground truth, divided by the area covered by the union of the two. Mean accuracy (mAcc) provides a comprehensive analysis of classification accuracy.

To provide context, difficult datasets like the NYU Depth Dataset V2 (Silberman, 2012) for point cloud segmentation may start with a low mAcc of 27.83% and mIoU of 19.46% (Lahoud, 2020) using early machine learning models. However, state-of-the-art models like GeminiFusion (Jia, 2024) have achieved a mAcc of 69.9% and mIoU of 56.8% on this dataset. In contrast, an easier dataset like the widely recognized S3DIS (Armeni, 2017) dataset may start from 66% mAcc (Wu, 2024) and reach up to 92% mAcc (Qian, 2022) in different areas. Unfortunately, no class analysis has been performed on the S3DIS dataset to offer a direct comparison. Our approach also differs by training on a synthetic dataset and testing on a separate real-world validation dataset, highlighting the generalizability of the method.

3.4.1 Dataset Balancing

Class balance significantly affected the model's performance. A balanced dataset consistently improved the model's ability to generalize across different object categories, particularly for underrepresented object types. For example, the accuracy for "Columns" increased from 21.33% in the unbalanced dataset to 66.67% in the balanced dataset (Table 3). Table 3 shows that the Balanced Dataset with HPR achieved the best results, with a Mean Accuracy of 57.02% and a Mean IoU of 39.27%. These results confirm the effectiveness of using balanced datasets combined with HPR to improve model performance. The model also performed particularly well in segmenting larger, flat surfaces like "Walls," achieving an accuracy of 92.50% in the balanced dataset with HPR.

These findings indicate that dataset balancing allows the model to generalize well across different object categories. It is expected that using these results to vote on cluster classes would result in much higher levels of object identification because the errors are distributed across all the other classes. Further research is required to validate this hypothesis.

Table 3. Performance metrics for evaluating point cloud generation and classification accuracy

Metric	Unbalanced Dataset	Unbalanced Dataset with (HPR)	Balanced Dataset	Balanced Dataset with HPR
Mean Accuracy (mAcc)	45.71 %	48.28%	55.10%	57.02%
Mean IoU (mIoU)	37.95%	33.61%	38.14%	39.27%

3.4.2 Hidden Point Removal

The impact of hidden point removal (HPR) further improved segmentation accuracy by simulating occlusions and providing varied perspectives of the point cloud data. As shown in Table 3, HPR enhances the realism of the dataset by introducing different occlusion effects, which helps the model to generalize better for larger classes. The use of HPR resulted in higher accuracy scores and lower loss values across most categories. In the balanced dataset with HPR, the accuracy for Beams increased from 45.38% without HPR to 47.25% with HPR; however, for unbalanced datasets, HPR decreased the accuracy for smaller objects as seen in Table 4. For "Door" the accuracy increased from 50.13% to 63.13%. This demonstrates that HPR helps the model generalize better by exposing it to more realistic scenarios where parts of objects are hidden from view. While larger classes like "Clutter" benefit from HPR, with accuracy increasing from 58.75% to 75.75%, smaller classes such as "Beams" and "CHS/Pipes" experience a decline in accuracy. Notably, the "Column" class improved slightly in unbalanced datasets, highlighting that certain small classes can still benefit depending on their representation. In balanced datasets, HPR also caused a drop in accuracy for some classes, like "Clutter" and "Column," suggesting that the technique may introduce too much complexity in certain cases.

Table 4. Mean accuracies (mAcc) for the objects in point cloud balance and HPR datasets.

Metric	Unbalanced Dataset	Unbalanced Dataset w/ (HPR)	Balanced Dataset	Balanced Dataset with HPR
Beam	25.50%	22.75%	45.38%	47.25%
CHS/Pipes	25.43%	21.88%	41.00%	32.50%
Clutter	58.75%	75.75%	81.50%	74.25%
Column	21.33%	23.33%	66.67%	44.00%
Door	69.00%	71.88%	50.13%	63.13%
Floor	28.50%	32.88%	32.00%	49.38%
Wall	78.00%	92.50%	89.17%	92.50%

The testing phase, as shown in Table 4, details the performance metrics across seven distinct object categories: Beams, Clutter, Columns, Doors, Floors, Pipes, and Walls. These categories were chosen to represent a broad range of industrial objects, facilitating an assessment of the model's segmentation precision and generalization capabilities.

The real-life dataset used in these experiments was divided into eight subsections, and Table 5 presents the point segmentation accuracy for different object classes across these subsections using the Balanced Dataset with HPR. These results represent the segmentation accuracy of individual points within each class for each scene, demonstrating how well the model performs across different parts of the dataset. It is important to note that these values reflect segmentation accuracy, not object identification accuracy.

Table 5. Point segmentation accuracy for different object classes across the eight real-life dataset subsections using the Balanced Dataset with HPR. The spaces mean that those objects were not in that scene.

Accuracy of Balanced Dataset with HPR per Scene							
Scenes	**Beams**	**Clutter**	**Columns**	**Doors**	**Floors**	**CHS/Pipes**	**Walls**
Scene 1	51%	60%	85%	-	65%	20%	98%
Scene 2	55%	33%	78%	-	77%	20%	-
Scene 3	64%	80%	72%	36%	69%	70%	95%
Scene 4	60%	16%	75%	-	51%	64%	100%
Scene 5	14%	30%	63%	-	42%	70%	-
Scene 6	38%	15%	67%	76%	77%	54%	84%
Scene 7	50%	15%	74%	-	57%	57%	90%
Scene 8	46%	11%	80%	20%	67%	40%	88%
Average	47%	33%	74%	44%	63%	49%	93%

3.4.3 Analysis

The results in Table 5 demonstrate significant variation in segmentation accuracy across different object classes and scenes. On average, the model achieved the highest segmentation accuracy for "Walls" (93%) and "Columns" (74%), indicating strong performance in identifying large, structured objects with consistent geometric features. Larger objects like walls and columns present distinct geometric characteristics that the model can learn effectively, leading to higher segmentation accuracy. This highlights the importance of model granularity, as these objects can often be treated as cohesive units, simplifying the segmentation process. Conversely, smaller or more complex objects such as "CHS/Pipes" and "Clutter" had average accuracies of 49% and 33%, respectively. These objects often have intricate shapes and may be occluded or intertwined with other structures, making them more challenging to segment accurately. The granularity of these smaller objects, particularly pipes or clutter with varying dimensions and connections, requires careful definition. Modeling them as separate components versus unified assemblies impacts segmentation performance and dataset complexity. The difference in performance suggests that while the model excels at recognizing large, well-defined objects, additional strategies may be needed to improve the segmentation of smaller or more complex classes.

Differences in accuracy across scenes are also evident. Scene 3 achieved the highest average segmentation accuracy at 69%, with particularly high accuracies for "Walls" (95%), "Clutter" (80%), and

"Beams" (64%). This indicates that Scene 3 had features more conducive to accurate segmentation, possibly due to clearer object boundaries or less occlusion. Clearer delineation of object granularity in Scene 3 likely contributed to this higher performance, as unambiguous segmentation benefits from well-defined geometric distinctions. In contrast, Scene 7 had a lower average accuracy of 44%. This scene features an elevated floor with numerous beams underneath shown in Figure 13, which may have caused confusion between the "Beams" and "Walls" classes due to overlapping structures and similar geometric appearances. The complex geometry and occlusions in this scene present challenges for the model, highlighting areas where the segmentation process can be further refined.

Overall, the experiments demonstrate that dataset balancing and the application of Hidden Point Removal significantly enhance any machine learning point cloud segmentation's accuracy. The Balanced Dataset with HPR consistently outperformed other configurations across multiple metrics and the difference is visible in Figure 14. The model exhibits strong capabilities in segmenting large and well-defined objects, while smaller or more complex objects present more of a challenge. Incorporating granularity considerations, such as modeling smaller objects independently or grouping them as assemblies, can optimize dataset representation. By simulating occlusions and providing varied perspectives, HPR helps the model to generalize better and improves segmentation performance. Understanding the specific characteristics of each object, such as the elevated floor and overlapping beams in scene seven, can inform strategies to address segmentation difficulties and improve model robustness across varied environments in future research.

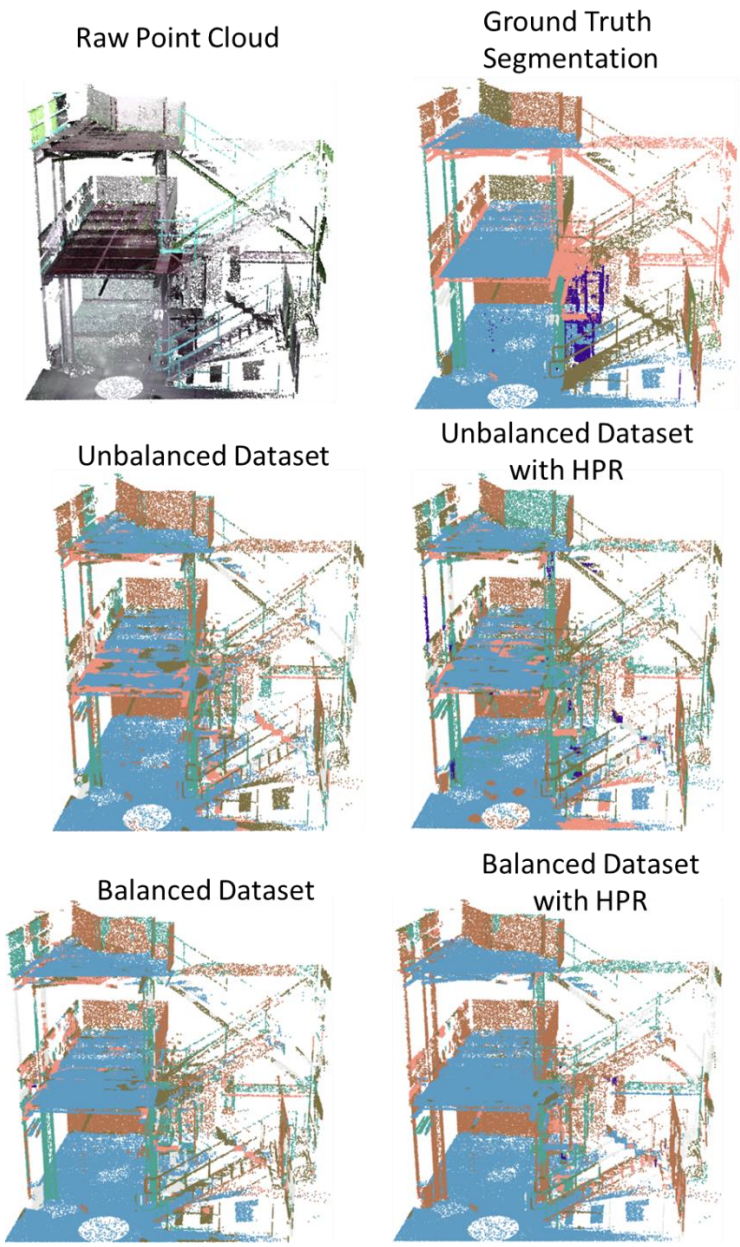


Figure 13. Application of the segmentation network on scene eight of the scan of an industrial point cloud using: i) Raw Scanned Point Cloud, ii) Segmentation's Ground Truth, iii) Unbalanced Dataset, iv) Unbalanced Dataset with HPR, v) Balanced Dataset, vi) Balanced Dataset with HPR.

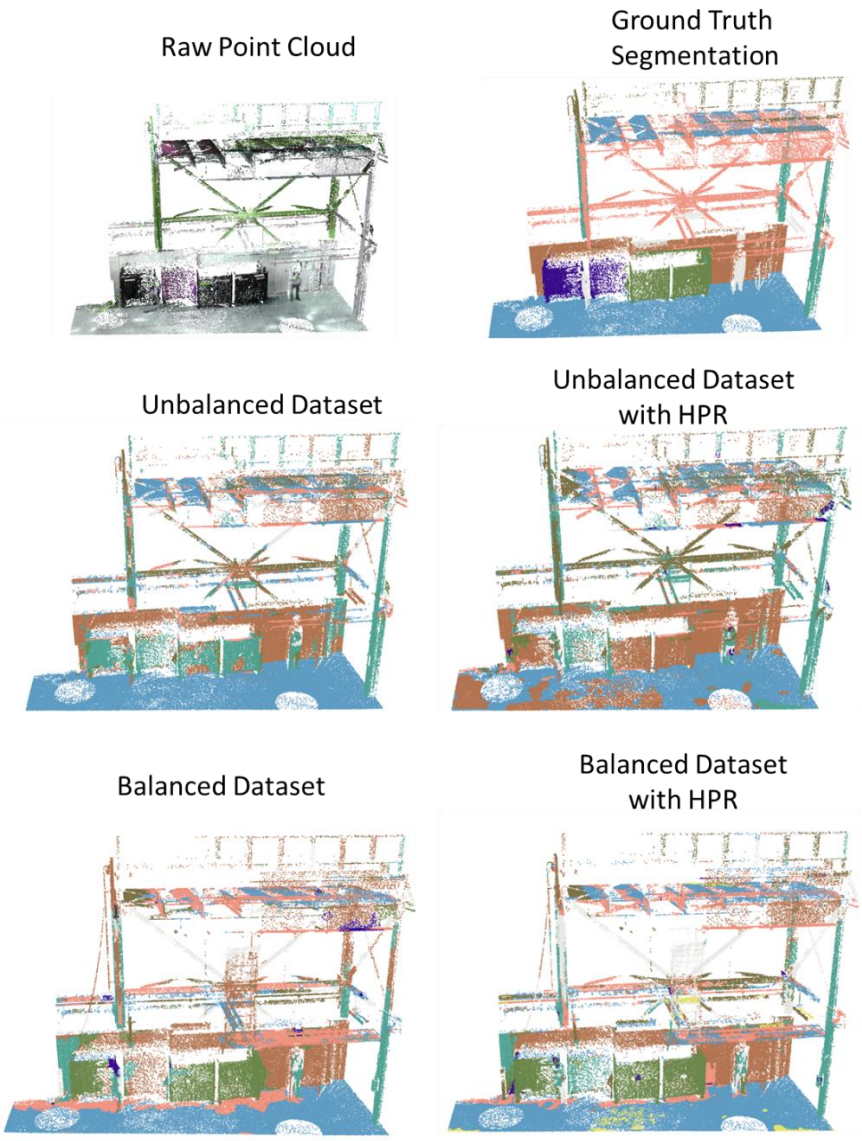


Figure 14. Application of the segmentation network on scene five of the scan, showing the entrance section of an industrial point cloud using: i) Raw Scanned Point Cloud, ii) Segmentation's Ground Truth, iii) Unbalanced Dataset, iv) Unbalanced Dataset with HPR, v) Balanced Dataset, vi) Balanced Dataset with HPR.

Chapter 4
Realistic Procedural Generation of Point Cloud Datasets for Training Segmentation Networks in Industrial Environments

Statement of Contributions

The research in this chapter was completed collaboratively with Gabriel Earle and Manolis Katsimpalis under the supervision of Dr. Carl Haas and Dr. Sriram Narasimhan at the University of Waterloo. Gabriel and Manolis were responsible for scanning the training facility from which the real-world scan was obtained. My primary contributions include the development of the procedural generation methodology, implementing all the Python code, training the network, and segmenting the real-world scan to generate ground truth values for validation. Harsh Patel contributed with a subsection of the python implementation of the object spawning and the CHS object interaction with the scene.

4.1 Introduction

The construction and manufacturing industries have seen significant technological advancements in the integration of software, machine learning, and robotics that collect and analyze data. In return there is improvement in precision, efficiency, and collaboration(Javaid, 2021). A key technology driving this progress is the use of point cloud data in Scan-to-BIM (Building Information Modeling) processes (Wang, 2019). Point cloud data, captured via 3D laser scanning, provides an accurate digital representation of existing structures. This technology is essential for converting physical environments into detailed digital models, crucial for renovation, quality control, and lifecycle management of buildings (Qu, 2015).

Traditional methods of capturing as-built conditions are labor-intensive, time-consuming, and often inaccurate. In contrast, point cloud data offers precise spatial information, enabling the creation of detailed as-built models (Qu, 2015). These models support various applications, including decommission or renovation planning, structural analysis, and facility management. Accurate point cloud data ensures that BIM models reflect the true state of the building, aiding informed decision-making throughout its lifecycle(Barrile, 2019). Nevertheless, using point cloud data presents challenges in data processing, noise reduction, and completeness.

One of the main methods to include important information in point cloud data or enrich the point cloud is machine learning point cloud segmentation. Point cloud segmentation involves adding values to the point cloud data so that it can be divided into meaningful subsets, which can then be analyzed and utilized for various applications such as object recognition, scene understanding, and 3D modeling(Wang, 2019). Recent advancements in machine learning and generative models show promise in addressing these challenges, enabling the creation of high-quality point cloud segmentations (Lopez, 2023). Ensuring the accuracy and reliability on point cloud data requires advanced data processing techniques and validation frameworks (Wang, 2019). However, a significant challenge in the industrial field is the lack of public datasets that can accurately represent realistic scenarios.

Generating objects and procedural environments is essential for creating realistic point cloud datasets. Achieving realism includes addressing various factors that replicate the imperfections, variability, and spatial arrangements found in real-world scan data (Triess, 2021). Realism in this context includes geometric accuracy, noise, incomplete data, color variations, and the precise placement of objects, all of which are important for developing robust and adaptable machine-learning models. Many existing datasets are proprietary or fail to capture the complexity of real-world industrial environments(Wang, 2019). This limitation hinders the development and testing of segmentation algorithms, as the models trained on these datasets may not generalize well to real-world applications. Therefore, there is a pressing need for datasets that encompass the diversity and intricacies of unique scenarios to improve the robustness and reliability of segmentation techniques.

This chapter aims to explore the requirements for realism in point cloud generation for machine learning segmentation networks, test whether realism in the dataset is truly necessary for achieving efficient point cloud segmentation on real-world data and stablish a methodology for generating procedurally generated realistic point cloud and object placement datasets. Afterwards, by identifying the key factors that influence the performance of segmentation networks, it may be possible to develop optimized training strategies and synthetic datasets that adequately mimic the essential features of real-world data. This approach could potentially reduce the dependency on proprietary and incomplete datasets, making the development of segmentation models more efficient and accessible.

4.2 Methodology

This section presents the methodology employed for generating realistic synthetic datasets, leveraging a graph-based structure for object generation combined with random placement and orientation techniques.

This approach ensures the creation of complex, realistic scenes essential for training robust machine learning models in object detection, segmentation, and recognition tasks. (i) Graph-based hierarchy for object spawning; (ii) Spatial Boundaries Spawns (iii) Object Transformation; and (iv) Point cloud Generation and Validation features, and to find its location, orientation, and dimensions. These components are described in this section.

4.2.1 Graph-Based Structure for Object Spawning

The foundation of our methodology is a hierarchical framework for object spawning, structured around the concept of a scene graph. This process, which underpins dataset generation, begins with the creation of a scene graph that serves as the organizational blueprint for object placement and spatial relationships within an industrial environment. As illustrated in Figure 15, the scene graph is a hierarchical, graph-based model that encodes the interrelations between objects in a scene. This modeling approach is widely employed in computer graphics and machine learning due to its capacity to capture complex spatial and functional dependencies among objects. In this framework, objects and their potential interactions are represented as nodes and edges within the graph. Each node corresponds to a specific object type, while edges define the spatial and probabilistic relationships between objects. Object spawning is governed by a set of predefined rules, wherein new objects are generated based on the characteristics of existing ones. For instance, a desk may spawn a chair within a certain proximity. Moreover, objects that are spawned may further trigger the spawning of other objects, establishing a nested hierarchy of dependencies and interactions.

In the context of industrial scene modeling, this approach offers significant advantages by accurately reflecting real-world constraints. For example, industrial machinery must be placed in precise spatial relationships relative to walls, floors, and other equipment. Similarly, components such as pipes and valves must adhere to specific spatial configurations, such as alignment along walls or routing near structural supports. The scene graph effectively encodes these spatial dependencies by representing object attributes, such as size, orientation, and positional constraints, and by defining the rules that govern their placement relative to other objects. The hierarchy of object spawning is formalized through a set of spatial constraints that dictate the conditions under which objects can be placed. For instance, walls may spawn from floors, desks may spawn from walls, and chairs may spawn from desks. These relationships are explicitly defined through a structured set of conditions, with each object type associated with specific spawning rules, as depicted in Figure 15.

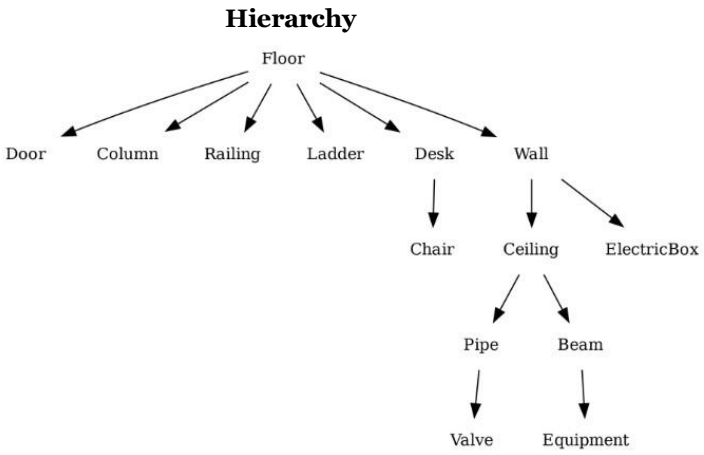


Figure 15. Overview of object hierarchy on spatial spawning.

4.2.2 Integration with Synthetic Dataset Generation

One of the most significant advantages of procedural generation is the ability to introduce variability into the placement and orientation of objects, a feature that is essential for creating a diverse and robust dataset. In real-world industrial environments, object placement can vary significantly depending on factors such as available space, safety regulations, and operational workflows. To reflect this variability, the procedural generation process incorporates randomization in the placement of certain objects, while still adhering to the overall constraints defined by the object spawning probabilistic boundaries.

This variability in object placement is critical for training machine learning models that need to generalize across a wide range of industrial settings. By exposing models to multiple configurations of the same objects, the dataset ensures that ML models learn to recognize objects in different spatial contexts, making them more robust and adaptable. For example, the same machine might be placed along different walls or oriented at different angles in various generated scenes while blocking use of certain angles that inhibit its utility. This introduces diversity while preserving realism in the dataset, which helps the model learn to recognize objects regardless of their specific placement or orientation.

In practice, this variability is achieved by introducing random adjustments to the placement and orientation of objects within predefined limits. For example, a machine that must be placed near a wall can be randomly positioned at different points along the wall, and its orientation can be adjusted within a specified range. Pipes and other infrastructure elements can also be routed differently in each scene, introducing further diversity.

The ability to generate a wide variety of scenes with minimal manual intervention is a key strength of procedural generation. This scalability allows for the rapid creation of extensive datasets, which can be used to train ML models on a broad range of industrial scenarios. In one scene, for example, a set of machinery might be aligned along a southern wall, while in another scene, the same machinery might be distributed across different areas of the floor. This variability ensures that the dataset is not only large in scale but also diverse in configuration, which is essential for building robust ML models.

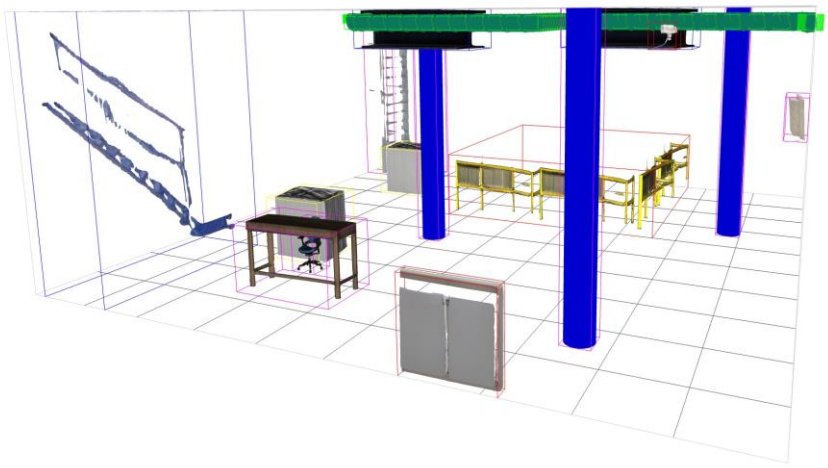


Figure 16. The scene shows the different objects placed and the wireframe used for verification of placement and rotation.

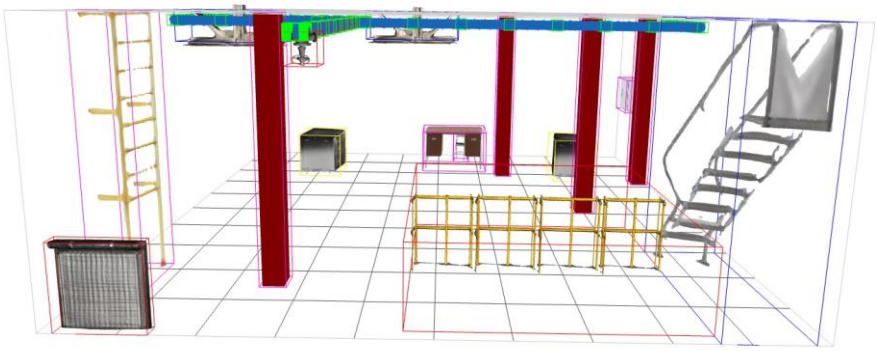


Figure 17. Angle view of a room made with the procedural generation algorithm.

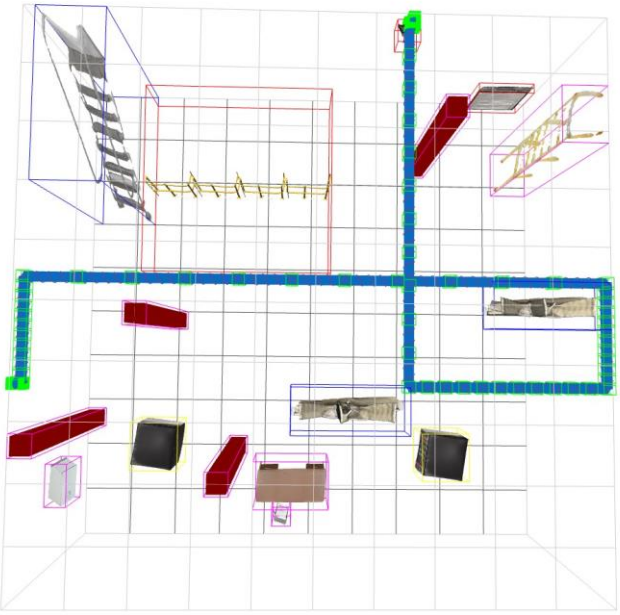


Figure 18. Top view of a room made with the procedural generation algorithm.

After placing each object, its weight was dynamically adjusted to reduce the likelihood of repeated placements, promoting diversity. The algorithm managed row height and reset the position when a row was filled, ensuring efficient use of space. Objects were distributed across multiple areas and rooms, with each area containing several rooms. The distribution algorithm selected objects based on their weights,

recalculating weights after each placement to reflect the reduced prevalence of placed objects. The scenes were saved with the format of the S3DIS dataset, creating point clouds in XYZRGB format of the objects saved as annotations and the combination of the objects, which is called a room. In total eighteen datasets are created each with ten areas, where each area has twenty rooms. This dataset ensures that every Area of the dataset contains all of the objects obtained and that can be independently used for training and evaluation.

4.2.3 Point Cloud Realism

The realism of point cloud data is paramount in generating synthetic scenes that are representative of real-world environments. Realism in point cloud data involves accurately capturing the geometric and color details of objects, as well as simulating the physical interactions and spatial relationships between them. Achieving minimal realism requires data collection and meticulous preprocessing to ensure that the point cloud reflects the true shape, scale, and texture of objects. This realism is crucial for various applications, including training machine learning models, conducting virtual simulations, and performing accurate 3D reconstructions.

4.2.3.1 Point Cloud Noise

Point cloud data collected in industrial environments is often noisy, as shown in Figure 19, due to factors such as sensor limitations, environmental interference, and surface reflectivity. To replicate these real-world imperfections, controlled noise is introduced into the synthetic point clouds generated in this study. Noise is added by displacing the points in the point cloud using a normal (Gaussian) distribution, ensuring that most points are only slightly displaced while a smaller number of points experience larger displacements. This simulates the kinds of errors typically encountered in real-world scanning processes. The level of noise is controlled by a user-defined parameter, allowing it to be adjusted based on the specific requirements of the dataset, which may be driven by example by the performance of a specific TLS or the materials in a scene

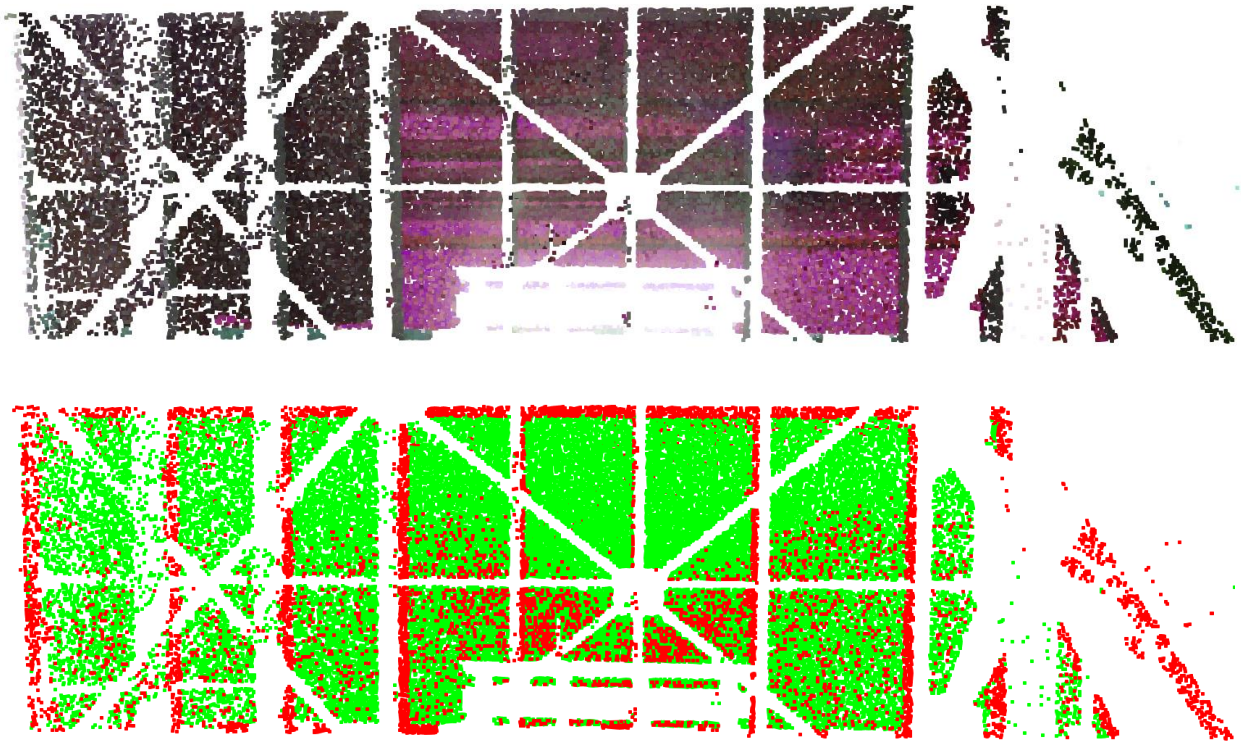


Figure 19 Point cloud noise from a floor of the real-world scan.

This added noise serves two important purposes. First, it enhances the generalization ability of machine learning models trained on the dataset. Models trained solely in noise-free data often perform poorly in real-world environments, where data is inherently noisy. By training on noisy data, models learn to handle these imperfections, improving their robustness and reliability. Second, the noise simulates realistic scanning defects such as sensor artifacts or distance errors. Gaussian noise affects the spatial coordinates of the points, as illustrated in Figure 20, with an adjustable standard deviation to represent different levels of sensor precision.

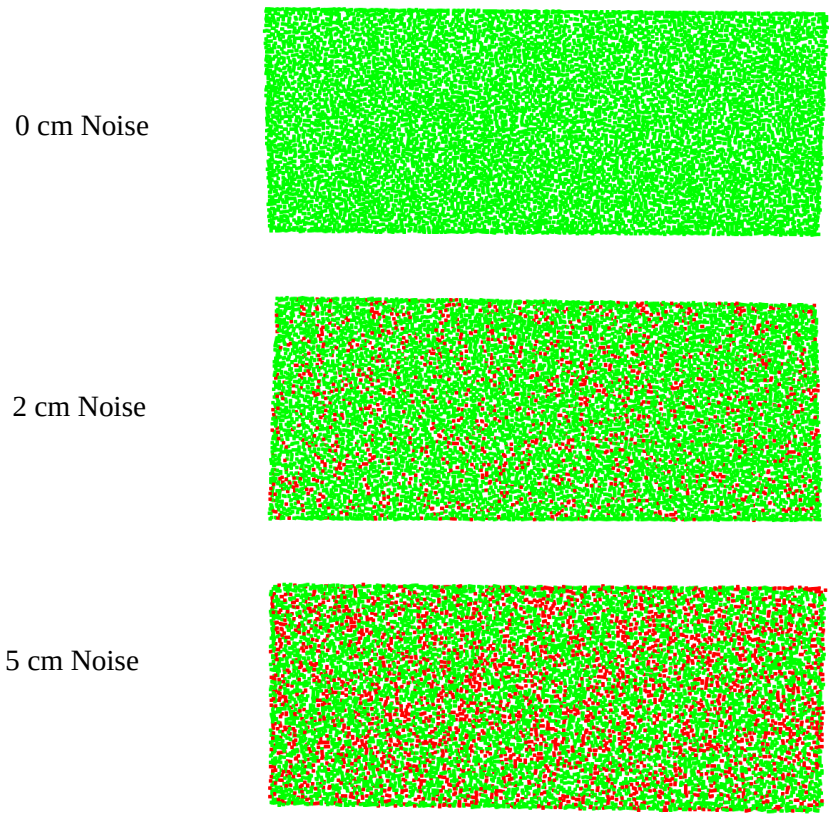


Figure 20 Section from a synthetic wall of the generated dataset, with 0 cm, 2 cm, and 5 cm added noise of the datasets. The outliers from the plane are shown in red.

4.2.3.2 Point Cloud Density

The point cloud density is modified to test multiple instances of point cloud density variability in real life scans which can occur from sparsity due to scan distance and to object reflectiveness. To modify the density the 3D meshes from GLB files are loaded, then vertex normals are computed to prepare the mesh for sampling. Poisson disk sampling is applied based on the surface area of the mesh, with the number of sampled points determined by a base density parameter, limited to a maximum of 50,000 points. This ensures that the method can handle varying mesh sizes while controlling computational load. The resulting point clouds are saved in PCD format for further use.

The approach includes functions for directory management and batch processing of multiple GLB files with varying point densities. Output directories are created automatically, and point clouds are generated

with densities of 1,000, 5,000, and 10,000 points per square unit of surface area. The method is designed to efficiently handle different 3D models and organize the output files. Error handling is built into the process to manage issues such as incomplete or incorrect mesh files. This workflow provides a streamlined approach for generating point clouds from 3D meshes.

4.2.3.3 Hidden Point Removal

Using the same process as in Lopez (2024), the generated scenes are processed to resemble lidar scans. This involves capturing multiple views of the scene and applying hidden point removal to simulate the occlusions and visibility constraints typical of lidar data. The center and diameter of the point cloud are calculated to define the viewing parameters. Four diagonal camera positions around the point cloud are defined to simulate different perspectives. For each position, hidden point removal is applied, removing points that are not visible from the selected camera position. This simulates the occlusion effects encountered in real-world lidar scans. One of the visible point clouds is randomly selected to replace the original, ensuring a realistic and varied representation of the scene. This approach enhances the visual realism of the generated datasets and provides a more accurate simulation of real-world sensor data, making the datasets more valuable for training and evaluating machine learning models in environments that mimic actual operational conditions.

4.2.4 Implementation Details

The implementation of the code in the GitHub repository demonstrates the complete methodology to creating a realistic dataset by incorporating various procedural rules. The *SceneGraph* class initializes a directed graph to store scene constraints and relationships, which is fundamental for maintaining consistency and coherence. The *parse constraints* method reads constraints from a JSON file and populates the graph, ensuring that all elements and their interactions are accurately represented. This approach ensures that the dataset remains consistent with the defined realism criteria.

The *ObjectPlacer* class is responsible for placing objects within the scene based on the constraints defined in the scene graph. It ensures that objects are placed logically and in a manner that reflects real-world industrial layouts. The *place _floor*, *place ceiling*, and *place walls* methods ensure the accurate placement of fundamental structural elements, providing a realistic foundation for the dataset. Each of these methods uses detailed object attributes and precise positioning to create a believable environment. In

addition to structural elements, the *ObjectPlacer* class also handles the placement of various industrial objects like stairs, pipes, railings, and machinery. The *place objects* method ensures that these objects are placed according to their specified constraints, considering factors like size, color, and spawning links. This method also uses a *PlacementHelper* to check for valid positions and prevent overlaps, further enhancing the realism of the scene. The Visualizer class enhances realism by creating wireframe representations and loading detailed 3D meshes for objects. The *create wireframe box* method generates accurate visual representations of objects, while the *load and place mesh* method loads and transforms 3D meshes to fit the scene. The meshes are then replaced with their corresponding point clouds. A pipe is generated and then subsampled to generate variable point clouds of pipes. These visual elements are crucial for creating an immersive and believable dataset, allowing users to visualize the industrial environment accurately. Moreover, the Visualizer class includes methods for visualizing the entire 3D grid and the scene graph, providing comprehensive views of the dataset. The *visualize_*3*d_grid* method displays the complete scene with all placed objects, while the *visualize_graph* method provides a graphical representation of the scene graph, highlighting the relationships and constraints among objects. The procedural rules also involve feedback and revision mechanisms. After the initial creation of the dataset, the code allows for visualization and saving of the scene, enabling users to review and refine the dataset. The ability to save scenes as .ply files ensures that the dataset can be easily shared, reviewed, and revised based on feedback from industry experts or other stakeholders. Lastly, the code integrates dynamic elements by considering variability and unpredictability in object placements. The *adjust_positions near spawning_links* method adjusts object positions based on spawning links, ensuring realistic interactions and placements. This dynamic approach reflects the complexity and variability of real-world industrial environments, further enhancing the realism of the dataset which is shown in Figure 21.

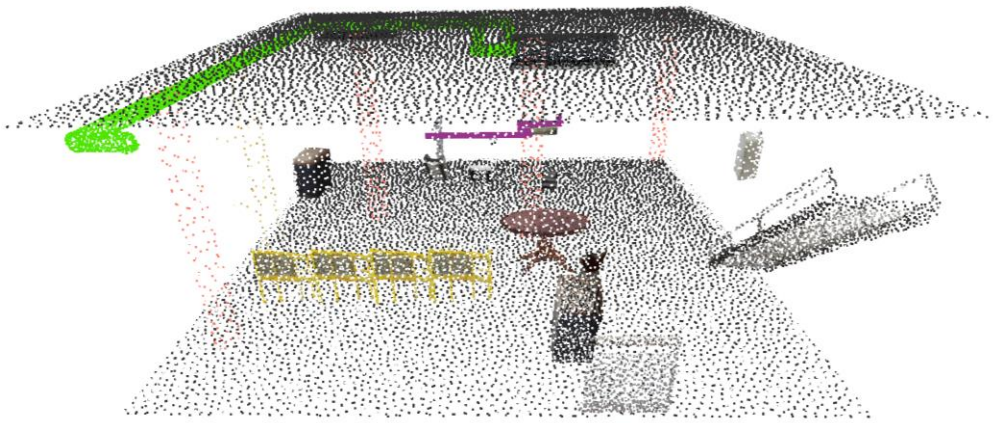


Figure 21. Visualization of a point cloud room with 1000 point minimum density and 2cm noise objects.

4.3 Experiments

The experimental evaluation was performed on a high-performance desktop equipped with 125.6 GiB of RAM, an AMD® Ryzen Threadripper 3960X 24-core processor, and an NVIDIA RTX 3090 GPU. The system featured 21.0 TB of disk space and operated on Ubuntu 20.04.6 LTS with GNOME 3.36.8, utilizing the X11 windowing system. This hardware configuration was used to train a Superpoint Transformer model. The Superpoint Transformer (SPT) network was configured with key hyperparameters, including a voxel size of 0.03 for down sampling, a batch size of 1 during data loading, K-Nearest Neighbors (KNN) set to 45 for graph construction, and an anisotropic scaling factor of 0.2 applied for data augmentation. The following experiments were conducted to assess the impact of various dataset parameters—including point density, placement realism, class balance, and hidden point removal (HPR)—on the performance of the trained network tested on a real-world point cloud scan, subdivided into eight sections. The goal was to evaluate how these factors influence segmentation accuracy and point classification performance in point cloud data. Effective object identification depends on the quality of point cloud segmentation.

4.3.1 Datasets

The datasets used in this study include a real-world scan of an industrial scene and a synthetic dataset based on Lopez et al. (2024). The synthetic dataset, procedurally generated to simulate industrial environments, offers greater control over object placement and variability, spanning 23 Areas and 45 Rooms per Area, with up to 30 objects per room. It includes 4500 beams, 900 ceilings, and other industrial objects, creating

a robust environment for model training and testing. The dataset totals 20GB in size and is used to evaluate model performance across diverse industrial scenes. This dataset uses the scanned dataset as validation.

The real-world dataset comprises 360 point clouds of an industrial space, segmented into various objects (e.g., beams, cables, chairs) and structural components, providing a detailed benchmark for model validation. This dataset includes 87 beams, 26 cables, and a range of other industrial objects and structures, capturing the complexity and variability typical of industrial environments. It serves as a crucial resource for validating model performance in real-world scenarios.

For this research 18 new datasets were created the realistic synthetic datasets generated using the methodology presented previously. These datasets were used to understand better the different aspects of realism and how they affect machine learning point cloud segmentation training. Our datasets are comprised of 9 Areas and the 10th Area to validate and compare the results using Lopez's real-world scan as the validation dataset in Area 10.

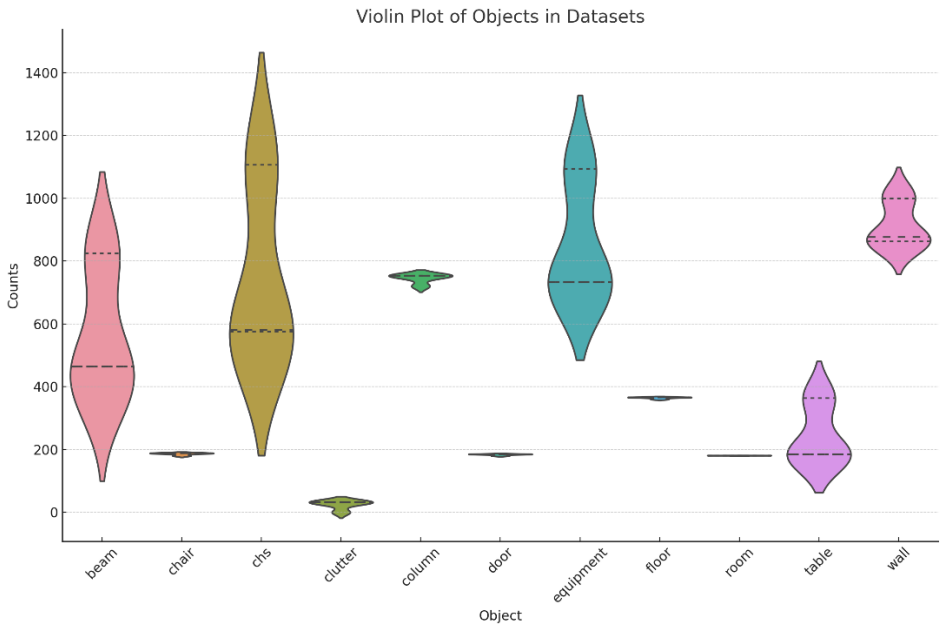


Figure 22. Violin Plot of distribution of objects in the Datasets

The design and preparation of datasets for training deep learning models to segment industrial scene point clouds involve carefully managing the diversity and consistency of object categories. In our datasets, which include objects such as beams, chairs, circular hollow sections (CHS), clutter, columns, doors,

equipment, floors, rooms, tables, and walls, we balance consistency in certain objects while encouraging variability in others. This approach enables the model to learn from both stable elements and diverse configurations, ensuring robust performance in complex industrial environments.

Shown in Figure 22, certain categories—such as doors, clutter, and floors—are characterized by low variability across datasets. For instance, the number of doors consistently ranges between 180 and 185, and the amount of clutter is kept low, often at 32 or zero. Similarly, floors are maintained between 360 and 366 objects in all datasets. By limiting the variation in these categories, we aim to understand how a model can handle environments where these objects appear consistently, while investigating whether maintaining such consistency affects its ability to focus on key structural components and dynamic features in the scene.

On the other hand, the dataset intentionally introduces substantial variability in key objects like equipment, beams, CHS, and walls. Equipment, for example, ranges from 720 to 1093 objects, reflecting the natural diversity in industrial environments where the amount and arrangement of machinery vary significantly. Similarly, beams fluctuate between 360 and 824 counts, offering diversity in structural layouts. CHS objects range from 533 to 1112, representing significant variability in these critical industrial structures. Walls exhibit a range from 848 to 1009, providing diversity in space partitioning. This variety helps the model adapt to different configurations and structural environments.

The contrast between low-diversity and high-diversity categories plays a key role in understanding how the model interprets industrial scenes. Low-diversity categories—such as doors, clutter, and floors—offer a stable foundation, helping to explore whether the model relies on the stability of certain objects or if it generalizes well even with limited variation. Conversely, high-diversity categories like equipment, beams, CHS, and walls are designed to challenge the model and assess its capacity to segment these objects under various conditions. This deliberate variation ensures the model is exposed to the broad range of scenarios that can occur in industrial environments.

4.4 Results

The primary objective of this study was to analyze how factors such as dataset balance, hidden point removal (HPR), noise variability, scene realism, and point cloud density affect the performance of the Superpoint Transformer segmentation model. This section presents an analysis of the results, explaining how each factor influences segmentation accuracy. Comparisons with previous studies are included to contextualize the findings, acknowledging differences in datasets and methodologies.

4.4.1 Point Cloud Density and Noise Variability

Point cloud density and noise levels significantly influenced the model's performance. Higher point densities generally led to improved segmentation accuracy, while increasing noise levels degraded performance. For example, the accuracy for the 'Beams' class increased from 16.58% at 1,000 points with no noise to 24.03% at 10,000 points with no noise. This improvement is due to denser point clouds providing more detailed geometric information, enabling the model to better capture the structures of beams. However, 'Beams' showed sensitivity to noise; accuracy dropped to 4.86% at noise level 5 with 1,000 points and improved to only 10.45% at 10,000 points under the same noise level. Noise introduces perturbations that distort geometric features critical for segmentation, particularly for slender objects like beams.

Classes such as 'Columns' and 'Floors' exhibited more stable performance across different noise and point density levels. 'Columns' achieved an accuracy of 76.30% at 1,000 points with no noise and 80.04% at 10,000 points with no noise. Their larger size and simpler geometric structures make them less affected by variations in point density and noise. The impact of noise and point density on overall performance is illustrated in Figure 23, which shows the mean accuracy performance of the complete point cloud and HPR across varying point densities and noise levels.

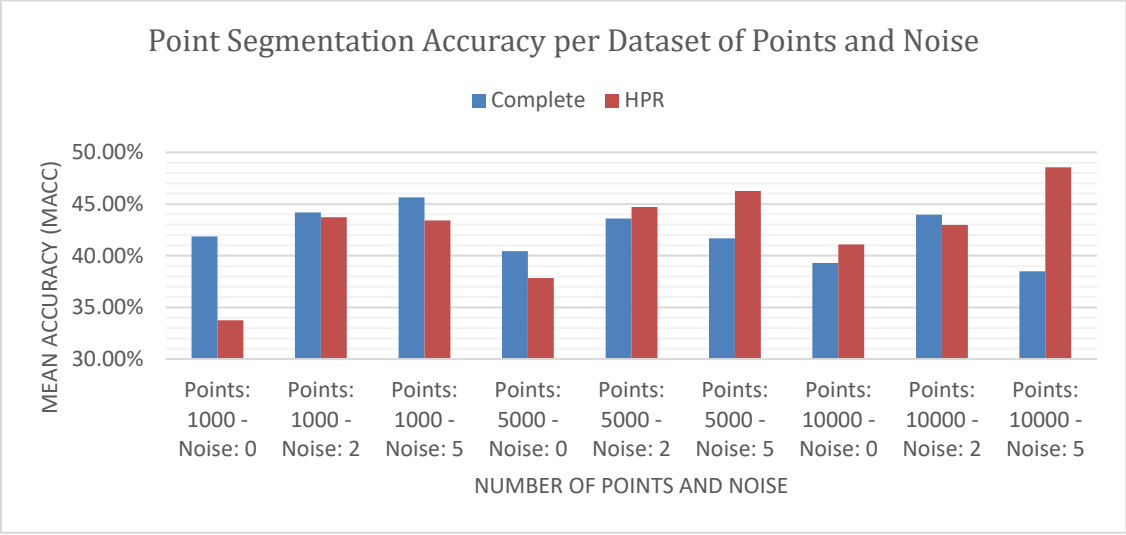


Figure 23. Mean point segmentation accuracy performance of complete point cloud and HPR (hidden point removal) across varying point densities and noise levels.

4.4.2 Hidden point Removal

Hidden Point Removal (HPR) is a preprocessing technique intended to eliminate occluded or non-contributory points from the dataset. Its impact on model performance varied across classes. In some cases, HPR improved performance by simulating the occlusion happening in the real life data, thus reducing noise and computational complexity. As seen in Figure 23, class 'Floors' benefited from HPR, with accuracy increasing from 79.19% (complete dataset) to 85.53% (dataset with HPR). This improvement suggests that HPR effectively filtered out unnecessary points that might confuse the model during training. Clutter often consists of random and irregular objects, so removing occluded or extraneous points helps the model focus on the essential features that define this class.

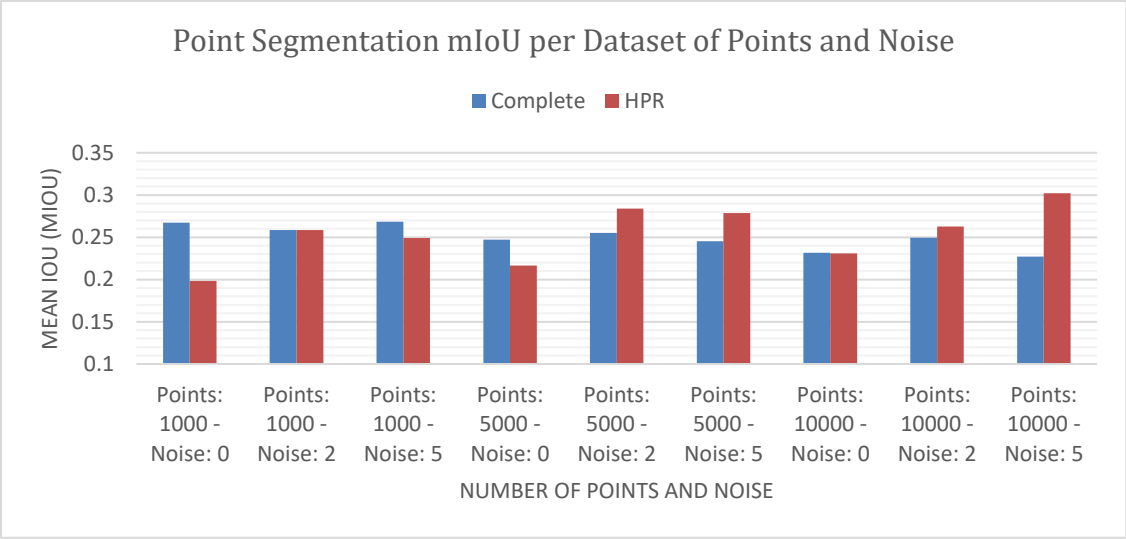


Figure 24 mIoU point cloud segmentation performance of complete point cloud and HPR (hidden point removal) across varying point densities and noise levels.

Conversely, 'Beams' and 'Pipes' experienced decreased performance with HPR. For 'Beams', the accuracy decreased from 25.50% (unbalanced dataset) to 22.75% (unbalanced dataset with HPR). This decline may be due to HPR inadvertently removing essential points that define the slender structures of beams and pipes. Since these classes have thin and elongated geometries, any reduction in point density—even through targeted removal—can significantly impact the model's ability to recognize them. The removal of hidden points may disproportionately affect these classes by eliminating critical geometric information.

Dataset balance, as per Figure 24, also played a significant role in model performance. Balancing the dataset increased the mean accuracy (mAcc) from 45.71% (unbalanced) to 55.10% (balanced). This improvement is attributed to providing the model with sufficient training examples for each class, preventing bias toward classes with more samples. When certain classes are underrepresented, the model may not learn their features adequately, leading to poor performance. Balancing addresses this issue by ensuring that each class is equally represented during training. When HPR was applied to the balanced dataset, mAcc further increased to 57.02%. The combination of dataset balancing and HPR seems to enhance model performance by ensuring that each class is adequately represented and that the training data is more focused and less noisy. However, the benefits of HPR in this context may vary depending on the class and the specific characteristics of the data.

4.4.3 Per-Class Performance Analysis

The segmentation performance varied significantly across different object classes due to their inherent geometric characteristics and the complexity of their representations in point cloud data.

'Beams' consistently exhibited lower accuracy scores compared to other classes, especially under high noise conditions. The slender and elongated geometry of beams makes them more susceptible to noise and occlusion. Even slight perturbations in point positions can disrupt the continuity of beams, leading the model to misclassify them. Additionally, the limited number of points representing beams in sparse point clouds hinders the model's ability to learn distinctive features. The combination of geometric complexity and sensitivity to data quality explains the lower performance for this class.

'Chairs' presented a challenging class due to their complex shapes and smaller size relative to other objects. The accuracy scores for chairs were generally moderate but improved significantly with increased point density. This improvement suggests that higher point density helps capture the intricate details of chairs, such as legs and backrests, which are critical for accurate classification. However, the presence of noise still affected the model's performance on chairs, indicating the need for robust feature extraction methods that can handle variability in the data.

'Columns' consistently achieved high accuracy scores across all configurations. Their prominent and uniform structure, characterized by cylindrical and I-beam shapes with consistent dimensions, makes them easier to detect and segment accurately. The model likely benefits from the regular geometric patterns of columns, which are less affected by noise and variations in point density. The distinctiveness of columns in terms of size and shape facilitates their recognition.

'Equipment' encompasses various objects with complex and irregular shapes, resulting in low accuracy scores for segmentation. These challenges stem from two primary factors: the granularity thresholds set during modeling and the limited availability of sufficient training data. Granularity thresholds often simplify finer details crucial for accurately identifying diverse equipment objects, while noise can obscure key features, further reducing segmentation performance. Additionally, the heterogeneity of equipment within this class requires the model to learn a wide range of features, which is difficult to achieve with the training data. The lack of sufficient examples prevents the model from generalizing effectively, exacerbating these issues.

'Floors' consistently achieved high accuracy scores, demonstrating the model's proficiency in segmenting large, flat surfaces. The extensive and planar nature of floors provides consistent geometric features that are easily distinguishable from other classes. Even in the presence of noise, the model can rely on the general flatness and horizontal orientation of floors to maintain high accuracy. The simplicity and consistency of floor features contribute to the robustness of the model's performance.

'Pipes', similar to 'Beams', are challenging due to their cylindrical and elongated shapes. The accuracy improved with increased point density but dropped significantly under higher noise levels. The model's difficulty in accurately segmenting pipes may also stem from their potential overlap with other cylindrical objects, such as columns, requiring more sophisticated feature discrimination. Differentiating between pipes and other similar structures demands high-quality data and advanced modeling techniques.

'Walls' showed moderate to high accuracy scores, benefiting from their large surfaces and distinctive vertical orientation. The model could effectively leverage these features to distinguish walls from other classes. Noise had a lesser impact on walls compared to smaller or more complex objects, likely because the overall structure remains recognizable even with point perturbations. The consistent planar geometry and orientation of walls make them relatively easier to segment.

Table 6. Mean point segmentation accuracy per object class in each dataset

	Datasets	*Beams*	*Chair*	*Columns*	*Doors*	*Equipment*	*Floors*	*CHS*	*Walls*
Complete	Points: 1000 - Noise: 0	16.60%	47.71%	75.96%	1.01%	14.32%	71.68%	13.29%	73.96%
	Points: 1000 - Noise: 2	11.76%	52.66%	69.21%	2.02%	10.11%	75.57%	24.21%	**89.87%**
	Points: 1000 - Noise: 5	4.54%	58.22%	67.50%	30.76%	15.03%	79.19%	37.37%	70.09%
	Points: 5000 - Noise: 0	12.48%	22.50%	76.99%	7.16%	6.31%	60.11%	24.19%	60.22%
	Points: 5000 - Noise: 2	13.23%	**79.17%**	69.43%	33.16%	13.44%	53.53%	**43.23%**	59.68%
	Points: 5000 - Noise: 5	12.08%	26.86%	74.58%	5.95%	8.54%	62.28%	36.17%	51.96%

	Points: 10000 - Noise: 0	9.18%	42.82%	77.81%	**33.57%**	1.51%	50.86%	32.91%	64.72%
	Points: 10000 - Noise: 2	17.32%	54.99%	**84.34%**	1.95%	15.28%	56.44%	39.61%	63.42%
	Points: 10000 - Noise: 5	**18.10%**	50.13%	80.26%	3.95%	5.55%	48.75%	35.21%	50.82%
HPR	Points: 1000 - Noise: 0	8.54%	7.40%	52.72%	2.32%	11.41%	65.90%	12.54%	70.66%
	Points: 1000 - Noise: 2	11.76%	52.66%	69.21%	2.02%	10.11%	75.57%	24.21%	78.64%
	Points: 1000 - Noise: 5	14.41%	51.82%	80.92%	14.30%	16.13%	58.92%	19.44%	75.13%
	Points: 5000 - Noise: 0	5.43%	18.83%	62.88%	3.79%	14.49%	64.73%	21.36%	64.64%
	Points: 5000 - Noise: 2	13.47%	61.77%	76.40%	9.61%	11.73%	81.75%	22.85%	62.28%
	Points: 5000 - Noise: 5	15.89%	31.64%	76.78%	2.82%	16.70%	73.70%	27.08%	68.42%
	Points: 10000 - Noise: 0	2.93%	61.88%	71.01%	16.33%	20.68%	63.80%	28.41%	59.82%
	Points: 10000 - Noise: 2	15.23%	59.92%	78.90%	18.55%	9.04%	73.56%	27.04%	50.31%
	Points: 10000 - Noise: 5	17.07%	72.45%	73.47%	2.07%	**21.44%**	**85.53%**	33.19%	63.06%

4.4.4 Realism and Complexity

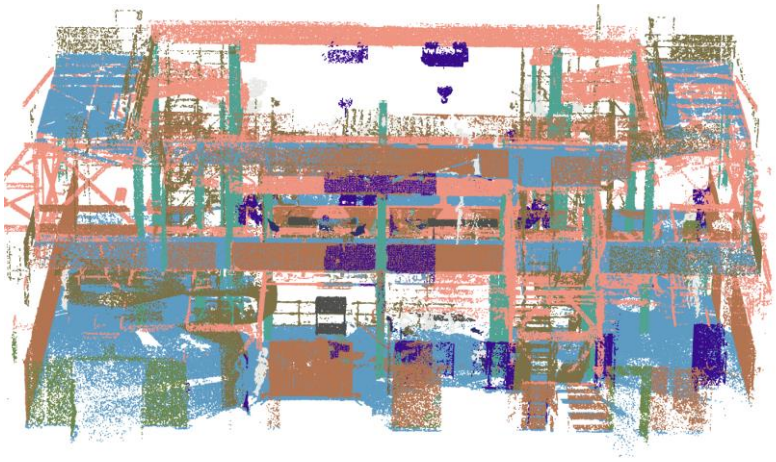
The complexity and realism of scenes played a crucial role in segmentation performance as it can be seen in Table 6. Realistic scenes often contain occlusions, varying lighting conditions, and diverse object arrangements, posing challenges for the model. Classes like 'Beams', 'Pipes', and 'Equipment' were more affected by scene complexity. Occlusions can hide parts of these objects, reducing the number of points representing them and making it harder for the model to learn their features. Complex backgrounds and overlapping objects increase the difficulty of distinguishing these classes. For example, 'Pipes' experienced a significant accuracy drop when noise levels increased in realistic scenes. The model may confuse pipes with other elongated structures or fail to detect them altogether due to partial occlusions.

Intricate geometry and positioning of equipment in cluttered environments such as Area 10 can also lead to misclassification. The variability in equipment types and the potential for them to be partially hidden or overlapping with other objects complicate the segmentation task. These challenges highlight the need for models that can handle occlusions and extract context-aware features. Conversely, larger and simpler structures like 'Floors', 'Walls', and 'Columns' were less impacted by scene complexity. Their extensive surfaces provide ample data points, and they are less likely to be entirely occluded. The model could rely on consistent geometric features, such as flatness for floors and vertical orientation for walls, to maintain high segmentation accuracy even in complex scenes. The prominence and regularity of these structures make them more resilient to the challenges posed by realistic environments.

4.5 Analysis and Discussion

High point density and low noise levels are conducive to better segmentation performance. Denser point clouds provide more comprehensive geometric information, enabling the model to capture intricate features necessary for distinguishing between classes. This is particularly important for classes with complex geometries or small sizes, such as 'Chairs', 'Beams', and 'Pipes'. The additional data points help the model learn more detailed representations.

Noise adversely affects performance by introducing random perturbations that obscure critical features and edges. Small and complex objects are especially susceptible to noise because their recognition relies on precise geometric details. For example, slight deviations in point positions can significantly alter the perceived shape of a beam or pipe, leading to misclassification. Generating datasets to develop models that are robust to noise is essential, especially for real-world applications where data imperfections are inevitable. Dataset balancing proved to be an effective strategy for improving overall performance, particularly for underrepresented classes. Ensuring that each class has sufficient training samples helps the model learn the diverse features required for accurate segmentation. Without adequate representation, the model may become biased toward more prevalent classes, neglecting the nuances of less common ones.



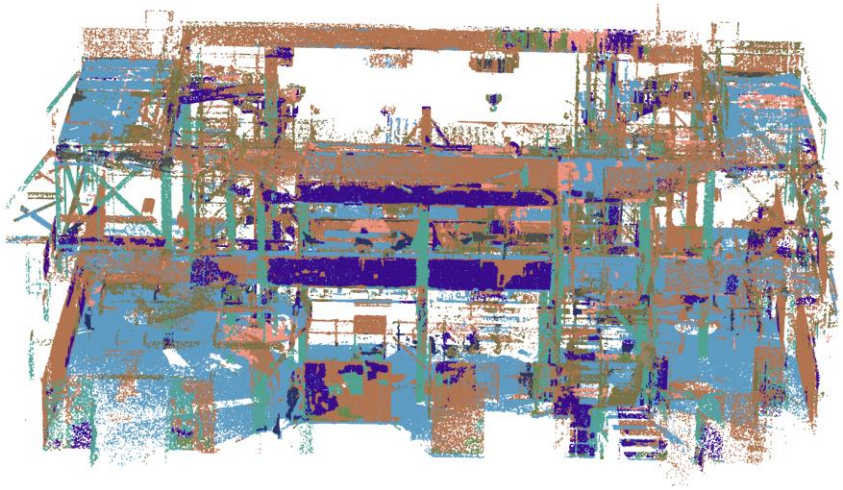


Figure 25. Complexity of Area 10 ground-truth segmentation and segmentation by the dataset of 10000 points with 5cm noise.

However, the combination of dataset balancing and HPR must be carefully managed, as HPR can sometimes remove essential information, depending on the class and the nature of the data. The mixed impact of HPR suggests that it is not universally beneficial and should be applied selectively. For classes where irrelevant or redundant points are prevalent, HPR can enhance performance by reducing noise. In contrast, for classes that require as much data as possible to capture their defining features, such as 'Beams' and 'Pipes', HPR may be detrimental.

The influence of scene realism and complexity, as seen in Figure 25, highlights the need for models that can handle occlusions and overlapping objects. Incorporating techniques such as attention mechanisms or context-aware feature extraction may help the model focus on relevant parts of the scene and improve segmentation accuracy for challenging classes. Additionally, data augmentation strategies that simulate realistic conditions could prepare the model to better handle variability in real-world applications.

Chapter 5
Specific Object Finding in Point Clouds based on Semantic Segmentation and Iterative Closest Point

Statement of Contributions

The research in this chapter was completed under the supervision of Dr. Carl Haas and Dr. Sriram Narasimhan at the University of Waterloo. I am the primary contributor of the specific object finding methodology, all the Python code implementation.

4.6 Introduction

3D point clouds are increasingly being used in many industries to support various aspects of asset management due partly to the ease of labeling and measurement in virtually similar scenarios. Some applications include object and scenario modeling, creating inventory, progress tracking, and quality assurance (Wang & Kim, 2019). Enabling these is labor-intensive, e.g., generating models, inventory counting, progress tracking, and verifying the quality of objects. Human labeling is slow, creating a bottleneck for automating applications that require finding specific objects in point clouds—especially when the clouds are dense and the objects belong to a class where there are only minor differences between the objects. Industrial examples of finding specific objects include finding a specific pipe spool in an industrial scan or an assembled component in a construction laydown yard which can come from scans or be synthetically generated (Ma, 2021; Pan, 2022). However, significant technical problems emerge from the unstructured nature of the data, scan noise, and partial occlusions of objects in the scan (Han, 2021). These problems negatively affect the object's point cloud quality and thus deteriorate the object's features, thereby reducing the object-finding performance of the machine learning algorithms.

Object finding is the process in which an object's presence and its geometric characteristics, such as dimensions, location, and orientation, are identified (Sharif, 2017). This can be undertaken by physically placing devices on the object, such as Radio Frequency Identification (RFID) tags, or algorithmically from visual perception data, such as scans. RFID tags, Bluetooth tags, and other physical labeling systems have been used in the past effectively in many industrial applications (Ko, 2010; Pau, 2021). However, adding such devices adds complexity to the process (e.g., repair, power issues, added cost), and in security-sensitive applications such as nuclear or defense, adding physical wireless devices to assets can be challenging from a regulatory approval standpoint. Hence, this chapter focuses on algorithmic approaches supported by laser

and image data that do not involve adding physical devices to assets for finding objects in industrial or commercial facilities. In other words, this chapter is focused on the problem of object finding in point clouds.

Finding specific objects involves associating the target object with the same object embedded in a scene. In this process, the dimensions and the object's geometric features are matched, and the object's location and orientation are obtained from the registration process. There are various approaches for obtaining object locations in point cloud scenes. Machine learning techniques, such as segmentation and object detection (Lai, 2022; Hu, 2023; Lang, 2018), play a crucial role. Non-machine learning-based methods include RANSAC surface matching, a widely used algorithm for fitting models to noisy data and matching 3D surfaces in point clouds (Li, 2021), as well as voting algorithms, which accumulate votes from data points to identify object clusters within the point cloud (Wang, 2015). Additionally, for determining both location and orientation, iterative closest point (ICP) algorithms are employed, facilitating point cloud registration and template matching (Arun, 1987; Poiesi, 2022).

Template matching is a powerful technique that has seen widespread use for object finding (Huang, 2013; Liu & Wan, 2020). Point cloud template matching starts with a point cloud target, also called a template, to be found in a scene (Varga, 2021). Then, the point cloud is pre-processed to assign heuristic measures to geometrical features. Afterward, the heuristics between the scene and the template are compared, and a match is refined using ICP. This process can generate fast matches for pre-processed scenes but lacks robustness with heavily heterogeneous point density. The heuristics generated by the algorithms will use the point cloud geometry as a feature-matching comparison without considering the match's context (Vock, 2019). For example, a flat door point cloud's heuristic features will look very similar to the floor or ceiling and will match if the size is the same, even if the surfaces have different horizontal and vertical differences. Another disadvantage is that new scenes and objects must be pre-processed through the slow heuristic mapping algorithm (Miao, 2021).

Machine learning approaches to template matching have yet to be developed. There is a gap between finding a broad class of objects in a point cloud and finding a specific object's position and orientation. Methodologies that search the location of a point cloud class, such as point cloud object detection (Quintana, 2018; Lang, 2018), or point cloud segmentation (Lai, 2022; Choy & Gwak, 2019; Ma, 2022; Ma, 2020) are fast at labeling the point clouds and learn the features of multiple objects in the same domain and cluster multiple learned features in the same class (Liu & Gao, 2020). For example, a segmentation network trained on multiple cars can learn the features of cars and identify points belonging to cars in a point cloud but cannot

identify a car of a specific model. Therefore, finding specific objects using segmentation networks is an insurmountable task since a dataset must be created with multiple instances of each particular object as a class, and unseen but geometrically closely related objects would still be misidentified. On the other hand, point cloud registration networks use the geometric features of two point clouds to find the orientation of the point clouds that match the closest points (Qi, 2022; Aoki, 2019; Zhang, 2020). However, machine-learning-based point cloud registration suffers from inaccuracies when the point clouds are of different sizes and, like template matching, also suffers from context confusion.

The literature gap described previously and presents a novel network-driven object finding method given a template point cloud. At its core, this method: (i) introduces a hard constraint for ICP object acceptance using point cloud resampling; (ii) establishes a point cloud candidate modification using geometrical point cloud colorization and a double registration strategy to prune candidates that do not meet the object acceptance constraints when compared to the template's point cloud. This method's error rate is minimal for the rotation and translation for multiple matches because of its point cloud geometrical colorization when using ICP. The main contributions of our work are a network-driven specific-object-finding methodology using an object resampling method to obtain an object specificity threshold, a candidate generation using point cloud segmentation, and the registration using geometrically colorized point clouds with color ICP.

This chapter is organized as follows: Section 2 starts with a description of the related work describing the algorithmic methods for template matching, along with machine learning alternatives for object localization and registration in a template matching pipeline. Section 3 explains the methodology for template matching using target resampling, point cloud segmentation, and geometric colorization in point cloud registration. Section 4 describes the experiments, the dataset, implementation details, and the evaluation of the experimental results. Finally, Section 5 discusses the advantages of the framework and the future work for this research.

4.7 Methodology

The overall framework developed in this chapter for object finding consists of four steps (shown in Figure 26):(i) point cloud resampling to establish the object similarity thresholds; (ii) point cloud scene segmentation; (iii) selection of object candidates; and (iv) point matching and registration to match the object's geometric features, and to find its location, orientation, and dimensions. These components are described in this section next.

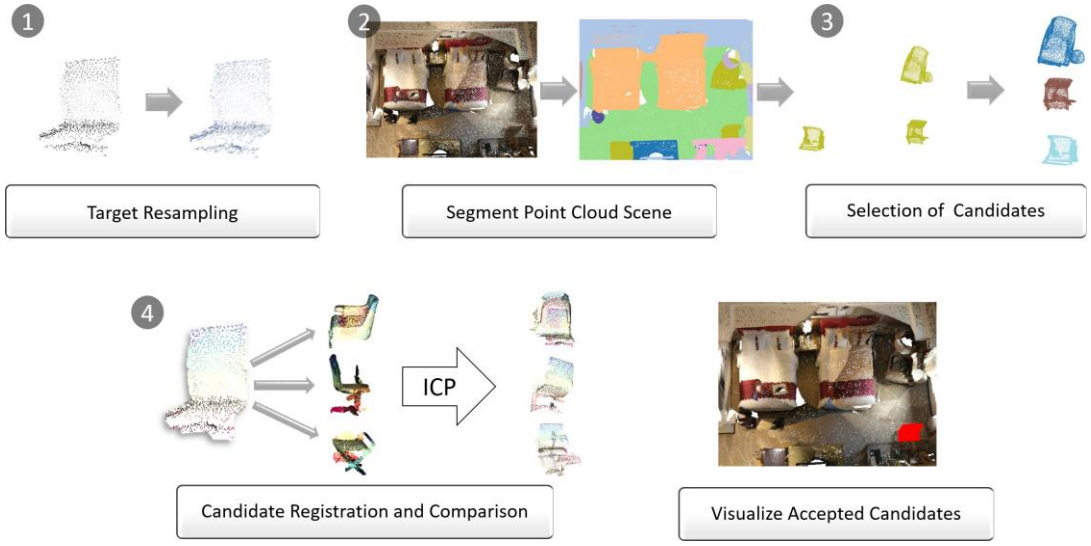


Figure 26. Overview of the specific object-finding methodology. (1) Target resampling for candidate similarity thresholds. (2) Original scene is segmented, and (3) segmented objects with the same labels are clustered and become candidates. (4) Candidates are registered and compared to the thresholds.

4.7.1 Target Point Cloud Resampling to Set Similarity Thresholds

Finding a specific object requires finding correlated geometric features specific to that object and being able to distinguish from similar objects. To do this, similarity thresholds must be set. The process starts with a template object, obtained through a prior scan of the target object. This template is then meshed, and new points are resampled. After obtaining the new points, a point-to-point ICP comparison is conducted between the template and the new points. The results of this comparison are then employed to distinguish between different objects. The ICP outputs are (i) the number of correspondences between the point clouds; (ii) the fitness as measured by the number of inlier correspondences over the number of points in the target, and (iii) the root-mean-square deviation (RMSE) of all inlier correspondences also called the Inlier RMSE. An object's geometry, size, and point cloud density affect the registration. The point cloud (Figure 27a) is meshed using the Tetra mesh (Figure 27b). The tetra mesh uses the point cloud points as vertices and generates faces using Delaunay triangulation to create a surface mesh of interconnected triangles, adding the surface tetrahedrons and optimizing point selection as described by Gao (2020). The mesh is then sub-sampled using the Poisson-disk sampling algorithm by Corsini (2012) with half of the points of the original point cloud (Figure 27c). Halving the number of sampling points allows the algorithm to run faster and lower the point correspondence threshold without sacrificing the specificity of the object. The sampling outputs a point cloud that can be registered to the template, and the registration output produces the minimum

threshold of the fitness and Inlier RMSE values for object similarity. Thus, scale and object character invariance are established for the similarity metric between candidate objects (point clusters) and the target object for the process stage in which a specific match is obtained.

4.7.2 Point Cloud Scene segmentation

Using machine learning for Point cloud segmentation requires datasets with raw point cloud data and labeled point clouds that have classes of objects representative of the environment. In cases such as finding a specific pipe spool in a laydown yard, finding a specific piece of equipment in a heavy construction scene, or finding a specific cabinet in an interior finish staging area for a building project, specialized datasets are required to train the point cloud segmentation network. As of this writing, there are no such datasets available for construction scenarios (Agapaki, 2021). However, the Stanford Large-Scale 3D Indoor Spaces Dataset (S3DIS) (Armeni, 2017) can be used as a prototype while bespoke construction datasets begin to become more available; this is demonstrated in principle in the experimental section later using a synthetic industrial scene.

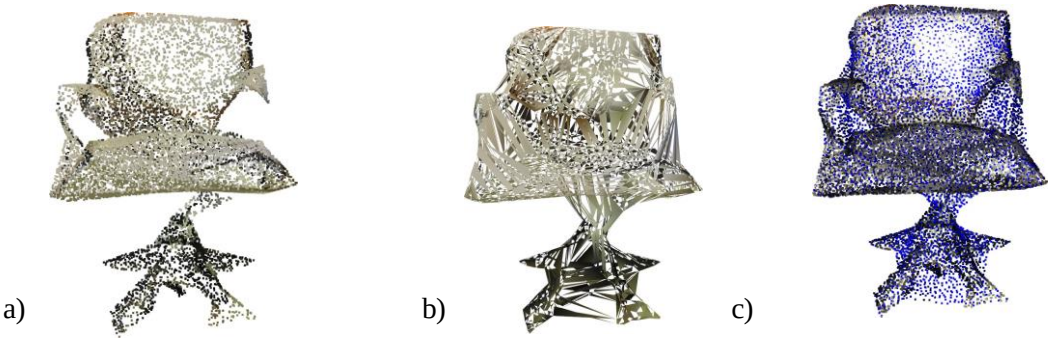


Figure 27. Resampling methodology is used on chair 2 in office 3 from Area 4 from S3DIS point cloud dataset. a) Point cloud template b) meshed template c) Sampled template mesh with newly obtained points represented in blue.

The S3DIS dataset has a set of point clouds of indoor scenes, which can translate to smaller point cloud scenes with similar geometrical features. The classes of objects represented in this dataset are wall, floor, cabinet, bed, chair, sofa, table, door, window, bookcase, picture, counter, desk, curtain, refrigerator, shower curtain, sink, and bathtub. The nature of this domain and its classes are similar enough to the applications encountered in construction that it is reasonable to use it here, and it has been used in related ways in related work (Ma, 2020). The chosen segmentation network, Stratified-Transformer network, with a Residual U-Net architecture by Lai (2022) is used due to its efficiency and accuracy. First, the S3DIS point clouds are separated into raw and labeled point clouds. The dataset split is by Areas 1-4 and 6, made up of 214 scenes and 8158 object annotations, which make up 83.3% of the dataset used for training, and Area 5, with 68 scenes and 1628 annotations which makes up 16.6% of testing as standard protocol. The annotations are text files named after the object they represent in the scene containing the point cloud of the object in XYZRGB format. The raw and labeled point clouds are inputs to the Stratified Transformer network along with their point labels. The loss function is cross-entropy loss with label smoothing, and the optimizer is the Stochastic Gradient Descent (SGD). After training, the network achieves a mean Intersection over Union (mIoU) of 74.3% on the testing dataset. The output of the network is a labeled point cloud where each color represents a class label, as shown in Figure 28.

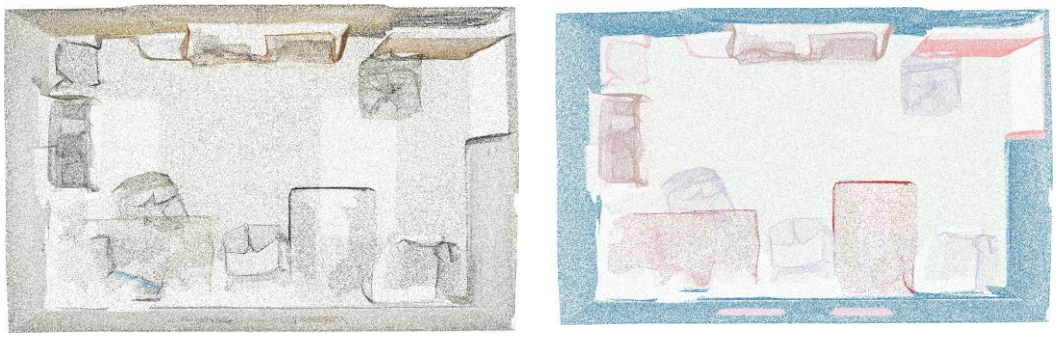
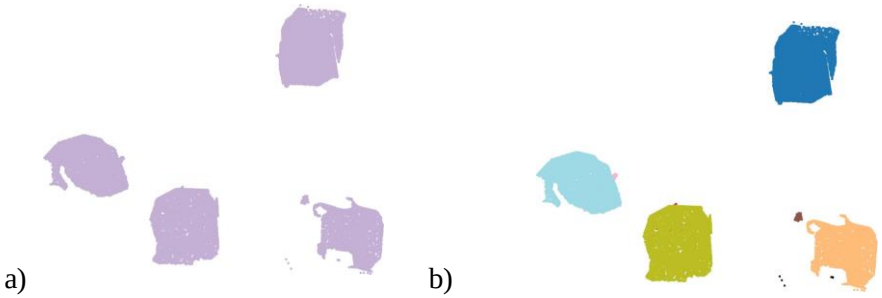


Figure 28. Room segmentation using Stratified-Transformer network Lai (2022). Visualized with Cloud Compare 2.11 (2022)

4.7.3 Isolation and Clustering

The template object's class is obtained by user input or by the point cloud file's name. The class is mapped to its corresponding color in the labeled point cloud. The points in the segmented point cloud that match the template object's class are isolated by comparing the point cloud's predicted class labels values to the desired class label. The output is a point cloud for which all the points belong to the class of the specific

object to be found, as seen in Figure 29a. In this isolated point cloud are the point groups that contain the class label of the template. Therefore, it assumes that one of the point groups is the specific object sought. That assumption can be relaxed by extending similarity comparisons to multiple scenes. The Density-Based Spatial Clustering of Applications with Noise (DBSCAN) algorithm by Schubert (2017) is used to separate the point groups into candidates, as shown in Figure 29b. As can be seen in Figure 29b, there are clusters that are too small and do not represent a viable candidate. Thus, the candidates are selected based on having more than 10% of the points of the template point cloud. Each candidate is then separated into its point cloud file to make it easier to compare afterward.



a) b)

Figure 29. Example of a) Isolating point clouds with same label from segmented scene and b) clustering point clouds to create candidates.

4.7.4 Geometrical Colorization of the Point Clouds and Color ICP

The template point cloud and each candidate become the input for the Fully Convolutional Geometric Features (FCGF) network by Choy & Park (2019). The FCGF network is a residual convolutional network with U-Net architecture used to extract geometric features. The input of this network is a point cloud, and the output is a geometrically colorized point cloud where each geometrically unique feature is mapped to a color. The importance of this step is that geometrically colorized features do not depend on the rotation or translation. Instead, the color features will change depending on the density and geometric placement of the points which increases the robustness of the point cloud registration when applying colored ICP. In the example of the chairs, the point cloud template geometrically colorized is shown in Figures 30 and 31.



Figure 30. Geometric colorization output of FCGF network when taking template point cloud as input.

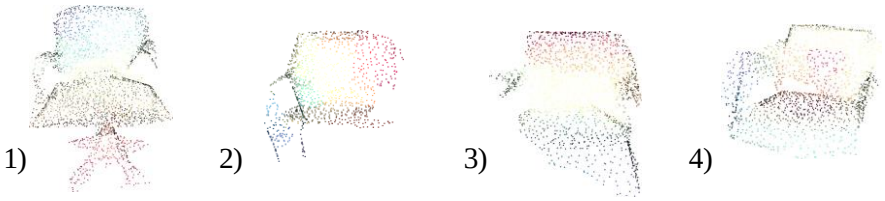


Figure 31. Colorization output of FCGF network of candidates. Candidate 1 should be accepted as it closely resembles the object. In contrast, candidates 2-4 display distinct geometries that deviate from template depicted in Figure 26.

The next step is registering the template's geometrically colorized point cloud to the candidate's geometrically colorized point cloud using the colored ICP algorithm by Park (2017), which optimizes a joint photometric and geometric objective that locks the alignment along the normal and the tangent plane (Figure 32). A one-way colored ICP registration from the template to the candidate's point cloud generates a result that is compared to the constraints set in the resampling step. The registration constraints required to select the candidate as the specific object are (i) the inlier RMSE to be lower than the resampled inlier RMSE; (2) the candidate's fitness to be higher than the resampled fitness; and (iii) the correspondence point set to be higher than 10% of the points. In the unlikely event that the candidate's point cloud could be denser and slightly different than the template's point cloud, passing the tests, a second registration from the candidate's point cloud to the template's point cloud is done to assure object similarity. Doing both registrations ensures the point cloud does not have extra points added in the DBSCAN step. If the constraints are passed, the candidate is set to be a found object. For every specific object that has been found, the point cloud registration transformation, the location, and the points are saved.



Figure 32. Color ICP registration of colorized candidate point clouds to colorized template point cloud. The result of the ICP can be seen as candidates overlapped with template object. Candidate 1 has very similar

1) 2) 3) 4)

Registering each candidate's geometrically colorized point clouds to the template's geometrically colorized point cloud shown in Figure 32 results in the registration values seen in Table 7. If no candidate passes the resampling thresholds, the algorithm outputs a warning saying template object not found. If one or more objects are found, their point clouds can be colorized and exalted or isolated, as shown in Figure 33.

Table 7. Registration results from resampled point cloud and candidates using Colored ICP. Candidates with higher fitness value, lower Inlier RMSE and higher Correspondence Set than resampled threshold are accepted as found objects.

Candidate	Fitness	Inlier RMSE (m)	Correspondence Set	Accepted
Resampled Threshold	0.671	0.011	2095	
Candidate 1	0.966	0.008	2421	Yes
Candidate 2	0.177	0.014	209	No
Candidate 3	0.226	0.013	307	No
Candidate 4	0.388	0.013	597	No

4.8 Experiments

All experiments were performed on an Intel Core i7-10700F CPU @ 2.90GHz x 16 with an NVIDIA GeForce 3070 graphics card in an Ubuntu 20.04.4 LTS operating system. The evaluation of the object-finding methodology is based on the location, rotation, and dimensions of the location found in the point cloud to the ground truth. In addition, an analysis of segmentation errors on the part of the dataset that inhibited finding the object in the point cloud is presented. The object-finding methodology is validated by showing that the automation does present help to the user in search time and accuracy.

The first experiment uses 600 randomly selected scene annotations containing the label of chair, table, sofa, bookcase, or door. Each scene folder consists of a scene point cloud file and annotation files representing the objects in the scene. An annotation file is selected as the template object, and the name of the annotation file is used as the object label, this ensures that the template object is in the scene. This first experiment is used to test the true positives and false negatives.

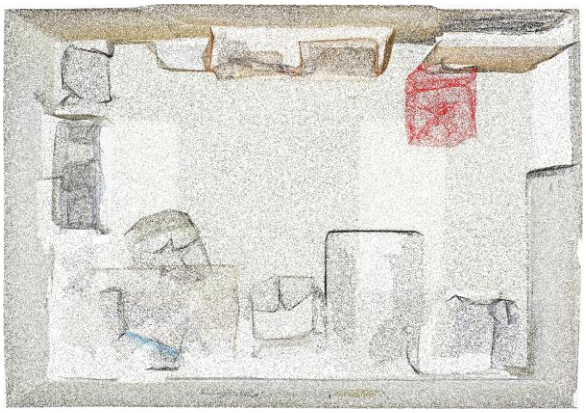


Figure 33. Object found represented in red color. Visualized with Cloud Compare 2.11 (2022)

The second experiment finds an object in multiple scenes. It takes 6 annotation objects as the template objects and searches in 100 scenes if the object is found. This second experiment is used to find false positives and true negatives.

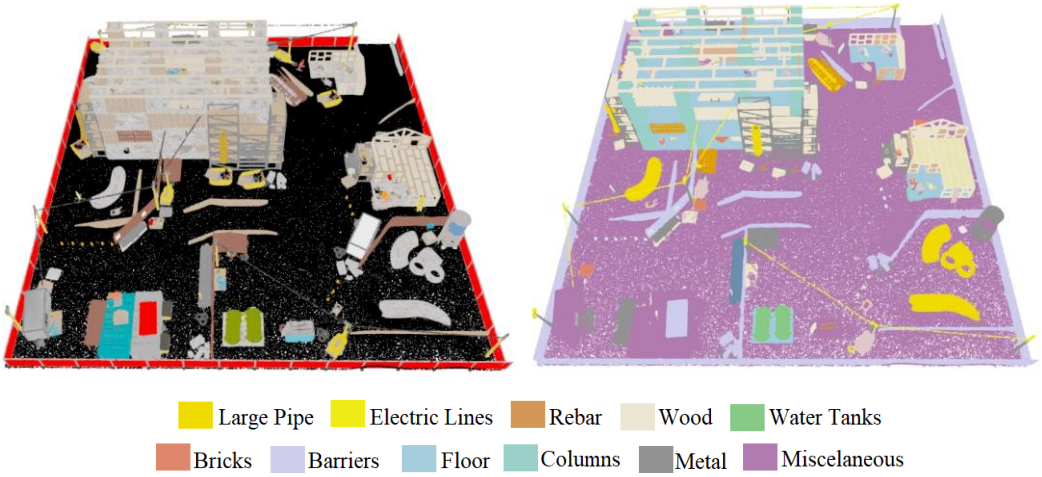


Figure 34. Synthetic point cloud industrial scene and corresponding scene with color labels.

A third experiment was conducted to explore the practical implementation of our methodology on a manually created synthetic industrial scene. The primary objective was to showcase its potential applicability in industrial environments characterized by larger sparsity of points. To achieve this, the experiment focused on larger objects with a lower point density, as illustrated in Figure 34. By examining the results in this scenario, we aimed to validate the adaptability and effectiveness of our methodology under more challenging conditions, where the methodology can be used on already segmented point cloud data. This investigation provides valuable insights into the versatility and robustness of our approach, strengthening its viability for real-world industrial applications with varying point cloud characteristics.

4.8.1 Dataset

Different scenes of the S3DIS (Armeni, 2017) dataset were used as the testing scenes for the object-finding methodology. The scenes used were selected for various challenges and variable settings. One limitation of the S3DIS dataset is the lack of sparsity variability in the point cloud, which could be present in larger point clouds and other domains. The objects in this dataset have 13 semantic classes with annotations of the objects that compose a scene.

4.8.2 Implementation Details

The data file structure followed is based on the PointNet PyTorch by Yan (2019). The dataset used is the Stanford3dDataset v1.2 aligned version of the indoor parsing dataset S3DIS which can be obtained through the Stanford project webpage. The dataset is structured into 6 Areas, each with multiple scenes. Each scene has annotations, and each annotation is a text file with the name of the object in the scene and the point cloud of the object.

The first step of the framework is to open the annotation file in Python and set it as the template object using the Open3D library, which will be the object to be found in the scene, and then resample it. The resampling is done by inputting the template point cloud through the triangle mesh creation from Open3d (Edelsbrunner, 1983) using the TetraMesh function from Open3D (Gao, 2020) as a parameter. The mesh is then sampled using Poisson disk point sampling (White, 2007) using the sample points Poisson disk in the Open3D library function with the sampling rate of the same number of points as the template point cloud. We register the point cloud using voxelization, RANSAC global registration, and ICP with voxel size based on the maximum nearest neighbor distance of the candidate's point cloud. This pipeline can be found under

the global registration function of Open3D. Using the pre-trained model from Lai (2022), the scene point clouds are fed into the model and generate the segmented point cloud. The trained weights can be found under Lai's Stratified Transformer project on GitHub. A Python dictionary maps from label name to segmentation color using the template point cloud's file name. The segmentation point clouds with the same color as the template point cloud are utilized to isolate the candidates. The isolation is generated using an iteration over the points with the same label color.

The clustering uses DBSCAN by Ester (1996), which can be found under the point cloud cluster DBSCAN function of Open3D, where the density parameter to find neighboring points is set to the maximum nearest neighbor distance of the candidate's point cloud, and the minimum points to generate a cluster are set to 10. Each cluster becomes a candidate, and each candidate becomes the FCGF network input (Choy & Park, 2019), which uses the pre-trained network using a voxel size half of the maximum nearest neighbor distance. The FCGF has a MinkUNet34C architecture which creates the geometrical colorization based on the point geometry and sparsity. The weights of this network can be found in the GitHub project page cited in the FCGF paper. Finally, the resampling thresholds are saved, and the geometrically colorized candidates are registered to the geometrically colorized template using the color ICP method. Two colored ICP registrations are done. The first registers the template to the candidate, and if the object passes the tests, a second registration from the candidate to the template is made. If the object passes the metric registration thresholds in both instances, the object is deemed as found in the scene point cloud, and the transformation matrix and point cloud location are saved.

In addition, a synthetic point cloud of an industrial scene was built manually, labeled by hand, and processed using the framework. The scene was created using Unreal Engine 5. The original meshes and colors are made into point clouds by sampling points using a Python script with the open3D library. A second identical scene was created, and the textures were modified to recreate a labeled scene to the color standards of the S3DIS dataset. Colors such as electric conduits, tanks, barriers, and miscellaneous are not specified in the S3DIS color map. The labeled scene is then made into point clouds by sampling

2.5 million points using open3D. The large gray tube on the right of Figure 37 is set as the template to be found by sampling the mesh for 10,000 points. The original color scene, the labeled scene, and the template are inputs to the framework. The methodology above is applied and results in finding the desired pipes, a generator, rebar, and a large ground pipe.

4.9 Results

The capabilities of the specific-object-finding methodology are demonstrated by Figures 35 - 37 and Table 8. In Figure 35, the point clouds of the template objects are at the right of the scenes where the objects were found. The found objects are highlighted in the scene with the red point cloud of where the object is found, along with the proper rotation. Figure 35 shows the scene (Area 3 office 1) where all four objects are identified as individuals and found in the scene even when there are multiple instances of the same chair class. Figure 36 demonstrates the ability of the object-finding methodology to find one specific object or multiple identical specific objects in the same scene. The sofas that are found are four different point cloud objects. However, it can find all four sofas using only one template because of the essentially identical geometric properties of the point clouds. Figure 37 shows that the framework can be applied to larger and sparser objects. The point clouds in the S3DIS dataset scenes have an average of the maximum nearest neighbor distance of 0.05, while the sampled synthetic scene was of 2.5 open3D distance units which are equivalent to 5cm and 2.5 meters, respectively.

Table 8 outlines the average size of the objects, the average number of points the scene has, the average number of points the template objects contain, and the average time taken to find each object. The objects presented reflect the scene objects. The scenes selected have a low number of candidates for doors, since most of the scenes are offices and usually only have one door, and halls have multiple. The number of chairs in the scenarios increases in the offices and declines in the halls. The bookcases have a larger number of template object points while having a similar number of average scene points to the other classes.

a)



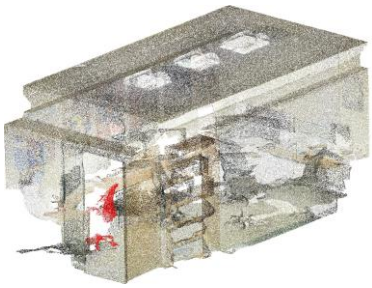


b)

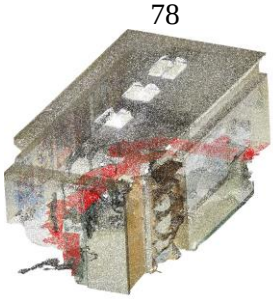




c)









d)

The class with the largest number of points in the template objects is the bookcase, and the smallest number of points is the chair. However, the chair is the one with the highest number of candidates since it appears more often in the scenes. There are 826 chairs, 207 doors, 293 bookcases, 357 tables, and 7 sofas in the tested scenes. Not presented in the table are the synthetic scene and objects.

The time taken to produce the result in Figure 37 was 20 seconds with 10 candidates and an average candidate point size of 25,000 points. The second large pipe in the ground took 17 seconds with the same candidates as the first pipe. The generator was of class miscellaneous and was not detected properly due to DBSCAN not properly clustering the 130 possible candidates. The candidates in this case would be clustered with points far from the point grouping. The rebar being class rebar had the same problem with continuity of the DBSCAN. The rebar would be selected as a continuous material and clustered with two or more sections and be rejected when compared.

The framework can work with segmented point clouds that reflect a good segmentation of the object to be found. In cases where the segmentation of the specific object in the scene is incorrect, the framework won't be able to find the object in the point cloud. Errors that do not affect the steps of isolation or the geometry generated by FCGF of the object do not impede the framework from working and finding the object. It is important to note that a failed automatic search has no penalty compared to a manual search, but a successful automated search saves manual search time according to Dimitrov and Golparvar-Fard (2015). Adequate performance would require that at least 50% of the objects are found and the amount of time for searching an object does not impede their work.

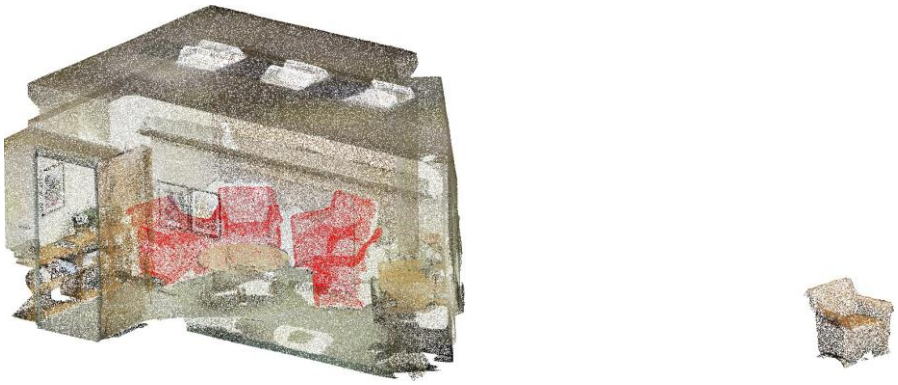


Figure 36. Four identical specific objects are found and visualized using template of a sofa.

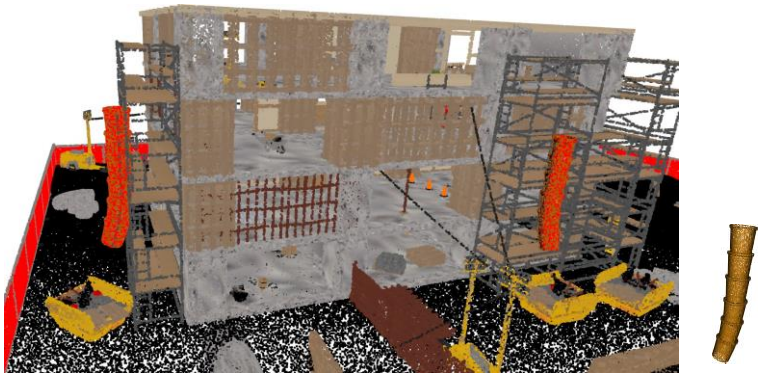


Figure 37. Finding specific object in construction environment. At right is the template of a large pipe. Left, shows the visualization of two identical specific pipes found using the framework

4.10 Evaluation

This chapter has a unique object-finding approach, since it is focused on object-finding accuracy and uses large point cloud scenes from the S3DIS dataset, unlike most template-matching articles that use the BOP: Benchmark for 6D Object Pose Estimation by Hodan (2018). Therefore, there is no reasonable comparison to other benchmarks to compare the results. The annotations from the S3DIS dataset are point clouds with an object's exact location and dimensions. Once an object is found, the annotation can be used to calculate the centroid of the point cloud and the rotation using ICP registration. This methodology's performance is

analyzed using equations 3-6, testing the accuracy, precision, recall, and F1 scores of the framework based on the confusion matrix in Figure 38.

Table 8 Result of object finding tests using the S3DIS dataset for scenes and annotations

Specific Object Finding Test				
Object	Average number of candidates	Average template points	Average scene points	Average Time (s)
Door	2.84	28,836	581,204	8.67
Chair	7.25	6,883	727,116	10.83
Bookcase	4.13	28,520	665,655	16.53
Table	5.58	15,965	720,016	10.37
Sofa	1.75	21,680	575,290	5.08

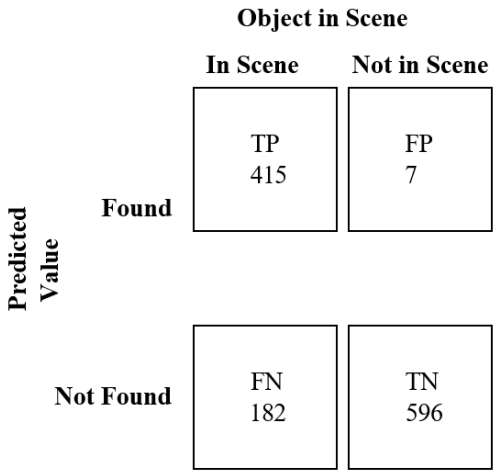


Figure 38. Confusion matrix of object finding framework**.**

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN} \quad (3)$$

$$Precision = \frac{TP}{TP+FP} \quad (4)$$

$$Recall = \frac{TP}{TP+FN} \quad (5)$$

$$F1 = \frac{2*Precision*Recall}{Precision+Recall} = \frac{2*TP}{2*TP+FP+FN} \quad (6)$$

The methodology uses 600 point cloud scenes where an annotated point cloud of that scene is used as the template object. Another 600 scenes are used where the template point clouds are not found in that scene. As seen in Figure 38, if a template object is in the scene and the object is found with very low rotation and translation error, the result is a true positive(TP). However, if the object was found and it was not in the scene, it is a false positive (FP). If the object was not found but was in the scene, it is a false negative (FN). Finally, if the object was not found and was not in the scene, it is considered a true negative (TN). The accuracy of this methodology is 83.55%, the precision is 98.34%, the recall is 68.36%, and the F1 score is 80.66%. These values are above the requirements for a typical user to have utility from the implementation of this methodology. It has to be noted that while there was rigor in establishing what objects were false positives, the objects were not dissimilar to the template objects themselves.

Some insights that are revealed from Table 8 is that the larger the number of points in the template object the longer it takes. This is due to the nature of resampling the object. However, the object only needs to be resampled once, and its resample point cloud can be stored for future use. Another time dependency is the number of candidates that are compared. The larger the number of candidates the longer the time is needed to process each through the FCGF network

4.11 Discussion

The results demonstrate a robust method for specific object detection within point clouds. Nevertheless, it is essential to acknowledge that the networks and algorithms employed in this study necessitate multiple parameters, which can impact the reliability of the outcomes. The resampling technique, employing TetraMesh, requires generation from the original point cloud, with voxel size being a critical factor. It resamples from the mesh created to produce a new point cloud. The resampling parameters, like voxel size, should be determined not solely based on the original point cloud but also considering the average point cloud distance across the scene. This information can be readily obtained using the open 3D library.

Additionally, the density of the object influences the resampling process. If more points are resampled than in the original object, it leads to an overly high threshold, necessitating a 10% reduction in the number of points taken from the original object. Conversely, the points taken from the mesh are too few or heavily

reliant on the original object. In that case, it may result in excessively low thresholds, potentially leading to increased false positives.

During the scene segmentation, it is crucial to account for the manageable size of the point cloud for processing. In cases where the size is too large, it may necessitate splitting. Subsequently, the isolated candidates are clustered using DBSCAN. It is imperative to consider the sparsity of the point cloud and a maximum cluster limit is set. While this method can cluster a large number of point cloud objects, one potential area for improvement of DBSCAN, not encountered in this chapter, arises in scenarios with substantial density differences across the point cloud. In real-world scans, this can lead to the epsilon parameter, which defines the gap between clusters, becoming too large or too small for the current density, potentially resulting in misclassified clusters. However, this challenge can be addressed through dataset area splitting based on the minimum average point distance. It is also pertinent to highlight that this methodology excels in automating specific object detection on segmented datasets passed through the segmentation network. However, this confines the objects to those within its training set. There are two potential workarounds to this limitation. Firstly, manually segmented scenes can be generated, provided the labels are accurately assigned, and then the methodology can be applied. Secondly, a non-labeled data class can be employed. If this class encompasses miscellaneous objects, it can be designated as a label, and the methodology can subsequently be utilized to identify a specific object within this miscellaneous category.

Chapter 5
Thesis Summary, Conclusions and Future Work

5.1 Summary

This thesis has developed and evaluated a comprehensive methodology for specific object detection in point clouds and point cloud segmentation and in industrial scenes, contributing significantly to the growing field of neural network applications for 3D data. Working with point clouds presents several inherent challenges, such as class imbalance, noise, and limited data availability, particularly in real-world industrial settings. These challenges are compounded by the unique geometric characteristics of objects in industrial environments, where structural elements like beams, pipes, and equipment often exhibit complex and irregular forms, making them difficult to segment and detect accurately. The methodology proposed in this thesis addresses these challenges by combining a novel object-finding approach with the use of Fully Convolutional Geometric Features (FCGF) and the Iterative Closest Point (ICP) algorithm, offering a precise and computationally efficient method for identifying specific objects within point clouds.

The core of the proposed methodology is an object-finding framework that operates on the geometric features of point clouds, enabling the identification of objects without relying on color data or extensive preprocessing. Unlike traditional template-matching techniques, which often require colorized point clouds or substantial preprocessing of scenes, this framework focuses on segmentation-based candidate generation and geometric matching. The results demonstrated that the methodology could successfully identify multiple instances of identical objects, both large and small, within complex industrial scenes. Furthermore, the method performed effectively in environments with multiple candidates and varying noise levels, showing robust detection even in cases where the scene was cluttered or where objects were partially occluded.

To ensure the robustness and generalizability of the framework, the thesis introduced a weight-balanced synthetic dataset designed specifically for industrial applications. Given the practical limitations of collecting comprehensive real-world datasets due to security concerns, data confidentiality, or physical inaccessibility in industrial settings, synthetic data generation became a crucial aspect of this research. The synthetic dataset closely mimicked the characteristics of real-world industrial scans, including variations in point cloud density, noise levels, and object distribution. By balancing the representation of different object classes, such as beams, pipes, equipment, and walls, the dataset provided a more equitable foundation for training neural networks, preventing biases toward overrepresented classes. The results showed that the

weight-balanced dataset improved the performance of the segmentation model, particularly for smaller or less common objects that are often underrepresented in traditional datasets.

An important consideration in the methodology was the role of point cloud density and noise variability in the segmentation process. Higher point cloud densities were found to significantly improve segmentation accuracy, particularly for geometrically complex objects like beams and equipment. Dense point clouds provide more detailed geometric information, enabling the model to better capture the intricate structures of these objects. Conversely, increased noise levels degraded the model's performance, particularly for objects with thin or elongated geometries, such as beams and pipes. The sensitivity of these classes to noise underscores the importance of high-quality point cloud data for accurate segmentation.

In addition to dataset balance, preprocessing techniques like Hidden Point Removal (HPR) were examined for their impact on model performance. HPR was found to improve segmentation accuracy for some classes, such as floors and walls, by eliminating irrelevant or occluded points that could otherwise confuse the model. However, for slender objects like beams and pipes, HPR resulted in decreased accuracy, as the removal of hidden points inadvertently reduced the model's ability to detect the critical geometric features of these objects. This finding highlights the need for class-specific preprocessing strategies to avoid negatively impacting the model's performance on certain object types.

Finally, the thesis explored the influence of scene realism and complexity on the segmentation model's performance. Realistic scenes, characterized by occlusions, varying lighting conditions, and complex object arrangements, posed significant challenges for specific object detection. While large, simple structures like walls and floors were less affected by these complexities, smaller or more intricate objects, such as chairs and equipment, experienced reduced segmentation accuracy in realistic scenes. This finding underscores the need for models that can handle the complexities of real-world industrial environments, where occlusions and overlapping objects are common.

5.2 Conclusions

The research presented in this thesis demonstrates significant advancements in specific object detection within point clouds, particularly in complex industrial environments. The developed methodology, leveraging Fully Convolutional Geometric Features (FCGF) and the Iterative Closest Point (ICP) algorithm, provides a precise and efficient approach to identifying objects in large point cloud scenes. Through rigorous testing, the method proved effective in detecting both large, easily recognizable objects and smaller, complex ones, even in challenging scenarios with multiple candidates and noise. The results

from experiments using the S3DIS dataset show that the object-finding accuracy remains high across various scenes, with a precision of 98.34% and an overall accuracy of 83.55%. The methodology's ability to handle occlusions and register objects accurately within noisy or cluttered environments further underscores its robustness for industrial applications.

- The specific object-finding methodology efficiently detects objects in large-scale point clouds using geometric features without relying on color data.
- It achieves high precision and accuracy across a range of objects, handling both large and small objects in complex, noisy industrial scenes.
- The framework is particularly effective at minimizing false positives and false negatives, making it suitable for use in real-world applications where object placement and scene dynamics are variable.

In addition to object detection, the research underscores the importance of dataset design in achieving high segmentation performance. Dataset balance and preprocessing techniques like Hidden Point Removal (HPR) play a crucial role in improving model accuracy. Balanced datasets ensure that smaller or underrepresented classes receive sufficient attention during training, reducing biases toward more prevalent objects. The results show that a balanced dataset improved the model's mean accuracy from 45.71% to 55.10%, and when combined with HPR, it further increased to 57.02%. However, HPR's impact varied between classes: while it improved the performance of simpler objects like floors and walls, it reduced the accuracy of slender objects such as beams and pipes by inadvertently removing critical geometric points.

- Higher point cloud density correlates with improved accuracy for all object classes, particularly for geometrically complex or smaller objects like chairs and beams.
- Noise negatively impacts segmentation accuracy, especially for thin or elongated objects such as beams and pipes, where even slight perturbations in geometry lead to misclassification.
- Dataset balancing enhances model generalization by providing equal representation of all object classes, mitigating biases toward overrepresented objects.
- Hidden Point Removal (HPR) benefits some classes, such as floors and walls, by reducing noise, but can decrease performance for slender objects by removing essential points critical for their geometric representation.

Overall, the results demonstrate that both the object-finding methodology and the careful design of datasets play key roles in improving the performance of neural networks in point cloud segmentation and object detection. The combination of these techniques significantly improves the model's ability to generalize across different industrial environments, enhancing its utility in real-world applications where noise, occlusion, and data imbalance are common challenges. However, further research is needed to refine the framework’s handling of noise, occlusions, and complex scenes, as well as to develop class-specific preprocessing techniques that can improve segmentation accuracy for slender or geometrically complex objects. With these improvements, the methodology has the potential to become a valuable tool for real-world industrial applications, where accurate and efficient object detection is critical.

5.3 Future Work

There are several promising directions for future exploration. One key area for improvement is the exploration of more advanced clustering algorithms to handle large numbers of point groups more effectively. While DBSCAN was used successfully in this thesis, future research could investigate clustering methods that are faster to manage even larger point cloud objects. This would prevent the overshooting of point cloud counts and ensure that the clusters generated are more consistent with and even larger template point cloud sizes. Improved clustering could enhance the accuracy of object segmentation and detection, especially in more complex industrial scenes.

Another important avenue for future work is the implementation of parallel processing techniques to increase efficiency. By allowing the GPU to handle candidate generation while the ICP algorithm selects the best fit, the overall processing time could be significantly reduced. This parallel approach would be particularly beneficial for large-scale industrial applications where quick processing of vast amounts of data is crucial. Enhancing the processing speed without compromising accuracy would make the methodology even more applicable to real-time scenarios, where rapid object detection and segmentation are required. In addition to these technical improvements, future research could also benefit from integrating emerging segmentation networks as they continue to evolve. Machine learning-based segmentation techniques are constantly advancing, and incorporating the latest innovations into the existing object-finding methodology could further boost its performance. By leveraging new architectures and algorithms, the framework could achieve even higher accuracy and robustness, keeping it at the forefront of neural network applications in industrial environments.

Expanding the complexity and variability of synthetic datasets is another valuable direction for future research. While this thesis developed a balanced synthetic dataset tailored to industrial scenes, increasing the diversity of objects, environments, and conditions within the dataset could further improve the generalizability of the models. Simulating a broader range of industrial objects, incorporating more environmental variables like lighting and noise, or even generating dynamic scenes where objects move in real time could provide the trained models with more comprehensive exposure to the challenges of real-world industrial settings. For instance, neural radiance fields (NeRFs) can be used to achieve photo-realistic rendering of synthetic datasets. Applying this approach to the procedural generation pipeline could significantly enhance the realism of synthetic datasets by incorporating lighting, texture, and radiance effects derived from real-world scenes. Such advancements would help bridge the gap between synthetic and real-world datasets, improving model performance across varied industrial scenarios (Hu, 2023). The real-time implementation of this methodology in live industrial environments represents a natural progression of this work. By deploying the system in real-time settings, future research can address practical challenges such as dynamic object movements, fluctuating lighting conditions, and the need for immediate data processing. These real-world tests would validate the framework's scalability and adaptability, providing insights into how it can be optimized for deployment in industrial automation, robotics, and monitoring systems.

Another promising direction is cross-domain adaptation, which would involve generating incrementally larger and more diverse datasets to further expand the applicability of the trained models across different industries or applications. By fine-tuning the models for various industrial environments or applying them to non-industrial domains such as construction, urban planning, or autonomous systems, the versatility and impact of the framework could be significantly broadened. Expanding the dataset size and diversity would allow the models to capture more nuanced variations in different domains, ensuring that they generalize effectively across a wide range of applications.

Finally, improving techniques for handling noise and occlusions in point cloud data remains a crucial area for further investigation. Future research could focus on not only advanced filtering methods or point cloud reconstruction algorithms but also on analyzing the impact of noise and determining how close synthetic noise needs to be to real-world noise to be optimally effective. By generating datasets with progressively more realistic noise patterns, researchers can explore the threshold at which synthetic noise approximates real-world conditions sufficiently to enhance model training. This would ensure that the

methodology remains robust in highly cluttered or visually noisy environments, where obtaining clean data is often impractical.

References

Abreu, N., Pinto, A., Matos, A., & Pires, M. (2023). Procedural Point Cloud Modelling in Scan-to-BIM and Scan-vs-BIM Applications: A Review. ISPRS International Journal of Geo-Information, 12(7), 260. https://doi.org/10.3390/ijgi12070260

Agapaki, E., & Brilakis, I. (2021). CLOI: An automated benchmark framework for generating geometric digital twins of industrial facilities. Journal of Construction Engineering and Management, 147(11). https://doi.org/10.1061/(ASCE)CO.1943-7862.0002171

Agapaki, E., Miatt, G., & Brilakis, I. (2018). Prioritizing object types for modelling existing industrial facilities. Automation in Construction, 96, 211–223. https://doi.org/10.1016/j.autcon.2018.09.011

Agarwal, P. K., & Matousek, J. (1992). Relative neighborhood graphs in three dimensions. Technical report, University of Kentucky.

Agarwal, S., & Bhowmick, B. (2017). 3d point cloud registration with shape constraint. In 2017 IEEE International Conference on Image Processing (ICIP), 2199-2203. https://doi.org/10.1109/ICIP.2017.8296673

Ahmed, M. F., Haas, C. T., & Haas, R. (2014). Automatic Detection of Cylindrical Objects in Built Facilities. *Journal of Computing in Civil Engineering*, *28*(3), 04014009. doi:10.1061/(ASCE)CP.1943-5487.0000329

Aijazi, A. K., Malaterre, L., Trassoudaine, L., Chateau, T., & Checchin, P. (2019). Automatic detection and modeling of underground pipes using a portable 3d lidar system. Sensors, 19(19), 4285. https://doi.org/10.3390/s19194285

Al-Sabbag, Z. A., Yeum, C. M., & Narasimhan, S. (2024). Distributed collaborative inspections through smart infrastructure metaverse. Automation in Construction, 165, 105503. https://doi.org/10.1016/j.autcon.2024.105503

Antova, G. (2019). Application of areal change detection methods using point clouds data. IOP Conference Series: Earth and Environmental Science, 221, 012082. https://doi.org/10.1088/1755-1315/221/1/012082

Aoki, Y., Goforth, H., Srivatsan, R. A., & Lucey, S. (2019). PointNetLK: Robust & efficient point cloud registration using PointNet. 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 7156-7165. https://doi.org/10.1109/cvpr.2019.00733

Armeni, I., Sax, A., Zamir, A. R., & Savarese, S. (2017). Joint 2D-3D-Semantic Data for Indoor Scene Understanding. https://arxiv.org/abs/1702.01105

Arnx, A. (2019). First neural network for beginners explained (with code).

Arun, K. S., Huang, T. S., & Blostein, S. D. (1987). Least-Squares Fitting of Two 3-D Point Sets. IEEE Transactions on Pattern Analysis and Machine Intelligence, PAMI-9(5), 698–700. https://doi.org/10.1109/TPAMI.1987.4767965

Arya, S., Mount, D. M., Netanyahu, N. S., Silverman, R., & Wu, A. Y. (1998). An optimal algorithm for approximate nearest neighbor searching fixed dimensions. Journal of the ACM, 45(6), 891-923. https://doi.org/10.1145/293347.293348

Autodesk. (2021). What is generative design: Tools software.

Bai, X., Luo, Z., Zhou, L., Fu, H., Quan, L., & Tai, C. L. (2020). D3feat: Joint learning of dense detection and description of 3d local features. arXiv preprint arXiv:2003.03164.

Banks, L. (2014). Procedurally modeling hoses and pipes in houdini.

Barrile, V., Candela, G., & Fotia, A. (2019). Point cloud segmentation using image processing techniques for structural analysis. The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences, 42, 187–193. https://doi.org/10.5194/isprs-archives-XLII-2-W11-187-2019

Bauer, U., & Polthier, K. (2009). Generating parametric models of tubes from laser scans. Computer-Aided Design, 41(10), 719-729. https://doi.org/10.1016/j.cad.2009.01.002

Beckley, M., Baker, S., Crosby, C. J., & Okal, M. H. (2018). A Terrestrial Laser Scanning Data Archive for Multi-Temporal Mapping Across Research Disciplines. AGU Fall Meeting Abstracts, 2018, G51C-0491.

Ben-Shabat, Y., Lindenbaum, M., & Fischer, A. (2017). 3d point cloud classification and segmentation using 3d modified fisher vector representation for convolutional neural networks. arXiv preprint arXiv:1711.08241.

Bergsjo, J. (2016). Photogrammetric point cloud generation and surface interpolation for change detection. PhD thesis, School of Architecture and the Built Environment (ABE).

Birdal, T., & Ilic, S. (2015). Point pair features based object detection and pose estimation revisited. In 2015 International Conference on 3D Vision, 527-535. https://doi.org/10.1109/3DV.2015.66

Bosche, F., & Haas, C. T. (2008). Automated retrieval of 3d cad model objects in construction range images. Automation in Construction, 17(4), 499-512. https://doi.org/10.1016/j.autcon.2007.09.001

Bosche, F., Ahmed, M., Turkan, Y., Haas, C. T., & Haas, R. (2015). The value of integrating scan-to-bim and scan-vs-bim techniques for construction monitoring using laser scanning and bim: The case of cylindrical mep components. Automation in Construction, 49, 201-213. https://doi.org/10.1016/j.autcon.2014.05.014

Boulch, A. (2020). Convpoint: Continuous convolutions for point cloud processing. Computers & Graphics, 88, 24-34. https://doi.org/10.1016/j.cag.2020.03.004

Boulch, A., Puy, G., & Marlet, R. (2020). FKAConv: Feature-Kernel Alignment for Point Cloud Convolution. In 15th Asian Conference on Computer Vision (ACCV 2020).

Bowers, A. W. (1964). Character recognition by area measurement. PhD thesis, Georgia Institute of Technology.

Bowers, J., Wang, R., Wei, L. Y., & Maletz, D. (2010). Parallel poisson disk sampling with spectrum analysis on surfaces. ACM Transactions on Graphics, 29(6), 166. https://doi.org/10.1145/1882261.1866188

Braun, A., Tuttas, S., Borrmann, A., & Stilla, U. (2015). A concept for automated construction progress monitoring using bim-based geometric constraints and photogrammetric point clouds. Journal of Information Technology in Construction, 20, 68-79.

Brock, A., Lim, T., Ritchie, J. M., & Weston, N. (2016). Generative and discriminative voxel modeling with convolutional neural networks. arXiv preprint arXiv:1608.04236.

Brock, A., Lim, T., Ritchie, J. M., & Weston, N. (2017). Neural photo editing with introspective adversarial networks. arXiv preprint arXiv:1609.07093.

Bronstein, M. M., Bruna, J., Cohen, T., & Velickovic, P. (2021). Geometric deep learning: Grids, groups, graphs, geodesics, and gauges. arXiv preprint arXiv:2104.13478.

Campbell, R. J., & Flynn, P. J. (2001). A survey of free-form object representation and recognition techniques. Computer Vision and Image Understanding, 81(2), 166-210. https://doi.org/10.1006/cviu.2000.0889

Cazorla, R., Poinel, L., Papadakis, P., & Buche, C. (2021). Bottleneck identification to semantic segmentation of industrial 3d point cloud scene via deep learning. In Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21, 4877-4878.

Cheng, G., & Han, J. (2016). A survey on object detection in optical remote sensing images. ISPRS Journal of Photogrammetry and Remote Sensing, 117, 11-28. https://doi.org/10.1016/j.isprsjprs.2016.03.014

Chin, R. T., & Dyer, C. (1986). Model-based recognition in robot vision. ACM Computing Surveys, 18(1), 67-108. https://doi.org/10.1145/6462.6464

Choi, M., Kim, S., & Kim, S. (2024). Semi-automated visualization method for visual inspection of buildings on bim using 3d point cloud. Journal of Building Engineering, 81, 108017. https://doi.org/10.1016/j.jobe.2023.108017

Choy, C., Gwak, J., & Savarese, S. (2019). 4D Spatio-Temporal ConvNets: Minkowski Convolutional Neural Networks. 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 3070-3079. https://doi.org/10.1109/cvpr.2019.00319

Choy, C., Park, J., & Koltun, V. (2019). Fully Convolutional Geometric Features. 2019 IEEE/CVF International Conference on Computer Vision (ICCV), 8957–8965. https://doi.org/10.1109/ICCV.2019.00905

Cline, D., Jeschke, S., White, K., Razdan, A., & Wonka, P. (2009). Dart throwing on surfaces. Computer Graphics Forum, 28, 1217-1226. https://doi.org/10.1111/j.1467-8659.2009.01499.x

Convercon. (2021). Generative design.

Cook, R. L. (1986). Stochastic sampling in computer graphics. ACM Transactions on Graphics, 5(1), 51-72. https://doi.org/10.1145/7529.8927

Corsini, M., Cignoni, P., & Scopigno, R. (2012). Efficient and Flexible Sampling with Blue Noise Properties of Triangular Meshes. IEEE Transactions on Visualization and Computer Graphics, 18, 914–924. https://doi.org/10.1109/TVCG.2012.34

Cuturi, M. (2013). Sinkhorn distances: Lightspeed computation of optimal transportation distances. arXiv preprint arXiv:1306.0895.

Dai, A., Chang, A. X., Savva, M., Halber, M., Funkhouser, T., & Nießner, M. (2017). ScanNet: Richly-annotated 3D reconstructions of indoor scenes. In IEEE Conference on Computer Vision and Pattern Recognition (CVPR).

Dai, J., Qi, H., Xiong, Y., Li, Y., Zhang, G., Hu, H., & Wei, Y. (2017). Deformable convolutional networks. arXiv preprint arXiv:1703.06211.

Danielczuk, M., Matl, M., Gupta, S., Li, A., Lee, A., Mahler, J., & Goldberg, K. (2019). Segmenting unknown 3d objects from real depth images using mask r-cnn trained on synthetic data. In 2019 International Conference on Robotics and Automation (ICRA), 7283-7290. https://doi.org/10.1109/ICRA.2019.8793744

Dantanarayana, H. G., & Huntley, J. M. (2017). Object recognition and localization from 3D point clouds by maximum-likelihood estimation. Royal Society Open Science, 4(8), 170670. https://doi.org/10.1098/rsos.170670

de Queiroz, T. A., & Miyazawa, F. K. (2013). Approaches for the 2d 0-1 knapsack problem with conflict graphs. In 2013 XXXIX Latin American Computing Conference (CLEI), 1-8. https://doi.org/10.1109/CLEI.2013.6670664

Dey, S., Dutta, A., Toledo, J. I., Ghosh, S. K., Llados, J., & Pal, U. (2017). Signet: Convolutional siamese network for writer independent offline signature verification. arXiv preprint arXiv:1707.02131.

Dimitrov, A., & Golparvar-Fard, M. (2015). Segmentation of building point cloud models including detailed architectural/structural features and MEP systems. Automation in Construction, 51, 32-45. https://doi.org/10.1016/j.autcon.2014.12.015

Diniz, R., Freitas, P. G., & Farias, M. C. Q. (2021). Color and Geometry Texture Descriptors for Point-Cloud Quality Assessment. IEEE Signal Processing Letters, 28, 1150-1154. https://doi.org/10.1109/LSP.2021.3088059

Dong, J., Liu, J., Yao, K., Chantler, M., Qi, L., Yu, H., & Jian, M. (2020). Survey of procedural methods for two-dimensional texture generation. Sensors, 20(4), 1135. https://doi.org/10.3390/s20041135

Drost, B., Ulrich, M., Navab, N., & Ilic, S. (2010). Model globally, match locally: Efficient and robust 3D object recognition. 2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 998-1005. https://doi.org/10.1109/cvpr.2010.5540108

Dyer, C. R., & Chin, R. T. (1984). Model-based industrial part recognition: Systems and algorithms. Technical Report TR-EE 84-15, School of Electrical Engineering, Purdue University.

Eapen, N. G., Samanta, D., Kaur, M., Al-Amri, J. F., & Masud, M. (2021). Elementary methods for generating three-dimensional coordinate estimation and image reconstruction from series of two-dimensional images. Mathematical Problems in Engineering, 2021, 1-11. https://doi.org/10.1155/2021/6687520

Edelsbrunner, H., Kirkpatrick, D., & Seidel, R. (1983). On the shape of a set of points in the plane. IEEE Transactions on Information Theory, 29(4), 551–559. https://doi.org/10.1109/tit.1983.1056714

Esser, P., Kulal, S., Blattmann, A., Entezari, R., Müller, J., Saini, H., Levi, Y., Lorenz, D., Sauer, A., Boesel, F., Podell, D., Dockhorn, T., English, Z., Lacey, K., Goodwin, A., Marek, Y., & Rombach, R. (2024). Scaling rectified flow transformers for high-resolution image synthesis. arXiv preprint arXiv:2402.08958.

Ester, M., Kriegel, H.-P., Sander, J., & Xu, X. (1996). A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. Proceedings of the Second International Conference on Knowledge Discovery and Data Mining, 226–231.

Fahmy, H., & Blostein, D. (1992). A survey of graph grammars: theory and applications. In Proceedings., 11th IAPR International Conference on Pattern Recognition. Vol.II. Conference B: Pattern Recognition Methodology and Systems, 294-298. https://doi.org/10.1109/ICPR.1992.201782

Fan, S., Dong, Q., Zhu, F., Lv, Y., Ye, P., & Wang, F. Y. (2021). SCF-Net: Learning spatial contextual features for large-scale point cloud segmentation. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 14504-14513.

Farmakis, D. (2021). What is generative construction?

Faugeras, O. D., & Hebert, M. (1983). A 3-d recognition and positioning algorithm using geometrical matching between primitive surfaces. In IJCAI.

Fei, B., Yang, W., Chen, W.-M., Li, Z., Li, Y., Ma, T., Hu, X., & Ma, L. (2022). Comprehensive review of deep learning-based 3d point cloud completion processing and analysis. IEEE Transactions on Intelligent Transportation Systems, 23(12), 22862–22883. https://doi.org/10.1109/TITS.2022.3150497

Feng, X., Liu, Y., & Dai, Q. (2009). Point-cloud refinement via specific matching. In 2009 IEEE International Conference on Multimedia and Expo, 926-929. https://doi.org/10.1109/ICME.2009.5202647

Fischer, K., Simon, M., Olsner, F., Milz, S., Gros, H.-M., & Mader, P. (2021). Stickypillars: Robust and efficient feature matching on point clouds using graph neural networks. 2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 313-323. https://doi.org/10.1109/cvpr46437.2021.00038

Fischler, M. A., & 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. https://doi.org/10.1145/358669.358692

Fritsch, J., Kuehnl, T., & Geiger, A. (2013). A new performance measure and evaluation benchmark for road detection algorithms. In International Conference on Intelligent Transportation Systems (ITSC).

Fu, K. C. (1968). Sequential methods in pattern recognition and machine learning. Academic press.

Fu, Y., & Zhou, B. (2008). Direct sampling on surfaces for high quality remeshing. In Proceedings of the 2008 ACM Symposium on Solid and Physical Modeling, 115-124. https://doi.org/10.1145/1364901.1364920

Gao, J., Chen, W., Xiang, T., Tsang, C. F., Jacobson, A., McGuire, M., & Fidler, S. (2020). Learning Deformable Tetrahedral Meshes for 3D Reconstruction. arXiv preprint arXiv:2011.01437.

Gao, R., Li, M., Yang, S. J., & Cho, K. (2022). Reflective Noise Filtering of Large-Scale Point Cloud Using Transformer. Remote Sensing, 14(3), 577. https://doi.org/10.3390/rs14030577

Gaschnig, J. (1977). A general backtrack algorithm that eliminates most redundant tests. In IJCAI.

Gelfand, N., Mitra, N. J., Guibas, L. J., & Pottmann, H. (2005). Robust global registration. In Symposium on Geometry Processing, 197-206.

Ghomi, H. M. (2013). Three-dimensional knapsack problem with pre-placed boxes and vertical stability. Master's thesis, University of Waterloo.

Gojcic, Z., Zhou, C., Wegner, J. D., & Wieser, A. (2018). The Perfect Match: 3D Point Cloud Matching with Smoothed Densities. arXiv preprint arXiv:1811.06879.

Gomes, R. B., da Silva, B. M. F., Rocha, L. K. D. M., Aroca, R. V., Velho, L. C. P. R., & Gonçalves, L. M. G. (2013). Efficient 3D object recognition using foveated point clouds. Computers and Graphics, 37(5), 496-508. https://doi.org/10.1016/j.cag.2013.03.005

Gordon, D., & Reynolds, R. A. (1985). Image space shading of 3-dimensional objects. Computer Vision, Graphics, and Image Processing, 29(3), 361-376. https://doi.org/10.1016/0734-189X(85)90160-1

Greenspan, M., & Boulanger, P. (1999). Efficient and reliable template set matching for 3D object recognition. In Second International Conference on 3-D Digital Imaging and Modeling, 230-239. https://doi.org/10.1109/IM.1999.805353

Griffiths, D., & Boehm, J. (2019). A Review on Deep Learning Techniques for 3D Sensed Data Classification. Remote Sensing, 11(12), 1499. https://doi.org/10.3390/rs11121499

Guan, W., Li, W., & Ren, Y. (2018). Point cloud registration based on improved icp algorithm. In 2018 Chinese Control And Decision Conference (CCDC), 1461-1465. https://doi.org/10.1109/CCDC.2018.8407354

Guarda, A. F. R., Pereira, F., & Rodrigues, N. M. M. (2020). Adaptive Deep Learning-Based Point Cloud Geometry Coding. IEEE Journal of Selected Topics in Signal Processing, 15(2), 415-430. https://doi.org/10.1109/jstsp.2020.3047521

Guo, G., Wang, H., Bell, D., Bi, Y., & Greer, K. (2003). KNN Model-Based Approach in Classification. In On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE, 986-996. https://doi.org/10.1007/978-3-540-39964-3_62

Hackel, T., Savinov, N., Ladicky, L., Wegner, J. D., Schindler, K., & Pollefeys, M. (2017). Semantic3d.net: A new large-scale point cloud classification benchmark. ISPRS Annals of the

Photogrammetry, Remote Sensing and Spatial Information Sciences, IV-1-W1, 91–98. https://doi.org/10.5194/isprs-annals-IV-1-W1-91-2017

Hagelskjær, F., & Buch, A. G. (2019). PointPoseNet: Accurate object detection and 6 DoF pose estimation in point clouds. arXiv preprint arXiv:1912.09057.

Hahn, E., Bose, P., & Whitehead, A. (2006). Persistent realtime building interior generation. In Proceedings of the 2006 ACM SIGGRAPH Symposium on Videogames, 179-186. https://doi.org/10.1145/1183316.1183338

Han, B., Ma, J. W., & Leite, F. (2021). A framework for semi-automatically identifying fully occluded objects in 3D models: Towards comprehensive construction design review in virtual reality. Advanced Engineering Informatics, 50, 101398. https://doi.org/10.1016/j.aei.2021.101398

Hassan, M. (2018). VGG16 - convolutional network for classification and detection.

Hassan, M. (2019). Residual CNNs for image classification tasks.

Haworth, A. (n.d.). Procedural placement (pipes).

He, K., Zhang, X., Ren, S., & Sun, J. (2015). Deep residual learning for image recognition. arXiv preprint arXiv:1512.03385.

Hellinga, N. (2019). Nathan-hellinga/subway-map-generator: A processing.py project to generate beautiful subway maps.

Hemami, H., McGhee, R. B., & Gardner, S. R. (1970). Towards a generalized template matching algorithm for pictorial pattern recognition. In 1970 IEEE Symposium on Adaptive Processes (9th) Decision and Control, 23-23. https://doi.org/10.1109/SAP.1970.269991

Hidalgo, J. L., Abad, F., Camahort, E., & Vicent, M. J. (2008). Procedural Graphics Model and Behavior Generation. In Computer Vision/Computer Graphics Collaboration Techniques (pp. 106-115). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-69387-1_12

Hinterstoisser, S., Holzer, S., Cagniart, C., Ilic, S., Konolige, K., Navab, N., & Lepetit, V. (2011). Multimodal templates for real-time detection of texture-less objects in heavily cluttered scenes. In 2011 International Conference on Computer Vision, 858-865. https://doi.org/10.1109/ICCV.2011.6126326

Hinterstoisser, S., Lepetit, V., Rajkumar, N., & Konolige, K. (2017). Going further with point pair features. arXiv preprint arXiv:1711.04061.

Hodaň, T., Michel, F., Brachmann, E., Kehl, W., Buch, A. G., Kraft, D., Drost, B., Vidal, J., Ihrke, S., Zabulis, X., Sahin, C., Manhardt, F., Tombari, F., Kim, T.-K., Matas, J., & Rother, C. (2018). BOP: Benchmark for 6D object pose estimation. Computer Vision (ECCV), 19-35. https://doi.org/10.1007/978-3-030-01249-6_2

Hu, Q., Yang, B., Xie, L., Rosa, S., Guo, Y., Wang, Z., Trigoni, N., & Markham, A. (2020). RandLA-Net: Efficient semantic segmentation of large-scale point clouds. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.

Hu, Q., Yang, B., Xie, L., Rosa, S., Guo, Y., Wang, Z., Trigoni, N., & Markham, A. (2021). Learning semantic segmentation of large-scale point clouds with random sampling. IEEE Transactions on Pattern Analysis and Machine Intelligence.

Hu, T., & Kahng, A. (2016). The Knapsack Problem. In Handbook of Combinatorial Optimization (pp. 87-101). Springer, Cham. https://doi.org/10.1007/978-3-319-32002-5_7

Hu, Z., & Brilakis, I. (2023). PriSeg: IFC-Supported Primitive Instance Geometry Segmentation with Unsupervised Clustering. In Computer Vision – ECCV 2022 Workshops. ECCV 2022. Lecture Notes in Computer Science, vol 13807. Springer, Cham. https://doi.org/10.1007/978-3-031-25082-8_13

Huang, J., & You, S. (2013). Detecting objects in scene point cloud: A combinational approach. In Proceedings - 2013 International Conference on 3D Vision (3DV), 175-182. https://doi.org/10.1109/3DV.2013.31

Huang, Q., Wang, W., & Neumann, U. (2018). Recurrent slice networks for 3D segmentation of point clouds. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR).

Huang, X., Mei, G., Zhang, J., & Abbas, R. (2021). A comprehensive survey on point cloud registration. arXiv preprint arXiv:2103.02690.

Ilčík, M., Musialski, P., Auzinger, T., & Wimmer, M. (2015). Layer-based procedural design of façades. Computer Graphics Forum, 34(2), 205-216. https://doi.org/10.1111/cgf.12556

Ioannou, Y., Taati, B., Harrap, R., & Greenspan, M. (2012). Difference of normals as a multi-scale operator in unorganized point clouds. In 2012 Second International Conference on 3D Imaging, Modeling, Processing, Visualization & Transmission, 501-508. https://doi.org/10.1109/3DIMPVT.2012.12

Javaid, M., Haleem, A., Suman, R., & Singh, R. P. (2021). Significant Applications of Big Data in Industry 4.0. Journal of Industrial Integration and Management, 06(04), 429-447. https://doi.org/10.1142/s2424862221500135

Ježek, B., Ouhrabka, A., & Slabý, A. (2020). Procedural Content Generation via Machine Learning in 2D Indoor Scene. In Information Systems Development: Information Systems Beyond 2020 (ISD2019 Proceedings). Springer. https://doi.org/10.1007/978-3-030-58465-8_3

Jia, D., Guo, J., Han, K., Wu, H., Zhang, C., Xu, C., & Chen, X. (2024). GeminiFusion: Efficient Pixel-wise Multimodal Fusion for Vision Transformer. arXiv preprint arXiv:2406.01210.

Jiao, L., Zhang, F., Liu, F., Yang, S., Li, L., Feng, Z., & Qu, R. (2019). A survey of deep learning-based object detection. IEEE Access, 7, 128837-128868. https://doi.org/10.1109/ACCESS.2019.2939201

Johnson, J. M., & Khoshgoftaar, T. M. (2019). Survey on deep learning with class imbalance. Journal of Big Data, 6(1), 27. https://doi.org/10.1186/s40537-019-0192-5

Jurie, F., & Dhome, M. (2002). Hyperplane approximation for template matching. IEEE Transactions on Pattern Analysis and Machine Intelligence, 24(7), 996-1000. https://doi.org/10.1109/TPAMI.2002.1017623

Kamousi, P., Lazard, S., Maheshwari, A., & Wuhrer, S. (2016). Analysis of farthest point sampling for approximating geodesics in a graph. Computational Geometry, 57, 1-7. https://doi.org/10.1016/j.comgeo.2016.04.001

Katsoulas, D. (2003). Robust extraction of vertices in range images by constraining the hough transform. In IbPRIA.

Kelkar, A., Jagtap, S., & Dahibhate, M. (2022). Procedural Foliage Generation. International Journal for Research in Applied Science and Engineering Technology, 10(5), 1628-1632. https://doi.org/10.22214/ijraset.2022.42410

Kelly, G., & McCabe, H. (2006). A survey of procedural techniques for city generation. The ITB Journal, 7(2), 5.

Khan, S. (2019). Towards synthetic dataset generation for semantic segmentation networks.

Khan, S., Phan, B., Salay, R., & Czarnecki, K. (2019). Procsy: Procedural synthetic dataset generation towards influence factor studies of semantic segmentation networks. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops.

Kim, H., et al. (2020). Deep-Learning-Based Retrieval of Piping Component Catalogs for Plant 3D CAD Model Reconstruction. Computers in Industry, 123, 103320. https://doi.org/10.1016/j.compind.2020.103320

Ko, C.-H. (2010). RFID 3D location sensing algorithms. Automation in Construction, 19(5), 588–595. https://doi.org/10.1016/j.autcon.2010.02.003

Kossaifi, J., Bulat, A., Tzimiropoulos, G., & Pantic, M. (2019). T-net: Parametrizing fully convolutional nets with a single high-order tensor. arXiv preprint arXiv:1904.08170.

Ku, T., Galanakis, S., Boom, B., Veltkamp, R. C., Bangera, D., Gangisetty, S., Stagakis, N., Arvanitis, G., & Moustakas, K. (2021). SHREC 2021: 3D point cloud change detection for street scenes. Computers & Graphics, 99, 192-200. https://doi.org/10.1016/j.cag.2021.06.014

Lague, D., Brodu, N., & Leroux, J. (2013). Accurate 3D comparison of complex topography with terrestrial laser scanner: Application to the Rangitikei canyon (N-Z). ISPRS Journal of Photogrammetry and Remote Sensing, 82, 10-26. https://doi.org/10.1016/j.isprsjprs.2013.04.009

Lahoud, J., & Ghanem, B. (2020). RGB-based Semantic Segmentation Using Self-Supervised Depth Pre-Training. arXiv preprint arXiv:2002.02200.

Lai, X., Liu, J., Jiang, L., Wang, L., Zhao, H., Liu, S., Qi, X., & Jia, J. (2022). Stratified transformer for 3D point cloud segmentation. arXiv preprint arXiv:2203.14508.

Landrieu, L., & Simonovsky, M. (2018). Large-scale point cloud semantic segmentation with superpoint graphs. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR).

Lang, A. H., Vora, S., Caesar, H., Zhou, L., Yang, J., & Beijbom, O. (2018). PointPillars: Fast Encoders for Object Detection from Point Clouds. arXiv preprint arXiv:1812.05784.

Larsen, P. M., Schmidt, S., & Schiøtz, J. (2016). Robust structural identification via polyhedral template matching. Modelling and Simulation in Materials Science and Engineering, 24(5), 055007. https://doi.org/10.1088/0965-0393/24/5/055007

Lauer, M., Lange, S., & Riedmiller, M. (2005). Calculating the perfect match: an efficient and accurate approach for robot self-localization. In Robot Soccer World Cup, 142-153. Springer.

Le, T., & Duan, Y. (2018). PointGrid: A deep network for 3d shape understanding. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR).

Li, C., Zhang, C., Waghwase, A., Lee, L.-H., Rameau, F., Yang, Y., Bae, S.-H., & Hong, C. S. (2023). Generative AI Meets 3D: A Survey on Text-to-3D in AIGC Era. arXiv preprint arXiv:2305.06131.

Li, G., Muller, M., Thabet, A., & Ghanem, B. (2019). DeepGCNs: Can GCNs Go as Deep as CNNs? In Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV).

Li, W., Cheng, H., & Zhang, X. (2021). Efficient 3D Object Recognition from Cluttered Point Cloud. Sensors, 21(17), 5850. https://doi.org/10.3390/s21175850

Li, Y., Bu, R., Sun, M., & Chen, B. (2018). PointCNN. arXiv preprint arXiv:1801.07791.

Liang, M., Yang, B., Hu, R., Chen, Y., Liao, R., Feng, S., & Urtasun, R. (2020). Learning Lane Graph Representations for Motion Forecasting. arXiv preprint arXiv:2007.13732.

Lin, C., & Chang, H.-J. (2011). Identification of pressurized water reactor transient using template matching. Annals of Nuclear Energy, 38(7), 1662-1666. https://doi.org/10.1016/j.anucene.2011.03.012

Linh, T., & Hiroshi, H. (2015). Global Iterative Closet Point Using Nested Annealing for Initialization. Procedia Computer Science, 60, 381–390. https://doi.org/10.1016/j.procs.2015.08.147

Liu, B., Zhang, T., Niu, D., Lin, J., Lai, K., & Xu, Y. (2018). Matching long text documents via graph convolutional networks. arXiv preprint arXiv:1802.07459.

Liu, B., Wang, M., Foroosh, H., Tappen, M., & Penksy, M. (2015). Sparse convolutional neural networks. In 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 806-814. https://doi.org/10.1109/CVPR.2015.7298681

Liu, J., Xu, D., Hyyppä, J., & Liang, Y. (2021). A survey of applications with combined BIM and 3D laser scanning in the life cycle of buildings. IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing, 14, 5627-5637. https://doi.org/10.1109/JSTARS.2021.3084311

Liu, J., et al. (2023). Real3D-AD: A Dataset of Point Cloud Anomaly Detection. arXiv preprint arXiv:2309.13226.

Liu, K., Gao, Z., Lin, F., & Chen, B. M. (2020). FG-Net: Fast Large-Scale LiDAR Point Clouds Understanding Network Leveraging Correlated Feature Mining and Geometric-Aware Modelling. arXiv preprint arXiv:2012.09439.

Liu, L., Ouyang, W., Wang, X., Fieguth, P., Chen, J., Liu, X., & Pietikäinen, M. (2019). Deep learning for generic object detection: A survey. International Journal of Computer Vision, 128(2), 261-318. https://doi.org/10.1007/s11263-019-01247-4

Liu, Z., Song, B., Guo, Y., & Xu, H. (2020). Improved Template Matching Based Stereo Vision Sparse 3D Reconstruction Algorithm. 2020 Chinese Control And Decision Conference (CCDC), 4305–4310. https://doi.org/10.1109/CCDC49329.2020.9164629

Liu, Z., Tang, H., Lin, Y., & Han, S. (2019). Point-voxel CNN for efficient 3d deep learning. arXiv preprint arXiv:1907.03739.

Lopez, D., Haas, C., & Narasimhan, S. (2023). Specific object finding in point clouds based on semantic segmentation and iterative closest point. Automation in Construction, 156, 105116. https://doi.org/10.1016/j.autcon.2023.105116

Lopez, D., Haas, C., & Narasimhan, S. (2024). Training Networks to Segment Industrial Scene Point Clouds: Dataset Generation Using TLS and Synthetic Data. Unpublished manuscript, Department of Civil & Environmental Engineering, University of Waterloo.

Lu, W., Wan, G., Zhou, Y., Fu, X., Yuan, P., & Song, S. (2019). DeepVCP: An End-to-End Deep Neural Network for Point Cloud Registration. 2019 IEEE/CVF International Conference on Computer Vision (ICCV), 12–21. https://doi.org/10.1109/ICCV.2019.00010

Lucas, B. D., & Kanade, T. (1981). An Iterative Image Registration Technique with an Application to Stereo Vision. IJCAI'81: 7th international joint conference on Artificial intelligence, 2, 674-679.

Lucanin, D. (2012). Visual definition of procedures for automatic virtual scene generation. arXiv preprint arXiv:1202.2868.

Ma, J. W., Czerniawski, T., & Leite, F. (2020). Semantic segmentation of point clouds of building interiors with deep learning: Augmenting training datasets with synthetic BIM-based point clouds. Automation in Construction, 113, 103144. https://doi.org/10.1016/j.autcon.2020.103144

Ma, J. W., Han, B., & Leite, F. (2021). An automated framework for generating synthetic point clouds from as-built BIM with semantic annotation for scan-to-BIM. 2021 Winter Simulation Conference (WSC), 1-10. IEEE. https://doi.org/10.1109/WSC52266.2021.9715301

Ma, J. W., & Leite, F. (2022). Performance boosting of conventional deep learning-based semantic segmentation leveraging unsupervised clustering. Automation in Construction, 136, 104167. https://doi.org/10.1016/j.autcon.2022.104167

Ma, Y., Chen, Z., Hu, W., & Wang, W. (2018). Packing irregular objects in 3d space via hybrid optimization. Computer Graphics Forum, 37(5), 49-59. https://doi.org/10.1111/cgf.13494

Ma, Y., Fan, Y., Ji, J., Wang, H., Sun, X., Jiang, G., Shu, A., & Ji, R. (2023). X-dreamer: Creating high-quality 3d content by bridging the domain gap between text-to-2d and text-to-3d generation. arXiv preprint arXiv:2310.02550.

Marani, R., Renò, V., Nitti, M., D'Orazio, T., & Stella, E. (2016). A modified iterative closest point algorithm for 3d point cloud registration. Computer-Aided Civil and Infrastructure Engineering, 31(7), 515-534. https://doi.org/10.1111/mice.12184

Marcon, M., Bellon, O. R. P., & Silva, L. (2020). Towards real-time object recognition and pose estimation in point clouds. arXiv preprint arXiv:2011.13669.

Maturana, D., & Scherer, S. (2015). VoxNet: A 3D Convolutional Neural Network for real-time object recognition. In 2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 922-928. https://doi.org/10.1109/IROS.2015.7353481

Mian, A. S., Bennamoun, M., & Owens, R. (2006). Three-dimensional model-based object recognition and segmentation in cluttered scenes. IEEE Transactions on Pattern Analysis and Machine Intelligence, 28(10), 1584-1601. https://doi.org/10.1109/TPAMI.2006.213

Michelon De Carli, D., Schetinger, V., Pozzer, C. T., & Bevilacqua, F. (2014, August 1). Procedural Generation of 3D Canyons. 2014 27th SIBGRAPI Conference on Graphics, Patterns and Images. https://doi.org/10.1109/sibgrapi.2014.41

Miao, C., Zhong, X., Zhong, X., Yang, Z., & Xu, M. (2021). Detection and Grasping of Texture-Less Objects Based on 3D Template Matching. 2021 40th Chinese Control Conference (CCC), 3943–3948. https://doi.org/10.23919/CCC52363.2021.9550615

Miller, A., Jain, V., & Mundy, J. L. (2011). Real-time rendering and dynamic updating of 3-d volumetric data. In GPGPU-4.

Millon, H. A. (1972). Rudolf Wittkower, "Architectural Principles in the Age of Humanism": Its influence on the development and interpretation of modern architecture. Journal of the Society of Architectural Historians, 31(2), 83-91. https://doi.org/10.2307/988684

Mohamed, O. M., & Jaïdane-Saïdane, M. (2009). Generalized gaussian mixture model. In 2009 17th European Signal Processing Conference, 2273-2277.

Mumuni, A., & Mumuni, F. (2022). Data Augmentation: A Comprehensive Survey of Modern Approaches. Array, 16, 100258. https://doi.org/10.1016/j.array.2022.100258

Munaga, H., & Jarugumalli, V. (2012). Performance evaluation: Ball-tree and kd-tree in the context of MST. Signal Processing and Information Technology, 225-228.

Nathan Silberman, P. K., Hoiem, D., & Fergus, R. (2012). Indoor Segmentation and Support Inference from RGBD Images. ECCV.

Noichl, F., Braun, A., & Borrmann, A. (2021). "BIM-to-Scan" for Scan-to-BIM: Generating Realistic Synthetic Ground Truth Point Clouds Based on Industrial 3D Models. In Proceedings of the 2021 European Conference on Computing in Construction.

Norrenbrock, C. (2016). Percolation threshold on planar Euclidean Gabriel graphs. European Physical Journal B, 89(5). https://doi.org/10.1140/epjb/e2016-60849-4

Notchenko, A. V., Kapushev, Y., & Burnaev, E. (2017). Large-scale shape retrieval with sparse 3d convolutional neural networks. In AIST.

Nuchter, A., Lingemann, K., & Hertzberg, J. (2007). Cached k-d tree search for ICP algorithms. In Sixth International Conference on 3-D Digital Imaging and Modeling (3DIM 2007), 419-426. https://doi.org/10.1109/3DIM.2007.15

Ogura, K., Yamada, Y., Kajita, S., Yamaguchi, H., Higashino, T., & Takai, M. (2019). Ground object recognition and segmentation from aerial image-based 3D point cloud. Computational Intelligence, 35(3), 625-642. https://doi.org/10.1111/coin.12207

Oshima, M., & Shirai, Y. (1983). Object Recognition Using Three-Dimensional Information. IEEE Transactions on Pattern Analysis and Machine Intelligence, PAMI-5(4), 353–361. https://doi.org/10.1109/TPAMI.1983.4767405

Osugi, M., & Vargas, D. V. (2022). Image generation from text and segmentation. In 2022 Tenth International Symposium on Computing and Networking Workshops (CANDARW). IEEE.

Palma, G., Cignoni, P., Boubekeur, T., & Scopigno, R. (2015). Detection of geometric temporal changes in point clouds. Computer Graphics Forum, 35(6), 33-45. https://doi.org/10.1111/cgf.12730

Pan, Y., Braun, A., Brilakis, I., & Borrmann, A. (2022). Enriching geometric digital twins of buildings with small objects by fusing laser scanning and AI-based image recognition. Automation in Construction, 140, 104375. https://doi.org/10.1016/j.autcon.2022.104375

Pan, Y., Gao, B., Mei, J., Geng, S., Li, C., & Zhao, H. (2020). SemanticPOSS: A Point Cloud Dataset with Large Quantity of Dynamic Instances. arXiv preprint arXiv:2002.09147.

Pang, G., Qiu, R., Huang, J., You, S., & Neumann, U. (2015). Automatic 3d industrial point cloud modeling and recognition. In 2015 14th IAPR International Conference on Machine Vision Applications (MVA), 22-25. https://doi.org/10.1109/MVA.2015.7153126

Paris, R. (2013). Modified half-edge data structure and its applications to 3d mesh generation for complex tube networks. Master's thesis, University of Louisville.

Park, J., Zhou, Q.-Y., & Koltun, V. (2017). Colored Point Cloud Registration Revisited. 2017 IEEE International Conference on Computer Vision (ICCV), 143–152. https://doi.org/10.1109/ICCV.2017.25

Parmar, R. (2018). Detection and segmentation through convnets.

Pau, G., Arena, F., Gebremariam, Y. E., & You, I. (2021). Bluetooth 5.1: An Analysis of Direction Finding Capability for High-Precision Location Services. Sensors, 21(11), 3589. https://doi.org/10.3390/s21113589

Peters, T., & Brenner, C. (2019). Automatic Generation of Large Point Cloud Training Datasets Using Label Transfer. arXiv preprint arXiv:1908.09269.

Petitjean, S. (2002). A survey of methods for recovering quadrics in triangle meshes. ACM Computing Surveys, 34(2), 211-262. https://doi.org/10.1145/508352.508354

Peyghambarzadeh, S. M., Azizmalayeri, F., Khotanlou, H., & Salarpour, A. (2020). Point-PlaneNet: Plane kernel based convolutional neural network for point clouds analysis. Digital Signal Processing, 98, 102633. https://doi.org/10.1016/j.dsp.2019.102633

Peyré, G., & Cuturi, M. (2018). Computational Optimal Transport. arXiv preprint arXiv:1803.00567.

Poiesi, F., & Boscaini, D. (2022). Learning general and distinctive 3D local deep descriptors for point cloud registration. IEEE Transactions on Pattern Analysis and Machine Intelligence, 45(3), 3979-3989. https://doi.org/10.1109/TPAMI.2022.3175371

Ponciano, J. J., Trémeau, A., & Boochs, F. (2019). Automatic detection of objects in 3D point clouds based on exclusively semantic guided processes. ISPRS International Journal of Geo-Information, 8(10), 442. https://doi.org/10.3390/ijgi8100442

Poullis, C. (2013). A framework for automatic modeling from point cloud data. IEEE Transactions on Pattern Analysis and Machine Intelligence, 35(11), 2563-2575. https://doi.org/10.1109/TPAMI.2013.64

Prusinkiewicz, P., Hammel, M., Hanan, J., & Mech, R. (1996). L-systems: From the theory to visual models of plants. In Proceedings of the 2nd CSIRO Symposium on Computational Challenges in Life Sciences (Vol. 3, pp. 1-32).

Pătrăucean, V., Armeni, I., Nahangi, M., Yeung, J., Brilakis, I., & Haas, C. (2015). State of research in automatic as-built modelling. Advanced Engineering Informatics, 29(2), 162-171. https://doi.org/10.1016/j.aei.2015.01.001

Qi, C. R., Su, H., Mo, K., & Guibas, L. J. (2017). PointNet: Deep learning on point sets for 3D classification and segmentation. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR).

Qi, C. R., Su, H., Nießner, M., Dai, A., Yan, M., & Guibas, L. J. (2016). Volumetric and multi-view CNNs for object classification on 3d data. arXiv preprint arXiv:1604.03265.

Qi, C. R., Yi, L., Su, H., & Guibas, L. J. (2017). PointNet++: Deep hierarchical feature learning on point sets in a metric space. arXiv preprint arXiv:1706.02413.

Qi, L., Wu, F., Ge, Z., & Sun, Y. (2022). DeepMatch: Toward Lightweight in Point Cloud Registration. Frontiers in Neurorobotics, 16, 891158. https://doi.org/10.3389/fnbot.2022.891158

Qian, G., Li, Y., Peng, H., Mai, J., Hammoud, H., Elhoseiny, M., & Ghanem, B. (2022). PointNeXt: Revisiting PointNet++ with Improved Training and Scaling Strategies. Advances in Neural Information Processing Systems (NeurIPS).

Qian, R., Lai, X., & Li, X. (2021). 3D object detection for autonomous driving: A survey. arXiv preprint arXiv:2106.10823.

Qin, G., Wang, L., Hou, Y., Gui, H., & Jian, Y. (2020). Construction and application of factory digital twin model based on BIM and point cloud. In E3S Web of Conferences (Vol. 293). EDP Sciences.

Qiu, S., Anwar, S., & Barnes, N. (2021). Semantic segmentation for real point cloud scenes via bilateral augmentation and adaptive fusion. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 1757-1767.

Qu, T., & Sun, W. (2015). Usage of 3D point cloud data in BIM (Building Information Modelling): Current applications and challenges. Journal of Civil Engineering and Architecture, 9(11), 1269-1278.

Quintana, B., Prieto, S. A., Adán, A., & Bosché, F. (2018). Door detection in 3D coloured point clouds of indoor environments. Automation in Construction, 85, 146–166. https://doi.org/10.1016/j.autcon.2017.10.016

Raistrick, A., Lipson, L., Ma, Z., Mei, L., Wang, M., Zuo, Y., Kayan, K., Wen, H., Han, B., Wang, Y., Newell, A., Law, H., Goyal, A., Yang, K., & Deng, J. (2023). Infinite photorealistic worlds using procedural generation. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 12630-12641.

Rao, Y., Zhang, M., Cheng, Z., Xue, J., Pu, J., & Wang, Z. (2021). Semantic point cloud segmentation using fast deep neural network and DCRF. Sensors, 21(8), 2808. https://doi.org/10.3390/s21082808

Riegler, G., Ulusoy, A. O., & Geiger, A. (2017). OctNet: Learning deep 3d representations at high resolutions. arXiv preprint arXiv:1611.05009.

Ringqvist, D. (2018). Perceptual evaluation of plausibility of virtual furniture layouts. PhD thesis, KTH Royal Institute of Technology.

Ronneberger, O., Fischer, P., & Brox, T. (2015). U-Net: Convolutional networks for biomedical image segmentation. arXiv preprint arXiv:1505.04597.

Roth, S. D. (1982). Ray casting for modeling solids. Computer Graphics and Image Processing, 18(2), 109-144. https://doi.org/10.1016/0146-664X(82)90169-1

Ruder, S. (2020). An overview of gradient descent optimization algorithms. arXiv preprint arXiv:1609.04747.

Rusu, R. B., Blodow, N., & Beetz, M. (2009). Fast Point Feature Histograms (FPFH) for 3D registration. 2009 IEEE International Conference on Robotics and Automation, 3212–3217. https://doi.org/10.1109/ROBOT.2009.5152473

Rutzinger, M., Rüf, B., Höfle, B., & Vetter, M. (2010). Change detection of building footprints from airborne laser scanning acquired in short time intervals. In ISPRS Technical Commission VII (Vol. 38, pp. 475-480).

Salehi, V., Wang, S., et al. (2017). Using point cloud technology for process simulation in the context of digital factory based on a systems engineering integrated approach. In DS 87-3 Proceedings of the 21st

International Conference on Engineering Design (ICED 17) Vol 3: Product, Services and Systems Design, Vancouver, Canada, 21-25.08. 2017, 011-020.

Sarode, V., Li, X., Goforth, H., Aoki, Y., Srivatsan, R. A., Lucey, S., & Choset, H. (2019). PCRNet: Point Cloud Registration Network using PointNet Encoding. arXiv preprint arXiv:1908.07906.

Schubert, E., Sander, J., Ester, M., Kriegel, H. P., & Xu, X. (2017). DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN. ACM Transactions on Database Systems, 42(3), 1-21. https://doi.org/10.1145/3068335

Sehgal, A., Cernea, D., & Makaveeva, M. (2010). Real-time scale invariant 3D range point cloud registration. In Image Analysis and Recognition (pp. 220-229). Springer Berlin Heidelberg.

Shahi, S., Wozniczka, P., Rausch, C., Trudeau, I., & Haas, C. (2021). A computational methodology for generating modular design options for building extensions. Automation in Construction, 127, 103700. https://doi.org/10.1016/j.autcon.2021.103700

Shajahan, D. A., T, M. V., & Muthuganapathy, R. (2021). Point transformer for shape classification and retrieval of 3D and ALS roof pointclouds. arXiv preprint arXiv:2106.02823.

Shapiro, L., & Haralick, R. (1981). Structural descriptions and inexact matching. IEEE Transactions on Pattern Analysis and Machine Intelligence, PAMI-3(5), 504-519. https://doi.org/10.1109/TPAMI.1981.4767144

Sharif, M. M., Chuo, S., & Haas, C. T. (2022). Rapid 3D Quality Control in Prefabrication Using a 3D Digital-Templates Framework. *Journal of Construction Engineering and Management*, *148*(11), 04022117. https://doi.org/10.1061/(ASCE)CO.1943-7862.0002353

Sharif, M., Nahangi, M., Haas, C., & West, J. (2017). Automated Model-Based Finding of 3D Objects in Cluttered Construction Point Cloud Models. Computer-Aided Civil and Infrastructure Engineering, 32(11), 893–908. https://doi.org/10.1111/mice.12306

Sharif, M., Rausch, C., Ndiongue, S., Haas, C., & Walbridge, S. (2021). Using 3D scanning for accurate estimation of termination points for dimensional quality assurance in pipe spool fabrication. International Journal of Industrialized Construction, 2, 54-69. https://doi.org/10.29173/ijic261

Shen, Y., Lindenbergh, R., & Wang, J. (2016). Change analysis in structural laser scanning point clouds: The baseline method. Sensors, 17(1), 26. https://doi.org/10.3390/s17010026

Shi, S., Jiang, L., Deng, J., Wang, Z., Guo, C., Shi, J., Wang, X., & Li, H. (2021). PV-RCNN++: Point-voxel feature set abstraction with local vector representation for 3D object detection. arXiv preprint arXiv:2102.00463.

Shi, S., Wang, Z., Wang, X., & Li, H. (2019). Part-A2 net: 3D part-aware and aggregation neural network for object detection from point cloud. arXiv preprint arXiv:1907.03670.

Simon, M., Fischer, K., Milz, S., Witt, C. T., & Gross, H. M. (2020). StickPillars: Robust feature matching on point clouds using graph neural networks. arXiv preprint arXiv:2002.03983.

Simonyan, K., & Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556.

Smith, R., & Smith, R. (1995). Chemical process design. McGraw-Hill New York.

Sobreira, H., Rocha, L., Costa, C., Lima, J., Costa, P., & Moreira, A. P. (2016). 2D cloud template matching - a comparison between iterative closest point and perfect match. In 2016 International Conference on Autonomous Robot Systems and Competitions (ICARSC), 53-59. https://doi.org/10.1109/ICARSC.2016.62

Song, S., Yu, F., Zeng, A., Chang, A. X., Savva, M., & Funkhouser, T. (2016). Semantic scene completion from a single depth image. arXiv preprint arXiv:1611.08974.

Song, S., Lichtenberg, S. P., & Xiao, J. (2015, June). SUN RGB-D: A RGB-D Scene Understanding Benchmark Suite. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR).

Stiny, G., & Mitchell, W. J. (1978). The palladian grammar. Environment and Planning B: Planning and Design, 5(1), 5-18. https://doi.org/10.1068/b050005

Sun, L., Manabe, Y., & Yata, N. (2018). Sparse group match for point cloud registration. ITE Transactions on Media Technology and Applications, 6(2), 151-161. https://doi.org/10.3169/mta.6.151

Sun, M., Xiao, J., Lim, E. G., Zhang, B., & Zhao, Y. (2020). Fast template matching and update for video object tracking and segmentation. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 10791-10799.

Sun, Q., Li, Y., Liu, Z., Huang, X., Liu, F., Liu, X., Ouyang, W., & Shao, J. (2023). UniG3D: A unified 3D object generation dataset. arXiv preprint arXiv:2310.09792.

Tang, P., Huber, D., Akinci, B., Lipman, R., & Lytle, A. (2010). Automatic reconstruction of as-built building information models from laser-scanned point clouds: A review of related techniques. Automation in Construction, 19(7), 829-843. https://doi.org/10.1016/j.autcon.2010.06.007

Thomas, H., Qi, C. R., Deschaud, J. E., Marcotegui, B., Goulette, F., & Guibas, L. J. (2019). KPConv: Flexible and deformable convolution for point clouds. In Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV).

Tian, Y., Wang, X., Shen, Y., Guo, Z., Wang, Z., & Wang, F. Y. (2021). Parallel point clouds: Hybrid point cloud generation and 3D model enhancement via virtual–real integration. Remote Sensing, 13(15), 2924. https://doi.org/10.3390/rs13152924

Tremblay, J., Prakash, A., Acuna, D., Brophy, M., Jampani, V., Anil, C., To, T., Cameracci, E., Boochoon, S., & Birchfield, S. (2018). Training deep networks with synthetic data: Bridging the reality gap by domain randomization. arXiv preprint arXiv:1804.06516.

Triess, L. T., Peter, D., Baur, S. A., & Zöllner, J. M. (2021). Quantifying Point Cloud Realism Through Adversarially Learned Latent Representations. In Pattern Recognition (pp. 681-696). Springer. https://doi.org/10.1007/978-3-030-92659-5_44

Uy, M. A., Pham, Q.-H., Hua, B.-S., Nguyen, D. T., & Yeung, S.-K. (2019). Revisiting Point Cloud Classification: A New Benchmark Dataset and Classification Model on Real-World Data. International Conference on Computer Vision (ICCV).

Valle, A., Queiroz, T., Miyazawa, F., & Xavier, E. (2012). Heuristics for two-dimensional knapsack and cutting stock problems with items of irregular shape. Expert Systems with Applications, 39(16), 12589-12598. https://doi.org/10.1016/j.eswa.2012.05.025

Varga, D., Szalai-Gindl, J. M., Formanek, B., Vaderna, P., Dobos, L., & Laki, S. (2021). Template Matching for 3D Objects in Large Point Clouds Using DBMS. IEEE Access, 9, 76894–76907. https://doi.org/10.1109/ACCESS.2021.3082848

Vedenov, A. A. (1994). Computer generated three-dimensional representations of objects. Physics-Uspekhi, 37(9), 889-899. https://doi.org/10.1070/PU1994v037n09ABEH000043

Villaggi, L., Stoddart, J., Nagy, D., & Benjamin, D. (2018). Survey-Based Simulation of User Satisfaction for Generative Design in Architecture. In Advances in Architectural Geometry 2018 (pp. 417-430). Springer Singapore. https://doi.org/10.1007/978-3-319-75544-7_20

Vock, R., Dieckmann, A., Ochmann, S., & Klein, R. (2019). Fast template matching and pose estimation in 3D point clouds. Computers & Graphics, 79, 36-45. https://doi.org/10.1016/j.cag.2018.12.007

Vora, S., Lang, A. H., Helou, B., & Beijbom, O. (2019). PointPainting: Sequential Fusion for 3D Object Detection. arXiv preprint arXiv:1911.10150.

Wang, B., Wang, Q., Cheng, J. C. P., & Yin, C. (2022). Object verification based on deep learning point feature comparison for scan-to-BIM. Automation in Construction, 142, 104515. https://doi.org/10.1016/j.autcon.2022.104515

Wang, C., Cho, Y. K., & Kim, C. (2015). Automatic BIM component extraction from point clouds of existing buildings for sustainability applications. Automation in Construction, 56, 1-13. https://doi.org/10.1016/j.autcon.2015.04.001

Wang, D. Z., & Posner, I. (2015). Voting for voting in Online Point Cloud Object Detection. Robotics: Science and Systems XI. https://doi.org/10.15607/rss.2015.xi.035

Wang, J. Y., & Cohen, F. S. (1994). Part II: 3-D object recognition and shape estimation from image contours using B-splines, shape invariant matching, and neural network. IEEE Transactions on Pattern Analysis and Machine Intelligence, 16(1), 13-23. https://doi.org/10.1109/34.273721

Wang, Q., Guo, J., & Kim, M.-K. (2019). An Application Oriented Scan-to-BIM Framework. Remote Sensing, 11(3), 365. https://doi.org/10.3390/rs11030365

Wang, Q., & Kim, M.-K. (2019). Applications of 3D point cloud data in the construction industry: A fifteen-year review from 2004 to 2018. Advanced Engineering Informatics, 39, 306-319. https://doi.org/10.1016/j.aei.2019.02.007

Wang, Q., Tan, Y., & Mei, Z. (2020). Computational methods of acquisition and processing of 3D point cloud data for construction applications. Archives of Computational Methods in Engineering, 27(2), 479-499. https://doi.org/10.1007/s11831-019-09320-4

Wang, X., Liu, S., Shen, X., Shen, C., & Jia, J. (2019). Associatively segmenting instances and semantics in point clouds. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR).

Wang, Y., & Solomon, J. M. (2019). PRNet: Self-supervised learning for partial-to-partial registration. arXiv preprint arXiv:1910.12240.

Wang, Y., Sun, Y., Liu, Z., Sarma, S. E., Bronstein, M. M., & Solomon, J. M. (2019). Dynamic graph CNN for learning on point clouds. arXiv preprint arXiv:1801.07829.

Wang, Z., & Lu, F. (2018). VoxSegNet: Volumetric CNNs for semantic part segmentation of 3D shapes. arXiv preprint arXiv:1809.00226.

Weng, W., & Zhu, X. (2021). INet: Convolutional Networks for Biomedical Image Segmentation. IEEE Access, 9, 16591-16603. https://doi.org/10.1109/ACCESS.2021.3053322

White, K. B., Cline, D., & Egbert, P. K. (2007). Poisson disk point sets by hierarchical dart throwing. 2007 IEEE Symposium on Interactive Ray Tracing, 129-132. https://doi.org/10.1109/rt.2007.4342600

Williams, R. M., & Ilieş, H. T. (2018). Practical shape analysis and segmentation methods for point cloud models. Computer Aided Geometric Design, 67, 97-120. https://doi.org/10.1016/j.cagd.2018.10.003

Wong, J. K. W., Ge, J., & He, S. X. (2018). Digitisation in facilities management: A literature review and future research directions. Automation in Construction, 92, 312-326. https://doi.org/10.1016/j.autcon.2018.04.006

Wu, C., Bi, X., Pfrommer, J., Cebulla, A., Mangold, S., & Beyerer, J. (2023). Sim2real transfer learning for point cloud segmentation: An industrial application case on autonomous disassembly. In Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV), 4531-4540.

Wu, X., Jiang, L., Wang, P.-S., Liu, Z., Liu, X., Qiao, Y., & Zhao, H. (2024). Point Transformer V3: Simpler, Faster, Stronger. arXiv preprint arXiv:2312.10035.

Wu, Z., Song, S., Khosla, A., Yu, F., Zhang, L., Tang, X., & Xiao, J. (2015). 3D ShapeNets: A deep representation for volumetric shapes. In 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 1912-1920. https://doi.org/10.1109/CVPR.2015.7298801

Xu, J., Cheng, W., Gao, Y., Wang, X., Gao, S., & Shan, Y. (2024). InstantMesh: Efficient 3D mesh generation from a single image with sparse-view large reconstruction models. arXiv preprint arXiv:2404.07191.

Xu, J., Chen, K., Xue, F., & Lu, W. (2021). 3D Point Cloud Data Enabled Facility Management: A Critical Review. In Proceedings of the 23rd International Symposium on Advancement of Construction Management and Real Estate (pp. 641-657). Springer Singapore. https://doi.org/10.1007/978-981-15-7463-4_53

Xu, Q., Sun, J., Liu, L., Jia, J., & Zhao, H. (2022). Stratified Transformer for 3D Point Cloud Segmentation. arXiv preprint arXiv:2203.14508.

Xu, Y., Sun, Z., Boerner, R., Koch, T., Hoegner, L., & Stilla, U. (2018). Generation of ground truth datasets for the analysis of 3D point clouds in urban scenes acquired via different sensors. The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences, 42, 2009-2015. https://doi.org/10.5194/isprs-archives-XLII-2-2009-2018

Xu, Q., Huang, Y., & Yang, Y. (2019). Grid-GCN for Fast and Scalable Point Cloud Learning. arXiv preprint arXiv:1912.02984.

Yan, X. (2019). Yanx27/Pointnet_Pointnet2_pytorch: PointNet and PointNet++ implemented by PyTorch (pure python) and on ModelNet, ShapeNet and S3DIS. GitHub repository. https://github.com/yanx27/Pointnet_Pointnet2_pytorch

Yang, J., Chen, J., Quan, S., Wang, W., & Zhang, Y. (2022). Correspondence Selection With Loose-Tight Geometric Voting for 3-D Point Cloud Registration. IEEE Transactions on Geoscience and Remote Sensing, 60, 1-14. https://doi.org/10.1109/TGRS.2022.3142074

Yeo, C., Kim, S., Kim, H., Kim, S., & Mun, D. (2020). Deep learning applications in an industrial process plant: Repository of segmented point clouds for pipework components. Data in Brief, 28, 104966. https://doi.org/10.1016/j.dib.2019.104966

Yotov, S. V. (2015). Towards Change Detection for Laser Scanning Point Clouds. University of Houston.

Zaharia, T. B., & Prêteux, F. J. (2002). Shape-based retrieval of 3D mesh models. Proceedings. IEEE International Conference on Multimedia and Expo, 1, 437-440. https://doi.org/10.1109/ICME.2002.1035809

Zaidi, S. S. A., Ansari, M. S., Aslam, A., Kanwal, N., Asghar, M. N., & Lee, B. A. (2021). A survey of modern deep learning based object detection models. arXiv preprint arXiv:2104.11892.

Zeng, B., Li, S., Feng, Y., Yang, L., Li, H., Gao, S., Liu, J., He, C., Zhang, W., Liu, J., Zhang, B., & Yan, S. (2024). IPDreamer: Appearance-controllable 3D object generation with complex image prompts. arXiv preprint arXiv:2402.05073.

Zeng, S., Chen, J., & Cho, Y. K. (2020). User exemplar-based building element retrieval from raw point clouds using deep point-level features. Automation in Construction, 114, 103159. https://doi.org/10.1016/j.autcon.2020.103159

Zhang, C., & Chen, T. (2001). Efficient feature extraction for 2D/3D objects in mesh representation. In Proceedings 2001 International Conference on Image Processing (Cat. No.01CH37205), 3, 935-938. https://doi.org/10.1109/ICIP.2001.958278

Zhang, J., & Singh, S. (2014). LOAM: Lidar Odometry and Mapping in Real-time. Robotics: Science and Systems, 109-111. https://doi.org/10.15607/RSS.2014.X.007

Zhang, X., Zou, J., He, K., & Sun, J. (2015). Accelerating very deep convolutional networks for classification and detection. arXiv preprint arXiv:1505.06798.

Zhang, Z. (1992). Iterative point matching for registration of free-form curves. International Journal of Computer Vision, 13, 119-152. https://doi.org/10.1007/BF01427149

Zhang, Z., Dai, Y., & Sun, J. (2020). Deep learning based point cloud registration: an overview. Virtual Reality & Intelligent Hardware, 2(3), 222-246. https://doi.org/10.1016/j.vrih.2020.05.002

Zhang, Z., Hua, B. S., & Yeung, S. K. (2019). ShellNet: Efficient point cloud convolutional neural networks using concentric shells statistics. In Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV).

Zhao, H., Jiang, L., Fu, C. W., & Jia, J. (2019). PointWeb: Enhancing local neighborhood features for point cloud processing. In CVPR.

Zhao, H., Jiang, L., Jia, J., Torr, P., & Koltun, V. (2021). Point transformer. arXiv preprint arXiv:2012.09164.

Zhao, L., & Tao, W. (2020). JSNet: Joint instance and semantic segmentation of 3D point clouds. In Thirty-Fourth AAAI Conference on Artificial Intelligence.

Zheng, J., Zhang, J., Li, J., Tang, R., Gao, S., & Zhou, Z. (2020). Structured3D: A large photo-realistic dataset for structured 3D modeling. In Proceedings of The European Conference on Computer Vision (ECCV).

Zhou, M., Wang, Y., Hou, J., Luo, C., Zhang, Z., & Peng, J. (2024). SceneX: Procedural Controllable Large-scale Scene Generation via Large-language Models. arXiv preprint arXiv:2403.15698.

Zhou, Q. Y., Park, J., & Koltun, V. (2018). Open3D: A Modern Library for 3D Data Processing. arXiv preprint arXiv:1801.09847.

Zhou, Y., & Tuzel, O. (2017). VoxelNet: End-to-end learning for point cloud based 3D object detection. arXiv preprint arXiv:1711.06396.

Zhou, Z., Chen, J., & Zhang, Z. (2021). Local feature extraction network for point cloud analysis. Symmetry, 13(2), 321. https://doi.org/10.3390/sym13020321

Zhu, B., Jiang, Z., Zhou, X., Li, Z., & Yu, G. (2019). Class-balanced grouping and sampling for point cloud 3D object detection. arXiv preprint arXiv:1908.09492.

Ziegler, M., Rudorfer, M., Kroischke, X., Krone, S., & Krüger, J. (2019). Point pair feature matching: Evaluating methods to detect simple shapes. In Computer Vision Systems (pp. 445-456). Springer International Publishing. https://doi.org/10.1007/978-3-030-34995-0_40

Zimmer, W., Ercelik, E., Zhou, X., Ortiz, X. J. D., & Knoll, A. (2022). A Survey of Robust 3D Object Detection Methods in Point Clouds. arXiv preprint arXiv:2204.00106.

Zou, Z., Shi, Z., Guo, Y., & Ye, J. (2019). Object detection in 20 years: A survey. arXiv preprint arXiv:1905.05055.

Garcia, G., et al. (2021). Measuring progress and productivity in model-driven engineering for capital project delivery. *Journal of Construction Engineering and Management*, *147*(4), 04021009. https://doi.org/10.1061/(ASCE)CO.1943-7862.0001969

Lytle, A. M. (2011). A framework for object recognition in construction using building information modeling and high frame rate 3D imaging. Doctoral dissertation, Virginia Tech. *Virginia Tech Digital Library*. http://hdl.handle.net/10919/26888

Dimitrov, A., et al. (2016). Non-uniform B-spline surface fitting from unordered 3D point clouds for as-built modeling. *Computer-Aided Civil and Infrastructure Engineering*, *31*(7), 483–498. https://doi.org/10.1111/mice.12192

Perez-Perez, Y., Golparvar-Fard, M., & El-Rayes, K. (2021). Scan2BIM-NET: Deep learning method for segmentation of point clouds for scan-to-BIM. *Journal of Construction Engineering and Management*, *147*(9), Article 04021107. https://doi.org/10.1061/(ASCE)CO.1943-7862.0002132

Perez-Perez, Y., et al. (2021). Segmentation of point clouds via joint semantic and geometric features for 3D modeling of the built environment. *Automation in Construction*, *125*, 103584. https://doi.org/10.1016/j.autcon.2021.103584

Roberts, D., Danielyan, A., Chu, H., Golparvar Fard, M., & Forsyth, D. A. (2021). LSD-StructureNet: Modeling Levels of Structural Detail in 3D Part Hierarchies. arXiv preprint arXiv: 2108.13459

DeGol, J., Golparvar Fard, M., & Hoiem, D. (2016). Geometry-Informed Material Recognition. arXiv preprint arXiv:1607.05338

Boogaard, F.P., van Henten, E.J. and Kootstra, G. (2022). Improved Point-Cloud Segmentation for Plant Phenotyping Through Class-Dependent Sampling of Training Data to Battle Class Imbalance. *Front. Plant Sci*. 13:838190. https://doi.org/10.3389/fpls.2022.838190

Qin, Z., Yu, H., Wang, C., Guo, Y., Peng, Y., & Xu, K. (2022). Geometric Transformer for Fast and Robust Point Cloud Registration. arXiv preprint arXiv: 2202.06688

Appendix A

Evaluation Metrics

Statement of Contributions

The research in this chapter is only a basic explanation of how computer vision and machine learning works for images and point clouds. I have not generated any of the images or research presented in this section.

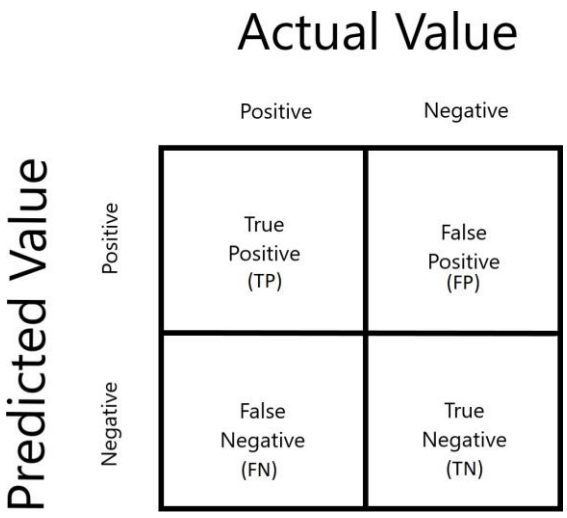


Figure 39. Confusion Matrix

The investigated network architectures will be evaluated based on semantic segmentation error metrics such as overall accuracy and mean Intersection over Union (mIoU). Overall accuracy (OA) metrics represent the mean accuracy of each pixel. The confusion matrix gives the accuracy in Figure 39 measured as $(TP + TN)/(TP + FP + FN + TN)$, which leads to the equation:

$$OverallAccuracy = \frac{1}{C} * \frac{N_{ii}}{(T_i)} \tag{9}$$

Where C corresponds to the total number of classes, N_{ii}, denotes the points being correctly classified. T_i is the total number of points in class i. The segmentation accuracy will be high as long as it can correctly predict most of the points in the class. However, it does not consider class object size, which generates a class size imbalance on the classes with a low points prevalence in the scene, such as a thin electric cable class along a wall of a factory class. For this reason, the evaluation metric mIoU used is based on $\frac{target \cap prediction}{target \cup prediction}$, which leads to the equation :

$$mIoU = \frac{1}{C} * \frac{N_{ii}}{(T_i + N_{ji} - N_{ii})} \tag{10}$$

Where N_{ij} corresponds to the false positive points belonging to another class j but wrongly labeled as class i, the mIoU considers the performance of lower

size objects in a class by penalizing the model performance rates on the classification of a lower prevalent class.

Appendix B

Machine Learning Fundamentals

Statement of Contributions

The research in this chapter is only a basic explanation of how computer vision and machine learning works for images and point clouds. I have not generated any of the images or research presented in this section.

B.1 Multi-Layer Perceptron

Machine Learning is the use and development of computer systems that can learn and adapt without following explicit instructions by using algorithms and statistical models to analyze and draw inferences from patterns in data (IBM, 2023).

A neuron is a simple computational unit that has one or multiple inputs and produces an output signal using an activation function discussed in subsection B.1.1. An artificial neural network, requires an input layer depicted as L_1, at least one hidden layer here depicted as L_2 and L_3, and an output layer L_4 (Arnx, 2019).

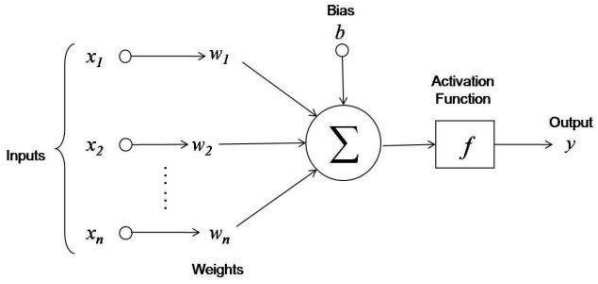


Figure 40. Machine Learning Neuron

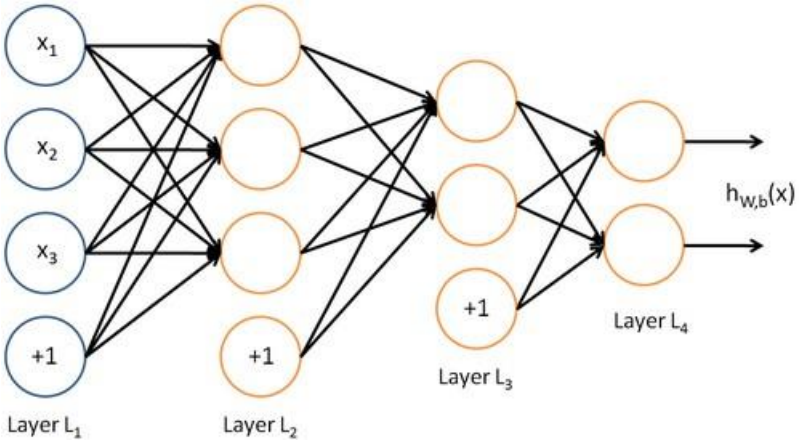


Figure 41, Artificial Neural Network

The data is transmitted from the input layer to the output layer; this process is called Feedforward. Once the data is in the output layer, it is compared to the true value. This comparison is done through the loss functions subsection B.1.2.

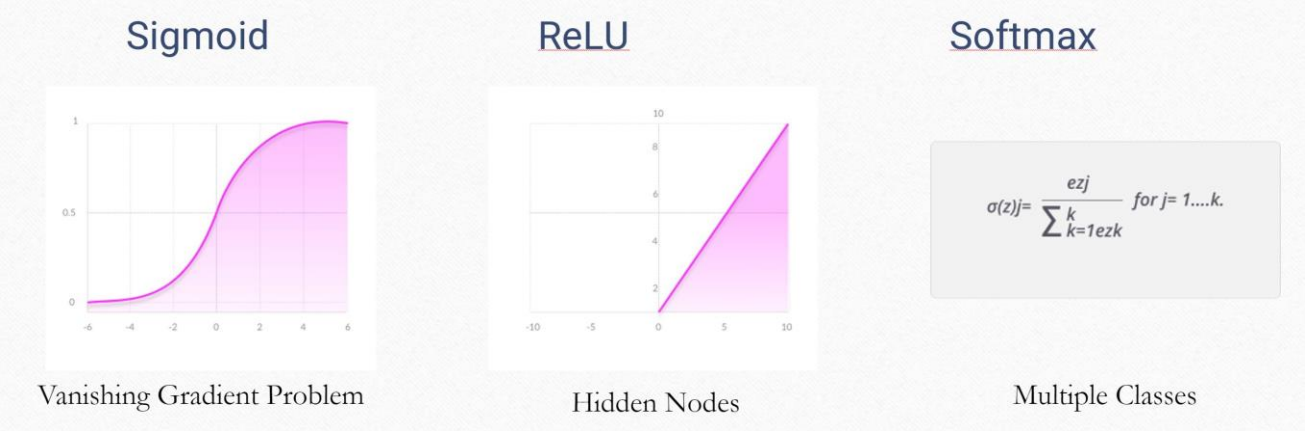


Figure 42 Activation Functions

B.1.1 Activation Functions

An activation function is a function that will activate or be sued for inputting the summed weights of the previous layer in the neuron, as seen in Figure 42. There are multiple activation Functions: Sigmoid, Tanh, ReLU, Softmax, LeakyReLU, PReLU, ELU, ThresholdedReLU, etc. (Keras, 2023): Ending with a Sigmoid function will give an output from 0 to 1. Ending with a ReLU will give a linear output. Ending with a SoftMax will give multiple probabilities, best for a categorical output.

B.1.2 Loss Functions

Loss functions will compare the output of the network $\hat{y}$ with the true value y to get the error or loss. While there are many loss classes and functions: Probabilistic losses: Binary Cross entropy, Categorical Cross entropy, binary cross entropy, categorical cross entropy Regression losses: Mean Squared Error, Mean Absolute Error, Mean Absolute Percentage Error, Mean Squared Logarithmic Error, mean absolute error, mean absolute percentage error, etc. (Keras, 2023)

The uses for each depend on the input and the needed output. For example, if the output is 0 or 1, the binary cross entropy would be the best. If the output is a category 1 or 2 or 3 or 4, then categorical cross entropy would be best. Always research the input and the output needed before doing any training.

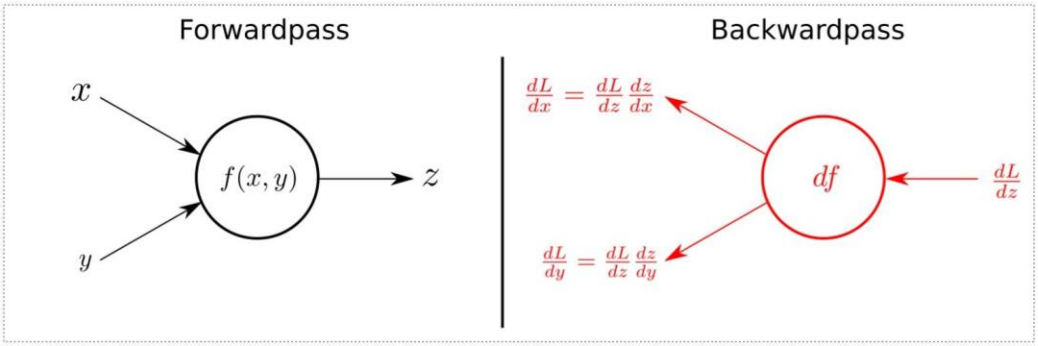


Figure 43. Back Propagation

B.1.3 Backpropagation

Once the network has the "Error" or "Loss", it redistributes the error through the weights using Backpropagation. We use a loss function to know how far off was the prediction from the real value. To know how much each weight is affecting the error, we propagate the error through the weight matrix as seen in Figure 43: Through each iteration (epoch), we modify the weights using the chain rule and an optimizer section B.1.3 until it reaches a local minimum.

B.1.4 Optimizers

Once the error is obtained, the optimizers will use this data to The optimizers (Keras, 2023) usually: SGD, RMSprop, Adam, Adadelta, Adagrad, Adamax, etc., use gradient decent where each epoch affects the weights $t\theta$. For SGD:

$$\theta = \theta - \alpha \cdot \nabla_\theta J(\theta) \tag{11}$$

Each time it gets each data point. However, every sample may modify the direction, making it noisy and never allowing it to get to the global minimum. Therefore, momentum is added:

$$v = \gamma \cdot v + \eta \cdot \nabla_\theta J(\theta) \tag{12}$$

$$\theta = \theta - \alpha v \tag{13}$$

Multiple optimizers have been created, and each one is used depending on the type of problem trying to be solved (Ruder, 2020).

Appendix C

Computer Vision

Statement of Contributions

The research in this chapter is only a basic explanation of how computer vision and machine learning works for images and point clouds. I have not generated any of the images or research presented in this section.

Before entering, it is vital to know the difference between what problems visual networks can solve. The most usual ones are Semantic Segmentation, Instance Segmentation, Image Classification, Object Detection, Domain Adaptation, and Image Generation as seen in Figure 44 (Parmar, 2018).

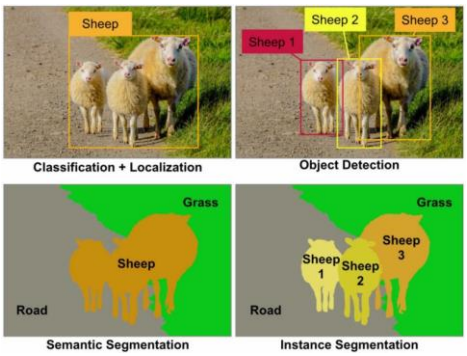


Figure 44. Computer Vision Common Problems (Parmar, 2018).

C.1 Convolutional Neural Network (CNN)

Convolutional Neural Networks are used mostly for complex visual problems; however, the architectures of each network are what makes the solution to each of the problems.

C.1.1 Convolutions

A Convolution is the product of functions, in this case, it is of values and kernels, and what makes the convolution a network layer is that each kernel (also called filter) is a grid of modifiable weights Figure 45 (Arnx, 2019).

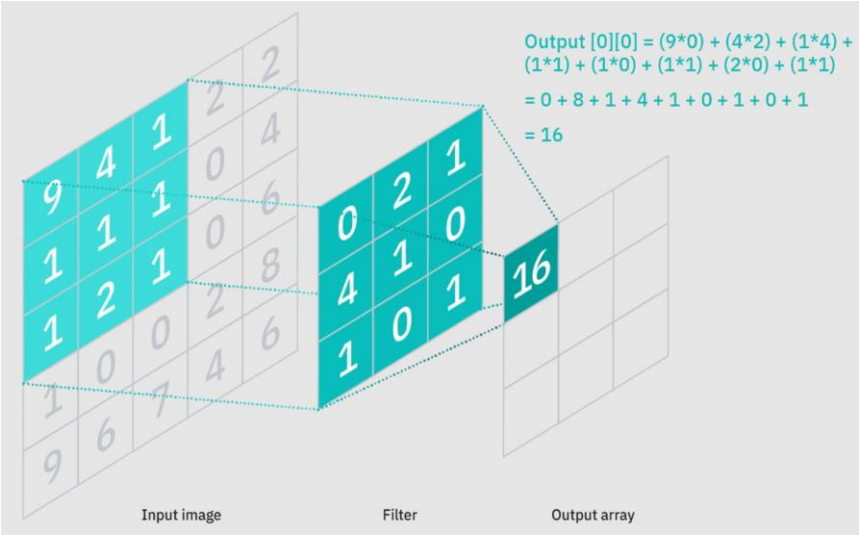


Figure 45. Convolution

C.1.2 Max Pooling

As the filter moves across the input, it selects the pixel with the maximum value to send to the output array. The pooling operation calculates the maximum value for patches of a feature map and creates a downsampled (pooled) feature map. It is usually used after a convolutional layer.

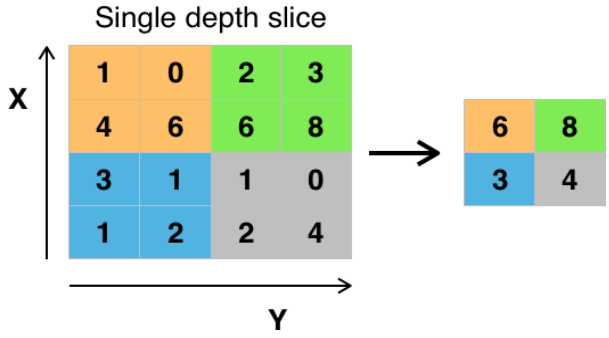


Figure 46. Max-pooling (Arnx, 2019)

C.2 CNN Networks (VGG16)

VGG16 is a convolutional neural network model proposed by K. Simonyan and A. Zisserman(Zhang, 2015) for image classification, and it can be modified for object detection. As seen in the layers, there are two convolutions and a max-pooling layer repeated until fully connected layers with a ReLU output. In the 7 (Hassan, 2018) the final output layer is a SoftMax which means the network is used for categorizing images.

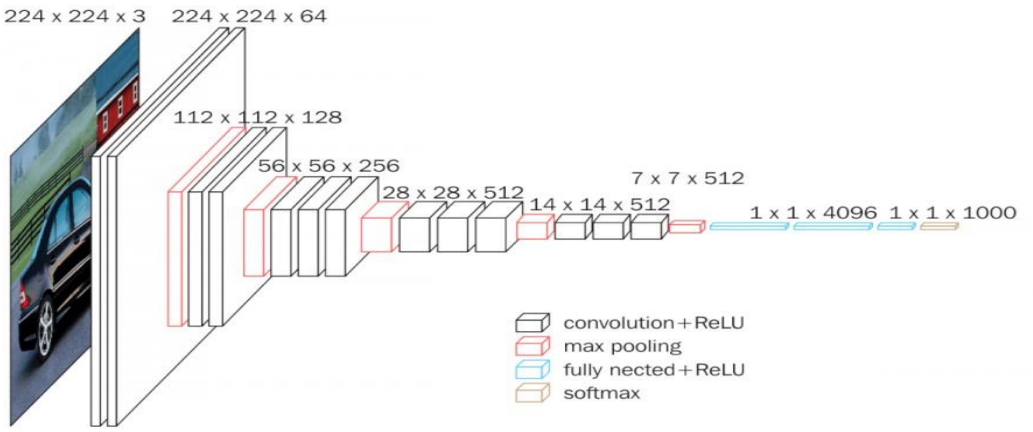


Figure 47. VGG-16 Architecture (Hassan, 2018)

C.3 Residual Networks

residual networks (res-nets) use convolutions to find features and copy the input of a layer to add it later into the network (Hassan, 2019) also called shortcuts. A ResNet was initially created for convolutional networks since accuracy gets saturated and decreases as the network depth increases. The ResNet revived the deep convolutional network by making shortcuts that allow deeper layers to learn more about the input (He, 2015).

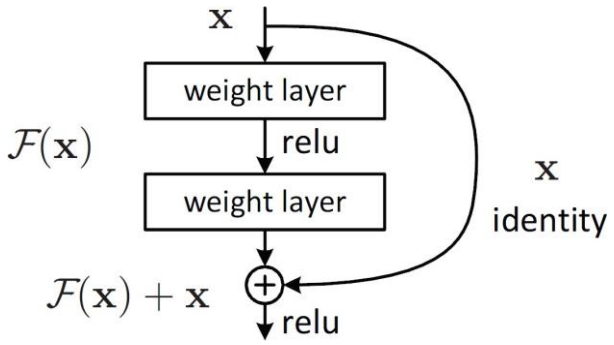


Figure 48. ResNet Module (He, 2015)

C.4 U-Net

The U-Nets are U-shaped convolutional network architecture for fast and precise segmentation of images. The architecture that uses the last layer of the very 3rd convolution before the *N*/2 convolutions and uses them as input to the next *N*/2 as seen in Figure 49 (Ronneberger, 2015)

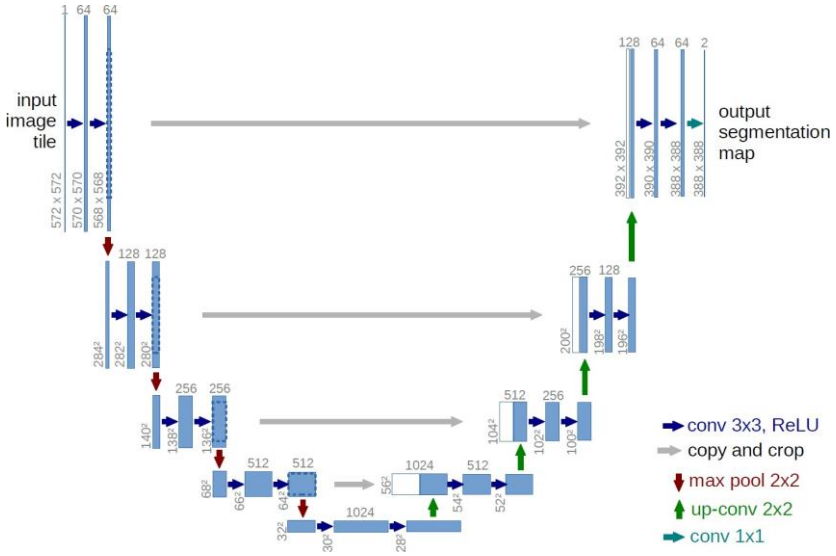


Figure 49. U-Net Architecture (Ronneberger, 2015)

C.5 Siamese Networks

Siamese network aims to bring the output feature vectors closer for input pairs labeled as similar and push the feature vectors away if the input pairs are dissimilar. As explained by the author Dey et al. (2017), the identical parallel networks are made to learn a similarity function.

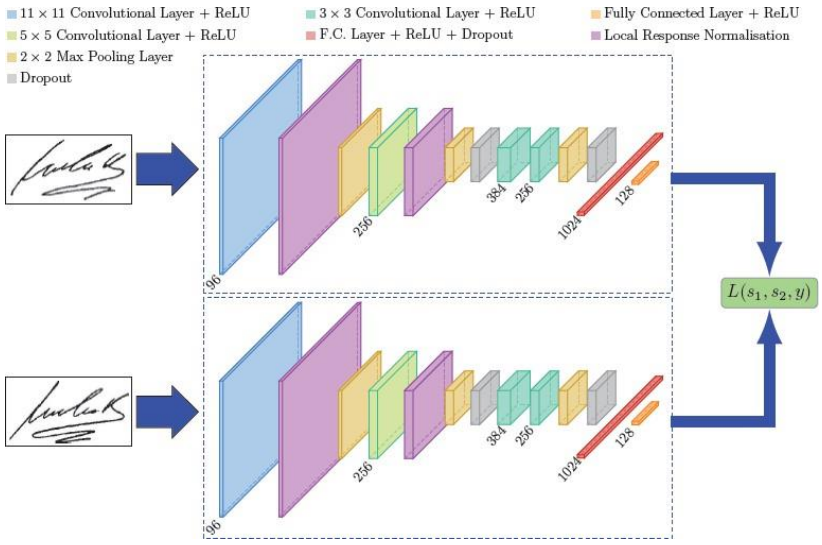


Figure 50 Siamese Network Architecture (Dey, 2017)

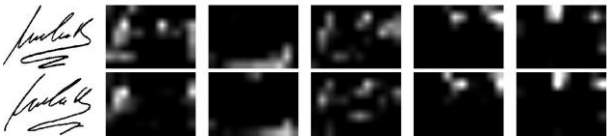


Figure 51 Siamese Network results per convolution (Dey, 2017)

The network uses parallel convolutional layers and a final latent space where the latent space saves the features and can compare the features depending on the output of both networks.

C.6 Graph neural networks

Graph Neural Networks (GNNs) are a class of deep learning methods designed to perform inference on graph data and can be applied directly to graphs, and predict node-level, edge level, and graph-level tasks. The graph network architecture described in Sticky Pillars (Simon, 2020) relies on two complete graphs, G_L and G_K,

whose nodes are related and equivalent to the pillars quantity. It is composed of intra-frame edges, i.e., self edges connecting each key-point within G_L and each key-point within G_K respectively(Simon, 2020).

C.7 T-Net

The *T -Net* structure is based on the pattern of flattening the input, passing the flat input through Orthogonal Regularization(Brock, 2017), then a conv bn composed of 1D convolution with batch normalization and a ReLu (Keras, 2023) layer. The Conv bn is repeated three times, and then a global max pooling is done to the result. Finally, we use the dense bn composed of a dense layer and a batch normalization with a ReLu output layer. This dense bn is used two times. The output of the two dense bn is used in a final dense layer of the size of a number of features squared. Finally, the output is reshaped into a grid of size of feature number x feature number, and the final result is the dot product of the inputs and the matrix(Kossaifi, 2019).

C.8 Other Neural Networks

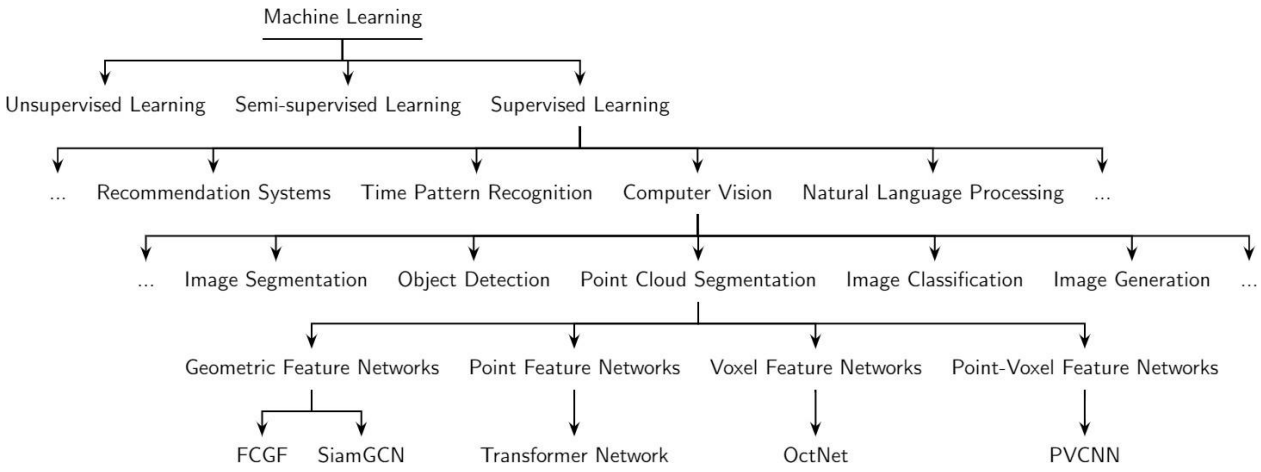


Figure 52. General Hierarchy of Networks

Point cloud data has been aggregated and exponentially increasing in the past few years due to increased computation power and the ability to capture high-definition laser scans of areas of interest. Point cloud data

format is defined as a set of 3D points where $\{Pi|i = 1,\ldots,n\}$ and each P_i are vectors of (x, y, z) values, and other possible additions such as color will be used. Initial statistical-based parametric models, fuzzy algorithms, and point density methods were used to find cylinder-like tubes and other geometric features in point clouds (Aijazi, 2019; Bauer, 2009). This approach would work well for the problem being solved; however, it is not generalizable for complex or highly convex geometric patterns (Aijazi, 2019). Recent advances in neural networks have immensely contributed to point cloud analysis, where a trained segmentation network can categorize each point of a point cloud scene. These advances broaden the object recognition spectrum to almost any object (Landrieu, 2017). This section focuses on point cloud neural segmentation networks, precisely four different machine learning approaches that have proven to work well in real-life applications. In Appendix B, there is a fast-paced introduction to machine learning basics and the evaluation metrics used in Appendix A.

C.8.1 Point Feature Analysis

Neural networks based on processing the raw point clouds without pre-processing or post-processing before passing the point cloud data to the neural network are point-based networks. The pioneering work of Qi et al., PointNet, (Qi, 2017) proposed a classification and segmentation network. The proposed solution for unordered point set $x_1, x_2, \ldots, x_n$ where $x_i \in \mathrm{R}^d$ defined the function $\boldsymbol{f}: \chi \rightarrow \mathrm{R}$ for a neural network maximization of the equation:

$$f\ (x_1, x_2, \ldots, x_n) = \gamma\ \left(MAX_{i=1,\ldots,n}\ \{h\ (x_i)\}\right) \tag{14}$$

Where γ and h are MLPs that take the input of n points with (x, y, z) point values and apply input and feature transformations, using *T -Net* (Appendix A.0.3) modules, in between the inputs and the MLPs, to have a segmentation network, the output of the second *T -Net* used for feature transformation is added to a higher latent space of global features and encoded back into the number of the inputs.

Even though the Overall Accuracy for the ModelNet40 (Notchenko, 2017) dataset was 98.2%, the dataset created for the network is made to segment parts of a point cloud object, which has to be lower than 1 million points. Unfortunately, PointNet would not capture complex scenes due to a lack of local context at different scales. Therefore, a hierarchical attention structure was needed, which was improved by Qi. Qi published PointNet++(Qi, 2017) a few months later, which used hierarchical point set feature learning. The hierarchy was composed of abstraction levels, taking $N \times (d + C)$ matrix as input of N points with d dimension coordinates and C dimension point features passed to a sampling and abstraction layer. The Sampling layer does farthest point sampling (FPS) (Kamousi, 2016) with the same output $N' \times (d + C')$. The grouping layer

takes $N \times (d + C)$ points and uses the Ball Query algorithm to generate a local region feature vector $N' \times K \times (d + C)$ where K is the centroid for the nearest neighbor (kNN) (Munaga, 2012). Once this prepossessing is done, the density adaptive PointNet layers learn to combine features from regions of different scales and use PointNet on those scales as layers. This hierarchical network adapts to the density of the points, allowing the PointNet++ to use a lower number of data points to train and get higher accuracy(Qi, 2017). Qi et al. (2017) when creating PointNet++ noted that CNNs could scan the space with fixed strides and can work on scaled features; their view was that the input point set is very different from CNN inputs since point clouds are unordered, without uniform constant density, which in cases could lead to sampling deficiency to allow the capture of patterns. However, Yangyan Li et al., (2018)when creating PointCNN, decided to approach the CNN kernel with higher dimensions as $\mathbf{K} = [k_\alpha, k_\beta, k_\gamma, k_\delta]^T$ transforming the input points through an MLP transformation using the χ-Conv operator to convert the point coordinates to learnable features.

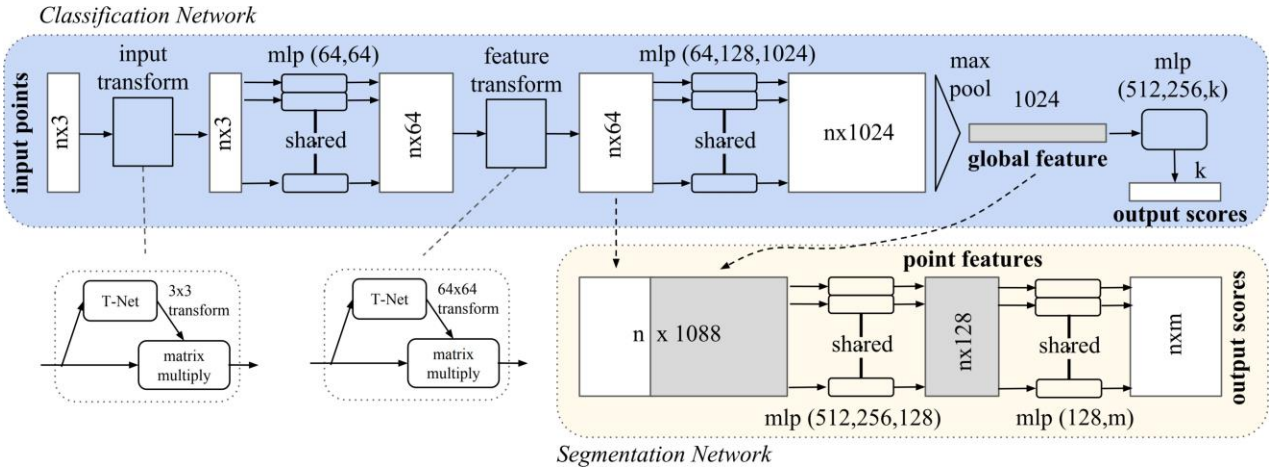


Figure 53 Point Net Network (Qi, 2017)

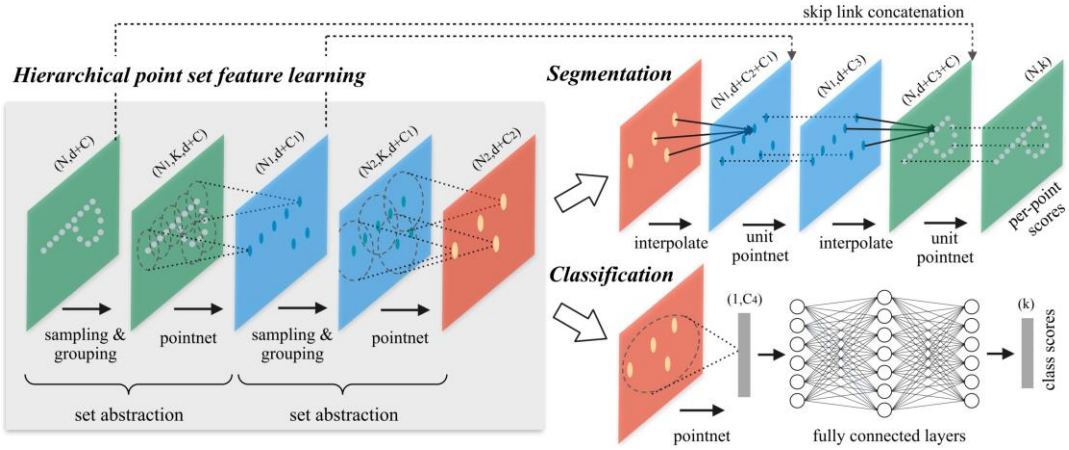


Figure 54 PointNet++ Network (Qi, 2017)

A similar approach to kernel modifications was done by Thomas et al. (2019) in KPConv, where the kernel convolution is deformable depending on the subsample of the neighboring input points' local shifts, such as sparsity, to modify the deformable kernel in the convolutional layer. This modification generates the kernel shifts for every convolution location (Dai, 2017) later used by the hybrid network DRINet.

C.8.2 Voxel Strategies

Another representation to point is the voxelization which transforms sparse point clouds into a volumetric occupancy grid representation, also called voxel for volumetric pixel. The initial voxelization technique for neural network segmentation was used by Maturana and Scherer (2015) in VoxNet by mapping each point (x, y, z) from the point cloud to a discrete voxel coordinate (i, j, k) depending on the orientation, origin, and size or resolution of the voxel in the virtual space. The voxel space orientation is based on the z-axis of the grid frame aligned to the direction of gravity.

The voxel size or resolution in VoxNet is done by fixing a spatial resolution of 0.1^3m. The quantity of the voxels would depend on the subdivision per point cloud resolution allowing a CNN with multiple vanilla MLPs or 3D kernels to do feature detection. This approach became the base for faster point cloud processing algorithms since the point cloud pre-processing would abstract the geometrical data(Song, 2016; Brock, 2016; Le, 2018; Wang, 2018; Qi, 2016; Wu, 2015). However, as stated by Maturana et al.(2015), there would be examples where sparse data would be lost due to the voxel size invariability. OctNet by Riegler

et al. (2017) approached voxelization using a Hybrid Grid-Octree Data Structure to solve this problem. Octrees generate hierarchical voxels and apply a tree-based algorithm approach to map all the points in the point cloud to the appropriate voxel. The voxelization starts with an initial eight voxels filling the scene's volume. Once a point has been detected on a voxel, the voxel is branched or split into eight voxels of equal volume, and the process is repeated to a designated tree depth. Therefore, if a voxel does not contain a point, it does not split, allowing for greater granularity without consuming memory from space with no features. The benefits of this voxelization are efficient bit encoding, which allows for lower access times and efficient GPGPU implementations(Riegler, 2017) when traversing the tree and generating point feature abstraction for large continuous point cloud areas.

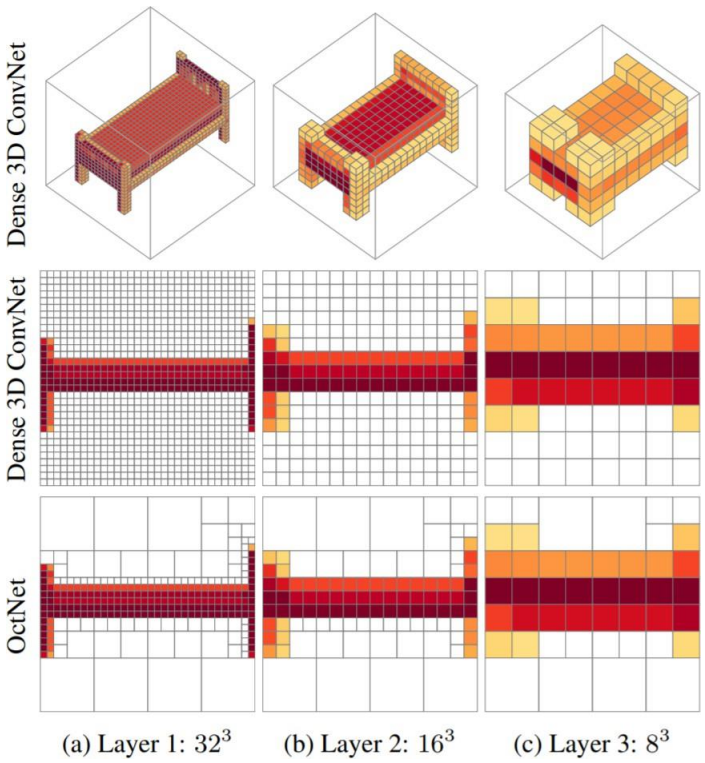


Figure 55 OctNet Voxelization (Riegler, 2017)

C.8.3 Hybrid Networks

Hybrid networks simultaneously utilize the point cloud points and other point cloud abstractions such as voxelizations or vectors to learn features of the point cloud. One of the first examples of hybrid networks was the 3D Modified Fisher Vectors Network (3DmFVNet) by Shabat et al.(2017), which had a grid's discrete structures and the continuous generalization of Fisher vectors as the input for concatenated CNNs.While this approach was not state-of-the-art for more than a few months with an overall accuracy of 84.3% on ModelNet40 (Wu, 2015), it began the introduction of hybrid networks. An interesting approach to hybrid networks is Point Voxel CNN based on PointNet++ (PVCNN++) (Liu, 2019) making it a fast option with a latency of 69.6ms, the accuracy of 87.12%, and mIoU of 58.98% per scene of 32k points. PVCNN++ takes advantage of both point feature learning using an MLP while using normalization by scaling the point clouds from 0 to 1 by having the origin at the gravity center and dividing the coordinate values by their maximum values $max||\hat{\boldsymbol{p}}_k||$ in the point cloud scene. Then the normalized points $\hat{\boldsymbol{p}}_k = (\hat{\boldsymbol{x}}_k, \hat{\boldsymbol{y}}_k, \hat{\boldsymbol{z}}_k)$, which

maintains the features $\mathbf{f_k}$ are voxelized by averaging all the features whose coordinate $\hat{\boldsymbol{p}}_k$ is part of the voxel grid $\mathbf{V}_{(u,v,w,c)}$ leading to the voxelization equation:

$$\boldsymbol{V}_{u,v,w,c} = \frac{1}{N_{u,v,w}} \sum_{k=1}^{n} \mathbb{I}\left[\text{ floor } (\hat{\boldsymbol{x}}_k \times r) = u,\ \text{floor } (\hat{\boldsymbol{y}}_k \times r) = v,\ \text{floor } (\hat{\boldsymbol{z}}_k \times r) = w\right] \times \boldsymbol{f}_{k,c} \tag{15}$$

Where $N_{u,v,w}$ is the number of points that correspond to the voxel (u, v, w), $\mathbb{I}$ is the binary flag of whether the point $\hat{\boldsymbol{p}}_k$ belongs to the voxel (u, v, w), r denotes the voxel size, and $f_{k,c}$ denotes the c^{th} channel feature for the point $\hat{\boldsymbol{p}}_k$. Using convolutions on the voxels and MLP for the fine point cloud, PVCNN can abstract large and continuous point features using the voxel-based convolutional network features while extracting the isolated and discontinuous features from the point-based MLP. The feature fusion is done by passing the learned convolutional features as an input to the MLP branch of the network.

The most accessible hybrid network to implement is the Minkowski Engine-based Unet Network or Minkowski Network (Choy, 2019) with an S3DIS dataset mIoU of 65.35%, which is only 6% less accurate than the Transformer Network. The network is based on the Sparse Convolutional Neural Networks (SCNN) (Liu, 2015), which uses sparse kernels explained in Appendix A.4.Most point cloud data is sparse due to the flatness and scattering of the points; therefore, the space between the points is not valuable data. The sparse kernel lowers the amount of processing needed by selecting the points to be processed from a volume without

processing the space of the volume itself. The benefit of using this selective kernel is its capability for higher dimension kernels, such as a hypercube volumetric moving window. However, as data increases linearly for 3D data, hypercubes or 4D data increases quadratically. The Minkowski Engine uses the cross-hypercube hybrid, as shown in Figure 57, to alleviate the curse of dimensionality. This method is currently implemented in the methodology subsection 3.1.3 with the proposal to implement the Transformer Network as it is the current best point cloud segmentation.

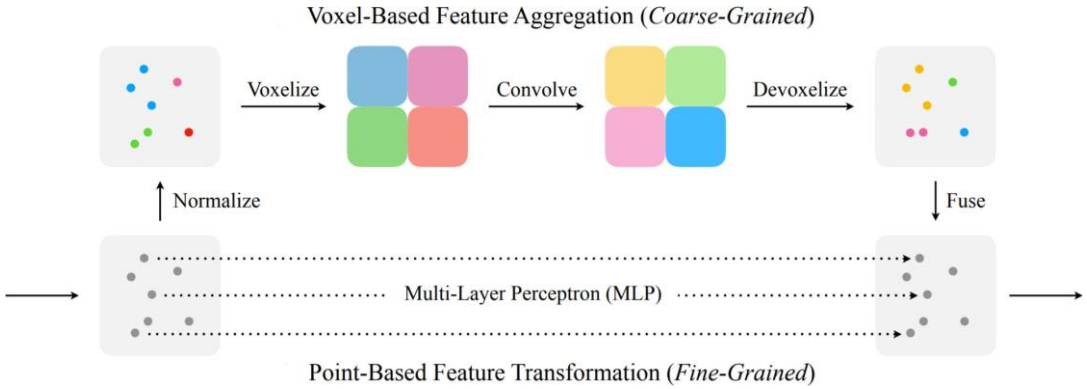


Figure 56 PVCNN architecture (Liu, 2019)

As seen in the literature review, PCRNet there are multiple ways that attention layer in transformers can be applied, such as in Figure 59. Some of the best-performing point segmentation algorithms use transformers, which are types of self-attention modules. Zhao et al. (2021) the Point Transformer, which is the current state-of-the-art for point cloud segmentation with a 73.5% mIoU on S3DIS Benchmark (Armeni, 2017). The point transformer is a module based on a vector self-attention

operator. The attention is described by letting $\chi = \{x_i\}$ be a set of feature vectors where

the vector self-attention layer would be represented by Equation 16 (Zhao, 2021):

$$\mathbf{y}_i = \sum_{\mathbf{x}_j \in \mathcal{X}(i)} \rho\left(\gamma\left(\varphi\left(\mathbf{x}_i\right) - \psi\left(\mathbf{x}_j\right) + \delta\right)\right) \odot \left(\alpha\left(\mathbf{x}_j\right) + \delta\right) \tag{16}$$

Where $\chi(i) \subseteq \chi$ is a set of points in of k nearest points of $\mathbf{x}_i$ and $\mathbf{y}_i$ is the learned feature, δ is the position encoding function defined as $\delta = \theta(\mathbf{p}_i - \mathbf{p}_j)$ where θ is an MLP that encodes the position difference between points $\mathbf{p}_i$ and $\mathbf{p}_j$, ρ is the normalization function and φ, ψ, α are linear projections. φ is an MLP that maps the attention vectors for feature aggregation using two linear layers and a ReLU output. The transformer layer or module(Figure 59) is based on an input of feature vectors $\mathbf{x}$ associated with the point $\mathbf{p}$, this input is then used as seen in Equation 16 to obtain the features vector $\mathbf{y}_i$ passed to the linear projections φ, ψ, α, and the points to the positional encoding MLP δ. The point encoding is added to both linear projections and passed to the γ MLP of two linear layers and one ReLU layer.

Finally, the α linear projection is aggregated with a dot product which results in the feature $\mathbf{y}$. The transformer module can be used in a segmentation network such as U-Net shown in Figure 49 and explained in section C.4. The internal modifications for point references require three main components: Point transformer block, downsampling called transition down in Figure 49 and upsampling called transition up in Figure 49. As explained in Appendix Figure 59, the transformer block is based on an input of all points and features, using two linear projections and the point transformer. The downsampling uses the transformer layer input and passes it through the farthest point sampling for $\mathbf{p}_i$. The sample is used as a neighborhood by passing it through a kNN, MLP, and a local max-pooling ending with the output of the feature y and the point subset and the feature for those points. The upsampling is done by taking the output of the previous transformer and the point transformer outputs from the previous specific size and passing them through a linear projection, an interpolation, and summation with the linear projection of the output of the previous point transformer layer. Figure 49 shows how the U-Net jumps the downsampling point transformations and the previous layer also being point transformation layers are passed to the upsampling. These transformations are finally passed through the MLP of N points and color dimensions. The transformer network is currently state-of-the-art and is planned to be the implementation and training basis for this thesis.

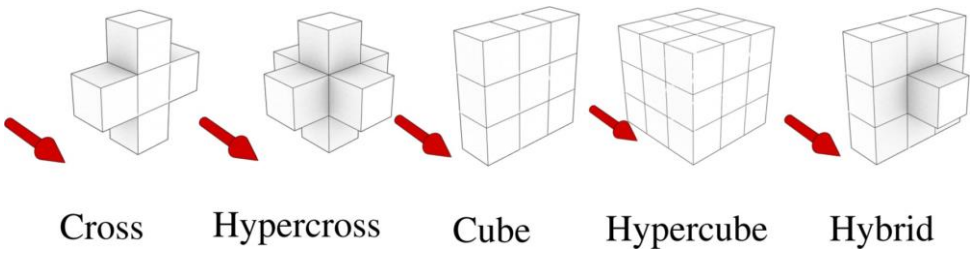


Figure 57. Hyper Dimensional Kernels for the Minkowski Engine (Choy, 2019)

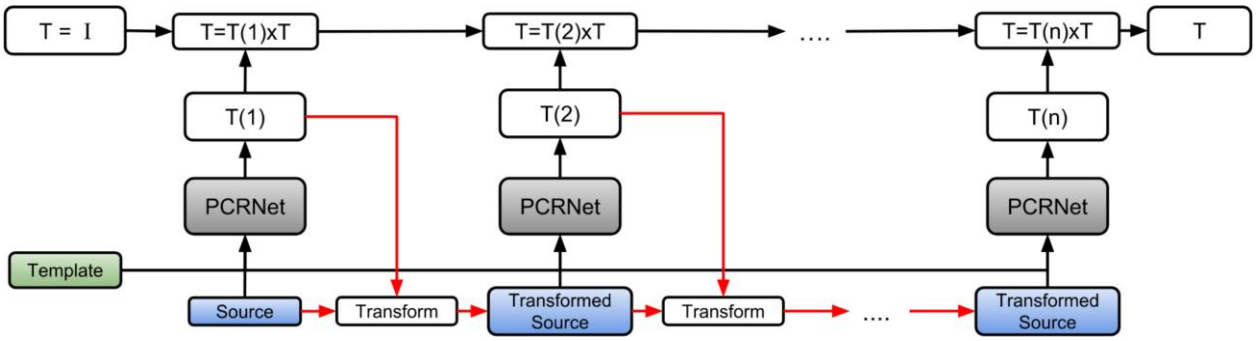


Figure 58 Transformer Layer (Zhao, 2021)

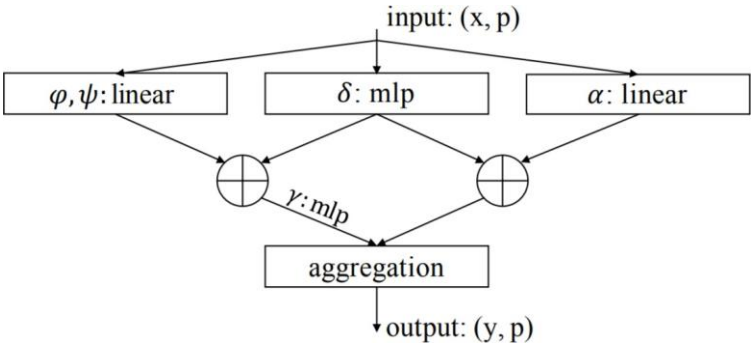


Figure 59 Transformer Layer (Zhao, 2021)

Appendix D

Statement of Contributions

The research in this chapter was completed under the supervision of Dr. Carl Haas and Dr. Sriram Narasimhan at the University of Waterloo. I am the primary contributor of the specific object finding Python code implementation.

Code for Specific Object Finding methodology:

```
# Import necessary libraries
import cv2
import warnings

warnings.filterwarnings("ignore")
import sys, os

old_stdout = sys.stdout
from decimal import Decimal
# Custom printing function to redirect output
def printing(text):
    sys.stdout = old_stdout
    print(text)
    sys.stdout = open(os.devnull, 'w')

sys.stdout = open(os.devnull, 'w')
# Import additional libraries
import gc
import numpy as np
import argparse
import open3d as o3d
import copy
```

```
import decimal
from os.path import exists
from urllib.request import urlretrieve
from visualization import get_colored_point_cloud_feature
from misc import extract_features
import time
from resunet import ResUNetBN2C
from os import listdir
import torch
import MinkowskiEngine as ME
from minkunet import MinkUNet34C
import matplotlib.pyplot as plt

# Check if the weights and file exist and download
device = torch.device('cuda')

scene_file = '2asci.ply'
weight_file = 'weights.pth'
action = 'store_true'
# Define class labels and color maps
CLASS_LABELS2 = ['wall', 'floor', 'cabinet', 'bed', 'chair', 'sofa', 'table',
                 'door', 'window', 'bookcase', 'picture', 'counter', 'desk',
                 'curtain', 'refrigerator', 'shower curtain', 'b', 'sink',
                 'bathtub', 'otherfurniture']
CLASS_LABELS = ['ceiling', 'floor', 'wall', 'bed', 'column', 'window', 'door',
                'table', 'chair', 'sofa', 'bookcase', 'whiteboard', 'otherfurniture',
                'curtain', 'refrigerator', 'shower curtain', 'b', 'sink',
                'bathtub', 'otherfurniture']

VALID_CLASS_IDS = [
```

```
    1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 14, 16, 24, 28, 33, 34, 36, 39
]


SCANNET_COLOR_MAP = {
    0: (0., 0., 0.),
    1: (174., 199., 232.),  # ceiling
    2: (152., 223., 138.),  # floor
    3: (31., 119., 180.),  # cabinet   wall
    4: (255., 187., 120.),  # bed
    5: (188., 189., 34.),  # column
    6: (140., 86., 75.),  # window
    7: (255., 152., 150.),  # door
    8: (214., 39., 40.),  # table
    9: (197., 176., 213.),  # chair
    10: (148., 103., 189.),  # sofa
    11: (196., 156., 148.),  # picture   bookcase
    12: (23., 190., 207.),  # whiteboard
    14: (247., 182., 210.),  # otherfurniture
    15: (66., 188., 102.),  # curtain
    16: (219., 219., 141.),  # refrigerator
    17: (140., 57., 197.),  # shower curtain
    18: (202., 185., 52.),  # b
    19: (51., 176., 203.),  # sink
    20: (200., 54., 131.),  # bathtub
    21: (92., 193., 61.),  #
    22: (78., 71., 183.),
    23: (172., 114., 82.),
    24: (255., 127., 14.),
    25: (91., 163., 138.),
    26: (153., 98., 156.),
```

```
    27: (140., 153., 101.),
    28: (158., 218., 229.),
    29: (100., 125., 154.),
    30: (178., 127., 135.),
    32: (146., 111., 194.),
    33: (44., 160., 44.),
    34: (112., 128., 144.),
    35: (96., 207., 209.),
    36: (227., 119., 194.),
    37: (213., 92., 176.),
    38: (94., 106., 211.),
    39: (82., 84., 163.),
    40: (100., 85., 144.),
}



# Function to convert txt file to ply format


def txt_2_ply(file_name):
    if (exists(file_name[:-4] + '.ply')):
        pass
    else:
        new_ply = open(file_name[:-4] + '.ply', 'w')
        nonply_lines = open(file_name, 'r').readlines()
        new_ply.write('ply\n')
        new_ply.write('format ascii 1.0\n')
        new_ply.write('comment VCGLIB generated\n')
        new_ply.write('element vertex ' + str(len(nonply_lines)) + '\n')
        new_ply.write('property float x\n')
        new_ply.write('property float y\n')
```

```
        new_ply.write('property float z\n')
        new_ply.write('property uchar red\n')
        new_ply.write('property uchar green\n')
        new_ply.write('property uchar blue\n')
        new_ply.write('property uchar alpha\n')
        new_ply.write('element face 0\n')
        new_ply.write('property list uchar int vertex_indices\n')
        new_ply.write('end_header\n')

        for line in nonply_lines:
            aline = ''
            for i, number in enumerate(line.split(' ')):
                if i < 3:
                    nnumber = decimal.Decimal(number).normalize()
                    aline = aline + str(nnumber) + ' '
                elif i == 5:
                    nnumber = number[:-8]
                    aline = aline + nnumber + ' '

                else:
                    nnumber = number[:-7]
                    aline = aline + nnumber + ' '
            new_ply.write(aline + '255 \n')
        new_ply.close()

# Function to load point cloud file
def load_file(file_name):
    if file_name.endswith('.pcd'):
        # print("got pcd")
        pcd = o3d.io.read_point_cloud(file_name)
```

```
        coords = np.array(pcd.points)
        colors = np.array(pcd.colors)
        return coords, colors, pcd
    elif file_name.endswith('.ply'):
        # print('got ply or obj')
        mesh = o3d.io.read_triangle_mesh(file_name)
        pts = len(mesh.vertices)
        # print(file_name + "points:" + str(pts))
        try:
            pcd = mesh.sample_points_poisson_disk(pts / 100)
            coords = np.array(pcd.points)
            colors = np.array(pcd.colors)
            return coords, colors, pcd
        except:
            pcd = o3d.io.read_point_cloud(file_name)
            coords = np.array(pcd.points)
            colors = np.array(pcd.colors)
            return coords, colors, pcd
    elif file_name.endswith('.txt'):
        txt_2_ply(file_name)
        load_file(file_name[:-3] + 'ply')
    else:
        print(file_name + " not recognized")


# Function to resample and perform ICP registration
def resample_icp(original_pcd, visualize):
    nnd = np.array(original_pcd.compute_nearest_neighbor_distance())
    value = nnd.max()
    alpha =value
    tetra_mesh, pt_map = o3d.geometry.TetraMesh.create_from_point_cloud(original_pcd)
```

```
    mesh = o3d.geometry.TriangleMesh.create_from_point_cloud_alpha_shape(original_pcd, alpha, tetra_mesh, pt_map)
    R = mesh.get_rotation_matrix_from_xyz((np.pi / 2, 0, np.pi / 4))
    mesh.rotate(R, center=(1, 0, 0))
    pcd_temp = mesh.sample_points_uniformly(int(len(mesh.vertices) / 5))
    pcd_temp.paint_uniform_color([1, 0.0, 0.0])
    pcd_temp.translate((1, 0.5, 0))

    if (visualize):
        o3d.visualization.draw_geometries([mesh])
        o3d.visualization.draw_geometries([pcd_temp, original_pcd])
    voxel_sizes = value * 0.6

    sources, targets, source_downs, target_downs, source_fpfhs, target_fpfhs = prepare_dataset(original_pcd, pcd_temp,
                                                                                                voxel_sizes, False)

    result_ransacs = execute_global_registration(source_downs, target_downs,
                                                 source_fpfhs, target_fpfhs,
                                                 voxel_sizes)
    result_icp = color_registration(sources, targets, result_ransacs, source_fpfhs, target_fpfhs, voxel_sizes)
    # printing(str(result_icp))
    return result_icp


# Function to normalize color values
def normalize_color(color: torch.Tensor, is_color_in_range_0_255: bool = False) -> torch.Tensor:
    r"""
    Convert color in range [0, 1] to [-0.5, 0.5]. If the color is in range [0,
    255], use the argument `is_color_in_range_0_255=True`.

    `color` (torch.Tensor): Nx3 color feature matrix
```

```
    `is_color_in_range_0_255` (bool): If the color is in range [0, 255] not [0, 1], normalize the color to [0, 1].
    """
    if is_color_in_range_0_255:
        color /= 255
    color -= 0.5
    return color.float()

# Function to concatenate two point clouds
def concatenate_pointclouds(pcd1, pcd2):
    final_pcd = o3d.geometry.PointCloud()
    pts1 = np.asarray(pcd1.points)
    colors1 = np.asarray(pcd1.colors)
    pts2 = np.asarray(pcd2.points)
    colors2 = np.asarray(pcd2.colors)

    concat_points = np.concatenate((pts1, pts2), axis=0)
    concat_colors = np.concatenate((colors1, colors2), axis=0)
    final_pcd.points = o3d.utility.Vector3dVector(concat_points)
    final_pcd.colors = o3d.utility.Vector3dVector(concat_colors)
    return final_pcd

# Function to create segmentation (placeholder)
def create_segmentation(scene_file, visualize):
    o3d.io.read_triangle_mesh(scene_file)

# Function to create segmentation using a trained model
def create_segmentation2(scene_file, visualize):
    # Create Segmentation

    # print(f"Using {device}")
```

```
# Define a model and load the weights
model = MinkUNet34C(3, 20).to(device)
model_dict = torch.load(weight_file)
model.load_state_dict(model_dict)
model.eval()

# Load Trained Network
coords, colors, pcd = load_file(scene_file)
if visualize:
    o3d.visualization.draw_geometries([pcd])

# Segment point cloud
with torch.no_grad():
    voxel_size = 0.02
    # Feed-forward pass and get the prediction
    in_field = ME.TensorField(
        features=normalize_color(torch.from_numpy(colors)),
        coordinates=ME.utils.batched_coordinates([coords / voxel_size], dtype=torch.float32),
        quantization_mode=ME.SparseTensorQuantizationMode.UNWEIGHTED_AVERAGE,
        minkowski_algorithm=ME.MinkowskiAlgorithm.SPEED_OPTIMIZED,
        device=device,
    )
    # Convert to a sparse tensor
    sinput = in_field.sparse()
    # Output sparse tensor
    soutput = model(sinput)
    # get the prediction on the input tensor field
    out_field = soutput.slice(in_field)
    logits = out_field.F
```

```
    _, pred = logits.max(1)
    pred = pred.cpu().numpy()
    # print(pred)
    # Create a point cloud file
    pred_pcd = o3d.geometry.PointCloud()
    # Map color
    colors = []
    for l in pred:
        my_valid_class = VALID_CLASS_IDS[l]
        my_color = SCANNET_COLOR_MAP[my_valid_class]
        colors.append(my_color)
    colors = np.array(colors)
    pred_pcd.points = o3d.utility.Vector3dVector(coords)
    pred_pcd.colors = o3d.utility.Vector3dVector(colors / 255)
    o3d.io.write_point_cloud(scene_file[:-4] + "_segmented.ply", pred_pcd,
                             write_ascii=True)
    if visualize:
        o3d.visualization.draw_geometries([pred_pcd])
    return pred_pcd



# Function to split segmentation based on object index
def split_segmentation(pred_pcd, object_index, visualize):
    target_color = np.array(SCANNET_COLOR_MAP[object_index]) / 255  #bookcase (148, 103, 189)  Tubes (255, 188, 0)
    points = np.array(pred_pcd.points)
    colors = np.array(pred_pcd.colors)

    mask = np.all(colors == target_color, axis=1)
    newpoints = points[mask]
```

```
    newcolors = colors[mask]


    final_pcd = o3d.geometry.PointCloud()
    final_pcd.points = o3d.utility.Vector3dVector(newpoints)
    final_pcd.colors = o3d.utility.Vector3dVector(newcolors)
    if visualize:
        o3d.visualization.draw_geometries([final_pcd])
    return final_pcd


# Function to cluster point clouds using DBSCAN
def cluster_pointclouds(segmented_pcd, visualize):
    nnd = np.array(segmented_pcd.compute_nearest_neighbor_distance())
    eps = nnd.max()
    with o3d.utility.VerbosityContextManager(
            o3d.utility.VerbosityLevel.Debug) as cm:
        labels = np.array(
            segmented_pcd.cluster_dbscan(eps=eps, min_points=7,
                                         print_progress=False))  # Set the eps depending on the sparcity
    max_label = labels.max()
    # print(f"point cloud has {max_label + 1} clusters")
    colors = plt.get_cmap("tab20")(labels / (max_label if max_label > 0 else 1))
    colors[labels < 0] = 0
    segmented_pcd.colors = o3d.utility.Vector3dVector(colors[:, :3])
    if visualize:
        o3d.visualization.draw_geometries([segmented_pcd])


    separated_points = {}
    for i, point in enumerate(segmented_pcd.points):
        col = segmented_pcd.colors[i]
        col = hash(tuple(col))
```

```
        if col in separated_points:
            separated_points[col].append(list(point))
        else:
            separated_points[col] = [list(point)]
    return separated_points


# Function to extract FCGF features
def FCGF(npcd):
    checkpoint = torch.load('ResUNetBN2C-16feat-3conv.pth')
    model = ResUNetBN2C(1, 16, normalize_feature=True, conv1_kernel_size=3, D=3)
    model.load_state_dict(checkpoint['state_dict'])
    model.eval()
    model = model.to(device)
    # Get Exact points
    # exact_points =  final_pcd#segmented_pcd#o3d.io.read_point_cloud(config.input)
    nnd = np.array(npcd.compute_nearest_neighbor_distance())
    voxel_size = nnd.max()
    try:
        xyz_down, feature = extract_features(
            model,
            xyz=np.array(npcd.points),
            voxel_size=voxel_size,
            device=device,
            skip_check=True)
    except:
        return npcd


    nvis_pcd = o3d.geometry.PointCloud()
    nvis_pcd.points = o3d.utility.Vector3dVector(xyz_down)
    nvis_pcd = get_colored_point_cloud_feature(nvis_pcd,
```

```
                                feature.detach().cpu().numpy(),
                                voxel_size)
    return nvis_pcd


# Function to visualize registration results
def draw_registration_result(source, target, transformation):
    source_temp = copy.deepcopy(source)
    target_temp = copy.deepcopy(target)
    source_temp.transform(transformation)
    o3d.visualization.draw_geometries([source_temp, target_temp])


# Function to preprocess point cloud for registration
def preprocess_point_cloud(pcd, voxel_size):
    # print(":: Downsample with a voxel size %.3f." % voxel_size)
    pcd_down = pcd.voxel_down_sample(voxel_size)


    radius_normal = voxel_size * 2
    # print(":: Estimate normal with search radius %.3f." % radius_normal)
    pcd_down.estimate_normals(
        o3d.geometry.KDTreeSearchParamHybrid(radius=radius_normal, max_nn=30))


    radius_feature = voxel_size * 5
    # print(":: Compute FPFH feature with search radius %.3f." % radius_feature)
    pcd_fpfh = o3d.pipelines.registration.compute_fpfh_feature(
        pcd_down,
        o3d.geometry.KDTreeSearchParamHybrid(radius=radius_feature, max_nn=100))
    return pcd_down, pcd_fpfh


# Function to prepare dataset for registration
def prepare_dataset(source, target, voxel_size, visualize, transforms=True):
```

```
    # print(":: Load two point clouds and disturb initial pose.")
    trans_init = np.asarray([[0.0, 0.0, 1.0, 0.0], [1.0, 0.0, 0.0, 0.0],
                             [0.0, 1.0, 0.0, 0.0], [0.0, 0.0, 0.0, 1.0]])
    if transforms:
        source.transform(trans_init)
    if (visualize):
        draw_registration_result(source, target, np.identity(4))

    source_down, source_fpfh = preprocess_point_cloud(source, voxel_size)
    target_down, target_fpfh = preprocess_point_cloud(target, voxel_size)
    return source, target, source_down, target_down, source_fpfh, target_fpfh


# Function to execute global registration using RANSAC
def execute_global_registration(source_down, target_down, source_fpfh,target_fpfh, voxel_size):
    distance_threshold = voxel_size * 1.5
    # print(":: RANSAC registration on downsampled point clouds.")
    # print("   Since the downsampling voxel size is %.3f," % voxel_size)
    # print("   we use a liberal distance threshold %.3f." % distance_threshold)
    result = o3d.pipelines.registration.registration_ransac_based_on_feature_matching(
        source_down, target_down, source_fpfh, target_fpfh, True,
        distance_threshold,
        o3d.pipelines.registration.TransformationEstimationPointToPoint(False),
        3, [
            o3d.pipelines.registration.CorrespondenceCheckerBasedOnEdgeLength(
                0.9),
            o3d.pipelines.registration.CorrespondenceCheckerBasedOnDistance(
                distance_threshold)
        ], o3d.pipelines.registration.RANSACConvergenceCriteria(100000, 0.999))

    return result
```

```
# Function to refine registration using ICP
def refine_registration(source, target, result_ransac, source_fpfh, target_fpfh, voxel_size):
    distance_threshold = voxel_size-0.01
    # print(":: Point-to-plane ICP registration is applied on original point")
    # print("   clouds to refine the alignment. This time we use a strict")
    # print("   distance threshold %.3f." % distance_threshold)
    result = o3d.pipelines.registration.registration_icp(
        source, target, distance_threshold, result_ransac.transformation,
        o3d.pipelines.registration.TransformationEstimationPointToPlane())
    return result


# Function to perform registration pipeline
def pipeline_registration(source, target):
    nnd = np.array(source.compute_nearest_neighbor_distance())
    voxel_size = nnd.max()
    source_down, source_fpfh = preprocess_point_cloud(source, voxel_size)
    target_down, target_fpfh = preprocess_point_cloud(target, voxel_size)
    result_ransac = execute_global_registration(source_down, target_down,
                                        source_fpfh, target_fpfh,
                                        voxel_size)


    result_icp = color_registration(source, target, result_ransac, source_fpfh, target_fpfh,
                                voxel_size)
    return result_icp




# Function to reassess point cloud density
def reassess(target, pointclouds):
```

```
    total_density = 0
    avg_density = 0
    for i,points in enumerate(pointclouds):
        pcds = o3d.geometry.PointCloud()
        pcds.points = o3d.utility.Vector3dVector(np.array(pointclouds[points]))
        nnd = np.array(pcds.compute_nearest_neighbor_distance())
        density = nnd.max()
        total_density = total_density+ density
        avg_density = total_density/i
    nnd = np.array(target.compute_nearest_neighbor_distance())
    density2 = nnd.max()
    n_target = target.uniform_down_sample(int(avg_density/density2))
    #o3d.visualization.draw_geometries([target])
    #o3d.visualization.draw_geometries([n_target])
    return n_target



# Function to perform color-based registration
def color_registration(source, target, result_ransac, source_fpfh, target_fpfh, voxel_size):
    distance_threshold = voxel_size * 1.4
    # print(":: Point-to-plane ICP registration is applied on original point")
    # print("   clouds to refine the alignment. This time we use a strict")
    # print("   distance threshold %.3f." % distance_threshold)
    # o3d.visualization.draw_geometries([source,target])
    try:
        result1 = o3d.pipelines.registration.registration_colored_icp(
            source, target, distance_threshold, result_ransac.transformation,
            o3d.pipelines.registration.TransformationEstimationForColoredICP())
        print("Color ICP")
    except:
```

```
        print("Regular ICP")

        result1 = o3d.pipelines.registration.registration_icp(
            source, target, distance_threshold, result_ransac.transformation,
            o3d.pipelines.registration.TransformationEstimationPointToPoint())
    return result1

# Function to detect objects in a point cloud
def detect_object(ply, visualize):
    # Create Segmentation

    # print(f"Using {device}")

    # Define a model and load the weights
    model = MinkUNet34C(3, 20).to(device)
    model_dict = torch.load(weight_file)
    model.load_state_dict(model_dict)
    model.eval()

    # Load Trained Network
    coords, colors, pcd = load_file(ply)
    # Segment point cloud
    with torch.no_grad():
        voxel_size = 0.02
        # Feed-forward pass and get the prediction
        in_field = ME.TensorField(
            features=normalize_color(torch.from_numpy(colors)),
            coordinates=ME.utils.batched_coordinates([coords / voxel_size], dtype=torch.float32),
            quantization_mode=ME.SparseTensorQuantizationMode.UNWEIGHTED_AVERAGE,
            minkowski_algorithm=ME.MinkowskiAlgorithm.SPEED_OPTIMIZED,
```

```
        device=device,
    )
    # Convert to a sparse tensor
    sinput = in_field.sparse()
    # Output sparse tensor
    soutput = model(sinput)
    # get the prediction on the input tensor field
    out_field = soutput.slice(in_field)
    logits = out_field.F

_, pred = logits.max(1)
pred = pred.cpu().numpy()
# print(pred)
# Create a point cloud file
pred_pcd = o3d.geometry.PointCloud()
# Map color
colors = []
for l in pred:
    colors.append(SCANNET_COLOR_MAP[VALID_CLASS_IDS[l]])
colors = np.array(colors)
pred_pcd.points = o3d.utility.Vector3dVector(coords)
pred_pcd.colors = o3d.utility.Vector3dVector(colors / 255)
if visualize:
    o3d.visualization.draw_geometries([pred_pcd])
vals, counts = np.unique(pred, return_counts=True)
index = np.argmax(counts)
index_object = vals[index] + 1

# print(index_object)
```

```
    return index_object


# Main execution block
# This code implements a 3D object detection and localization pipeline for point cloud data.
# It processes a dataset of 3D scenes, attempting to find specific objects within each scene.
# The pipeline includes steps for loading point clouds, segmentation, clustering, and registration.
# It uses deep learning models (like MinkUNet) for semantic segmentation and feature extraction.
# The core algorithm iterates through scenes, segments them, and tries to match known objects
# using various registration techniques (ICP, RANSAC). It handles multiple object classes and
# uses color information when available. The code includes options for visualization and debugging,
# and it saves detailed results including transformations and statistics. This pipeline is likely
# used in robotics or computer vision applications for 3D scene understanding and object localization.
if __name__ == '__main__':
    debugging = False
    visualizing = False
    last_visualization = False
    network = True



    skip_objects = ['floor', 'ceiling', 'wall', 'column','window']
    segmenteds = './Results2'
    area_list = listdir(segmenteds)
    directory = './Stanford3dDataset_v1.2_Aligned_Version/'
    i = 0
    flag = False
    objects = []
    objects_found = []
    transformations = {}

    for area in area_list:
```

```
# if area == 'Area_4_office_7.ply':
#     flag = True
# if flag:
#     pass
# else:
#     continue
a, area_number, title, place_number = area.split('_')
print(directory + a + "_" + area_number + '/' + title + '_' + place_number[:-4] + '/Annotations')
file_list = listdir(directory + a + "_" + area_number + '/' + title + '_' + place_number[:-4] + '/Annotations')
for file in file_list:
    # if file == 'bookcase_2.ply':
    #     pass
    # else:
    #     continue
    if (file.endswith('.ply')):
        objects.append(file)
        object_file = directory + a + "_" + area_number + '/' + title + '_' + place_number[
                                                    :-4] + '/Annotations/' + file
        # pcd = o3d.io.read_point_cloud(object_file, format='ply')
        # o3d.visualization.draw([pcd],show_skybox=False,point_size=2,width=1920, height=1080, bg_color=(255,255,255,255))
        scene_file = directory + a + "_" + area_number + '/' + title + '_' + place_number[
                                                    :-4] + '/' + title + '_' + place_number


        # object_file = "./ResearchPointClouds/office_8/Annotations/bookcase_2.ply"
        # scene_file = "./ResearchPointClouds/office_8/office_8.ply"



        print(object_file)
        print(scene_file)
        object_ID = 0
```

```
'''
#User input strategy
object_name = input("Please tell me what object is it?\n")
if (object_name in CLASS_LABELS):
    pass
else:

    while object_name not in CLASS_LABELS:
        print("Object not learned yet")
        object_name = input("Please tell me what learned object is it?\n")

'''

obj_name = object_file.rfind('/')
object_name = object_file[object_file.rfind('/') + 1:object_file.rfind('_', obj_name)]
if (object_name in CLASS_LABELS):
    if object_name not in skip_objects:
        printing(object_file + "-" + scene_file)
        object_ID = list(CLASS_LABELS).index(object_name) + 1
        my_valid_class = VALID_CLASS_IDS[object_ID]
        my_color = SCANNET_COLOR_MAP[my_valid_class]
        # main(object_file, area, scene_file, my_valid_class)
        times = []
        scene = scene_file[scene_file.rfind('/') + 1:-4]
        print(scene_file, scene)
        obj_name = object_name[object_name.rfind('/') + 1:-4]
        npoints, ncolors, object_pcd = load_file(object_file)  # Wanted object to be found
        npcd = copy.deepcopy(object_pcd)
        trans_init = np.asarray([[0.0, 0.0, 1.0, 0.0], [1.0, 0.0, 0.0, 0.0],
```

```
                    [0.0, 1.0, 0.0, 0.0], [0.0, 0.0, 0.0, 1.0]])
npcd.transform(trans_init)
averagesummed = []
averagetrans = []
found = False
for idx in range(3):
    if found == True:
        continue
    npcd.estimate_normals()
    if visualizing:
        o3d.visualization.draw([object_pcd], show_skybox=False, point_size=4, width=1920,
                               height=1080, bg_color=(255, 255, 255, 255))
    obj_pcd = copy.deepcopy(npcd)
    acolor = list(np.array(SCANNET_COLOR_MAP[object_ID]) / 255)
    obj_pcd.paint_uniform_color(acolor)
    if visualizing:
        # obj_pcd = o3d.io.read_point_cloud(object_name, format='ply')
        # obj_pcd = copy.deepcopy(npcd)
        # acolor = list(np.array(SCANNET_COLOR_MAP[object_ID]) / 255)
        # obj_pcd.paint_uniform_color(acolor)
        o3d.visualization.draw([obj_pcd], show_skybox=False, point_size=4, width=1920,
                               height=1080, bg_color=(255, 255, 255, 255))
    correspondence_threshold = 0.8
    resampled_icp_result = resample_icp(copy.deepcopy(npcd), visualizing)
    resampled_statistics = [resampled_icp_result.fitness*correspondence_threshold, resampled_icp_result.inlier_rmse*1.1,
                            int(len(np.asarray(resampled_icp_result.correspondence_set))*correspondence_threshold)]
    if debugging == True:
        printing("Resampled" + str(resampled_icp_result))
    # obj_ID = detect_object(object_file, visualization)
    start = time.time()
```

```
try:
    npoints, ncolors, pred_pcd = load_file(
        './Results2/' + area)  # create_segmentation(area, visualization)
    if visualizing:
        o3d.visualization.draw([pred_pcd], show_skybox=False, point_size=2, width=1920,
                               height=1080, bg_color=(255, 255, 255, 255))

    segmented_pcd = split_segmentation(pred_pcd, object_ID, visualizing)
except:
    #printing("Could not do " + area + object_name)
    continue
clustered_pointclouds = cluster_pointclouds(segmented_pcd, visualizing)

print("RESAMPLED ICP", resampled_icp_result.fitness)

target = npcd
if network:
    nvis_pcd = FCGF(npcd)
    nvis_pcd.estimate_normals()
    target = nvis_pcd
if visualizing:
    o3d.visualization.draw([target], show_skybox=False, point_size=3, width=1920,
                           height=1080, bg_color=(255, 255, 255, 255))

closest_match = 0
statistics = []
assets = []
candidates = len(clustered_pointclouds)
for i, segmentation in enumerate(clustered_pointclouds):
    # print('----------------------------',i)
```

```
apcd = o3d.geometry.PointCloud()
points = np.array(clustered_pointclouds[segmentation])
if debugging == True:
    printing('Candidate points:'+str(len(points)) + ' Min Points:' + str(len(object_pcd.points) * 0.2))
if len(points) < len(object_pcd.points) * 0.2:
    continue
apcd.points = o3d.utility.Vector3dVector(points)
apcd.estimate_normals()
source = apcd
if network:
    vis_pcd = FCGF(apcd)
    vis_pcd.estimate_normals()
    source = vis_pcd
if visualizing:
    o3d.visualization.draw_geometries([source])
nnd = np.array(pred_pcd.compute_nearest_neighbor_distance())
voxel_size = nnd.max()# means 5cm for this dataset

result_icp = pipeline_registration(source, target)
result_icp2 = pipeline_registration(target, source)

avg_fit = np.average([result_icp.fitness, result_icp2.fitness])
avg_inlier = np.min([result_icp.inlier_rmse, result_icp2.inlier_rmse])
avg_corr = np.average([len(np.asarray(result_icp.correspondence_set)),
                       len(np.asarray(result_icp2.correspondence_set))])
if debugging == True:
    printing('AVG '+str(avg_fit) + ' ' + str(avg_inlier) + ' ' + str(avg_corr))
    printing("RE" + str(resampled_statistics) + '\n' + str(result_icp) + '\n' + str(
        result_icp2))
```

```
print("Color Registration Results:" + str(result_icp))


if (result_icp == ''):
    print("No Correspondences")
else:

    print("RESULTS", result_icp.fitness, result_icp.inlier_rmse,
          len(np.asarray(result_icp.correspondence_set)))
    print("ICP:", result_icp, result_icp.transformation)
    if visualizing or debugging:
        o3d.visualization.draw_geometries([source,target])
        draw_registration_result(source, target, result_icp.transformation)
    plusses = 0

    if (statistics == []):
        statistics = [result_icp.fitness, result_icp.inlier_rmse,
                      len(np.asarray(result_icp.correspondence_set))]

    print("STATISTICS" + str(statistics))
    txtlen = len("ments.  Use numpy.asarray() to access data.")
    correspondence_threshold = 1
    correspondences_percentage = 0.1
    threshold = 0.2
    if debugging == True:
        printing("RE" + str(resampled_icp_result.fitness)[:4] + ' ' + str(
            resampled_icp_result.inlier_rmse)[:4] + ' ' +
                str(resampled_icp_result.correspondence_set)[-8 - txtlen:-txtlen])
        printing(
            "AV" + str(avg_fit)[:4] + ' ' + str(avg_inlier)[:4] + ' ' + str(avg_corr)[:6])
```

```
    printing("AV" + str(
        avg_fit > resampled_icp_result.fitness * correspondence_threshold) + ' ' +
            str(avg_inlier * correspondence_threshold < resampled_icp_result.inlier_rmse) + ' ' + str(avg_inlier < 0.02)+' '+
            str(avg_corr > len(
                np.asarray(
                    resampled_icp_result.correspondence_set)) * correspondences_percentage))


if visualizing:
    draw_registration_result(source, target, result_icp.transformation)
if avg_fit > resampled_icp_result.fitness * correspondence_threshold:
    if avg_inlier * correspondence_threshold < resampled_icp_result.inlier_rmse or avg_inlier < 0.02:
        if avg_corr > len(
                np.asarray(
                    resampled_icp_result.correspondence_set)) * correspondences_percentage:
            print("ADDED Asset!" + obj_name)
            #Visualize how close the point cloud is from the true position
            '''
            print("Transform", result_icp.transformation, result_ransac.transformation)
            mesh_tx = copy.deepcopy(npcd)


            mesh_tx.paint_uniform_color([1, 0, 0])
            coorsds, coslors, mainpcd = load_file(scene_name)
            #amain_pcd = concatenate_pointclouds(mesh_tx, mainpcd)
            key = list(clustered_pointclouds.keys())[i]
            points = np.array(clustered_pointclouds[key])
            the_pcd = o3d.geometry.PointCloud()
            the_pcd.points = o3d.utility.Vector3dVector(points)
            the_pcd.paint_uniform_color([0, 1, 0])
            o3d.visualization.draw_geometries([mainpcd,mesh_tx, the_pcd])
            mesh_tx.transform(result_icp.transformation)
```

```
            o3d.visualization.draw_geometries([mainpcd, mesh_tx, the_pcd])
            mesh_tx.transform(result_ransac.transformation)
            o3d.visualization.draw_geometries([mainpcd, mesh_tx, the_pcd])
            '''
            assets.append(i)
            objects_found.append(file)
            printing('Passed'+str(file))
            found= True
            print("Correspondences: " + str(
              len(np.asarray(result_icp.correspondence_set))) + " " +
               str(len(np.asarray(
                 resampled_icp_result.correspondence_set)) * correspondence_threshold))
        else:
            print("Correspondence Too Low")
            print(
              "Correspondences: " + str(
                len(np.asarray(result_icp.correspondence_set))) + " " +
              str(len(
                 np.asarray(
                   resampled_icp_result.correspondence_set)) * correspondence_threshold))
    else:
        print("Inlier Error Too High")
        print("Inliers RMSE: " + str(result_icp.inlier_rmse) + " " + str(
          resampled_icp_result.inlier_rmse))
else:
    print("Fitness Too Low")


coords, colors, original_pcd = load_file(scene_file)
center = pred_pcd.get_center() - original_pcd.get_center()
original_pcd.translate(center)
```

```
# o3d.visualization.draw_geometries([pred_pcd,original_pcd,npcd])
best_pcd = o3d.geometry.PointCloud()

if (assets == []):
    print("No assets Found!")
    end = time.time()
    print("total time: ", end - start)
else:
    for index in assets:
        key = list(clustered_pointclouds.keys())[index]
        points = np.array(clustered_pointclouds[key])
        best_pcd.points = o3d.utility.Vector3dVector(points)
        best_pcd.paint_uniform_color([1, 0, 0])
        best_pcd.estimate_normals()

    original_pcd = concatenate_pointclouds(best_pcd, original_pcd)

    sce_ctr = original_pcd.get_center()
    sce_bounds = original_pcd.get_max_bound()

    obj_ctr = object_pcd.get_center()
    to_center = sce_ctr - obj_ctr
    object_pcd.translate(to_center)
    obj_bounds = object_pcd.get_max_bound()
    to_side = [0, sce_bounds[1] + obj_bounds[1], 0]

    object_pcd.translate(to_side)

    final_pcd = concatenate_pointclouds(original_pcd, object_pcd)
```

```
# o3d.visualization.draw_geometries([final_pcd])
if visualizing or last_visualization:
    o3d.visualization.draw_geometries([final_pcd])
o3d.io.write_point_cloud(
    "./Results/" + a + "_" + area_number + '_' + title + '_' + place_number[
                                                :-4] + '_' + file + ".ply",
    final_pcd,
    write_ascii=True)

end = time.time()
print("total time: ", end - start)
gc.collect()
torch.cuda.empty_cache()
time_taken = end - start
times.append(time_taken)
nnd = np.array(original_pcd.compute_nearest_neighbor_distance())
voxel_sizes = nnd.max()
nepts, _, nepcd = load_file(object_file)
scpts, _, ori_pcd = load_file(scene_file)
center = pred_pcd.get_center() - ori_pcd.get_center()
nepcd.translate(center)
ori_pcd.translate(center)
# o3d.visualization.draw_geometries([original_pcd, ori_pcd])
# o3d.visualization.draw_geometries([nepcd, ori_pcd])
# o3d.visualization.draw_geometries([nepcd, ori_pcd,best_pcd])
nsources, ntargets, nsource_downs, ntarget_downs, nsource_fpfhs, ntarget_fpfhs = prepare_dataset(
    nepcd, best_pcd,
    voxel_sizes, False, transforms=False)

result_ransacs = execute_global_registration(nsource_downs, ntarget_downs,
```

```
                                    nsource_fpfhs, ntarget_fpfhs,
                                    voxel_sizes)
    results = refine_registration(nepcd, best_pcd, result_ransacs, nsource_fpfhs,
                                  ntarget_fpfhs, voxel_sizes)
    transformations[area + object_name + str(idx)] = results.transformation
    print(results, "_-------------------------------------------------")



if (times == []):
    pass
else:
    afile = open("./Results/" + a + "_" + area_number + '_' + title + '_' + place_number[
                                                    :-4] + '_' + file + ".txt",
              'w')
    afile.write(scene + " and " + obj_name + "\n")
    afile.write("Time: "+str(time_taken)+ "\n")
    afile.write("ObjectVertices: " + str(len(nepts))+ "\n")
    afile.write("SceneVertices: " + str(len(scpts)) + "\n")
    afile.write("Candidates: " + str(candidates) + "\n")
    afile.write('Requrired threshold:'+str(resampled_icp_result)+ "\n")
    afile.write("First registration:"+str(result_icp)+ "\n")
    afile.write("Second registration:" + str(result_icp2)+ "\n")
    afile.write("Transformation Error:\n"+str(results.transformation))
    afile.close()
    continue
```

Appendix E

Statement of Contributions

The research in this chapter was completed under the supervision of Dr. Carl Haas and Dr. Sriram Narasimhan at the University of Waterloo. I am the primary contributor of the random placement dataset generator Python code implementation.

Code for random object placement dataset generation methodology:

```
#Import Necessary libraries
import os
import numpy as np
import open3d as o3d
from collections import defaultdict
from tqdm import tqdm
import random
import shutil


#Define Object properties
class PointCloudObject:
    def __init__(self, points, point_density, class_count, class_name):
        self.points = points
        self.point_density = point_density
        self.class_count = class_count
        self.class_name = class_name
        self.bounds = self.calculate_bounds()
        self.weight = self.calculate_weight()
        self.occurrences = 0  # Track occurrences in scenes


    def calculate_bounds(self):
```

```
        min_bounds = np.min(self.points[:, :3], axis=0)
        max_bounds = np.max(self.points[:, :3], axis=0)
        return min_bounds, max_bounds

    def get_dimensions(self):
        min_bounds, max_bounds = self.bounds
        return max_bounds - min_bounds

    def calculate_weight(self):
        return len(self.points) / self.point_density / self.class_count

    def adjust_weight(self, placed):
        if placed:
            self.weight /= 2
        else:
            self.weight *= 1.1

    def increment_occurrence(self):
        self.occurrences += 1


def get_class_counts(folder_path):
    class_counts = defaultdict(int)
    for root, _, files in os.walk(folder_path):
        class_name = os.path.basename(root)
        txt_files = [file for file in files if file.endswith('.txt')]
        class_counts[class_name] += len(txt_files)
    return class_counts
```

```
def load_point_clouds_from_folder(folder_path, point_density, class_counts):
    point_clouds = []
    for root, _, files in tqdm(os.walk(folder_path), desc="Loading Point Clouds"):
        for file_name in files:
            if file_name.endswith('.txt'):
                file_path = os.path.join(root, file_name)
                points = np.loadtxt(file_path)
                class_name = os.path.basename(root)
                class_count = class_counts.get(class_name, 1)
                point_clouds.append(PointCloudObject(points, point_density, class_count, class_name))
    return point_clouds



def place_objects_in_scene(point_clouds, max_objects, num_stories=3, story_height=3, padding=0.5):
    placed_objects = []
    for story in range(num_stories):
        current_position = np.array([0, 0, story * story_height])
        x_offset = 0
        y_offset = 0
        row_max_height = 0

        objects_per_story = max_objects // num_stories
        for i in range(objects_per_story):
            if i >= len(point_clouds):
                break
            point_cloud = point_clouds[i]
            dimensions = point_cloud.get_dimensions()

            # Check if the object can fit in the current row, otherwise move to the next row
            if x_offset + dimensions[0] > np.sqrt(objects_per_story) * (dimensions[0] + padding):
```

```
            x_offset = 0
            y_offset += row_max_height + padding
            row_max_height = 0

        # Place the object at the current position
        position = current_position.copy()
        position[0] += x_offset
        position[1] += y_offset
        placed_objects.append((point_cloud, position))
        point_cloud.adjust_weight(placed=True)

        # Update the current position for the next object placement
        x_offset += dimensions[0] + padding
        row_max_height = max(row_max_height, dimensions[2])

    return placed_objects


def distribute_objects_across_scenes(point_clouds, num_areas, rooms_per_area, max_objects_per_scene):
    scenes = []
    for _ in range(num_areas):
        area = []
        for _ in range(rooms_per_area):
            area.append([])
        scenes.append(area)

    total_rooms = num_areas * rooms_per_area
    for room_index in range(total_rooms):
        random.shuffle(point_clouds)  # Shuffle point_clouds in place
        room_objects = point_clouds[:max_objects_per_scene]
```

```
        for obj in room_objects:
            obj.adjust_weight(placed=True)
            obj.increment_occurrence()
        for obj in point_clouds:
            if obj not in room_objects:
                obj.adjust_weight(placed=False)
        area_index = room_index // rooms_per_area
        room_in_area_index = room_index % rooms_per_area
        scenes[area_index][room_in_area_index] = room_objects

    return scenes


def create_plane(vertices, color):
    plane = o3d.geometry.TriangleMesh()
    plane.vertices = o3d.utility.Vector3dVector(vertices)
    plane.triangles = o3d.utility.Vector3iVector([
        [0, 1, 2], [2, 3, 0]
    ])
    plane.compute_vertex_normals()
    plane.paint_uniform_color(color)
    return plane


def mesh_to_point_cloud(mesh, num_points=1000):
    sampled_points = mesh.sample_points_uniformly(number_of_points=num_points)

    # Get existing colors or create a default color if not present
    if len(sampled_points.colors) == 0:
        colors = np.ones((num_points, 3)) * 128  # Default to gray if no colors
```

```
    else:
        colors = np.asarray(sampled_points.colors) * 255  # Convert to 255 scale

    # Add randomness to each color within ±10
    random_colors = colors + np.random.randint(-10, 11, colors.shape)
    random_colors = np.clip(random_colors, 0, 255)  # Ensure values are within RGB range
    sampled_points.colors = o3d.utility.Vector3dVector(random_colors / 255.0)  # Convert back to 0-1 scale

    return sampled_points


def add_planes(annotation_path, room_points, point_density, noise_level, num_stories, story_height):
    pcd = o3d.geometry.PointCloud()
    pcd.points = o3d.utility.Vector3dVector(room_points)
    aabb = pcd.get_axis_aligned_bounding_box()
    min_bound = aabb.min_bound
    max_bound = aabb.max_bound

    floors = []

    for story in range(num_stories):
        z_offset = story * story_height

        floor_vertices = [
            [min_bound[0], min_bound[1], min_bound[2] + z_offset],
            [max_bound[0], min_bound[1], min_bound[2] + z_offset],
            [max_bound[0], max_bound[1], min_bound[2] + z_offset],
            [min_bound[0], max_bound[1], min_bound[2] + z_offset]
        ]
        floor_color = [204, 204, 204]  # Light gray
```

```
    floors.append(create_plane(floor_vertices, np.array(floor_color) / 255))

ceiling_vertices = [
    [min_bound[0], min_bound[1], max_bound[2]],
    [max_bound[0], min_bound[1], max_bound[2]],
    [max_bound[0], max_bound[1], max_bound[2]],
    [min_bound[0], max_bound[1], max_bound[2]]
]

wall1_vertices = [
    [min_bound[0], min_bound[1], min_bound[2]],
    [min_bound[0], max_bound[1], min_bound[2]],
    [min_bound[0], max_bound[1], max_bound[2]],
    [min_bound[0], min_bound[1], max_bound[2]]
]

wall2_vertices = [
    [max_bound[0], min_bound[1], min_bound[2]],
    [max_bound[0], max_bound[1], min_bound[2]],
    [max_bound[0], max_bound[1], max_bound[2]],
    [max_bound[0], min_bound[1], max_bound[2]]
]

wall3_vertices = [
    [min_bound[0], min_bound[1], min_bound[2]],
    [max_bound[0], min_bound[1], min_bound[2]],
    [max_bound[0], min_bound[1], max_bound[2]],
    [min_bound[0], min_bound[1], max_bound[2]]
]
```

```
wall4_vertices = [
    [min_bound[0], max_bound[1], min_bound[2]],
    [max_bound[0], max_bound[1], min_bound[2]],
    [max_bound[0], max_bound[1], max_bound[2]],
    [min_bound[0], max_bound[1], max_bound[2]]
]

# Different colors for floor, ceiling, and walls
ceiling_color = [230, 230, 230]  # Very light gray
wall1_color = [240, 235, 214]  # Eggshell white
wall2_color = [217, 217, 217]  # Light gray
wall3_color = [230, 230, 230]  # Very light gray
wall4_color = [204, 204, 204]  # Light gray

# Add noise to wall colors based on noise level
def add_noise_to_color(color, noise_level):
    color = np.array(color)
    noisy_color = color + np.random.randint(-noise_level, noise_level + 1, color.shape)
    return np.clip(noisy_color, 0, 255) / 255.0  # Normalize back to 0-1 range

ceiling = create_plane(ceiling_vertices, np.array(ceiling_color) / 255)
wall1 = create_plane(wall1_vertices, add_noise_to_color(wall1_color, noise_level))
wall2 = create_plane(wall2_vertices, add_noise_to_color(wall2_color, noise_level))
wall3 = create_plane(wall3_vertices, add_noise_to_color(wall3_color, noise_level))
wall4 = create_plane(wall4_vertices, add_noise_to_color(wall4_color, noise_level))

floor_pcs = [mesh_to_point_cloud(floor, point_density) for floor in floors]
ceiling_pc = mesh_to_point_cloud(ceiling, point_density)
wall1_pc = mesh_to_point_cloud(wall1, point_density)
wall2_pc = mesh_to_point_cloud(wall2, point_density)
```

```
    wall3_pc = mesh_to_point_cloud(wall3, point_density)
    wall4_pc = mesh_to_point_cloud(wall4, point_density)

    return floor_pcs, ceiling_pc, wall1_pc, wall2_pc, wall3_pc, wall4_pc


def save_scene(scene, base_path, area_index, room_index, point_density, noise_level, num_stories, story_height,
               rotation_angle, axis):
    room_path = os.path.join(base_path, f'Area_{area_index + 1}', f'room_{room_index + 1}')
    annotation_path = os.path.join(room_path, 'Annotations')

    os.makedirs(annotation_path, exist_ok=True)

    room_points = []
    room_colors = []
    class_counter = defaultdict(int)

    # Define rotation matrix for given axis and angle
    rotation_matrix = None
    angle_rad = np.radians(rotation_angle)
    if axis == 'x':
        rotation_matrix = np.array([
            [1, 0, 0],
            [0, np.cos(angle_rad), -np.sin(angle_rad)],
            [0, np.sin(angle_rad), np.cos(angle_rad)]
        ])
    elif axis == 'y':
        rotation_matrix = np.array([
            [np.cos(angle_rad), 0, np.sin(angle_rad)],
            [0, 1, 0],
```

```
        [-np.sin(angle_rad), 0, np.cos(angle_rad)]
    ])
elif axis == 'z':
    rotation_matrix = np.array([
        [np.cos(angle_rad), -np.sin(angle_rad), 0],
        [np.sin(angle_rad), np.cos(angle_rad), 0],
        [0, 0, 1]
    ])

for i, (obj, position) in enumerate(scene):
    class_counter[obj.class_name] += 1
    class_occurrence = class_counter[obj.class_name]

    translated_points = obj.points[:, :3] + position
    rotated_points = np.dot(translated_points, rotation_matrix.T)
    if obj.points.shape[1] == 6:  # If RGB information is present
        translated_points_with_color = np.hstack((rotated_points, obj.points[:, 3:]))
    else:
        translated_points_with_color = np.hstack((rotated_points, np.zeros((rotated_points.shape[0], 3))))
    room_points.append(translated_points_with_color[:, :3])
    room_colors.append(translated_points_with_color[:, 3:])  # Ensure colors are in 255 scale

    object_file_path = os.path.join(annotation_path, f'{obj.class_name}_{class_occurrence}.txt')
    np.savetxt(object_file_path, translated_points_with_color, fmt='%f')

room_points = np.concatenate(room_points, axis=0)
room_colors = np.concatenate(room_colors, axis=0)

# Add planes for multiple stories
floor_pcs, ceiling_pc, wall1_pc, wall2_pc, wall3_pc, wall4_pc = add_planes(
```

```
    annotation_path, room_points, point_density, noise_level, num_stories, story_height)

# Save individual plane point clouds for visualization
for idx, floor_pc in enumerate(floor_pcs):
    floor_points_with_color = np.hstack((np.asarray(floor_pc.points), np.asarray(floor_pc.colors) * 255))
    floor_file_path = os.path.join(annotation_path, f'floor_{idx + 1}.txt')
    np.savetxt(floor_file_path, floor_points_with_color, fmt='%f')

# ceiling_points_with_color = np.hstack((np.asarray(ceiling_pc.points), np.asarray(ceiling_pc.colors) * 255))
# ceiling_file_path = os.path.join(annotation_path, f'ceiling_1.txt')
# np.savetxt(ceiling_file_path, ceiling_points_with_color, fmt='%f')

wall1_points_with_color = np.hstack((np.asarray(wall1_pc.points), np.asarray(wall1_pc.colors) * 255))
wall1_file_path = os.path.join(annotation_path, f'wall_1.txt')
np.savetxt(wall1_file_path, wall1_points_with_color, fmt='%f')

wall2_points_with_color = np.hstack((np.asarray(wall2_pc.points), np.asarray(wall2_pc.colors) * 255))
wall2_file_path = os.path.join(annotation_path, f'wall_2.txt')
np.savetxt(wall2_file_path, wall2_points_with_color, fmt='%f')

wall3_points_with_color = np.hstack((np.asarray(wall3_pc.points), np.asarray(wall3_pc.colors) * 255))
wall3_file_path = os.path.join(annotation_path, f'wall_3.txt')
np.savetxt(wall3_file_path, wall3_points_with_color, fmt='%f')

wall4_points_with_color = np.hstack((np.asarray(wall4_pc.points), np.asarray(wall4_pc.colors) * 255))
wall4_file_path = os.path.join(annotation_path, f'wall_4.txt')
np.savetxt(wall4_file_path, wall4_points_with_color, fmt='%f')

all_planes_points = np.vstack(
    [np.asarray(pc.points) for pc in floor_pcs + [ceiling_pc, wall1_pc, wall2_pc, wall3_pc, wall4_pc]])
```

```
    all_planes_colors = np.vstack(
        [np.asarray(pc.colors) * 255 for pc in floor_pcs + [ceiling_pc, wall1_pc, wall2_pc, wall3_pc, wall4_pc]])

    room_points = np.concatenate([room_points, all_planes_points], axis=0)
    room_colors = np.concatenate([room_colors, all_planes_colors], axis=0)

    room_points_with_colors = np.hstack((room_points, room_colors))

    combined_room_file_path = os.path.join(room_path, f'room_{room_index + 1}.txt')
    np.savetxt(combined_room_file_path, room_points_with_colors, fmt='%f')


def save_scenes(scenes, base_path, point_density, max_objects_per_scene, noise_level):
    for area_index, area in tqdm(enumerate(scenes), desc="Saving Scenes"):
        alignment_angles = []
        for room_index, room_objects in enumerate(area):
            num_stories = random.randint(2, 3)
            story_height = 3
            scene = place_objects_in_scene(room_objects, max_objects=max_objects_per_scene, num_stories=num_stories,
                                           story_height=story_height)
            save_scene(scene, base_path, area_index, room_index, point_density, noise_level, num_stories, story_height,
                       90, 'x')

            alignment_angle = random.choice([0,90,180,270])
            alignment_angles.append((room_index + 1, alignment_angle))
        save_alignment_angles(base_path, area_index, alignment_angles)


def save_alignment_angles(base_path, area_index, alignment_angles):
    alignment_file_path = os.path.join(base_path, f'Area_{area_index + 1}', f'Area_{area_index + 1}_alignmentAngle.txt')
```

```
    with open(alignment_file_path, 'w') as f:
        f.write(f"## Global alignment angle per disjoint space in Area_{area_index + 1} ##\n")
        f.write("## Disjoint Space Name Global Alignment Angle ##\n")
        for room_index, alignment_angle in alignment_angles:
            f.write(f"room_{room_index} {alignment_angle}\n")



def visualize_scene_from_annotations(base_path, area_index, room_index):
    annotation_path = os.path.join(base_path, f'Area_{area_index + 1}', f'room_{room_index + 1}', 'Annotations')
    room_files = [f for f in os.listdir(annotation_path) if f.endswith('.txt') and not f.startswith('room')]

    point_clouds = []
    for room_file in room_files:
        file_path = os.path.join(annotation_path, room_file)
        points = np.loadtxt(file_path)
        pcd = o3d.geometry.PointCloud()
        pcd.points = o3d.utility.Vector3dVector(points[:, :3])
        if points.shape[1] == 6:
            pcd.colors = o3d.utility.Vector3dVector(points[:, 3:] / 255.0)
        else:
            pcd.colors = o3d.utility.Vector3dVector(np.zeros((points.shape[0], 3)))
        point_clouds.append(pcd)

    mesh = o3d.geometry.TriangleMesh.create_coordinate_frame()
    point_clouds.append(mesh)

    vis = o3d.visualization.Visualizer()
    vis.create_window()
    vis.get_render_option().background_color = np.array([0.5, 0.5, 0.5])
    for pcd in point_clouds:
```

```
        vis.add_geometry(pcd)
    vis.run()
    vis.destroy_window()



point_density = 5000
dataset_names = [f'subsample_{point_density}_0cm', f'subsample_{point_density}_2cm', f'subsample_{point_density}_5cm']
num_areas = 23
rooms_per_area = 45
max_objects_per_scene = 30



def delete_output_folder(base_path):
    if os.path.exists(base_path):
        shutil.rmtree(base_path)




def replace_folder(target_path, to_replace):
    # Check if the target folder exists
    if os.path.exists(target_path):
        # Remove the target folder and all its contents
        shutil.rmtree(target_path)
        print(f"Deleted folder: {target_path}")
    else:
        print(f"Target folder does not exist: {target_path}")

    # Copy the replacement folder to the target location
    shutil.copytree(to_replace, target_path)
    print(f"Replaced with folder from: {to_replace}")
```

```
delete_output_folder(f'/home/daniel/PycharmProjects/DatasetGenerator/S3DIS_Scenes2/')



for name in dataset_names:

    folder_path = f'/home/daniel/PycharmProjects/DatasetGenerator/ObjectsTXT/{name}'
    base_path = f'/home/daniel/PycharmProjects/DatasetGenerator/S3DIS_Scenes2/{name}'

    class_counts = get_class_counts(folder_path)
    point_clouds = load_point_clouds_from_folder(folder_path, point_density, class_counts)

    if '0cm' in name:
        noise_level = 0
    elif '2cm' in name:
        noise_level = 2
    elif '5cm' in name:
        noise_level = 5

    scenes = distribute_objects_across_scenes(point_clouds, num_areas, rooms_per_area, max_objects_per_scene)
    save_scenes(scenes, base_path, point_density, max_objects_per_scene, noise_level)
    #if scenes:
    #    visualize_scene_from_annotations(base_path, 0, 0)
#area = '/home/daniel/PycharmProjects/DatasetGenerator/Area_102'
#replaces = ['/home/daniel/PycharmProjects/DatasetGenerator/S3DIS_Scenes2/subsample_5000_0cm/Area_10',
#            '/home/daniel/PycharmProjects/DatasetGenerator/S3DIS_Scenes2/subsample_5000_2cm/Area_10',
#            '/home/daniel/PycharmProjects/DatasetGenerator/S3DIS_Scenes2/subsample_5000_5cm/Area_10']
#for target_path in replaces:
#    replace_folder(target_path, area)
```

Appendix F

Statement of Contributions

The research in this chapter was completed under the supervision of Dr. Carl Haas and Dr. Sriram Narasimhan at the University of Waterloo. I am the primary contributor of the realistic procedural generator Python code implementation.

Code for realistic procedural dataset generation methodology:

```
import open3d as o3d  # For 3D visualization and mesh manipulation
import numpy as np  # For numerical operations
import json  # For handling JSON files
import networkx as nx  # For creating and manipulating graphs
import matplotlib.pyplot as plt  # For plotting graphs
import random  # For random number generation
from random import randrange  # For generating random numbers within a range
import os  # For file operations
from tqdm import tqdm
from Lidarlike import copy_directory_with_files, process_rooms_in_areas
from Scaler import scale_dataset
from BasicRenamer import rename_files_and_folders


basic_pcds = {}


def sample_point_cloud_poisson(mesh, base_density=5000, max_points=10000):
    """Sample a point cloud from the mesh using Poisson disk sampling."""
    surface_area = mesh.get_surface_area()
    desired_num_points = int(base_density * surface_area)
    desired_num_points = min(desired_num_points, max_points)
```

```
    # print(f"Desired Number of Points (Poisson Sampling): {desired_num_points}")
    pcd = mesh.sample_points_poisson_disk(number_of_points=desired_num_points)
    # print(f"Sampled {len(pcd.points)} points using Poisson disk sampling.")
    return pcd



def create_alignment_files(base_directory):
    for root, dirs, files in os.walk(base_directory):
        area_folder = os.path.basename(root)

        # Check if there are any folders named 'room' in the current directory
        rooms = [d for d in dirs if d.startswith('room')]
        if rooms:
            area_index = int(area_folder.split('_')[-1]) - 1  # Extract the area index from the folder name

            # Alignment angle file path
            alignment_file_path = os.path.join(root, f'Area_{area_index + 1}_alignmentAngle.txt')

            # Create alignment angles and write them to the file
            with open(alignment_file_path, 'w') as f:
                f.write(f"## Global alignment angle per disjoint space in Area_{area_index + 1} ##\n")
                f.write("## Disjoint Space Name Global Alignment Angle ##\n")

                # Loop through the rooms and assign a random alignment angle
                for room_index, room in enumerate(range(number_of_rooms)):
                    alignment_angle = random.choice([0, 90, 180, 270])
                    f.write(f"room_{room + 1} {alignment_angle}\n")

            print(f"Created alignment file: {alignment_file_path}")
```

```
def merge_txt_files_in_annotations(base_directory):
    # Walk through all directories and subdirectories
    for root, dirs, files in os.walk(base_directory):
        # Check if current directory has an Annotations folder
        if 'Annotations' in dirs:
            annotations_path = os.path.join(root, 'Annotations')
            parent_folder_name = os.path.basename(root)
            output_file = os.path.join(root, f'{parent_folder_name}.txt')  # Save outside Annotations

            # Collect and merge all .txt files from Annotations folder
            with open(output_file, 'w') as outfile:
                for file_name in os.listdir(annotations_path):
                    if file_name.endswith('.txt'):
                        file_path = os.path.join(annotations_path, file_name)
                        with open(file_path, 'r') as infile:
                            outfile.write(infile.read())
                            outfile.write('\n')  # Add a newline after each file's content
            print(f'Merged files from {annotations_path} into {output_file}')



class SceneGraph:
    def __init__(self):
        self.graph = nx.DiGraph()  # Initialize a directed graph using NetworkX

    def parse_constraints(self, file_path):
        with open(file_path, 'r') as f:
            data = json.load(f)  # Load the JSON data from the file
```

```
        for node, attributes in data.items():
            self.graph.add_node(node, **attributes)
            for link in attributes.get("SpawningLink", []):
                self.graph.add_edge(link, node)



class ObjectPlacer:
    def __init__(self, scene_graph, room_size, visualizer, noise, density):
        self.scene_graph = scene_graph
        self.room_size = room_size
        self.visualizer = visualizer
        self.helper = PlacementHelper(room_size, scene_graph)
        self.placed_objects = {}
        self.color_map = {}
        self.point_noise = noise / 100
        self.point_density = density

    def generate_unique_colors(self):
        """
        Generate a unique color map for the scene.
        """
        object_types = list(self.scene_graph.graph.nodes)
        self.color_map = {obj: [random.random(), random.random(), random.random()] for obj in object_types}

    def ensure_object_list(self, object_name):
        if object_name not in self.placed_objects:
            self.placed_objects[object_name] = []

    def place_floor(self):
        object_name = "Floor"
```

```
    color = self.color_map[object_name]
    object_attributes = self.scene_graph.graph.nodes[object_name]
    object_size = (tuple(object_attributes["Size"])[0], tuple(object_attributes["Size"])[1], 0)
    position = (0, 0, 0)
    self.ensure_object_list(object_name)

    # Create a flat box (rectangle) as the floor
    mesh_box = o3d.geometry.TriangleMesh.create_box(width=object_size[0], height=object_size[1], depth=0.01)
    mesh_box.translate(position)  # Move the mesh to the correct position

    # Sample points from the mesh box using Poisson disk sampling
    if 'Floor' not in basic_pcds.keys():
        pcd = sample_point_cloud_poisson(mesh_box, 10000, self.point_density+10000)
        dark_gray_color = np.full((np.asarray(pcd.points).shape[0], 3), fill_value=0.2)
        pcd.colors = o3d.utility.Vector3dVector(dark_gray_color)
    else:
        pcd = basic_pcds['Floor']
    # Add the sampled point cloud to the visual scene
    self.visualizer.visual_scene.append(pcd)
    self.visualizer.save_object(pcd, object_name)
    self.placed_objects[object_name].append((position, object_size, pcd))

def place_ceiling(self):
    object_name = "Floor"
    color = self.color_map[object_name]
    object_attributes = self.scene_graph.graph.nodes[object_name]
    ceiling_size = (tuple(object_attributes["Size"])[0], tuple(object_attributes["Size"])[1], 0)
    wall_size = tuple(self.scene_graph.graph.nodes["Wall"]["Size"])
    position = (0, 0, wall_size[2])
    self.ensure_object_list(object_name)
```

```
    # Create a flat box (rectangle) as the ceiling
    mesh_box = o3d.geometry.TriangleMesh.create_box(width=ceiling_size[0], height=ceiling_size[1], depth=0.01)
    mesh_box.translate(position)  # Move the mesh to the correct position
    if 'ceiling' not in basic_pcds.keys():
        pcd = sample_point_cloud_poisson(mesh_box, 10000, self.point_density+10000)
        dark_gray_color = np.full((np.asarray(pcd.points).shape[0], 3), fill_value=0.2)
        pcd.colors = o3d.utility.Vector3dVector(dark_gray_color)
        basic_pcds['ceiling'] = pcd
    else:
        pcd = basic_pcds['ceiling']

    # Add the sampled point cloud to the visual scene
    self.visualizer.visual_scene.append(pcd)
    self.visualizer.save_object(pcd, object_name)
    self.placed_objects[object_name].append((position, ceiling_size, pcd))

def place_walls(self):
    object_name = "Wall"
    wall_attributes = self.scene_graph.graph.nodes[object_name]
    floor_attributes = self.scene_graph.graph.nodes["Floor"]

    wall_thickness = wall_attributes["Size"][0]
    wall_height = wall_attributes["Size"][2]  # This should match the z-position of the ceiling
    floor_length, floor_width, _ = tuple(floor_attributes["Size"])

    self.ensure_object_list(object_name)

    # Define the four walls with correct positions and sizes
    wall_definitions = [
```

```
    # Left wall
    ((0, 0, 0), (wall_thickness, floor_width, wall_height)),
    # Right wall
    ((floor_length - wall_thickness, 0, 0), (wall_thickness, floor_width, wall_height)),
    # Front wall
    ((0, 0, 0), (floor_length, wall_thickness, wall_height)),
    # Back wall
    ((0, floor_width - wall_thickness, 0), (floor_length, wall_thickness, wall_height))
]

for i, (position, size) in enumerate(wall_definitions):
    # Create a wall mesh
    mesh_wall = o3d.geometry.TriangleMesh.create_box(width=size[0], height=size[1], depth=size[2])
    mesh_wall.translate(position)

    # Sample points from the wall mesh using Poisson disk sampling
    pcd = sample_point_cloud_poisson(mesh_wall, base_density=self.point_density, max_points=10000)

    # Color the point cloud (white in this case)
    white_color = np.ones((np.asarray(pcd.points).shape[0], 3))
    pcd.colors = o3d.utility.Vector3dVector(white_color)

    # Add the sampled point cloud to the visual scene
    self.visualizer.visual_scene.append(pcd)

    # Save the wall with a distinct filename

    self.visualizer.save_object(pcd, object_name)

    self.placed_objects[object_name].append((position, size, pcd))
```

```
def place_walls2(self):
    object_name = "Wall"
    object_attributes = self.scene_graph.graph.nodes[object_name]
    wall_size = tuple(object_attributes["Size"])  # (thickness, height, length)
    floor_size = tuple(self.scene_graph.graph.nodes["Floor"]["Size"])  # (length, width)
    self.ensure_object_list(object_name)

    # Define the size and position for a single encompassing wall box
    # Assuming walls surround the floor, we'll create a box that covers all walls
    # The box will have thickness equal to wall_size[0], height wall_size[1], and length as needed

    # Calculate overall dimensions
    overall_length = floor_size[0]
    overall_width = floor_size[0]
    thickness = wall_size[0]
    height = wall_size[2]

    # Create a single box that encompasses all four walls
    # The box dimensions will cover the perimeter of the floor
    # We'll set the depth to the wall thickness

    mesh_box = o3d.geometry.TriangleMesh.create_box(wall_size[1], floor_size[1], wall_size[2])
    mesh_box.compute_vertex_normals()

    # Position the box so that the inner area aligns with the floor
    # Translate the box by (-thickness, 0, -thickness)
    mesh_box.translate((-thickness, 0.0, -thickness))

    # Remove the top and bottom faces
```

```
# To do this, we'll identify and exclude triangles with normals in +Z and -Z directions

# Get the triangles and their normals
triangles = np.asarray(mesh_box.triangles)
triangle_normals = np.asarray(mesh_box.triangle_normals)

# Define a small epsilon for numerical precision
epsilon = 1e-6

# Create masks for top and bottom faces
top_face_mask = np.abs(triangle_normals[:, 2] - 1.0) < epsilon
bottom_face_mask = np.abs(triangle_normals[:, 2] + 1.0) < epsilon

# Combine masks to identify top and bottom triangles
top_bottom_mask = top_face_mask | bottom_face_mask

# Select triangles that are NOT top or bottom
side_triangles = triangles[~top_bottom_mask]

# Function to create a mesh from selected triangles
def create_side_mesh(original_mesh, selected_triangles):
    side_mesh = o3d.geometry.TriangleMesh()
    # Get unique vertex indices
    unique_indices = np.unique(selected_triangles)
    # Map old indices to new
    index_map = {old_idx: new_idx for new_idx, old_idx in enumerate(unique_indices)}
    # Create vertices
    vertices = np.asarray(original_mesh.vertices)[unique_indices]
    side_mesh.vertices = o3d.utility.Vector3dVector(vertices)
    # Remap triangle indices
```

```
        triangles_mapped = [[index_map[idx] for idx in tri] for tri in selected_triangles]
        side_mesh.triangles = o3d.utility.Vector3iVector(triangles_mapped)
        side_mesh.compute_vertex_normals()
        return side_mesh

    # Create the side mesh without top and bottom
    side_mesh = create_side_mesh(mesh_box, side_triangles)

    # Optionally, assign a color to the walls (e.g., white)
    wall_color = [1.0, 1.0, 1.0]  # White
    num_vertices = len(side_mesh.vertices)
    side_mesh.paint_uniform_color(wall_color)

    # Sample points from the side mesh using Poisson disk sampling
    # Assuming `sample_point_cloud_poisson` is a predefined method
    pcd = sample_point_cloud_poisson(side_mesh, base_density=self.point_density, max_points=10000)

    # If you wish to color the point cloud differently, you can modify the colors here
    # For example, keeping them white as above
    white_color = np.ones((np.asarray(pcd.points).shape[0], 3))
    pcd.colors = o3d.utility.Vector3dVector(white_color)

    # Add the sampled point cloud to the visual scene
    self.visualizer.visual_scene.append(pcd)
    self.visualizer.save_object(pcd, object_name)
    self.placed_objects[object_name].append(((-thickness, 0.0, -thickness),(wall_size[1], floor_size[1], wall_size[2]),pcd))

def place_walls1(self):
    object_name = "Wall"
    object_attributes = self.scene_graph.graph.nodes[object_name]
```

```
    wall_size = tuple(object_attributes["Size"])  # (thickness, height, length)
    floor_size = tuple(self.scene_graph.graph.nodes["Floor"]["Size"])  # (length, width)
    self.ensure_object_list(object_name)

    wall_positions = [
        ((0, 0, 0), (wall_size[1], floor_size[1], wall_size[2])),
        ((floor_size[0] - wall_size[1], 0, 0), (wall_size[1], floor_size[1], wall_size[2])),
        ((0, 0, 0), (floor_size[0], wall_size[1], wall_size[2])),
        ((0, floor_size[1] - wall_size[1], 0), (floor_size[0], wall_size[1], wall_size[2]))
    ]

    for pos, size in wall_positions:
        # Create a flat box (rectangle) for each wall with a small depth (thickness of the wall)
        mesh_box = o3d.geometry.TriangleMesh.create_box(width=size[0], height=size[1], depth=size[2])
        mesh_box.translate(pos)  # Move the mesh to the correct position

        # Sample points from the mesh box using Poisson disk sampling
        pcd = sample_point_cloud_poisson(mesh_box, 500, self.point_density)
        white_color = np.ones((np.asarray(pcd.points).shape[0], 3))
        pcd.colors = o3d.utility.Vector3dVector(white_color)

        # Add the sampled point cloud to the visual scene

        self.visualizer.visual_scene.append(pcd)
        self.visualizer.save_object(pcd, object_name)
        self.placed_objects[object_name].append((pos, size, pcd))

def place_objects(self, object_name):
    self.ensure_object_list(object_name)
    object_attributes = self.scene_graph.graph.nodes[object_name]
```

```
    size = tuple(object_attributes["Size"])
    instances = object_attributes.get('Instances', 1)
    color = self.color_map[object_name]
    spawning_links = object_attributes.get("SpawningLink", [])

    for _ in range(instances):
        potential_positions = [
            (x, y, 0) for x in range(self.room_size[0] - size[0] + 1)
            for y in range(self.room_size[1] - size[1] + 1)
        ]
        valid_positions = self.helper.check_grounding(potential_positions, object_name, size)

        if spawning_links:
            valid_positions = self.adjust_positions_near_spawning_links(valid_positions, spawning_links, size)

        valid_positions = [pos for pos in valid_positions if not self.helper.check_overlap(pos, size, 0)]

        if valid_positions:
            chosen_position = random.choice(valid_positions)
            wireframe = self.visualizer.create_wireframe_box(object_name, size, color, translate=chosen_position)
            self.placed_objects[object_name].append((chosen_position, size, wireframe))
            self.helper.mark_grid(chosen_position, size)
        else:
            break

def adjust_positions_near_spawning_links(self, potential_positions, spawning_links, size):
    adjusted_positions = []
    for link in spawning_links:
        if link.lower() in ['floor', 'walls', 'ceiling']:
            continue
```

```
        link_positions = [pos for pos, _, _ in self.placed_objects.get(link, [])]
        link_size = tuple(self.scene_graph.graph.nodes[link]["Size"])

        for (lx, ly, _) in link_positions:
            for y in range(ly, ly + link_size[1]):
                adjusted_positions.append((lx - size[0], y, 0))
                adjusted_positions.append((lx + link_size[0], y, 0))
            for x in range(lx, lx + link_size[0]):
                adjusted_positions.append((x, ly - size[1], 0))
                adjusted_positions.append((x, ly + link_size[1], 0))

        adjusted_positions = list(set(adjusted_positions))

    return adjusted_positions if adjusted_positions else potential_positions

def place_pipes_and_valve(self):
    object_name = "Pipe"
    self.ensure_object_list(object_name)
    support_positions = [pos for pos, _, _ in self.placed_objects.get("Support", [])]
    if not support_positions:
        return

    default_size = (1, 1, 1)
    object_attributes = self.scene_graph.graph.nodes.get(object_name, {})
    size = tuple(object_attributes.get("Size", default_size))

    path_points = self.generate_pipe_path(support_positions)
    radius_random = random.uniform(0.01, 0.7)
    if path_points is not None and not self.helper.check_pipe_overlap(path_points, radius=radius_random):
        tube, helix_wire = self.create_tube_from_path(path_points, radius=radius_random)
```

```
            wireframe = self.create_wireframe_for_tube(path_points, size)
            self.placed_objects[object_name].append((path_points, size, tube, wireframe, helix_wire))
            self.visualizer.add_tube_to_scene(tube)
            # self.visualizer.add_wireframe_to_scene(wireframe)
            # self.visualizer.add_tube_to_scene(helix_wire)
            self.helper.mark_grid_for_pipe(path_points, radius=radius_random)

            self.place_valve_on_pipe(path_points, radius_random)
        else:
            print("Failed to place pipe due to overlap.")

    def save_temporary_mesh(self, mesh, filename):
        o3d.io.write_triangle_mesh(filename, mesh)

    def save_temporary_line_set(self, line_set, filename):
        o3d.io.write_line_set(filename, line_set)

    def create_tube_from_path(self, path_points, radius, segments=20):
        vertices = []
        faces = []
        colors = []
        n = len(path_points)

        pipe_color = [random.uniform(0, 1) for _ in range(3)]
        while all(c > 0.9 for c in pipe_color):
            pipe_color = [random.uniform(0, 1) for _ in range(3)]

        for i in range(n):
            point = path_points[i]
            if i == 0:
```

```
    next_point = path_points[i + 1]
    direction = next_point - point
elif i == n - 1:
    prev_point = path_points[i - 1]
    direction = point - prev_point
else:
    prev_point = path_points[i - 1]
    next_point = path_points[i + 1]
    direction = next_point - prev_point

norm_direction = np.linalg.norm(direction)
if norm_direction > 0:
    direction /= norm_direction
else:
    direction = np.array([1, 0, 0], dtype=float)

if np.abs(direction[0]) > np.abs(direction[2]):
    ortho = np.array([-direction[1], direction[0], 0], dtype=float)
else:
    ortho = np.array([0, -direction[2], direction[1]], dtype=float)

ortho /= np.linalg.norm(ortho)
bitangent = np.cross(direction, ortho)

circle = []
for j in range(segments):
    angle = 2 * np.pi * j / segments
    circle_point = (radius * np.cos(angle) * ortho +
                    radius * np.sin(angle) * bitangent)
    circle.append(point + circle_point)
```

```
            colors.append(pipe_color)

        vertices.extend(circle)

        if i > 0:
            start_idx = (i - 1) * segments
            for j in range(segments):
                next_j = (j + 1) % segments
                faces.append([start_idx + j, start_idx + next_j, start_idx + j + segments])
                faces.append([start_idx + next_j, start_idx + next_j + segments, start_idx + j + segments])

    vertices = np.array(vertices)
    faces = np.array(faces)
    colors = np.array(colors)

    tube_o3d = o3d.geometry.TriangleMesh(vertices=o3d.utility.Vector3dVector(vertices),
                                         triangles=o3d.utility.Vector3iVector(faces))
    tube_o3d.vertex_colors = o3d.utility.Vector3dVector(colors)

    helix_wire = self.create_helix_wire(path_points, radius, 0.05, turns=20)

    return tube_o3d, helix_wire

def create_helix_wire(self, path_points, pipe_radius, wire_radius, turns=20, segments_per_turn=20):
    helix_vertices = []
    wire_radius_total = pipe_radius + wire_radius

    wire_color = [random.uniform(0, 1) for _ in range(3)]
    while all(c > 0.9 for c in wire_color):
        wire_color = [random.uniform(0, 1) for _ in range(3)]
```

```
for i in range(len(path_points) - 1):
    start_point = path_points[i]
    end_point = path_points[i + 1]
    segment_length = np.linalg.norm(end_point - start_point)
    num_segments = int(segment_length * turns)

    direction = end_point - start_point

    direction = direction / (np.linalg.norm(direction) + 1e-6)

    if np.abs(direction[0]) > np.abs(direction[2]):
        ortho = np.array([-direction[1], direction[0], 0], dtype=float)
    else:
        ortho = np.array([0, -direction[2], direction[1]], dtype=float)

    ortho /= (np.linalg.norm(ortho) + 1e-6)
    bitangent = np.cross(direction, ortho)

    for j in range(num_segments):
        t = j / segments_per_turn * 2 * np.pi
        x_offset = wire_radius_total * np.cos(t)
        y_offset = wire_radius_total * np.sin(t)
        offset = x_offset * ortho + y_offset * bitangent
        point = start_point + j / num_segments * (end_point - start_point) + offset
        helix_vertices.append(point)

t = 2 * np.pi * turns
x_offset = wire_radius_total * np.cos(t)
y_offset = wire_radius_total * np.sin(t)
```

```
    offset = x_offset * ortho + y_offset * bitangent
    helix_vertices.append(end_point + offset)

    helix_vertices = np.array(helix_vertices)
    lines = [[i, i + 1] for i in range(len(helix_vertices) - 1)]
    colors = [wire_color for _ in range(len(lines))]

    helix_wire = o3d.geometry.LineSet()
    helix_wire.points = o3d.utility.Vector3dVector(helix_vertices)
    helix_wire.lines = o3d.utility.Vector2iVector(lines)
    helix_wire.colors = o3d.utility.Vector3dVector(colors)

    return helix_wire

def generate_pipe_path(self, support_positions, num_points=100, length=10):
    path_points = []
    ceiling_height = self.room_size[2] - 1

    start_wall = np.random.choice(['x', 'y'])
    if start_wall == 'x':
        start_point = np.array([0, np.random.uniform(0.5, self.room_size[1] - 0.5), ceiling_height])
        initial_turn = np.array([0.5, start_point[1], ceiling_height])
    else:
        start_point = np.array([np.random.uniform(0.5, self.room_size[0] - 0.5), 0, ceiling_height])
        initial_turn = np.array([start_point[0], 0.5, ceiling_height])

    path_points.append(start_point)
    path_points.append(initial_turn)

    current_point = initial_turn
```

```
for support_pos in support_positions:
    next_point = np.array([support_pos[0], support_pos[1], ceiling_height])
    if current_point[0] != next_point[0]:
        intermediate_point = np.array([next_point[0], current_point[1], ceiling_height])
        path_points.append(intermediate_point)
    path_points.append(next_point)
    current_point = next_point

end_wall_choice = np.random.choice(['perpendicular', 'parallel'])
if end_wall_choice == 'perpendicular':
    end_wall = 'y' if start_wall == 'x' else 'x'
else:
    end_wall = start_wall

if end_wall == 'x':
    end_point = np.array([self.room_size[0], np.random.uniform(0.5, self.room_size[1] - 0.5), ceiling_height])
    final_turn = np.array([self.room_size[0] - 0.5, end_point[1], ceiling_height])
else:
    end_point = np.array([np.random.uniform(0.5, self.room_size[0] - 0.5), self.room_size[1], ceiling_height])
    final_turn = np.array([end_point[0], self.room_size[1] - 0.5, ceiling_height])

if current_point[0] != final_turn[0]:
    intermediate_point = np.array([final_turn[0], current_point[1], ceiling_height])
    path_points.append(intermediate_point)

path_points.append(final_turn)
path_points.append(end_point)

path_points = np.array(path_points)
```

```
    fine_points = []
    for i in range(len(path_points) - 1):
        segment_points = np.linspace(path_points[i], path_points[i + 1], num_points // (len(path_points) - 1))
        fine_points.append(segment_points)

    spline_points = np.vstack(fine_points)
    return spline_points

def create_wireframe_for_tube(self, path_points, size):
    lines = []
    points = []
    color = [0.0, 1.0, 0.0]

    radius = 0.5
    for i in range(len(path_points) - 1):
        p1 = path_points[i]
        p2 = path_points[i + 1]
        box1 = [
            [p1[0] - radius, p1[1] - radius, p1[2] - radius],
            [p1[0] + radius, p1[1] - radius, p1[2] - radius],
            [p1[0] + radius, p1[1] + radius, p1[2] - radius],
            [p1[0] - radius, p1[1] + radius, p1[2] - radius],
            [p1[0] - radius, p1[1] - radius, p1[2] + radius],
            [p1[0] + radius, p1[1] - radius, p1[2] + radius],
            [p1[0] + radius, p1[1] + radius, p1[2] + radius],
            [p1[0] - radius, p1[1] + radius, p1[2] + radius],
        ]
        box2 = [
            [p2[0] - radius, p2[1] - radius, p2[2] - radius],
            [p2[0] + radius, p2[1] - radius, p2[2] - radius],
```

```
            [p2[0] + radius, p2[1] + radius, p2[2] - radius],
            [p2[0] - radius, p2[1] + radius, p2[2] - radius],
            [p2[0] - radius, p2[1] - radius, p2[2] + radius],
            [p2[0] + radius, p2[1] - radius, p2[2] + radius],
            [p2[0] + radius, p2[1] + radius, p2[2] + radius],
            [p2[0] - radius, p2[1] + radius, p2[2] + radius],
        ]
        points.extend(box1 + box2)
        lines.extend([
            [i * 16, i * 16 + 1], [i * 16 + 1, i * 16 + 2], [i * 16 + 2, i * 16 + 3], [i * 16 + 3, i * 16],
            [i * 16 + 4, i * 16 + 5], [i * 16 + 5, i * 16 + 6], [i * 16 + 6, i * 16 + 7], [i * 16 + 7, i * 16 + 4],
            [i * 16, i * 16 + 4], [i * 16 + 1, i * 16 + 5], [i * 16 + 2, i * 16 + 6], [i * 16 + 3, i * 16 + 7],
            [i * 16 + 8, i * 16 + 9], [i * 16 + 9, i * 16 + 10], [i * 16 + 10, i * 16 + 11],
            [i * 16 + 11, i * 16 + 8],
            [i * 16 + 12, i * 16 + 13], [i * 16 + 13, i * 16 + 14], [i * 16 + 14, i * 16 + 15],
            [i * 16 + 15, i * 16 + 12],
            [i * 16 + 8, i * 16 + 12], [i * 16 + 9, i * 16 + 13], [i * 16 + 10, i * 16 + 14],
            [i * 16 + 11, i * 16 + 15],
        ])

    points = np.array(points)
    colors = [color for _ in range(len(lines))]

    wireframe = o3d.geometry.LineSet()
    wireframe.points = o3d.utility.Vector3dVector(points)
    wireframe.lines = o3d.utility.Vector2iVector(lines)
    wireframe.colors = o3d.utility.Vector3dVector(colors)

    return wireframe
```

```
def place_valve_on_pipe(self, path_points, pipe_radius):
    valve_name = "Valve"
    self.ensure_object_list(valve_name)
    valve_attributes = self.scene_graph.graph.nodes[valve_name]
    valve_size = tuple(valve_attributes["Size"])
    color = self.color_map[valve_name]

    pipe_length = len(path_points)
    valve_position_index = random.randint(0, pipe_length - 1)
    valve_position = path_points[valve_position_index].copy()

    valve_position[0] = valve_position[0] - (pipe_radius / 2)
    valve_position[1] = valve_position[1] - (pipe_radius / 2)
    valve_position[2] -= (pipe_radius * 3)

    valve_position = tuple(map(int, valve_position))

    wireframe = self.visualizer.create_wireframe_box(valve_name, valve_size, color, translate=valve_position)
    self.placed_objects[valve_name].append((valve_position, valve_size, wireframe))
    self.helper.mark_grid(valve_position, valve_size)

def bfs_traverse_and_place(self):
    queue = ["Floor", "Wall", "Ceiling"]
    visited = set(queue)

    self.place_floor()
    self.place_walls()
    self.place_ceiling()

    while queue:
```

```
            current_object = queue.pop(0)
            for neighbor in self.scene_graph.graph.successors(current_object):
                if neighbor not in visited:
                    if neighbor not in ["Valve", "Wire"]:
                        self.place_objects(neighbor)
                    queue.append(neighbor)
                    visited.add(neighbor)

        self.place_pipes_and_valve()



class PlacementHelper:
    def __init__(self, room_size, scene_graph):
        self.grid = np.zeros(room_size, dtype=bool)
        self.room_size = room_size
        self.scene_graph = scene_graph

    def mark_grid(self, position, size):
        x, y, z = map(int, position)
        sx, sy, sz = size
        self.grid[x:x + sx, y:y + sy, z:z + sz] = True

    def mark_grid_for_pipe(self, path_points, radius):
        for point in path_points:
            x, y, z = map(int, point)
            sx, sy, sz = int(radius * 2), int(radius * 2), int(radius * 2)
            self.grid[max(0, x - sx // 2):min(self.room_size[0], x + sx // 2 + 1),
            max(0, y - sy // 2):min(self.room_size[1], y + sy // 2 + 1),
            max(0, z - sz // 2):min(self.room_size[2], z + sz // 2 + 1)] = True
```

```
def check_grounding(self, potential_points, object_name, size):
    object_attributes = self.scene_graph.graph.nodes[object_name]
    grounding = object_attributes.get('Conditions', {}).get('Grounding', 'Floor')
    distance = object_attributes.get('Conditions', {}).get('DistanceFromObjects', 0)
    wall_thickness = self.scene_graph.graph.nodes["Wall"]["Size"][0]

    grounded_points = []
    if grounding == 'Wall':
        if object_name == 'Door':
            y = 0
            for x in range(wall_thickness, self.room_size[0] - wall_thickness - size[0] + 1):
                if not self.check_overlap((x, y, 0), size, distance):
                    grounded_points.append((x, y, 0))

        elif object_name == 'ElectricBox':
            x = self.room_size[0] - size[0]
            z = (self.room_size[2] // 2) - (size[2] // 2)
            for y in range(wall_thickness, self.room_size[1] - wall_thickness - size[1] + 1):
                if not self.check_overlap((x, y, z), size, distance):
                    grounded_points.append((x, y, z))

        elif object_name == 'Valve':
            x = self.room_size[0] - size[0]
            z = (self.room_size[2] // 2) - (size[2] // 2)
            for y in range(wall_thickness, self.room_size[1] - wall_thickness - size[1] + 1):
                if not self.check_overlap((x, y, z), size, distance):
                    grounded_points.append((x, y, z))

        elif object_name == 'Ladder':
            x = 0
```

```
            for y in range(wall_thickness, self.room_size[1] - wall_thickness - size[1] + 1):
                if not self.check_overlap((x, y, 0), size, distance):
                    grounded_points.append((x, y, 0))

        elif object_name == 'Stairs':
            x_positions = [wall_thickness, self.room_size[0] - wall_thickness - size[0]]
            for x in x_positions:
                if not self.check_overlap((x, 0, 0), size, distance):
                    grounded_points.append((x, 0, 0))

    elif grounding == 'Floor' or grounding == 'Ceiling':
        z_ground = 0 if grounding == 'Floor' else self.room_size[2] - size[2]
        for x, y, _ in potential_points:
            if not self.check_overlap((x, y, z_ground), size, distance):
                grounded_points.append((x, y, z_ground))

    return grounded_points

def check_overlap(self, position, size, distance):
    x, y, z = position
    sx, sy, sz = size
    expanded_x = max(0, x - distance)
    expanded_y = max(0, y - distance)
    expanded_sx = min(self.room_size[0], x + sx + distance)
    expanded_sy = min(self.room_size[1], y + sy + distance)

    if np.any(self.grid[expanded_x:expanded_sx, expanded_y:expanded_sy]):
        return True
    return False
```

```
def check_pipe_overlap(self, path_points, radius):
    for point in path_points:
        x, y, z = map(int, point)
        sx, sy, sz = int(radius * 2), int(radius * 2), int(radius * 2)
        expanded_x = max(0, x - sx // 2)
        expanded_y = max(0, y - sy // 2)
        expanded_sx = min(self.room_size[0], x + sx // 2)
        expanded_sy = min(self.room_size[1], y + sy // 2)

        if np.any(self.grid[expanded_x:expanded_sx, expanded_y:expanded_sy]):
            return True
    return False

def save_object(self, object_name, position, color):
    """Save the object as a point cloud (.txt) in the specified folder structure."""
    dataset_path = os.path.join(self.object_placer.dataset_name, f"Area_{self.object_placer.area_count}",
                                f"room_{self.object_placer.room_count}", "Annotations")
    os.makedirs(dataset_path, exist_ok=True)

    # Track object count for naming convention
    if object_name not in self.object_placer.object_count:
        self.object_placer.object_count[object_name] = 0
    else:
        self.object_placer.object_count[object_name] += 1

    object_filename = f"{object_name.lower()}_{self.object_placer.object_count[object_name]}.txt"
    object_path = os.path.join(dataset_path, object_filename)

    # Simulating point cloud data for the object
    points = np.random.rand(100, 3) * 10  # Simulated (x, y, z)
```

```
        colors = np.random.rand(100, 3) * 255  # Simulated (r, g, b)
        point_cloud_data = np.hstack((points, colors))

        # Save the object as a point cloud .txt file
        np.savetxt(object_path, point_cloud_data, fmt='%.6f %.6f %.6f %.6f %.6f %.6f')



class Visualizer:
    def __init__(self, object_placer, dataset_name, area, room, point_noise, point_density):
        self.object_placer = object_placer
        self.visual_scene = []
        self.area = area
        self.room = room
        self.dataset_name = dataset_name
        self.object_counter = {}
        self.point_noise = point_noise / 100
        self.point_density = point_density

    def add_tube_to_scene(self, tube):
        pcd = sample_point_cloud_poisson(tube, self.point_density)
        self.save_object(pcd, 'pipe')
        self.visual_scene.append(pcd)

    def create_wireframe_box(self, object_name, size, color, translate=(0, 0, 0), grid_density=10):
        lines = [
            [0, 1], [1, 2], [2, 3], [3, 0],
            [4, 5], [5, 6], [6, 7], [7, 4],
            [0, 4], [1, 5], [2, 6], [3, 7]
        ]
        points = [
```

```
    [0, 0, 0], [size[0], 0, 0], [size[0], size[1], 0], [0, size[1], 0],
    [0, 0, size[2]], [size[0], 0, size[2]], [size[0], size[1], size[2]], [0, size[1], size[2]]
]
points = np.array(points) + np.array(translate)

if object_name == 'Floor' or object_name == 'Ceiling':
    points = points.tolist()
    for i in range(1, grid_density):
        fraction = i / grid_density
        start = np.array([0, fraction * size[1], size[2]]) + np.array(translate)
        end = np.array([size[0], fraction * size[1], size[2]]) + np.array(translate)
        points.extend([start.tolist(), end.tolist()])
        lines.append([len(points) - 2, len(points) - 1])

        start = np.array([fraction * size[0], 0, size[2]]) + np.array(translate)
        end = np.array([fraction * size[0], size[1], size[2]]) + np.array(translate)
        points.extend([start.tolist(), end.tolist()])
        lines.append([len(points) - 2, len(points) - 1])

    colors = [color for _ in range(len(lines))]
    line_set = o3d.geometry.LineSet()
    line_set.points = o3d.utility.Vector3dVector(points)
    line_set.lines = o3d.utility.Vector2iVector(lines)
    line_set.colors = o3d.utility.Vector3dVector(colors)
else:
    colors = [color for _ in range(len(lines))]

    line_set = o3d.geometry.LineSet()
    line_set.points = o3d.utility.Vector3dVector(points)
    line_set.lines = o3d.utility.Vector2iVector(lines)
```

```
        line_set.colors = o3d.utility.Vector3dVector(colors)

    # self.visual_scene.append(line_set)
    return line_set

def load_and_place_mesh(self, pcd_path, size, translate, orientation, object_name):
    pcd = o3d.io.read_point_cloud(pcd_path)
    pcd_bounds = pcd.get_axis_aligned_bounding_box()
    current_extent = np.array(pcd_bounds.get_extent())

    if object_name == "Railing":
        pcd_bounds_min = pcd.get_min_bound()
        pcd_bounds_max = pcd.get_max_bound()

        # Calculate the extent of the point cloud in x, y, z dimensions
        current_extent = np.array(pcd_bounds_max) - np.array(pcd_bounds_min)

        # Calculate scaling factors for each axis
        scale_x = size[0] / current_extent[0]
        scale_y = size[1] / current_extent[1]
        scale_z = size[2] / current_extent[2]
        scale_factor_xy = min(scale_x, scale_y)
        scale_factors = np.array([scale_factor_xy, scale_factor_xy, scale_z])
        scale_transform = np.diag(np.append(scale_factors, [1]))
        pcd.transform(scale_transform)

        rotation_matrix = o3d.geometry.get_rotation_matrix_from_xyz(np.radians(orientation))
        pcd.rotate(rotation_matrix, center=pcd.get_center())

        pcd_bounds = pcd.get_axis_aligned_bounding_box()
```

```
        pcd_center = np.array(pcd_bounds.get_center())
        desired_center = np.array(translate) + np.array(size) / 2
        translation_vector = desired_center - pcd_center

        pcd.translate(translation_vector)
        self.save_object(pcd, object_name)
        self.visual_scene.append(pcd)
        return pcd

    else:
        sorted_indices = np.argsort(-current_extent)
        sorted_size = np.array(size)[np.argsort(-np.array(size))]
        scale_factors = sorted_size / (current_extent[sorted_indices] + 1e-6)

        original_scale_factors = np.zeros_like(scale_factors)
        original_scale_factors[sorted_indices] = scale_factors
        scale_transform = np.diag(np.append(original_scale_factors, [1]))
        pcd.transform(scale_transform)

        rotation_matrix = o3d.geometry.get_rotation_matrix_from_xyz(np.radians(orientation))
        pcd.rotate(rotation_matrix, center=pcd.get_center())

        pcd_bounds = pcd.get_axis_aligned_bounding_box()
        pcd_center = np.array(pcd_bounds.get_center())
        pcd_extent = np.array(pcd_bounds.get_extent())

        wireframe_center = np.array(translate) + np.array(size) / 2
        final_translation = wireframe_center - pcd_center

        if final_translation.ndim > 1:
```

```
            final_translation = final_translation[0]

        pcd.translate(final_translation)
        self.save_object(pcd, object_name)
        self.visual_scene.append(pcd)
        return pcd

def load_meshes_for_objects(self, path, density):
    orientation_mapping = {
        'Desk': f'{path}/Orientations/desk.txt',
        'Column': f'{path}/Orientations/column.txt',
        'Chair': f'{path}/Orientations/chair.txt',
        'Stairs': f'{path}/Orientations/stairs.txt',
        'Door': f'{path}/Orientations/door.txt',
        'ElectricBox': f'{path}/Orientations/electricbox.txt',
        'Equipment': f'{path}/Orientations/equipment.txt',
        'Ladder': f'{path}/Orientations/ladder.txt',
        'Support': f'{path}/Orientations/support.txt',
        'Valve': f'{path}/Orientations/valve.txt',
        'Railing': f'{path}/Orientations/railing.txt',
    }

    orientation_indices = {}

    for key in orientation_mapping:
        with open(orientation_mapping[key], 'r') as file:
            lines = file.readlines()
        orientation_indices[key] = randrange(len(lines))

    mesh_mapping = {
```

```
    'Desk': f'{path}/PCD_Objects_{density}/desk/desk_{orientation_indices["Desk"] + 1}.pcd',
    'Column': f'{path}/PCD_Objects_{density}/column/column_{orientation_indices["Column"] + 1}.pcd',
    'Chair': f'{path}/PCD_Objects_{density}/chair/chair_{orientation_indices["Chair"] + 1}.pcd',
    'Stairs': f'{path}/PCD_Objects_{density}/stairs/stairs_{orientation_indices["Stairs"] + 1}.pcd',
    'Door': f'{path}/PCD_Objects_{density}/door/door_{orientation_indices["Door"] + 1}.pcd',
    'ElectricBox': f'{path}/PCD_Objects_{density}/electricbox/electricbox_{orientation_indices["ElectricBox"] + 1}.pcd',
    'Equipment': f'{path}/PCD_Objects_{density}/equipment/equipment_{orientation_indices["Equipment"] + 1}.pcd',
    'Ladder': f'{path}/PCD_Objects_{density}/ladder/ladder_{orientation_indices["Ladder"] + 1}.pcd',
    'Support': f'{path}/PCD_Objects_{density}/support/support_{orientation_indices["Support"] + 1}.pcd',
    'Valve': f'{path}/PCD_Objects_{density}/valve/valver_{orientation_indices["Valve"] + 1}.pcd',
    'Railing': f'{path}/PCD_Objects_{density}/railing/railing_{orientation_indices["Railing"] + 1}.pcd'
}

color_map = self.object_placer.color_map

for object_name, data in self.object_placer.placed_objects.items():
    for index, values in enumerate(data):
        position, size, wireframe = values[:3]
        if object_name in mesh_mapping:
            orientation_index = orientation_indices[object_name]
            with open(f'{path}/Orientations/{object_name}.txt', 'r') as file:
                lines = file.readlines()
            orientation_value = lines[orientation_index].strip()
            orientation = tuple(map(float, orientation_value.split(',')))
            mesh_path = mesh_mapping[object_name]
            mesh = self.load_and_place_mesh(mesh_path, size, position, orientation, object_name)
            # mesh.paint_uniform_color(color_map.get(object_name, [1.0, 1.0, 1.0]))
            self.object_placer.placed_objects[object_name][index] = (position, size, (wireframe, mesh))

        elif object_name == "Pipe":
```

```
            continue

def visualize_3d_grid(self):
    o3d.visualization.draw_geometries(self.visual_scene)

def visualize_graph(self, scene_graph):
    pos = nx.spring_layout(scene_graph.graph, k=2)
    plt.figure(figsize=(10, 8))
    nx.draw(scene_graph.graph, pos, with_labels=True, node_color='skyblue', node_size=4000, font_size=12)
    plt.title('Scene Graph Visualization')
    plt.show()

def line_set_to_mesh(self, line_set, radius=0.02, resolution=60):
    mesh = o3d.geometry.TriangleMesh()
    points = np.asarray(line_set.points)
    lines = np.asarray(line_set.lines)
    colors = np.asarray(line_set.colors)

    for i, line in enumerate(lines):
        start = points[line[0]]
        end = points[line[1]]
        color = colors[i]
        cylinder = self.create_cylinder_between_points(start, end, radius, color, resolution)
        mesh += cylinder

    return mesh

def create_cylinder_between_points(self, start, end, radius, color, resolution=60):
    height = np.linalg.norm(end - start)
    cylinder = o3d.geometry.TriangleMesh.create_cylinder(radius, height, resolution=resolution)
```

```
    cylinder.compute_vertex_normals()

    direction = (end - start) / height
    z_axis = np.array([0, 0, 1])
    axis = np.cross(z_axis, direction)
    angle = np.arccos(np.dot(z_axis, direction))
    if np.linalg.norm(axis) > 0:
        rotation_matrix = o3d.geometry.get_rotation_matrix_from_axis_angle(axis * angle)
        cylinder.rotate(rotation_matrix, center=(0, 0, 0))

    cylinder.translate(start)

    cylinder.paint_uniform_color(color)

    return cylinder

def save_object(self, pcd, object_name):
    """Save the object point cloud as a .txt file in the specified dataset structure."""
    dataset_path = os.path.join(self.dataset_name, f"Area_{self.area}", f"room_{self.room}", "Annotations")
    os.makedirs(dataset_path, exist_ok=True)

    # Track object count for naming convention
    if object_name not in self.object_counter:
        self.object_counter[object_name] = 1
    else:
        self.object_counter[object_name] += 1

    object_filename = f"{object_name.lower()}_{self.object_counter[object_name]}.txt"
    object_path = os.path.join(dataset_path, object_filename)
```

```
        # Extract point cloud data (x, y, z, r, g, b) from the pcd
        points = np.asarray(pcd.points)  # (x, y, z)
        colors = np.asarray(pcd.colors) * 255  # (r, g, b) scaled to 0-255
        noise_array = np.random.normal(0, self.point_noise, points.shape)  # Create noise with mean 0 and stddev = noise
        points_with_noise = points + noise_array
        point_cloud_data = np.hstack((points_with_noise, colors))  # Combine (x, y, z) with (r, g, b)

        # Save the point cloud data to a .txt file without any extra headers
        np.savetxt(object_path, point_cloud_data, fmt='%.6f %.6f %.6f %.6f %.6f %.6f')


if __name__ == '__main__':


    number_of_areas = 10
    number_of_rooms = 45
    test_mode = False
    dataset_main = r'E:\PycharmProjects\LowPython\Fixed-Datasets'

    point_densities = [10000]
    point_noises = [5]

    for density in tqdm(point_densities, desc="Point Densities"):
        object_path = f'E:\PycharmProjects\LowPython'
        for noise in tqdm(point_noises, desc="Point Noises", leave=False):
            dataset_name = f"{dataset_main}\Datasets_{noise}_{density}_final"
            for area in tqdm(range(number_of_areas-1), desc="Areas", leave=False):
                for room in tqdm(range(number_of_rooms), desc="Rooms", leave=False):
                    scene_graph = SceneGraph()
                    scene_graph.parse_constraints('constraints.json')
```

```
        object_placer = ObjectPlacer(scene_graph, (50, 50, 20), None, noise, density)
        visualizer = Visualizer(object_placer, dataset_name, area + 1, room + 1, point_noise=noise,
                                point_density=density)
        object_placer.visualizer = visualizer

        object_placer.generate_unique_colors()  # Generate unique colors for each scene
        object_placer.bfs_traverse_and_place()
        visualizer.load_meshes_for_objects(object_path, density)

        if test_mode:
            visualizer.visualize_3d_grid()
            visualizer.visualize_graph(scene_graph)
        # visualizer.save_scene("Datasets")
create_alignment_files(dataset_name)
made_datasets = dataset_name, dataset_name + '_lidar'
copy_directory_with_files(made_datasets[0], made_datasets[1])
process_rooms_in_areas(made_datasets[1])
scale = 0.25
for base_directory in made_datasets:
    merge_txt_files_in_annotations(base_directory)
    scale_dataset(base_directory, scale)
    create_alignment_files(base_directory)
    print(base_directory)
    rename_files_and_folders(base_directory, original_name='ladder', new_name='stairs')
    rename_files_and_folders(base_directory, original_name='desk', new_name='table')
    rename_files_and_folders(base_directory, original_name='pipe', new_name='chs')
    rename_files_and_folders(base_directory, original_name='support', new_name='beam')
    rename_files_and_folders(base_directory, original_name='valve', new_name='equipment')
    rename_files_and_folders(base_directory, original_name='electricbox', new_name='equipment')
    rename_files_and_folders(base_directory, original_name='railing', new_name='wall')
```

```
rename_files_and_folders(base_directory, original_name='stairs', new_name='chs')

#SpaceScene(base_directory, output_root_directory)
```