Mapping and Real-time Navigation with Application to Small UAS Urgent Landing

by

Jeremy D. Castagno

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Robotics)
in the University of Michigan
2021

Doctoral Committee:

Professor Ella Atkins, Chair
Assistant Professor Maani Ghaffari Jadidi
Professor Nadine Sarter
Professor Quentin Stout

Jeremy D. Castagno

jdcasta@umich.edu

ORCID iD: 0000-0001-5458-9787

© Jeremy D. Castagno 2021

ACKNOWLEDGMENTS

I have received a great deal of support and assistance throughout the writing of this dissertation. I first want to thank my advisor, Prof. Ella Aktins, for her unwavering support these last five years. You have consistently helped me every week in research, writing, and even classwork! Whenever I doubted myself or struggled you helped me step back and see the bigger picture of my research goals and work. You paved the runway and provided the wind for my own research to take flight.

I would like to thank my committee members for valuable feedback on my work and ideas. Prof. Sarter, thank you for your encouragement to persevere in adversity and teaching me that rejection is par for the course. Prof. Stout, thank you for instilling in me a passion for writing efficient software and providing the knowledge to create parallel programs. Thank you Prof. Ghaffari for having an open ear and giving me opportunities in academic service.

Thank you to the past teachers and mentors who have supported me to this point. Prof. Hedengren, thank you for hiring me as an undergraduate assistant which began my journey to pursue higher education. Mr. Lenski (Jerry), thank you for being a wonderful mentor and teaching me everything I know about industrial control systems. You taught me that with patience and careful planning even seemingly insurmountable engineering tasks can be accomplished.

Thank you to everyone in the A2SYS lab (in the order I met them): Pedro, Mia, Hossein, Cosme, Brian, Prashin, Matt, Prince, Joseph, Paul, Mark, Anne, John. It has been a great honor in my life getting to know each of you. Matt, Prince, and Prashin, I will always remember the hackathons we joined and the thrill of competing together. Cosme and Brian, you are wonderful friends and research partners. Thank you for being a great example for me and listening to my crazy ideas.

Next I want to thank my family. Mom, you shine as an example of love and service. Thank you for all your support especially during the vulnerable parts of my childhood. To my siblings JD and Meghan, thank you for your love, affection, and friendship. To my kids Kiara, Emily, and Alan: you are treasures in my life and bring me joy every day. It is a privilege to be your Guardian and learn from you. Finally, I want to acknowledge my wife, Natalia. We have taken this journey together. Whenever life became stressful and I faltered, you always stepped up and supported me through adversity. I love you completely. I am so excited for our next journey!

Funding This work was supported by the Rackham Merit Fellowship Program and NSF I/UCRC Award 1738714.

TABLE OF CONTENTS

Acknowledgments . . . . . **ii**

List of Figures . . . . . **viii**

List of Tables . . . . . **xi**

List of Algorithms . . . . . **xiii**

List of Acronyms . . . . . **xiv**

Abstract . . . . . **xvi**

Chapter

1 Introduction **1**

1.1 Motivation 1
1.2 UAS Emergency Landing Background 2
1.3 Geographic Information System Background 4
1.4 Problem Statement 5
1.5 Research Approach and Thesis Outline 6
1.6 Contributions and Innovations 7
1.7 Products 8

2 Polygons from 2D Point Sets **10**

2.1 Introduction 10
2.2 Background 10
2.3 Preliminaries 12
2.4 Methods 13
2.4.1 Triangulation with Half-Edge Decomposition 13
2.4.2 Triangle Filtering 14
2.4.3 Triangular Mesh Region Extraction 14
2.4.4 2D Polygon Extraction 15
2.4.5 Time Complexity 18
2.5 Benchmarking Comparisons 20
2.5.1 Plane Segmented Point Clouds from RGBD Images 20
2.5.2 State Shapes 21
2.5.3 Alphabet Shapes 22
2.6 Random Polygon Tests 23

2.7 Discussion 25
2.8 Conclusion 26

3 Polygons from 3D Data 27

3.1 Introduction 27
3.2 Background 29
3.2.1 Planar Segmentation 29
3.2.2 Polygonal Shape Extraction 31
3.2.3 3D Data Denoising 33
3.2.4 Dominant Plane Normal Estimation 34
3.3 Preliminaries 36
3.4 Mesh Creation 36
3.4.1 Unorganized 3D Point Clouds 36
3.4.2 Organized 3D Point Clouds 37
3.4.3 User Provided Meshes 40
3.5 Mesh Smoothing 42
3.5.1 Laplacian Filter 42
3.5.2 Bilateral Filter 43
3.6 Dominant Plane Normal Estimation 44
3.6.1 Gaussian Accumulator 44
3.6.2 Peak Detection 48
3.7 Planar Segmentation and Polygon Extraction 50
3.7.1 Planar Segmentation 50
3.7.2 Polygon Extraction 53
3.7.3 Algorithm Parallelization 55
3.8 Post Processing 55
3.9 Results 56
3.9.1 Dominant Plane Normal Estimation 56
3.9.2 Unorganized 3D Point Clouds 57
3.9.3 Organized 3D Point Clouds 62
3.9.4 User-Defined Meshes 68
3.10 Discussion 71
3.10.1 Point Cloud Characteristics and Parameter Selection 72
3.10.2 Algorithmic Complexity 74
3.10.3 Future Work 74
3.11 Conclusions 75

4 Roof Shape Classification from Satellite Images and LiDAR Data 76

4.1 Introduction 76
4.2 Background 78
4.2.1 Roof Geometry Classification 78
4.2.2 The Convolutional Neural Network (CNN) 80
4.2.3 Feature Extraction and Classical Machine Learning 81
4.3 GIS Data Processing, Image Generation, and Training 82
4.3.1 Classified Image Set Generation 82

4.3.2 LiDAR Image Construction . . . 83
4.3.3 Satellite Image Construction . . . 86
4.3.4 Stage 1: CNN Architectures and Training . . . 87
4.3.5 Stage 2: SVM and Random Forest Classifier Training . . . 88
4.4 Results . . . 89
4.4.1 Case Study Dataset Generation . . . 89
4.4.2 CNN Training and Results . . . 93
4.4.3 Feature Extraction for SVM and Random Forest Training . . . 94
4.4.4 Analysis of Final Dual Input Model . . . 96
4.5 Discussion and Future Work . . . 99
4.6 Conclusions . . . 101

5 Map-Based Planning for Small UAS Rooftop Landing . . . 102

5.1 Introduction . . . 102
5.2 Background . . . 103
5.2.1 Sensor Based Planning . . . 104
5.2.2 Map-Based Planning . . . 104
5.2.3 Multi-Goal Planning . . . 105
5.2.4 Urban Landscape and Rooftop Landings . . . 107
5.3 Preliminaries . . . 108
5.3.1 Coordinates and Landing Sites . . . 108
5.3.2 3D Path Planning with Mapped Obstacles . . . 108
5.4 Landing Site Database . . . 109
5.4.1 Flat-like Roof Identification . . . 110
5.4.2 Flat Surface Extraction for Usable Landing Area . . . 110
5.4.3 Touchdown Points . . . 112
5.4.4 Landing Site Risk Model . . . 113
5.5 Three-Dimensional Maps for Path Planning . . . 116
5.6 Planning Risk Metric Analysis and Integration . . . 118
5.6.1 Real-time Map-Based Planner Architecture . . . 119
5.6.2 Trade-off Between Landing Site and Path Risk . . . 119
5.6.3 Multi-Goal Planner . . . 120
5.7 Maps and Simulation Results . . . 123
5.7.1 Landing Sites and Risk Maps . . . 125
5.7.2 Case Studies . . . 125
5.7.3 Urgent Landing Statistical Analysis . . . 128
5.8 Discussion and Future Work . . . 130
5.9 Conclusion . . . 132

6 Rooftop Touchdown Point Selection Using On-Board LIDAR and Vision . . . 136

6.1 Introduction . . . 136
6.2 Related Work . . . 138
6.2.1 Unprepared Landing Site Selection . . . 138
6.2.2 Semantic Segmentation . . . 139
6.2.3 Polygon Extraction from Depth Data . . . 140

6.3 Problem Statement 140
6.4 Definitions 142
6.5 Touchdown Point Selection 142
6.5.1 Semantic Segmentation for Scene Understanding 143
6.5.2 Semantic Polygon Extraction 143
6.5.3 Contingency Planning Overview 145
6.6 Simulation Environment 146
6.6.1 Analysis of Rooftops 146
6.6.2 Generating City Rooftop Environments 147
6.6.3 Vehicle, Camera, and LiDAR Models 149
6.7 Semantic Segmentation Results 151
6.7.1 Creating Image Dataset 151
6.7.2 Training and Testing Results 151
6.8 Touchdown Point Selection Results 152
6.8.1 Semantic Polylidar3D Accuracy and Speed 153
6.8.2 Decision Height Analysis 157
6.9 Discussion 157
6.10 Conclusion and Future Work 159

7 Flight Experiment Results for Touchdown Point Selection 161

7.1 Introduction 161
7.2 Touchdown Point Selection 161
7.3 Experimental Setup 162
7.3.1 Sensor Package Construction 162
7.3.2 Sensor Coordinate Frames 163
7.3.3 Quadrotor Frame and Sensor Package Integration 163
7.3.4 Environment 164
7.3.5 Hardware and Software Integration 165
7.4 Experimental Results for Touchdown Point Selection 167
7.4.1 Hand Carry Test 167
7.4.2 Flight Tests 168
7.4.3 Execution Time 169
7.4.4 Trajectory Error Analysis of Intel RealSense T265 171
7.5 Conclusion and Future Work 172

8 Conclusions 174

8.1 Contributions 174
8.2 Future Work 176
8.2.1 Robustly Segmenting Rooftop Point Clouds 176
8.2.2 Improving Polylidar3D 176
8.2.3 Extending Touchdown Point Definition to Fixed-Wing Aircraft 177
8.2.4 Remembering the Human Factor 177
8.2.5 Gaining Confidence in the Data 178
8.2.6 Creating a More Complete Picture of Risk 179

Appendices 181

A.1 Polylidar3D Source Code Summary 182
A.2 Fast Gaussian Accumulator Source Code Summary 185
A.3 Organized Point Filters Source Code Summary 188

Bibliography 190

LIST OF FIGURES

FIGURE

1.1 Motivation for rooftop landing 1
1.2 Common data types used in GIS 5
1.3 Overview of dissertation topics and structure 7

2.1 Demonstration of Polylidar 11
2.2 Delaunay triangulation example 13
2.3 Region growing example from triangulation 14
2.4 Polylidar datastructures 15
2.5 Polylidar boundary following procedure 17
2.6 Extracting a polygon from a plane-segmented RGBD point cloud 22
2.7 Execution and accuracy results from state shape benchmark 23
2.8 Visual comparison of different polygon extraction methods 23
2.9 Example of high and low convexity polygons 24

3.1 Overview of Polylidar3D framework 28
3.2 Example polygons that can be generated from plane segmented point clouds 31
3.3 Example Gaussian Accumulators 35
3.4 Converting an unorganized 3D point cloud to a 3D triangular mesh 37
3.5 Converting an organized point cloud into a 3D triangular mesh 38
3.6 Example non-manifold meshes with condition one violations 42
3.7 Example non-manifold meshes with condition two violations 42
3.8 Visualization of a triangle’s neighborhood during bilateral filtering 44
3.9 Approximation of the unit sphere with an icosahedron 45
3.10 Space filling curve (SFC) of a level four refined icosahedron 46
3.11 Linear prediction model for space filling curve indices on a Gaussian Accumulator . . 48
3.12 Example using Fast Gaussian Accumulator on a triangular mesh 49
3.13 Demonstration of Polylidar3D extracting planes and their polygonal representations . 53
3.14 Example polygon extraction from a planar triangular segment 54
3.15 Example of polygon post processing 56
3.16 Example of Polylidar3D used with unorganized point cloud data 59
3.17 Example of Polylidar3D used with the KITTI autonomous driving dataset 60
3.18 Example of Polylidar3D used with Red Green Blue Depth (RGBD) cameras 64
3.19 Example of using Polylidar3D on a SynPEB scene with the highest noise level 65
3.20 Example of Polylidar3D used with user defined meshes 70

3.21 Results of parallel speedup and execution timing of Polylidar3D 71

4.1 Roof classification data fusion and processing pipeline 77
4.2 Example of a fully connected and convolutional neural network 80
4.3 Example of a SVM and random forest classifier. 82
4.4 Demonstration of LiDAR filtering 86
4.5 Satellite image processing 87
4.6 Convolutional Neural Networks (CNN) architecture templates 88
4.7 Feature extraction for use in SVM and random forest model training 89
4.8 RGB and LiDAR example images of roof shapes. 91
4.9 RGB and LiDAR example images classified as `unknown` 91
4.10 Results of CNN networks on validation set 93
4.11 Accuracy between region-specific and combined training datasets 95
4.12 Test Set 1 Accuracy (Witten/Manhattan) 96
4.13 Confusion Matrices for Test Set 1 (Witten/Manhattan) and Test Set 2 (Ann Arbor). . . 97
4.14 Test Set 1 confidence threshold impact on precision and recall for multiple classes. . . 99
4.15 Test Set 2 confidence threshold impact on precision and recall for multiple classes. . . 100

5.1 Emergency planning logic. 104
5.2 Comparison of multi-goal planning definitions 106
5.3 Satellite image of an urban environment with multiple flat roof landing sites 107
5.4 Processing pipeline to construct landing site and occupancy map databases 109
5.5 Maps of predicted flat rooftops in three cities. 110
5.6 Flat surface extraction from rooftops 111
5.7 Touchdown point extraction on rooftops 112
5.8 Mapping area size to risk 115
5.9 Example occupancy and risk map of New York City 118
5.10 Flow chart of proposed map-based planner 118
5.11 Example Pareto frontier for landing site and path risk 120
5.12 Maps of landing sites and associated risk 126
5.13 Maps of case studies for emergency landing 127
5.14 Pareto frontier of case studies 133
5.15 Maximum distance between landing sites 134
5.16 Metrics for map-based planner 135

6.1 Overview of Semantic Polylidar3D for touchdown point selection 137
6.2 Rooftop-based contingency landing planning overview 141
6.3 Coordinate frames for sensor package 142
6.4 Visualization of Semantic Polylidar3D 145
6.5 Map of Manhattan buildings observed for rooftop assets 147
6.6 Histogram of twelve common rooftop items observed from a Manhattan dataset 148
6.7 Examples of rooftop asset modelling and customization 149
6.8 Example simulated urban city 150
6.9 LiDAR model used in simulation 150
6.10 Image sampling strategy for creating an annotated dataset of rooftop segmentation . . 152

6.11 Gathering test data in simulation environment 154
6.12 Three examples of touchdown point selection on rooftops 155
6.13 Example of Semantic Polylidar3D on a challenging rooftop 156
6.14 Comparison of Semantic Polylidar3D IoU accuracy 156
6.15 Decision height analysis 158
6.16 Proposed rooftop archival polygon update procedure 159

7.1 Sensor package components 162
7.2 Sensor package coordinate frames 163
7.3 Sensor package and quadrotor integration 164
7.4 Flight lab setup 165
7.5 Overview of hardware interfaces and software architecture 166
7.6 Picture of terminal user interface for urgent landing 167
7.7 Real-time constructed meshes and polygons during hand carry test 168
7.8 Visualization of flight path used in all experiments 169
7.9 Flow diagram of flight experiment protocol 169
7.10 Real-time constructed meshes and polygons during flight test 170
7.11 Trajectory of quadrotor from flight #1 171
7.12 Comparison of Motion Capture System (MCS) versus T265 172

A.1 Page 1 of the Polylidar3D repository 182
A.2 Page 2 of the Polylidar3D repository 183
A.3 Page 3 of the Polylidar3D repository 184
A.4 Page 1 of the Fast Gaussian Accumulator repository 185
A.5 Page 2 of the Fast Gaussian Accumulator repository 186
A.6 Page 3 of the Fast Gaussian Accumulator repository 187
A.7 Page 1 of the Organized Point Filter repository 188
A.8 Page 2 of the Organized Point Filter repository 189

LIST OF TABLES

TABLE

1.1 Data sources proposed for small UAS contingency management plans. 4

2.1 Concave hull extraction methods 12
2.2 Parameters for test cases 21
2.3 RGBD plane segmented point clouds benchmark results 21
2.4 Alphabet letter benchmark results, 26 shapes 24
2.5 Random polygon tests; CV = Convexity Metric 24
2.6 Algorithm timings - Mean of 30 runs in milliseconds 26

3.1 Levels of refinement for an icosahedron. 45
3.2 Execution time comparisons for synthetic and real world datasets 57
3.3 Polylidar3D parameters for rooftop detection. 58
3.4 Polylidar3D parameters for KITTI. 61
3.5 Mean execution timings (ms) of Polylidar3D on KITTI 62
3.6 Intel RealSense SDK post-processing filter parameters. 62
3.7 Polylidar3D parameters for RealSense RGBD 63
3.8 Mean execution timings (ms) of Polylidar3D with RGBD data. 64
3.9 Polylidar3D parameters for the SynPEB benchmark test set 66
3.10 SynPEB benchmark results. 66
3.11 Mean execution timings (ms) and accuracy of Polylidar3D on SynPEB 67
3.12 Execution timing (ms) for one and five iterations of Laplacian filtering 68
3.13 Execution timing (ms) for one and five iterations of bilateral filtering 68
3.14 Polylidar3D parameters for the basement mesh 69
3.15 Polylidar3D parameters for the main floor mesh 69

4.1 CNN architectures and hyperparameters 88
4.2 SVM and random forest training configurations 89
4.3 Satellite, LiDAR, and building data sources 90
4.4 Breakdown of roof labels by city 92
4.5 Best CNN model architectures 94
4.6 Best Classifiers using CNN extracted features 96
4.7 Results for recall, precision, and quality evaluation metrics for Test Set 1 97
4.8 Results for recall, precision, and quality evaluation metrics for Test Set 2 98

5.1 Terrain type and property cost 114

5.2 Satellite, LiDAR, and building data sources 125
5.3 Emergency landing case study parameters 125
5.4 Emergency landing case study locations 128

6.1 Common rooftop items with average quantities 147
6.2 Semantic segmentation accuracy results 152
6.3 Semantic Polylidar3D parameters 153
6.4 Execution time (ms) 157
6.5 Maximum height for Semantic Polylidar3D to identify a human on the roof 157

7.1 Sensor package details 165
7.2 Mean and standard deviation of execution times (ms) 170
7.3 Mean Absolute Trajectory Error (ATE) for T265 171

LIST OF ALGORITHMS

ALGORITHM

2.1 Initialize Data Structures for Polylidar . . . 16
2.2 Extract Linear Ring . . . 18
2.3 Extract Holes . . . 19

3.1 Extract Triangles from Organized Point Cloud (OPC) . . . 39
3.2 Extract Half-Edges from OPC . . . 40
3.3 Find Cell Index . . . 47
3.4 Group Assignment . . . 51
3.5 Region Growing Task . . . 52

4.1 Filtering of Airborne LiDAR Point Cloud . . . 85

5.1 Touchdown Point Extraction . . . 113
5.2 Multi-Goal Search . . . 124

6.1 Semantic Triangle Filtering . . . 146

LIST OF ACRONYMS

AHC Agglomerative Hierarchical Clustering

CNN Convolutional Neural Networks

DSM Digital Surface Model

ECAL Enhanced Communication and Abstraction Library

GIS Geographic Information System

GPS Global Positioning System

GPU Graphics Processing Unit

GCS Geographic Coordinate System

INS Inertial Navigation System

IOU Intersection over Union

LiDAR Light Detection and Ranging

ML Machine Learning

MCS Motion Capture System

OPC Organized Point Cloud

OSM OpenStreetMap

PIC Pilot in Command

RGB Red Green Blue

RGBD Red Green Blue Depth

SfM Structure from Motion

SBC Single Board Computer

UAS Unmanned Aircraft Systems

UTM UAS Traffic Management

sUAS small Unmanned Aircraft Systems

VTOL Vertical Take-of and Landing

VR Virtual Reality

ABSTRACT

Small Unmanned Aircraft Systems (sUAS) operating in low-altitude airspace require flight near buildings and over people. Robust urgent landing capabilities including landing site selection are needed. However, conventional fixed-wing emergency landing sites such as open fields and empty roadways are rare in cities. This motivates our work to uniquely consider unoccupied flat rooftops as possible nearby landing sites. We propose novel methods to identify flat rooftop buildings, isolate their flat surfaces, and find touchdown points that maximize distance to obstacles. We model flat rooftop surfaces as polygons that capture their boundaries and possible obstructions on them.

This thesis offers five specific contributions to support urgent rooftop landing. First, the Polylidar algorithm is developed which enables efficient non-convex polygon extraction with interior holes from 2D point sets. A key insight of this work is a novel boundary following method that contrasts computationally expensive geometric unions of triangles. Results from real-world and synthetic benchmarks show comparable accuracy and more than four times speedup compared to other state-of-the-art methods.

Second, we extend polygon extraction from 2D to 3D data where polygons represent flat surfaces and interior holes representing obstacles. Our Polylidar3D algorithm transforms point clouds into a triangular mesh where dominant plane normals are identified and used to parallelize and regularize planar segmentation and polygon extraction. The result is a versatile and extremely fast algorithm for non-convex polygon extraction of 3D data.

Third, we propose a framework for classifying roof shape (e.g., flat) within a city. We process satellite images, airborne LiDAR point clouds, and building outlines to generate both a satellite and depth image of each building. Convolutional neural networks are trained for each modality to extract high level features and sent to a random forest classifier for roof shape prediction. This research contributes the largest multi-city annotated dataset with over 4,500 rooftops used to train and test models. Our results show flat-like rooftops are identified with $> 90\%$ precision and recall.

Fourth, we integrate Polylidar3D and our roof shape prediction model to extract flat rooftop surfaces from archived data sources. We uniquely identify optimal touchdown points for all landing sites. We model risk as an innovative combination of landing site and path risk metrics and conduct a multi-objective Pareto front analysis for sUAS urgent landing in cities. Our proposed emergency planning framework guarantees a risk-optimal landing site and flight plan is selected.

Fifth, we verify a chosen rooftop landing site on real-time vertical approach with on-board LiDAR and camera sensors. Our method contributes an innovative fusion of semantic segmentation using neural networks with computational geometry that is robust to individual sensor and method failure. We construct a high-fidelity simulated city in the Unreal game engine with a statistically-accurate representation of rooftop obstacles. We show our method leads to greater than 4% improvement in accuracy for landing site identification compared to using LiDAR only.

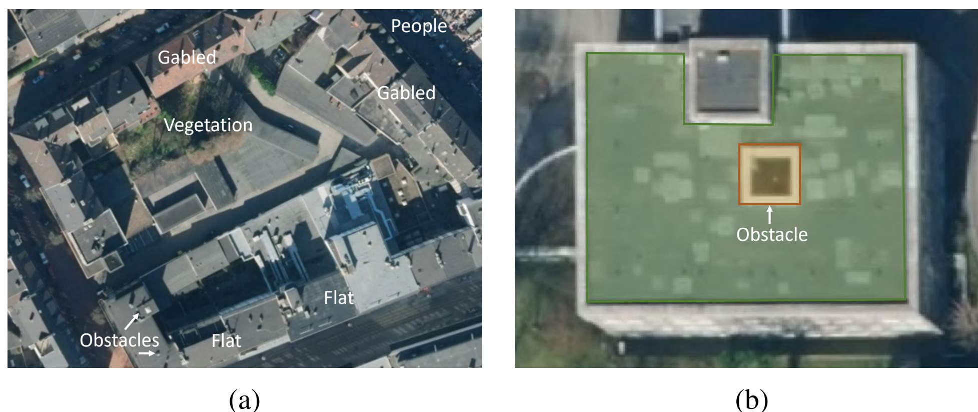
This work has broad impact for the safety of sUAS in cities as well as Urban Air Mobility (UAM). Our methods identify thousands of additional rooftop landing sites in cities which can provide safe landing zones in the event of emergencies. However, the maps we create are limited by the availability, accuracy, and resolution of archived data. Methods for quantifying data uncertainty or performing real-time map updates from a fleet of sUAS are left for future work.

CHAPTER 1

Introduction

1.1 Motivation

Unmanned Aircraft Systems (UAS) are expected to proliferate in low-altitude airspace over the coming decade. Drones with vertical takeoff and landing (VTOL) capability have been proposed to offer fast package delivery, monitor and secure assets, perform inspections, and entertain. Low-altitude operation of UAS in urban areas will require flight near buildings and over people. A primary safety concern is ensuring a robust urgent landing capability [1, 2]. Urgent landing requires landing site selection, trajectory planning, and stable flight control to actually reach the selected site [3]. It is possible a UAS may identify a safe site within sensor range allowing for an immediate landing. However, when no safe site is within range the UAS must devote time and energy to exploring sites beyond sensor range or else utilize pre-processed data to identify a safe site [4, 5]. An onboard database of maps including landing sites can be incorporated into an efficient autonomous decision making framework [6]. Offline data sources such as satellite images, airborne LiDAR point clouds, and existing map data may be used to aid map construction.



(a) (b)

Figure 1.1: Motivation for rooftop landing. (a) Satellite image of an urban environment with multiple flat rooftops. Select roof shapes and obstacles are labeled. (b) A rooftop surface can be represented as a non-convex polygon with an exterior shell (green) and interior hole(s) (orange).

Conventional fixed-wing emergency landing sites such as open fields and empty roadways

are extremely rare in and around cities. Consider the satellite image in Figure 1.1a which shows the typical sparsity of conventional landing zones in the city Witten, Germany. However, these images show a multitude of unoccupied flat rooftops that may provide nearby landing sites for small lightweight UAS. Obstacles such as air conditioning units, skylights, and rooftop entrances on these surfaces may be present and must be explicitly modeled and avoided during an urgent landing. Polygons can accurately and simply represent these flat surfaces as well as obstacles embedded on them per Figure 1.1b. Archived data can be processed such that rooftop landing sites substantially augment existing conventional landing site databases.

1.2 UAS Emergency Landing Background

UAS may experience a number of anomalies requiring emergency landing such as low battery energy, lost communication link, adverse weather, sensor or actuator failure, or operator emergency landing directives. Currently available UAS emergency recovery capabilities are limited to simple methods such as loitering or returning to a predefined waypoint such as the launch location [7, 8]. However, loitering or returning to a distant waypoint may present a higher level of risk to the overflown population and property compared to landing at nearby alternative sites. For this reason, emergency landing research has held a long term focus on autonomous landing site selection and trajectory generation. This dissertation defines related terms as follows.

Definition 1 (Prepared Landing Site)**.** *An area designated for aircraft landing, e.g., runways, heliports, and vertiports. Surface improvements ensure suitable size, shape, and slope characteristics. Markers maximize visibility. Protective barriers and routine inspections minimize probability of obstacles/debris.*

Definition 2 (Unprepared Landing Site)**.** *An open area with no improvements or protections, e.g., fields, roadways, and rooftops. Landing requires a visual assessment to identify or validate a suitable touchdown point.*

Definition 3 (Touchdown Point)**.** *The geographic point within a landing site area where the aircraft intends to first contact the surface.*

The landing site selection process is typically accomplished by utilizing on-board sensors or a database of prepared/unprepared landing sites [9, 4]. Prepared landing sites can be assumed capable of supporting a safe landing provided no other traffic is present. Unprepared landing sites can be mapped *a priori* or identified in real-time. Unprepared sites are not protected thus must be confirmed safe prior to touchdown.

On-board Sensors On-board exteroceptive sensors including camera, radar, and LiDAR can provide a wealth of information about the surrounding environment for use in emergency landing planning. However, the range and field-of-view of the sensor will limit the amount of useful data available. Vision-based systems that employ monocular cameras are widely used because they are lightweight and low cost [10]. Much research focuses on finding landing pads or predefined targets within images using computer vision techniques [11]. Unprepared and unstructured landing sites are found by identifying flat surfaces with sufficient size and shape characteristics within images [9]. Many of these methods find unprepared landing sites from a single image or from multiple images. Single image methods utilize techniques like Canny edge detectors and assume homogeneous textures indicate planarity [12, 13]. Structure from Motion (SfM) algorithms may convert multiple images into points clouds to constructs height maps for planarity assessment [14, 15]. LiDAR sensors offer a significant increase in range, precision, and accuracy but at the cost of increased expense and weight. Fully autonomous landing of helicopters utilizing only on-board LiDAR has been demonstrated [16, 17]. A fusion of sonar and LiDAR for small UAS landing site identification has also shown to be effective at finding unprepared landing sites [18]. However, all such methods require unprepared landing sites to be within range of on-board sensors.

On-board Database Increased data availability, low-cost storage, and accessible cloud-based infrastructure provide new opportunities for UAS risk management. A Geographic Information System (GIS) allows free or low-cost access to a variety of geographic data including elevation maps, man-made structures, land use, and population density. Table 1.1 lists GIS database sources and their use in prior research to support UAS emergency landing. Prior work demonstrates combining and analyzing a variety of data sources to identify unprepared landing sites and accompanying maps for trajectory generation. Additionally, risk models are formulated allowing an autonomous decision framework to choose from multiple candidate landing sites.

Rooftop Landing All references in Table 1.1 strictly focus on identifying and assessing risk for terrain-based landing sites such as open fields or roadways. Per. [10] there has been significantly less research conducted on using building rooftops for landings. Only emerging small UAS are sufficiently lightweight to land on an unprepared roof without risking structural damage or collapse. Ref. [27] proposed rooftop landing methods that rely on pre-positioned landing pads to guide landing site selection and pose estimation. However, this approach assumes a rooftop has already been established as a prepared site, whereas only a few manned helicopter vertiports have actually been established in each urban area today. Ref. [15] uses a monocular camera and employs a dense motion stereo approach for 3D reconstruction while assessing planarity and slope for appropriately-sized landing sites. However, this method is limited to identifying rooftop landing sites only within

Table 1.1: Data sources proposed for small UAS contingency management plans.

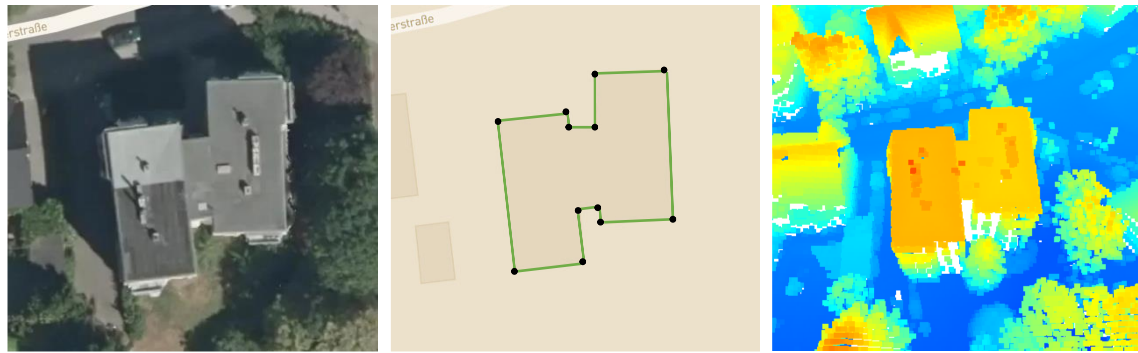
Information Type	Reference	Source
Elevation Data	[19, 20]	Shuttle Radar Topographic Mission (SRTM)
	[4]	US Geological Survey, National Elevation Set
	[21, 22, 23]	Airborne LiDAR Point Clouds
Structures/Buildings	[4]	New York City Land Use Tax-lot
	[24, 19]	OpenStreetMap
Land Use Type	[4]	US Geological Survey, National Land Cover Dataset
	[24, 19]	OpenStreetMap
Population	[4, 25]	US Census, LandScan
	[26]	National Human Activity Pattern Survey
	[24]	Mobile Phone Call Detail Records

the camera field of view and at low altitudes. Our work shows that there may be hundreds of viable rooftop landing sites within 200 meters of a small UAS randomly positioned in a dense urban environment.

1.3 Geographic Information System Background

A Geographic Information System (GIS) is a framework to acquire, label, and update geographic data that model economic, environmental, and social properties [28]. Each GIS data record is marked by a geographic location in an Earth-referenced coordinate frame. Earth-referenced coordinate systems allow heterogeneous data to be aligned in multi-layer maps. This dissertation utilizes three main GIS data types: raster, vector, and point cloud data. Publicly available point cloud data are becoming increasingly available worldwide from both regional and federal governments. Figure 1.2 shows GIS data examples.

Reference Coordinate Systems The most widely used Geographic Coordinate System (GCS) is WGS84 [29] measured in degrees of latitude and longitude where positive values correspond to North of the Equator and East of the Prime Meridian, respectively. This spherical coordinate system is unsuitable for use in metric distance and area calculations needed for landing site selection so projected coordinate systems map the spherical surface to a two-dimensional Cartesian planes called map projections [30]. This dissertation utilizes the Universal Transverse Mercator (UTM) system unless otherwise noted. Data from disparate coordinate systems may be overlaid and aligned by transforming coordinates from one coordinate reference system to another. These transformation



(a) Raster, Satellite Image (b) Vector, Building Outline (c) Airborne LiDAR Point Cloud

Figure 1.2: Common data types used in GIS

are done by specialized software, such as PROJ4 [31].

Raster Data A raster is composed of a matrix of cells organized into rows and columns, where each cell represents information. Common rasters include images, digital terrain elevation models, and rasterized population density maps. Imagery can be captured by satellite or aircraft. Satellite data offer global coverage with the highest resolution currently offered at 30 cm/pixel [32]. Aerial imagery has resolutions as high as 2.5 cm/pixel [33, 32].

Vector Data Vector data provides a more precise representation of the location and shape of world features in comparison to raster data [30]. Common vector features include streets, park/field outlines, and building footprints. Vector features are constructed as ordered pairs of (x,y) vertices which reside on a map projection plane. A vector feature may be represented as points, polylines, or polygons.

Point Cloud Data A point cloud is a collection of 3D points defined in a Cartesian reference frame. Point clouds in GIS are often generated with a top-down vantage point from aircraft outfitted with Light Detection and Ranging (LiDAR), Global Positioning System (GPS), and an Inertial Navigation System (INS). LiDAR scans are stitched together in an airborne LiDAR point cloud.

1.4 Problem Statement

This dissertation addresses the following challenges:

1. Where can a small UAS safely land in a city during an urgent or emergency landing situation?
2. How can maps be constructed and risk evaluated for landing site selection and path planning?
3. How can a small UAS verify a landing zone is safe in real-time on approach to that site?

Previous research has proposed the use of public GIS datasets such as satellite images, airborne LiDAR point clouds, digital elevation maps, census data, and building outlines for terrain-based landing site identification, risk assessment, and path planning [34, 24, 35, 19]. However increased use of small Unmanned Aircraft Systems (sUAS) in urban cities motivates this work to additionally consider building rooftops as landing sites. To our knowledge this dissertation is the first work to explicitly use GIS data to automatically identify flat rooftop buildings, isolate flat surfaces, and find risk-minimum touchdown points that maximize distance to obstacles.

1.5 Research Approach and Thesis Outline

This dissertation proposes to process existing GIS data to extract safe landing sites and risk evaluate for landing site selection. To accomplish this computational geometry algorithms have been developed to enable efficient extraction of flat surfaces as non-convex polygons. The overall structure of this dissertation is shown in Figure 1.3 and can be summarized as accomplishing the following seven tasks:

1. Efficiently extract non-convex polygons with interior holes from 2D point sets. (Ch. 2)
2. Extend polygon extraction methods from 2D point sets to 3D data where polygons represent flat surfaces and interior holes represent obstacles embedded on the surface. (Ch. 3)
3. Classify rooftop shape in a city (e.g., flat) with high confidence in model prediction. (Ch. 4)
4. Assimilate publicly available GIS data such as satellite images, airborne LiDAR point clouds, and building outlines into an urgent landing site database with associated risk metrics as well as occupancy maps for path planning. (Ch. 5)
5. Create a multi-goal planner to efficiently search candidate sites and account for landing site risk as well as flight path risk to that site. The planner identifies a landing site/path pair to minimize combined *total* risk as well as computation time for this search. (Ch. 5)
6. Create a real-time touchdown point selection algorithm to verify a landing zone on approach. Construct a high fidelity simulated city for testing. (Ch. 6)
7. Perform flight experiments to verify the planned touchdown location is safe in real-time using on-board LiDAR. (Ch. 7)

Our research approach begins with developing methods for extracting non-convex polygons from 2D & 3D data in Chapters 2 and 3, respectively. Though the chapters present methods and examples which are general to many domains, this thesis will highlight their use for rooftop surface extraction. Chapter 4 presents our methods of labelling rooftop shapes with a machine learning framework by utilizing satellite image and airborne LiDAR point clouds. Chapter 5 integrates polygon extraction and roof shape prediction in order to create an emergency landing site database

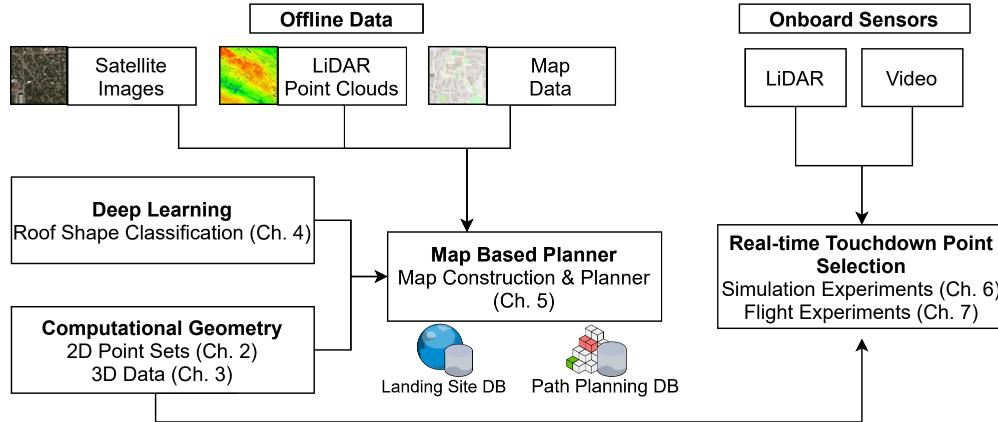


Figure 1.3: Overview of dissertation topics and structure.

that uniquely includes rooftops. We propose several risk metrics that allow a multi-goal planner to choose the risk-minimal landing site and path/pair. Chapter 6 proposes methods to validate a landing site on approach by using on-board LiDAR and camera sensors. We extend Polylidar3D to integrate with semantic segmentation and show improved robustness from individual sensor failure. Finally, Chapter 7 experimentally evaluates the use of Polylidar3D for real-time touchdown point selection with on-board LiDAR.

1.6 Contributions and Innovations

Specific contributions of this dissertation include:

- A faster open source library for non-convex (multi)polygon extraction from 2D point sets. An open source benchmark comparison of leading non-convex polygon extraction techniques in terms of accuracy and speed is provided.
- An efficient and versatile open source framework, Polylidar3D, for non-convex (multi)polygon extraction from 3D data representing flat surfaces. Polylidar3D can handle unorganized and organized 3D point clouds as well as triangular meshes. Computation time is minimized with CPU multi-threading and GPU acceleration.
- Multiple open source and reproducible experiments showing qualitative and quantitative benchmark results of Polylidar3D applied to sensors including LiDAR and RGBD cameras.
- A fast open source dominant plane normal estimation library, FastGA, using a novel Gaussian Accumulator with efficient search.
- A deep learning framework for predicting roof shapes from a fusion of satellite image, airborne LiDAR point cloud, and existing building outline data. Over 4,500 buildings rooftops spanning three cities were manually classified for training, validation, and testing.

- A multi-goal planner that guarantees a risk-optimal solution is found rapidly by avoiding exploration of high-risk options. Plans are optimized over a combination of landing site and path risk metrics.
- Experimental UAS results of real-time landing site validation with touchdown point selection using Polylidar3D.

Specific innovations of this dissertation are:

- A novel computationally-efficient algorithm to extract non-convex polygons from 2D point sets while accounting for holes. The proposed method utilizes half-edge boundary following with edge-case detection instead of alternatives such as the expensive union of triangles.
- The first parallelized non-convex polygon extraction framework working with several forms of 3D data. Polygons represent dominant planar surfaces with interior holes representing the shape of obstacles embedded on their surface. Polylidar3D's speed and data input versatility allow its use for many applications.
- The first large-scale multi-city analysis for predicting roof shapes. The provided annotated dataset contains diverse examples from small to large metropolitan city centers.
- The first rigorous method to incorporate flat rooftops as candidate landing sites for urgent landing of small UAS in cities. Optimal touchdown sites on rooftop surface are determined using geometric methods with risk evaluated according to size and proximity to alternative touchdown sites.

1.7 Products

Publications, software, and datasets connected to this dissertation are:

Conference

- J. Castagno, C. Ochoa, and E. Atkins, "Comprehensive Risk-based Planning for Small Unmanned Aircraft System Rooftop Landing," in 2018 *International Conference on Unmanned Aircraft Systems* (ICUAS), Jun. 2018, pp. 1031–1040, doi: 10.1109/ICUAS.2018.8453483.
- K. McDonough, J. Castagno, J. Player, and E. Atkins, "RANGR: Risk Aware Navigation and Guidance Resilience," presented at the *AUVSI Xponential*, Denver, CO, USA, May 2018.
- J. Castagno and E. M. Atkins, "Automatic Classification of Roof Shapes for Multicopter Emergency Landing Site Selection," in 2018 *Aviation Technology, Integration, and Operations Conference, American Institute of Aeronautics and Astronautics*, 2018.

Journal

- J. Castagno, Y. Yao, and E. Atkins, “Autonomous Rooftop Touchdown Point Selection”. *Journal of Intelligent Robot Systems*. Submitted, Under Review.
- J. Castagno, M. Romano, P. Kuevor, and E. Atkins, “Multi-UAV Wildire Boundary Estimation using a Semantic Segmentation Neural Network,” *Journal of Aerospace Information Systems*, vol. 18, no. 5, pp. 231–249, 2021, doi: 10.2514/1.I010912.
- J. Castagno and E. Atkins, “Map-Based Planning for Small Unmanned Aircraft Rooftop Landing,” in *Handbook on Reinforcement Learning and Control*, Springer, 2021. doi:10.1007/978-3-030-60990-0.
- J. Castagno and E. Atkins, “Polylidar3D - Fast Polygon Extraction from 3D Data,” *Sensors*, vol. 20, no. 17, Art. no. 17, Jan. 2020, doi: 10.3390/s20174819.
- J. Castagno and E. Atkins, “Polylidar - Polygons From Triangular Meshes,” *IEEE Robotics and Automation Letters*, vol. 5, no. 3, pp. 4634–4641, Jul. 2020, doi: 10.1109/LRA.2020.3002212.
- J. Castagno and E. Atkins, “Roof Shape Classification from LiDAR and Satellite Image Data Fusion Using Supervised Learning,” *Sensors*, vol. 18, no. 11, Art. no. 11, Nov. 2018, doi: 10.3390/s18113960.

Open Source C++ libraries (with Python bindings)

- J. Castagno, “Polylidar3D,” [Online] Available: `https://github.com/JeremyBYU/polylidar`, 2020.
- J. Castagno, “Fast Gaussian Accumulator,” [Online] Available: `https://github.com/JeremyBYU/FastGaussianAccumulator`, 2020.
- J. Castagno, “Organized Point Filters,” [Online] Available: `https://github.com/JeremyBYU/OrganizedPointFilters`, 2020.

Open Data Sets

- J. Castagno, “Annotated Roof Shape Dataset,” [Online] Available: `https://www.mdpi.com/1424-8220/18/11/3960/s1`, 2018.
- J. Castagno, “Annotated Roof Asset Dataset,” [Online] Available: `https://github.com/JeremyBYU/UnrealRooftopLanding/blob/master/assets/data/manhattan/buildings.csv`, 2021.

CHAPTER 2

Polygons from 2D Point Sets

2.1 Introduction

This chapter presents Polylidar, an efficient algorithm to transform 2D point sets into simplified non-convex (i.e., concave) polygons with holes. Polylidar begins by triangulating the point set and filtering triangles given user-specified parameters such as maximum triangle edge length. Once filtering is complete, edge-connected triangles are combined into regions creating a set of triangular meshes representing the shape of the point set. Next, Polylidar converts each mesh region to a polygon through a novel boundary following method which accounts for holes. Figure 2.1b shows Polylidar applied to a 2D point set while (c) shows Polylidar used on a plane segmented point cloud from an RGBD image.

We show that the Polylidar algorithm is approximately four times faster than leading open source approaches for concave polygon extraction. Polylidar's speed is attributed to rapidly identifying boundary edges (shell and holes) and then performing boundary following to ensure a valid polygon is returned.

Contributions of this chapter are:

- A faster open source [36] concave (multi)polygon extraction algorithm from 2D point sets.
- A benchmark comparison of leading concave polygon extraction techniques in terms of accuracy and speed.

Below, Sections 2.2 and 2.3 provide background on non-convex shape generation and mathematical preliminaries, respectively. Section 2.4 describes Polylidar algorithms, while Section 2.5 shows benchmark test results of Polylidar versus other methods. Section 2.6 describes test results. Sections 2.7 and 2.8 provide discussion and conclusions.

2.2 Background

Characterizing the shape of a set of 2D points $\mathcal{P}$ has been a long-term focus of computational geometry research. A convex hull is defined as the smallest convex polygon that fully encapsu-

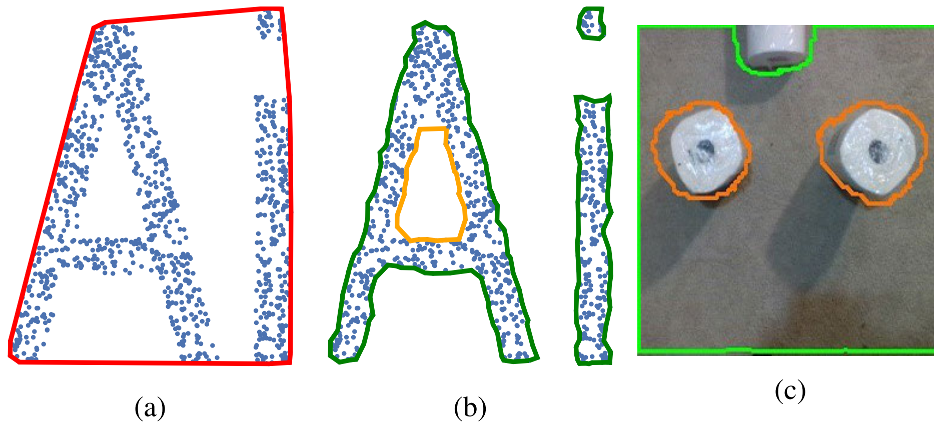


Figure 2.1: Demonstration of Polylidar. (a) Convex hull of a point set (red); (b) MultiPolygon extraction using Polylidar (green). (c) Polygon extraction from a plane segmented point cloud from an Intel RealSense RGBD camera capturing paper towel rolls on a basement floor. Note that Polylidar also identifies holes (orange).

lates all points in a set $\mathcal{P}$. Although widely used to estimate shape, point sets with non-convex distributions are poorly characterized by a convex hull [37]. Convex hull over-estimation can be a serious issue when the points represent physical objects, e.g., obstacle free navigable areas. Several algorithms have been developed to construct shapes that "fit" or "cover" point sets more closely.

Figure 2.1 compares convex and concave hulls. Figure 2.1b is the multipolygon output of Polylidar described below. While there is a unique convex hull, there is no true or unique concave hull. Concave hull algorithm implementations can also have different output types. Some return only an unordered set of edges while others return a single polygon. Some algorithms return multiple disconnected polygons (multipolygon), and some can generate holes inside a polygon.

The α-shape algorithm is an early strategy to generate a family of shapes ranging from a convex hull to a point set [38]. The parameter α dictates the radius of a closed disk used to prune/remove area in the convex hull. This disk is allowed to move freely shaving off the excess shape until it finds points. When disk radius is large, ideally infinite, the convex hull is produced; when disk radius is infinitesimally small only the points remain. A common implementation of α-shape organizes points using Delaunay triangulation and filters triangles whose circumcircle radius is less than α. The final shape is represented by the remaining edges and triangles. Note that the α-shape method creates multiple non-intersecting shapes with the possibility of holes.

The algorithm in [37] produces polygons from point sets called χ-shapes. Like some α-shape implementations, the χ-shape approach begins with Delaunay triangulation to order and spatially connect data points. The algorithm differs by iteratively removing the longest exterior edges from triangulation based on a specified maximum length parameter l. A corner case occurs when edge removal results in a non-simple polygon, i.e., the polygon wraps into itself; in this case edge removal

is skipped. The χ-shape produced is a single polygon with no possibility of holes.

The geospatial software library Spatialite [39], an extension to SQLite [40], contains a concave hull extraction procedure. The algorithm again starts with Delaunay triangulation then analyzes the distribution of each triangle's edge length to determine mean μ_l and standard deviation σ_l. Any triangle with edge length greater than $C \cdot \sigma_l + \mu_l$ is removed, where C is a user-defined parameter. The final geometry returned is the union of all triangles computed with GEOS, a high performance open source geometry engine. The output may be a multipolygon (i.e., multiple disjoint polygons) with the possibility of holes inside each.

PostGIS is a geospatial database of computational geometry routines such as the concave hull method in [41]. This algorithm first calculates the convex hull and then shrinks the hull by adjusting vertex connections to closer points which "cave in" the hull. This process recursively shrinks a boundary until a user-specified percent reduction in area from the convex hull is achieved. The resulting shape is a single polygon with the possibility of holes.

Table 2.1: Concave hull extraction methods

Algorithm	Output	Holes?
χ-shapes	unordered set of edges	No
CGAL α-shape	unordered set of edges	Yes
Spatialite	(multi)polygon	Yes
PostGIS	polygon	Yes
Polylidar (new)	(multi)polygon	Yes

Table 2.1 provides a summary of the concave hull algorithms discussed above. The Computational Geometry Algorithms Library (CGAL) is used as the implementation of the α-shape method [42]. Note that the time complexity of all algorithm implementations, with the exception of PostGIS, is $\mathcal{O}(n \log n)$. This chapter contributes a procedure to more rapidly compute (multi)polygon output with the possibility of holes. Though this is a complex output to generate, we show through benchmarks that our algorithm and implementation outperforms other available approaches.

2.3 Preliminaries

A 2D *point set* is an arbitrarily ordered set of two dimensional points in a Cartesian reference frame. Each point is defined by orthogonal bases $\hat{\mathbf{e}}_x$ and $\hat{\mathbf{e}}_y$ with

$$\vec{p_i} = x\,\hat{\mathbf{e}}_x + y\,\hat{\mathbf{e}}_y = [x, y] \tag{2.1}$$

where x, y are plane coordinates.

An n-point array $\mathcal{P} = \{\vec{p_1}, \vec{p_i}, \ldots, \vec{p_n}\}$ contains points $\vec{p_i} \in \mathbb{R}^2$ indexed by i. A triangular mesh $\mathcal{T}$ is defined by

$$\mathcal{T} = \{t_1, t_i, \ldots, t_k\} \tag{2.2}$$

where each t_i is a triangle with vertices defined by three point indices $\{i_1, i_2, i_3\} \in [1, n]$ referencing points in $\mathcal{P}$.

We follow the Open Geospatial Consortium (OGC) standard [43] for defining *linear ring* and *polygon*. A linear ring is a consecutive list of points that is both closed and simple. This requires a linear ring to have non-intersecting line segments that join to form a closed path. The key components of a valid polygon are a single exterior linear ring representing the *shell* of the polygon and a set of linear rings (possibly empty) representing *holes* inside the polygon.

2.4 Methods

Sections 2.4.1, 2.4.2, and 2.4.3 describe the triangulation data structures, filtering, and mesh extraction respectively. Section 2.4.4 describes polygon extraction.

2.4.1 Triangulation with Half-Edge Decomposition

Polylidar begins with the Delaunator library [44] performing a Delaunay triangulation of point set $\mathcal{P}$. The original algorithm was written in JavaScript but a C++ port of the library is used in Polylidar [45]. Note that we have modified Delaunator to use robust geometric predicates to ensure correctness during triangulation [46]. Delaunator was chosen for its ease of integration, speed, and output data structure which returns a *half-edge* triangulation. A half-edge triangulation decomposes a shared edge using two half-edges A→B and B→A. An example of this decomposition and resulting data structures is shown in Figure 2.2.

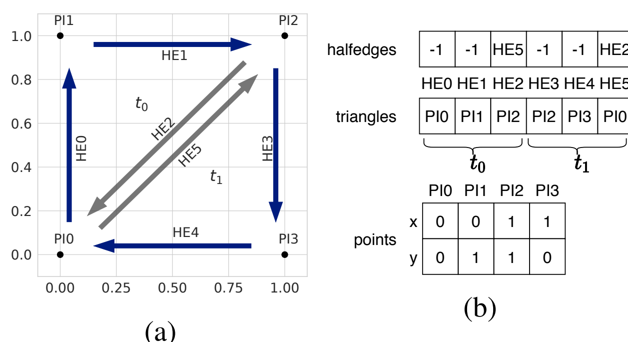


Figure 2.2: Delaunay triangulation example. (a) Triangulation of a square point set using Delaunator [44] with output data structure indexed by half-edge ids in (b). HE=half-edge, PI=point index, t=triangle. Grey edges show shared edges decomposed individually.

Figure 2.2a triangulates point set {PI0, PI1, PI2, PI3}. Triangulation produces two triangles, t_0 and t_1, with half-edges {HE0, HE1, HE2} and {HE3, HE4, HE5}, respectively. Each half-edge supports clockwise travel to the next half-edge in that triangle's edge set. Figure 2.2b lists the resulting $halfedges$, $triangles$, and $points$ data structures. The $halfedges$ array is indexed by a half-edge reference id. It provides the opposite half-edge of a shared edge if it exists; otherwise -1 is returned. The $triangles$ array is also indexed by half-edge id and gives the starting point index of the associated half edge. The relationship between half-edge and triangle indices is $t = \text{floor}(he/3)$.

2.4.2 Triangle Filtering

As with Spatialite and α-shape methods the initial shape starts with k triangles in $\mathcal{T}$ per Eqn. 2.2 returned from Delaunay triangulation. Also similar to α-shape and Spatialite methods, Polylidar filters triangles by configurable criteria for each triangle. Polylidar allows the user to perform triangle filtering using either the α parameter or maximum triangle edge length parameter l_{max}. The filtered triangle set is denoted $\mathcal{T}_f$.

2.4.3 Triangular Mesh Region Extraction

An iterative plane extraction procedure inspired from [47] generates subsets of $\mathcal{T}_f$ that are spatially connected. These subsets are denoted $\mathcal{T}_r$ which represent triangular mesh *regions*. A spatial connection between triangles exists when they share an edge. A random seed triangle is selected from $\mathcal{T}_f$ where a new region is created and expanded by its adjacent edge neighbors from the *halfedges* data structure. Region growth halts when no more triangles in $\mathcal{T}_f$ connect to the region. The process repeats with another seed triangle until all triangles in $\mathcal{T}_f$ have been examined.

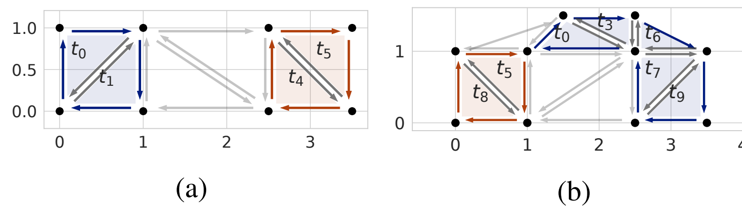


Figure 2.3: Region growing example from triangulation. (a) Example of two regions extracted denoted by orange and blue. Triangles t_0 and t_1 are one region while t_4 and t_5 are another. (b) Two regions are also extracted with a shared vertex.

Figure 2.3a shows triangular mesh region examples. Distinct regions are shown in orange and blue; light grey edges denote triangles that have been filtered out. The output of this step is a set of spatially connected triangular mesh regions, $\mathcal{T}_R$, where each specific region, $\mathcal{T}_{r,i}$, is a set of triangle *indices*. We denote the set of m triangular mesh regions as:

$$\mathcal{T}_R = \{\mathcal{T}_{r,1}, \mathcal{T}_{r,i}, \dots, \mathcal{T}_{r,m}\} \tag{2.3}$$

$$\mathcal{T}_{r,i} = \{t_i, \dots, t_j\} \tag{2.4}$$

2.4.4 2D Polygon Extraction

Polygon extraction has three steps: data structure initialization, concave shell extraction, and hole(s) extraction. Each of these steps is described below. Note that polygon extraction is independent of the specific triangular mesh regions $\mathcal{T}_{r,i}$, thus subsequent notation will drop the i index for brevity when used in algorithms. The following steps are executed for each of the m regions in $\mathcal{T}_R$ to generate m polygons.

2.4.4.1 Data structure initialization

Data structure initialization is shown in Algorithm 2.1 which produces three data structures: a boundary half-edge set, a point index hash map, and the extreme point. A visual example of these data structures is shown in Figure 2.4. Boundary half-edge set $\mathcal{BE}$ contains the half-edge indices that are on the exterior border of a region, marked in blue in Figure 2.4a. A half-edge is marked as a boundary if it has no opposite half-edge (meaning it is on the convex hull of the full triangulated set) or if its adjacent triangle is not in $\mathcal{T}_{r,i}$. The last check is important because a half-edge may share an edge with an interior triangle that is not part of $\mathcal{T}_{r,i}$ as seen in the rightmost edge for the blue region in Figure 2.3a. The *halfedges* data structure is fixed at triangulation and is not aware of filtered triangles or the regions discussed in Section 2.4.3.

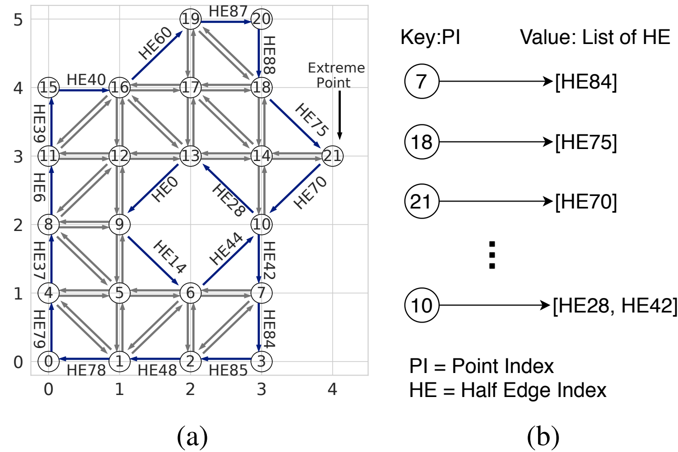


Figure 2.4: Polylidar datastructures. (a) The boundary half-edge set is marked in blue and point index 21 (PI21), the farthest point on the x-axis, is noted. (b) A sample of the resulting point index hash map, PtE is shown. Note that the display order has been arbitrarily chosen.

The second data structure is a point index hash map, PtE, whose *key* is a point index and *value* is a *list* of outgoing boundary half-edges from the keyed point index. This *unordered* hash map is represented in Figure 2.4b; note the keyed point index 7 mapping to the single element list containing half-edge 84. The final data structure represents an extreme point in the triangle mesh, referring to the point farthest to the right on the x-axis. This point will be used as the starting point index when extracting the concave hull to help ensure extraction does not start on a hole edge. Multiple points may exist on the extreme edge; the algorithm will track the first one found.

Algorithm 2.1: Initialize Data Structures for Polylidar

Input : Triangular Mesh Region , $\mathcal{T}_r = \{t_i, \ldots, t_k\}$
Shared Halfedges, $halfedges$
Triangles Point Index, $triangles$

Output : Half Edge Set , $\mathcal{BE} = \{he_i, \ldots, he_n\}$
Point Index Hash Map, PtE
Extreme Point, pi_{xp}

```
1  BE = ∅ ;                                   // boundary half-edge set
2  PtE = ∅ ;                                  // Point to half-edge hashmap
3  pi_xp = 0 ;                                // will be overwritten
4  for t_i ∈ T_r do
5      for he_i ∈ t_i do
6          he_j = halfedges[he_i];            // opposite edge
7          t_j = floor(he_j/3) ;              // adjacent tri
8          if t_j ∉ T_r then
9              BE = BE + he_i ;               // boundary edge
10             pi = triangles[he_i]
11             pi_xp = TrackXp(pi, pi_xp)
12             if pi ∉ PtE then
                   /* create half-edge list                              */
13                 PtE[pi] = [he_i]
14             else
15                 Append(PtE[pi], he_i)
16     end
17 end
18 return BE, PtE, pi_xp
```

2.4.4.2 Concave Shell Extraction

Outer shell extraction begins by traversing the half-edge graph, starting with the half-edge provided by the extreme point. As the edges are traversed the point indices are recorded in a list representing the linear ring of the concave hull. Edges are removed from the boundary half-edge set, $\mathcal{BE}$, as they are traversed. In Figure 2.4a the extreme point index is PI21 and the starting half-edge is

HE70. This starting half edge and start point index are arguments to the `ExtractLinearRing` procedure in Algorithm 2.2, with the procedure halting when edge traversal returns back to the starting point index, indicating a closed linear ring has been extracted. The hole in this shape, represented by edges (HE28, HE0, HE14, HE44), with a shared vertex at PI10, must be carefully handled as explained below. This is an example of an non-manifold mesh.

The example in Figure 2.4 begins with HE70 traversing to PI10. The outgoing boundary half-edges for this point index are determined from PtE which provides a list of both HE28 and HE42. However HE28 is an edge for a hole in this polygon while HE42 is the correct half-edge to traverse for the outer shell. The `SelectEdge` procedure determines which of these edges to choose and is visually outlined in Figure 2.5a. Angles between the proposed edges and previous edge HE70 are calculated and the edge with the largest angle is chosen which guarantees the largest concave hull. This edge cannot be a hole edge because that would imply that the hole is outside the concave shell, which is invalid.

On rare occasions the extreme point may have more than one outgoing half edge, meaning that a hole is connected to it. This can be handled in the same way stated above by using the `SelectEdge` procedure. The only difference is that the previous hull edge is not known (the procedure has just started), but since we know we are on the far right of the hull we can substitute the previous edge for the unit vector [0,1] per Figure 2.5b. This unit vector is guaranteed to provide a stable order of the angle differences which would have been provided by the actual previous hull edge.

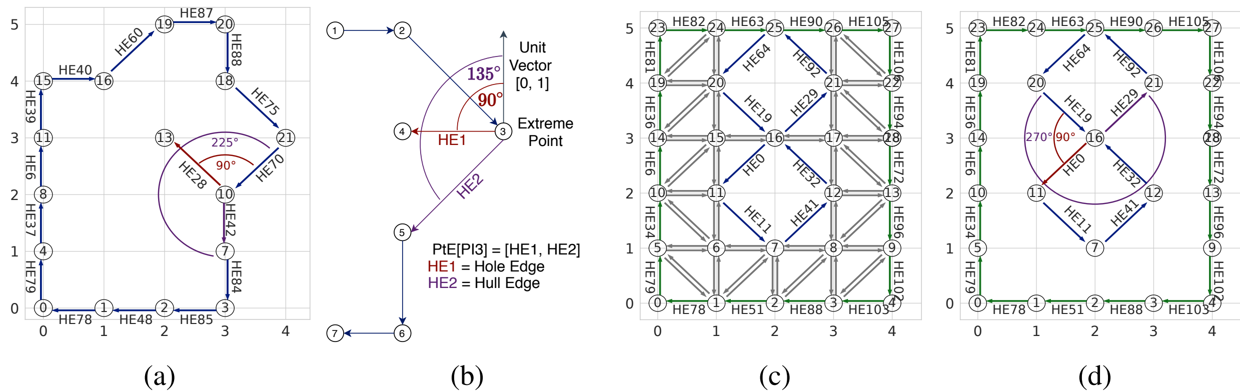


Figure 2.5: Polylidar boundary following procedure. (a) Edge selection for Fig. 2.4a. HE70 leads to point index PI10 during shell extraction. Half-edges HE28 and HE42 leave PI10. The correct edge to follow (HE42) has the greatest angle with HE70. (b) If the extreme point has two outgoing edges (HE1, HE2), choose the edge with largest angle difference with the unit vector [0,1]. This is edge HE2. (c) Edge case of two holes sharing the same vertex at PI16. The outer shell (green) is already extracted. (d) When traversing from HE19 to point index PI16, two outgoing edges (HE0 and HE29) are found. Edge HE29 with the largest angle difference from HE19 is chosen.

Algorithm 2.2: ExtractLinearRing

```
  Input  : Half Edge Set , BE = {he_i, ..., he_n}
           Point Index Hash Map, PtE
           Starting half-edge, he
           Start point index, startPI
           Triangles Point Index, triangles
  Output : Linear Ring , lr = [pi_1, ..., pi_k]
1  lr = [ ] ;                                      /* empty linear ring */
2  while True do
3  |  BE = BE \ he
4  |  he_t = NextTriangleEdge(he)
5  |  pi = triangles[he_t]
6  |  Append(lr, pi)
7  |  if pi is startPI then
   |  |  /* closed linear ring                                        */
8  |  |  break
9  |  nextEdges = PtE[pi]
10 |  he = SelectEdge(he, nextEdges)
11 end
12 return lr
```

2.4.4.3 Hole(s) Extraction

After the outer shell of the concave hull has been determined, only the holes remain to be found (if any holes exist). Any edges that remains inside $\mathcal{BE}$ are hole edges and will be extracted using Algorithm 2.3. A half-edge is randomly chosen from $\mathcal{BE}$ for which the same `ExtractLinearRing` procedure is run. Figure 2.5c shows a corner case of a non-manifold mesh that must be handled if two holes share the same vertex. The previously extracted concave shell is displayed in green while the remaining half-edges to be processed are in blue; note the shared vertex at PI16. Figure 2.5d shows the event when HE19 is randomly chosen for hole extraction leading to PI16. HE0 or HE29 is chosen in the manner previously discussed: the edge with largest angle guarantees the smallest hole thus is chosen. If the other edge was chosen this would indicate a hole inside a hole which is invalid.

2.4.5 Time Complexity

This section describes the time complexity of Polylidar. The Delaunay triangulation is first computed in $\mathcal{O}(n \log n)$ [48]. Region extraction from Section 2.4.3 is $\mathcal{O}(n)$:

- At most $t = 2n - 2$ triangles are returned from Delaunay triangulation by Euler's formula so t is linear in n.

Algorithm 2.3: Extract Holes

Input : Half Edge Set , $\mathcal{BE} = \{he_i, \ldots, he_n\}$
Point Index Hash Map, PtE
Triangles Point Index, $triangles$

Output : Set of Linear Ring Holes , $\mathcal{HR} = \{lr_1, \ldots, lr_k\}$

```
1 HR = ∅ ;                                            /* empty hole set */
2 while BE is not empty do
3     he = RandomChoice(BE)
4     pi = triangles[he]
5     lr = ExtractLinearRing(BE, PtE, he, pi, triangles)
6     HR = HR + lr
7 end
8 return HR
```

- The maximum of $2 * t$ iterations occurs. One loop filters triangles; worst case no triangles are removed. A second loop occurs over remaining connected triangles.
- Determining shared edges for expansion requires an $\mathcal{O}(1)$ lookup in the $halfedges$ array.
- Output generation $\mathcal{T}_r$ is $\mathcal{O}(1)$ for each insertion.

Polygon extraction is also $\mathcal{O}(n)$. The initialization procedure in Algorithm 2.1 is $\mathcal{O}(n)$ per the following analysis:

- Assuming the worst case, the algorithm loops through every edge of every triangle, providing a maximum number of iterations of $3 \cdot t$. Therefore the number of iterations is linear with n.
- Determining if an edge is a boundary edge is an $\mathcal{O}(1)$ operation. Line 6 is $\mathcal{O}(1)$ lookup in halfedges array. Line 8 is $\mathcal{O}(1)$ lookup in the input triangle set.
- Output generation, the half-edge set and point index hash map, is $\mathcal{O}(1)$ for each insertion.

Algorithm 2.2 is linear in n for similar reasons:

- The number of iterations is the number of boundary edges computed, a subset of the iterations in Alg. 2.1.
- The output linear ring is $\mathcal{O}(1)$ for each insertion.
- The `SelectEdge` procedure is $\mathcal{O}(1)$ with no loops.

Finally Algorithm 2.3 is also linear with respect to n. It is executed as many times as there are holes in a polygon. A hole contains a minimum of one triangle; we have shown previously that t grows linearly with n. Selecting a random half-edge, Line 3, is an $\mathcal{O}(1)$ operation. Overall, Polylidar has complexity $\mathcal{O}(n \log n)$ from the initial triangulation.

2.5 Benchmarking Comparisons

This section benchmarks Polylidar against other common concave hull extraction methods which also extract holes; all code is open source[1]. Three other implementations are tested: CGAL's Alpha Shape function and the ST_ConcaveHull function from PostGIS and Spatialite. For uniformity, Polylidar and CGAL are set to use the same α parameter to guarantee exact shape reproduction. Note that CGAL's Alpha Shape returns an unordered set of boundary edges; it does not convert these edges into a valid (multi)polygon. These edges produce the same shape as Polylidar when drawn on a canvas, but lack the desired polygon semantic data structure. PostGIS's concave hull implementation only returns single polygons, so MultiPolygon test cases are not evaluated against it. Both PostGIS and Spatialite are databases which require upload of the point set prior to algorithm execution; benchmark timing does not include data upload time.

Section 2.5.1 provides a benchmark from plane segmented point clouds produced by an RGBD camera. Section 2.5.2 generates synthetic 2D point sets from the state shapes of California (CA) and Hawaii (HI) to explore how the algorithms scale with respect to point size. Section 2.5.3 shows a similar benchmark but with the English alphabet. All utilize ground truth (multi)polygon shape GT to evaluate shape accuracy. Each implementation takes as input a point set and produces a concave shape, CS, which is similar to the ground truth polygon. The L^2 error norm, the area of the symmetric difference between GT and CS, is computed to enable evaluation of shape error $\frac{area((GT-CS)\cup(CS-GT))}{area(CS)}$.

Each implementation contains its own parameter(s) modified to minimize L^2 error. Shape accuracy is therefore subject to parameter selection. Table 2.2 displays the parameters chosen and used for all test cases (RGBD, CA, HI, Alphabet). Rows with two parameters separated by a semicolon indicate parameters for use with non-hole and hole cases. Polylidar and CGAL use the same α parameter adjusted on a case by case basis. For each case we calculate point density p_d and compute parameter α as $2p_d^{-1}$. This gives reasonable but not necessarily optimal results. Spatialite's concave hull implementation has parameter C which at its default value ($C = 3$) produces excellent results. C is adjusted as needed (for CA, HI) to further reduce error. PostGIS' *target percent* is set to provide the optimal accuracy based on percent area reduction required. The most important takeaway when interpreting accuracy is thus trends in accuracy, not small numerical differences.

2.5.1 Plane Segmented Point Clouds from RGBD Images

Point clouds were generated with an Intel RealSense D435i camera at 424X240 resolution from eleven different scenes. Ten scenes were taken with the camera 1.5m above ground level pointing directly downward as shown in the top of Figure 2.6. Floor obstacle positions and orientations were

[1]https://github.com/JeremyBYU/concavehull-evaluation

Table 2.2: Parameters for test cases

Algorithm	Parameter	RGBD	CA	HI	Alph.
CGAL/Polylidar	α	$2p_d^{-1}$	$2p_d^{-1}$	$2p_d^{-1}$	$2p_d^{-1}$
Spatialite	C	3.0	2.0	2.0;1.3	3
PostGIS	$target$ %	Varies	0.76;0.72	-	Varies

changed in each scene. The camera was placed higher and angled for the eleventh scene shown in the bottom of Figure 2.6. The floor can be quickly segmented using planar segmentation techniques [49, 50]. However for this experiment the floor was manually segmented, rotated to align with the XY image plane, and subsequently projected. This creates a 2D point set of the floors 3D point cloud. The ground truth polygon of each segmented point cloud was labeled by hand to provide accuracy scores. The average size of the eleven segmented point clouds is 83,184 points. Table 2.3 displays the aggregate execution timings and accuracy results of all eleven points clouds for each algorithm. Polylidar is fastest. Polylidar, CGAL, and Spatialite have similar accuracies. Note that Polylidar and CGAL are configured to produce the same shape and therefore have the same L^2 error values.

Table 2.3: RGBD plane segmented point clouds benchmark results

	L^2 error %			Time (ms)		
Algorithm	mean	std	max	mean	std	max
Polylidar	2.2	1.5	6.4	47.9	4.3	50.9
CGAL	2.2	1.5	6.4	248.3	25.0	267.7
PostGIS	7.5	1.6	9.9	2734.7	249.3	2939.9
Spatialite	2.2	1.5	6.3	13333.0	2486.6	16386.5

2.5.2 State Shapes

Figure 2.7 shows CA and HI test case geometries (first column), execution times (second column), and error results (third column). Each state shape is processed with and without random holes ; dashed lines indicate results where holes are included in the ground truth polygon. Point sets are randomly sampled from the state shapes. Each test was run 10 times with input point set sizes ranging from (2, 4, 8, 16, 32, 64) thousand points with mean timing and error plotted. Confidence intervals are provided for execution timing, however they are almost imperceptible because the variance is low at this scale. Polylidar and CGAL are significantly faster than the other methods, with Spatialite having the slowest implementation. An inset (zoomed) box that focuses solely on CGAL and Polylidar is shown in the second column, showing that on average Polylidar is ~ 4

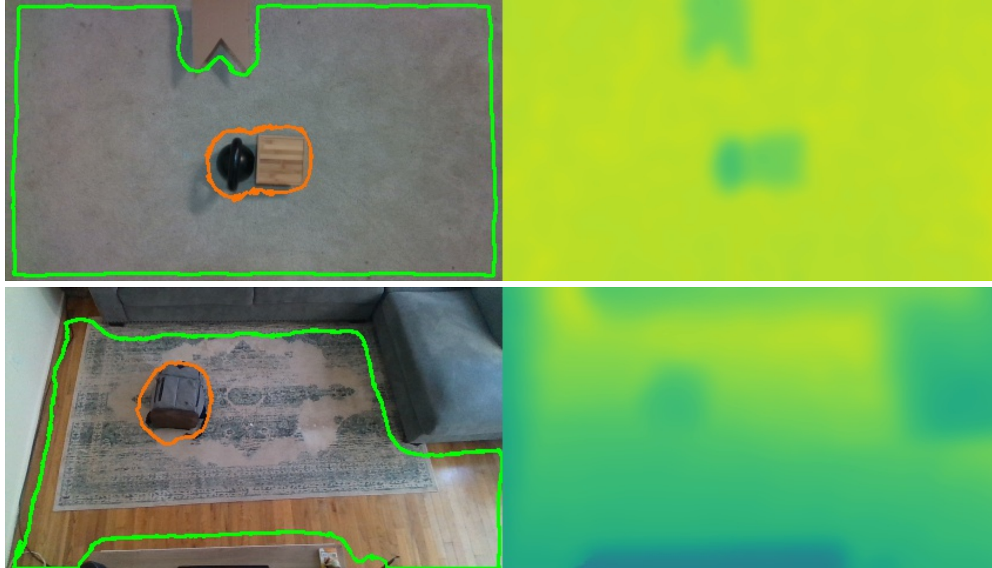


Figure 2.6: Extracting a polygon from a plane-segmented RGBD point cloud. Two example scenes (top/bottom) from RGBD benchmark. A point cloud is generated from depth image (right) and manually segmented to include only the ground floor. The polygonal output of Polylidar is shown in the RGB image (left). Green is the hull, orange represents holes.

times faster than CGAL. The presence of holes affected each method differently: decreased time in Spatialite (fewer triangles to union), increased time for PostGIS (a decrease in *target percent* increases run-time). No significant changes were noted for CGAL and Polylidar.

Spatialite produced shapes with the least error, followed by Polylidar/CGAL and then PostGIS. Spatialite has the lowest error because it incorporates triangle edge length statistics into its triangle filtering which better handles random sampling. In contrast, Polylidar/CGAL offer comparable accuracies with RGBD data due to the more uniform point distribution in top-down RGBD imagery. PostGIS error increased markedly with holes since it did not accurately reproduce them. Figure 2.8 shows a visual comparison of CA concave polygon outputs for each algorithm.

2.5.3 Alphabet Shapes

Polygons from 26 capital letters of the English alphabet were generated and 2000 points randomly sampled inside. The "A" in Figure 2.1b shows an example capital letter with the output of Polylidar's concave hull. Table 2.4 provides aggregate statistics of all 26 test cases. Polylidar continues to lead in speed. Spatialite leads in accuracy by a marginal amount. The alphabet shapes are significantly more concave than previous benchmarks. Documentation of PostGIS indicates that the run time grows quadratically as concavity increases leading to the high execution times observed [41].

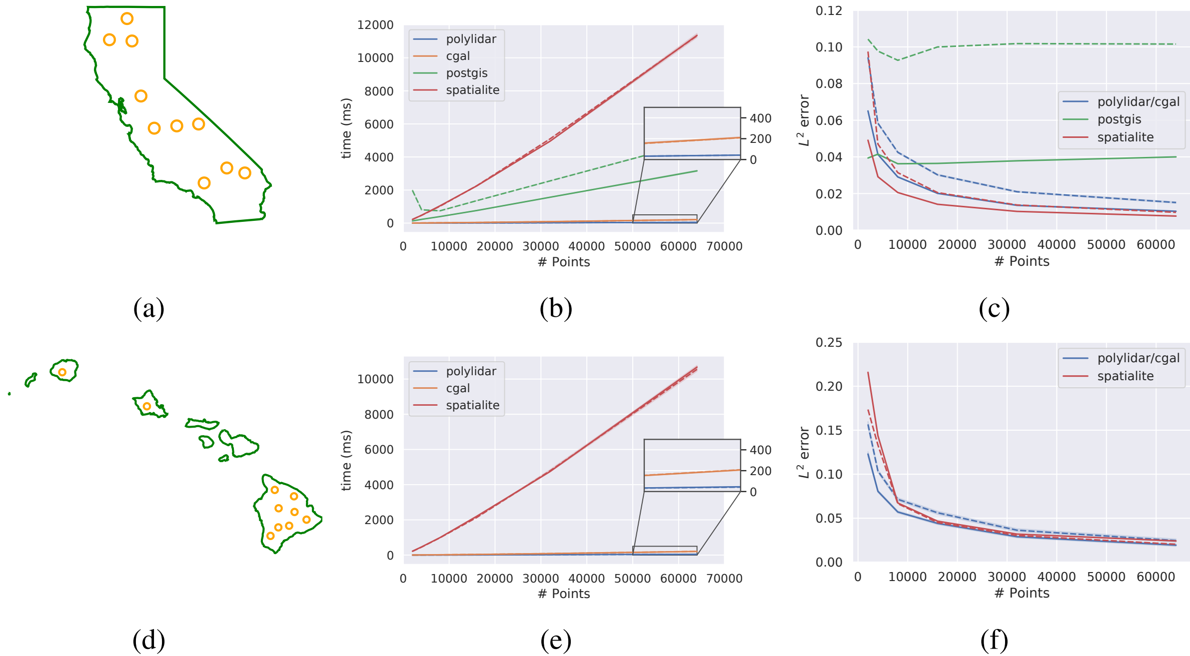


Figure 2.7: Execution and accuracy results from state shape benchmark. Rows from top to bottom correspond to outlines of California (CA) (a, b, c), and Hawaii (HI) (d, e, f) with random holes inserted. The first column shows ground truth polygons with circular holes in orange. The second column shows execution time as a function of number of 2D points provided. The third column shows shape error as a function of number of 2D points provided. Dashed lines show results where holes were placed inside the polygon outline, while solid lines show results with no holes. PostGIS cannot handle MultiPolygons thus was not tested for HI.

Figure 2.8: Visual comparison of different polygon extraction methods. Concave polygon output from Polylidar/CGAL (left), Spatialite (center), and PostGIS (right). Input to each algorithm was a 4000 point set sampled from the California (CA) polygon with holes per Figure 2.7a.

2.6 Random Polygon Tests

More than 19,600 polygons were randomly generated to test Polylidar. Half the test cases had random holes. Polygon complexity is characterized by convexity metric

$$CV = \frac{Area(P)}{Area(CH(P))}$$

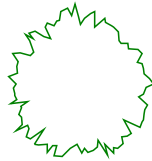
Table 2.4: Alphabet letter benchmark results, 26 shapes

	L^2 error %			Time (ms)		
Algorithm	mean	std	max	mean	std	max
Polylidar	12.8	1.8	16.8	1.2	0.3	2.4
CGAL	12.8	1.8	16.8	5.4	0.9	7.2
PostGIS	36.5	9.9	53.7	13091.8	7500.6	28451.0
Spatialite	11.2	4.5	22.1	230.2	6.3	242.9

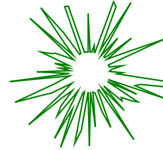
where P is the polygon and $CH()$ is the convex hull function. A convexity of 1 indicates the sample polygon is its convex hull. 8,000 points were randomly sampled for each polygon and input to Polylidar with the α parameter from Table 2.2. Execution time and accuracy are summarized in Table 2.5. The table is partitioned into high, medium, and low ground truth polygon convexity defined by $CV \geq 0.75$, $0.75 < CV \geq 0.55$, and $CV < 0.55$ respectively. Every polygon produced by Polylidar was confirmed valid independently by the GEOS geometry library. As polygon convexity (CV) decreases Polylidar shape estimation accuracy also decreases. Polygons in our "low" convexity class have extremely non-convex shapes, the lowest with $CV = 0.26$ per Figure 2.9.

Table 2.5: Random polygon tests; CV = Convexity Metric

	L^2 error %			Time (ms)		
CV	mean	std	max	mean	std	max
hi	4.4	0.5	6.1	4.6	0.1	5.0
mid	8.0	1.1	13.0	4.6	0.1	8.1
low	15.5	3.0	25.0	4.7	0.2	9.9



(a)



(b)

Figure 2.9: Example of high and low convexity polygons. (a) High convexity polygon; CV = 86.1%. (b) Low convexity polygon; CV = 26.2%.

2.7 Discussion

The benchmarks above indicate that Polylidar is the faster concave (multi)polygon extraction algorithm with the possibility of holes. This section discusses why Polylidar was faster in comparison to others. We specifically analyze the execution time of the major steps in Polylidar in comparison to other triangulation-based methods, namely CGAL and Spatialite. The three major steps are:

1. Triangulation - The point set is triangulated creating a mesh of faces, edges, and vertices.
2. Shape Extraction - Mesh simplices are removed based upon the α parameter or edge length. Remaining triangles, edges, and vertices represent the "shape".
3. Polygon Extraction - The "shape" is converted to a (multi)polygon with the possibility of holes.

Triangulation All perform Delaunay triangulation using robust geometric predicates but use different libraries to do so. Polylidar uses Delaunator, CGAL uses its own 2D triangulation, and Spatialite uses GEOS.

Shape Extraction Polylidar and Spatialite are most similar, focusing only on filtering triangles in the mesh. However Polylidar goes further with region growing (Section 2.4.3) that isolates disconnected regions in the mesh. For memory efficiency and speed we represent the filtered triangle set $\mathcal{T}_f$ as a bit array with 1/0 indicating in/out of set. This allows rapid triangle filtering and region growing which was previously profiled to be slower when using hashmaps. On the other hand CGAL first creates "interval hashmaps" for its simplices, including triangles, edges, and vertices. These hashmaps store data detailing at what α-interval a specific simplex would be in the α-complex. These ordered hashmaps give the ability to more quickly compute a *family* of α-shapes from a point set. These data structures are implemented as C++ multimaps with $\mathcal{O}(\log n)$ for insertion/look-up in comparison to unordered maps having $\mathcal{O}(1)$. This design choice leads to shape extraction having an $\mathcal{O}(n \log n)$ complexity for CGAL. By creating hash maps for edges and vertices CGAL can also return the *singular* points and edges which are isolated and not attached to any triangle in the α-complex (e.g., a single point far removed from all others). Polylidar need not do this because singular points and edges cannot be polygons thus are not required steps in shape extraction.

Polygon Extraction Polylidar independently converts each region into a polygon. Algorithm 2 quickly identifies all border edges and uses efficient unordered contiguous memory hashmaps to store this information in $\mathcal{BE}$ and PtE. The essence of Algorithms 3 and 4 are entirely border-edge based leading to a significant speed up compared to triangle based methods (i.e., perimeter vs. area). Spatialite uses GEOS to take the union of all unfiltered triangles to generate a valid multipolygon.

CGAL's Alpha Shape produces an unordered list of the boundary edges of the α-shape. However CGAL does not provide any explicit function to convert this list to a valid (multi)polygon.

Table 2.6: Algorithm timings - Mean of 30 runs in milliseconds

Algorithm	triangulation	shape extraction	polygon extraction	total
Polylidar	36.0	4.4	1.0	41.4
CGAL	44.5	154.0	–	198.5
Spatialite	234.2	135.3	10788.7	11158.1

Table 2.6 summarizes mean execution timings for each of the main steps for Polylidar, CGAL, and Spatialite. The 64,000 point set in the shape of California (with holes) is used, with each algorithm executed 30 times with the mean presented. Relative execution times with other point sets are similar. Delaunator in Polylidar triangulated this specific point set fastest with CGAL a close second. Polylidar achieves a more significant speed-up in shape extraction for which Polylidar is 35 and 32 times faster than CGAL and Spatialite, respectively. Also, Polylidar's polygon extraction is about four orders of magnitude faster than Spatialite whereas CGAL does not extract polygons. CGAL instead offers a general purpose α-shape construction routine to compute a family of shapes from different α-values.

2.8 Conclusion

This chapter has introduced Polylidar, an efficient 2D concave hull extraction algorithm which produces (multi)polygon output with holes. Comparison benchmarks of numerous test sets, similarly done in [37], show Polylidar is faster than competing approaches with comparable or better accuracy. Additionally we perform random polygon tests that confirm every polygon produced by Polylidar is valid. In future work we will extend Polylidar to operate directly on 3D point cloud data by performing both planar segmentation and polygon extraction. We will remove Polylidar's reliance on Delaunay triangulation when used with organized point clouds (e.g., range images) similar to [51]. Triangulation can be performed in $\mathcal{O}(n)$ time by exploiting the spatial relationship inherit in range images. The OpenMP library will be used to parallelize iteration independent loops such as triangle filtering. Additionally we will explore task-based parallelization by making use of the data independence between polygons, i.e., spawning polygon extraction tasks immediately after a plane is segmented [52].

CHAPTER 3

Polygons from 3D Data

3.1 Introduction

Flat surfaces are pervasive in engineered structures and also occur in natural terrain. For example, structures such as walls, floors, rooftops, and roadways are often flat or "flat-like". Similarly, home and office furnishings are typically composed of multiple flat surfaces. Sensors such as LiDAR and RGBD cameras generate dense 3D point clouds of these predominately flat surface environments. This observation has been exploited for tasks in localization and mapping [53], digital preservation with Photogrammetry and laser scanning [54, 55, 56], and point cloud registration [57]. Planar segmentation techniques are often used to group points together belonging to a flat surface [49, 50, 58]. However points clouds are dense incurring a high computational cost when used directly in higher level tasks. Planar point clouds can be converted to lower dimensional representations such as polygons. Polygons reduce map size, accelerate matching for localization [59], and support model reconstruction and object detection [47].

Planar points clouds may be converted to convex polygons [60]. Convex polygons are simple and efficient to generate but often do not represent the true shape of a point set. Non-convex polygons may be generated using techniques such as α-shapes but operate strictly on 2D data, requiring the projection of each 3D planar point cloud and expensive triangulation [51, 38]. Pixel-level boundary following of organized point clouds can be used to extract non-convex polygons but often only captures the exterior shell of the polygon [59]. These methods are not able to capture *interior* holes in a polygon representing the shape of obstacles on flat surfaces. Finally, speed is an important consideration for many of the applications mentioned previously. Parallel algorithms written for multi-core CPUs and GPUs should be used to reduce latency.

This chapter presents Polylidar3D, a non-convex polygon extraction algorithm which takes as input either unorganized 3D point clouds (e.g., airborne LiDAR point clouds), organized point clouds (e.g., range images), or user provided meshes. The non-convex polygons extracted represent flat surfaces in an environment, while interior holes represent obstacles on these surfaces. Figure 3.1 provides an overview of Polylidar3D's data input, front-end, back-end, and output. Currently

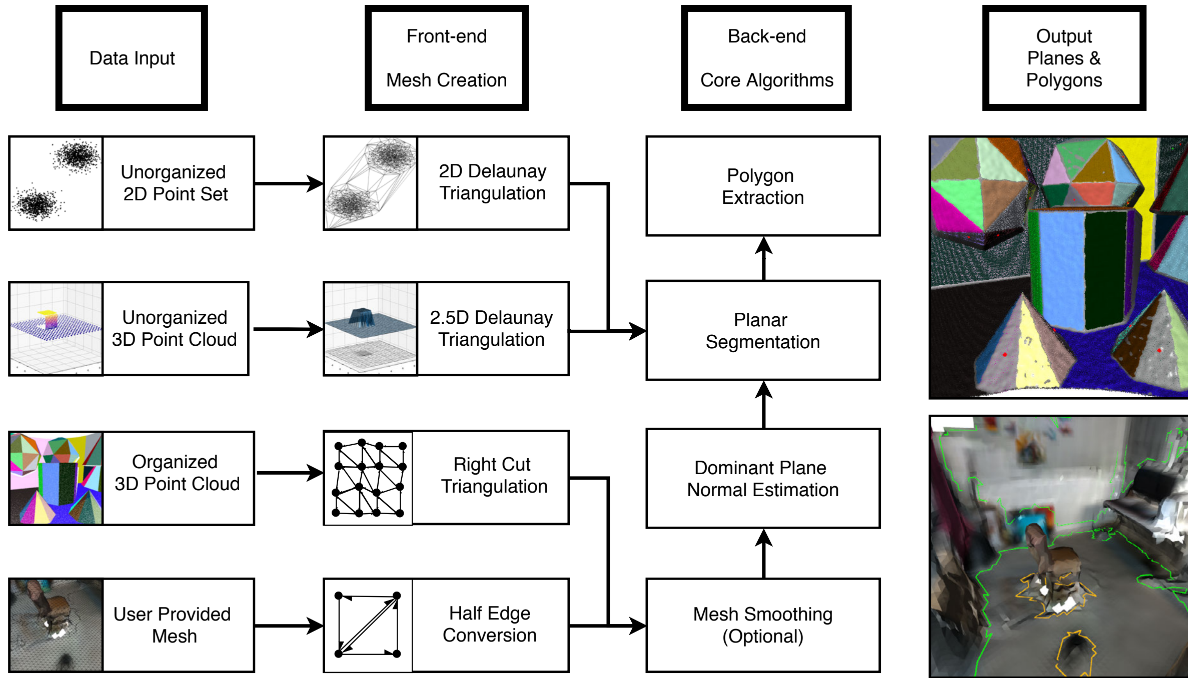


Figure 3.1: Overview of Polylidar3D framework. Input data can be 2D point sets, unorganized/organized 3D point clouds, or user-provided meshes. Polylidar3D's front-end transforms input data to a half-edge triangulation structure. The back-end is responsible for mesh smoothing, dominant plane normal estimation, planar segmentation, and polygon extraction. Polylidar3D outputs both planes (sets of spatially connected triangles) and corresponding polygonal representations. An example output of color-coded extracted planes from organized point clouds is shown (top right). An example of extracted polygons from a user-provided mesh is shown (bottom right). The green line represents the concave hull; orange lines show interior holes representing obstacles.

only one planar direction can be extracted from unorganized 3D point clouds while all other 3D data inputs do not have this limitation. The front-end transforms input data into a half-edge triangular mesh. This representation provides a common level of abstraction offering increased efficiency for back-end operations. The back-end is composed of four core algorithms: mesh smoothing, dominant plane normal estimation, planar segment extraction, and polygon extraction. Polylidar3D outputs planar triangular segments, sets of flat connected triangles, and their polygonal representations. Polylidar3D is extremely fast, typically executing in a few milliseconds. It makes use of CPU multi-threading and Graphics Processing Unit (GPU) acceleration when available. Polylidar3D is a substantial extension to Polylidar, the 2D algorithm presented above. The baseline Polylidar algorithm only operated on 2D point sets and offered no parallelism. The primary contributions of this chapter are:

- An efficient and versatile open source [36] framework for concave (multi)polygon extraction for 3D data. Input can be unorganized/organized 3D point clouds or user-provided meshes.

- A fast open source [61] dominant plane normal estimation procedure using a Gaussian Accumulator that can also be used as a stand-alone algorithm.
- Multiple diverse open source experiments showing qualitative and quantitative benchmark results from data sources including LiDAR and RGBD cameras [62, 63, 64].
- Improved half-edge triangulation efficiency for organized point clouds; CPU multi-threaded and GPU accelerated mesh smoothing [65].
- Planar segmentation and polygon extraction performed in tandem using task-based parallelism to reduce latency for time-critical applications.

Below, Sections 3.2 and 3.3 provide background and mathematical preliminaries, respectively. Section 3.4 describes Polylidar3D's front-end methods for mesh creation. Section 3.5 outlines optional mesh smoothing while Section 3.6 introduces our dominant plane normal estimation algorithm. Section 3.7 describes plane and polygon extraction with parallelization techniques. Section 3.8 proposes optional post-processing methods to refine and simplify the polygons. Section 3.9 provides qualitative results as well as quantitative benchmarks. Sections 3.10 and 3.11 provide discussion and conclusion, respectively.

3.2 Background

This section summarizes baseline methods on which Polylidar3D is constructed. Plane segmentation and polygon extraction background is followed by a description of 3D data denoising techniques such as mesh smoothing and plane normal estimation with Gaussian accumulators.

3.2.1 Planar Segmentation

Planar segmentation processes an input 3D point cloud and segments it into groups of points representing flat surfaces. These point groups are often informally called "planes" but differ from the geometric definition. A geometric plane is defined by a unit normal $\hat{n} \in \mathbb{R}^3$ and a single point on the plane $p \in \mathbb{R}^3$. Flat surface representation as a point set is advantageous because:

1. Point sets are naturally bounded (i.e., have finite extent). Bounded surfaces better correspond with most real-world flat surfaces.
2. Holes inside a plane may be represented implicitly by the absence of points. This representation can also indicate obstacles embedded in a flat surface.
3. Best-fit geometric planes can also be computed after segmentation using least-squares, principal component analysis (PCA), or RANSAC based methods [66].

4. Merging similar planes can be rapidly performed by combining their points sets.

Planar segmentation can be performed with region growing methods. Algorithms can exploit the spatial structure of an organized point cloud for which data is arranged into rows/columns like an image (e.g., range images). Region growing algorithms extract connected components in point clouds with neighborhood information (e.g., pixel neighbors or k nearest neighbors). A seed is chosen and assigned a unique label, then its neighbors are iteratively analyzed and assigned the seed's label if their characteristics are sufficiently similar to the seed's [66]. Characteristics such as normal orientation, color, or Euclidean distance from each other may be used. A unique label assigned to each point denotes a grouping in a planar surface.

Holz et al. [51] outlines planar segmentation by employing approximate polygonal meshing from organized point clouds. Mesh construction exploits the organized structure of a range image. Point normals are computed from the mesh and smoothed using bilateral filtering techniques which preserve edges. Region growing is performed sequentially until all possible points have been examined. Points are merged based on differences in normal angles and Euclidean distances. Reported benchmarks show that a 320 $\times$ 240 range image can be segmented in approximately 125 ms. Feng et al. [49] proposes the use of agglomerative hierarchical clustering (AHC) on organized point clouds to perform fast planar segmentation. The algorithm first creates a graph by uniformly dividing the points in image space. Initial node size (e.g., 4 $\times$ 4 pixel group) is user-configurable and allows a trade-off between execution speed and the detail of extracted planes. Nodes belonging to the same plane are merged through AHC until plane fitting error exceeds a user-defined threshold. A final refinement is done though pixel-wise region growing with possible plane merging. The algorithm is extremely fast; a 640 $\times$ 480 image with an initial node size of 10 $\times$ 10 can be segmented in $\approx$30 ms.

Schaefer et al. [58] details a probabilistic plane extraction (PPE) algorithm to detect planes in organized 3D laser range scans. The algorithm utilizes AHC with individual laser reflection in the scene to define an initial candidate plane set. Each plane is then iteratively merged with adjacent planes that maximize the measurement likelihood of the scan. Measurement likelihood is computed using a Gaussian probability density function modelling ray length. The algorithm is implemented in MATLAB with GPU acceleration but has execution times exceeding one hour for a 500 $\times$ 500 scan. Trevor et al. [67] similarly operates on organized point clouds and exploits neighbor information for merging. Surface normals are estimated for every point using methods in [51]. Points are merged if their normals and orthogonal distances are below a threshold which creates locally planar segments later refined through plane fitting and filtering out segments that exceed curvature constraints.

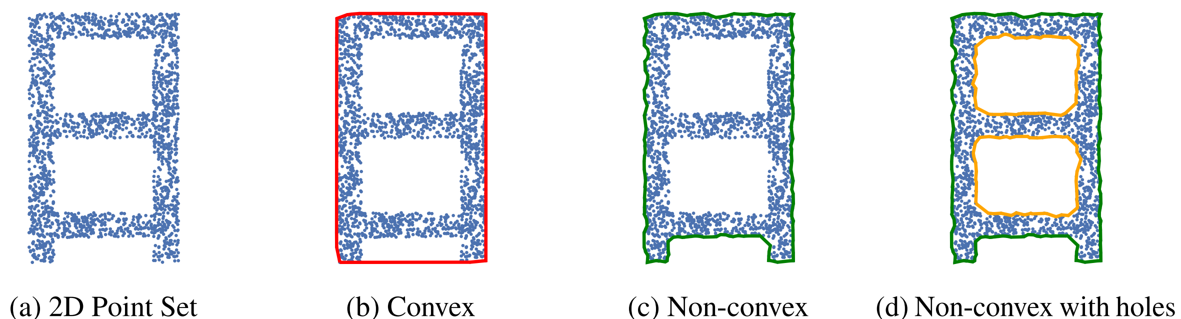
Salas-Moreno et al. [68] outlines a fast GPU accelerated planar segmentation method for use in simultaneous localization and mapping (SLAM) from range images. A range image is first

converted into surfels (surface elements) describing a point position and orientation. A surfel similarity bit mask is created and marked 1 if neighbor surfels to the left and above are similar in normals and plane distance, 0 otherwise. This bit mask is used to perform region growing where similar and contiguous pixels are merged to the same plane. The algorithm executes in real-time and can perform planar extraction with SLAM in 66 ms. Oesau et al. [69] presents a parallel plane extraction method specifically designed for unorganized point clouds. First, points are organized using an octree from which multiple seed points are uniformly selected. Region growing occurs in parallel for each seed point with points inside the same cell in the octree used for plane fitting. Region growing is periodically interrupted to perform regularization of the planar shapes detected. Regularization is carried out by merging planes captured on the same flat surface by refitting through PCA. The implementation is GPU accelerated and can segment point clouds with 1.1 million points in approximately three seconds.

Polylidar3D segments points clouds through region growing but operates on triangles instead of the points themselves. Region growing is regularized and parallelized by first identifying dominant plane normals in the mesh. Triangles having similar normals to a dominant plane are grouped. Each group (in parallel) then performs region growing accounting for normal orientation, Euclidean distance, and point to plane distance. Note this method relies upon the data to be properly denoised.

3.2.2 Polygonal Shape Extraction

Representing planar surfaces as point sets has the disadvantage of high memory and computational overhead. Dense planar point sets have redundant information about the underlying surface they represent. Polygonal representations of point sets removes redundant information. We consider convex polygons, non-convex polygons, and non-convex polygons with holes per Figure 3.2.



(a) 2D Point Set (b) Convex (c) Non-convex (d) Non-convex with holes

Figure 3.2: Example polygons that can be generated from plane segmented point clouds. (**a**) 2D point set representation of a floor diagram with interior offices; (**b**) Convex polygon; (**c**) non-convex polygon; (**d**) and non-convex polygon with holes. The exterior hull (green) and interior holes (orange) are indicated.

Biswas et al. [60] represents flat surfaces as convex polygons extracted from range images. First, points are randomly sampled in the image with nearby pixel neighbors used for fast RANSAC plane fitting. This returns numerous sparse point subsets which may be coplanar. The convex hull is computed for each of these subsets generating many convex polygons in the scene. Polygons belonging to the same surface are then merged with GPU accelerated correspondence matching. The sparse random sampling of the point cloud and efficient generation of convex hulls allows the algorithm to run in real-time (less than 2 ms). However, convex hulls ignore boundary concavities, overestimate the area of the enclosed point set, and do no not account for holes per Figure 3.2b. Poppinga et al. [70] outlines a method to convert plane segmented range images into convex polygons. Each plane segment is decomposed into a set of convex polygons. Each polygon is progressively built through scan-lines; a new polygon is generated when convexity constraints are not met. This allows a concave plane to be represented by multiple convex polygons. Lee et al. [59] generates non-convex polygons from range images. The range image is first planar-segmented using an eight-way flood fill algorithm. This involves region growing which accounts for the normal vector to each point. Each of the planar segments is then converted to a non-convex polygon. Exterior boundary pixels of a plane segment are sampled and neighboring samples connected to create a non-convex polygon. However interior holes in the plane segments are not explicitly captured as shown in Figure 3.2c. Trevor et al. [67] performs a similar polygon extraction procedure through boundary tracing of the exterior hull.

Non-convex polygons with holes may be generated through a variety of methods, many under the name of concave hulls [38, 39, 41]. Many of these methods strictly operate on 2D data, requiring the 3D planar point cloud segments be projected to the best fit geometric plane to produce 2D point sets. Holz et al. [51] proposes this technique and the use of α-shapes to extract such polygons [38]. In Chapter 2 we developed a faster open source polygon extraction algorithm, Polylidar, which extracts non-convex polygons with holes from 2D point sets. The point set is converted to a 2D mesh through Delaunay triangulation, and triangles are subsequently filtered by edge length creating the "shape" of the point set. This filtered mesh is then converted to a polygon through boundary following while accounting for holes. Benchmarks demonstrate that our algorithm is a minimum of four times faster than leading methods [71]. This chapter extends Polylidar to operate directly on 3D data, performing planar segmentation and polygon extraction in parallel. This integration allows Polylidar3D to skip expensive Delaunay triangulation previously required for organized point clouds as shown in Section 3.4.2. Planar segments represented as non-convex polygons with holes gives the following advantages:

1. Significantly reduced memory requirements, on the order of square root (perimeter vs area).
2. Faster computation of geometric values of interest, e.g., centroid, area, perimeter.

3. Ability to dilate, erode, and simplify polygons through computational geometry routines.

4. Holes inside a polygon account for gaps or obstacles on flat surfaces.

3.2.3 3D Data Denoising

This section discusses two methods to smooth a mesh. The Laplacian filter performs weighted averaging of nearby vertex neighbors to reduce noise [72]. Vertices are updated according to

$$v_o = v_i \cdot \frac{\lambda}{W} \sum_{j=1}^{N} w_j \cdot (v_i - v_j) \tag{3.1}$$

$$w_j = ||v_i - v_j||^{-1} \quad W = \sum_{j=1}^{N} w_j \tag{3.2}$$

where v_o, v_i, v_j, and N denote the output (smoothed) vertex, input vertex, neighboring vertex, and total number of neighbors, respectively. Weighting for each neighbor vertex w_j is the inverse of its Euclidean distance and is normalized with W. Parameter λ adjusts smoothing [0-1], though multiple iterations may be performed to increase smoothing. The Laplacian filter is not edge-preserving.

Zheng et al. [73] proposes a bilateral filtering technique on triangular meshes that is analogous to images. Filtering occurs in two stages: normal smoothing and vertex updating. The first stage performs local iterative normal filtering to smooth normals but preserves edges as given by:

$$n_o = K \sum_{j=1}^{N} W_c(||c_i - c_j||) \cdot W_s(||n_i - n_j||) \cdot n_j \tag{3.3}$$

$$W_c(||c_i - c_j||) = \exp(-||c_i - c_j||^2/2\sigma_c^2) \tag{3.4}$$

$$W_s(||n_i - n_j||) = \exp(-||n_i - n_j||^2/2\sigma_s^2) \tag{3.5}$$

$$K = 1/\sum_{j=1}^{N} W_c(||c_i - c_j||) \cdot W_s(||n_i - n_j||) \tag{3.6}$$

where n_o, n_i, c_i, n_j, c_j, and N denote smoothed triangle normal, input normal, centroid, neighbor normal, neighbor centroid, and number of neighbors, respectively. Triangle weights W_c and W_s exponentially decay based upon deviation from the triangle position and normal and parameters σ_c and σ_s. Sharp edges can be preserved. Holz et al. [51] performs a similar normal filtering technique but includes an optional intensity term for colored point clouds. A second stage updates vertices using a method proposed by Sun et al. [74] which executes weighted averaging of neighboring vertices using the newly smoothed normals. Note that smoothing may not be possible if a triangular mesh is so noisy that neighboring triangles have significantly different normals.

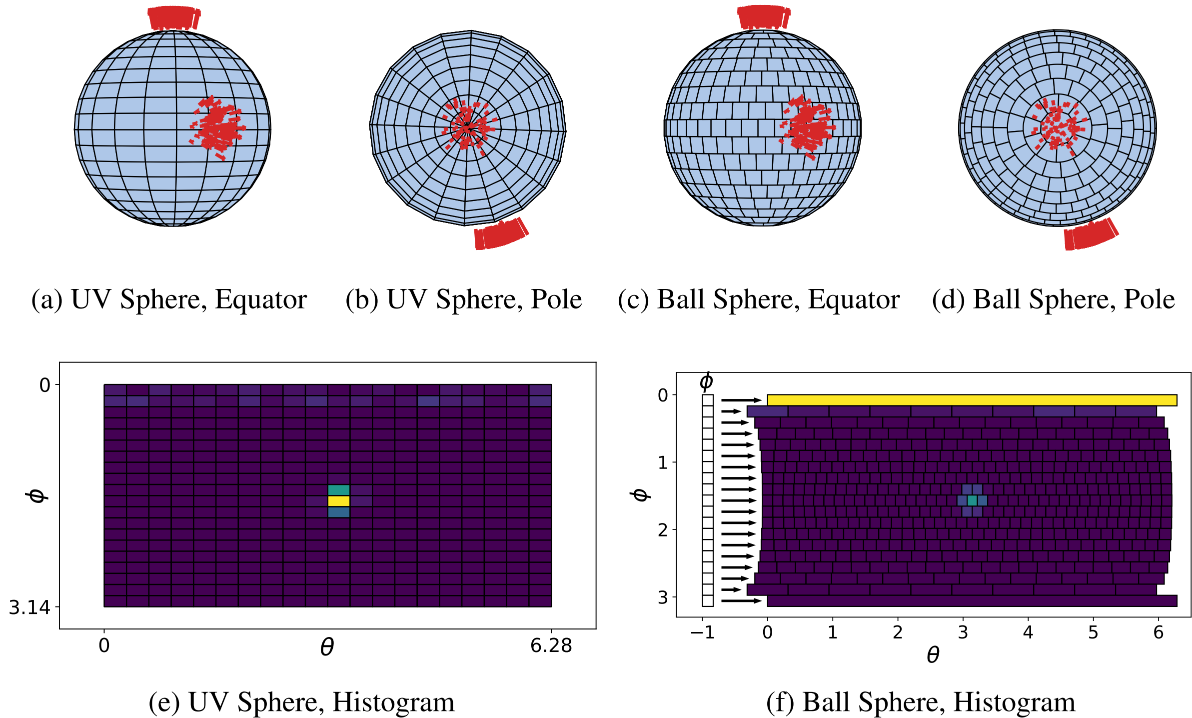
Both Laplacian and bilateral filtering rely upon neighboring triangles for smoothing. The neighbors are often limited to their 1-ring neighbors defined by vertex or edge neighbors. In Section 3.2.3 we provide accelerated implementations of these algorithms for use with organized point clouds. The organized structure allows an implicit triangular mesh to be defined (i.e., no data structures is needed to store the graph) with the ability to use arbitrary kernel sizes to expand the neighorhood graph, a necessary feature with dense noisy point clouds.

3.2.4 Dominant Plane Normal Estimation

The Gaussian accumulator (spherical histogram) is a widely used method for detecting planar surfaces [75]. It discretizes the surface of the unit sphere (S2) into individual cells, creating “bins” or “buckets” of a histogram. A “vote” for a possible plane, often in the form of unit normal and origin offset, are accumulated into this histogram. Peak detection strategies on the histogram can then find dominant plane normals. Many discretization strategies of S2 exist with tradeoffs in speed, memory requirements, and subsequent peak detection.

The UV Sphere discretization strategy (Figure 3.3a,b) decomposes S2 into a 2D array by polar coordinates θ and ϕ. Each dimension is discretized in equal steps creating a fixed number of cells, $n_\theta \times n_\phi$. A unit normal to be integrated, $\hat{n}_i \in \mathbb{R}^3$, is converted to polar coordinates $\theta_i \in [0, 360]$, $\phi_i \in [0, 180]$ to identify its corresponding histogram cell in the 2D array. Finding the 2D array index requires a simple operation, e.g., $\theta_{index} = \theta_i / 360^\circ \cdot n_\theta$. However UV Sphere cells have different shapes and area with very small cells at the poles resulting in three issues: unequal weighting (voting) during accumulation, singularities at the poles, and non-equivariant kernels for peak detection. Figure 3.3e shows an example UV sphere histogram that fails to detect a plane at the top (North) pole.

Borrmann et al. [75] and Limberger et al. [76] recommend adjusting azimuth step size based upon elevation angle leading to more uniform cell area. This creates a ‘Ball” Sphere with strips with a varying number of cells for each elevation angle stored as a list of lists (Figure 3.3c,d). Cell areas are similar but have different shapes; a substantially larger cap is placed at the poles. Limberger et al. [76] attempts to handle the singularity near the poles after peak detection through a vote weighting scheme. However any discretization strategy by polar coordinates will not have equivariant kernels during peak detection caused by anisotropic cells [77].



(a) UV Sphere, Equator (b) UV Sphere, Pole (c) Ball Sphere, Equator (d) Ball Sphere, Pole

(e) UV Sphere, Histogram (f) Ball Sphere, Histogram

Figure 3.3: Example Gaussian Accumulators. Two identically "noisy" planes (red arrows, Gaussian distributed) are integrated into a UV Sphere (**a**,**b**) and Ball Sphere (**c**,**d**). Note the anisotropic property of sphere cells caused by unequal area and shape; (**e**) The UV Sphere histogram is unable to detect the peak at the pole; (**f**) The Ball sphere is able to detect both peaks, but the north pole cell is significantly larger leading to an incorrectly higher value than the equator cell.

Toony et al. [78] proposes unit sphere tessellation into 1996 equilateral triangle cells. This approach gives near uniform cells in area and shape, resolving previous issues with unequal weighting, pole singularities, and non-equivariant kernels. The process of integrating a unit normal into the histogram is no longer an indexing scheme. They propose to use a K-D tree to spatially index each cell using its triangle normal. A nearest neighbor search must be conducted for every unit normal integrated into the histogram. Peak detection is not performed, instead the sorted histogram distribution is analyzed to predict the shape of the object being integrated (e.g., circle, plane, or torus).

In Polylidar3D we tessellate the unit sphere with triangles by recursively subdividing the primary faces of an icosahedron. The recursion level dictates the approximation of the unit sphere. Our search strategy does not rely upon K-D trees but instead uses a global index from space filling curves followed by local neighborhood search. We unfold the icosahedron into a 2D image in a particular way that guarantees equivariant kernels as outlined in [77]. Standard 2D image peak detection is performed with nearby peaks clustered using Agglomerative Hierarchical Clustering (AHC).

3.3 Preliminaries

A 3D point $\vec{p}$ is defined in a Cartesian reference frame by orthogonal bases $\hat{\mathbf{e}}_x$, $\hat{\mathbf{e}}_y$, and $\hat{\mathbf{e}}_z$:

$$\vec{p} = x\,\hat{\mathbf{e}}_x + y\,\hat{\mathbf{e}}_y + z\,\hat{\mathbf{e}}_z = [x, y, z] \tag{3.7}$$

An *unorganized* 3D point cloud is an arbitrarily ordered array of points denoted as $\mathcal{P} = \{\vec{p_0}, \vec{p_i}, \ldots, \vec{p}_{n-1}\}$ with an index $i \in [0, n-1]$. An *organized* 3D point cloud is structured with 2D indices $u \in [0, M-1], v \in [1, N-1]$ such that $\vec{p}_{u,v} = [\vec{x}_{u,v}, \vec{y}_{u,v}, \vec{z}_{u,v}]$. Neighboring 2D indices (u, v) and $(u+1, v+1)$ represent 3D proximity relationships between $p_{u,v}$ and $p_{u+1,v+1}$ when they lie on the same surface [49]. These 2D indices create an *image space* with M and N denoting the rows and columns. Note that the 2D indices can be collapsed to a 1D stacked array by $i = u \cdot N + v$. A triangular mesh $\mathcal{T}$ with k triangles is defined by

$$\mathcal{T} = \{t_0, t_i, \ldots, t_{k-1}\} \tag{3.8}$$

where each t_i is a triangle with vertices defined by three point indices $\{i_0, i_1, i_2\} \in [0, n-1]$ referencing points in $\mathcal{P}$. A half-edge triangulation further decomposes each triangle into three individual half-edges. Specifically each edge in the mesh is split into two oriented half-edges, often called twin or opposite edges [79]. Each half-edge is represented by a unique id he_j in triangle $t_i = \text{floor}(he_j/3)$. An ordered array $\mathcal{HE}$ is created to find corresponding twin edges. Specifically the twin edge of he_i can be found at index i in $\mathcal{HE}$. If no twin exists, i.e., the edge is on a border, then -1 is returned.

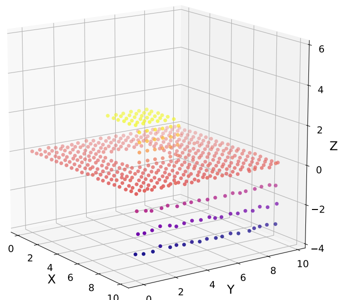
3.4 Mesh Creation

Polylidar3D requires a half-edge triangulated mesh to perform plane and polygon extraction. Mesh generation for unorganized and organized 3D points clouds is described below, followed by details on converting a user-provided triangular mesh to half-edge form.

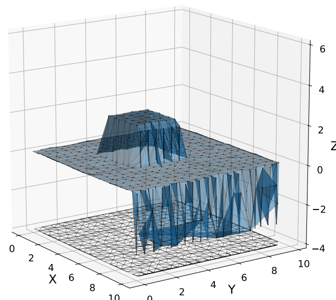
3.4.1 Unorganized 3D Point Clouds

We convert an unorganized 3D point cloud $\mathcal{P}$ into a 3D triangular mesh through 2.5D Delaunay triangulation [48]. $\mathcal{P}$ is projected to the xy plane, creating a corresponding 2D point set that is subsequently triangulated. Half-edge triangulation is provided by the Delaunator library with robust geometric predicates [44, 46]. Although triangulation is performed in 2D, both 2D and 3D point sets have 1:1 correspondence, allowing dual construction of the 3D mesh. Figure 3.4 demonstrates this technique applied to a synthetic rooftop scene with noisy point clouds from an overhead sensor.

The rooftop is captured at a slight angle providing points of one side of a building wall. Only planar segments roughly aligned with xy plane can be extracted with this technique, i.e., only the rooftop can be extracted, no walls. This type of conversion is most suitable for 3D points clouds generated from a top down viewpoint, such as airborne LiDAR point clouds as shown in Section 3.9.2.1. In this situation the plane normal to be extracted is already aligned with the xy plane.



(a) Unorganized 3D Point Cloud



(b) 2D and 3D Mesh

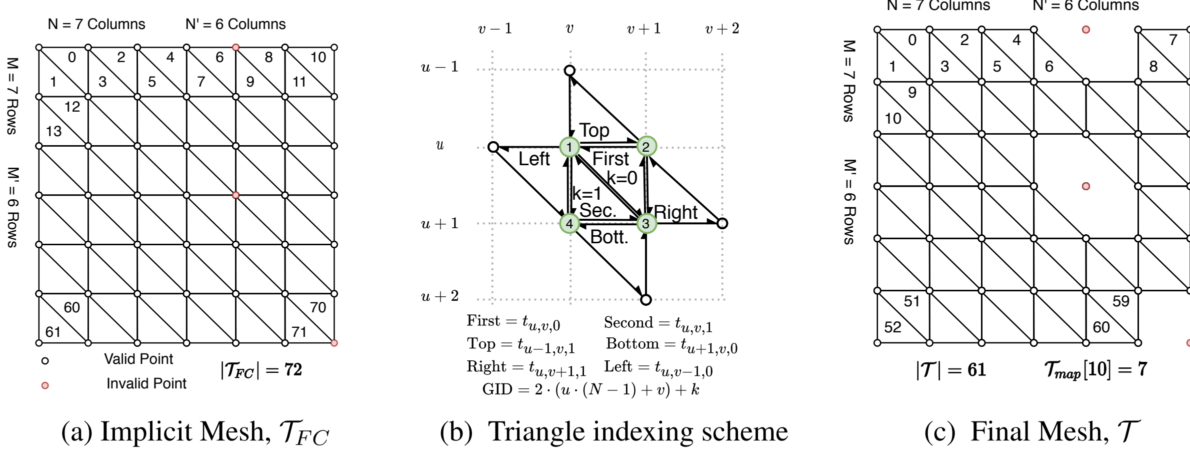
Figure 3.4: Converting an unorganized 3D point cloud to a 3D triangular mesh. (**a**) Synthetic point cloud of a rooftop scene generated from an overhead laser scanner. A single wall is captured because the scanner is slightly angled; (**b**) The point cloud is projected to the xy plane and triangulated, generating the dual 3D mesh. Only planes aligned with the xy plane can be captured.

Plane normals may not be aligned with the inertial xy plane, e.g., 3D laser scanner rigidly mounted on an automobile. The point cloud, generated in the sensor frame, must then be rotated such that desired plane to be extracted is aligned with xy plane. This requires a priori knowledge of the plane normal and rigid body transformation necessary to align the sensor frame point cloud as demonstrated in Section 3.9.2.2 where the ground plane (road) is extracted from point clouds generated by a spinning LiDAR sensor mounted on a car.

3.4.2 Organized 3D Point Clouds

Half-edge triangulation of an M $\times$ N organized point cloud can be quickly computed using spatial relationships from the image space. The triangulation is computed using neighboring image indices to define triangle vertex connections. This is in contrast to Delaunay triangulation which operates on the points themselves and maximizes the minimum angle of all the angles for each triangle in the triangulation [48]. Our procedure is similar to Holz et al. [51] except our method creates an explicit half-edge triangulation and only uses right-cut triangles. Our half-edge triangulation allows efficient triangle region growing which is a requirement for real-time polygon extraction. Holz et al. [51] performs adaptive meshing, switching between right and left

cut triangles, to better handle missing data at the expense of increased computational demand. Figure 3.5 demonstrates an example conversion of a 7×7 organized point cloud to a half-edge mesh using our procedure. The procedure creates the triangle set $\mathcal{T}$, half-edge array $\mathcal{HE}$, and a triangle map $\mathcal{T}_{map}$ as documented below.



(a) Implicit Mesh, $\mathcal{T}_{FC}$ (b) Triangle indexing scheme (c) Final Mesh, $\mathcal{T}$

Figure 3.5: Converting an organized point cloud into a 3D triangular mesh. Example conversion of a 7×7 Organized Point Cloud (OPC) to a half-edge triangular mesh. Points are represented by circles; red indicates an invalid value (e.g., 0 depth measurement). (**a**) Implicit mesh of OPC with right cut triangles, $\mathcal{T}_{FC}$. A unique global id $GID = 2 \cdot (u \cdot (N-1) + v) + k$ is shown inside each triangle; (**b**) Indexing scheme to define GIDs for triangles in $\mathcal{T}_{FC}$; (**c**) Final mesh $\mathcal{T}$ with triangles created if and only if all vertices are valid. Unique indices into $\mathcal{T}$ are marked. $\mathcal{T}_{map}$ maps between GIDs in $\mathcal{T}_{FC}$ to $\mathcal{T}$.

First, any invalid data in the point cloud is set to an NaN floating point value. This keeps the point cloud organized and prompts removal of invalid triangles in the mesh. An implicit fully-connected right-cut mesh triangulates all points, including invalid points marked red in Figure 3.5a and denoted $\mathcal{T}_{FC}$. Each 2×2 grid in the OPC creates two triangles, which we denote first and second as shown in Figure 3.5b. Each triangle in $\mathcal{T}_{FC}$ is indexed by $t_{u,v,k}$ where $k \in \{0, 1\}$ represents the first or second triangle, respectively. A unique global id $GID = 2 \cdot (u \cdot (N-1) + v) + k$ is shown inside each triangle in Figure 3.5a. This GID represents the triangle order within $\mathcal{T}_{FC}$. The final mesh returned $\mathcal{T}$ is shown in Figure 3.5c with construction outlined in Algorithm 3.1. The data structure $\mathcal{T}_{map}$ defines a mapping of global ids in $\mathcal{T}_{FC}$ to their index positions in $\mathcal{T}$ (if they exist, else -1) which is used later in half-edge extraction. Algorithm 3.1 begins by iterating over all 2×2 point grids and constructs the first and second triangle for each. These respective triangles are only added to $\mathcal{T}$ if all three points are valid. Triangle point indices are added counter-clockwise with the three half-edges implicitly defined by the ordered traversal of point indices, e.g., the first

triangle's half-edges are [PI3 $\rightarrow$ PI2, PI2 $\rightarrow$ PI1, PI1 $\rightarrow$ PI3].

Algorithm 3.1: Extract Triangles from OPC

```
Input  : Organized Point Cloud: 𝒫
         Rows: M, Columns: N
Output : Triangle Set: 𝒯
         Triangle Map: 𝒯_map
1  N' = N − 1, M' = M − 1
2  𝒯 = ∅
3  SV = −1 /* Sentinel Value indicating invalid triangle        */
4  𝒯_map = [SV, SV, ..., SV] /* |T_map| = 2 · M' · N'           */
5  n_tri = 0
6  for u ← 0 to M' − 1 do
7      for v ← 0 to N' − 1 do
8          First_GID = 2 · (u · N' + v), Second_GID = 2 · (u · N' + v) + 1
9          p1, p2, p3, p4 = GetPointIndices(u, v)
           /* First Triangle                                      */
10         if NotNan(p1, p2, p3, 𝒫):
11             𝒯 = 𝒯 + { p3, p2, p1 }
12             𝒯_map[First_GID] = n_tri
13             n_tri = n_tri + 1
           /* Second Triangle                                     */
14         if NotNan(p1, p3, p4, 𝒫):
15             𝒯 = 𝒯 + { p1, p4, p3 }
16             𝒯_map[Second_GID] = n_tri
17             n_tri = n_tri + 1
18     end
19 end
20 return 𝒯, 𝒯_map
```

The half-edge array $\mathcal{HE}$ is constructed using the previously calculated $\mathcal{T}_{map}$ and is shown in Algorithm 3.2. The algorithm begins at Line 3 by setting all half-edges in $\mathcal{HE}$ to the default sentinel value of -1 indicating no shared edge. Line 4 and 5 then begin iterating through every 2×2 grid in the OPC inspecting the first and second triangles in $\mathcal{T}_{FC}$. Line 6 and 7 retrieve the index of these triangles in $\mathcal{T}$ using $\mathcal{T}_{map}$ if they exist. If these triangles exist then their neighboring triangles may be assigned in Lines 9 and 18, respectively. For example in Line 13 if the right triangle neighbor exists then its first edge corresponding to a half-edge id of $3 \cdot Right_{idx}$ will be linked to the first

half-edge of the first triangle.

Algorithm 3.2: Extract Half-Edges from OPC

```
Input  : Triangle Map: T_map
         Rows: M, Columns: N
Output : Half-Edge Set: HE
1  N' = N − 1, M' = M − 1
2  SV = −1 /* Sentinel value indicating no shared edge              */
3  HE = [SV, SV, ..., SV] /*  |HE| = 3 · |T|                       */
4  for u ← 0 to M' − 1 do
5      for v ← 0 to N' − 1 do
6          First_idx = T_map[2 · (u · N' + v)]
7          Second_idx = T_map[2 · (u · N' + v) + 1]
8          Top_GID, Right_GID, Bottom_GID, Left_GID = GetNeighborsGID(u,v)
9          if First_idx != SV:
10             Top_idx = T_map[Top_GID]
11             Right_idx = T_map[Right_GID]
12             if Right_idx != SV:
13                 HE[First_idx · 3] = Right_idx · 3
14             if Top_idx != SV:
15                 HE[First_idx · 3 + 1] = Top_idx ·3 + 1
16             if Second_idx != SV:
17                 HE[First_idx · 3 + 2] = Second_idx ·3 + 2
18         if Second_idx != SV:
19             Bottom_idx = T_map[Bottom_GID]
20             Left_idx = T_map[Left_GID]
21             if Left_idx != SV:
22                 HE[Second_idx · 3] = Left_idx ·3
23             if Bottom_idx != SV:
24                 HE[Second_idx · 3 + 1] = Bottom_idx ·3 + 1
25             if First_idx != SV:
26                 HE[Second_idx · 3 + 2] = First_idx ·3 + 2
27     end
28 end
29 return T, T_map
```

3.4.3 User Provided Meshes

We define a user-provided triangle mesh as a triangle set $\mathcal{T}$ with a corresponding 3D point cloud $\mathcal{P}$. These meshes can be generated from 3D data using a variety of methods [80, 81, 82]. The front-end of Polylidar3D creates the half-edge set $\mathcal{HE}$ of this mesh to determine shared edges in similar manner to [83]. This entails first constructing half-edge hashmaps where the key is each half-edge's ordered point indices and value its half-edge ID. Opposite half-edges for any half-edge

can then be found by reversing the order of its point indices and performing a hashmap lookup. If a shared half-edge is found then its half-edge ID is mapped into $\mathcal{HE}$.

Certain forms of non-manifold meshes must be explicitly handled. We focus on a subclass of meshes that are not two-manifold. First we define two key properties of a two-manifold mesh:

1. Every vertex connects to a single edge-connected set of triangles.

2. Every edge is shared by one or two triangles.

Figure 3.6 shows examples of non-manifold meshes where condition (1) is violated. Polylidar3D handles violations of (1) using methods from our previous work [84]. The missing triangles (shown as white) are explicitly captured as holes inside a polygon for Figure 3.6a–c, while the mesh is split into two polygons for Figure 3.6d. Figure 3.7 shows cases of non-manifold meshes that violate condition (2). No mesh generated per Sections 3.4.1 and 3.4.2 will violate (2) because triangulation occurs in 2D space so all edges share at most two triangles. However a user-provided 3D mesh may not satisfy condition (2).

The half-edge array $\mathcal{HE}$ used for neighbor expansion during planar segment extraction in Section 3.7.1 only maps twin half-edges, making condition (2) mesh violations problematic. Three options can handle cases when more than two shared edges exist:

1. Store only the first pair of edges found and ignore any others.

2. Select the pair of edges that are most similar. Similarity between edges is defined by comparing angular distance of their owning triangle normals.

3. Ignore all of them by labelling all as boundary edges.

Option one is advantageous in speed and will generally have minimal consequences in the event an incorrect half-edge pairing is chosen, e.g., a green and orange triangle edge are linked in Figure 3.7. If green and orange triangle normals are sufficiently different then planar segment extraction will not connect them. However there is no guarantee that this may occur and may fail as in (Figure 3.7c). Option two attempts to remove the issue entirely by connecting only the pair of edges that are most similar (edges shared by green triangles). This technique will work for (Figure 3.7a,b) but will fail once again on (Figure 3.7c). Finally option three is the safest, it links none of the shared edges and treats them as border edges (edges sharing no neighbor). This keeps the critical invariant that no condition (2) violation will exist in an extracted planar mesh. However superfluous border edges will exist which can be handled downstream. Currently only option (1) is implemented in Polylidar3D [36] with future plans to allow the user to choose between any of the three proposed solutions.

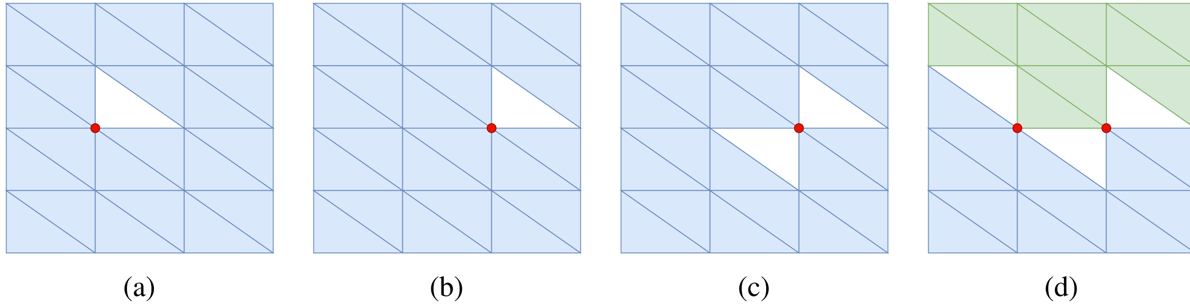
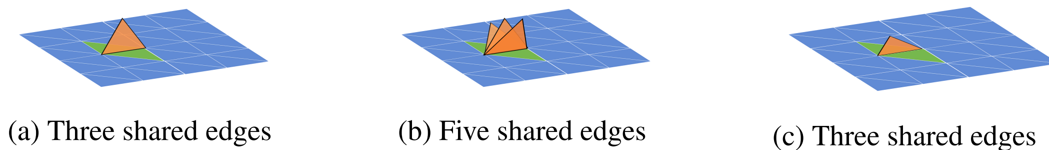


Figure 3.6: Example non-manifold meshes that exhibit condition (1) violations. Red vertices specifically show examples where the vertex triangle set is not fully edge-connected. (**a–c**) will extract any missing triangles as holes from the blue triangular segment; (**d**) The missing triangles in this mesh also cause condition (1) violations but will *not* be captured as holes. They cause the green and blue portion of the mesh to not be edge-connected for region growing in Section 3.7.1. They will be extracted as two separate segments with no holes in their respective interiors

.



(a) Three shared edges (b) Five shared edges (c) Three shared edges

Figure 3.7: Example non-manifold meshes with condition two violations. Edges are shared by more than two triangles. This common edge is shared by green and orange triangles. The green triangles form a two-manifold mesh with the blue triangles while the orange triangle(s) do not. The orange triangles in (Figure 3.7a,b) have sufficiently different normals such the green triangles half-edges can be easily linked. However all triangles in (Figure 3.7c) have nearly equal normals making this impossible.

3.5 Mesh Smoothing

Mesh smoothing for user-provided triangular meshes is performed using Intel Open3D smoothing procedures [83]. The sections below describe our implementation of Laplacian and bilateral filtering for the organized point cloud meshes created in Section 3.4.2. Our implementation is open source and provides single-threaded CPU, multi-threaded CPU, and GPU accelerated routines [65].

3.5.1 Laplacian Filter

We implement the standard Laplacian filter for organized point clouds with the benefit that no explicit triangular mesh is required, only the point cloud itself. The filtering, as described in Equation (3.1), results in smoothed vertices of the mesh, i.e, the point cloud is denoised. Ver-

tex neighborhood information is defined implicitly by the image space indices of the organized point cloud. The neighborhood size is configured by adjusting the kernel size of the filter, e.g, a kernel size of three implies eight vertex neighbors. Filtering this way offers the following benefits:

1. Neighboring vertices do not need to be found through lookup over $\mathcal{T}$, $\mathcal{HE}$, or an adjacency list.
2. Neighborhood size can be increased by adjusting filter kernel-size. Increasing the kernel size is critical for extremely dense and noisy point clouds.
3. Parallelization is trivial, similar to image filters, with all necessary neighborhood data for a vertex located close in memory.

The amount of filtering is controlled by λ, the kernel size, and the number of iterations. As kernel size and number of iterations increase the computational demand of the filter also increases. Mesh borders in image space have no defined neighbors on the exterior thus are not filtered. This gives a negative drawback of a noisy border but a positive benefit of reducing the mesh shrinkage inherit to Laplacian filtering. One may think of the fixed border as "pinning" the mesh to prevent overshrinkage.

3.5.2 Bilateral Filter

We implement the bilateral mesh filtering algorithm presented by Zheng et al. [73] but for organized point clouds. Smoothing occurs on the implicit fully-connected organized mesh $\mathcal{T}_{FC}$, described in Section 3.4.2. Recall the mesh spatial structure is defined through image indices (u, v) with a final index $k \in \{0, 1\}$ representing the first or second triangle in a 2×2 quad (see Figure 3.5b). Bilateral filtering per Equation (3.3) requires data structures for each triangle's centroid and normal, which we denote as $\mathcal{C}$ and $\mathcal{N}$. These are constructed in parallel (if multi-core CPU is available) and laid out in contiguous memory with the same indexing scheme as $\mathcal{T}_{FC}$, i.e., the centroid of triangle $t_{u,v,k}$ is $c_{u,v,k}$. If any of the vertices of a triangle in $\mathcal{T}_{FC}$ are NaN then the associated centroid and normal will also be NaN.

The algorithm partitions triangle smoothing in image space coordinates, smoothing both the first and second triangles as one unit of work. Each triangle's normal is updated using centroid and normal information from neighboring triangles. The neighbors of a triangle are determined by a user-configurable kernel size as shown in Figure 3.8. Note that defining neighbors in this way mixes both n-ring and $(n + 1)$-ring triangle neighbors. However the exponential decay of the bilateral filter in Equations 3.4 and 3.5 ensures that only triangles of similar properties (close in position and orientation) will be integrated into the smoothed normal. Neighbors are not integrated if they have

NaN values for their centroid/normal. The end result is smoothed normals for $\mathcal{T}_{FC}$; however, what is actually desired are smoothed normals for $\mathcal{T}$ per Figure 3.5c. This is quickly achieved by using $\mathcal{T}_{map}$ to identify the valid normals for $\mathcal{T}$. Zheng et al. [73] follows up with vertex updating, but we do not perform this step. Vertex updating is an expensive operation which provides minimal benefit for triangle region growing downstream. Only the smoothed normals are needed in Polylidar3D.

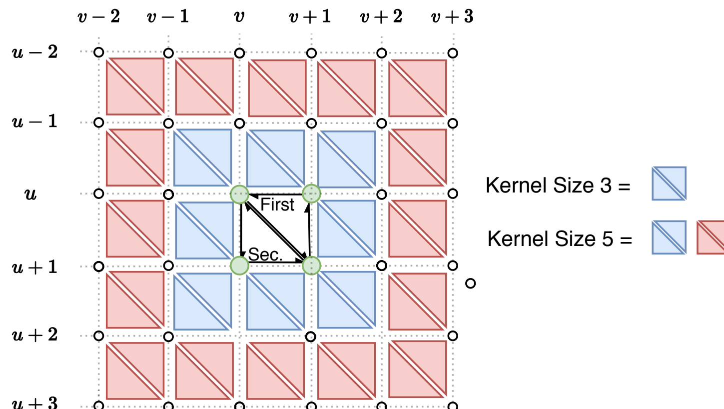


Figure 3.8: Visualization of a triangle's neighborhood during bilateral filtering. Each 2×2 point group forms two triangles creating a mesh. Each triangle's neighbors are defined by the kernel size. For example a kernel size of five includes all blue and red triangles.

The advantage of implementing bilateral smoothing in this manner is that all information is laid out in contiguous memory to increase cache locality. These are important characteristic for CPU and especially GPU performance. However nontrivial excess work is performed if most of the point cloud is invalid, e.g, invalid depth measurements in a range image. The entire procedure is controlled by σ_c^2, σ_s^2, kernel size, and number of iterations.

3.6 Dominant Plane Normal Estimation

We present a new method for constructing and using a Gaussian Accumulator to identify dominant plane normals in a scene. We call this method the Fast Gaussian Accumulator (FastGA). The input to this method is a list of k unit normals $\mathcal{N} = \{\hat{n}_0, \ldots, \hat{n}_{k-1}\}$ which have been sampled from a scene. Use of denoised data is advantageous but not required. Sections 3.6.1 and 3.6.2 discuss constructing the Gaussian Accumulator and performing peak detection, respectively.

3.6.1 Gaussian Accumulator

The following subsections describe the process to approximate a sphere using an icosahedron, construct the Gaussian Accumulator, and our method to integrate information into the accumulator.

3.6.1.1 Refined Icosahedron

A geodesic polyhedron is first constructed by using an icosahedron as the base model approximation of a unit sphere. The icosahedron is composed of 12 vertices and 20 faces and can be seen on the far left in Figure 3.9. This polyhedron is refined by recursively dividing each face into four equilateral triangles and then projecting the new vertices onto the surface of a sphere. The number of iterations or levels of recursion is user configurable with higher levels better approximating a sphere. This is a Class I geodesic polyhedron defined with the Schläfli symbol $\{3, 5+\}_{1,0}$ with frequency doubling at each level [85]. Figure 3.9 shows refinement up to level four while Table 3.1 displays the change in number of vertices, triangles, and approximate angular separation between each triangle. We denote each triangle as the cell or bucket of the histogram of S2. The number of cells, n, and their properties described below are fixed once a refinement level is chosen.

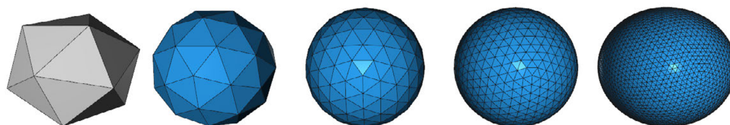


Figure 3.9: Approximation of the unit sphere with an icosahedron. The level 0 icosahedron is shown on the left with increasing refinements to the right. Triangle cells become buckets of a histogram on S2.

Table 3.1: Levels of refinement for an icosahedron.

Level	# Vertices	# Triangles	Separation
0	12	20	$41.8°$
1	42	80	$18.0°$
2	162	320	$6.9°$
3	642	1280	$3.1°$
4	2562	5120	$1.5°$

3.6.1.2 Gaussian Accumulator Properties

A space-filling curve (SFC) maps a multi-dimensional space into a one-dimensional space, e.g., $\mathcal{R}^2 \rightarrow \mathcal{R}$. Hilbert curves are a widely used SFC because they preserve locality well during transformation [86]. This means that points close in 1-D space are close in N-D, though the converse is not guaranteed to be true. In practice a SFC is approximated using discrete integers. The S2 Geometry library [87] provides a SFC routine that transforms any real-valued unit normal $\hat{n}_i \in \mathcal{R}^3$ to a 64 bit unsigned integer. The method works by projecting the unit sphere to a cube, creating 2D $\rightarrow$ 1D Hilbert curves for each of the six faces, and finally stitching them together to make one

unbroken linear chain. Each cell in the refined icosahedron has a surface normal $\hat{n}_i^c$ that can be mapped to a unique ID denoted $s2id$ using this procedure. This generates a one-dimensional thread that passes through every cell such that each cell is visited exactly once as seen in Figure 3.10.

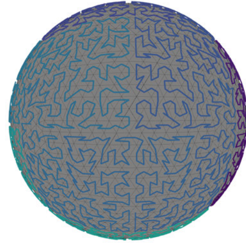


Figure 3.10: Space filling curve (SFC) of a level 4 icosahedron. Curve generated using the S2 Geometry Library. Each cell's surface normal is mapped to an integer creating a linear ordering for a curve. The curve is colored according to this mapping and traverses each cell.

The final Gaussian Accumulator (GA) is then an ordered array of $Cells = [c_i, \ldots, c_{n-1}]$. Each cell contains its surface normal $\hat{n}_i^c$, unique $s2id_i$, and an accumulating integer $count_i$. The cell array is sorted by $s2id$, creating the invariant that cells close together in the array are close in physical space. A neighborhood data structure $Nbrs_{i,j}$ is constructed as an N×12 matrix in which the ith row contains the 12 neighboring cell indices of the ith cell in the $Cells$ array. Neighboring triangles are defined as those in the 1-ring vertex adjacency. A maximum of 60 triangles at any level of refinement have only 11 neighbors; all others have 12. For these cells the 12th neighbor index is given a sentinel value of -1 to indicate no neighbor is present.

3.6.1.3 Integrating the Gaussian Accumulator With Search

Integrating a list of k unit normals $\mathcal{N} = \{\hat{n}_0, \ldots, \hat{n}_{k-1}\}$ into the Gaussian Accumulator is done through a search that finds the corresponding cell whose surface normal is closest to an input normal $\hat{n}_i$ then incrementing the cell's $count_i$ member. Instead of a K-D tree search we propose combining a sorted integer search with a local neighborhood search. Though similar, there are nontrivial differences and optimizations that make our method faster. The main components of the search are as follows: map $\hat{n}_i$ to an integer $s2id$, perform sorted integer interpolation search to reduce search bounds, perform branchless binary search within these bounds in the $Cells$ array, then perform local neighborhood search to find the correct cell. Algorithm 3.3 outlines this search routine and is explained below.

There exist several methods for sorted integer search such as interpolation and binary search. Interpolation search works by predicting the index of a value in a sorted array by interpolating between the first and last value of the sorted array (thus computing a slope). The process continues iteratively, each time reducing the search window and recomputing a new line for improved predic-

tion. Interpolation search is best used for linear data but still often underperforms in comparison to binary search in practice due to its use of repeated computationally expensive calculations of slope [88].

Algorithm 3.3: Find Cell Index

```
   Input  : Unit Normal: n̂_i ∈ R^3
            Cells Array: Cells
            Neighbor Matrix: Nbrs
   Output : Cell Index: k_best ∈ [0, |Cells|]
1  s2id = GetS2ID(n̂_i)
2  [k_min, k_max] = SearchWindow(s2id)
   /* get closest neighbor by s2id                          */
3  k' = BranchlessBinarySearch(s2id, Cells, k_min, k_max)
4  k_best = k'
5  dist_best = ||n̂^c_k' − n̂_i||
   /* local neighbor search by actual distance              */
6  for j ← 0 to 12 do
7  |  k_nbr = Nbrs_{k',j}
8  |  if k_nbr is −1:
9  |  |  continue
10 |  dist = ||n̂^c_{k_nbr} − n̂_i||
11 |  if dist < dist_best:
12 |  |  k_best = k_nbr
13 |  |  dist_best = dist
14 end
15 return k_best
```

Figure 3.11a shows a graph of cells in the Gaussian Accumulator where the x-axis is the $s2id$ and the y-axis is the corresponding index into the sorted $Cells$ array. We use least squares regression to fit a line to the data shown in Figure 3.11a, in contrast to only using the first and last values typical of interpolation search. Figure 3.11a shows this regressed line (green) accurately fits the data overall. In a zoomed plot (Figure 3.11b), model error, the difference between actual and predicted cell array index, is shown as the red line with values on the right vertical axis. Since the model/data domain and range are ordered and finite we can compute the negative and positive error bounds which is fixed once the GA refinement level is chosen. This is significant because one can reliably predict the correct index position of a cell with small known error bounds. This means that one does not need to perform a full binary search through the array of histogram cells but only a small subset of it. For example, refinement level four with 5120 histogram cells has maximal error bounds of -16 and +16 from any predicted position, thereby reducing the search from 5120 to 32 cells. This technique brings the benefits of linear interpolation search without excess computational overhead

because the model can be computed at compile time.

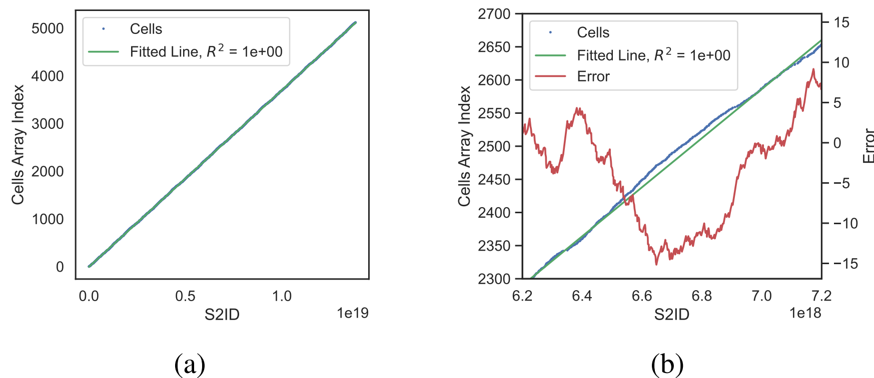


Figure 3.11: Linear prediction model for space filling curve indices on a Gaussian Accumulator. The 5120 GA cells are sorted in an array ($Cells$) by their corresponding spatial index $s2id$; (**a**) Plot relating cell $s2id$ and index position in the $Cells$ array with a regressed line (green) to the data; (**b**) Zoomed-in view showing model error (red line) indicating difference between predicted and actual index for each $s2id$.

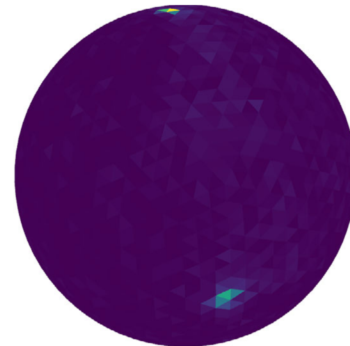
This predicted index and maximal bounds are used to create a binary search window in the $Cells$ array, shown at Line 2 in Algorithm 3.3. A branchless binary search is used which is faster than standard binary search for arrays of small sizes that fit into CPU L1/L2 caches [89]. All the search windows at realistic levels of refinement are sufficiently small to meet this criterion. The output of branchless binary search is an index k' into $Cells$ with $s2id$ closest to the mapped $s2id$ of $\hat{n}_i$ (Line 3). There is no guarantee this cell's surface normal $\hat{n}^c_{k'}$ is closest to $\hat{n}_i$ than neighboring cells though it is guaranteed to be close. Therefore a local neighbor search is performed where all 12 neighboring cells' surface normals are compared to $\hat{n}_i$. The cell index with closest surface normal is then returned.

3.6.2 Peak Detection

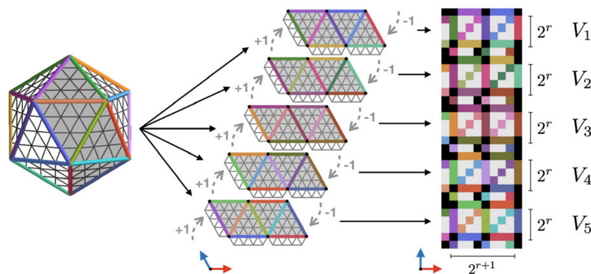
The histogram of the Gaussian Accumulator is normalized between the range [0–255]. Figure 3.12a shows an example mesh of a basement where the dominant planes are the floor and walls. Figure 3.12b shows a colored visualization of the GA after integrating triangle normals of this mesh. Higher values are bright yellow; lower values are dark purple. Peaks representing the basement floor and walls are clearly visible near the top and side of the sphere, respectively. Note that more peaks exist on unseen sides of the sphere. We use the technique described by Cohen et al. [77] to unwrap the refined icosahedron into a 2D image as shown in Figure 3.12c. The center image shows unwrapping of the icosahedron to create five charts. The vertices of these five charts map to hard-coded correspondences of pixels in the right image. This requires every vertex take the average value of its neighboring triangles. Finally a one-pixel padding is performed on the edges

of each chart by copying neighbors of adjoining charts. This creates duplicate pixel values on the bottom and left of the image as well as between charts. The end result is a 2D image guaranteed to provide equivariant convolution for kernels. The unwrapped image of the example GA is shown in the Figure 3.12d.

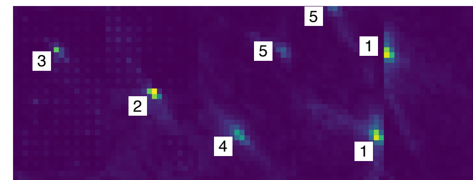
(a) 3D mesh of basement



(b) Colorized Gaussian Accumulator



(c) Unwrapping Process. Reprinted from [77]
Published open access under a CC-BY 4.0 license
http://proceedings.mlr.press/v97/cohen19d.html



(d) Unwrapped GA, rotated

Figure 3.12: Example using Fast Gaussian Accumulator on a triangular mesh. (**a**) Example basement scene mesh; (**b**) Mesh triangle normals are integrated into the Gaussian Accumulator and colorized showing peaks for the floor and walls; (**c**) Overview of unwrapping a refined icosahedron into a 2D image. Five overlapping charts are stitched together to create a grid. Padding between charts is accomplished by copying adjoining chart neighbors using the unwrapping process and its illustration from [77]; (**d**) Unwrapped Gaussian Accumulator creating a 2D image used for peak detection. White boxes indicate detected peaks. Duplicate peaks are merged (1 & 5) with agglomerative hierarchical clustering.

We use a standard 2D peak detector algorithm to find local peaks in the image. A peak is in the center of a 3X3 pixel group if it is the maximum in the group and its value is higher than a user-configurable v_{min}. Once a peak is detected in the 2D image it is converted to its corresponding surface normal on the GA. Duplicate peaks may be detected near chart borders because of copy padding discussed above, or two peaks may be close together. In either case it is desirable to

collapse them into a single peak. AHC is used to merge these peaks and take their weighted average. AHC will only merge peaks whose Euclidean distance is less than d_{peak}.

3.6.2.1 Application To Polylidar3D

Polylidar3D uses the Fast Gaussian Accumulator (FastGA) to estimate dominant plane normals. Triangle normals from the half-edge triangular mesh are input to FastGA. Not all triangle normals are needed to achieve acceptable results, so a user-configurable percent sampling parameter $sample_{pct}$ is used to reduce computational demand. After peak detection the l unique dominant plane normals are returned as a list $\mathcal{N}^d = \{\hat{n}^d_0, \ldots, \hat{n}^d_{l-1}\}$ for plane and polygon extraction. Note that alternative strategies of generating input normals such as fast RANSAC plane fitting with weighted voting may also be used [76].

3.7 Planar Segmentation and Polygon Extraction

The following sections build upon our previous work in polygon extraction from 2D triangular meshes. Section 3.7.1 describes planar segmentation while Section 3.7.2 outlines polygon extraction.

3.7.1 Planar Segmentation

The main input for planar segmentation is the half-edge triangular mesh, composed of $\mathcal{P}, \mathcal{T}, \mathcal{N}$, $\mathcal{HE}$, and the set of l dominant plane normals $\mathcal{N}^d$. Polylidar3D performs parallelized and regularized triangle mesh region growing via partitioning with dominant plane normals. Triangles having similar normals to a dominant plane are grouped for region growing. Different groups are grown in parallel. This process is controlled through user-provided parameters including maximum triangle edge length l_{max}, minimum angular similarity ang_{min}, maximum point to plane distance ptp_{max}, minimum number of triangles tri_{min}, and minimum number of vertices in a hole $vertices^{hole}_{min}$. These parameters limit the maximum distance between points for spatial connectivity, ensure common normal orientation in planar segments, force planar constraints, and remove spurious/small planes and holes. Note that ang_{min} is computed from the dot product between a triangle normal and its closest dominant plane normal; a value of 1.0 requires exact alignment while a value of 0.96 allows a $\approx 14°$ difference.

We first create triangle group array $\mathcal{G}$ to store group labels for each of the k triangles in $\mathcal{T}$. Algorithm 3.4 outlines this procedure and begins with iterating through all triangles (Line 4). $\mathcal{G}$ is composed of 8-bit unsigned integers [0–255] with 255 being a reserved sentinel value indicating a triangle does not belong to any planar segment. The following steps filter unused triangles and cluster triangles by normal orientation. The first geometric predicate (Line 5) removes triangles whose edge length exceeds a user-specified value. Lines 9–15 iterate though all dominant plane

Algorithm 3.4: Group Assignment

```
    Input  : Triangle Set: 𝒯, Point Cloud: 𝒫, Triangle Normals: 𝒩
             Dominate Plane Normals: 𝒩^d, Max Length: l_max, Min Angular Similarity:
    ang_min
    Output : Triangle Group Set: 𝒢
 1  SentinelValue = 255
 2  k = |𝒯|
 3  l = |𝒩^d|
    /* Loop through every triangle                                      */
 4  for t ← 0 to k do
 5  |  edge_length = GetMaximumTriangleEdgeLength(t, 𝒯, 𝒫)
 6  |  if edge_length ¿ l_max:
 7  |  |  𝒢[t] = SentinelValue
 8  |  |  continue
 9  |  max_similarity = -1.0
    |  /* Loop through every dominant plane normal                      */
10  |  for j ← 0 to l do
11  |  |  similarity = n̂_t · n̂^d_j
12  |  |  if similarity ¿ max_similarity:
13  |  |  |  𝒢[t] = j
14  |  |  |  max_similarity = similarity
15  |  end
16  |  if max_similarity ¡ ang_min:
17  |  |  𝒢[t] = SentinelValue
18  end
19  return 𝒢
```

normals finding the one most similar to the triangle's surface normal $\hat{n}_t$. Line 16 performs a check to ensure the triangle normal is within an angular tolerance of its nearest dominant plane normal. If a triangle is assigned the group 255 it will not participate in subsequent region growing. Using 8-bit integers limits the maximum number of dominant plane normals extracted to 254. This procedure is iteration-independent and is parallelized by OpenMP [90]. Figure 3.13a,b show an example input mesh and color-coded group assignments, respectively. In this example the floor (blue) and the wall (red) are the two dominant plane normals to be extracted. Note that the seat of the chair is assigned the same group label as the floor, and that superfluous triangles are also assigned in the top left of Figure 3.13b.

Region growing is decomposed using *task-based* parallelism, where l dominant plane normals create l separate tasks of regions growing. These tasks are executed in parallel by a threadpool and can themselves spawn additional dynamic tasks [52]. Each independent task performs a serial region growing procedure that is similar to our previous work on 2D meshes [84] and was inspired

by [47]. Algorithm 3.5 outlines this procedure for a single group g. The routine begins by creating empty sets to store planar triangular segments and their corresponding polygonal representations, denoted $\mathcal{T}^g$ and $\mathcal{PL}^g$. An iterative plane extraction procedure begins with a seed triangle t verified to belong to group g (Line 5). Subroutine `ExtractPlanarSegment` uses the seed triangle to create edge-connected triangular subsets from $\mathcal{T}$ which have the same group label in $\mathcal{G}$ and meet user-provided planarity constraints (Line 7). If a user-specified minimum number of triangles is met then this set, $\mathcal{T}_i^g$, is added to $\mathcal{T}^g$. A dynamic task is then created to perform polygon extraction for this segment (Line 10). This procedure call is non-blocking; the region growing task continues to extract any remaining spatially connected planar segments before terminating. This means planar segmentation and polygon extraction may occur in parallel if multi-core is enabled.

Algorithm 3.5: Region Growing Task

Input : Triangle Set: $\mathcal{T}$, Point Cloud: $\mathcal{P}$, Half Edge Set: $\mathcal{HE}$, Triangle Group Set: $\mathcal{G}$
Dominate Plane Normal: n^d, Dominate Plane Label: g
Point To Plane: ptp_{max}, Min Triangles: tri_{min}, Min Hole Vertices: $vertices_{min}^{hole}$

Output : Planar Segment Set: $\mathcal{T}^g$, Polygon Set: $\mathcal{PL}^g$

1 $\mathcal{T}^g = \emptyset$
2 $\mathcal{PL}^g = \emptyset$
3 $k = |\mathcal{T}|$
/* Loop through every triangle */
4 **for** $t \leftarrow 0$ **to** k **do**
5 | **if** $\mathcal{G}[t] \neq g$**:**
6 | | **continue**
7 | $\mathcal{T}_i^g = \text{ExtractPlanarSegment}(t, \mathcal{T}, \mathcal{P}, \mathcal{HE}, \mathcal{G}, n^d, ptp_{max})$
8 | **if** —$\mathcal{T}_i^g| > tri_{min}$**:**
9 | | $\mathcal{T}^g = \mathcal{T}^g + \mathcal{T}_i^g$
10 | | $\mathcal{PL}^g = \mathcal{PL}^g + \text{SpawnTask}(\texttt{PolygonExtraction}, \mathcal{T}_i^g, vertices_{min}^{hole})$
11 **end**
12 WaitForTasks
13 return $\mathcal{T}^g, \mathcal{PL}^g$

Figure 3.13c shows three planar segments extracted that represent the floor, chair seat, and wall. The floor and chair surfaces have similar surface normals but are not spatially connected so independent planar segments and corresponding polygons are created. The small bump on the floor did not meet the planarity constraints (configured with ptp_{max}) thus is not included in the floor planar segment. This hole in the mesh will be extracted as an explicit interior hole of a polygon. The wall surface belongs to a separate group and is extracted in parallel with the floor and chair.

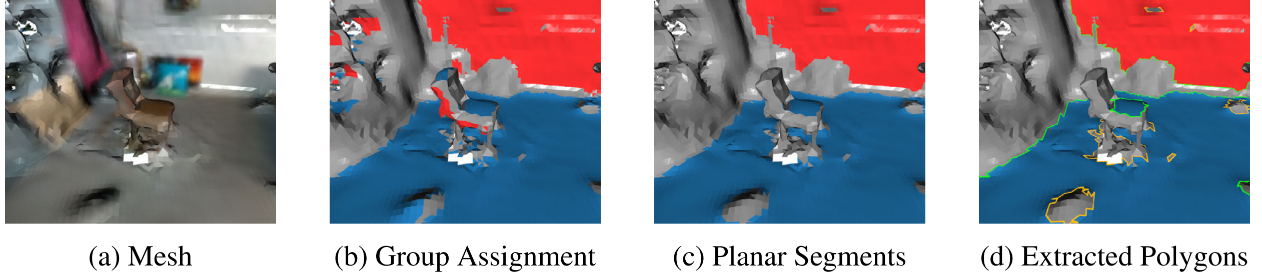

(a) Mesh (b) Group Assignment (c) Planar Segments (d) Extracted Polygons

Figure 3.13: Demonstration of Polylidar3D extracting planes and their polygonal representations. (**a**) An example mesh to demonstrate planar segmentation and polygon extraction using two dominant plane normals, represented by the floor and wall;(**b**) Every triangle is inspected for filtering and clustered through group assignment. Blue and red triangles meet triangle edge length constraints and are within an angular tolerance of the floor and wall surface normals, respectively; (**c**) Region growing is performed in parallel for the blue and red triangles. The top chair surface and floor are distinct planar segments; (**d**) Polygonal representations for each planar segment are shown. The green line represents the concave hull; the orange line depicts any interior holes. Note that small segments and small interior holes are filtered.

3.7.2 Polygon Extraction

Polygon extraction is performed on each planar mesh segment $\mathcal{T}_i^g$. Each polygon is defined by a single linear ring of points representing the concave hull/shell and a (possibly empty) set of linear rings representing interior holes. The same boundary following method we proposed in our previous work [84] is used with small modifications because triangular meshes are no longer 2D. Polygons are defined in a 2D subspace and are provided explicit guarantees through their definition per [43]. For example the edges in linear rings must not cross in this 2D space. For this reason boundary following in polygon extraction is carried out in the 2D projection of $\mathcal{T}_i^g$ on its geometric plane. Note that only the boundary edges of $\mathcal{T}_i^g$ need to be projected. Figure 3.14 shows the projection of $\mathcal{T}_i^g$ to its geometric plane and extraction of its polygonal representation. The three main components of polygon extraction are:

1. Data Structure Initialization
2. Extract Exterior Hull/Shell
3. Extract Interior Holes

The data structure initialization identifies all boundary half-edges inside $\mathcal{T}_i^g$ which are highlighted in purple in Figure 3.14b and denoted $\mathcal{BE}$. Additionally a mapping between point indices and these boundary half-edges are created denoted PtE. Finally any point on the exterior on the shell is found denoted pi_{xp}. The outer exterior shell is then extracting beginning with pi_{xp}. Boundary

following is performed by progressively building a linear ring by following each points outgoing half-edge(s) using PtE. Special routines handle scenarios when a hole is connected to the exterior hull. After the hull is extracted any interior holes remaining are extracted with the same special routines to handle rare scenarios when holes are connected.

The only modification to our previous work occurs in projecting the boundary edges. This is needed in finding pi_{xp} and for the special routines in handling multiple outgoing edges during boundary following. Note that the final polygons returned are represented as point indices in $\mathcal{P}$. The underlying 3D structure of the polygon is retained, i.e., it will follow a noisy surface per Figure 3.13d. The polygon can also be projected to the surface's geometric plane as described in Section 3.8 for post-processing.

Given noisy and dense planar mesh segments, border edges may cross during projection to the geometric plane. When this occurs an invalid polygon will be generated, most often a small self-intersection. This issue does not occur with unorganized point clouds because they are projected to the x-y plane where triangulation has already taken place; this guarantees edges do not cross. However planar segments from user-provided meshes and organized point clouds may be projected to arbitrary geometric planes. Additionally the tolerance in "flatness" of the planar triangular segment is user-configurable. This issue, if it occurs, is managed in polygon post-processing as described in Section 3.8. Although rare, if this condition must be handled before post-processing one might instead project all vertices of $\mathcal{T}_i^g$ to the geometric plane and perform polygon extraction on the 2D point set as shown in our previous work [84].

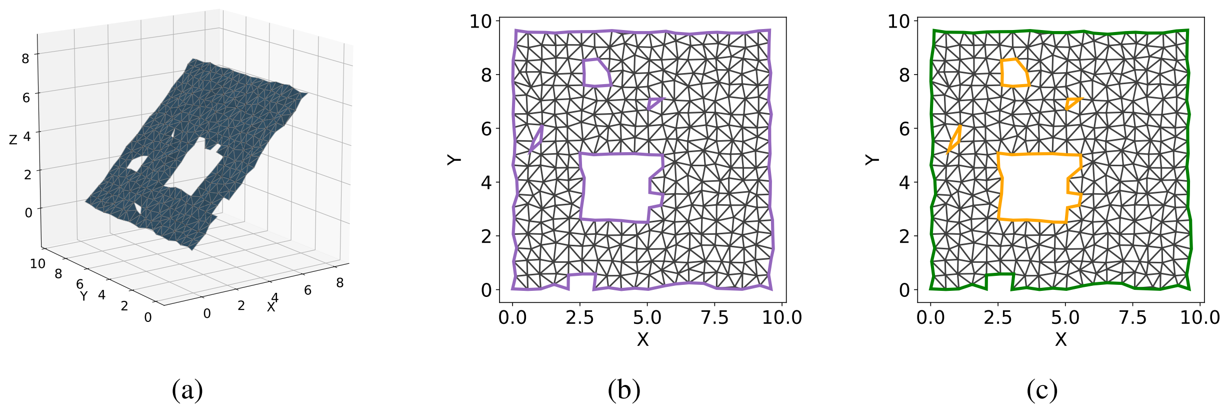


Figure 3.14: Example polygon extraction from a planar triangular segment. (**a**) Planar triangular segment $\mathcal{T}_i^g$. Note the four holes in the mesh; (**b**) Projection of a triangle segment to a geometric plane. Only border edges (purple) are actually needed for projection; (**c**) A polygon is extracted from border edges with a concave hull (green) and multiple interior holes (orange).

3.7.3 Algorithm Parallelization

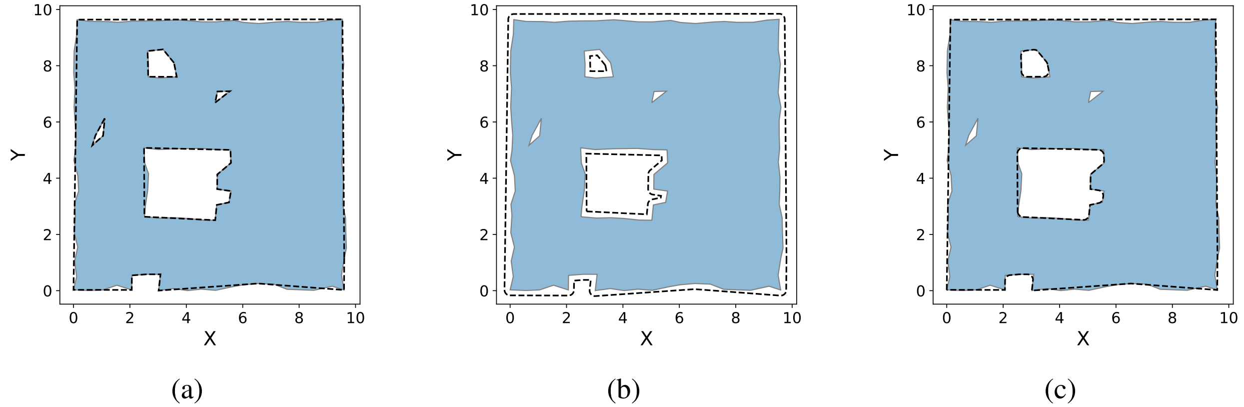
Planar segmentation and polygon extraction use both data and task-based parallelism. We use `OpenMP` for data parallelism which is carried out in "hot" loops that are iteration independent, e.g., triangle group assignment in Algorithm 3.4 as well as computing triangle normals. We use the `MARL` library to handle task scheduling and synchronization primitives [91]. Note that region growing of a single dominant plane normal is still a serial process as is the polygon extraction process of a single planar triangular segment. Therefore if only one dominant plane normal exists than task-based parallelism will provide minimal speed up. However group assignment is still fully parallelized. Benefits of parallelism are further explored in Section 3.9.4 experiments where speedup is calculated as number of threads and number of dominant plane normals vary.

3.8 Post Processing

The polygons returned by Polylidar3D can be further processed to improve visualization and filter superfluous polygons and/or holes. All operations are implemented on the 2D projection of the polygon on its geometric plane. The following sequential operations are executed:

1. Polygon is simplified with parameter α meters
2. Polygon is buffered outward by parameter β_{pos} meters
3. Polygon is buffered inward by parameter β_{neg} meters
4. Polygon is removed if its area is less than γ meters
5. Interior holes are removed if area is less than δ meters

The simplification algorithm is used to remove redundant vertices and "smooth" the polygon [92]. The α parameter indicates the maximum distance between any point in the new polygon from the original. This reduction of superfluous vertices also decreases the computational demand for subsequent buffering. The buffering process is defined as the Minkowski difference of the polygon with a circle of radius equal to a buffer distance β [93]. A positive buffer will expand a polygon and may fill in holes, while a negative buffer enlarges holes and recedes the concave hull. A positive buffer will fix any small self-intersections that may have occurred during the projection. Small polygons and/or interior holes are then filtered by area. Currently all of these steps are single threaded and handled in Python using the geometry processing library Shapely which binds to the C++ GEOS library [94]. An example of the first three steps of this process are shown in Figure 3.15.



(a) (b) (c)

Figure 3.15: Example of polygon post processing. The shaded blue polygon is the original polygon extracted (see Figure 3.14c). All dashed lines indicate a new polygon generated from a step of post processing. (**a**) The polygon is simplified; (**b**) The polygon is applied a positive buffer. Two small holes have been “filled” in; (**c**) The polygon is applied a negative buffer; only two holes remain.

3.9 Results

We present several examples of our methods applied to real-world and synthetic 3D data. Section 3.9.1 provides execution time benchmarks evaluating the speed of our proposed Fast Gaussian Accumulator. Section 3.9.2 shows examples of Polylidar3D applied to unorganized 3D points including airborne LiDAR point clouds and point clouds generated on a moving vehicle. Section 3.9.3 shows Polylidar3D applied to organized point clouds including RGBD cameras as well as a challenging synthetic benchmark set. Finally Section 3.9.4 presents Polylidar3D applied to 3D meshes and explores how polygon extraction scales with additional CPU cores.

All experiments/benchmarks use the same consumer desktop computer. The CPU is an AMD Ryzen 3900X 12 Core CPU with a frequency at 4.2 GHz equipped with 32 GB of RAM. Note that all results obtained with CPU parallelization are annotated with number of threads used; the default is four. An NVIDIA GeForce RTX 2070 Super is used for GPU acceleration.

3.9.1 Dominant Plane Normal Estimation

This section evaluates the Fast Gaussian Accumulator (FastGA) proposed for dominant plane normal estimation. We specifically analyze CPU execution time needed to integrate a set of k unit normals into the accumulator with k varied over the tests. Per Section 3.6.1.2 we use sorted integer search coupled with local neighborhood search instead of K-D trees [78]. To allow comparison, we created an alternative K-D tree Gaussian Accumulator implementation that uses nanoflann, a high performance C++ K-D tree library [95]. A leaf size of eight is used which offers the best results for our test cases. Results were generated on two test sets with hundreds of runs to provide

statistically significant results [96]. All benchmark code is open source [61].

A GA with refinement level four (5120 triangle cells) was used for all tests. The first test generated 100,000 randomly distributed surface normals on the unit sphere and integrated them into the GA. The second test integrated all 60,620 triangle normals from the basement mesh previously shown in Figure 3.12a. Recall that the GA is fixed once refinement level is chosen, so building the K-D tree index is not part of execution timing. Results of integrating all k normals for each test set are shown in Table 3.2. FastGA is more than two times faster than using a K-D tree, though the K-D tree implementation is also fast and could be used as an alternative method if desired. We can conclude that exploiting the known fixed structure of triangular cells on S2 (using space filling curves and sorted integer search) outperforms a general purpose K-D tree method.

Table 3.2: Execution time comparisons for synthetic and real world datasets

Algorithm	**Mean (ms)**	**Std (ms)**
a Synthetic: 100,000 Random Normals		
K-D tree	20.0	0.1
FastGA (ours)	9.1	0.1
b Real World: 60,620 Normals		
K-D tree	9.7	0.2
FastGA (ours)	4.4	0.1

Peak detection is currently implemented in Python using the `scikit-image` image processing library [97]. Agglomerative hierarchical clustering (AHC) of any detected peaks is implemented in Python with the `scipy` library [98]. The unwrapped 2D image of the icosahedron does not depend on the number of integrated normals but only the refinement level of the GA. Generated images are rather small (e.g., 90×34 pixels for a level four GA) resulting in very fast peak detection and clustering, e.g., it takes approximately 1 ms to detect peaks and perform AHC on a level four refined GA. FastGA results from additional datasets are shown below.

3.9.2 Unorganized 3D Point Clouds

Sections 3.9.2.1 and 3.9.2.2 describe results from Polylidar3D applied to airborne LiDAR point clouds and point clouds generated on a moving vehicle, respectively. Both datasets offer real-world unorganized 3D point cloud evaluation of Polylidar3D.

3.9.2.1 Rooftop Detection

This section presents qualitative results of Polylidar3D extracting flat rooftops in cities from airborne LiDAR point clouds of buildings. These unorganized point clouds are captured from an overhead viewpoint but are typically angled based on sensor location; wall surfaces can therefore be visible. Point cloud xy components are in a planar projected coordinate system while the z component represents elevation. Therefore points of a flat rooftop surface are already aligned with the xy plane making them suitable for 2.5D Delaunay triangulation to create a half-edge triangular mesh. The mesh is smoothed with Laplacian and bilateral filtering using Open3D [83]. Flat surfaces are then extracted as polygons and any non-flat obstacles on them become holes. Figure 3.16 shows the extracted polygons of buildings in Witten, Germany. Satellite imagery is overlaid with colorized point clouds. Each flat surface is extracted as a polygon with holes. All parameters used for this dataset are shown in Table 3.3.

Table 3.3: Polylidar3D parameters for rooftop detection.

Algorithm	Parameters
Laplacian Filter	λ = 1.0, iterations = 2
Bilateral Filter	σ_l = 0.1, σ_a = 0.1 , iterations = 2
Plane/Poly Extr.	tri_{min} = 200, ang_{min} = 0.94, l_{max} = 0.9, $vertices^{hole}_{min}$ = 8, $ptp_{max} = 0.20$
Poly. Filtering	$\alpha = 0.1$, β_{neg} = 0.1, β_{pos} = 0.00, γ = 16, δ = 0.5

Figure 3.16a shows a single building with two flat surfaces identifiable from the overlaid blue and purple points representing higher and lower elevation respectively. Polylidar3D successfully separates both of these flat surfaces as two polygons. Rooftop obstacles such as air vents and A/C units are captured as holes. Figure 3.16b,c images show additional examples of obstacle detection and surface separation, respectively. The large building on the left in (Figure 3.16d) hosts a structure on top of its own flat rooftop (bright yellow points). This small structure is distinguished, and its own smaller flat rooftop is also extracted. The building in (Figure 3.16e) also has a small rooftop structure captured as a hole in the larger building's rooftop surface, but this structures rooftop is too small to meet the minimum area constraint used during polygon filtering. Note that Polylidar3D failed to extract several small obstacles in (Figure 3.16d) for the building on the right. Such obstacles are too small to be extracted after mesh smoothing.

3.9.2.2 Ground and Obstacle Detection

The KITTI Vision Benchmark Suite provides raw datasets of Velodyne LiDAR point clouds, color video, and calibration data captured from a car while driving in Karlsruhe, Germany [101].

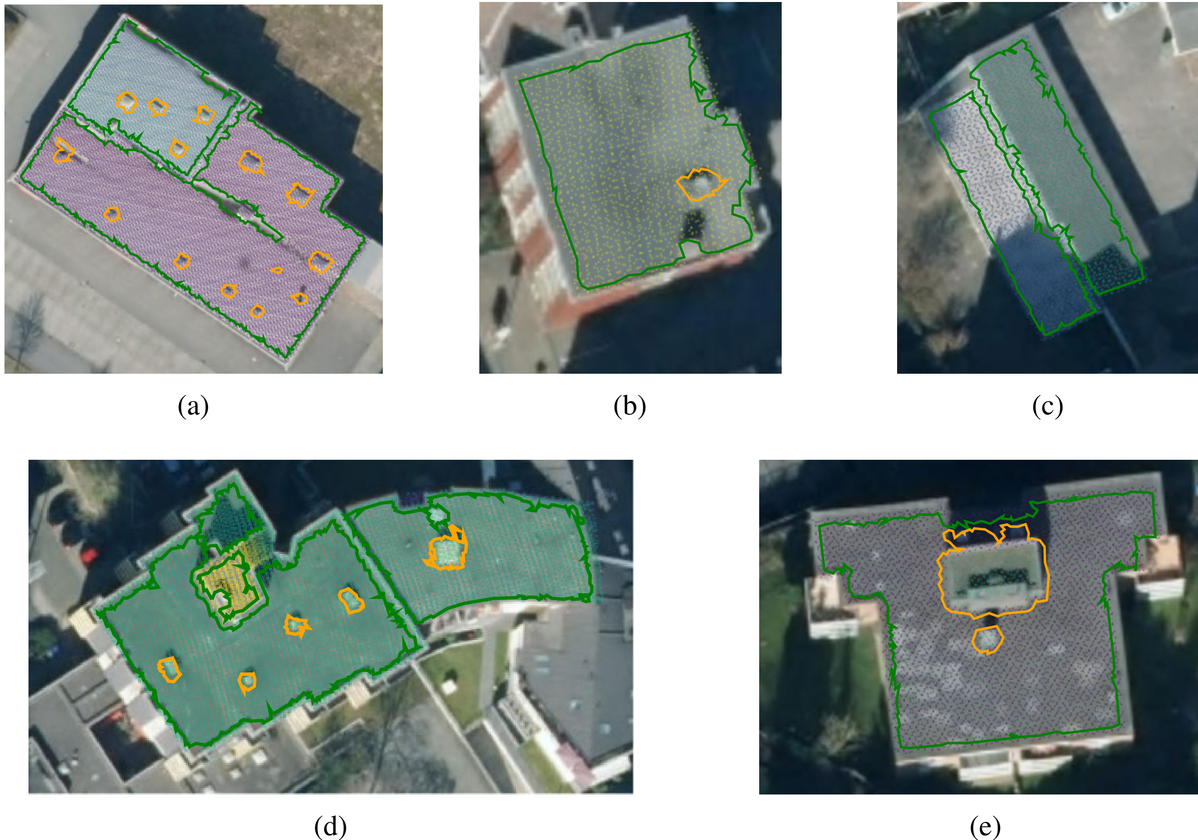


Figure 3.16: Example of Polylidar3D used with unorganized point cloud data. Demonstrates flat rooftop extraction from airborne LiDAR. Each figure (**a**,**b**,**c**,**d**,**e**) shows the satellite image overlaid with the extracted polygons (green) representing flat surfaces with interior holes (orange) representing obstacles. A colorized point cloud is also overlaid ranging from dark purple to bright yellow denoting a normalized low to high elevation. LiDAR data and satellite images are provided from [99] and [100] respectively.

Calibration data gives fixed transformations between vehicle body frame, Velodyne LiDAR frame, and camera frame. Raw point cloud is projected into the video image, and points outside the image are removed. Next, the point cloud is reduced to half its original size by skipping every odd point index. The filtered point cloud is then transformed to the vehicle body frame. A single beam/point is deemed an outlier and removed if its left and right neighboring beams are part of a common flat surface and the point strongly deviates from this surface. The filtered point cloud is then sent to the Polylidar3D front-end for 2.5D Delaunay triangulation. Flat connected surfaces on the mesh are extracted as polygons, capturing any obstacles as interior holes. Polygons are filtered and simplified using methods in Section 3.8. Filtered polygons are then transformed back to the camera frame and projected into the color image for visualization (e.g., top of Figure 3.17a). A second image is generated of the 3D point cloud and polygons from a bird's eye viewpoint (e.g., bottom of Figure 3.17a). This process is repeated for every frame of 24 distinct "drives" (continuous

video/LiDAR sequences) provided by KITTI. Visual qualitative results and execution timings are presented in Sections 3.9.2.3 and 3.9.2.4 respectively. All code is open source [62] and the videos of the generated polygons in their entirety can be found here [102].

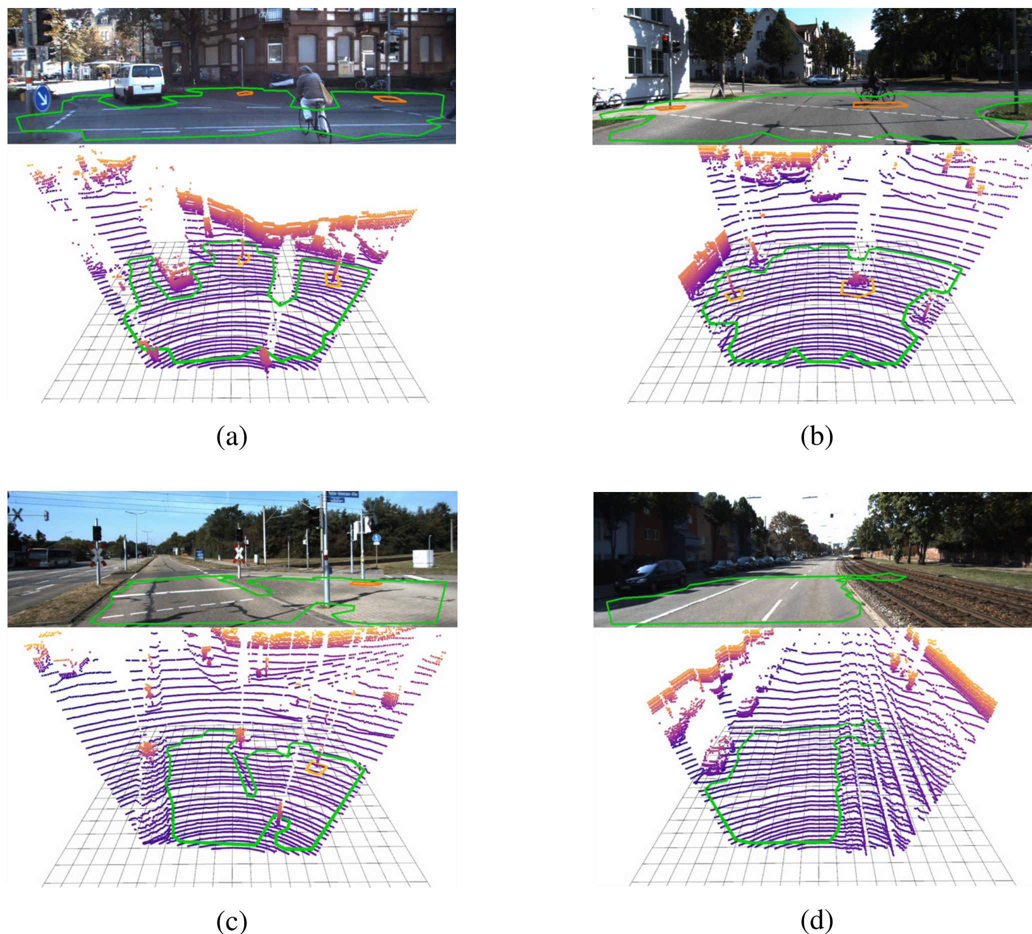


Figure 3.17: Example of Polylidar3D used with the KITTI autonomous driving dataset [101]. Four scenes (**a**,**b**,**c**,**d**) are shown, each with two subimages. The top subimage shows polygons projected into the color image while the bottom image shows 3D point cloud and surface polygon(s) from a bird's eye view. Obstacles on the ground such as the light and signal in (**a**) are extracted as (orange) holes.

3.9.2.3 Qualitative Results

Roads are not truly flat; they often have elevation changes such as a raised center line for drainage. For this reason we do not use the point-to-plane distance parameter in Polylidar3D to allow flexibility in capturing semi-flat ground surfaces in street environments. Table 3.4 shows the set of parameters used for plane/polygon extraction and filtering with KITTI. Parameter tri_{min} filters out small planes, ang_{min} provides the tolerance for flatness, l_{max} sets the max distance

between points, and $vertices^{hole}_{min}$ filters small holes. The post-processing step of polygon filtering further removes spurious holes and polygons while simplifying polygons for visualization. Note that Polylidar3D is neither designed nor trained specifically to find road surfaces; it is designed to extract flat surfaces as polygons and capture obstacles as holes. The results below thus must not be misconstrued as author intent to apply Polylidar3D for standalone road detection.

Table 3.4: Polylidar3D parameters for KITTI.

Algorithm	**Parameters**
Plane/Poly Extr.	tri_{min} = 3500, ang_{min} = 0.97, l_{max} = 1.25, $vertices^{hole}_{min}$ = 6
Poly. Filtering	$\alpha = 0.2$, β_{neg} = 0.3, β_{pos} = 0.02, γ = 30, δ = 0.5

Figure 3.17a shows Polylidar3D extracting the road and connected pedestrian walkway as one flat connected surface (green line). A light post and traffic signal are captured as holes because they are in the polygon interior. The cyclist and white vehicle are not captured as holes because they are exterior to the concave hull of the polygon. At greater distances vertical beam spacing becomes greater than l_{max} preventing additional planar surface from being included in the polygon. Figure 3.17b shows a scene where a cyclist is explicitly captured as a hole. Figure 3.17c displays the street and a slightly elevated pedestrian walkway being extracted as one polygon. Surfaces are connected at the smooth wheel chair access transition. This occurs because without a point to plane distance constraint "flat" surfaces with similar normals and a smooth spatial connection will be extracted together. However failures can occur when sensor noise inadvertently dominates a small height change between two surfaces. This is seen in Figure 3.17d when the road and part of the adjoining railroad tracks are extracted together.

3.9.2.4 Execution Timings

Polylidar3D processed 6608 frames from 24 recorded "drives" from the raw KITTI dataset. Mean execution timings for each processing step are presented in Table 3.5. The average size of the point cloud processed by Polylidar3D was 9316 points. No mesh smoothing is performed; the LiDAR is precise and has significant vertical spacing between beams such that the mesh is already sufficiently smooth. Only plane and polygon extraction are run in parallel with a maximum of four threads. The most demanding step is the post processing of polygons through filtering and simplification.

Table 3.5: Mean execution timings (ms) of Polylidar3D on KITTI

Point Outlier Removal	Mesh Creation	Plane/Poly Ext.	Polygon Filtering	Total
5.1	4.1	0.7	6.8	16.7

3.9.3 Organized 3D Point Clouds

Sections 3.9.3.1 and 3.9.3.2 show Polylidar3D applied to RGBD imagery captured and a benchmark planar segmentation dataset, respectively. Both datasets are stored as organized 3D point clouds.

3.9.3.1 RGBD Cameras

We used an Intel RealSense D435i to capture depth and RGB frames in two home environment scenes. The D435i uses stereo infrared cameras to generate a depth map. Depth noise grows quadratically with distance, and empirical evidence indicates as much as four centimeter RMS error at a two meter distance [103]. The Intel RealSense SDK provides denoising post-processing filters including decimation (downsampling), spatial bilateral smoothing, temporal filtering, and depth thresholding [104]. Parameters used for each of these filters are shown in Table 3.6, and Polylidar3D parameters used for captured RGBD data are shown in Table 3.7. Each sensor is sampled at 424 $\times$ 240 resolution in a well-lit indoor environment shielded from direct sunlight. Raw data is recorded to assure all qualitative and quantitative results can be reproduced [63].

Table 3.6: Intel RealSense SDK post-processing filter parameters.

Algorithm	Parameters
Decimation	magnitude = 2
Temporal	α = 0.3, δ = 60.0, persistence = 2
Spatial	α = 0.35, δ = 8.0, magnitude = 2, hole fill = 1
Threshold	max distance = 2.5 m

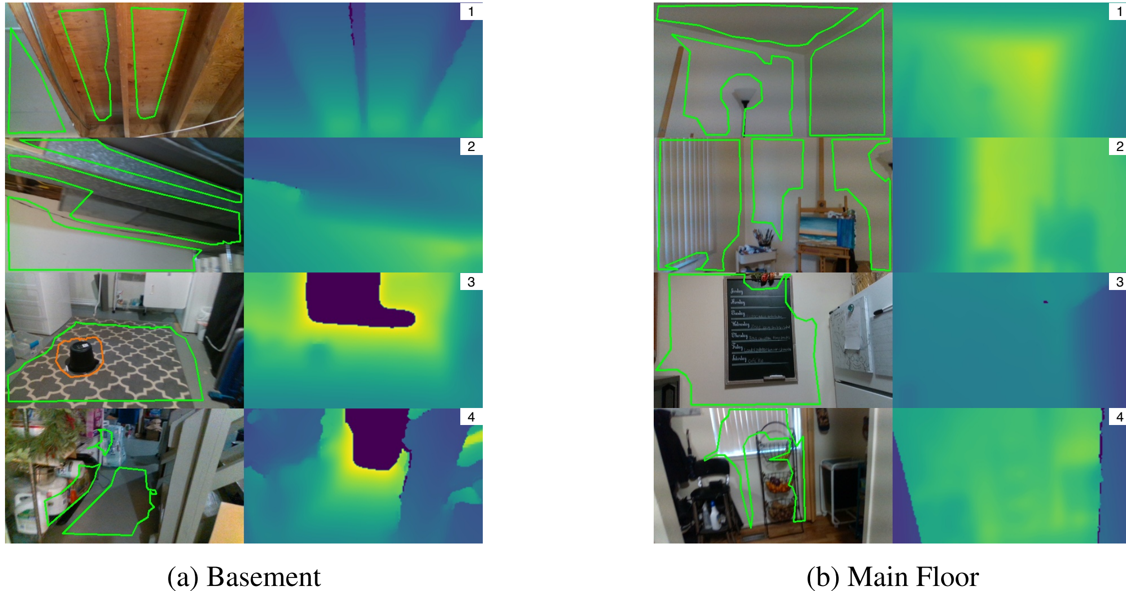
Table 3.7: Polylidar3D parameters for RealSense RGBD

Algorithm	**Parameters**
Laplacian Filter	λ = 1.0, kernel size = 3, iterations = 2
Bilateral Filter	σ_l = 0.1, σ_a = 0.15, kernel size = 3, iterations = 2
FastGA	level = 3, $v_{min} = 50, d_{peak} = 0.28, sample_{pct} = 12\%$
Plane/Poly Extr.	tri_{min} = 500, ang_{min} = 0.96, l_{max} = 0.05, $ptp_{max} = 0.1$, $vertices_{min}^{hole}$ = 10
Poly. Filtering	$\alpha = 0.02$, β_{neg} = 0.02, β_{pos} = 0.005, γ = 0.1, δ = 0.1

The first scene is composed of 2246 frames (74 s) with the camera traversing from one side of a basement to the other. While walking the camera is pointed in many directions including the floor, ceiling, and walls. Multiple dominant planar surfaces are captured at the same time. Small surfaces are explicitly removed by filtering planes and polygons that do not meet minimum number of triangles and area constraints. Figure 3.18a shows several image pairs for this scene. The left image is the RGB video with overlaid 3D polygon projections; the right image is the associated filtered and colorized depth map. Image (1) shows three planar segments with a common surface normal being extracted from the ceiling. Image (2) shows three planar segments with different normals being extracted. Image (3) shows the floor being extracted with a hole representing a bucket obstacle. The last bottom image shows Polylidar3D incorrectly capturing items on a shelf as polygons which erroneously appear as "planar" surfaces. This occurs because the small gaps between the items on the shelf become smoothed and appear planar after RealSense post-processing of the depth image.

The second scene is composed of 2735 frames (91 s) with the camera moving on the main floor through dining area, kitchen, and living room. The camera is pointed in many directions and with many dominant planar surfaces extracted as polygons. Figure 3.18b shows several still images captured. The first and second images show Polylidar3D capturing three planar segments on walls and a ceiling. A lamp and an art stand break up extracted planar segments with the polygons forming around them. For the third image, Polylidar3D does not distinguish the wall from the chalkboard because the depth difference is too small after the RealSense post-processing filters are applied. The fourth image shows Polylidar3D extracting curtains and a lower portion of a wall as one flat surface which surrounds the fruit basket. The floor is not captured because its resulting polygon did not mean minimum area constraints. The adjacent wall to the left did not meet planarity constraints with the window forming a separate segment which in itself was too small and filtered.

Table 3.8 summarizes mean execution timings for each step of Polylidar3D over all frames of each respective RGBD scene. Polygon filtering is most computationally demanding and includes polygon buffering and simplification routines. Image and mesh filtering (RealSense SDK and our



(a) Basement (b) Main Floor

Figure 3.18: Examples of Polylidar3D used with RGBD cameras. Four scenes are shown from the basement (**a**) and main floor (**b**) datasets. Color and depth frames are shown side by side for each scene. Polygons are projected onto the color image. Green denotes the exterior hull, and orange denotes any interior holes in the polygon.

mesh filtering) are extremely fast because they take advantage of the organized point cloud data structure. Dominant plane normal estimation with FastGA is quick and effective. Planar segmentation and polygon extraction are both completed in less than 2ms. Note that GPU acceleration is used for Laplacian and bilateral filtering, and four threads are used for plane/polygon extraction. All other steps are single-threaded.

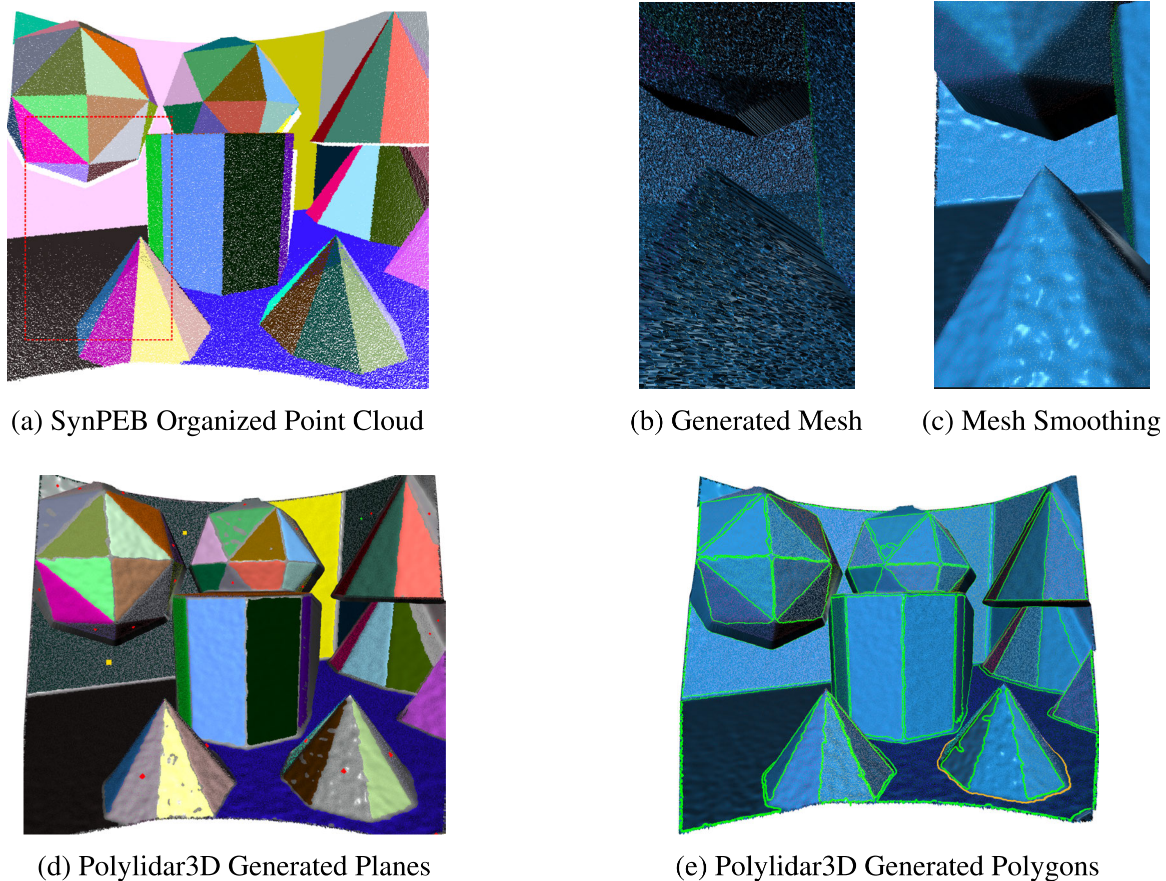
Table 3.8: Mean execution timings (ms) of Polylidar3D with RGBD data.

Scene	**RS Filters**	**Mesh**	**Laplacian**	**Bilateral**	**FastGA**	**Plane/Poly Ext.**	**Poly. Filt.**	**Total**
Basement	2.4	0.4	0.4	0.5	1.2	1.7	4.8	11.4
Main Floor	2.4	0.4	0.4	0.5	1.3	1.6	5.1	11.7

3.9.3.2 SynPEB Benchmark

We also evaluated Polylidar3D on SynPEB, a challenging benchmark dataset used to evaluate plane segmentation algorithms, created by the authors of PPE [58]. This synthetic dataset is generated from a room populated with various polyhedra resulting with an average of 42.6 planes. LiDAR scans are simulated with different levels of normally distributed radial and tangential noise producing organized point clouds. There are four levels of tangential noise in the dataset with 0.5 mdeg, 1 mdeg, 2 mdeg, and 4 mdeg standard deviation. Data is partitioned into a training set

to tune algorithms parameters and a test set for evaluation. The combination of high-noise data and numerous small, connected, but distinct planes results in challenges for plane segmentation as shown in Figure 3.19. The illustrated example uses the highest noise level (4 mdeg tangential standard deviation) from the benchmark set.



(a) SynPEB Organized Point Cloud (b) Generated Mesh (c) Mesh Smoothing

(d) Polylidar3D Generated Planes (e) Polylidar3D Generated Polygons

Figure 3.19: Example of using Polylidar3D on a SynPEB scene with the highest noise level. Point cloud, generated mesh, and mesh smoothed through Laplacian and bilateral filtering are shown in (**a**–**c**), respectively. Planes and polygons generated by Polylidar3D are shown in (**d**,**e**). Red, green, and yellow blocks in (**d**) represent missed, spurious, and oversegmented planes.

We used the training set to tune our methods parameters including mesh smoothing (Laplacian and bilateral filter), dominant plane normal estimation (FastGA), and plane/polygon extraction. We found that the most important parameter was the number of iterations of Laplacian smoothing needed. We trained a linear regression model to predict the most suitable number of iterations given an estimate of point cloud noise. All parameters used for test set reproduction are shown in Table 3.9. Note that the significant number of noisy distinct planes, up to 72, required a higher than expected refinement level for FastGA and an increased focus on smoothing.

Table 3.9: Polylidar3D parameters for the SynPEB benchmark test set

Algorithm	**Parameters**
Laplacian Filter	λ = 1.0, kernel size = 5, iterations = varies (predicted)
Bilateral Filter	σ_l = 0.1, σ_a = 0.1 , kernel size = 3, iterations = 2
FastGA	level = 5, $v_{min} = 2, d_{peak} = 0.1, sample_{pct} = 12\%$
Plane/Poly Extr.	tri_{min} = 1000, ang_{min} = 0.95, l_{max} = 0.1, $ptp_{max} = 0.07$, $vertices^{hole}_{min}$ = 10

Table 3.10 shows benchmark test results of Polylidar3D against other plane segmentation methods. The test set is limited to 1mdeg of tangential noise with results of other methods including timings provided by Schaefer et al. [58]. Note that execution times cannot be directly compared but will give an idea of real-time capability. Polylidar3D produces both a point set and polygonal representation of identified planes, however this benchmark must be evaluated by the point set. A "plane" is considered correctly identified if its point set overlaps with the ground truth plane with the standard 80% threshold described in Hoover et al. [105]. Key metrics are f representing the percent of ground truth planes identified, k indicating percent of the point cloud correctly identified, and RMSE quantifying accuracy of each plane fit. Variables n_o, n_u, n_m, and n_s represent the absolute numbers of oversegmented, undersegmented, missing, and spurious planes, respectively, compared to the ground-truth segmentation. See [58, 105] for detailed definitions of these metrics. An f metric of 47.3% indicates that Polylidar3D did not capture most of the planes in the benchmark, however the k metric of 78.3% indicates our algorithm did well in capturing the large dominant planes comprising most of the point cloud. Additionally there are fewer spurious, over segmented, and under segmented planes generated by Polylidar3D than with other methods. The RMSE value is also the lowest, indicating the predicated planes have a good fit. Plane segmentation is accomplished in significantly less time, especially in comparison to the front runner PPE. PPE's f and k metrics indicate it does an excellent job of capturing the numerous small planes in the scene, but it fails more often in capturing the large dominant planes. Polylidar3D also uniquely generates concave polygons which provide a condensed representation of identified planes.

Table 3.10: SynPEB benchmark results.

Method	**f [%]**	**k [%]**	**RMSE [mm]**	$\mathbf{n_o}$	$\mathbf{n_u}$	$\mathbf{n_m}$	$\mathbf{n_s}$	**time**
PEAC [49]	29.1	60.4	28.6	0.7	1.0	26.7	7.4	33 ms
MSAC [106]	7.3	35.6	34.3	0.3	1.0	36.3	10.9	1.1 s
PPE [58]	73.6	77.9	14.5	1.5	1.1	7.1	16.5	1.6 hr
Polylidar3D (proposed)	47.3	78.3	7.2	0.1	0.3	22.8	4.9	34 ms

Table 3.11 shows mean execution timings for each Polylidar3D method applied to SynPEB. Each organized point cloud is 500X500 but can be efficiently downsampled by striding over rows and columns. This reduces computational complexity at the cost of reduced accuracy. Note that GPU acceleration is used for both Laplacian and bilateral filtering, while plane/polygon extraction is parallelized up to four CPU cores; all other algorithm steps are single threaded.

Table 3.11: Mean execution timings (ms) and accuracy of Polylidar3D on SynPEB

Point Cloud	**Mesh Creation**	**Laplacian**	**Bilateral**	**FastGA**	**Plane/Poly Ext.**	**Total**	**f [%]**
500×500	9.3	1.1	3.0	6.6	14.9	33.9	47.3
250×250	2.0	0.5	0.7	2.5	4.1	9.8	44.6

3.9.3.3 Organized Point Cloud Mesh Smoothing

This section provides execution timing analysis of our accelerated mesh smoothing algorithms on OPC per Section 3.5. Laplacian and bilateral filtering are tested on two organized point clouds; one from a random scene in SynPEB and another random frame from our RGBD dataset. Execution timing is most influenced by point cloud size which varies substantially for these two examples. For example the SynPEB OPC has $499 \cdot 499 \cdot 2 = 498,002$ triangles whereas the RGBD frame has at most 50,218 triangles. For each filter we report CPU single-threaded, CPU multi-threaded, and GPU accelerated timings. Only four threads are used in multi-threaded runs, and a kernel size of three is used in all runs. We compare our filters with Open3D's general purpose triangle mesh Laplacian filter [83]. Note that Open3D uses a general filter and does not take into account the organized structure of the mesh and must therefore create an adjacency list for each vertex to deterimine neighbors. Additional overhead occurs by returning a new triangle mesh whereas our Laplacian implementation returns only the smoothed vertices. Open3D does not have a bilateral filter implementation nor is its Laplacian filter CPU parallelized or GPU accelerated. The smoothed meshes produced by Open3D and ours are nearly the same except for a noisy one pixel border on the boundary of our mesh.

Table 3.12 shows the results of our Laplacian filter for one and five iterations with results separated by a semicolon. Our CPU single-threaded performance is faster than Open3D. This is mostly explained by not needing to compute a vertex adjacency list. Our CPU multi-threaded results nearly reach the ideal 4X speedup in most scenarios. GPU acceleration is quite fast but has substantial overhead on the first iteration of smoothing. This is because the OPC in CPU memory must be transferred to GPU memory which is an expensive operation. This penalty is only paid once no matter how many iterations of smoothing occur.

Table 3.12: Execution timing (ms) for one and five iterations of Laplacian filtering

Data & Size	Ours			Open3D
	CPU-S	CPU-M	GPU	CPU-S
SynPEB, 500 × 500	7.0; 35.0	2.0; 9.2	0.8; 0.9	205.9; 240.6
RGBD, 120 × 212	0.7; 3.5	0.2; 0.9	0.1; 0.2	14.3; 17.5

Table 3.13 shows execution timing results of our bilateral filter for one and five iterations with results separated by a semicolon. This filter is substantially slower than the Laplacian filter primarily because both triangle normals and their centroids must be computed before this filter can run (included in timing). Additionally each of these data structures is nearly twice as large in memory as the input OPC ($\approx$ 2 triangles per vertex). The weighting of neighbors in Equations (3.4) and 3.5 relies on an exponential function which is significantly slower than the floating point multiplication and division required for Laplacian filter per Equation (3.2). Finally significantly more neighbors and data are used in Equation (3.3) to produce a smoothed normal. A maximum of 16 triangle neighbors (accessing both their normals and centroids) are used for the bilateral filter whereas the Laplacian uses a max of eight vertex neighbors. The memory transfer from CPU to GPU is significantly higher as well because 4X as much memory is needed.

Table 3.13: Execution timing (ms) for one and five iterations of bilateral filtering

Data & Size	CPU-S	CPU-M	GPU
SynPEB, 500 × 500	73.0; 354.0	19.4; 90.9	3.2; 4.4
RGBD, 120 × 212	7.2; 35.0	1.9; 9.2	0.5; 0.5

3.9.4 User-Defined Meshes

We apply Polylidar3D on two meshes of an indoor home environment. Both meshes were generated by gathering color and depth frames from an Intel RealSense D435i camera and integrating them into a triangular mesh using methods from Zhou et al. [82] implemented in Open3D [83]. The method works by integrating the frames into a voxel grid and then performing marching cubes on the grid to create a triangular mesh. An important parameter in this process is the voxel size which if small makes a denser mesh that may also integrate noise from the sensor. The first mesh is of a basement and has a 5 cm voxel size leading to a smoother approximation. Note this spacing is significantly higher than the noise of the sensor. The second mesh is of the main floor and is much larger and denser with 1 cm voxel spacing leading to significantly more noise from the

RGBD camera. Only the main floor mesh is post-processed with Laplacian filtering using Open3D. The basement mesh is composed of 60,620 triangles while the main floor mesh has 3,618,750 triangles. The parameters for Polylidar3D for both meshes are shown in Tables 3.14 and 3.15. The most significant difference in parameter sets is that the basement is configured to capture small surfaces (lower tri_{min} and γ) in comparison to the main floor.

Table 3.14: Polylidar3D parameters for the basement mesh

Algorithm	**Parameters**
FastGA	level = 4, $v_{min} = 15, d_{peak} = 0.1, sample_{pct} = 12\%$
Plane/Poly Extr.	tri_{min} = 80, ang_{min} = 0.95, l_{max} = 0.1, $ptp_{max} = 0.08$, $vertices^{hole}_{min}$ = 6
Poly. Filtering	$\alpha = 0.01$, β_{neg} = 0.025, β_{pos} = 0.0, γ = 0.07, δ = 0.05

Table 3.15: Polylidar3D parameters for the main floor mesh

Algorithm	**Parameters**
FastGA	level = 4, $v_{min} = 15, d_{peak} = 0.1, sample_{pct} = 12\%$
Plane/Poly Extr.	tri_{min} = 1000, ang_{min} = 0.95, l_{max} = 0.1, $ptp_{max} = 0.08$, $vertices^{hole}_{min}$ = 6
Poly. Filtering	$\alpha = 0.02$, β_{neg} = 0.05, β_{pos} = 0.02, γ = 0.25, δ = 0.1

Figure 3.20a,b show the polygons output from Polylidar3D on the basement mesh. The floor and all walls are appropriately captured as well as any obstacles on their flat surfaces. The top surface of the chair, table top, and monitor have also been captured. However there are several small planar segments on an occluded wall in (Figure 3.20b) which may not be desirable for capture. In this same image a collection of stacked boxes are not truly flat and the polygon line segment goes "behind" the mesh surface. Figure 3.20c,d show polygons output for the dense first floor mesh. The floor and most walls have been successfully captured. However some walls have too much noise thus do not meet planarity constraints, e.g., the far wall in (Figure 3.20c). The ground floor is not extracted as one continuous polygon instead separating at the edge of the mesh in (Figure 3.20d). This occurs because the floor areas have differences in height (in the mesh, not in reality); the point-to-plane distance constraint is exceeded between these two surfaces causing two extractions. This can be remedied by increasing ptp_{max} by 1cm but is left here to highlight the issue.

3.9.4.1 Parallelization Analysis

This section explores how Polylidar3D scales with additional CPU threads. We specifically focus on plane/polygon extraction in Polylidar3D's back-end. Both meshes are used in these steps

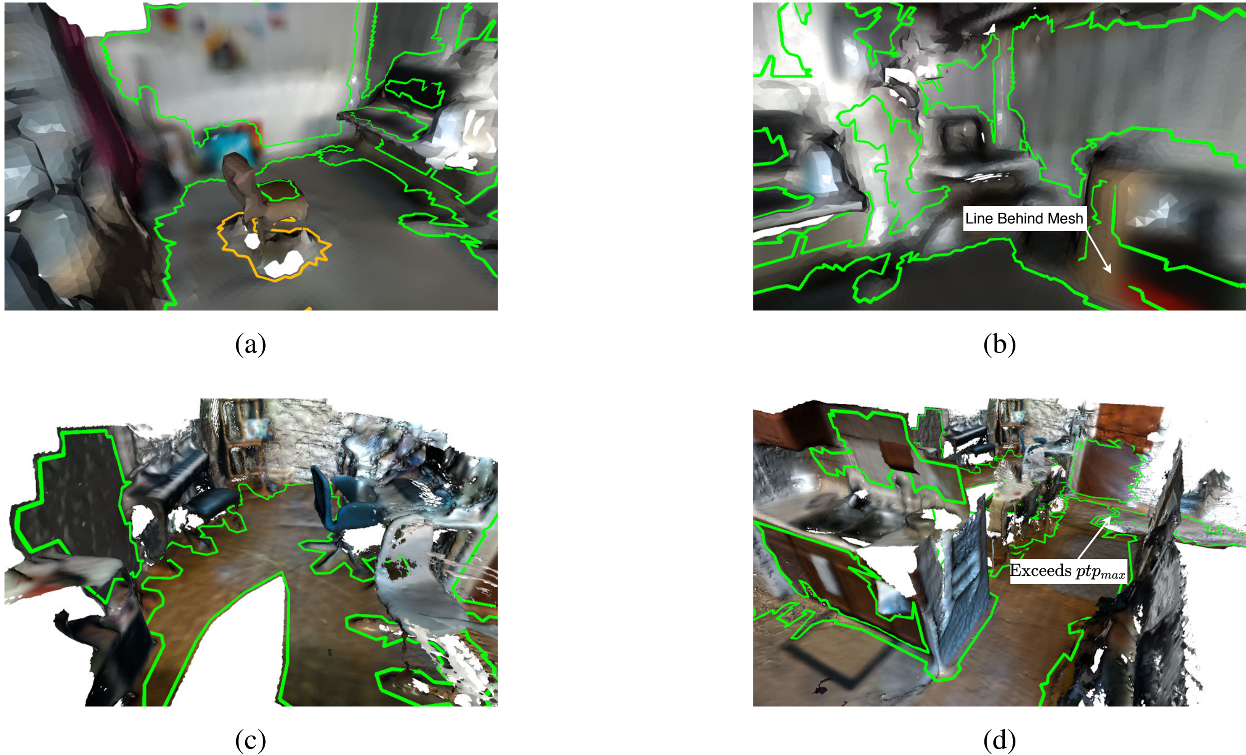
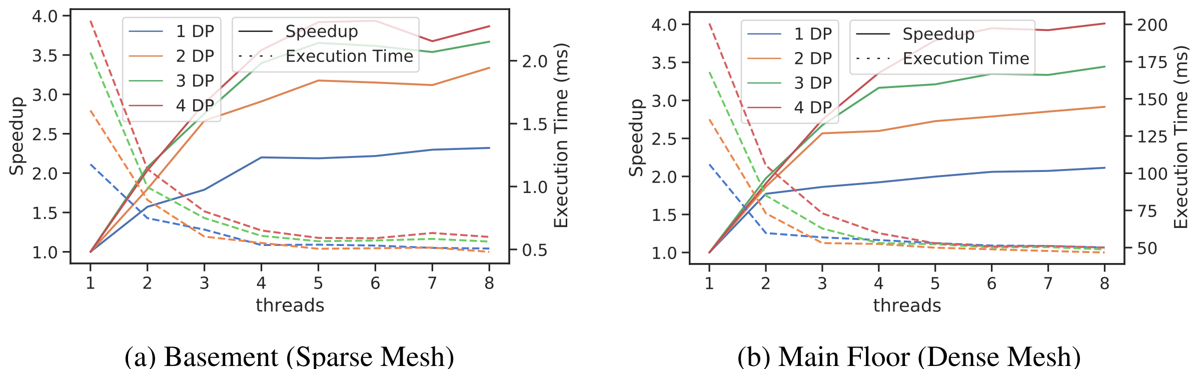


Figure 3.20: Example of Polylidar3D used with user defined meshes. Meshes are of an indoor home environment. (**a**,**b**) show results on a basement mesh which has a smoother approximation and less noise; (**c**,**d**) show results on a significantly larger, denser, and nosier mesh of the main floor.

and we limit the dominant plane normals to the top four in the scene, i.e., only floors and walls are extracted. Figure 3.21 shows the parallelization speedup and execution timing of plane/polygon extraction as up to eight threads are provided. The color of the line indicates how many dominant plane normals are requested for extraction, i.e., blue indicates only the floor while orange indicates both the floor and one wall. The more dominant plane normals requested the more CPU cycles are needed.

Figure 3.21a shows the speedup and execution timings of the sparse basement mesh. The parallel speedup does not go any higher than 2.4 with one dominant plane normal (blue-solid) and reaches approximately 4.0 with four dominant planes (red-solid). The execution timings (dashed lines) clearly show the diminishing returns as more threads are provided and plateaus around 0.5 ms at 4 threads. Figure 3.21b shows a similar trend for the much larger and more dense main floor mesh. The trends are clear that greater speedup is possible as more unique dominant planes normals are requested because this work gets partitioned to independent tasks. However there is a limit to this parallelism as not all procedures within the tasks are themselves parallelized. This is a clear example of Amdahl's law in effect which explains a theoretical limit to speedup as a function of the percent of a program that is actually parallelizable [107]. In essence the theoretical speedup is

always limited by the serial tasks, which in our case becomes (roughly) the combined execution time of planar segmentation of the single largest dominant plane normal and the polygon extraction of its largest planar segment. New threads do not reduce the time to complete these tasks because their algorithms are serial.



(a) Basement (Sparse Mesh)

(b) Main Floor (Dense Mesh)

Figure 3.21: Results of parallel speedup and execution timing of Polylidar3D. Both basement (**a**) and main floor meshes (**b**) are analyzed. Solid lines indicate parallel speedup and link to the left y-axis while the dashed lines indicate execution time and link to the right y-axis. The color indicates number of dominant plane (DP) normals extracted.

3.10 Discussion

Results show Polylidar3D successfully extracts flat surfaces as polygons with interior holes from unorganized/organized point clouds and user-provided meshes. One of Polylidar3D's primary strengths is its polygon extraction speed. The key to this speed is the fast construction of half-edge triangular meshes used directly in polygon extraction. No secondary re-triangulation is necessary after planar segmentation. Our Fast Gaussian Accumulator was benchmarked against competing K-D tree methods and shown to be two times faster and effective at identifying dominant plane normals. Data and task-based parallelism is also exploited to efficiently allocate work to available CPU cores.

Results also illustrate limitations. First, rooftop and ground extraction in Section 3.9.2 shows that only one plane normal can be extracted from unorganized 3D points clouds. As described in our methods the front-end currently performs 2.5D Delaunay triangulation which requires 3D $\rightarrow$ 2D projection. This projection is most suitable when the sensing viewpoint and flat surface of interest are aligned, as is for airborne LiDAR point clouds. However this is not a hard requirement as shown with ground detection from the KITTI dataset. We chose 2.5D Delaunay triangulation for its speed, however other methods may be used such as the ball pivot algorithm [81] or Poisson surface reconstruction [80]. These methods created 3D meshes which could then be processed by Polylidar3D.

Polylidar3D planar segmentation expects a mesh to be reasonably smoothed. The amount of smoothing depends on user-specified parameters for surface extraction and the noise of the input data. If only large distinct flat surfaces are required then minimal smoothing is necessary. We define distinct surfaces as plane normals that are well-separated on the Gaussian Accumulator (e.g., $90°$). This smoothing aids in GA peak detection and appropriately groups triangles during planar segmentation.

The Fast Gaussian Accumulator can only detect plane normals; it currently has no concept of origin offset. This means that if there are two flat surfaces separated far from each other in a scene, with similar (but not the same) surface normals, it is possible they will appear near each other on the GA and be merged. Noise in the mesh affects how close these two peaks can be on the GA and still be detected as distinct peaks. As the mesh is further smoothed (with edge-preservation filters) the noise is reduced and the peaks become more defined. This is exactly what had to be done to detect the numerous noisy planes in the SynPEB benchmark. Also, group assignment in Algorithm 3.4 will assign common triangles to these detected peaks if they meet a user defined angular threshold ang_{min} from a detected peak. This means any detected peaks should be greater than $2 \cdot arcos(ang_{min})$ from each other to guarantee no overlap. Note also that ang_{min} can be increased as the mesh is smoothed.

Only dominant planes, flat surfaces that account for most of the 3D data, can be reliably captured from organized point clouds and user-provided meshes. We see numerous qualitative examples of this from RGBD sensor data, the SynPEB benchmark, and user-provided meshes. Polylidar3D is only able to extract 47.3% of the average 42.6 planes in the SynPEB test scenes. However the percent of point cloud metric k at 78% shows Polylidar3D doing an excellent job of capturing large dominant planes. Scenes in this benchmark are the antithesis for what Polylidar3D was designed for (dozens of small noisy planes), yet we show Polylidar3D still performs well in important metrics such as minimizing the number of over/under-segmented planes, spurious predictions, and execution time. We believe these metrics taken as a whole demonstrate Polylidar3D's efficiency and reliability for polygon extraction of dominant planar surfaces.

3.10.1 Point Cloud Characteristics and Parameter Selection

Each of the experimental result sections use different sensors resulting in dissimilar point cloud characteristics such as density, spatial distribution, noise, and accuracy to the ground truth surface. These characteristics have been studied both for Airborne Laser Scanning (ALS) technology [108, 109], Velodyne LiDAR [110] and Intel RGBD [103] sensors. The airborne LiDAR point clouds used in Section 3.9.2.1 were captured in swathes with a Nominal Point Spacing (NPS) of 30 cm creating a semi-random distribution with less than 2.5 cm RMSE [99]. Section 3.9.2.2 uses single scan Velodyne HDL-64E point clouds that are dense in azimuth/rotation but sparsely

distributed in elevation due to beam spacing with a reported depth accuracy of less than 2 cm [101]. Finally RGBD sensor data analyzed in Section 3.9.3.1 contains dense and uniformly distributed point clouds but with noise growing quadratically with distance [111].

The Polylidar3D parameters in Tables 3.3, 3.4, and 3.7 were chosen to give the best qualitative results for each sensor and then applied to all data in their respective section. Most of the parameters are straightforward and can be interpreted and justified when taking into consideration point cloud characteristics. First, the parameters for Laplacian and Bilateral filtering are influenced by the noise and density of the point cloud. No smoothing was required on the KITTI dataset (Velodyne sensor) because the sensor had minimal noise with large point spacing leading to long skinny triangles for a smooth mesh. The airborne LiDAR point clouds of rooftops were much denser leading to smaller noisy triangles requiring two iterations of Laplacian and bilateral filtering. The raw RGBD depth image is extremely dense and noisy requiring the use of Intel's own post-processing image filters with parameters described in [104]. However the generated point clouds were still noisy needing additional smoothing using three iterations of both Laplacian and bilateral filtering.

Our Fast Gaussian Accumulator (FastGA) algorithm has several parameters which are influenced both by point cloud characteristics as well as real-time computational needs. First the $sample_{pct}$ parameter will downsample triangle normals in the mesh as input to FastGA to reduce computation time. The authors found that a 12% down-sample was more than sufficient to extract dominant planes in all scenes in the experiments. GA refinement level should be set to 3 or 4 depending on the noise of the point cloud and accuracy required for extracted dominant plane normals. A lower refinement level has a coarser tessellation of the sphere but is better for noisy data because noise is "smoothed" into larger histogram cells. Peak detection parameter v_{min} is scaled between 0–255; the authors found a value between 15–50 to be best. If only large dominant planes are needed then a high value may be used but must be lowered to detect smaller planar segments.

Plane and polygon extraction algorithm parameters are influenced by density, point spacing, and noise in the point cloud. The parameter l_{max} should be set to the maximum triangle edge length in the mesh expected for a flat surface which is in turn influenced by the point spacing. For example, KITTI dataset points are well-separated requiring l_{max} to be set higher than one meter while the dense RGBD point clouds require only 5 cm. The parameter ang_{min} depends on mesh smoothness and the user's tolerance for deviation from perfect flatness. The authors found a value between 0.94–0.99 to be ideal which corresponds to a 20°–10° allowance in angle deviation. The parameter ptp_{max} forces all segmented points to be within a certain distance from a segment's geometric plane which is influenced by point cloud noise. Small planar segments will be quickly removed if the number of triangles is below the parameter tri_{min}. Additionally, small interior holes in polygons will be removed if their number of vertices are below the parameter $verrtices^{hole}_{min}$. Both these parameters are influenced by point cloud density and the user's tolerance for superfluous planes

and/or holes in polygons.

Polygon post-processing parameter values are chosen more from scene context than from underlying point cloud properties. For example parameters γ and δ remove polygons and holes based on minimum area constraints and are not influenced by the point cloud. If a user desires to extract only large surfaces (e.g., walls and floors) they can set γ to be high to filter small planar patches such as a chair seat. Buffering and simplification parameters $(\alpha, \beta_{neg}, \beta_{pos})$ are used to remove redundant vertices and extraneous details for visualization and subsequent processing.

3.10.2 Algorithmic Complexity

The time complexity of Polylidar3D varies depending upon the data input. Unorganized 2D and 3D point clouds have a total time complexity of $\mathcal{O}(n \log n)$ where n is the number of points. This is limited by the 2D and 2.5D Delaunay triangulation in the front-end which can only be completed in $\mathcal{O}(n \log n)$ time [48]. However the time complexity for organized 3D point clouds and user provided meshes is $\mathcal{O}(n)$. The mesh creation procedure for organized point clouds exploits the image structure to quickly determine pixel neighbors and creates a mesh in $\mathcal{O}(n)$ time. This time complexity is the same for user-provided meshes which uses $\mathcal{O}(1)$ insert/access hashmaps to determine half-edge relationships. Mesh smoothing on organized point clouds is $\mathcal{O}(n)$ by once again exploiting the image structure to determine neighboring triangles for smoothing. Dominant plane normal estimation using FastGA is likewise completed in $\mathcal{O}(n)$ time. The $s2id$ generation (using Hilbert curves) is independent of the number of points n being integrated into the sphere. The search process for the histogram cell is likewise independent of n and only influenced by the refinement level of the GA which is known at compile time. Region growing for planar segmentation is also completed in $\mathcal{O}(n)$ by having quick $\mathcal{O}(1)$ access to neighboring triangles using half-edges array $\mathcal{HE}$. Polygon extraction itself is completed in $\mathcal{O}(n)$ time [84]. Any plane and polygon extraction parallelism reduces time by a constant factor which does not affect the overall time complexity of these algorithms.

3.10.3 Future Work

There are three significant techniques that will improve Polylidar3D's robustness in future work: polygon merging, time integration, and integrating intensity/color data. Many planar segmentation algorithms perform "plane" merging of extracted segments (point sets) which are deemed similar by Euclidean distance and plane-fit error tolerance [49, 69, 67, 60]. This is most often used to combine oversegmented predictions of a common surface. Polylidar3D can be extended to perform the same action with polygons. Detailed meta-data about each polygon can be stored to aid in the merging process including geometric plane normal, centroid, axis-aligned bounding box, and even

the convex hull if necessary. This information will aid the pairwise matching between polygons in a scene before a possibly expensive polygon merger. There are several methods to perform a non-convex polygon merge including morphological operations such as dilation and erosion.

Polylidar3D processes each point cloud distinctly. Time integration incorporates data from multiple data frames in a sequence by filtering and refining extracted polygons based on previous results. In a static scene with fixed sensor viewpoint time integration can reduce the variance of polygons produced over time. All linear rings of the polygon (both hull and holes) can be explicitly tracked using meta-data previously discussed and removed if certain thresholds are not met. With a dynamic scene or moving sensor time integration would require significant extension to Polylidar3D to incorporate additional data such as sensor (vehicle) motion estimates and even semantic scene information. Additional work investigating the use of Bayesian filtering will be done.

Data such as intensity and/or color of the point cloud can be used to further determine similarity between neighbors in the point cloud during region growing. Such data has been shown to improve results for point cloud registration [57] and mesh smoothing [51]. Additionally, deep neural network may perform semantic segmentation on RGBD images to quickly output class labels for each pixel in the image [112]. This information can then be fused into Polylidar3D to better inform partitioning of work and similarity between neighboring triangles.

3.11 Conclusions

This chapter introduced Polylidar3D, a non-convex polygon extraction method capturing flat surfaces from a variety of 3D data sources. Front-end methods transform unorganized point clouds, organized point clouds, and 3D triangular meshes to a common half-edge triangular mesh format. Back-end core algorithms perform mesh smoothing, dominant plane normal estimation, planar segmentation, and polygon extraction. A novel Gaussian accumulator, FastGA, was demonstrated robust and quick at detecting dominant plane normals in a 3D scene. These dominant plane normals are used to parallelize planar segmentation and polygon extraction. Polylidar3D is evaluated in five separate experiments with airborne LiDAR point clouds, automotive LiDAR point clouds, RGBD videos, synthetic LiDAR benchmark data, and meshes of indoor environments. Qualitative and quantitative results demonstrate Polylidar3D's speed and versatility. All of Polylidar3D is open source and available to be freely used and improved upon by the community [36, 61, 65].

CHAPTER 4

Roof Shape Classification from Satellite Images and LiDAR Data

4.1 Introduction

Geographic information system (GIS) data are openly available for a variety of applications. Data on terrain height and type have historically been available, with high-accuracy labeled data now increasingly available, e.g., building footprints and heights. Systematic characterization of building roof architecture and slope offers a new dimension to traditional terrain data. These data could be used to rapidly identify building change or damage from the air, to improve in-flight localization capabilities in GPS-denied areas, and to inform small Unmanned Aircraft Systems (UAS) of alternative ditching sites, a problem previously investigated by the authors [5, 113]. Databases such as OpenStreetMap (OSM) [114] provide limited roof information, but such data have been manually entered to-date thus is sparse.

This chapter fuses satellite imagery and airborne Light Detection and Ranging (LiDAR) data through multiple stages of machine learning classifiers to accurately characterize building rooftops. With these results, roof geometries worldwide can be stored in an easily-accessible format for UAS and other applications. Supervised training datasets are automatically generated by combining building outlines, satellite, and LiDAR data. The resulting annotated dataset provides individual satellite image and LiDAR (depth) image representations for each building roof. Roof shapes are automatically categorized through a novel combination of convolutional neural networks (CNNs) and classical machine learning. Transfer learning is employed in which multiple pre-trained CNN model architectures and hyper-parameters are fine-tuned and tested. The best performing CNN for both satellite and LiDAR data inputs is used to extract a reduced feature set which is then fed into either support vector machine (SVM) or random forest classifiers to provide a single roof geometry decision. Validation and test set accuracies are evaluated over a suite of different classifier options to determine the best model(s). A range of urban environments are used to train and test the proposed models. Data from Witten, Germany; Ann Arbor, Michigan; and the Manhattan borough of New York City, New York are collected and manually labeled to represent small to large metropolitan city centers. We show that combining datasets from both small and large cities leads

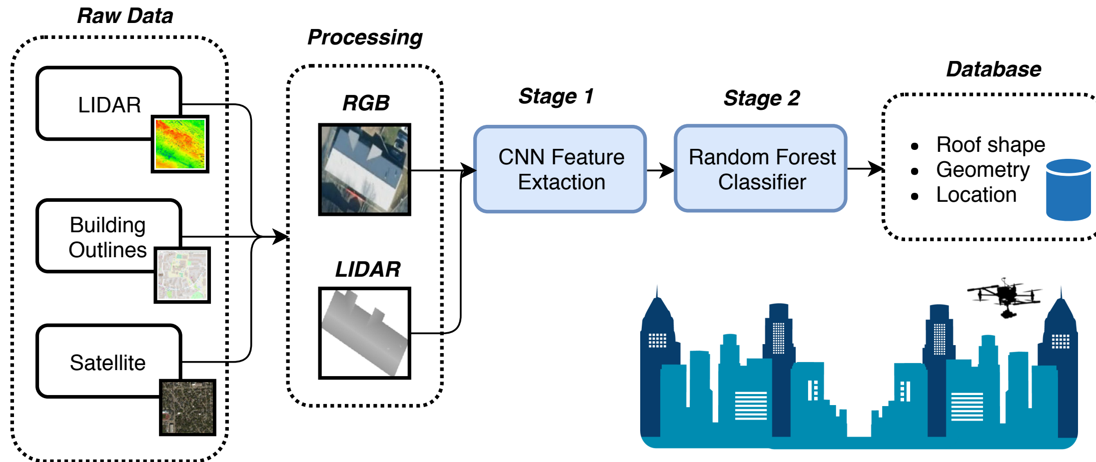


Figure 4.1: Roof classification data fusion and processing pipeline. LiDAR, building outlines, and satellite images are processed to construct RGB and LiDAR images of a building rooftop. In Stage 1, these images are fed into a CNN for feature extraction, while Stage 2 uses these features with a random forest for roof classification. These data can be stored for quick reference, e.g., navigation or emergency landing site purposes.

to a more generalized model and improves performance. Figure 4.1 provides an overview of the data processing pipeline and illustrates a UAS localization and contingency landing use case [113]. Specific contributions include:

- Over 4500 building roofs spanning three cities have been manually classified and archived with a satellite and LiDAR depth image pair. This dataset is released with this chapter.
- New “complex-flat” and “unknown” roof shape classes enable the machine classifier to distinguish flat roofs with infrastructure (e.g., air conditioning and water towers), unfamiliar roof shapes, and images of poor quality.
- This work significantly reduces the set of outliers that previously required manual removal for training and test datasets (from 45% in [115] down to 5% in this work). This chapter’s test set accuracies represent a reasonable expectation of results when deployed in new areas.
- An analysis of confidence thresholding is presented to improve the model’s predictive power. This ensures only correct labels are assigned which is critical for use in high risk scenarios.
- Expanded results are presented from use of a single trained classifier (over Witten and Manhattan) tested with datasets from three cities, one of which (Ann Arbor) was never used for training or validation.

The chapter is structured as follows. First, GIS data sources and prior roof geometry classification work are summarized. Next, background in machine learning and data extraction methods

is provided. Specific methods to extract data for input to this chapter's machine learning feature extraction and classification system are presented, followed by a description of training, validation, and test runs performed. Statistical accuracy results are presented followed by a discussion and conclusions.

4.2 Background

This section summarizes related work. First, GIS data sources and previous efforts to extract roof geometries are reviewed. Next, convolutional neural networks (CNNs) and their application to feature extraction are reviewed.

4.2.1 Roof Geometry Classification

Satellite color images and 3D point cloud data from airborne LiDAR sensors provide complementary roof information sources. High resolution satellite images offer rich information content and are generally available worldwide. However, extracting 3D building information from 2D images is difficult due to occlusion, poor contrast, shadows, and skewed image perspectives [116]. LiDAR point clouds provide depth and intensity measurements that capture the features of roof shapes, yet LiDAR does not offer other world feature information from ambient lighting intensity and color. LiDAR point cloud data are often processed and converted to digital surface models (DSM) representing the top surface layer of any terrain.

The amount of detail desired for roof geometry influences data processing methods. Detailed reconstruction of 3D city maps for visualization or simulation purposes often requires a detailed representation of the geometric elements in a 3D building model. This is often accomplished using a model based or data driven approach. In a model-based approach, a collection of parameterized building models are selected as possible candidates given prior knowledge of buildings in the geographic region of interest. Buildings are then fit to these models using the gathered data points, and the best 3D model is chosen. This method can reliably extract parameters from data points so long as the building shape is simple and roof details are not required [117]. A data-driven approach does not require a priori knowledge of building structures, instead using large datasets to generate a high-fidelity model. Data points are grouped to define planar surfaces which in turn are used to construct 3D lines fully specifying building geometry. For example, work by Ref. [118] segments potential roof points in a building through their normal vectors, which are later collapsed into planar elements that conform to the defined constraints of roof planes.

The photogrammetry community has demonstrated recent success in applying data driven approaches for 3D building reconstruction. Ref. [119] proposed a dynamic multi-projection-contour (DMPCA) framework that uses super generalized stereo pairs (SGSP) to generate and iteratively

refine 3D buildings models. This method minimizes the total difference between the projection-contour of a building across SGSPs and the projection-contours of the simulated 3D model. Using building images captured by a UAS, Ref. [54] generated a dense point cloud from image matching. This point cloud is then clustered by RANSAC shape detection. Planar geometry is then determined through least squares fitting, and finally refined details (e.g., dormers and eaves) are modeled. Ref. [120] proposed the use of both thermal infrared (TIR) and RGB images taken by UAS to generate point clouds. These distinct point clouds are then aligned with an iterative closest point (ICP) procedure generating a high fidelity building model with accompanying RGB textures. Similarly, Ref. [121] proposed a roof-contour and texture-image guided interpolation (RTGI) method that generates facades as well as texture maps of buildings. A common theme in most of the above research is the increased use of UAS to capture high resolution data from multiple viewpoints to improve model accuracy.

The localization and landing site applications for UAS referenced by this chapter only require a simple classification of building roof shape. In fact, complex model representations are undesirable given that UAS applications would be computed by a low-power lightweight embedded processor. Classical machine learning algorithms such as support vector machines (SVM), logistic regression, and decision trees are often used in these classification scenarios but invariably face computational complexity challenges caused by the high dimensionality found in these GIS data sources. To employ these algorithms, a reduction in dimensionality through feature selection is often performed. Recent work by Ref. [122] performed roof classification through SVM's by reducing a DSM image of a roof to a set of handcrafted features such as the number of roof surfaces for each building and the distribution of the binned slope angles. A set of 717 buildings in Geneva, Switzerland were manually labeled for training and testing purposes of the model, resulting in an overall accuracy of 66% for a six roof type classification. The same authors also experimented using a random forest classifier with similarly handcrafted features from a DSM on a 1252 building dataset from Switzerland. The test set was a 25% random sampling of the labeled dataset with a reported total accuracy of 70% when identifying six roof types [123].

Recent advances with deep learning with techniques such as convolutional neural networks (CNN) have demonstrated the ability to accurately and robustly classify high dimensional data sources such as camera images [124]. The GIS community has begun to apply CNNs to roof identification. Perhaps most closely related to this chapter, Ref. [125] trained CNNs using satellite Red Green Blue (RGB) imagery and Digital Surface Model (DSM) images to label basic roof shapes. However, the final predicted roof shape was simply taken as the highest probability result between the two models (RGB, DSM); no feature fusion or training was performed between different modalities. Training and test set sizes are not explicitly provided, however two test set accuracies are reported: 95% and 88% using the the authors' best model.

Complementary work by Ref. [126] fine-tuned a CNN using patched satellite images of building rooftops. Using the fine-tuned CNN, the authors extracted high-level features of images as inputs to a second-stage SVM classifier. Approximately 3000 images in Munich, Germany were used for training and testing resulting in 76% total accuracy. Our chapter adopts an analogous two-stage processing approach to roof classification with the novel addition of LiDAR and satellite image feature fusion. Specifically, this fusion allows the creation of a nonlinear decision function that exploits the strengths of each modality. Finally, unlike all previous work we have encountered, this chapter incorporates data from geographically diverse cities and assesses models on their ability to generalize across regions.

4.2.2 The Convolutional Neural Network (CNN)

An artificial neural network is composed of a series of functional layers connected in a weighted graph structure. Each neural network layer consists of a *node* vector, a node activation function, and weighted edges typically feeding forward to the next network layer. A layer is considered fully connected (FC) if every node in the layer is connected to every node in the previous layer. Repeating layers are called blocks and can have unique structural and functional designs. An example is shown in Figure 4.2a.

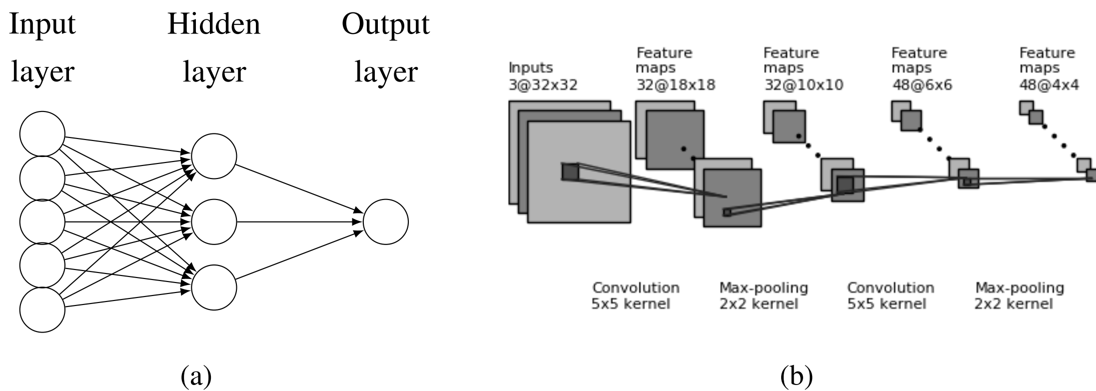


Figure 4.2: Example of a fully connected and convolutional neural network. (**a**) Fully connected neural network with one hidden layer. (**b**) CNN with two convolutional blocks.

Convolutional neural networks (CNNs) are primarily distinguished by their shared weights and translation-invariance characteristics. CNNs hold multiple convolutional blocks that are generally composed of a convolutional filter layer, an activation layer, and finally a pooling or downsampling layer. These blocks generate high level features from their inputs which are then fed into the next set of blocks. Figure 4.2b shows an example of an input image passing through two convolution blocks. Eventually, a final feature set is produced which feeds into fully-connected layers generating an output feature vector or classification. The dimensions of CNN blocks and how they interconnect

with each other and subsequent layers determines the *architecture* of the network. Researchers have developed several CNN architectures that have been tested against large image sets such as Imagenet [127]. These networks are trained from scratch, meaning their weights are randomly initialized, and take weeks (of real-time) to converge even with the aid of general purpose graphics processing units (GPGPUs). For example, the Large Scale Visual Recognition Challenge 2012 (ILSVRC2012) holds a dataset of over a million images with the task of distinguishing between 1000 categories. CNN classifiers achieved "Top 5" accuracies of greater than 95%.

For a CNN to be applied to an application such as roof classification, a large supervised domain-specific training set is needed. If a large training dataset is not available, a technique called transfer learning can be applied. Transfer learning accelerates machine learning by transferring knowledge from a related, perhaps generalized, domain to a new domain [128]. This technique requires the use of an existing *pre-trained* CNN. The beginning layers of the pre-trained CNN often generate domain-independent features (e.g., features which distinguish lines or color changes) that will be useful for other domains. The base architecture and associated weights are used as the starting layers in a new CNN to be trained. An opportunity also arises during the training process to *freeze* a variable number of initial layers' weights, thereby reducing the number of parameters to learn and overall training time. In essence, the more initial layers that are frozen, the more the CNN relies upon the pre-trained model's domain knowledge.

In addition to transfer learning, image augmentation (rotation, cropping, etc.) can be used to artificially inflate the training dataset, which tends to reduce overfitting. Parameters such as the size of the fully connected layers or number of frozen initial layers influence the accuracy of the model. Optimal parameters are determined by evaluating multiple trained networks against a validation set and assessing its accuracy. Parameter adjustments are grouped as hyperparameters to determine an optimal model structure.

4.2.3 Feature Extraction and Classical Machine Learning

Supervised learning classification algorithms such as support vector machines (SVM) and decision trees have difficulty handling large GIS datasets such as images or point clouds. However, when given a reduced feature set, both approaches can be effective for final classification [129, 122]. Researchers have begun to use CNN's to extract a "Stage 1" reduced feature set that is then fed into a downstream "Stage 2" classifier. Support vector machines (SVM) divide a feature space into linear hyperplanes for class separation, but often use kernels to project input features into higher-dimensional spaces to create non-linear decision boundaries. The best kernel to be used is dependent upon the feature set provided; however, linear, polynomial, and radial based function (rbf) kernels are often the first used. Figure 4.3a shows an SVM separating a binary class (red/green) with the line that maximizes margin distance between classes; a linear kernel is used. Similarly, random

forest classifiers create nonlinear decision boundaries through ensemble learning, a technique that trains many decision trees on random subsets of the training data as shown in Figure 4.3b. The forest is represented by the many decision trees created and trained, and the final classification is the statistical mode of the trees' collected predictions. The forest is often limited by the number of trees (i.e., number of estimators) as well as the maximum depth of any tree in its collection. Random forest classifiers are resilient to overfitting through the collected knowledge of the ensemble. This chapter will train both SVM and random forest classifiers on CNN extracted features from satellite and LiDAR building images in an effort to improve classification accuracy.

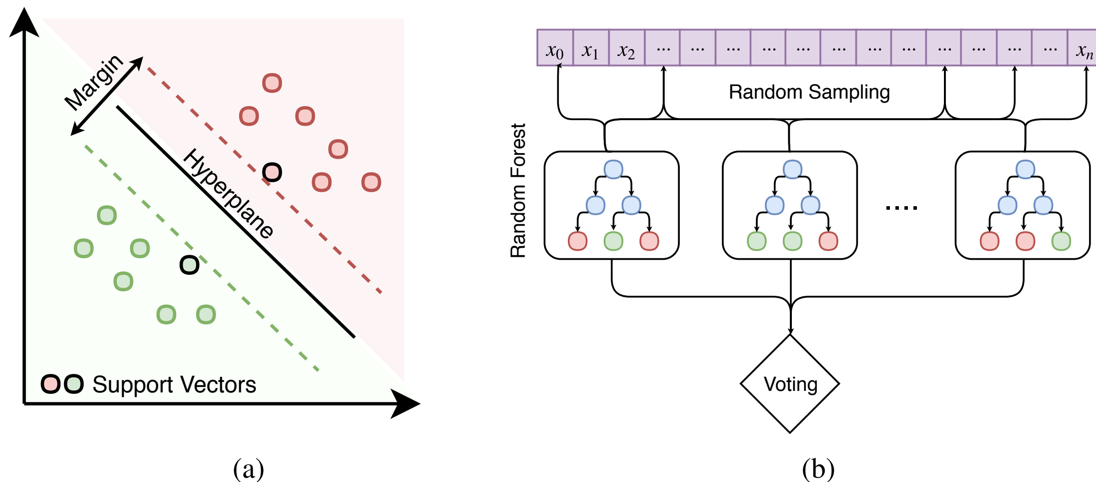


Figure 4.3: Example of a SVM and random forest classifier. (**a**) SVM separating two classes with a hyperplane. Optimal class separation is guaranteed by maximizing margin size. (**b**) Random forest with multiple decision trees being trained on random samples from the training data.

4.3 GIS Data Processing, Image Generation, and Training

Section 4.3.1 details the process of generating an annotated dataset and its random split into distinct training, validation, and testing subsets. Sections 4.3.2 and 4.3.3 outline image generation techniques from LiDAR and satellite data, respectively. Section 4.3.4 details the specific CNN architectures and training procedures, followed by validation assessment. Section 4.3.5 explores CNN feature extraction as input for several chosen classical machine learning algorithms and their associated parameters.

4.3.1 Classified Image Set Generation

Generation of an annotated roof dataset requires three data sources for each building: satellite imagery, airborne LiDAR data, and building outlines with corresponding roof labels (from manual classification). Buildings outlines are used to extract individual roofs from satellite and LiDAR data.

Using building outlines to filter such data sources is a technique used within the GIS community [130, 131]. For example, Ref. [132] used 2D cadastral maps to clip buildings from a DSM for 3D building reconstruction. This clipping step allows for the subsequent generation of images focused on the building of interest and enhances feature extraction.

All three of these data sources must be properly geo-referenced so they can be fused together. Care must be taken to select a geographic area where data sources for all of these items are present. Although OSM provides the necessary building outlines in many geographic regions, the associated roof shape label is most often incomplete. Some geographic regions (e.g., Germany) are more likely to have a denser collection of labeled roof shapes through a higher volunteer involvement. Previous work by the authors relied upon pre-labeled roof shapes provided by the OSM database [115] in Witten, Germany. However, this chapter broadens the categories of classifiable roof shapes as well as sampling from diverse regions including small to large city centers. The authors found that OSM did not provide sufficient pre-labeled buildings, necessitating manual classification of thousands of roof shapes (by the first author). Once the appropriate data sources are found or generated, the methods described below can be employed to generate satellite and LiDAR images for each building in preparation for supervised learning and subsequent use in roof shape classification.

Satellite, LiDAR, and building outline data sources have their own spatial reference systems (SRS). The SRS defines a map projection and determines the transformations needed to convert to a different SRS. These reference systems are uniquely identified though a spatial reference system identifier (SRID) which designates an authority and an identifier. For example, the European Petroleum Survey Group (EPSG) can be used to specify SRIDs. Many map vendors, such as OSM, choose to store building outlines as polygons, with each vertex stored in WGS84 (EPSG:4326). Satellite images from common map vendors (ArcGIS, Bing, and Google) often use WGS84/Pseudo-Mercator (EPSG:3857). LiDAR data are usually stored in a region-specific SRS; for example, data for Witten, Germany uses EPSG:5555. To convert a point stored in one SRS to another, a program specialized in these transformations, such as `proj.4`, must be used [31]. Building polygons are transformed to their LiDAR and satellite counterpart coordinate systems so that the building outlines are consistent.

4.3.2 LiDAR Image Construction

A depth image representation of each building's roof shape is generated from a LiDAR point cloud. However, many outlier points can inadvertently be present during image generation leading to poor quality or misleading images. To attenuate these effects, bulk preprocessing and per-building filtering steps are performed as described below.

4.3.2.1 Bulk Preprocessing

LiDAR point cloud data are often stored and publicly released in an industry-standard LASer file binary format [133]. This specification not only details the storage of the xyz coordinates of each point, but also supports data classification. If the LAS file's ground points have been classified previously, one can filter the ground points from the file to improve image generation. However, if the ground points are not already classified, ground point removal per building can be performed as outlined in Section 4.3.2.2.

Airborne LiDAR point clouds often include points from building wall surfaces that are not of interest for roof shape classification. These points appear as noise around the edges of the generated LiDAR image and can be removed by estimating the normal vectors for each 3D point and removing points that are nearly orthogonal to the unit vector $\hat{\boldsymbol{k}}$ facing up. Normal vectors may be estimated by gathering points in a configurable search radius, r, and then performing a least squares fit to a plane. The authors chose to use the open source *White Box Analysis Tools* for generating normal vectors in bulk [134]. A search radius of one meter was chosen to generate a point normal, $\hat{\boldsymbol{n}}_{\boldsymbol{i}}$ for each point $\hat{\boldsymbol{p}}_{\boldsymbol{i}}$, with points stored that satisfy $\left|\hat{\boldsymbol{n}}_{\boldsymbol{i}} \cdot \hat{\mathbf{k}}\right| > 0.3$. This ensures that only points with normals that are within 72° of $\pm\hat{\mathbf{k}}$ are kept for further use.

4.3.2.2 Individual Building Filtering and Projection

Individual building LiDAR filtering begins by constructing a 2D planar bounding box (BBOX) from a polygon building outline. This BBOX is used first to quickly remove points in the point cloud that are not related to the building of interest. The resulting subset of points is filtered again using the polygon roof outline, resulting in only points encapsulated in the building outline. Points are determined to be within the polygon by employing a ray casting algorithm [135]. At this time, the 3D point cloud may be noisy and contain undesirable points.

Ground points not already removed due to a ground label per Section 4.3.2.1 must now be removed. First, the minimum ground height z_{min} must be identified; this value is specific to the building of interest. Ground height can be determined by applying a buffer to the BBOX ensuring a ground point is within the set and then finding the point with the minimum height. Any point whose z coordinate, $\hat{\boldsymbol{p}}_{\boldsymbol{i,z}}$, less than z_{min} plus a configurable threshold z_{buff} can be considered a ground point and then removed, as shown in Equation (4.1). The authors found $z_{buff} = 2.5$ meters is sufficient to remove most ground points. Note this fractional z_{buff} accounts for sheds, etc. with low height.

$$z_{min} + z_{buff} < \hat{\boldsymbol{p}}_{\boldsymbol{i,z}} \tag{4.1}$$

A final step of filtering will remove stray points often caused by overhanging trees or other interference. This technique relies upon analyzing the distribution of the z-coordinates of each

building's point cloud. This chapter employs median absolute deviation (MAD) to construct a modified `z-score` that measures how deviant each point is from the MAD as in [136]. This method only applies to unimodal distributions; however not all buildings height are distributed as such. For example, there exist complex flat buildings that contain multiple height levels resulting in a multimodal distribution. To distinguish these buildings, the dip test statistic is employed which measures multi-modality in a sample distribution [137]. The test outputs a `p-value` ranging from zero to one, with values 0.10 or less suggesting bimodality with marginal significance [138]. Any building with a `p-value` greater than 0.2 is considered unimodal, and outlier removal is performed as shown in Algorithm 4.1. Results of this filtering technique are shown in Figure 4.4.

Algorithm 4.1: Filtering of Airborne LiDAR Point Cloud

```
   Input  : Collection of 3D points, A
   Output : Filtered 3D point cloud, B
 1 Z = A_z
 2 B = ∅
 3 p-value = diptest(Z)
 4 if p-value ≥ .2 then
 5 |   MAD = median ( |Z_i − median(Z)| )
 6 |   for p in A do
 7 |   |   diff = |p_z − median(Z)|
 8 |   |   z-score = 0.6745 · diff/ MAD
 9 |   |   if z-score ≤ 3.0 then
10 |   |   |   B = B + p
11 |   |   end
12 |   end
13 else
14 |   B = A
15 end
16 return B
```

Once LiDAR point extraction is complete, the points are projected onto a plane, creating a 2D grid that takes the value of each point's height information. The 2D grid world dimensions are the same as the bounding box of the building, with the discrete grid size being the desired square image resolution. Grid points use interpolation of nearest neighbor if no point is available. Afterward, this grid is converted into a grayscale image, where each value is scaled from 0 to 255 with higher values appearing whiter and lower areas darker. Figure 4.4c demonstrates this process. The CNN's used in this chapter require the grayscale LiDAR data be converted to a three-channel RGB image by duplicating the single channel across all three color channels. This final image is referred to as the LiDAR image.

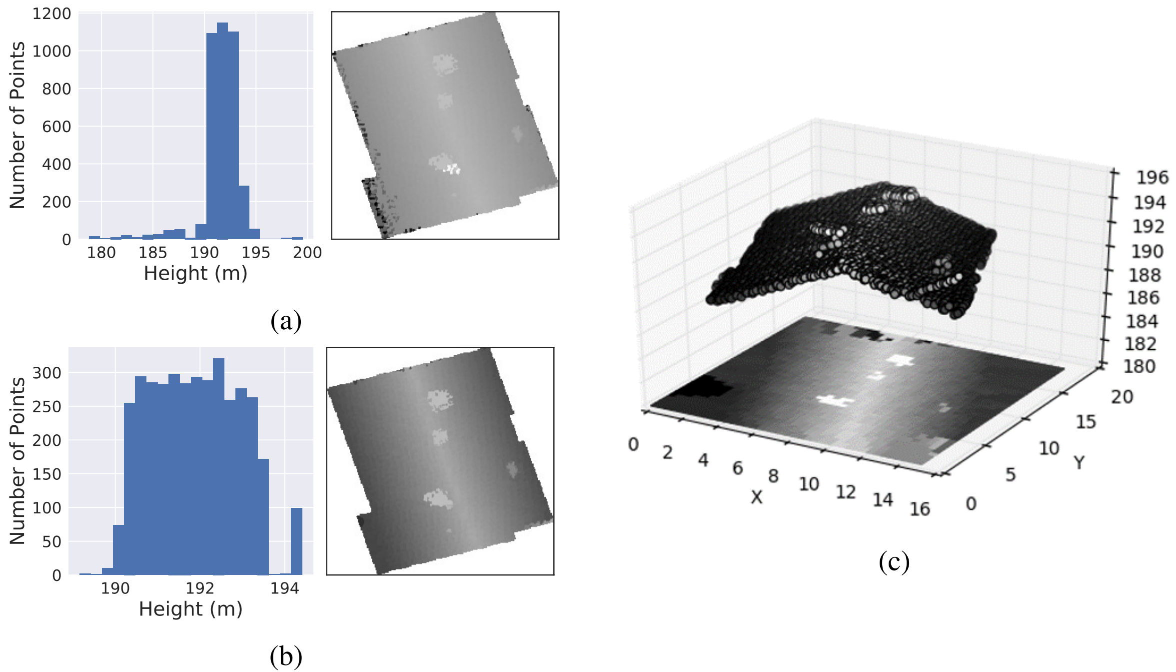
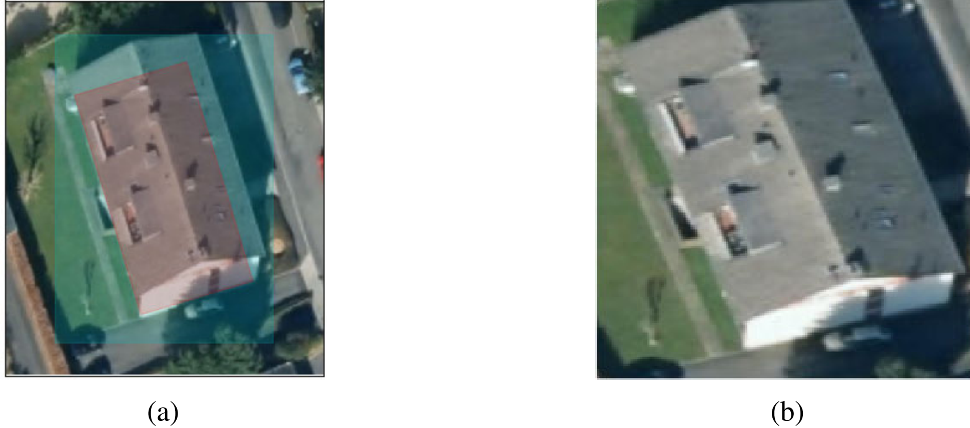


Figure 4.4: Demonstration of LiDAR filtering. LiDAR data of a gabled roof. Histogram of height distribution and generated image (**a**) before filtering and (**b**) after filtering, using median absolute deviation. (**c**) Projection of filtered point cloud.

4.3.3 Satellite Image Construction

It is preferable that the satellite imagery be orthorectified to remove image tilt and relief effects. Ideally, the building polygon can be used to completely stamp out a roof shape image. However, if the aforementioned issues are present in the image, it is unlikely that the polygon will exactly match the building outline in the image. To work around these issues, an enlarged crop can be made around the building. The enlarged crop is produced by generating a buffer around the building polygon by a configurable constant, and then using the bounding box of the new polygon as the identifying stamp. After the image is produced, the image is resized to the square image resolution required by the CNN. The authors found this technique to be necessary only in Witten, while Manhattan and Ann Arbor building outlines were fairly consistent with satellite images. After experimentation, this configurable constant was set to three meters when processing the Witten dataset. Figure 4.5a shows an example original building outline (red shade) overlaid on a satellite image, and the expanded polygon bounding box in cyan. The resulting generated image is shown in Figure 4.5b. This final image is referred to as the RGB image below.



(a) (b)

Figure 4.5: Satellite image processing. (**a**) Witten building outline in red shading overlaid on the satellite image. The enlarged crop area is shown in cyan shading. (**b**) The final generated image, resized.

4.3.4 Stage 1: CNN Architectures and Training

The CNN base architectures chosen for experimentation are Resnet50 [139], Inceptionv3 [140], and Inception-ResNet [141]. All three of these architecture structures are distinct; when trained and tested on ImageNet [127] they received "Top 5" accuracy scores of 92.8%, 93.9%, and 95.3%, respectively. The computational complexity and size of the network increases progressively from Resnet50 to Inceptionv3, with the Inception-ResNet architecture combining the previous architectures to produce a deeper overall network. Each CNN makes use of successive convolutional blocks to generate a final feature map (referred to as the base layers) which are subsequently used by downstream fully-connected layers to make a 1000 categorical prediction (referred to as the top layers). The top layers are domain specific and are not needed for roof classification thus are removed. This chapter applies a global average pooling layer after the final feature layer of each architecture, reducing the convolved feature layers to be used as input into a roof classifying layer. This final classifying layer is composed of an optional fully connected layer (FC1) and a softmax prediction layer as shown in Figure 4.6. A FC1 size of 0 means the fully connected layer is omitted, and the features map directly to the softmax layer. These models are then trained individually on the RGB and LiDAR images.

Training initializes base layer weights with their respective parent architecture. The optimizer chosen for gradient descent is Adam [142] for its ability to effectively adjust learning rate automatically for individual weights; this optimizer is kept consistent for all architectures and training sessions with learning rate initialized at 0.001. The option of freezing initial layers is exploited with a variable number of frozen layers chosen. When Layer 11 is said to be frozen, this means all previous layers, (Layers 1–11), are frozen during training. All base architectures and tested hyperparameters are shown in Table 4.1.

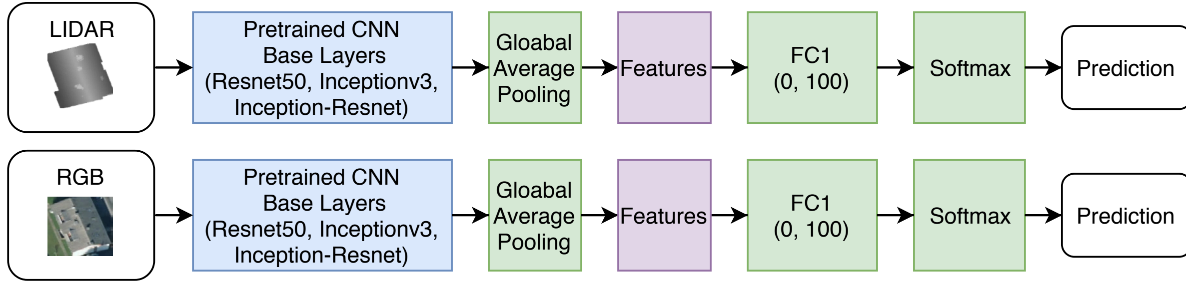


Figure 4.6: CNN architecture templates.

Table 4.1: CNN architectures and hyperparameters

Base CNN Model	FC1 Size	Frozen Layers
Resnet50	0, 100	50, 80
Inceptionv3	0, 100	11, 18, 41
Inception-Resnet	0, 100	11, 18, 41

Keras [143], a high-level neural network API written in Python, is used to import the pretrained CNN models and construct the new architectures discussed above. A maximum of 1000 epochs are run during the training process, while early stopping is employed at the end of each epoch. Early stopping is a technique where after each epoch, the model is run against the validation set and accuracy metrics are reported. If validation accuracy is not improved after seven epochs, training is halted. This ensures that the the model does not needlessly overfit the training data, and the most generalized model is saved. Data augmentation is performed randomly with horizontal and vertical image flips as well as rotations ranging from $0°–45°$.

After training is complete on all CNN architectures and hyperparameters, the best performing CNN with respect to the validation set accuracy for both LiDAR and RGB images is selected for further use. Another training session is performed to determine if region-specific training improves region model accuracy, i.e., whether a model that is trained with data in a specific region (city) will be better at predicting roof shapes in that region compared to a model trained on more diverse data. In this study, model architecture is held constant; only training data quantity and diversity are manipulated.

4.3.5 Stage 2: SVM and Random Forest Classifier Training

The best CNN models are used to extract high level image features as input to a downstream "Stage 2" classifier. This step determines if improved results can be obtained by combining both classical and deep learning models together, as shown in Figure 4.7. In this scenario, only the layers

up to global average pooling are used to generate a condensed feature map for each image in the dataset. The augmented training set images are *reduced* to this small feature vector and are used to train both sets of classifiers (SVM and random forest) over a variety of configurations, as shown in Table 4.2. The Python machine learning library `Scikit-learn` is used to train and validate the models [144]. The final model is chosen which holds the highest test score accuracy.

Table 4.2: SVM and random forest training configurations

Classifier	Parameters
SVM	Regularization Constant (C): 1, 10, 100
	Kernel: linear, rbf, poly, sigmoid
Random Forest	Criterion: gini, entropy
	Number of Estimators: 5, 10, 50
	Max Depth: 5, 10, 50

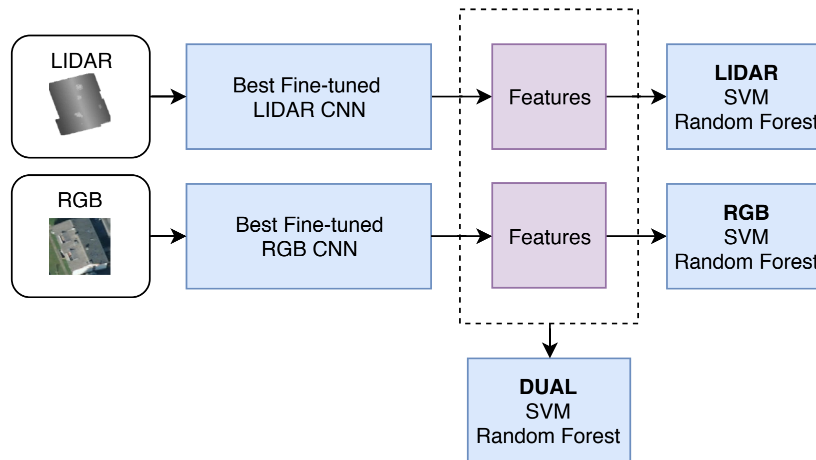


Figure 4.7: Feature extraction for use in SVM and random forest model training. The "dual" model refers to both LiDAR and RGB features being combined as input for model training and prediction.

4.4 Results

4.4.1 Case Study Dataset Generation

This section outlines the data sources of several cities used to generate images of building rooftops for this chapter's case studies. Procedures for manually labeling images are discussed, and a complete breakdown of labeled roof categories is presented. Example images are shown for each category along with explanations of training, validation, and testing datasets.

4.4.1.1 Data Sources

The geographic regions used in the following case studies are chosen to maximize diversity in roof shape architectural examples. Diversity within each class translates to image differences such as colors and outline shapes for roofs. Data from the cities of Witten, Germany; the Manhattan borough of New York City, New York; and Ann Arbor, Michigan are used to generate case study data. Witten represents a small urban city with minimal high rise buildings and numerous single-family residential buildings, whereas Manhattan represents a sprawling metropolis with a diverse range of flat-like building roofs with structural additions to the rooftops (antennas, water towers, air conditioning units, etc.). Ann Arbor, used only as an independent test set, includes a combination of building architectures found in Witten and Manhattan. Each of these cities provide publicly available high resolution satellite images, LiDAR data, and building outlines per Table 4.3. Building sampling was random in the downtown districts of Ann Arbor and Manhattan, while Witten was sampled uniformly over the entire city.

Table 4.3: Satellite, LiDAR, and building data sources

City	Satellite		LiDAR		Buildings
	Provider	**Resolution**	**Provider**	**Spacing**	**Provider**
Witten	Land NRW [100]	0.10 m/px	Open NRW [99]	0.30 m	OSM [114]
New York	NY State [145]	0.15 m/px	USGS [146]	0.70 m	NYC Open Data [147]
Ann Arbor	Bing [148]	0.15 m/px	USGS [149]	0.53 m	OSM [114]

4.4.1.2 Image Generation and Labeling

Using the methods described in Section 4.3, RGB and LiDAR images are generated for each building roof in all cities and then randomly downsampled. All data are treated as unlabeled, requiring manual classification by the authors. One of eight roof shape labels can be assigned to each image: `unknown`, `complex-flat`, `flat`, `gabled`, `half-hipped`, `hipped`, `pyramidal`, and `skillion (shed)`. This set was determined by observing the most abundant roof architectures present in Witten and Manhattan and merging them together. `Unknown` is a catch-all category used to account for roof shapes outside the other seven, often labeled `complex` in other literature [122, 125]. Additionally, poor quality images unsuitable for roof prediction are also marked `unknown`. A `complex-flat` roof differs from a `flat` roof in the significance of obstructions on the surface of the roof, or if there are multiple height layers. A `flat` roof should have minimal objects and a near homogeneous height profile, while a `complex-flat` roof may have additional items such as water towers or superstructures but still contain sufficient flat structure, e.g., for a

safe small UAS landing. This distinction is more apparent in Manhattan than Witten; separating these categories is beneficial to provide class diversity in an otherwise architecturally binary dataset. Practically all roofs in Manhattan are either `flat-like` or classified as `unknown`. Examples of RGB and LiDAR images for the seven classes of roof shapes are shown in Figure 4.8 while examples of the `unknown` class are found in Figure 4.9.

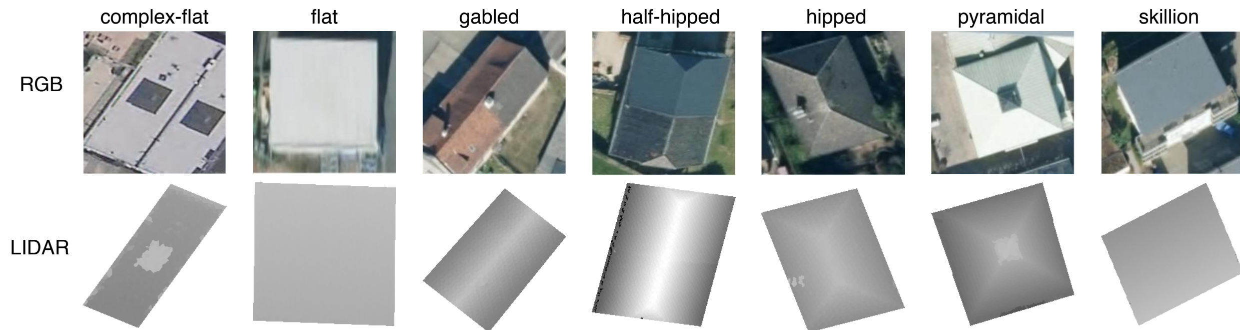


Figure 4.8: RGB and LiDAR example images of roof shapes.

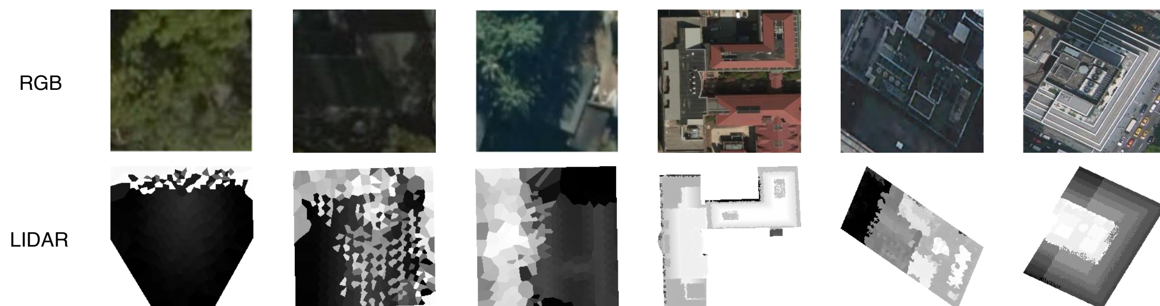


Figure 4.9: RGB and LiDAR example images classified as `unknown`. This category includes buildings with poor quality images as well as complex roof structures.

LiDAR and satellite images may in some cases be labeled differently. For example, a building with an actual gabled roof may have a LiDAR image which is malformed leading to an `unknown` class label, while the RGB image may be clear leading to a `gabled` label. These differences must be noted to prevent models from being trained on incorrect classifications; we want the LiDAR model to learn that the LiDAR image is poor and that an `unknown` classification should be given while the RGB model should learn the true label. When label differences occur, both labels are kept for training and model validation, leading to *differences* between the LiDAR and RGB **training** and **validation** datasets. However, the **test** dataset *do not* have these label difference between modalities; the test set instead marks every image with the true building label. This ensures that the test set presents an accurate prediction of results with slightly lower classifier accuracy than validation datasets. If both modality images are poor, then the true label is `unknown` because no prediction is possible.

A final rare case exists where one modality is clear and correctly labeled but the other modality is *misleading* with an incorrect label. This occurs especially in LiDAR images of `half-hipped` buildings appearing as though they are `gabled`. There is often only a subtle difference between the two classes, a small triangular dip near the edge of the building, that may not be captured fully in the LiDAR image. When this occurs, the LiDAR image is removed from the training/validation set because one does not want to train on an image that will give inaccurate results. However, the test dataset is left intact. In all cases, the test dataset holds the true roof label based on manual classification, and performance of all machine learning models is assessed in comparison to predicting the true label.

Models that require both input modalities for prediction must have a single label reference for training. If a conflict exists between the two image labels, then the true label is used as was done in the test dataset. This is beneficial as it forces the model to learn to rely on another modality when one input is known to be incorrect. A complete breakdown of the annotated dataset by city is in Table 4.4. Witten and Manhattan data are combined together and divided into training, validation, and testing data in a 60/20/20 random split. The Ann Arbor data are used only as a secondary test set to determine generalizability of the model and results.

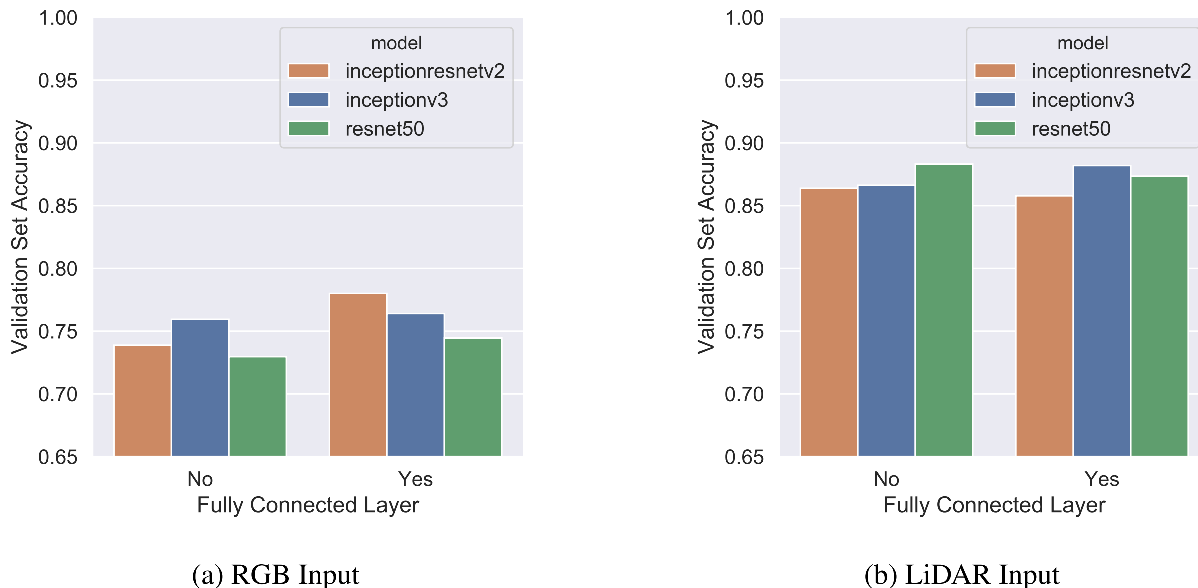
Table 4.4: Breakdown of roof labels by city

Roof Shape	**Witten**	**Manhattan**	**Ann Arbor**
unknown	133	792	14
complex-flat	125	785	37
flat	454	129	24
gabled	572	7	96
half-hipped	436	0	3
hipped	591	3	20
pyramidal	110	0	0
skillion	189	0	2
Total	2610	1716	196
Removed	212	65	0

Note that some data were removed from each city because of discrepancies between satellite and LiDAR data resulting from the time the data were recorded. For example, a newly constructed neighborhood in Witten has newer satellite images capturing the neighborhood while old LiDAR data show a flat undeveloped area. This situation was attenuated in Manhattan by looking at building construction dates and only using buildings whose date of construction is before the creation of earliest data sources. However, this information was not able to be found for Witten leading to a much higher removal rate. Overall, about 5.6% of the data were manually discarded for Dataset 1 (Witten and Manhattan). No buildings were removed from the Ann Arbor dataset used for testing.

4.4.2 CNN Training and Results

All training was performed on the University of Michigan Flux system, providing a server with a minimum of six gigabytes of RAM, two CPU cores, and a single NVIDIA Tesla K40. The training and validation was performed only on Dataset 1, the combination of the Manhattan and Witten data. Figure 4.10a plots validation set accuracy for the best-performing CNN models with RGB input, while Figure 4.10b displays results for LiDAR input. The horizontal axis of both figures indicates whether the network uses a fully connected layer after features are extracted from each CNN. Consistent with previous research, accuracy results are substantially higher (∼10%) using LiDAR data versus RGB data. The best performing network for RGB input is `Inception-Resnet` with a fully connected layer providing a validation set accuracy of 78.0%. Accuracy appears to increase for RGB input models with increasing CNN model complexity as well as the addition of a fully connected layer.



(a) RGB Input

(b) LiDAR Input

Figure 4.10: Results of CNN networks on validation set. Colors indicate the base model used while the horizontal axis specifies whether a fully connected layer is part of the architecture. (**a**) RGB (satellite) image input and (**b**) LiDAR image input.

The best performing model for LiDAR input was Resnet50 with a validation set accuracy of 88.3%, which narrowly outpaced Inceptionv3 with a score of 88.1%. The accuracy differences are statistically insignificant, however the difference in model complexity in terms of memory and computations is significant. Resnet50 is approximately 50% smaller in amount of floating-point operations and took 36 min to train versus the 81 min Inceptionv3 required [150]. In fact, all models performed similarly, and the addition of a fully connected layer (adding more complexity) provided marginal benefit for accuracy. All these factors indicate that a simpler model is desirable for LiDAR

input. Intuitively, the complex nature of satellite RGB images would necessitate a deeper network to extract useful features, while the more simplistic LiDAR images would require a less complicated model. The final model architectures chosen are displayed in Table 4.5 along with their training parameters.

Table 4.5: Best CNN model architectures

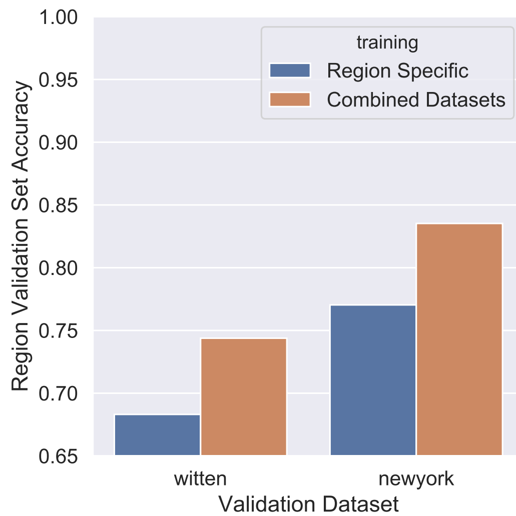
Input	**Base Model**	**FC Layer?**	**Frozen Layers**
RGB	Inception-Resnet	Yes	11
LiDAR	Resnet50	No	80

Using the best performing models, as shown in Table 4.5, another region-specific training session was performed. Concretely, the training and validation datasets are separated by region, one for Witten and one for Manhattan (New York), and the same architectures are retrained on this subset of the original combined data. Figure 4.11 shows the results of comparing these new region-specific models to the previous combined models. Accuracy results are significantly higher for RGB input by using the model trained on the combined dataset, clearly demonstrating the benefits of data quantity and diversity. However, LiDAR input has mixed results, with Witten performing better with additional data and Manhattan performing worse. It is possible that the limited amount of class diversity in the Manhattan dataset has not benefited by the diverse architectural examples Witten provides. However, the results as a whole indicate that the models trained on the combined dataset are overall more accurate and should be chosen for use in new cities.

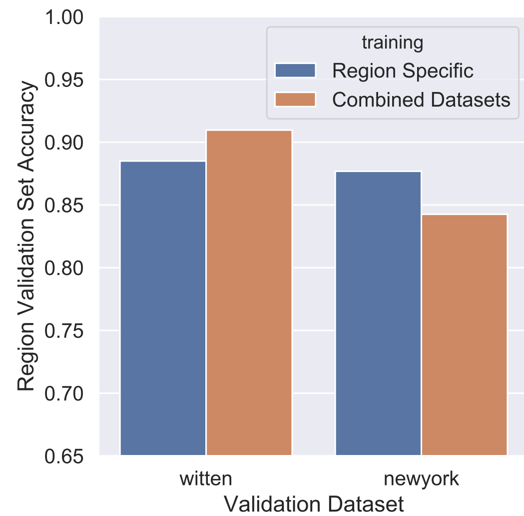
4.4.3 Feature Extraction for SVM and Random Forest Training

Training set images from Manhattan and Witten have their salient features extracted using the trained models in Table 4.5. These features come after the global average pooling layer and are vector sizes of 1536 and 2048 for `Inception-Resnet` and `Resnet`, respectively. This new high level feature training set is then fed to SVM and random forest classifiers with varied configurations for training as specified previously in Table 4.2. Once all classifiers are trained, they are run against Test Set 1 (Witten and Manhattan). Results are shown in the Figure 4.12 swarm plot where each dot represents a model trained at a different configuration; input modality is determined by its placement on the horizontal axis. The color represents base model type, and CNN accuracies are also shown for comparison. The y-axis is configured to begin at 45% accuracy, truncating low accuracy outlier models. There are six outliers *not* shown which are all SVM models using a polynomial kernel.

As before we see an increase in model accuracy using LiDAR data in comparison to only RGB, and even higher accuracy is achieved by combining the features into a "dual" input classifier.



(a) RGB Input



(b) LiDAR Input

Figure 4.11: Accuracy between region-specific and combined training datasets. Results labeled "combined dataset" are trained on images from both Witten and Manhattan. Validation set accuracy on the vertical axis is specific to the region indicated on the horizontal axis. (**a**) RGB (satellite) image input and (**b**) LiDAR image input.

Focusing on RGB input, the best classifiers are all random forest, with the top classifier achieving 73.3% accuracy. This result scores higher than CNN accuracy, underscoring the strengths of random forests for generalized classification. In this instance, the random forest was configured with 50 maximum estimators, an entropy split, and a maximum depth of 10.

LiDAR models score significantly higher, with both SVM and random forest models achieving similar top accuracies of 84.8% versus 84.4%, respectively. This top scoring SVM is configured to use a radial basis function (rbf) kernel with a regularization constant of 10, while the random forest is the same configuration that scored highest for RGB input. Once again, these classical machine learning algorithms outperformed the CNN network in classification on the reduced feature set.

The dual input results validate previous research in that combining multiple streams of modality data can lead to greater accuracy than use of either data type individually. The top classifier is once again a random forest with the same configuration previously discussed; this configuration performs consistently well in all classification tasks. Overall, an improvement of 2.4% is observed by fusing features together resulting in an accuracy of 87.2%. Table 4.6 shows the top model classifiers and associated parameters for each modality. The authors chose the "dual" input random forest classifier for the final analyses described below.

Three SVM model outliers can be seen for all three inputs. The RGB outlier model used a sigmoid kernel, while the LiDAR and dual input model outliers used a polynomial kernel. No

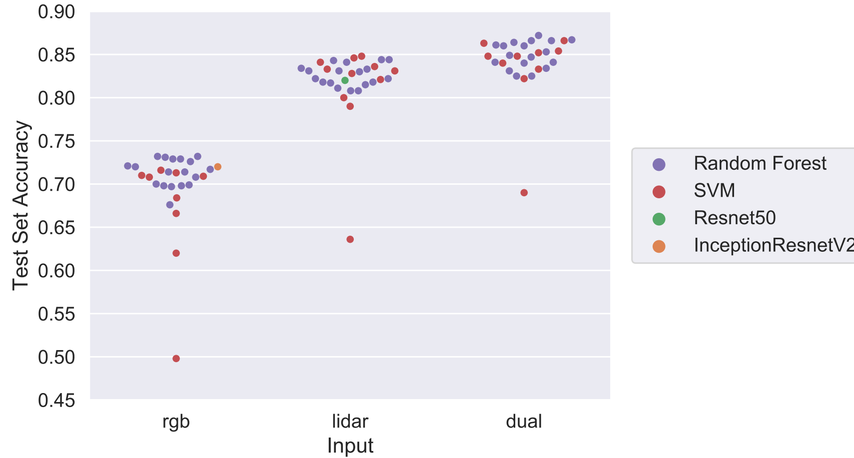


Figure 4.12: Test Set 1 Accuracy (Witten/Manhattan). Comparison of using CNN feature extraction coupled with SVM and random forest classifiers.

random forest model provided a low test accuracy to be considered an outlier.

Table 4.6: Best Classifiers using CNN extracted features

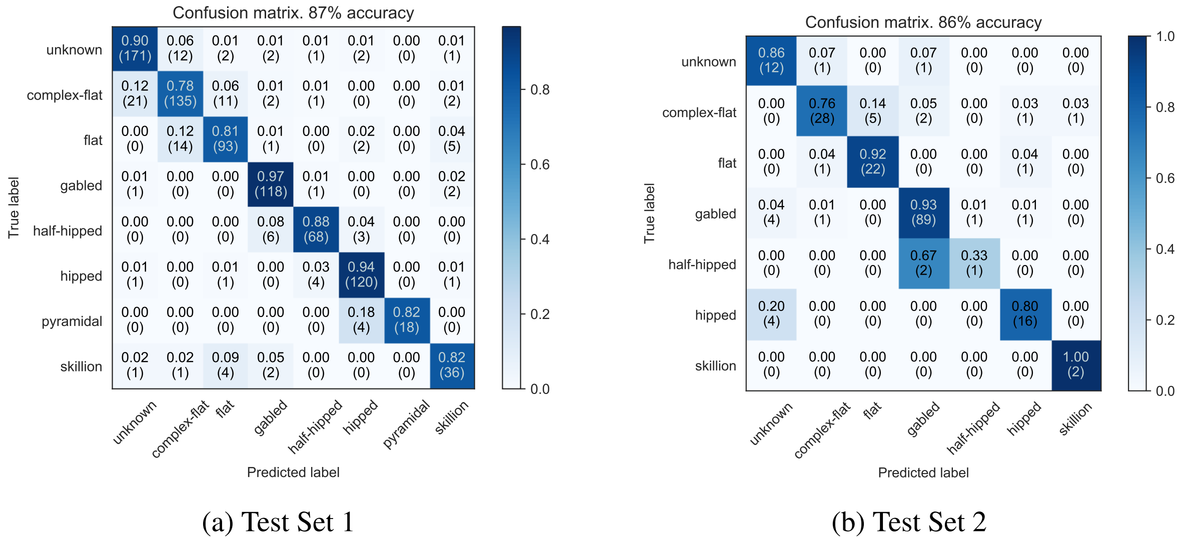
Input	**Model**	**Parameters**	**Test Set 1 Accuracy**
RGB	Random Forest	Criteria: Entropy, # Estimators: 50, Max Depth: 10	73.2%
LiDAR	SVM	Regularization Coefficient: 10, kernel: rbf	84.8%
Dual	Random Forest	Criteria: Entropy, #Estimators: 50, Max Depth: 10	87.2%

4.4.4 Analysis of Final Dual Input Model

Section 4.4.4.1 provides analysis for the final dual input random forest model by generating confusion matrices for both Test Set 1 and Test Set 2. Section 4.4.4.2 aggregates flat-like classes for the UAS emergency landing application and evaluates the tradeoff between precision and recall through confidence thresholding.

4.4.4.1 Confusion Matrices

The total accuracy for Test Set 1 (Witten and Manhattan) is 87.2%, while Test Set 2 (Ann Arbor) scored 86.7%. The final dual input model's confusion matrices for Test Set 1 and Test Set 2 are shown in Figure 4.13a,b, respectively. The row-wise percentage of each cell is computed and color coded along with the specific quantity classified in parentheses underneath. We can see that for both test sets one of the largest errors comes from the confusion between `complex-flat` and `flat` roofs. The authors found difficulty in labeling some flat-like roof examples, especially ones that bear traits of both classes; it is clear this confusion carried over into the trained model. In



(a) Test Set 1 (b) Test Set 2

Figure 4.13: Confusion Matrices for Test Set 1 (Witten/Manhattan) and Test Set 2 (Ann Arbor).

some cases, a roof is on the threshold of being `flat` or `complex-flat`, and this ambiguity makes it difficult to provide a consistent "correct" answer. Indeed, this case often applies between the `complex-flat` and `unknown` labels as well: When does a `complex-flat` roof become too complex to support a safe small UAS landing? The authors attempted to be consistent in answering this question when labeling data, however edge cases were observed. Table 4.7 and 4.8 list results for recall (completeness), precision (correctness), and quality for Test Set 1 and Test Set 2, respectively. Note that there were no `pyramidal` roofs shapes in the Ann Arbor test set and too few `half-hipped` and `skillion` roofs to calculate valid metric results.

Table 4.7: Results for recall, precision, and quality evaluation metrics for Test Set 1

Type	**Recall**	**Precision**	**Quality**
Unknown	0.90	0.88	0.80
Complex-Flat	0.79	0.83	0.68
Flat	0.81	0.84	0.70
Gabled	0.97	0.90	0.87
Half-Hipped	0.88	0.91	0.81
Hipped	0.95	0.92	0.87
Pyramidal	0.82	1.00	0.82
Skillion	0.82	0.77	0.66

Table 4.8: Results for recall, precision, and quality evaluation metrics for Test Set 2

Type	Recall	Precision	Quality
Unknown	0.86	0.60	0.55
Complex-Flat	0.76	0.90	0.70
Flat	0.92	0.82	0.76
Gabled	0.93	0.96	0.88
Half-Hipped	N/A	N/A	N/A
Hipped	0.80	0.84	0.70
Pyramidal	N/A	N/A	N/A
Skillion	N/A	N/A	N/A

4.4.4.2 Confidence Thresholding

With every model prediction, there is a probability distribution of the likelihood the example belongs to a class. The class with the highest probability is then chosen as the final prediction. Model precision can be increased by adjusting the confidence threshold a model requires to make a prediction, and, if not met, the example is marked `unknown`. This will generally decrease the number of false positives at the expense of an increase in false negatives. For the UAS emergency landing use case, operators need confidence that a roof labeled as "flat-like" is actually flat. We use confidence thresholding to combine `complex-flat` and `flat` roofs into one `flat-like` category used for UAS roof identification. Figure 4.14 shows individual graphs of how the model's predictive power on Test Set 1 is impacted as the confidence threshold is manipulated. This process is repeated on Test Set 2 in Figure 4.15, with `half-hipped`, `skillion`, and `pyramdial` classes omitted due to lack of examples.

We can clearly see the inverse relationship between precision and recall as the required confidence threshold is increased. Unfortunately, this relationship is clearly not linear for all classes; moderate increases in precision come at a large decrease in recall for `flat-like`, `gabled`, `half-hipped`, and `hipped` classes. Indeed, as precision increases above 95% recall drops exponentially to around 60%. These figures certainly show there is a limit to the effectiveness of confidence thresholding; setting too high a confidence threshold may even lead to a *drop* in precision in some cases as seen for the `hipped` class in Figure 4.14. However, these results show promise that class-specific thresholds can be set to ensure high-precision predictions are generated. For UAS landing, these results indicate we can achieve near-perfect precision at the expense of only finding $\sim$60% of the flat roofs within a region.

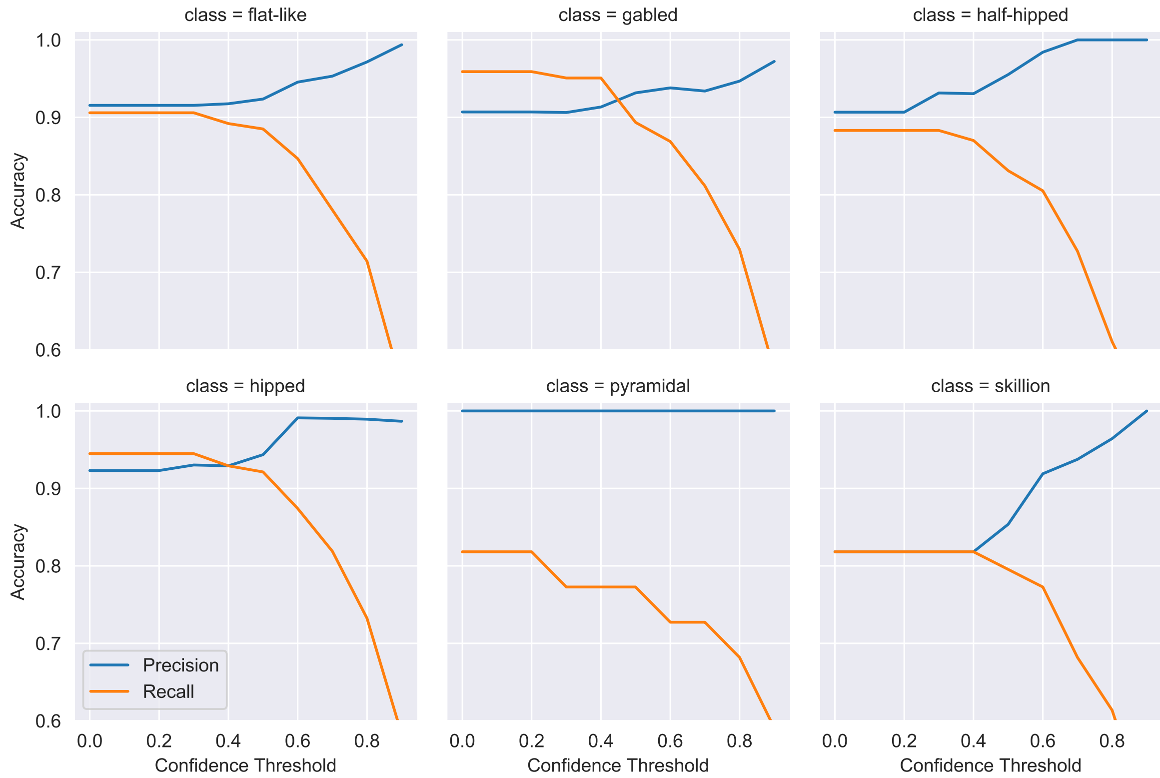


Figure 4.14: Test Set 1 confidence threshold impact on precision and recall for multiple classes.

4.5 Discussion and Future Work

The presented study demonstrates that a combination of a Stage 1 CNN feature extractor coupled with a Stage 2 random forest classifier can reliably and effectively label city wide roof shapes with publicly available GIS data. In addition, we show good generalization of our final model on diverse city landscapes ranging from small to large urban centers. Two independent test sets show similar results in model quality metrics providing a realistic expectation of model performance, where one set, Ann Arbor, was not used in training. Others have successfully performed roof shape classification through machine learning, but no previous work to-date has demonstrated effectiveness to the scale analyzed here in both breadth and depth. Over 9000 images (two for each building) have been manually labeled from three diverse cities to generate the training, validation, and test sets. In comparison, the largest labeled dataset the authors found in the literature for roof top classification is 3000 images and encompasses only one city [126].

A comparison of our accuracy results with other work is difficult because no benchmark test set has been available to date for roof shape classification. Benchmarking datasets are of critical importance to compare the results of applying different algorithms [151]. Since no such benchmarking data exist for roof shape classification, the authors propose this chapter's released

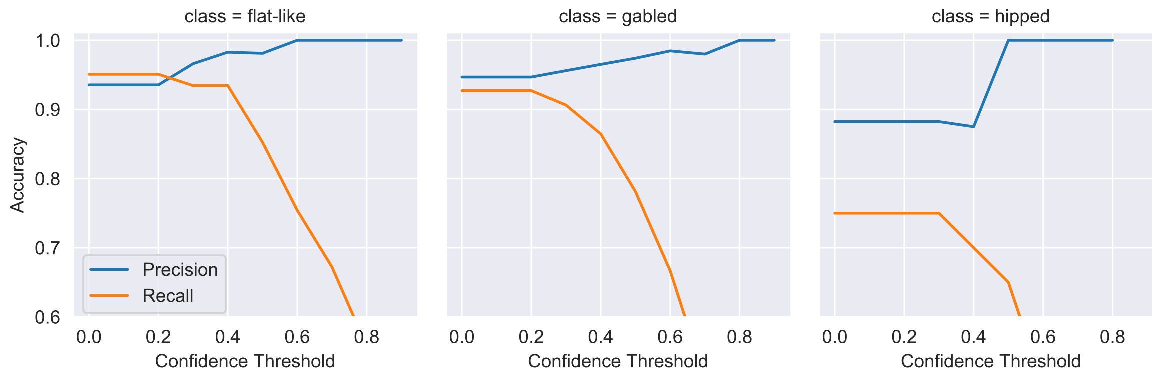


Figure 4.15: Test Set 2 confidence threshold impact on precision and recall for multiple classes.

annotated dataset serve as an initial dataset for future roof shape classification research.

The challenge of comparing algorithms is compounded by differences in expected model input. Many models preprocess LiDAR input into handcrafted features, such as slope, aspect, and number of roof surfaces [123, 122]. Others rely on a raw DSM image of a roof, while our work relies upon automatically generating a depth image from point clouds specifically filtered for each building roof. Our work is one of the few that relies upon both satellite images and LiDAR data for classification, and is the only one that uses deep learning to train on both modalities together to enhance model accuracy. In addition, our work classifies eight roof categories, naturally bringing down accuracy results in comparison to most others works attempting to classify six or at most seven roof shapes.

The largest weakness in this study comes from one of its greatest strengths: the fusion of LiDAR and satellite input is only effective if both data sources observe the same thing. If one modality sees a newly constructed neighborhood and the other sees undeveloped area, for example, the model will become confused. The authors attempted to mitigate this issue by looking at construction dates for buildings, and removing buildings constructed during/after the earliest data source. However, this construction information is difficult to obtain in all cities/countries, and does not guarantee the removal of all possible data source inconsistencies. Future work is needed to automatically detect inconsistent datasets if present and automatically label the roof as `unknown`. Note that inconsistent datasets are immediately apparent to the human eye.

As the authors have continually refined the LiDAR pre/post-processing methods for depth image generation, they have concluded that an alternative method may be more suitable. Instead of painstakingly converting point clouds to high quality depth images for a CNN, it should theoretically be better to operate directly on the point cloud itself in a deep learning model. Several advances have been proposed in deep learning for both point cloud segmentation and classification, e.g., PointNet and SpiderCNN [152, 153]. These neural network architectures sample from the point cloud and directly learn global and local geometric features of the point cloud surface. These methods have

been shown to be successful in small scale object classification (household items, pedestrians, etc.) using high resolution LiDAR data; future work should investigate their use on airborne LiDAR data.

Small UAS rooftop landing requires a high degree of confidence that a flat-like surface exists for safe landing. This chapter demonstrates that flat-like roofs can be be reliably predicted with high precision by adjusting the final model's confidence threshold. After flat-like roofs have been identified, further post processing may be performed to quantify metrics such as ideal landing position, surface roughness, and rooftop geometry. The output of this future work can then reliably generate a database of emergency landing sites that is risk-aware.

4.6 Conclusions

Building outline and height information is useful for visualization and 3D reconstruction but roof shape is often missing or at best incomplete in existing databases. GIS data such as satellite images, LiDAR point clouds, and building outlines are often available. This chapter processes these data to construct individual image representations of depth and color of roof shapes. Datasets are constructed and manually labeled across multiple cities. The final model uses deep learning for feature extraction and a random forest algorithm for subsequent roof shape classification. Two test sets from diverse cities show good generalization of the trained model, reporting total accuracies near 87%. Confidence thresholds are manipulated leading to greater than 98% precision in labeling `flat-like` roofs in all three tested cities, an important increase in precision for applications such as UAS rooftop landing. The generalized models and test datasets show promise for applying machine learning to automatically label roof shapes around the world with high confidence.

CHAPTER 5

Map-Based Planning for Small UAS Rooftop Landing

5.1 Introduction

A primary safety concern for UAS is ensuring a robust emergency landing capability [1, 2]. Emergency landing requires landing site selection, trajectory planning, and stable flight control to actually reach the selected site [3]. It is possible a UAS may identify a safe site within sensor range allowing for an immediate landing. However, when no safe site is within range the UAS must devote time and energy to exploring sites beyond sensor range or else utilize pre-processed data to identify a safe site [4, 5]. An onboard database of maps including landing sites can be incorporated into an efficient autonomous decision making framework. For example, Refs. [3, 34] and [6] utilize airborne flight risk models to build emergency landing plans for fixed-wing and urban flight operations, respectively. We call such a decision making framework a map-based planner.

Urban areas typically do not offer classic emergency landing sites such as unpopulated open fields. This requires a planner to consider unconventional yet safe alternatives. We propose flat building rooftops as viable urgent landing sites for small UAS. These UAS will likely operate at low altitudes and at times even in urban canyons. During landing site selection, a map-based planner must be able to assess *landing site risk* posed to the aircraft and bystanders at touchdown. The UAS may pose risk to people and property it overflies enroute, so the planner must assess *path risk* once the landing flight plan is known. Landing site and path risk together offer an estimate of *total risk*.

Map data used for emergency landing planning must have high integrity and low-latency access to support timely decision making. This chapter proposes an offline data processing pipeline and online multi-objective, multi-goal landing planner that enable a UAS to minimize total risk when a nearby emergency landing is required. Landing site and local area map information is pre-processed and stored onboard. The multi-goal onboard emergency landing planner explores available ground and rooftop landing sites likely to minimize overall total risk while greedily pruning options known to have higher risk. Pareto fronts over landing site and path risk are generated for three urban regions: Witten, Germany; Ann Arbor, Michigan; and mid-town Manhattan in New York City. We statistically analyze results with respect to availability and quality of landing sites as well as

execution time of the map-based planner.

The first contribution of this chapter is a novel method to identify flat rooftop surfaces from airborne LIDAR data to identify the largest clear small UAS landing location on each flat roof. Risk metrics are quantified from offline construction of a database using public data sources. A second contribution is our proposed approach to model and optimize plans over a combination of landing site and path risk metrics. Our third contribution is a multi-goal onboard planner that guarantees a risk-optimal solution is found rapidly by avoiding exploration of high-risk options. The proposed emergency planning framework enables a UAS to select an emergency landing site and corresponding flight plan with minimum total risk.

The chapter is structured as follows. Section 5.2 provides background in emergency landing planning and multi-goal path planning. Section 5.3 reviews preliminaries in data sources and planning. Section 5.4 discusses offline construction of a risk-aware landing site database. Section 5.5 outlines methods for generating occupancy and risk maps used for 3D path planning, while Section 5.6 summarizes our map-based planning approach. Section 5.7 presents case studies with focus on analysis of trade-offs between landing site and path risk and statistics on required planning time. Section 5.9 presents conclusions and future work.

5.2 Background

Emergency landing planning is typically accomplished with either onboard sensor-based planning or map-based planning [9, 4]. Sensor-based planners rely strictly on real-time data streams while map-based planner use information previously gathered and stored onboard. Planners using a combination of maps and onboard sensors can capitalize on both data sources [4]. Section 5.2.1 outlines sensor-based planning while an overview of map-based planning is presented in Section 5.2.2. Sec 5.2.3 provides background on multi-goal planning which our map-based planner utilizes. A meta-level framework to unify sensor and map-based planning has been proposed for general (fixed-wind) aviation [3] as well as multicopter UAS [4] as shown in Fig. 5.1. This planner relies on local sensor data to land if the immediate area is safe but calls a map-based planner otherwise. Sensor and map based methods can be merged to capitalize on each other's strengths. Previous work all focuses on open landscapes and/or runways and considers only a single landing site in each planning epoch. We propose map-based planning with consideration of rooftop landing sites and dual risks from both the landing site and its path for the first time in this chapter. Previous research in sensor-based planning, map-based planning, and multi-goal planning is summarized below.

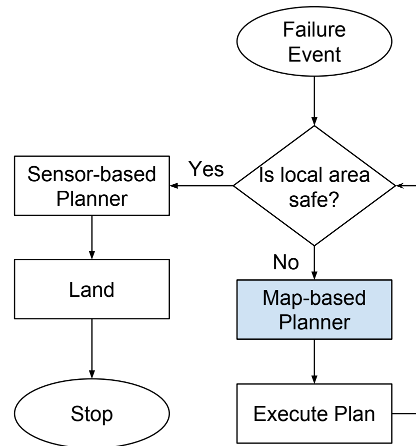


Figure 5.1: Emergency planning logic. Reprinted from Ref. [4], originally published open access under a CC-BY 4.0 license. https://link.springer.com/article/10.1007/s10846-016-0370-z

5.2.1 Sensor Based Planning

Onboard exteroceptive sensors including camera, radar, and LiDAR can provide a wealth of information about the surrounding environment for use in emergency landing planning. Ref. [9] uses downward facing camera data to identify and characterize possible landing sites according to size, shape, slope, and nearby obstacles. Ref. [17] provides methods and experimental results of autonomous local landing using video and LiDAR data. Ref. [15] specifically identifies candidate landing sites on rooftops using a single camera, while Ref. [14] identifies terrain-based landing sites in an image plane from 2D probabilistic elevation maps generated over terrain. In all cases landing site identification is only possible within the sensor field of view.

5.2.2 Map-Based Planning

There are two main approaches in generating landing sites for use in map-based UAS emergency landing: one producing a specific landing site database and the other generating a risk grid from which landing sites can be selected. Both approaches, sometimes used together, rely on similar data sources such as census records, DSM and map vector data but differ in output representations. A georeferenced database contains vector geometries of landing sites with associated meta data used for risk evaluation. A risk grid is a two or three dimensional data structure where each cell refers to the risk of a specific location on Earth.

Ref. [24] uses data such as census records, OpenStreetMap (OSM), and mobile phone records to quantify risk for unmanned aircraft requiring an emergency landing. Risk factors include risk to the vehicle, human population, property, and an area risk to assess the quality of a landing site. Landing sites such as open fields, grasslands, and highways are identified, risk evaluated, and stored

in a landing site database. The proposed planning architecture uses this onboard database to select a risk-optimal landing site within a feasible flight footprint. After a landing site is chosen, path planning is performed at a constant altitude, assessing risk to people through a fusion of census data and mobile phone activity.

Ref. [4] proposes the use of three dimensional risk grids to identify landing sites and perform path planning for an energy-constrained multicopter in urban environments. A 3D occupancy grid to represent risk is generated as a combination of terrain, population, and obstacle costs. Terrain risk is evaluated using slope as well as terrain type, census data is used to determine population risk, and property risk is assigned equally to all buildings. A linear combination of these risks is used to generate a final 3D grid for flight planning. In this grid a landing site is a *terrain* cell that has a lower cost than a configurable threshold. All landing sites are treated equally, meaning the first landing site found by the planner is the "optimal" choice returned.

Ref. [154] proposes generation of a 2D risk map that quantifies risk to population on the ground. The map is created taking into account an aircraft's model parameters and the local environment conditions while considering their uncertainties. The risk map is defined for a specific altitude and is the combination of several risk layers including population density, obstacles, sheltering, and no fly zones.

We propose an emergency landing framework that uses both a landing site database and a 3D risk grid to evaluate landing site risk and path risk as independent metrics. Our work is focused on Vertical Take-Off and Landing (VTOL) UAS that might require an urgent landing but still have sufficient flight control to stably follow a prescribed path to touchdown. We provide a multi-goal planner to trade off risk of landing sites with risk-optimal paths to each site while efficiently finding the minimum total risk solution. We uniquely identify usable area on flat rooftops using machine learning and computational geometry techniques. Previous efforts to quantify an *area* risk of a landing site approximated risk as length or area of a site's buffered geometry [24]. However not all area in a landing site is suitable for landing. For example, many rooftops have obstacles such as air conditioning units, vents, and rooftop entrances. Additionally, the choice of a singular touchdown point at a chosen landing site is often simplified to be the centroid or pseudo-centroid of the site or cell [24, 8]. However both of these choices do not represent the optimal touchdown location with respect to ensuring flatness and distance from obstacles. We formulate the problem of choosing a touchdown position as finding the largest inscribed circle in its 2D polygon representation. Circle placement guarantees landing target coordinates are maximally separated from any obstacle or edge.

5.2.3 Multi-Goal Planning

Multi-goal planning has two different definitions:

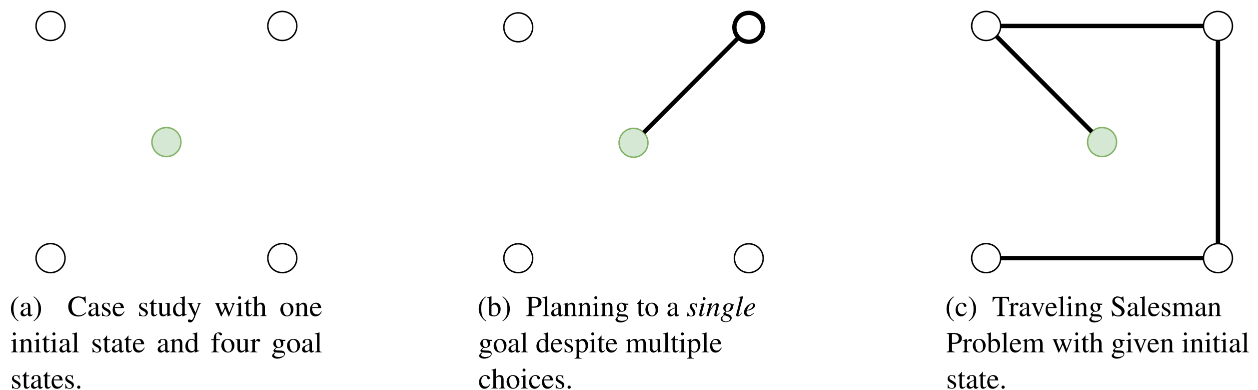
1. One start state and multiple goal states. The algorithm seeks to find the *singular* goal/path pair that minimizes/maximizes an objective function.

2. One start state, intermediary goal states, and a final end goal state. The algorithm seeks to find a connecting route, which includes *multiple* connected state pairs, that minimizes/maximizes some objective function. This formulation is commonly called the travelling salesman problem (TSP) [155].

A visual representation of these definitions is shown in Figure 5.2. This chapter focuses on the first definition per Figure 5.2b. Work by Ref. [156] investigated a form of multi-goal search using uninformed planners in a 2D grid. In this work all goals are valued equally with the objective to return paths for all start/goal combinations. Each start/goal pair can be treated as an independent path planning problem. Lim et al. propose reusing previous information, e.g., node expansions and costs, to reduce search overhead for the next goal. Results indicate that retaining information from previous searches reduces overhead in 2D grids when using uninformed search.

An objective or cost function may consider both the cost of a start/goal path and also the worth of the goal itself. Ref. [157] investigates efficient methods to conduct search for multiple agents seeking different products in a free market. The authors propose a multi-goal planner that maximizes expected overall utility from the set of opportunities found (e.g., products/goals) minus the costs associated with finding that set. In this work the worth or value of a goal is probabilistic and can only be ascertained through search. The objective function aims to balance goal achievement reward against the cost of obtaining that goal.

In our work multiple landing sites (goals) will typically be identified. Risk is a function of the path to each site as well as the site itself. Our objective is to find the singular goal/path pair which minimizes the combined risk of the landing site and a flight path from the initial UAS location to that site.



(a) Case study with one initial state and four goal states.

(b) Planning to a *single* goal despite multiple choices.

(c) Traveling Salesman Problem with given initial state.

Figure 5.2: Comparison of multi-goal planning definitions. The green state depicts the initial state, e.g., where the UAS begins its emergency landing trajectory. Unfilled circles are goal states.

5.2.4 Urban Landscape and Rooftop Landings

An emergency landing requires first identifying safe nearby landings sites. Cities lack conventional emergency landing sites such as open fields or grasslands. Empty lots are often sparsely distributed, and parks may be unexpectedly occupied. The satellite image in Fig. 5.3 illustrates a typical urban landscape that affords rooftop landing. Obstacles on a flat rooftop, e.g., air conditioning units, can be removed from potential landing site surfaces. We identify rooftop landing sites given sufficient flatness and distance from obstacles and edges.

Figure 5.3: Satellite image of an urban environment with multiple flat roof landing sites. Select roof shapes and obstacles are labeled.

Our proposed flight planner requires the vehicle to execute a stable approach to an emergency landing site. Failure scenarios can be detected and handled in time to execute a controlled landing. Scenarios in which an urgent landing can be achieved without loss-of-control include: low battery energy, lost communication link, adverse weather, non-essential sensor or actuator failure, operator emergency landing directive, and non-cooperative aircraft nearby. The methods and optimization techniques discussed in this chapter can be used for any VTOL aircraft. The case studies and simulations presented are specific to a multicopter in an urban environment.

Steps required to construct a database of buildings and suitable ground-based landing zones (e.g., parks, fields, etc.) from OSM are described in our previous work [113]. This database is geospatial, in the sense that each row refers to a geographic entity (e.g., building or field) with polygonal shape. Meta data is also captured about each entity, e.g., height and land use.

5.3 Preliminaries

5.3.1 Coordinates and Landing Sites

While map data will be globally georeferenced, low-altitude urban flights can be planned in a local Cartesian reference frame. Let orthogonal bases for this Cartesian coordinate frame be denoted $\hat{e}_x$, $\hat{e}_y$, and $\hat{e}_z$. The position of the UAS body frame with respect to the local Cartesian reference frame can then be defined as:

$$O_{UAS} = x\,\hat{\mathbf{e}}_x + y\,\hat{\mathbf{e}}_y + z\,\hat{\mathbf{e}}_z = [x, y, z]. \tag{5.1}$$

A set of candidate landing sites are generated within a radial footprint R defined as

$$\mathcal{S}_{ls} = \{l_i, \ldots, l_n\} \tag{5.2}$$

where each l_i refers to a landing site with properties

$$l_i = \{\mathbf{c},\ r_l,\ r_p\} \tag{5.3}$$

$$\mathbf{c} \in \mathbb{R}^3 \tag{5.4}$$

$$r_l, r_p \in \mathbb{R} \tag{5.5}$$

where **c** is landing site location in the Cartesian reference frame, r_l is landing site risk, and r_p is path risk. Both risk values are in domain $[0, 1]$. Landing site risk is calculated offline and represents the risk intrinsic to touching down at that landing site. Path risk must be calculated online and accounts for the path distance and proximity to obstacles. The calculation of r_l is shown in Section 5.4.4, while r_p is described in Section 5.5.

5.3.2 3D Path Planning with Mapped Obstacles

Path planning requires a cost function to guide search space exploration in pursuit of a feasible and optimal solution. For discrete search planners, the space must first be discretized into a graph $G(V, E)$, where V denotes graph vertices with edge set E and associated transition costs. This graph is defined implicitly for a 3D grid. The vertices are the cells/voxels accessed with indices (i, j, k). The edge of each cell are dynamically computed from the 26 neighbors. In a mapped environment, A* search applies a heuristic to reduce search overhead. A* sums actual path cost

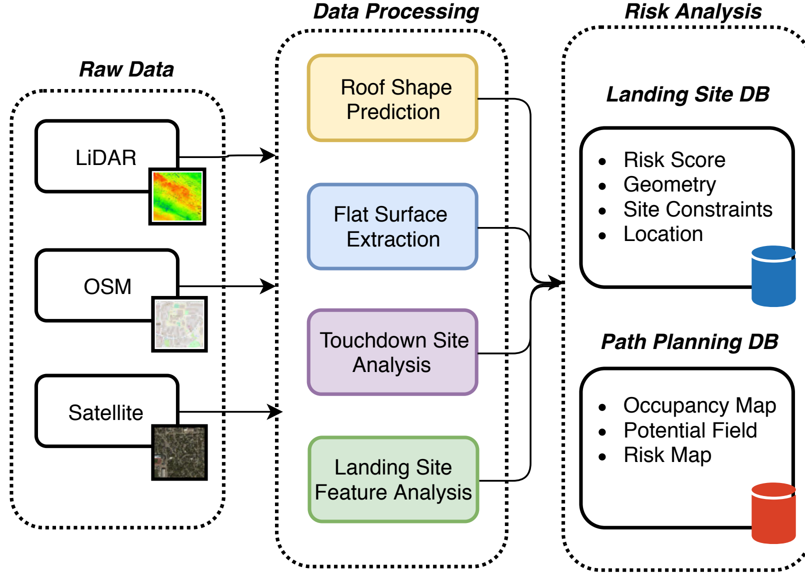


Figure 5.4: Processing pipeline to construct landing site and occupancy map databases. Landing sites and occupancy map are risk evaluated.

$g(n)$ and heuristic $h(n)$ estimating cost-to-go to form node n total cost $f(n)$:

$$c(n', n) = \text{dist}(n', n) \cdot (1 + \text{risk}(n)) \tag{5.6}$$

$$g(n) = g(n') + c(n', n) \tag{5.7}$$

$$f(n) = g(n) + h(n) \tag{5.8}$$

where c is transition cost from previous node n' to current node n, **dist**$(\cdot)$ is Euclidean distance between adjacent nodes n' and n, and **risk**$(\cdot)$ represents normalized risk encoded in the 3D map. The search space is limited to cells within the the radial footprint R to bound worst case scenarios. We use a 3D octile distance heuristic $h(n)$ which has been shown much more effective than Euclidean distance [158].

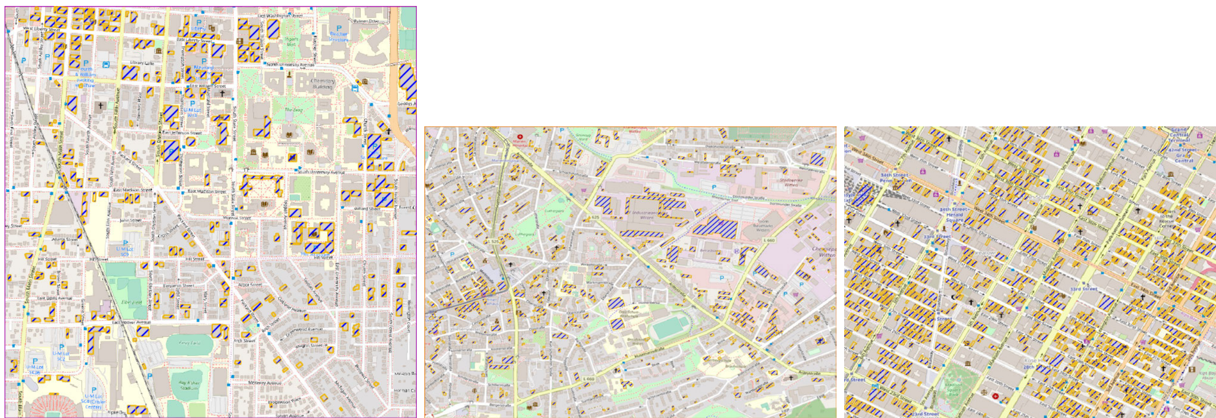
5.4 Landing Site Database

Data is fused from OSM, LiDAR, and satellite image sources to construct a feature-rich landing site database. Figure 5.4 provides an overview of the data processing pipeline that provides the features needed to construct a landing site database with risk metrics and maps. Below, Section 5.4.1 summarizes the Machine Learning (ML) process detailed in Chapter 4 to find flat-like rooftops from which landing sites are identified. Section 5.4.2 describes our procedure to refine each landing site by determining usable area. Section 5.4.3 describes a method to compute the touchdown location

on large flat surfaces. Landing site risk models are described in Section 5.4.4.

5.4.1 Flat-like Roof Identification

Information on building roof shape is sparse in existing databases. Chapter 4 shows that a total accuracy of 86% is achievable when evaluating a trained ML pipeline on independent test sets. By adjusting the confidence threshold to 50%, precision and recall of 95% and 75% can be achieved in classifying flat-like roofs in cities. This chapter employs the roof shape prediction model from Chapter 4 to label building roof shapes for subsequent landing site identification. High precision is necessary to assure that few false positives will be identified. Note that offline map building allows human inspection of city-wide rooftop landing sites to assure each identified unsafe site is pruned prior to landing site database use by a UAS.



(a) Ann Arbor, Michigan (b) Witten, Germany (c) Manhattan, New York

Figure 5.5: Maps of predicted flat rooftops in three cities. Buildings with a predicted flat-like roof shape are outlined in dark yellow with blue dashed lines through the center. Parks and grasslands are shown in green. Maps from ©OpenStreetMap contributors and ©CARTO. License: Open Database License: https://www.openstreetmap.org/copyright

Figure 5.5 shows results of our roof prediction model for the cities of Ann Arbor, Michigan, Witten, Germany, and mid-town Manhattan in New York City. In previous work only parks and grasslands were processed to identify emergency landing zones. This work adds new landing site options identified from the illustrated flat-like rooftops as described below.

5.4.2 Flat Surface Extraction for Usable Landing Area

Airborne LiDAR point cloud data is used to determine planarity, shape, and extent of potential UAS landing site surfaces. Proposed polygonal landing sites are buildings with predicted flat-like roof shapes and terrain locations with land use keywords listed in Table 5.1. A point-in-polygon ray

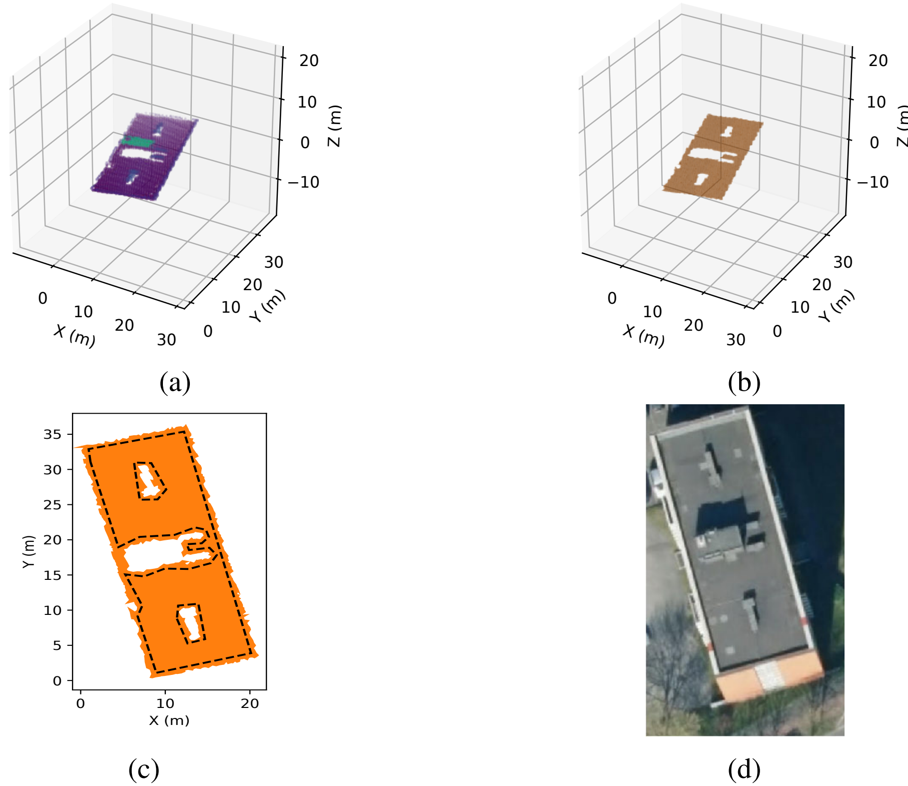


Figure 5.6: Flat surface extraction from rooftops. Point cloud data for a building is displayed in (a); (b) shows the generated planar triangular mesh. Conversion of the 3D mesh to a 2D polygon is shown in (c) with subsequent polygon inward buffering and simplification shown in dashed lines. A reference satellite image is shown in (d).

casting algorithm is used to generate the point set P_{ls} [135] where point clouds only reside in the outline of the landing site.

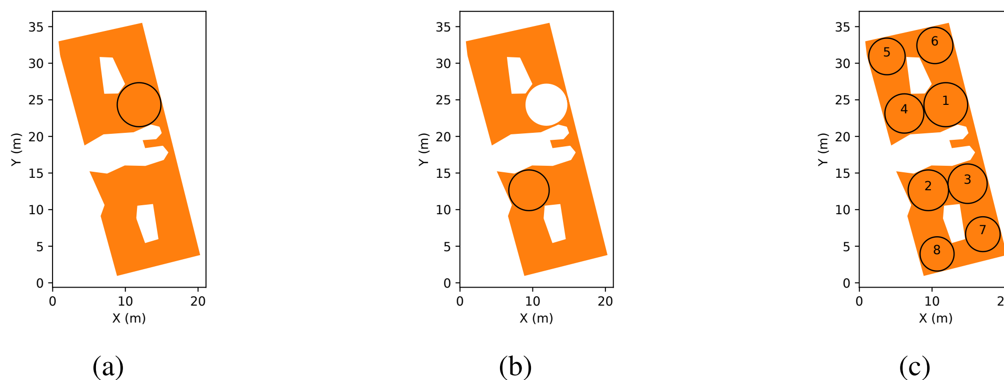
Next, flat surface extraction from P_{ls} is performed using the Polylidar3D algorithm developed by the authors in Chapter 3. The algorithm works by generating triangular meshes of an input point cloud, filtering triangles by planarity and edge length, extracting subsets of the mesh which are spatially connected, and finally converting the mesh to a polygon format. Polylidar3D is configurable by user provided planarity constraints and maximum triangle edge length. This guarantees the polygon can represent flat surfaces with interior holes denoting obstacles. Highlights of this procedure are shown in Figure 5.6.

After the flat surface is extracted as the orange polygon in Figure 5.6c, the polygon is buffered inward and simplified as denoted by dashed lines. The buffering process is defined as the Minkowski difference of the polygon with a circle with radius equal to a buffer distance [93]. We set the buffer distance to 0.5 meters which contracts the exterior hull and expands the interior holes. Afterwards we use the Douglas-Peucker's simplification algorithm to remove redundant vertices and "smooth"

the polygon [92]. Narrow flat surfaces can be removed in this process as shown near the (6, 17) point in subfigure (c). The final output of this procedure is a set of polygons denoting flat and obstacle free surfaces.

5.4.3 Touchdown Points

Once flat surface(s) have been extracted from potential rooftop and ground-based landing sites and represented as polygons, an ideal landing *touchdown point* must be defined. We define this ideal point to be farthest from any non-flat region or obstacle. In other words, the touchdown point is the furthest distance away from the exterior hull and any interior holes. This requirement may be framed as the Poles of Inaccessibility problem [159] and the Polylabel algorithm as proposed in [160] provides a solution. This algorithm aims at efficiently determining the largest inscribed circle in a polygon within a prescribed tolerance. The largest inscribed circle for the same rooftop in Figure 5.6 is shown in Figure 5.7a. There are additional suitable touchdown sites on this rooftop that can be used. We propose Algorithm 5.1 to capture the remaining touchdown points as a ranked list of circles. The algorithm begins by calling Polylabel to find the point and radius representing the largest circle inside an input polygon P. A 16-sided polygon representation of this circle is created denoted P_c. If the radius is below a user provided minimum radius r_{min} then the empty set is returned. P_c is subtracted from the input polygon to create a smaller polygon P_{diff}. The procedure ends by returning the union of P_c and the result of a recursive call to `TouchdownExtraction` with P_{diff} as the input polygon. This recursive call continues the process of finding next largest circle. The end result is an ordered set of circular polygons with a radius greater than r_{min}. This minimal radius is user defined and should determined by UAS characteristics. Visualization after the first and second procedure call are in Figure 5.7a,b respectively, with the final rankings shown in Figure 5.7c.



(a) (b) (c)

Figure 5.7: Touchdown point extraction on rooftops. Rooftop surface shown in orange. The best (largest circle) landing zone is shown in (a). (b) Polygon subtraction of a previously found touchdown zone. The complete ranked touchdown site set is shown in (c).

Algorithm 5.1: TouchdownExtraction

Input : Polygon: P, Mininum Radius: r_{min}
Output : Set of Polgyon Circles

1. $point, r = \text{Polylabel}(P)$
2. $P_c = \text{PolgyonCircle}(point, r)$
3. **if** $r < r_{min}$**:**
4. │ return $\emptyset$
5. P_{diff} = $P - P_c$
6. return $P_c \cup \text{TouchdownExtraction}(P_{diff}, r_{min})$

The final landing site chosen for each surface is the top ranked touchdown site. The remaining circles are kept in the database for further use in risk assessment.

5.4.4 Landing Site Risk Model

Each landing site corresponds to the largest clear touchdown area on either flat terrain or building rooftop surfaces. This chapter adopts the risk model first presented in Ref. [113]. Risk is quantified as vehicle cost (C_v), property cost (C_p), and human occupancy cost (C_o). Each of these risks are numeric values created from a functional composition of the attributes of each feature (land use, available area, etc.) as outlined in Sections 5.4.4.1, 5.4.4.5, and 5.4.4.6 respectively. Landing site risk is the weighted sum

$$r_l = w_v \cdot C_v + w_s \cdot C_p + w_o \cdot C_o \tag{5.9}$$
$$w_v + w_s + w_o = 1$$

where $w_v, w_s,$ and w_o are weights for vehicle, property and human occupancy cost respectively.

5.4.4.1 Vehicle Cost

We denote risk to the UAS ”vehicle” as

$$C_v = w_t C_t + w_a C_a + w_{ca} C_{ca} \tag{5.10}$$
$$C_v,\ C_t,\ C_a,\ C_{ca} \in [0, 1] \tag{5.11}$$

where C_t, C_a, and C_{ca} are risk-based costs associated with terrain type, usable area, and cumulative usable area respectively. The variables w_t, w_a, and w_{ca} are the user-defined weights aggregating these metrics into a total C_v value.

5.4.4.2 Terrain Cost

Terrain cost C_t approximates the risk posed to the vehicle due to landing on a specific type of terrain. We use keywords gathered from OSM that describe type of terrain. Following a similar taxonomy from [24], these keywords are aggregated into groups and assigned costs as shown in Table 5.1. The trend of these costs is that groups with generally unoccupied open areas, such as Group 2, have lower risk than groups with possible cluttered areas, such as Group 5. Building rooftops, Group 1, have a slightly higher terrain cost than Group 2 because of increased risk at landing at higher altitudes. Group 7, industrial and commercial areas, have more diverse and uncertain terrain characteristics and assigned a higher cost. These costs are subjective in nature thus would be refined later by stakeholders.

Table 5.1: Terrain type and property cost

Group	Keywords	Terrain (C_t)	Property (C_p)
Group 1	building rooftops	0.25	0.5
Group 2	brownfield, grass, grassland village green, greenfield	0.0	0.0
Group 3	meadow, cemetery, scrub	0.25	0.0
Group 4	water, riverbank	0.75	0.0
Group 5	recreation ground, garden, golf course, track, pitch, playground, common, park	0.5	0.25
Group 6	parking	0.75	0.75
Group 7	industrial, commercial	1.0	1.0

5.4.4.3 Area Cost

Large flat surfaces pose less risk to UAS than small clearings. Our proposed risk model quantifies this as an area cost C_a. We propose Eq. 5.12 to map small areas to high risk (1), average areas to medium risk (0.5), and large areas to low risk (0). This piecewise exponentially decaying function is governed by user-defined minimum area A_{min} and maximum area A_{avg}. Note that A_{min} is the area of the circle with radius r_{min} used in Algorithm 5.1. These values would take UAS-specific

landing area requirements into account.

$$\text{Area Cost}(a) = \begin{cases} 1 & a \leq A_{min} \\ e^{-c \cdot a} & a > A_{min} \end{cases} \tag{5.12}$$

$$c = \frac{\ln 2}{A_{avg} - A_{min}} \tag{5.13}$$

An example of this mapping is shown in Figure 5.8.

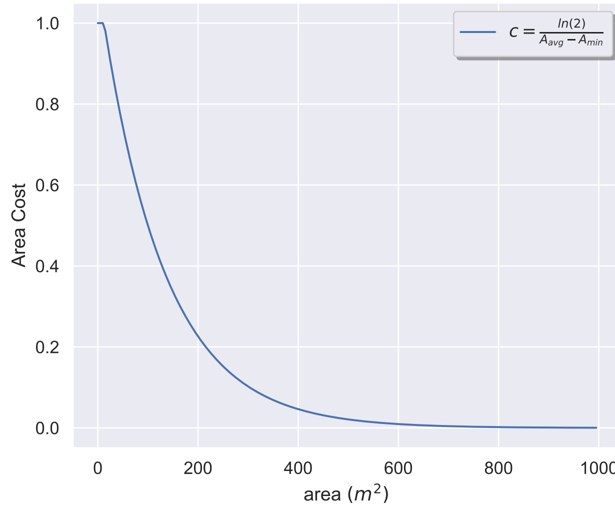


Figure 5.8: Mapping area size to risk. Function maps area size to a cost value between [0, 1]. In this example A_{avg} and A_{min} are 100 and 12.5m^2, respectively.

5.4.4.4 Cumulative Area Cost

A landing site that is nearby additional sites has advantage to landing sites with no other nearby options. We capture this metric as a cumulative area cost C_{ca}. The cumulative area of all touchdown sites available on an individual building or terrain surface is computed as shown in Figure 5.7c. This area is then mapped to C_{ca} using equation 5.12.

5.4.4.5 Property Cost

UAS may inadvertently damage a landing site in the event of an unplanned landing. Landing site characteristics can impact the likelihood, severity, and cost of damage. Table 5.1 provides example normalized estimates for the costs of similar landing sites. The proposed metric assigns increase cost to areas that may be damaged in the event of a high impact crash. Natural terrain areas have low cost while buildings and other maintained properties have higher cost. Parking lots and industrial areas are marked with the highest cost to people and property, e.g., cars, pedestrians.

5.4.4.6 Human Occupancy Risk Mapping

Overflight risk to people is typically estimated with an occupancy grid based on Census records [161, 4, 162]. Census records in the United States are released every 10 years in an aggregated form in which the smallest areal unit is the census block. The size of a census block may vary from a city block to a much larger area in a rural community. For example in New York City the average size of a census block is a 121X121 meter square but with substantial variance over the city landscape. This average resolution is suitable for city-wide risk assessment but not ideal for higher-resolution occupancy mapping. Techniques such as daysymetric modelling [163] are used to improve the spatial resolution of Census datasets at the cost of greater uncertainty [164].

Census records only provide information of where people reside which is most often representative of a region's nighttime population [24]. UAS will fly at all times of the day so time-varying population estimates are critical for accurate risk assessment. Work by Ref [24] fused census data with publicly available mobile phone call detail records (CDR) in Italy to generate a temporal population model for UAS risk assessment. Results indicated significant population migration throughout the day reinforcing the inadequacy of using a static population model. However, CDRs are not generally open for public use, and access to other real-time data streams such as aggregated mobile phone GPS is limited. Most of the landing sites proposed in this chapter are on flat rooftops likely to be unoccupied despite high building occupancy. Many population risk models introduce a shelter factor which estimates zero casualties whenever the building is not penetrated, e.g., during a UAS rooftop landing [26].

This work requires risk assessment for landings sites and paths over small radial footprints (< 250 m). The authors have previously used census data to assess population risk in this situation. However, a static low-resolution population model may provide misleading risk assessments. Therefore this work does not use population risk metrics by setting w_o to 0 in Eq. 5.9. Our combined risk model will include path length, which when minimized, is a proxy to minimizing the risk to people during urgent landings so long as population is uniformly distributed rather than clustered, e.g., for special events not modeled in census data.

5.5 Three-Dimensional Maps for Path Planning

Let the city occupancy and risk map generated for path planning be denoted R_{map}. This map is a dense 3D voxel structure of size $M \times N \times K$, where the rows, columns and slices are M, N and K respectively. Each cell is indexed by triplet (i, j, k) returning occupancy and risk information for a specific position. Publicly available airborne point cloud data or a DSM may be used as the primary data source to construct R_{map}. A DSM is a raster where each pixel holds a height value above Earth's Mean Sea Level (MSL) including buildings and foliage. Such data sources are

often georeferenced in a projected coordinate system which minimizes distortion of shape, area, or distance. This Cartesian coordinate system is ideal for path planning thus is carried into the voxel map. The rest of this procedure assumes the use of a DSM with equal pixel resolution and associated affine transformation matrix to convert from pixel space to the local Cartesian frame.

The procedure begins with a city DSM of size $M \times N$. The minimum and maximum height, z_{min} and z_{max}, are computed from the DSM. The value of z_{max} is bounded at 400 feet above local terrain level in this work. Note that FAA Part 107 restricts flying small UAS more than 400 feet above the tallest nearby obstacle [165]. An affine transformation matrix A is generated for R_{map}:

$$A = \begin{bmatrix} x_{res} & 0 & 0 & x_{min} \\ 0 & -y_{res} & 0 & y_{max} \\ 0 & 0 & z_{res} & z_{min} \end{bmatrix} \tag{5.14}$$

$$\begin{bmatrix} x & y & z \end{bmatrix}^T = A \cdot \begin{bmatrix} i & j & k \end{bmatrix}^T \tag{5.15}$$

where x_{min}, x_{res}, y_{max}, and y_{res} are provided by the DSM affine matrix. Eq. 5.15 performs the conversion from 3D voxel space to the local Caretesian coordinate system. The number of slices, K, is equal to $\left\lfloor \frac{z_{max} - z_{min}}{z_{res}} \right\rfloor$. An $M \times N \times K$ data structure storing unsigned 8 bit integers is zero initialized to represent R_{map}.

The occupancy map is generated similar to [4] where each (i, j) cell in the DSM is matched to an R_{map} cell (i, j, k). The index k is calculated by $\left\lfloor \frac{DSM(i,j) - z_{min}}{z_{res}} \right\rfloor$. All cells in R_{map} at the (i, j) position and below the k slice are then set to the value 255 to indicate an obstacle exists. This is done for each pixel in the DSM until a full 3D occupancy map is generated for R_{map}. Afterwards a potential field cost is applied to R_{map} which fills empty cells near each obstacle cell with nonzero risk values. Three possible levels of potential field risk are assigned to an empty cell based upon its shortest Manhattan distance to an obstacle cell. Distances of one, two, and three provide integer risk values of 254, 170, and 85 respectively. An example 3D grid of New York City is shown in Figure 5.9.

An A* path planner is used to generate optimal collision free trajectories inside R_{map}. Obstacle nodes, cells with a value of 255, are ignored for state transitions. The risk function described in Eq. 5.6 is defined as

$$\text{risk}(i, j, k) = \frac{R_{map}(i, j, k)}{255} \tag{5.16}$$

The path risk to a goal cell, n_g, is the path cost to the goal node normalized by R

$$r_p = \frac{g(n_g)}{R} \tag{5.17}$$

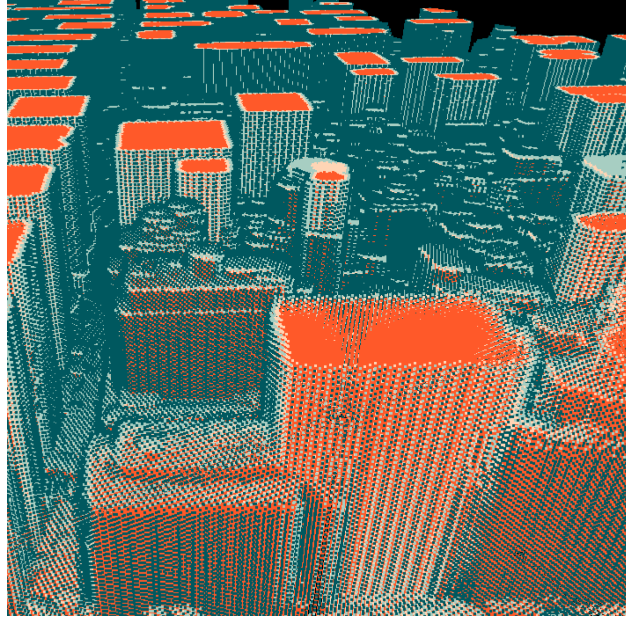


Figure 5.9: Example occupancy and risk map of New York City. Obstacles are colored orange with a surrounding potential field denoted by pink, light blue, and dark blue colors. Buildings which do not fit in the map (higher then 400 feet AGL), are shown with orange roofs.

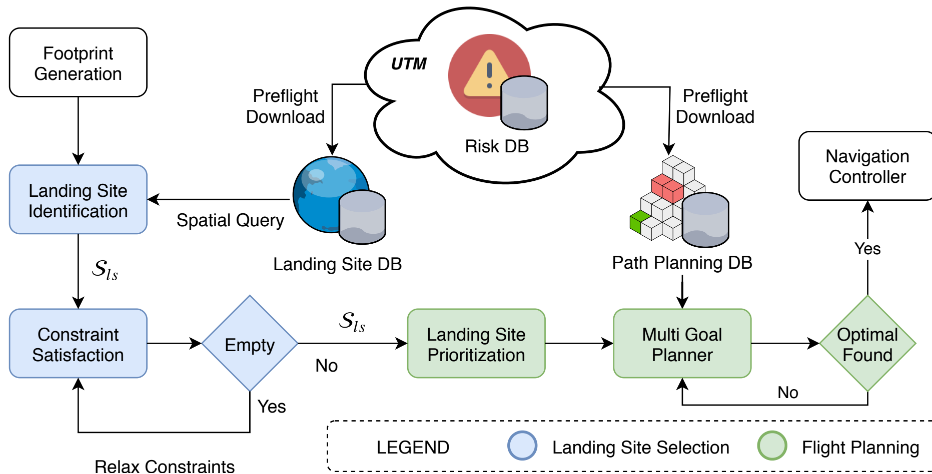


Figure 5.10: Flow chart of proposed map-based planner.

5.6 Planning Risk Metric Analysis and Integration

Section 5.6.1 describes the architecture of our proposed map-based planner. Section 5.6.2 outlines the inherent trade-off between landing site and path risk and our planning method. Section 5.6.3 provides underlying theoretical and algorithmic formulations of our multi-goal planner.

5.6.1 Real-time Map-Based Planner Architecture

The proposed architecture for our map-based planner is shown in Figure 5.10 which is modified from our previous work [3, 113]. The landing site and path planning database might be provided as part of NASA's UAS Traffic Management (UTM) service [166]. Before mission operations begin, a preflight download commences from UTM servers to retrieve relevant data for the flight operational area. These data are lightweight thus can be stored onboard the UAS. In the event an urgent landing situation arises our map-based planner logic will be executed. First a footprint specifying the bounds of the reachable landing area is generated. Construction of such a footprint is not the focus of this chapter, however work by [167, 3] and [4] have investigated its generation for fixed-wing and multicopter aircraft, respectively. For simplification our footprint is a circle of radius R whose center is the UAS position. Next we efficiently query the spatially indexed landing site database to provide the set of risk-evaluated landing sites $\mathcal{S}_{ls}$. Landing sites are filtered by user defined constraints such as landing site height or area. If no valid landing sites are found constraints are relaxed as needed.

The planner must identify a low risk landing site and flight plan from the set of candidate landing sites $\mathcal{S}_{ls}$. In order to assess each landing site's path risk the physical path to each landing site is needed. Optimal collision-free path planning in three dimensional space can take a significant amount of time thus is impractical to perform in real-time for the numerous potential landing sites that may be available for a small UAS in a city environment. Therefore we use a heuristic to prioritize landing sites in $\mathcal{S}_{ls}$ by minimum total risk. This sorted list is then sent to our multi-goal planner which efficiently searches over landing sites until the risk-optimal landing site/path pair is found. Finally the landing site and path is sent to a navigation controller.

5.6.2 Trade-off Between Landing Site and Path Risk

Minimizing the landing site risk and path risk to a site requires solving a multi-objective (MO) optimization problem from which there may not be a single solution simultaneously optimal over both objectives. An analysis of trade-offs is required for MO problems by computing and analyzing a Pareto frontier. Frontier visualization aids system designers in choosing the relative weighting of objective trade-offs to select a single "best" solution [168].

Fig. 5.11 shows an example Pareto frontier that minimizes two objectives: landing site risk and path risk. Each purple dot represents a landing site. The x-axis represents landing site risk and the y-axis represents path risk to that site. The green line connects three points on the Pareto frontier, the set of non-dominated landing sites for which any improvement in one objective results in a negative trade-off in the other. Each of these three landing sites is "optimal", and a quantifiable relationship between each objective must be constructed to select a final choice. A linear weighting

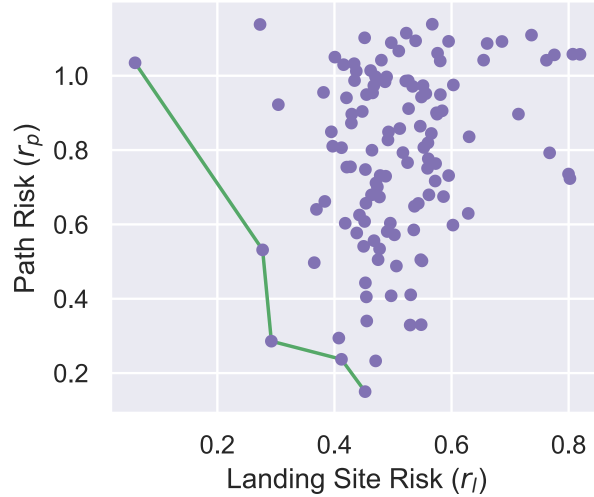


Figure 5.11: Example Pareto frontier for landing site and path risk. Demonstrates trade-off between to minimize both objectives. Points in the Pareto frontier are connected by a green line.

scheme between the objectives is proposed below for each landing site $l_i \in \mathcal{S}_{ls}$:

$$r_t = w_l \cdot r_l + w_p \cdot r_p \tag{5.18}$$

where r_t refers to the total risk and w_l and w_p are weights for landing site risk and path risk, respectively. The optimal landing site can then be found by solving the optimization problem shown in Eq. 5.19.

$$l_{i^*} = \underset{l_i \,\in\, \mathcal{S}_{ls}}{\arg\min}\ r_t \tag{5.19}$$

5.6.3 Multi-Goal Planner

Our multi-goal planner selects a landing site that minimizes total risk given a user-defined weighted trade-off between landing site risk and path risk. Each landing site's position and risk is assumed known *a priori*. Path risk cannot be known until the physical path is computed. Because this work relies on preprocessed map data rather than real-time perception, path planning has the highest real-time computational overhead. Multi-goal search allows exploration of many low-risk landing sites but will be computationally expensive. Our planner efficiently prunes high risk goals/paths from the search space to reduce computational overhead. Ultimately the planner returns one goal/path pair minimizing combined total risk. The algorithm begins by creating an array of

landing site data structures with the following form:

$$l_i = \{\mathbf{c},\ r_l,\ r_p,\ r_{t,min},\ found\} \tag{5.20}$$

$$\mathbf{c} \in \mathbb{R}^3 \tag{5.21}$$

$$r_l, r_p, r_{t,min} \in \mathbb{R} \tag{5.22}$$

$$found \in \{\ 0, 1\ \} \tag{5.23}$$

where $\mathbf{c}$ is landing site position, r_l is landing site risk, r_p is path risk, $r_{t,min}$ is minimum total risk, and $found$ is a Boolean indicating whether a path has been found to landing site i. The minimum total risk is computed from

$$r_{t,min} = w_l \cdot r_l + w_p \cdot h(\mathbf{O}_{UAS}, \mathbf{c})/R \tag{5.24}$$

where $h()$ is an admissible heuristic, 3D octile distance in this work. We access elements in l_i through dot (.) notation, e.g. $l_i.r_{t,min}$ refers to the minimum total risk of the ith landing site in $\mathcal{S}_{ls}$. Section 5.6.3.1 provides definitions and a theorem for our multi-goal planner. Section 5.6.3.2 describes the planning algorithm and its implementation.

5.6.3.1 Theory

Definition 4. *Total Ordered Set Binary relation, $\leq$, is a total order on set $\mathcal{X}$ if $\forall a, b\ \in \mathcal{X}$*

1. $a \leq b$ *and* $b \leq a \implies a = b$ *Anti-symmetry*
2. $a \leq b$ *and* $b \leq c \implies a \leq c$ *Transitivity*
3. $a \leq b$ *or* $b \leq a$ *Connexity*

We define binary operator $\leq$ on the set $\mathcal{S}_{ls}$ $\forall l_i, l_j \in \mathcal{S}_{ls}$:

$$l_i \leq l_j : l_i.r_{t,min} \leq l_j.r_{t,min} \tag{5.25}$$

This operator is used to sort $\mathcal{S}_{ls}$ such that the natural numbers $i, j \in [1, N]$ index with the following property:

$$\forall l_i, l_j \in \mathcal{S}_{ls}\ l_i \leq l_j \iff i \leq j \tag{5.26}$$

where $N = |\mathcal{S}_{ls}|$. We use bracket operator $[\cdot]$ to index $\mathcal{S}_{ls}$, e.g., $l_i = S_{ls}[i]$.

Theorem 1. *Let $i^* \in [1, N]$ and $k \in [1, N]$ be natural numbers where $i^* \leq k$. If $\forall j \in [1, k],\ l_{i^*}.r_t \leq l_j.r_t$ and $l_{i^*}.r_t \leq l_{k+1}.r_{t,min}$ then l_{i^*} has the minimum total risk:*

$$l_{i^*} = \underset{l_i \,\in\, \mathcal{S}_{ls}}{\arg\min}\ l_i.r_t \tag{5.27}$$

Proof. The stated inequalities partition $\mathcal{S}_{ls}$ into two ordered sets for some index $k \in [1, N]$. We denote $\mathcal{S}_{ls}^{low} = \{l_1, \ldots, l_k\}$ where $l_{i^*} \in \mathcal{S}_{ls}^{low}$ and $\mathcal{S}_{ls}^{hi} = \{l_{k+1}, \ldots, l_N\} \cup \{l_{i^*}\}$. We must show that l_{i^*} represents the minimum total risk in both sets. When $\forall j \in [1, k]\ l_{i^*}.r_t \leq l_j.r_t$ holds true, by the definition of argmin we know that:

(1) $$l_{i^*} = \underset{l_i \,\in\, \mathcal{S}_{ls}^{low}}{\arg\min}\, l_i.r_t$$

To show l_{i^*} has the minimum actual total risk of $\mathcal{S}_{ls}^{hi}$ we begin by noting that for each landing site l_j

(2) $$\forall j \in [1, N],\ l_j.r_{t,min} \leq l_j.r_t$$

If a landing site l_i has total risk $l_i.r_t$ which is less than the minimum total risk of landing site l_j denoted $l_j.r_{t,min}$ then

(3) $$\forall\ i, j \in [1, N]\ l_i.r_t \leq l_j.r_{t,min} \implies l_i.r_t \leq l_j.r_t$$

From the transitivity property of $\mathcal{S}_{ls}$ we obtain

(4) $$\exists\ i^*, k \in [1, N]\ s.t.\ l_{i^*}.r_t \leq l_{k+1}.r_{t,min} \implies \forall j \in [k+1, N]\ l_{i^*}.r_t \leq l_j.r_{t,min}$$

Finally by combining (3) and (4) we obtain

(5) $$\exists\ i^*, k \in [1, N]\ s.t.\ l_{i^*}.r_t \leq l_{k+1}.r_{t,min} \implies \forall j \in [k+1, N]\ l_{i^*}.r_t \leq l_j.r_t$$

Using the definition of argmin we restate (5) as

(6) $$\exists\ i^*, k \in [1, N]\ s.t.\ l_{i^*}.r_t \leq l_{k+1}.r_{t,min} \implies l_{i^*} = \underset{l_i \,\in\, \mathcal{S}_{ls}^{hi}}{\arg\min}\, l_i.r_t$$

Statements (1) and (6) show that l_{i^*} has the minimum total risk in $\mathcal{S}_{ls}^{low}$ and $\mathcal{S}_{ls}^{hi}$ if both qualifying predicates hold true. Therefore the union of these sets, $\mathcal{S}_{ls}$, has the same minimum $l_{i^*}.r_t$.

□

Remark. There may *not* exist a k for which the second clause $l_{i^*}.r_t \leq l_{k+1}.r_{t,min}$ in Theorem 1

holds true. In this case $\mathcal{S}_{ls}^{low} = \mathcal{S}_{ls}$ and path planning must be performed for every landing site to guarantee risk-optimality.

5.6.3.2 Multi-goal Path Planning Algorithm

Our multi-goal path planner is shown in Algorithm 5.2. First, $\mathcal{S}_{ls}$ is sorted by minimum total risk as described in Definition 4. The algorithm next initializes variables $\hat{l}_{min}, \hat{p}_{min},$ and $\hat{r}_t$ to track the minimum risk landing site, associated path, and total risk, respectively. These variables represent the current best landing site/path pair and are updated each time a lower risk landing site is found.

Our algorithm repeatedly investigates the next most promising landing site l_i from $\mathcal{S}_{ls}$. Line 11 starts a path planning sequence with 3D octile distance heuristic guiding the planner to l_i. The planner is opportunistic so that other landing sites (goals) may be found during the search. For this reason the identified landing site is returned from function `PathPlanning`, l_j, and will not always be equal to l_i. The true total risk of l_j is then calculated for the full flight plan and its *found* flag set. This landing site's total risk is then compared to the current best and updated if appropriate, ensuring the first clause of Theorem 1 is satisfied.

The first element in $\mathcal{S}_{ls}$ which has not been found is returned in Line 18 as the next unplanned landing site that has minimum total risk. The tracked total risk is compared with this element's minimum total risk, and if less or equal will satisfy the second clause in Theorem 1. Once both predicates are satisfied iteration terminates and $\hat{l}_{min}$, $\hat{p}_{min}$ are returned as the risk-optimal landing site and plan, respectively. If the optimal site is not found then the procedure continues. Line 21 ensures that if l_i was not found (l_j is *not* l_i) then the planner will retry l_i in the next iteration. This is accomplished by decrementing the loop variable i. The total number of iterations is equal to k in Theorem 1 which represents the number of landing sites searched.

Note that the algorithm can return the best found landing site and path at any iteration step if computational time becomes a concern. In addition a worst case bound of unnecessary risk can be computed from the difference between the returned landing site's total risk and the minimum total risk of the next landing site to be searched.

5.7 Maps and Simulation Results

Landing site databases and path planning obstacle maps were generated for the cities of Ann Arbor, Michigan; Witten, Germany; and mid-town Manhattan, New York City, New York using the methods outlined in Sections 5.4 and 5.5. Public data sources used for all three cities are shown in Table 5.2. Visualization and analysis of each city's databases and maps are found in Section 5.7.1. Section 5.7.2 presents two urgent landing scenarios for each city with results from our proposed framework. Section 5.7.3 provides statistical analysis of our planners speed and efficacy in all three

Algorithm 5.2: Multi-Goal Search

Input : Landing Site Set ($\mathcal{S}_{ls}$),
Map (M),
UAS Location ($\mathbf{O}_{UAS}$)
Footprint Radius (R),
Weighting Trade-off (w_l, w_p),
Heuristic ($h(c_i, c_j)$)

Output : Best landing site and path, $\hat{l}_{min}$ and $\hat{p}_{min}$

1 $\mathcal{S}_{ls}$ = $\text{LandingSitePrioritization}(\mathcal{S}_{ls}, \mathbf{O}_{UAS}, R, w_l, w_p, h)$
2 N = $\mid \mathcal{S}_{ls} \mid$
3 $\hat{l}_{min} = None$
4 $\hat{p}_{min} = None$
5 $\hat{r}_t = \infty$
6 **for** $i = 0$ **to** N **do**
7 $\quad l_i = \mathcal{S}_{ls}[i]$
8 $\quad$ **if** l_i.*found* **then**
9 $\quad\quad$ continue
10 $\quad goals = \{\forall\ l_i \in\ \mathcal{S}_{ls} \mid not\ l_i.found\}$
11 $\quad l_j, p_j$ = PathPlanning($M, \mathbf{O}_{UAS}, R, h, l_i, goals$)
12 $\quad l_j$.found = true
13 $\quad l_j.r_t$ = $w_l \cdot l_j.r_l + w_p \cdot \frac{\text{Cost}(p_j)}{R}$
14 $\quad$ **if** $l_j.r_t < \hat{r}_t$ **then**
15 $\quad\quad \hat{r}_t = l_j.r_t$
16 $\quad\quad \hat{l}_{min} = l_j$
17 $\quad\quad \hat{p}_{min} = p_j$
18 $\quad l_n = \text{FirstElement}(\{\forall\ l_i \in \mathcal{S}_{ls} \mid not\ l_i.found\})$
19 $\quad$ **if** $\hat{r}_t <= l_n.r_{t,min}$ **then**
20 $\quad\quad$ break
21 $\quad$ **if** *not* l_i.*found* **then**
22 $\quad\quad$ i = i - 1
23 **end**
24 return $\hat{l}_{min}, \hat{p}_{min}$

cities. Table 5.3 displays parameters used throughout all case studies and simulations. The average touchdown site areas A_{avg} in each city were set to 150, 100, and 100 square meters for Ann Arbor, Witten, and New York respectively. The minimum touchdown radius and corresponding area were set to 2 meters and 12.5 square meters, respectively, for all cases.

Table 5.2: Satellite, LiDAR, and building data sources

City	Satellite	LiDAR	Buildings
Ann Arbor	Bing [148]	USGS [149]	OSM[114]
Witten	Land NRW [100]	Open NRW [99]	OSM[114]
New York	New York State [145]	USGS [146]	OSM[114]

Table 5.3: Emergency landing case study parameters

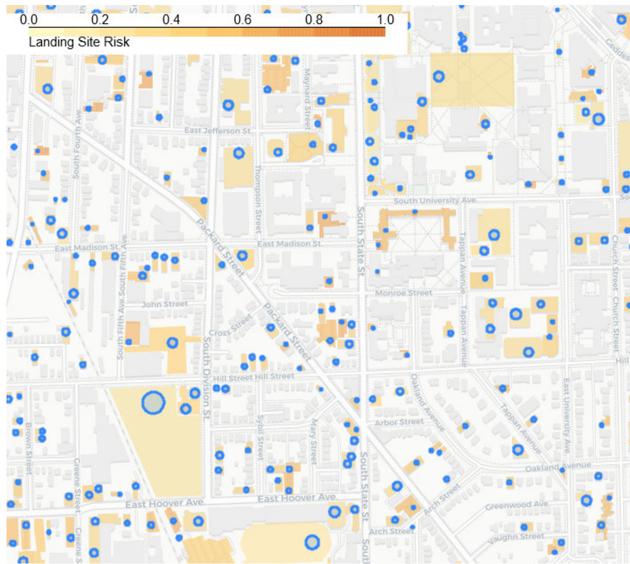
Group	Parameter	Value	Description
Landing Site Risk	w_v	0.8	Weight for vehicle cost
	w_s	0.2	Weight for property cost
	w_o	0.0	Weight for human occupancy cost
Vehicle Cost	w_t	0.4	Weight for terrain type cost
	w_a	0.4	Weight for area cost
	w_{ca}	0.2	Weight for cumulative area cost
Multi-Goal Planner	w_l	0.6	Weight for landing site risk
	w_p	0.4	Weight for path risk
	R	250m	Search radius footprint

5.7.1 Landing Sites and Risk Maps

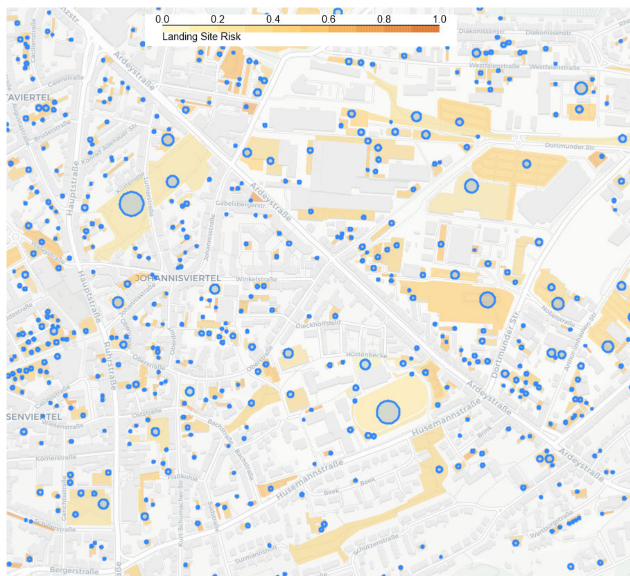
Figure 5.12 shows extracted landings sites and their associated risks for the cities of Ann Arbor (a), Witten (b), and New York (c). Landing site risk is color-coded from low (light yellow) to high (dark orange). Touchdown sites are displayed as blue circles. The operating regions are approximately 1500×1500, 1500×1300, and 1500×3000 meters for Ann Arbor (AA), Witten (WT), and New York (NY), respectively. Three-dimensional risk grids as described in Section 5.5 were generated for all three cities. Each voxel in R_{map} is a cube with two meter edge length. This resolution provides a balance between file size and providing sufficient resolution for use in path planning. The resulting file sizes are approximately 37, 26, and 67 MB for AA, WT, and NY respectively.

5.7.2 Case Studies

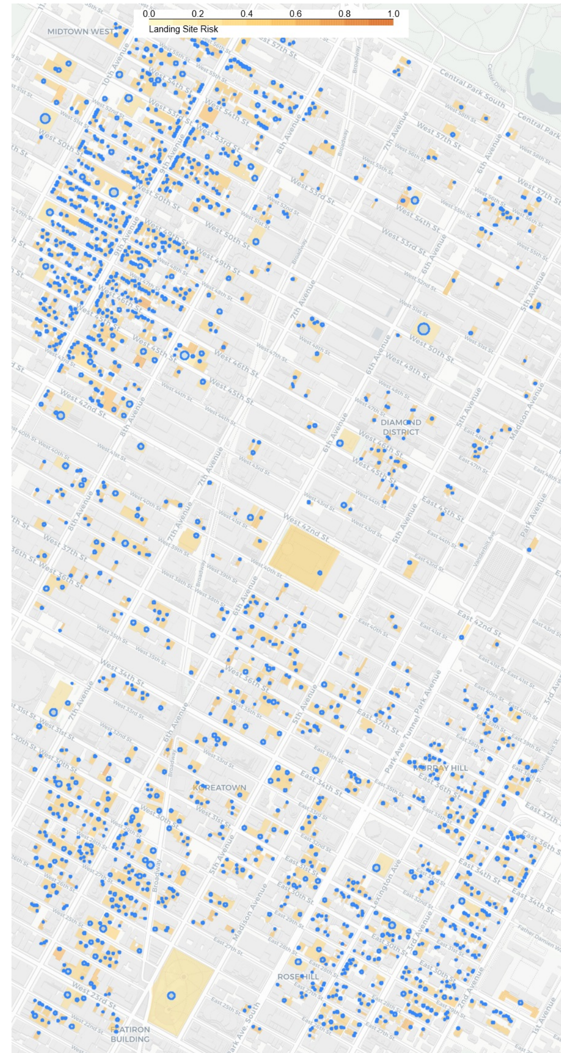
We present two case studies for each city where an urgent landing is required for a small UAS. Figure 5.13 presents a map of each case study with locations shown in Table 5.7.2. The first row (a,b) is for Ann Arbor, the second row (c,d) is for Witten, and the final row (e,f) is for Manhattan. Position of the UAS during the urgent landing event is indicated by the green marker. Landing site risk is colorized from low (yellow) to high (dark orange) risk with associated touchdown sites marked as blue circles. The lowest risk landing sites, not considering path risk, are ranked and



(a) Ann Arbor



(b) Witten



(c) New York

Figure 5.12: Maps of landing sites and associated risk. Landings site risk for the cities of Ann Arbor (a), Witten (b), and New York (c). Landing sites are color-coded from low risk (light yellow) to high risk (dark orange). Touchdown sites are denoted by blue circles. Maps from ©OpenStreetMap contributors and ©CARTO. License: Open Database License: https://www.openstreetmap.org/copyright

marked with blue numbered icons. Our planner's chosen landing site is marked in red which trades off landing site and path risk. Pareto plots are also provided for each of these case studies in Fig. 5.14. To generate these plots, collision-free paths to *all* landing sites for each scenario were generated, providing their actual path risk. The first, second, and third row correspond to Ann Arbor,

Witten, and New York, respectively. Each scenario graph has the same axis limits, allowing the reader to compare each scenario visually. Each purple dot represents a landing site, while the red dot represents the landing site chosen by the map-based planner. The Pareto set for each scenario is depicted by the green line.

Figure 5.13: Maps of case studies for emergency landing. Maps of Ann Arbor (a,b), Witten (c,d), and New York (e, f). Failure position of the UAS is indicated by the green marker. Landing site risk is colorized from low (yellow) to high (dark orange) risk, with associated touchdown sites marked as blue circles. The lowest risk landing sites are ranked and marked with blue numbered icons. Our planner's chosen landing site is marked in red which trades off landing site and path risk. Maps from ©OpenStreetMap contributors and ©CARTO. License: Open Database License: https://www.openstreetmap.org/copyright

Table 5.4: Emergency landing case study locations

Scenario	Lng/Lat (degrees)	Height, MSL (m)
AA CS#1	42.2783, -83.7473	260
AA CS#2	42.2748, -83.7357	270
WT CS#1	51.4391, 7.3369	109
WT CS#2	51.4443, 7.3359	130
NY CS#1	40.7460, -73.9905	19
NY CS#2	40.7662, -73.9903	17

Figure 5.13a,b shows results of our map-based planner in Ann Arbor case studies 1 and 2. In case study 1 landing sties with low landing site risk are nearby, generating the Pareto frontier seen in the first row and column in 5.14. The resulting front is approximately linear and the multi-goal planner, which favors landing site risk per Table 5.3 chooses the landing site with lowest r_l. Case study 2 has the unfortunate situation where the best landing sites are far away, resulting in a Pareto front that has a sharp vertical drop. This drop allows the planner to make a trade-off between landing site risk and path risk which favors the point near the bottom of the front which represents a landing site near the UAS.

Witten case studies are shown in Figure 5.13c,d. The Pareto front for case study 1 is nearly linear with the exception of a dip caused by one landing site (red point). The significant drop in path risk causes it to have the minimum total risk and be selected. Case study 2 shows a Pareto front shifted far to the right, indicating that few landing sites with low r_l are available. In addition few landing sites are immediately nearby for landing, forcing the planner to select a landing site which has a higher total risk than seen in case study 1.

New York City case studies are shown in Figure 5.13e,f. A clear difference from the prior city case studies is the increased number of landing sites that are available. In New York hundreds of flat rooftops offer viable landing site options. However it should be noted that quantity does not necessarily imply quality as many of these landing sites have high landing site risk. Case study 2 shows the fortunate situation where the UAS failure is next to a low risk landing site causing the elbow shape Pareto front in the last row/col in Figure 5.14. This point (marked in red) is selected by our planner because it provides the minimum total risk.

5.7.3 Urgent Landing Statistical Analysis

Two practical questions emerge from this study. First, how practical will it be for a small UAS to land in each analyzed city? Second, how quickly can the UAS identify a solution using the Algorithm 2 planner? Section 5.7.3.1 analyzes the minimum search radius needed to guarantee a landing site, offering a practical constraint on required UAS range for an urgent landing in a given

region. Section 5.7.3.2 analyzes key performance metrics of our proposed planner.

5.7.3.1 Minimum Radius Footprint

A search radius footprint, R, determines the maximum distance the planner will search for available landing sites. If this number is too small it is possible that no landing sites will be returned, potentially requiring an unsafe ditching / flight termination. It is therefore desirable to quantify the minimum radius footprint necessary to guarantee at least one landing site will be found anywhere in a mapped region. Figure 5.15 shows a planning area map of Ann Arbor, Witten, and New York in meters. All landing sites with a minimum radius of two meters are denoted by blue circles while the center of the red circle represents the point on the map farthest from any landing site. This point is found by finding the largest inscribed circle contained in the map that does not touch or contain any landing site. These maximum distances, d_{max}, are computed to be 123, 106, and 207 meters for Ann Arbor, Witten, and New York, respectively. Note that this technique does not account for points near the edge of the map which may be farther from landing sites (such as northeast Manhattan). Therefore to guarantee a landing site is found with $R \geq d_{max}$, the operating region of this city map must be shrunk by each city's respective d_{max}. Alternatively, one can set $R \geq 2 \cdot d_{max}$.

5.7.3.2 Performance Benchmarks

Monte Carlo simulations were performed to gather four key performance metrics of our proposed urgent landing planner: number of available landing sites in the landing footprint, database query time, multi-goal planning time, and number of landing site searched (k in Theorem 1). All results were computed using a desktop computer running an AMD 3900X 4.1 GHz processor. Each city had 500 uniformly sampled failure positions for which our framework provided a landing site and path with minimum total risk. Parameters were set to those from Table 5.3. Figure 5.16 displays a swarm plot with a box and whisker plot overlay showing results. Each data point is shown with the box capturing the inter-quartile range, the line in the box representing the median, and whiskers denoting 0-95th percentile. Outliers, if existing, are labeled in the top right of the graph.

Figure 5.16a shows the number of available landing sites in the reachable footprint for each of the cities. Ann Arbor has the least number of landings sites, followed by Witten and then New York. Manhattan in particular has the highest number of possible landings sites. In some sections of the borough there are more than 300 landing sites within reach; this is most often near small clustered flat buildings found in the Northeast map region per Figure 5.12c. However, in one particular case there is only one landing site available in Southern Central Park. In all simulations at least one landing site is available.

Figure 5.16b displays the number of milliseconds needed to query the database to provide

available landing sites $\mathcal{S}_{ls}$. The median time to execute this geospatial query is under 2 ms for all cities. New York once again has a longer tail distribution since hundreds of possible landing sites are returned. This query is made efficient through the use of R* trees for spatial indexing allowing for fast lookup in the database.

Figure 5.16c shows multi-goal planner execution time in milliseconds. Mean planning times are 2.0, 3.1, and 9.4 milliseconds for Ann Arbor, Witten, and New York, respectively. Although the mean is low, all cities have a long tail distribution, with New York requiring up to 166 milliseconds in one scenario. Some scenarios take longer than average because the 3D octile distance heuristic underestimates true path length particularly in New York due to of its many high rise buildings presenting obstacles to be avoided. The degraded heuristic affects the planner in two ways: A* path planning takes longer due to substantial search node expansion, and the multi-goal planner must search for more landing sites to prove the risk-optimal site is found. Figure 5.16d shows the number of landing sites searched by the multi-goal planner, i.e., the number of loop iterations in Algorithm 5.2. Each of these iterations requires an independent A* path planning procedure. The worst case is found in New York with 11 planning iterations requiring an overall multi-goal planning time of 166ms.

5.8 Discussion and Future Work

The presented landing site risk model evaluates risk to both the vehicle and nearby property. Vehicle risk currently includes terrain type, planarity, and area size metrics. However, future work should investigate load bearing capability of buildings, ingress and egress clearance and complexity, wind patterns, and the direction of abort paths [16]. The risk to overflown population could not be calculated in this chapter because population density data at high spatial resolution is not widely publicly available. Census records have low spatial resolution and do not accurately represent population distribution during the day. Anonymized call detail records can be used to augment this data to create temporal population models but are only available as sample datasets in a few regions [24]. Additionally, the population models must have high spatial resolution in order to discriminate between landing sites within the small radial footprints needed for sUAS urgent landing. This data will allow a more complete picture of the risk to humans especially when coupled with building sheltering factors [26].

The risk models presented in this chapter closely follow multiattribute utility theory while substituting utility maximization for risk minimization[169]. This theory assumes that the overall risk of a decision is the sum of the magnitude of each attribute multiplied by a risk score. These magnitudes are subjective and require thoughtful human determination. However, humans in urgent problem solving situations do not have time to consider all factors and often rely on simplified

heuristics or shortcuts [170]. Decisions generated with heuristics are often acceptable (satisficing), quickly determined, but non-optimal [171]. Results from this chapter indicate that our proposed multi-goal planner is quick, providing a risk-optimal landing site/path decision in less than 100ms. Future work should be performed to gather expert pilot opinions on whether the proposed metrics are sufficient and if additional metrics are needed. Participant will also rank, categorize, and group the most important attributes. Scenarios similar to the presented case studies can be shown where pilots will choose a landing site/path pair. A fully integrated visualization of 2D maps, 3D environments, and risk graphs can be presented to allow research participants to make informed decisions. The interface will be designed such that participants can dynamically adjust rankings and/or weights of attributes to visualize changing Pareto frontiers. The results of this work will inform attribute and weight definitions in our final risk models.

In the event of an emergency, our proposed emergency planner can operate locally and autonomously; a data-link for remote operator action is not required. However, it may be desirable to give a time-limited opportunity for a remote human operator to participate in the emergency response process. Some failure scenarios may pose high-risk toward humans requiring an immediate decision (i.e., sub-second) for landing. These situations do not allow elaborate human interaction with an emergency planner interface; we must prioritize the necessary speed of the autonomous system against the value a human operator may provide. A simple interface displaying the optimal landing site/path pair with a confirmation or "go" button may be used in such situations. These confirmation displays are often used in high risk situations, e.g., the Iron Dome autonomous weapons system where a human operator has less than one second to confirm the launch of intercepting missiles against incoming short-range rockets [172]. However, many sUAS failures will not pose immediate high risk to overflown populations such that prompt human feedback may be beneficial. Humans in this role should *not* focus on low level details of landing site identification or flight planning. Instead they act in a supervisory role by choosing the best course of action that is presented [173].

For this purpose, an intuitive user interface for our emergency planner should be carefully defined. The type, amount, and form of information presented should be balanced with cognitive strain humans encounter during time-sensitive, high risk, and uncertain situations [170]. Research indicates that humans in this "problem-solving" mode look for cues from data, perform hypothesis generation and selection, and finally action selection [171]. Therefore our user interface should be limited to elements that successfully aid humans through this decision making process. First, our emergency planner interface should have separate *pre-takoff configuration* and *emergency action selection* screens. The pre-takeoff screen allows a user to configure the mission specific constraints and attribute rankings for risk minimization during landing site selection. Constraints such as flight altitude, flight time, landing site distance, and landing site type (e.g., prepared or unprepared) should be able to be removed and added through the interface. Additionally, users may select an option to

bias landing site rank to those closer to a critical destination point (e.g., a hospital) rather than being near the sUAS itself. A slider can also be presented that changes the ranking of landing sites based on landing site or path risk metrics. This screen may be information dense as time is not a constraint during this pre-flight process. In contrast, the emergency action selection screen must afford quick operator selection. For example, showing all available landing sites, which may total more than 300, is not recommended because it is likely to overwhelm and distract the user. Instead, the top ranked sites should be presented to the operator to bring attention to the most likely of choices. Concise summary views of each landing sites and their paths should be presented. Future work should gather opinions of pilots and user interface design engineers to begin the process of both interface designs.

5.9 Conclusion

UAS operating in cities need to identify safe landing sites and associated paths in real-time whenever an urgent landing is required. This chapter proposed the use of nearby flat rooftops to augment traditional emergency landing sites such as parks and fields. We showed that fusion of deep learning for roof shape identification and computation geometry for flat surface extraction results in suitable landing site identification. Landing site locations and associated risks were stored onboard UAS along with obstacle and risk maps of the local flight area. While previous risk-based planners have been proposed, our map-based planner is the first to explicitly trade off landing site risk and path risk to minimize combined total risk. Our multi-goal planning algorithm efficiently selected the landing site/path pair guaranteed to minimize a weighted total risk function. Landing site databases and 3D risk maps were generated for three diverse cities with results presented from six case studies. Additional Monte Carlo simulations were run on all three cities to assess key performance metrics, showing that our planner finds risk-optimal landing sites and paths in less than 50ms for 95% cases. Worst-case execution time across our tests was 166ms in New York City. Future work can address this with a distributed or cloud-based path planner that offers speed up through parallelization. Additional risk factors such as wind patterns, rooftop material and strength, and dynamic population data will improve results. Ultimately, multiple UAS sensor data streams can be incorporated into mapping and real-time planning systems to refine risk databases.

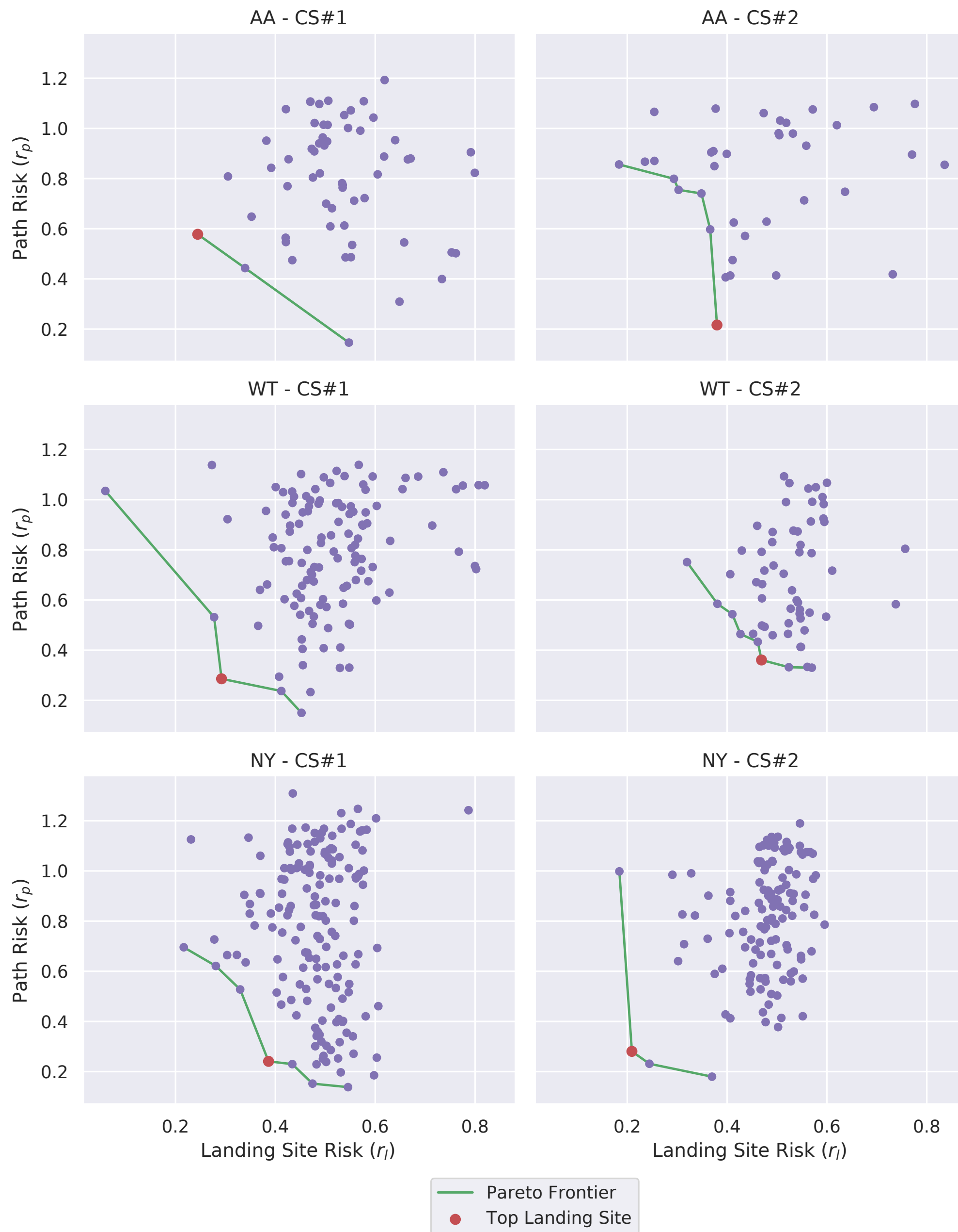
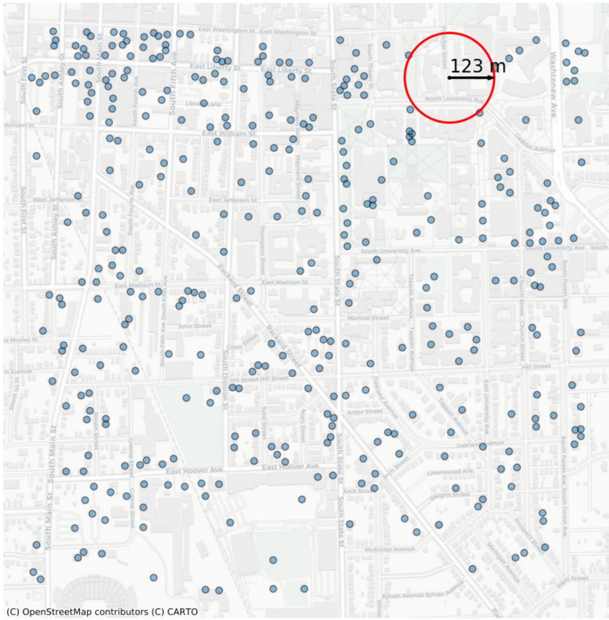
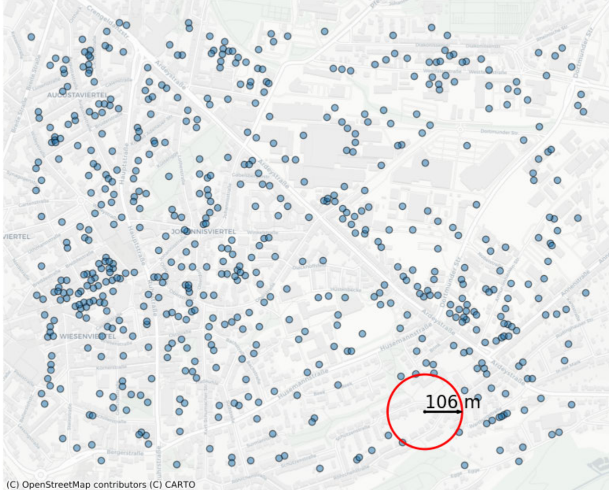


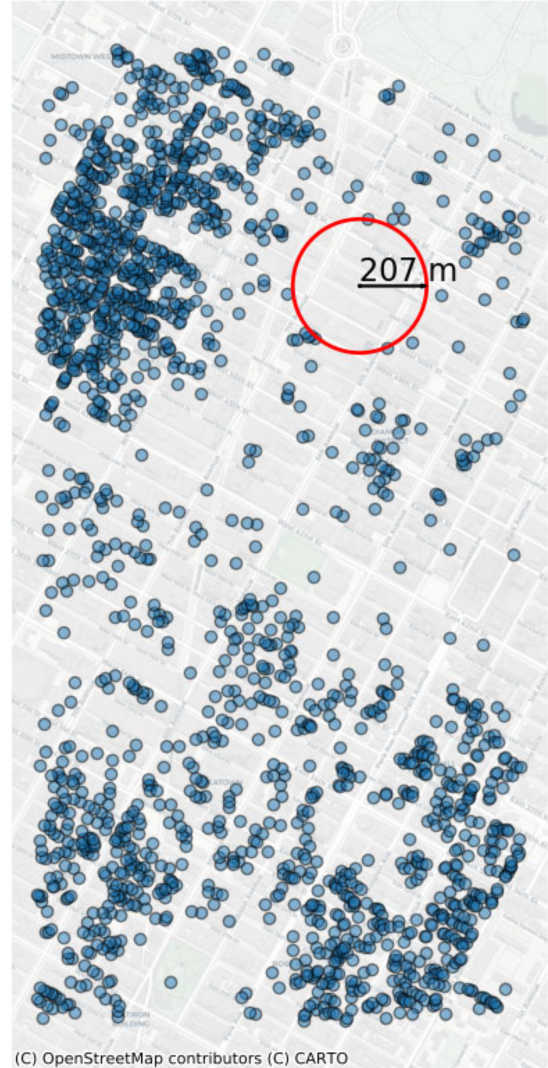
Figure 5.14: Pareto frontier of case studies. Simulation results for six case studies performed in Ann Arbor (first row), Witten (second row), and New York City (third row). The x and y axes are landing site risk and path risk, respectively. Each purple dots represents a landing site and its associated path, while the red dot signifies the planner's choice which minimizes total weighted risk.



(a) Ann Arbor



(b) Witten



(c) New York

Figure 5.15: Maximum distance between landing sites. Each landing site is displayed as a blue circle with the red circle center labelling the point farthest from any landing site. Maps from ©OpenStreetMap contributors and ©CARTO. License: Open Database License: https://www.openstreetmap.org/copyright

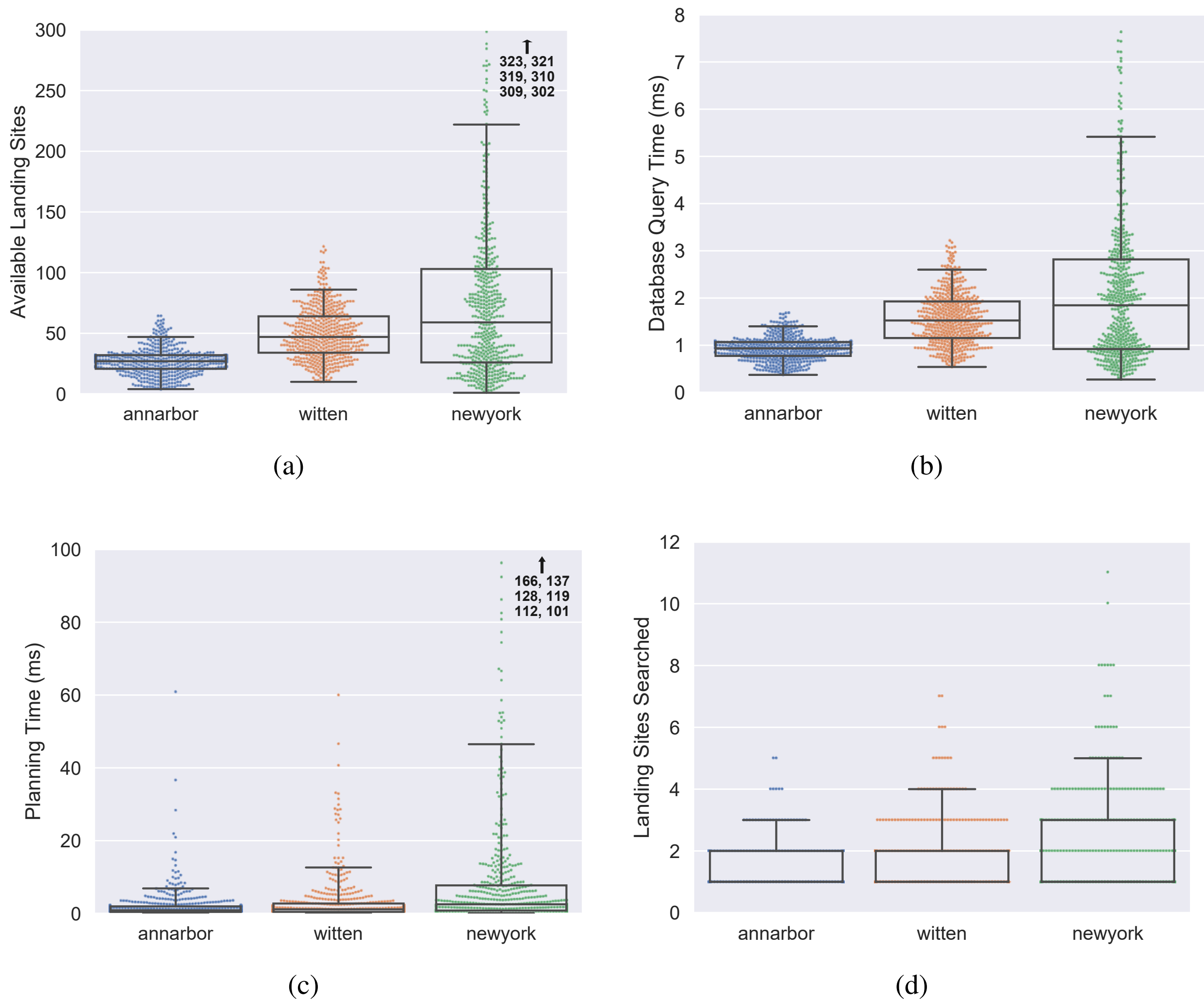


Figure 5.16: Metrics for map-based planner. Comparison between the cities of Ann Arbor, Witten, and New York in 500 random UAS initial positions with a search radius of 250 m. Each data point is shown with the box capturing interquartile range and whiskers denoting 0-95th percentile. Statistics are provided for number of available landing sites (a), time to query the database (b), time to find the risk-optimal landing site/path pair (c), and number of landing sites searched (d).

CHAPTER 6

Rooftop Touchdown Point Selection Using On-Board LIDAR and Vision

6.1 Introduction

Recent advances in small Unmanned Aircraft Systems (UAS) perception systems are beginning to enable safe three-dimensional navigation through complex uncertain environments for applications such as aerial photography, infrastructure inspection, search and rescue, and package delivery. Urban UAS operations will necessarily occur above buildings and over people. A safe urgent landing capability is a necessity, but no terrain-based or prepared vertiport landing option [35, 3, 24] may be available. In densely-populated urban regions, building rooftops can offer nearby safe landing zones for small UAS [15]. Urban roofs often have flat-like characteristics and are usually free from human presence [174]. However, landing on urban buildings provides unique challenges such as avoiding auxiliary structures hosted on each rooftop. Chapter 5 showed that a database of flat rooftops, their topologies, and optimal touchdown points can be computed and stored a priori from data such as satellite imagery and airborne survey point clouds [175]. However, a sUAS must identify a touchdown point on approach to an unprepared rooftop landing site to confirm the landing zone is clear or replan as needed.

This chapter proposes Semantic Polylidar3D, a suite of computational geometry (Polylidar3D) and deep neural network (semantic segmentation [176, 177]) algorithms to identify and select safe rooftop landing zones in real time using a combination of LiDAR and camera sensors. A high-fidelity simulated city is constructed in the Unreal game engine [178] with particular attention given to creating a statistically-accurate representation of rooftop obstacles that create obstructions to safe landing, e.g., water towers, vents, air conditioning units, rooftop building access doors. AirSim [179], a robotic vehicle simulator plugin for Unreal, generates onboard small UAS video and LiDAR data feeds as the small UAS navigates through the simulated Unreal environment. Semantic Polylidar3D fuses small UAS image and LiDAR data to compute the optimal obstacle-free touchdown circle on a rooftop within UAS sensor field of view. Figure 6.1 provides a graphical overview of processing steps. A LiDAR point cloud is classified by projecting data into a semantic image generated by a neural network. The classified point cloud is then rapidly converted into a

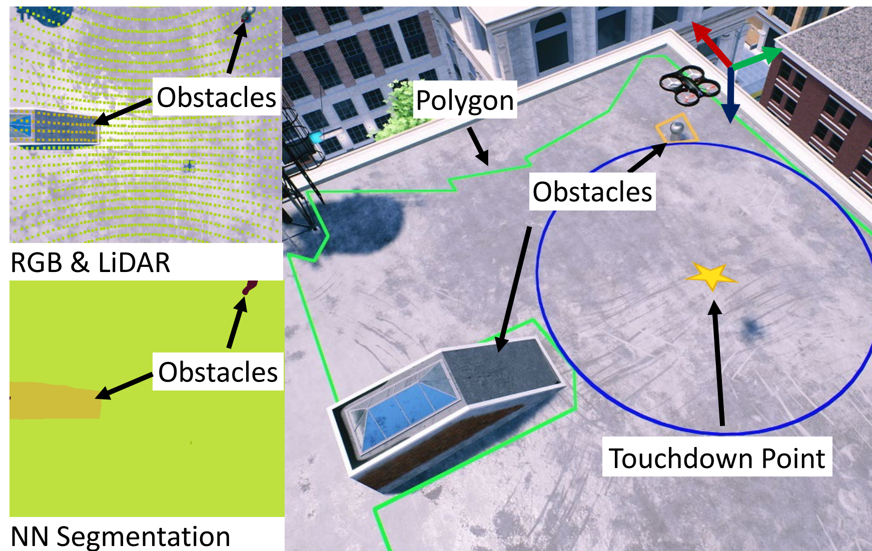


Figure 6.1: Overview of Semantic Polylidar3D for touchdown point selection. Camera RGB (red-green-blue) images are transformed into a segmented image through a neural network. LiDAR point cloud data is projected into the segmented image for classification. Flat surfaces are extracted with Semantic Polylidar3D from the *classified* point cloud as shown in the right image indicating a green candidate landing site polygon with orange interior "obstacle cutouts". The blue circle represents the largest flat, obstacle-free touchdown point in the polygon.

polygon representing clear landing area accounting for both geometric and semantic information. Key contributions include:

- Construction of a high-fidelity visual city model from real world data of the rooftops in midtown Manhattan, New York.
- A hybrid algorithm for planar extraction accounting for semantic information using computational geometry and deep learning.
- A novel method for finding optimal touchdown points on rooftops in a processing pipeline viable for real-time deployment.
- A comparative study of state-of-the-art semantic segmentation models to show their classification accuracy.

Below, a summary of related work (Section 6.2) is followed by a problem statement (Section 6.3) and definitions (Section 6.4). Section 6.5 describes our touchdown point selection procedure using Semantic Polylidar 3D. The urban rooftop simulation environment is presented in Section 6.6, and results are presented for semantic segmentation (Section 6.7) and the integrated Semantic Polylidar3D pipeline (Section 6.8). The chapter concludes with a discussion (Section 6.9) followed by a brief conclusion (Section 6.10).

6.2 Related Work

This section first summarizes the literature in aircraft unprepared landing site selection with focus on vertical takeoff and landing (VTOL) platforms including multicopter small UAS. Background in image pixel classification for semantic segmentation and polygon extraction is also provided.

6.2.1 Unprepared Landing Site Selection

VTOL aircraft have been flown extensively in unmapped and dynamic environments, historically with onboard radar and vision guiding approach to landing in manned helicopter operations [16]. Per [10] and [15], terrain landing sites are commonly investigated and still remain the primary alternative to runways/vertiports for VTOL aircraft. Rooftop landings have only recently been considered [15, 113, 175] since only emerging small UAS are sufficiently lightweight to land on a roof without risking structural damage or collapse. Regardless of touchdown site specifics, landing can be decomposed into three steps: landing site identification and selection, landing trajectory generation (flight planning), and flight plan execution [3, 180]. This chapter is specifically focused on real-time local perception for rooftop-based landing site identification and selection.

Cameras are a common sensors used for landing site identification. For example, Ref. [10] relies on monocular vision, an inexpensive lightweight option for small UAS. Monocular cameras can use structure from motion (SfM) to generate 3D point clouds of the environment to aid in scene understanding [181], but map accuracy is often limited from computational constraints. Ref. [15] uses stereo vision to aid in depth mapping, while Ref. [182] uses a custom LiDAR system to avoid obstacles and identify clear/flat terrain for approach and landing. Recently LiDAR sensors have become sufficiently lightweight and economical to be carried onboard small UAS. LiDAR provides precise range estimates to surfaces and can be directly transformed to 3D point clouds. However this precision comes at a cost increase, and point clouds may experience distortion when mounted on moving vehicles such as small UAS. Solid state LiDAR sensors with few to no moving parts reduce motion distortion, and offer corresponding reductions in weight and cost [183, 184, 185]. We assume next-generation small UAS tasked with accurately navigating a complex urban landscape will carry a camera and LiDAR.

Researchers often use predefined landing site markers and perform image feature matching to recognize the site [186, 11]. These algorithms use known landing site geometry patterns to robustly estimate relative state of the aircraft to guide it through a safe touchdown. Our work is focused on unprepared rooftop landing sites where markers will not be available so site identification and final approach guidance must be performed from natural environment features.

Ref. [187] presents methods for a return landing to the UAS' starting position in an unstructured environment. The work implements a visual teach-and-repeat method where images are recorded

during take-off and serve as known control/guide points in landing. During landing the drone localizes to these images and descends along a similar path back to its initial position. This procedure requires the drone to be above the original UAS take-off position, making it unsuitable for use in areas never before visited. Ref. [15] identifies candidate touchdown points on rooftops using a single camera to perform 3D scene reconstruction with structure from motion generating a disparity map of a rooftop. They note variance of the disparity map along the gravity vector corresponds to the planarity of the landing surface. Smaller changes in variance correspond to flatter surfaces. With this assumption the authors apply a kernel filter across the disparity image to identify pixels that are deemed planar, normalize the resulting image between [0,1] and perform Gaussian process smoothing. This algorithm is run over a downsampled image space to select the candidate pixel having the "flattest" region. This procedure for candidate landing site selection guarantees a minimum distance from obstacles but does not maximize this distance, instead optimizing over planarity.

Ref. [14] identifies terrain-based candidate touchdown points in an image plane from 2D probabilistic elevation maps generated over terrain. As in [15], a monocular camera using structure from motion provides depth information for each pixel. A height discrepancy filter is applied to the depth image to determine planarity, and a distance transform is applied to the image to select the flat pixel farthest away from any non-flat site (pixel). The computational complexity of the distance transform necessitates limiting the size of the map to 100X100 pixels at all altitudes.

Instead of representing surfaces as 2D discretized elevation maps one can instead extract continuous flat sections as polygons [59]. In Chapter 3 we developed Polylidar3D [188], an algorithm and accompanying open-source software for extracting flat surfaces as non-convex polygons with interior holes representing obstacles. Polylidar3D was used to find rooftop landing sites from archived airborne LiDAR point clouds in Chapter 5. This offline processing pipeline also found touchdown points on identified flat polygons that maximize distance to any edge or interior obstacle. This chapter combines Polylidar3D with a deep neural network to generate a semantic map of visible features from camera images and uses fused results for real-time touchdown point selection.

6.2.2 Semantic Segmentation

Semantic segmentation describes the process of associating each pixel of an image with a class label such as *sky* or *rooftop*. Fully convolutional networks (FCN) were first proposed for image semantic segmentation [189] to learn an end-to-end encoder-decoder model capable of segmentation. The encoder model is a deep CNN that extracts image features with multiple resolutions while the decoder model contains transposed convolutions (upsampling) to predict segmentations with different resolutions. U-Net [177] further takes advantage of high-resolution features by decoding

after each encoding CNN block. SegNet [190] is an encoder-decoder model that upsamples from a feature map by storing maxpooling indices from the corresponding encoder layer. Bayesian SegNet [191] improves this model by adding dropout layers to incorporate prediction uncertainties.

Other semantic segmentation work utilizes context-aware models such as DeepLab [192, 193] and temporal models [194]. These models have relatively high weights for mobile device applications compared to FCN-based methods. This work compares the performance of different combinations of lightweight CNN encoders and FCN-based decoders for urban rooftop image semantic segmentation.

6.2.3 Polygon Extraction from Depth Data

Convex polygons from RGBD images have been extracted by Ref. [60] and [70]. However, convex polygons cannot represent boundary concavities or account for holes in the polygon. Ref. [59] generated non-convex polygons from range images using region growing but ignored interior holes. Ref. [67] performed polygon extraction through boundary tracing of plane-segmented range images but also ignored interior holes.

Several methods can extract non-convex polygons with interior holes, e.g., [38, 39]. These methods strictly operate on 2D data requiring the 3D planar point cloud segments be projected to the best fit geometric plane to produce 2D point sets. Ref. [51] proposes this technique and the use of α-shapes to extract polygons. However this method requires computationally-expensive projection and Delaunay triangulation operations making it unsuitable for real-time applications. Polylidar3D is a faster parallelized non-convex polygon extraction method that also accounts for interior holes.

6.3 Problem Statement

This chapter investigates rooftop touchdown point selection to enable small UAS operating in urban environments to perform safe urgent landings when necessary. Figure 6.2 illustrates logic and data flow for the rooftop contingency landing problem. This framework unifies sensor-based and map-based planning methods previously connected in Ref. [4]. A small UAS requiring an urgent or emergency landing must perform landing site identification and touchdown point selection to assure landing will avoid or minimize risk to people, property, and the UAS. If an observed site is safe the UAS will plan and execute a trajectory to that site. If the area is not safe the on-board map will be updated and a map-based planner will identify an alternate site. Archived map data may be used to find an alternate minimum-risk landing site that may be beyond UAS sensor field of view. On approach to this alternate site, real-time map and touchdown point confirmation will occur. The cycle will repeat as needed.

This chapter builds on previous work in map-based planning with geographic information

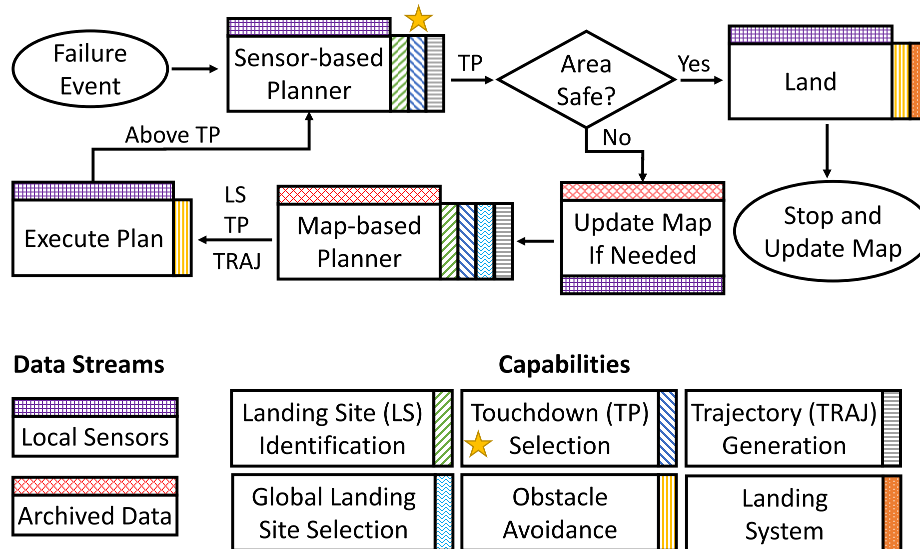


Figure 6.2: Rooftop-based contingency landing planning overview. Boxes with gold stars indicate research focus for this chapter.

system (GIS) data processed offline. In Chapter 5, GIS satellite imagery and airborne LiDAR point clouds were used to construct a database of flat rooftops and their associated optimal touchdown points [175]. Such a database can be loaded on the UAS before takeoff and used by a map-based planner to guide landing site selection and trajectory generation. This work builds on previous terrain-based landing site identification summarized above (e.g., [16]) to support complex flat rooftop sensor-based landing site confirmation and touchdown point selection using on-board LiDAR and camera sensors.

This work assumes the small UAS is equipped with a LiDAR sensor and monocular camera mounted underneath the vehicle. The small UAS also must carry a computer sufficient for sensor data processing and fusion. This work assumes the UAS has previously selected a rooftop landing site from onboard maps and that the UAS has executed an approach trajectory to a hover waypoint ten meters above the mapped touchdown point. This work proposes a process to integrate visual (camera) and depth (LiDAR) data streams to verify the landing site is safe (e.g., flat and clear) and adjust the touchdown point if needed. If a safe touchdown point is found a controlled landing will then be executed; otherwise the UAS must fly to an alternate mapped landing site. By converting depth data to planar surfaces and video data to surface type with semantic segmentation, the UAS can conform/identify a clear and suitable touchdown point in real-time. Because a number of semantic segmentation methods have been developed, comparative benchmarking over realistic datasets is required to quantitatively assess options. This work relies on simulation-based datasets designed to be statistically similar to rooftops in Manhattan since we cannot reasonably fly small UAS above Manhattan buildings to collect actual flight data.

6.4 Definitions

A scanning LiDAR that completes a full revolution generates a range image. This image has M rows denoting the number of beams in the vertical direction and N columns each representing a laser return in the full sequential scan. The range image can be converted to an organized 3D point cloud $\mathcal{P}$.

A *linear ring* is a consecutive list of points that is both closed and simple [43]. A linear ring must have non-intersecting line segments that join to form a closed path. A valid *polygon* has a single exterior linear ring representing the shell of the polygon and a set of linear rings (possibly empty) representing holes inside the polygon. The vertices of the polygon may be 3D points assuming all points lie on a 2D plane.

Figure 6.3 shows the reference frames defining vehicle body, camera, and LiDAR sensor placements and orientations. Vehicle body frame $\{\mathrm{B}\}$ has x-axis pointing forward, z axis pointing down, and y-axis completing a right-hand orthogonal frame. Camera frame $\{\mathrm{C}\}$ and LiDAR $\{\mathrm{L}\}$ have a translation offset of $\begin{pmatrix}-0.05 & 0 & -0.05\end{pmatrix}$ and $\begin{pmatrix}0.05 & 0 & -0.05\end{pmatrix}$ respectively in the body frame. LiDAR frame $\{\mathrm{L}\}$ follows the same conventions as body frame except rotated $90°$ about body y such that the LiDAR x axis points directly down (aligned with body z). These axes conventions are the standard reference designs provided by the AirSim plugin used for simulation-based tests. The reference frame of data will be indicated by a superscript, e.g., $\mathcal{P}^L$ denotes a point cloud in the LiDAR frame. A homogeneous transformation from frame A to frame B is denoted $\mathbf{H}_B^A$.

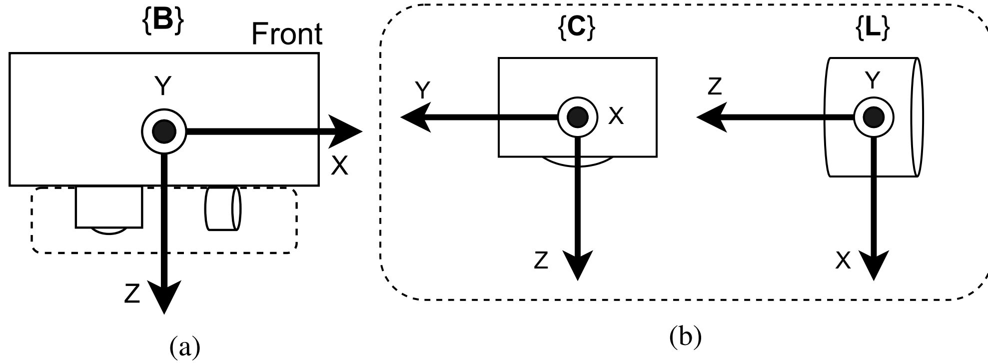


Figure 6.3: Reference frames from a side view. (a) UAS body frame $\{\mathrm{B}\}$. (b) Camera $\{\mathrm{C}\}$ and LiDAR $\{\mathrm{L}\}$ frames.

6.5 Touchdown Point Selection

Our proposed real-time touchdown point selection strategy is a hybrid algorithm with computer vision and computational geometry functions. Sec. 6.5.1 details the computer vision models used

for scene understanding while Sec. 6.5.2 discusses joining this information using polygon extraction. Finally, Sec. 6.5.3 details contingency planning steps used to identify landing sites.

6.5.1 Semantic Segmentation for Scene Understanding

Deep neural networks for computer vision can now accurately extract image features and segment images into semantic classes. Most semantic segmentation neural networks consist of two modules: convolutional neural network (CNN) backbones and meta-architecture elements. CNN backbones are feature extractors or encoder networks that downsample input images to obtain high-dimensional features. This chapter compares the performance of two backbone CNN networks: MobileNets [176] and ShuffleNet [195]. MobileNets are lightweight deep networks designed for mobile devices. A standard convolution operation is factorized into a depth convolution and a pointwise convolution, termed depthwise separable convolution. ShuffleNet generalizes depthwise separable convolution and group convolution to achieve an efficient CNN encoder for a mobile device. A channel shuffle operation is applied to realize the connectivity between the input and output of different grouped convolutions.

Meta-architectures are upsampling or decoder networks that reconstruct a segmentation image from downsampled feature maps. This chapter compares two meta-architectures based on [194]: FCN [189] and U-Net [177]. FCN combines CNN features from different depths of the encoder network during upsampling to utilize the information from a higher resolution image. The FCN model applied in this work combines feature maps from *pool3, pool4* and *conv7* layers to achieve better precision, known in FCN as stride 8 or FCN8s. U-Net takes advantage of the higher resolution feature by upsampling from each stage of the CNN encoder. At the end of each CNN block, the feature map is both input to the next CNN block and combined with the upsampled feature map. Upsampling continues until a final segmentation map is created. Our work implements and evaluates different image semantic segmentation models [194] on a desktop platform with an RTX 2080 graphics processor.

6.5.2 Semantic Polygon Extraction

Chapter 3 introduced Polylidar3D which transforms organized point clouds into meshes and extracts planar segments. Planar segmentation is a region growing process where a seed triangle is chosen and edge-connected triangles are expanded based solely on planarity constraints. Polygon extraction is then performed for each planar segment. This chapter introduces extensions to utilize semantic information during the region growing process; we refer to this upgraded package as Semantic Polylidar3D. The process has three steps:

1. Classify Points: The LiDAR point cloud is projected into a semantic image specifying the

class of each pixel.

2. Semantic Polylidar3D: Polygons are extracted from rooftops utilizing both geometric and semantic data.

3. Touchdown Site: The largest inscribed circle containing only viable landing polygons is defined.

6.5.2.1 Classify Points

Each point in organized point cloud $\mathcal{P}^L$ is transformed to the camera frame and subsequently projected to the semantic image per

$$p^C = \mathbf{H}_C^L \cdot p^L \;\; u = f_x \frac{x^C}{z^C} + c_x \;\; v = f_y \frac{y^C}{z^C} + c_y \tag{6.1}$$

where $p^L = (x, y, z, 1)^T$ is a point in the LiDAR frame, $\mathbf{H}_C^L$ is the transformation matrix, (f_x, f_y) are camera focal length values, (c_x, c_y) are camera principle point offsets, and (u, v) are camera pixel coordinates. The pixel position allows each point to have a class assignment from the semantic image. Some projected points may be outside the camera image and are classified as an "unknown" class, e.g. $u, v < 0$. Class assignments are stored in auxiliary data structure $\mathcal{C}$ of length $\mathcal{P}^L$.

6.5.2.2 Semantic Polylidar3D

Classified organized point cloud $\mathcal{P}^L$ is rapidly transformed into a half-edge triangular mesh $\mathcal{T}$ per procedures in Chapter 3. The vertices of this mesh have corresponding classifications from $\mathcal{C}$. Fig. 6.4a shows these colorized classifications where green, orange, and blue denote rooftop, obstacle, and unknown classes, respectively. The plane normal of the rooftop, $\hat{n}_r$, is estimated using a Gaussian Accumulator and shown as a red arrow in Fig. 6.4a. Mesh triangles are then filtered by geometric and semantic constraints per Algorithm 6.1. The algorithm first loops over all triangles and calculates each triangle's maximum edge length (l_t), angle between its normal and the roof normal (θ_t), and number of vertices in the rooftop and unknown class (lines 4-7). If the sum of rooftop and unknown vertices matches or exceeds $vert_{min}$ then semantic constraints pass (line 8). Next, the algorithm dynamically reduces the angular geometric constraint if the triangle belongs to the rooftop class (lines 9-12). This means a high confidence in semantic information may reduce the confidence threshold requirements for planarity, allowing slightly more noisy data to pass geometric constraints shown in line 13. Finally both constraints must pass for the triangle to be included in the filtered triangle set $\mathcal{T}_f$ shown as shaded blue triangles in Fig 6.4a. A polygon is then extracted from the planar mesh segment using methods from Chapter 3.

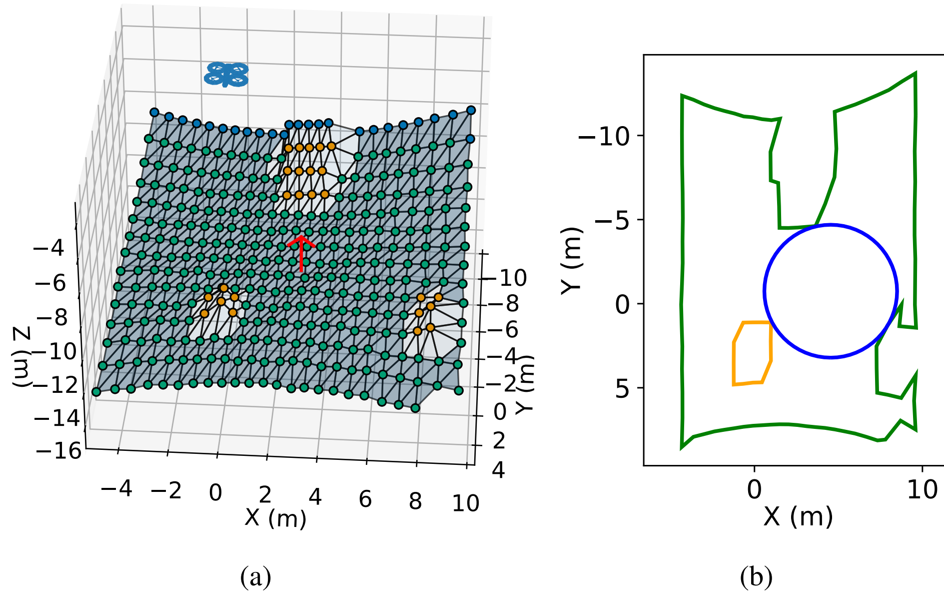


Figure 6.4: Visualization of Semantic Polylidar3D. (a) Classified LiDAR point cloud with triangular mesh: green is rooftop, orange is obstacles, blue is unknown. Triangles meeting semantic and planarity constraints are shaded light blue. (b) Polygon extraction from planar mesh. Green is hull, orange are interior holes, and blue is the touchdown site.

6.5.2.3 Touchdown Site

Several polygons representing disjoint flat surfaces may be returned from Sec. 6.5.2.2. These polygons may be filtered by area size, shape complexity, or even distance from the UAS. Currently the polygon with the largest area that is nearest to the drone is selected. The candidate touchdown site is determined by finding the center of the largest inscribed circle in the polygon. The largest inscribed circle in the polygon maximizes the distance between the exterior hull and any obstacles within the polygon [175, 159]. We use the software polylabel to perform this function [160]. Fig. 6.4b shows the greatest inscribed circle (blue) inside the polygon. A touchdown site is considered safe if it meets an aircraft-specific minimum radius constraint for safe landing clearance. If no safe touchdown site can be found then contingency planning to a new site must be performed as described below.

6.5.3 Contingency Planning Overview

If no touchdown site can be found at the initial UAS-approached rooftop then the UAS must land elsewhere and updates to inaccurate map data of that rooftop can be proposed. This may occur because of structural changes, temporary objects being placed on the surface, or dynamic obstacles (e.g., people) being present. Control authority must then switch from the sensor-based planner to the

Algorithm 6.1: Semantic Triangle Filtering

Input : Triangles: $\mathcal{T}$, Points: $\mathcal{P}^L$, Class: $\mathcal{C}$, Rooftop Normal: $\hat{n}_r$
Geometric Constraint Parameters: $l_{max}, \theta_{min}, \theta_{max}$
Semantic Constraint Parameters: $c_r, c_{uk}, vert_{min}$

Output : Filtered Triangle Set, $\mathcal{T}_f$

1. $k = |\mathcal{T}|$
2. $\mathcal{T}_f = \emptyset$
   `/* Loop through every triangle */`
3. **for** $t \leftarrow 0$ **to** k **do**
4. $\quad l_t$ = GetMaxTriangleLength$(t, \mathcal{T}, \mathcal{P})$
5. $\quad \theta_t = \arccos(\hat{n}_t \cdot \hat{n}_r)$ `/* triangle and roof */`
6. $\quad vert_r$ = CountVertices$(\mathcal{C}, c_r, t)$
7. $\quad vert_{uk}$ = CountVertices$(\mathcal{C}, c_{uk}, t)$
   `/* Check Semantic Constraint */`
8. $\quad$ `semantic_pass` = $vert_r + vert_{uk} >= vert_{min}$
   `/* Update Angular Geometric Constraint */`
9. $\quad$ **if** $vert_r >= vert_{min}$**:**
10. $\quad\quad \theta_{req} = \theta_{max}$
11. $\quad$ **else:**
12. $\quad\quad \theta_{req} = \theta_{min}$
    `/* Check Geometric Constraint */`
13. $\quad$ `geometric_pass` = $l_t >= l_{max}$ **and** $\theta_t < \theta_{req}$
    `/* Must pass both constraints */`
14. $\quad$ **if** *semantic_pass* **and** *geometric_pass***:**
15. $\quad\quad \mathcal{T}_f = \mathcal{T}_f + t$
16. **end**
17. return $\mathcal{T}_f$

map-based planner as shown in Figure 6.2. The map-based planner will then select a new landing site and touchdown point that optimizes both travel distance and landing site suitability. The process then repeats with the sensor-based planner verifying the landing site during each approach.

6.6 Simulation Environment

6.6.1 Analysis of Rooftops

Before constructing the simulation environment, an analysis of rooftops in Manhattan was performed. Since our work is focused on flat rooftop landing sites, only flat-like roofs in Manhattan were sampled. Data was collected manually by inspecting high resolution satellite and aerial imagery of buildings and recording the rooftop assets and associated quantities observed. Figure 6.5 shows the locations of 112 buildings randomly chosen from Manhattan near the Southwest corner

of Central Park. The data are released in [196].

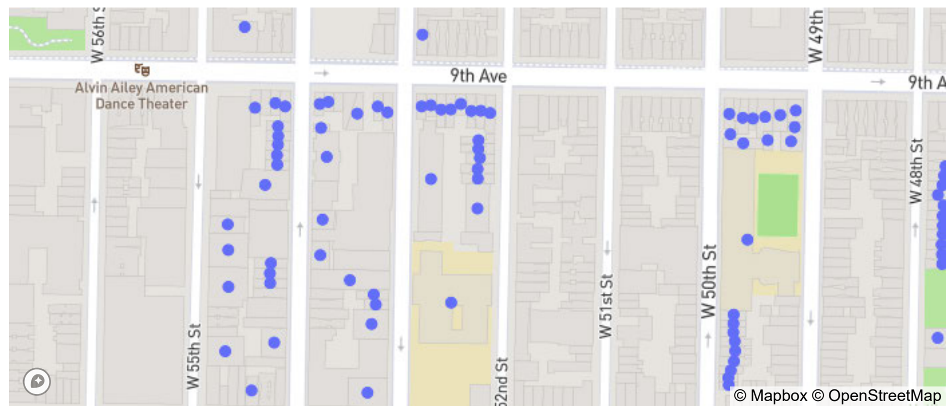


Figure 6.5: Map of Manhattan buildings observed for rooftop assets. Each blue circle represents a rooftop used for asset analysis.

Table 6.1 lists the 12 most common object types found on a building rooftop in midtown Manhattan and the average quantity observed. If a building does not contain an asset its quantity is recorded as zero. The full histogram of rooftop assets is shown in Figure 6.6 with a logarithmic vertical axis scale.

Table 6.1: Common rooftop items with average quantities

Item	Mean Quantity
air-vents	1.12
small-rooftop-entrance	0.88
skylight	0.51
small-building	0.45
ac-unit	0.28
seating	0.12
air-ducts	0.11
water-tower	0.10
chimney	0.05
enclosed-water-tower	0.04
tarp	0.03
vegetation	0.02

6.6.2 Generating City Rooftop Environments

Game assets for urban city buildings were purchased from [197], and high quality rooftop assets (e.g., ac-units, water towers) were purchased from [198]. Assets were modified to support configurability in textures and material properties to enhance world diversification. For example, an

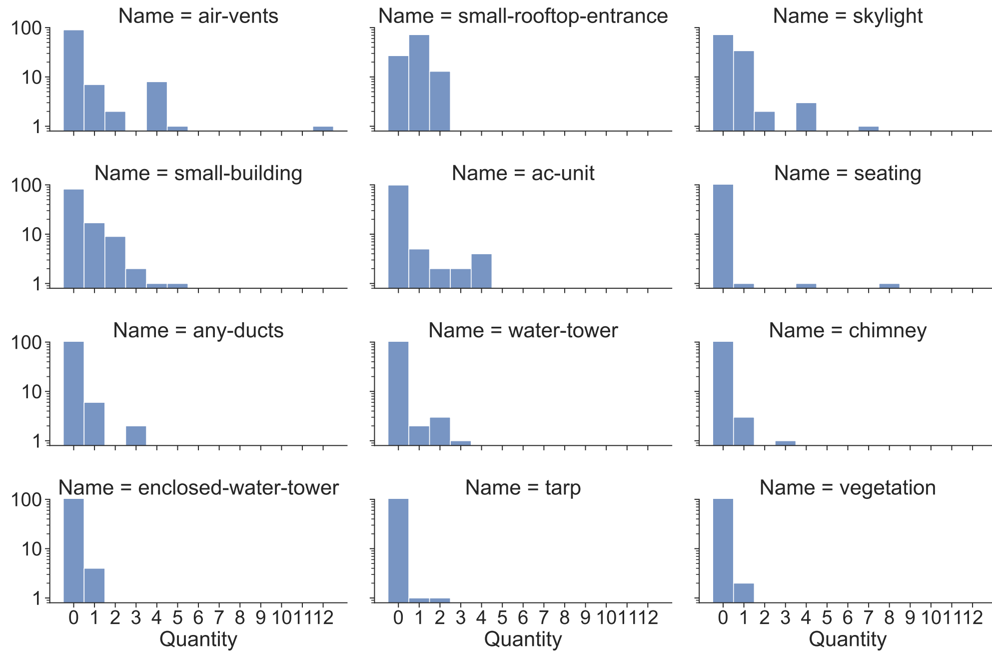


Figure 6.6: Histogram of twelve common rooftop items observed from a Manhattan dataset.

air-vent can be configured to take on a variety of different metal textures and reflectivity properties. Figure 6.7 shows a small sample of the diversity in asset classes. A base city was constructed with 33 buildings having different building textures, sizes, and shapes. This base city served as a starting template to generate random worlds per a world generation script described below. These worlds served as the basis for training and testing purposes. Note that for this work diversity is focused on building rooftop assets, not the buildings themselves.

A map of the base world colorized by height is shown in Figure 6.8a. Each polygon represents the flat surface of a rooftop with known height. A world generation script takes as input this map as well as an asset configuration file and places assets on each rooftop in a new 3D world. This script supports configurable options including: probability and quantity of asset placement, spatial location on the rooftop, asset orientation, and appearance properties (e.g. materials, textures, meshes). The distribution curve for new asset placement is assumed independent of assets already placed on a building for simplicity. The quantity of assets can either be configured to follow a uniform distribution or model the histogram shown in Figure 6.6. Each world is seeded with a different number to create diverse and reproducible worlds. Figure 6.8b shows an example of building assets being placed randomly on a roof per the world generation script. The world generation script is released with this manuscript and may be used in any Unreal Engine project [196].

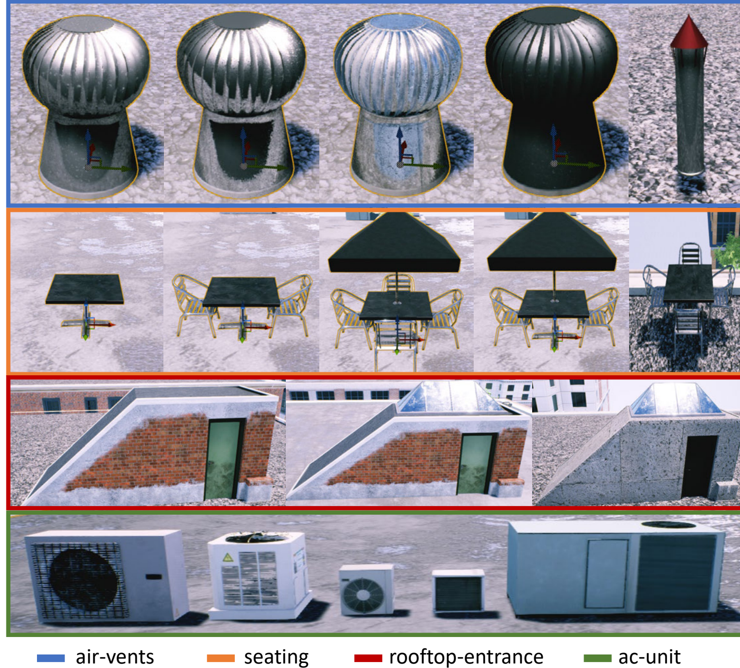


Figure 6.7: Examples of rooftop asset modelling and customization. Four rooftop assets (air-vents, seating, rooftop-entrance, ac-unit) and a subset of their customization in random worlds are shown. Metallic, texture, and static mesh properties can be altered for each asset type.

6.6.3 Vehicle, Camera, and LiDAR Models

The vehicle model and simulator physics engine were provided by the AirSim plugin for Unreal Engine. The model treats an aircraft as a rigid body with k actuators generating forces and torques. Details of the model and physics engine are found in Ref. [179]. AirSim generates UAS-based camera and LiDAR sensor data feeds. The camera uses a pinhole camera model with configurable random noise. The LiDAR sensor is modeled as a spinning set of n_l beams distributed equally within a vertical angular field of view (VFOV) and rotates clockwise within a horizontal field of view (HFOV). Simulated beam distance is calculated exactly and perfectly using ray casting. The authors found this model insufficient because it lacked noise and did not output an organized structure of point cloud data (i.e., range image) as spinning LiDAR systems provide. Therefore the LiDAR model was modified to resolve these two issues as described below.

Figure 6.9 shows the LiDAR and error model developed in this work. Each beam is defined by spherical coordinates with a range (d), azimuth angle (θ), and fixed elevation angle (ϕ). A properly-calibrated spinning LiDAR system has many forms of error including but not limited to range and encoder noise. Drawing inspiration from Refs. [58, 199], our LiDAR model assumes the radial and azimuth error is distributed by zero-mean Gaussian noise, e_d and e_θ, respectively. Ref. [200] verifies this assumption holds for an angle of incidence below a critical angle, found to be

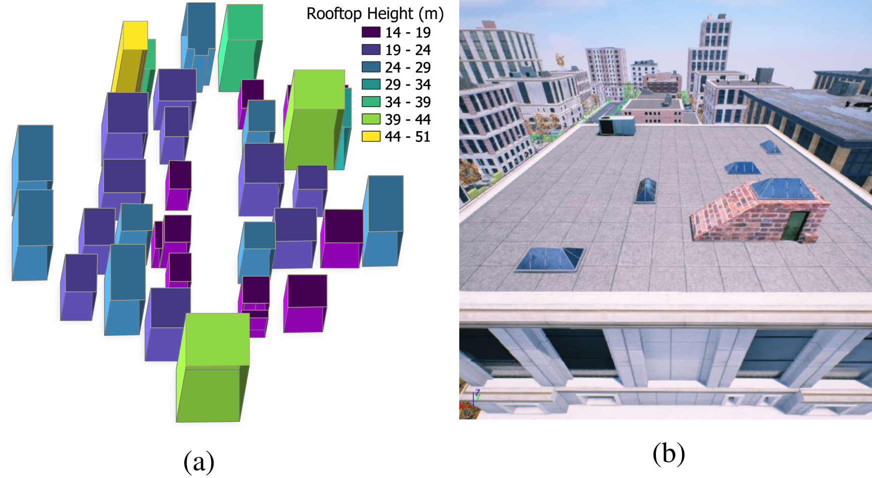


Figure 6.8: Example simulated urban city. (a) Map of the 33 base city flat rooftops colorized by height. (b) Example of randomized asset placement from world generation script.

$\sim 65°$. This reference also showed a calibrated 64-beam Velodyne sensor had range and azimuth RMSE of approximately 3.2 cm and $0.03°$, respectively. We adopt similar values in our simulations.

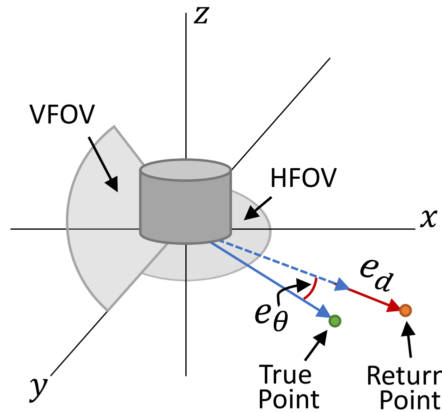


Figure 6.9: LiDAR model used in simulation. Error model account for range (e_d) and azimuth (e_θ) error.

The simulated LiDAR used in our work is configured with $n_l = 64$ beams with a VFOV = $(-45°, 45°)$ and a HFOV=$(-45°, 45°)$. A scan is collected after the full HFOV is traversed and recorded as an organized point cloud, with n_l rows and n_c columns, where n_c is fixed and determined by the scanning and rotation rate:

$$n_c = \frac{90°}{360°} \cdot \frac{\text{PointsPerSecond}}{\text{RotationsPerSecond}} \cdot \frac{1\text{Beam}}{64\text{Beam}}$$

Rotation rate was set to 20Hz with 655360 points per seconds as modeled from an Ouster OS0

sensor[201]. Note that this LiDAR model, like Refs. [58, 199], does not account for LiDAR motion distortion for a moving vehicle. However, motion distortion can be removed through efficient processing using feature scanning with an on-board inertial measurement unit (IMU) [202, 203]. Additionally, this work assumes the drone is in a stable hover such that it can be ignored in this model.

6.7 Semantic Segmentation Results

6.7.1 Creating Image Dataset

Training, validating, and testing a neural network to segment rooftops and obstructions requires a large annotated image dataset. To accomplish this we first generated eight random worlds used for training the neural network. The world generation script was seeded with different numbers generating different random worlds. Each world assumed equal likelihood for all asset placement leading to rich and diverse rooftop assets. Note that the buildings and lighting conditions are held constant, only rooftop assets are different in each world. An image collection script was created that captured images of rooftops and their obstructions with corresponding ground truth segmentation labels. The images are captured at a variety of positions and orientations pointed at the center of the rooftops as shown in Figure 6.10. The blue arrows represent the position and direction of the camera while the green rectangle denotes the rooftop. A sphere centered at the rooftop with a radius five meters greater than the rooftops radial footprint fixes these sampling configurations. A total of 11,989 images were collected and split 80/20 into a training and validation set, respectively. The test set was created in a similar manner except from random worlds following the asset quantity distribution recorded from the Manhattan dataset. A weighted sampling procedure was used where the weights were directly used from the data histogram per [204]. A total of 5,465 images were collected for the test dataset.

6.7.2 Training and Testing Results

Implementation details: We evaluated four combinations of CNN backbones and decoders for image semantic segmentation: MobileNet + FCN8s, ShuffleNet + FCN8s, MobileNet + UNet, and ShuffleNet + UNet. We modified models based on tensorflow implementations [194] and perform training and testing on a system with an Nvidia RTX 2080 GPU. Each model was trained for 100 epochs with early stopping enabled using the validation dataset.

Metrics: We evaluate mean intersection over union (IoU) and per-class IoU for each method on the test dataset.

Quantitative results: As can be seen in Table 6.2, the MobileNet + UNet model achieves the best

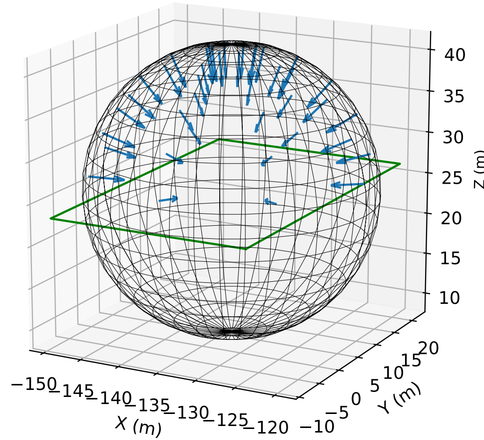


Figure 6.10: Image sampling strategy for creating an annotated dataset of rooftop segmentation. Blue arrows denote the position and orientation of the camera. The green polygon denotes the rooftop.

mean IoU and the best per-class IoU on most cases while MobileNet+FCNs performs the second best in most Table II cases. Specifically both models outperform the ShuffleNet based methods on small-rooftop-entrance, skylight, air-vents and ac-units which appear frequently in the real world and are more important to rooftop landing tasks. The trained MobileNet + UNet model is chosen for performing semantic segmentation to evaluate our proposed methods.

Table 6.2: The per-class IoU and mean IoU of different image semantic segmentation networks on our urban rooftop dataset. The top mean IoU is highlighted in bold. The chimney class is absent from the test dataset so its IoUs are not available.

	sky	ground	building wall	building rooftop	small rooftop entrance	sky light	air-vents	ac-unit	seating	air-ducts	water tower	tarp	vegetation	Mean IoU
MobileNet + FCN8s	0.99	0.93	0.96	0.98	0.82	0.79	0.48	0.78	0.42	0.80	0.74	0.90	0.81	0.74
ShuffleNet + FCN8s	0.99	0.92	0.95	0.97	0.78	0.76	0.41	0.75	0.37	0.73	0.68	0.84	0.79	0.71
MobileNet + UNet	0.99	0.93	0.97	0.98	0.83	0.84	0.50	0.81	0.47	0.81	0.78	0.84	0.90	**0.76**
ShuffleNet + UNet	0.99	0.94	0.96	0.98	0.79	0.79	0.36	0.77	0.34	0.74	0.76	0.91	0.88	0.73

6.8 Touchdown Point Selection Results

Our proposed touchdown point selection procedure was evaluated in a newly generated simulated city environment not used for training the semantic segmentation neural network. The rooftop assets are newly randomized and follow the asset quantity distribution of Manhattan. The full parameters used for Semantic Polylidar3D are shown in Table 6.3. Please see Chapter 3 for a full explanation of

selected parameters. Note that θ_{max} and $vert_{min}$ are new parameters governing semantic integration in polygon extraction as described in Section 6.5.2.2. The LiDAR model was configured with 64 beams/channels with range and azimuth error of 5 cm and 0.1 degrees, respectively. The camera image size was 500 $\times$ 500.

Table 6.3: Semantic Polylidar3D parameters

Algorithm	Parameters
Laplacian Filter	λ=0.65, kernel=3, iterations=1
Bilateral Filter	σ_l=0.3, σ_a=0.2, kernel=3, iterations=4
FastGA	level=5, v_{min}=50, d_{peak}=0.28
	$sample_{pct}$=50%
Plane/Poly Extr.	tri_{min}=500, l_{max}=1.5, θ_{min}=0.96
	$ptp_{max} = 0.2$, $vertices^{hole}_{min}$ = 4
	$\boldsymbol{\theta_{max} = 0.90}$, $\mathbf{vert_{min} = 2}$
Poly. Filtering	$\alpha = 0.25$, β_{pos} = 0.1, β_{neg} = 0.25,
	γ = 4, δ = 0.1

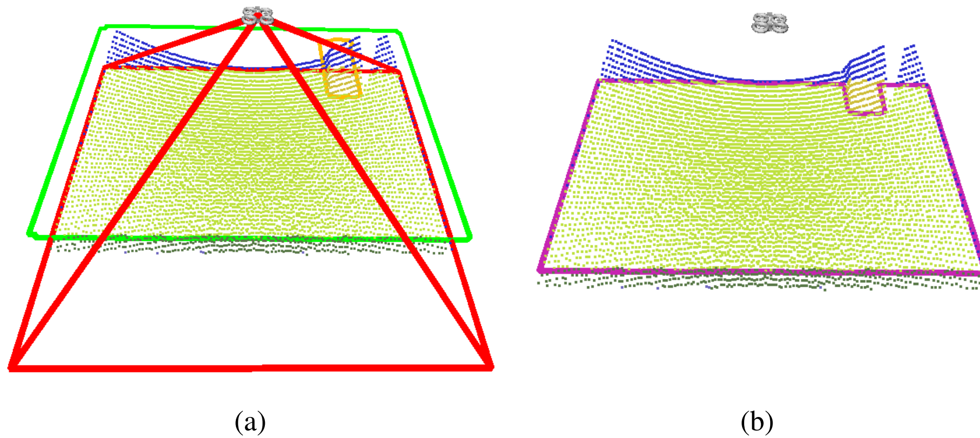
Two analyses were performed that quantified the accuracy and speed of Semantic Polylidar3D and evaluated the maximum height before obstacle identification failures occurred (decision height). Each is described below.

6.8.1 Semantic Polylidar3D Accuracy and Speed

6.8.1.1 Creating Test Data

LiDAR and camera image data were collected at 10 meters above each of the 33 rooftops. For each rooftop, the drone was positioned and orientated in five different configurations. One configuration is at the center of the rooftop with the drone aligned with the x-axis. The other four configurations are offset from the rooftop center ± 5 meters in both the x/y axes with the drone facing towards the rooftop center. The drone is assumed in a stable level hover with sensors pointing down as described above. Each configuration gathers two samples, providing a total of 330 independent samples. A sample consists of a LiDAR scan and an image from the monocular camera. The LiDAR has random noise making the two samples independent. To assess accuracy, ground truth polygons including obstacles were required for each rooftop. However, the field of view of the LiDAR and camera sensors are limited thus may not contain the full rooftop. Fig. 6.11a shows the ground truth polygon of the rooftop (green/orange), the classified point cloud, and the FOV of the camera sensor (red frustum). To resolve this issue the ground truth polygon is clipped to the camera field of view as shown in Fig. 6.11b. This same clipping procedure is also performed on the output of predicted polygons. This allows any method to be fairly assessed from data within

the the camera FOV. Accuracy is assessed as the Intersection over Union (IoU) of the predicted polygon and the clipped ground truth polygon.



(a) (b)

Figure 6.11: Gathering test data in simulation environment. (a) Rooftop ground truth polygon (exterior=green,holes=orange), classified point cloud, and camera FOV frustum (red). (b) The ground truth polygon is clipped (purple) to be inside the frustum.

6.8.1.2 Qualitative Results

Figure 6.12 displays three examples of our touchdown point selection process, each with two images. The left image shows the neural network semantic classification map while the right image displays the camera image overlaid with polygons. Semantic Polylidar3D polygons are shown in green/orange (orange is interior obstacles); the clipped ground truth polygons are indicated by a dashed purple line, and the center of the blue circle represents the optimal touchdown point. A safe touchdown point is found in each of these examples. Generated polygons map reasonably to ground truth polygons but may overestimate the size of large obstacles such as the rooftop entrances seen in (c). This occurs when obstacles occlude perception by the camera and LiDAR sensors, leaving a "shadow" of missing information behind them. Our method is conservative in that such regions will not be considered touchdown options.

Fig. 6.13 demonstrates a challenging scenario where Semantic Polylidar3D does an excellent job of correctly identifying a landing site and selecting a touchdown point. This building has a rooftop-entrance as well as a slightly non-planar (wrinkled) tarp on its surface as shown in (a). The rooftop-entrance has a concrete-like texture that is similar to the texture of the rooftop itself. At a distance (e.g., 10 m) they look nearly identical which causes a neural network segmentation failure as shown in (b). However, because both LiDAR and vision are used in the Semantic Polylidar3D algorithm, failure in one modality did not lead to a failure in identifying the rooftop entrance. The orange obstacle line fully encapsulates the rooftop-entrance as shown in (c). In addition, the neural

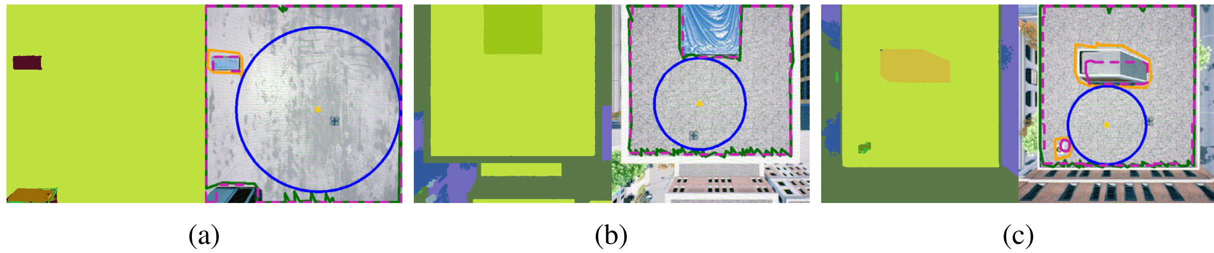


Figure 6.12: Three examples of touchdown point selection on rooftops. Predicted segmentation and camera image with overlaying projected polygons are shown on the left and right, respectively. Dashed purple lines represent ground truth polygons while green/orange represent the predicted polygon. The center of the blue circle is the selected touchdown point.

network was able to correctly identify the tarp allowing Semantic Polylidar3D to exclude the tarp from the landing site polygon. The tarp is outside the polygon boundary in (c). Baseline Polylidar3D shown in (d) only uses LiDAR and is not able to fully identify the tarp obstacle because its planarity is similar to the ground. Only a small section on the border is captured as an interior hole. This example shows Semantic Polylidar3D's increased robustness to neural network errors in vision and range error from LiDAR by fusing modalities during polygon extraction.

6.8.1.3 Accuracy Assessment

The IoU of the predicted polygon and ground truth polygon (clipped to camera FOV) is computed for each sample to assess accuracy. We also compare these results with baseline Polylidar3D which only uses LiDAR data in flat surface extraction. Figure 6.14 shows overlaying histograms and kernel density estimators of the IoU results where blue and orange denote the baseline against our new proposed method, respectively. Above the histogram are the respective mean and σ error bars. Semantic Polylidar3D has a mean and sigma of $91.2 \pm 4\%$ against baseline Polylidar3D with $86.8 \pm 6\%$. A T-test was conducted between each group giving a t-statistic of 11.0 with a p-value $<$ 0.001. These results indicate a substantial improvement in touchdown point selection with Semantic Polylidar3D's integration of computer vision and computational geometry methods.

6.8.1.4 Execution Time

Table 6.4 displays the mean and 1σ execution time in milliseconds for all major steps in the proposed touchdown point selection algorithm. The GPU accelerated neural network segmentation is the most time consuming portion. Note that no optimization techniques have been performed such as quantizing the segmentation model or reducing the number of classes. The total execution time of the method is sufficiently low to be executed in near real-time with UAS sensor data streams.

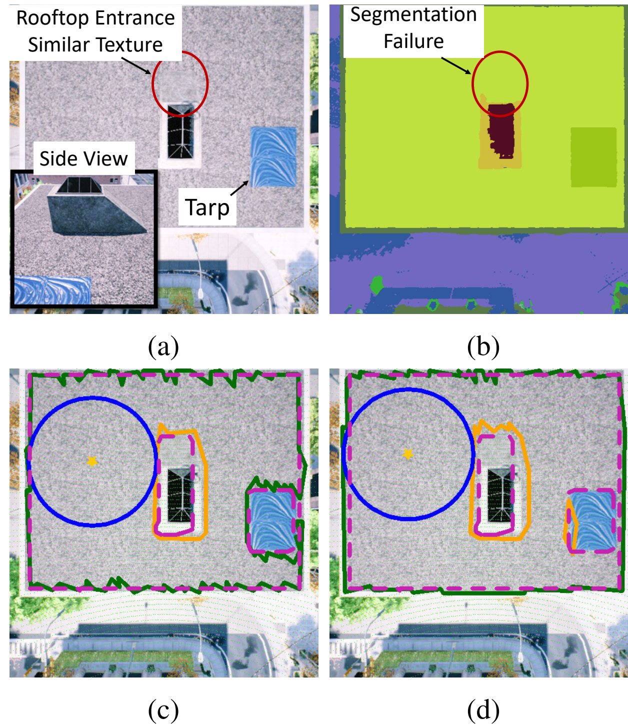


Figure 6.13: Example of Semantic Polylidar3D on a challenging rooftop. (a) Rooftop with tarp and rooftop-entrance. The entrance texture is similar to the rooftops. (b) Neural network semantic segmentation. (c) Proposed Semantic Polylidar3D polygon extraction. (d) Baseline Polylidar3D fails to identify the tarp.

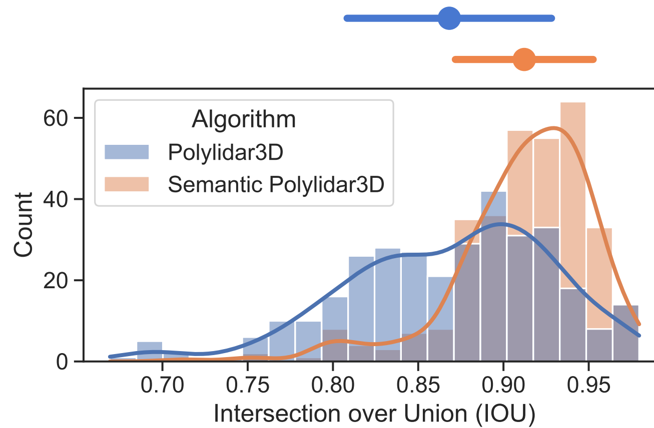


Figure 6.14: Comparison of Semantic Polylidar3D IoU accuracy. Polylidar3D (blue, baseline) versus Semantic Polylidar3D (orange, proposed). A higher IoU indicates the algorithm captured more of the surface and correctly identified obstacles. Mean and 1σ bars are shown on top.

Table 6.4: Execution time (ms)

Segment Image	Classify Points	Semantic Polylidar	Touchdown Site	Total
21.2 ± 1.1	1.7 ± 0.1	9.5 ± 1.5	0.1	32.6 ± 1.8

6.8.2 Decision Height Analysis

Landing decision height for VTOL aircraft refers to the height needed to limit glide slope and trajectory tracking errors to smoothly transition from approach to a stable hover over a touchdown point [205]. This height depends on vehicle performance characteristics, local environmental conditions (e.g. wind speed), and sensor errors. The touchdown point selection algorithm proposed in this manuscript must handle some error from both visual and range sensors as well as neural network segmentation. We performed a series of experiments that determined the maximum hover height at which our proposed algorithm can accurately identify a human on a rooftop as an obstacle as shown in Figure 6.15a.

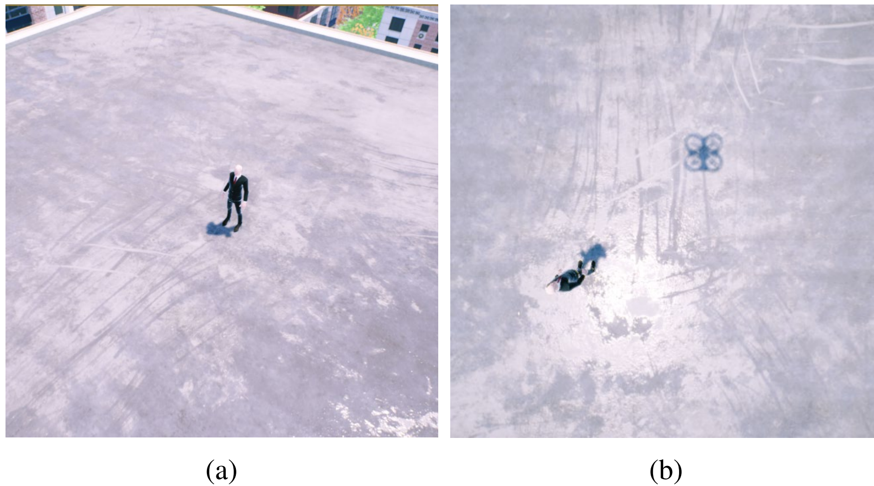
The same simulation and algorithm parameters described in Section 6.8.1 were used for this evaluation. A multi-factor experiment was conducted over hover height set (5, 10, 15, 20, 25, 30) meters and LiDAR sensor set (16, 32, 64) beams, respectively. The drone hovered at each height level and covered a constant-height 2×2 meter grid directly above the person. The drone observed the rooftop from 25 positions within the grid, positioning itself at 0.5 meter intervals. At each observation point the roof and obstacle were extracted using Semantic Polylidar3D. The highest height level where $\frac{24}{25}$ observation points correctly captured the human as an obstacle are shown in Table 6.5. The greater the number of beams, the higher the point cloud coverage of the human rooftop "obstacle". Note that at ≈ 25 meters the person appears as only a few pixels in a 500×500 camera image such that the neural network is unable to distinguish the person from the roof. This means a camera can not solely detect the person at or above 25 meters.

Table 6.5: Maximum height for Semantic Polylidar3D to identify a human on the roof

	16 Beams	32 Beams	64 Beams
Decision height	10 m	20 m	30 m

6.9 Discussion

The presented results demonstrate that Semantic Polylidar3D offers improvement in accurately identifying touchdown points over a purely geometric method. Overall Semantic Polylidar3D had a



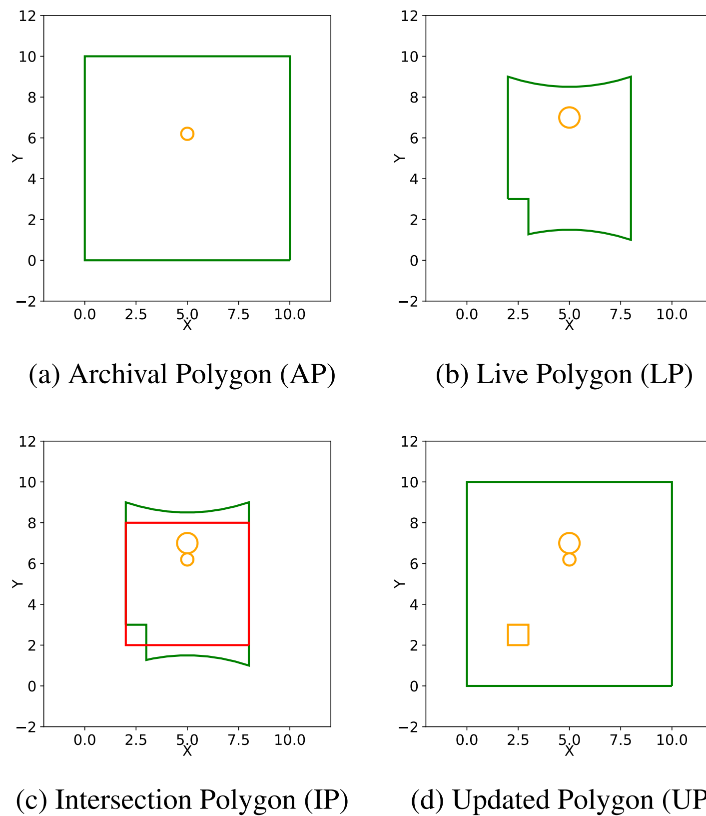
(a) (b)

Figure 6.15: Decision height analysis. (a) Rooftop environment for analysis. (b) Example observation point of a drone five meters above a person.

4% improvement in IoU accuracy compared to the geometric baseline. Additionally, Figure 6.14 shows that adding semantic information has reduced the long tail distribution of low accuracy/worst case examples. The results show our proposed method is robust to individual sensor failures as seen in Figure 6.13. For example, a segmentation failure to identify an obstacle does not lead to total failure unless it is also missed by the LiDAR sensor. Additionally, successful identification of a rooftop using the neural network attenuates range noise by reducing geometric constraints to reduce false negatives.

The total execution of the touchdown point selection pipeline takes $\approx$ 30 ms, strictly dominated by the neural network. It is likely this execution time can be significantly reduced with model quantization, simplification, and limiting/retraining the network to binary segmentation to identify rooftop / non-rooftop surfaces only.

The archived map for safe rooftop landing sites must be updated when the sensor-based planner fails to find a safe touchdown point during the 10m hovering rooftop scans described in this work. The "live polygons" (LP) extracted from the onboard sensor can be used to resolve the inaccuracies in the "archival polygon" (AP) of the rooftop. First, the LP of the landing surface (if it exists) must be transformed from the sensor frame to GPS coordinates to align with the map data. This requires the aircraft to have onboard GPS with sufficient localization accuracy. Figure 6.16 shows an example of an archival and live polygon in (a) and (b), respectively. The archival and live polygon are intersected to create an intersection polygon (IP) shown in (c). This gives the most conservative estimate for observable landing area that can be updated. The camera FOV is overlaid in red. Finally an updated polygon (UP) is created through the union of the IP with the AP subtracted from the FOV. Camera FOV limits the scope of area available for updating.



(a) Archival Polygon (AP) (b) Live Polygon (LP)

(c) Intersection Polygon (IP) (d) Updated Polygon (UP)

Figure 6.16: Proposed rooftop archival polygon update procedure. The AP and LP are intersected to create (c). The final polygon to update the rooftop is shown in (d).

6.10 Conclusion and Future Work

This chapter presented a real time touchdown point selection algorithm that extracts polygons representing flat surfaces by fusing camera images and LiDAR point cloud data captured by a hovering UAS above the potential unprepared landing site. Our method, Semantic Polylidar3D, combines computer vision using neural networks with computational geometry to create a hybrid algorithm robust to individual sensor/method failure. Evaluation was performed in a high fidelity simulated Unreal Engine city constructed from real world rooftop asset statistics collected from midtown Manhattan, New York. Semantic Polylidar3D showed a greater than 4% improvement in IoU accuracy for landing site identification compared to a baseline method and took $\approx$ 30 ms in computation time. The full algorithm, data on Manhattan rooftops, and Unreal Engine world generation scripts are open source and available at [196].

Although our simulated cities model rooftop obstacles accurately, future work is required to build more complete real-world city models that account for building height, textures, streets, parks, etc. Additionally, our current simulation environment only generates sunny weather image data thus

requires extension to more general lighting and weather conditions. As weather conditions worsen, sensors will have degraded performance which will impact the methods presented. Future work should investigate these issues. Real-world experiments of our touchdown point selection algorithm must also be performed in the future.

CHAPTER 7

Flight Experiment Results for Touchdown Point Selection

7.1 Introduction

Small UAS carrying LiDAR, RGBD cameras, or monocular cameras using Structure from Motion (SfM) can generate 3D point clouds of nearby landing sites. Polylidar3D can transform these dense point clouds to polygonal representations of flat surfaces in real-time while accounting for obstacles. Chapter 6 presented simulation results using Polylidar3D for real-time touchdown point selection for rooftops using on-board LiDAR and cameras. This chapter extends simulations to real-world experiments conducted at the University of Michigan Ford Motor Company Robotics Building Fly Lab. This work demonstrates the integration of multiple LiDAR scans into a larger cohesive mesh in which noise is reduced. The final mesh is then sent to Polylidar3D for polygon extraction and touchdown point selection.

We assembled a sensor package that hosts an Intel RealSense L515 LiDAR, RealSense T265 Tracking Camera, and an Odyssey x86 Single Board Computer (SBC). Together they provide color/depth images, 6DOF tracking, and the computation power needed to implement our touchdown point selection methods presented in Chapter 6. Safe touchdown points are found in a cluttered indoor environment. Two separate experiments are performed: one with a hand-carried sensor package and another with the package mounted underneath a flying quadrotor. Results indicate that our presented methods are able to identity safe touchdown points accurately and efficiently.

7.2 Touchdown Point Selection

Chapter 6 proposed our method of selecting a touchdown point from a single scan of a rotating LiDAR sensor mounted underneath a sUAS. The single range image is quickly transformed into a mesh of the environment. This chapter proposes to alternatively integrate multiple scans to create a unified mesh of the environment. This integration process requires the sUAS to have precise localization capabilities. This capability is provided by the Intel RealSense T265 tracking sensor and mounted on-board the quadrotor and validated by a motion capture system.

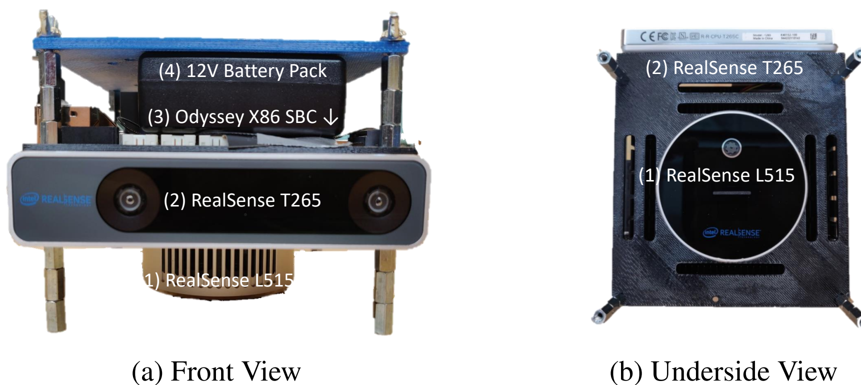
The L515 LiDAR is able to produce an RGBD image by aligning the depth and color data streams. Multiple RGBD frames are integrated into a cohesive map using methods from Zhou et al. [82] and implemented in Open3D [83]. The technique works by creating a truncated signed distance field within a voxel volume. The volume is updated by deprojecting points from an RGBD image into the volume which requires both intrinsic and extrinsic parameters of the camera. The signed distance field is then extracted as a triangular mesh using the marching cubes algorithm [206]. In this work we use a voxel size of 5cm which provides more than enough resolution for landing site decisions. At 3m of distance the range noise of the L515 LiDAR is less than 1cm [185]; noise is nearly removed after integrating multiple frames.

After the mesh is extracted, Polylidar3D is used to extract flat surfaces as polygons. Any obstacles embedded on the surface are captured as interior holes. The largest inscribed circle of the largest polygon is used to select a landing zone. A circle must meet the minimum radial footprint of 0.75 meters, determined by quadrotor size, or no touchdown point is selected. We develop an open-source hardware and software platform that implements this functionality and validate it experimentally.

7.3 Experimental Setup

7.3.1 Sensor Package Construction

The sensor package is shown in Figure 7.1. The package is composed of a (1) RealSense L515 LiDAR/Color Sensor, (2) RealSense T265 Tracking Camera, (3) Odyssey x86 SBC, and a (4) 12V Li-ion battery pack. The package is held together with 3D printed plates and standoffs having a total mass of 720 grams. The dimensions of the package are $120 \times 110 \times 50mm$. The L515 device is mounted directly underneath the sensor package pointing down, while the T265 is mounted in the front. The Odyssey SBC and battery are placed between the printed plates.



(a) Front View (b) Underside View

Figure 7.1: Sensor package components.

7.3.2 Sensor Coordinate Frames

The coordinate frames of the L515 LiDAR and T265 tracking sensors are shown in Figure 7.2 and denoted as $\{L\}$ and $\{T\}$, respectively. The L515 follows nominal camera axis conventions (z-axis forward, x-axis right) while the tracking sensor follows Virtual Reality (VR) conventions (z-axis backwards, x-axis right). The T265 outputs a 6DOF pose (position and orientation) with respect to a world frame origin denoted $\{WT\}$ set during initialization. As a result $\{WT\}$ follows VR axes conventions which is non-standard in Aerospace applications. Another world frame $\{WNED\}$ is created to be coincident to $\{WT\}$ but rotated such that it follows NED conventions (z-axis down, x-axis forward). The reference frame of data will be indicated by a superscript, e.g., $\mathcal{P}^L$ denotes a point cloud in the LiDAR frame. A homogeneous transformation from frame A to frame B is denoted $\mathbf{H}_B^A$.

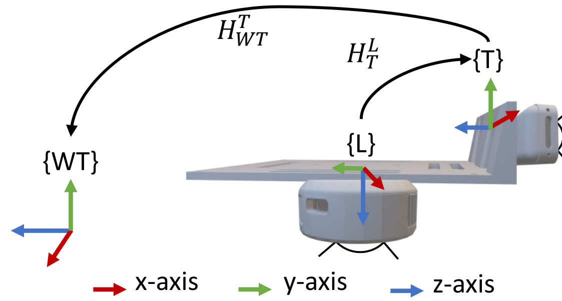


Figure 7.2: Sensor package coordinate frames.

Volume integration as described in Section 7.2 requires LiDAR camera intrinsic and extrinsic calibration parameters. The intrinsics of the L515 are factory calibrated and provided by the RealSense SDK [111]. The T265 pose provides the extrinsics $\mathbf{H}_{WT}^T$ of the T265 sensor with respect to $\{WT\}$. This pose must be appropriately transformed to create the extrinsics of the L515 camera in the world NED frame:

$$\mathbf{H}_{WNED}^L = \mathbf{R}_{WNED}^{WT} \cdot \mathbf{H}_{WT}^T \cdot \mathbf{H}_T^L$$

where $\mathbf{R}_{WNED}^{WT}$ denotes the rotation from $\{WT\}$ to $\{WNED\}$. The matrix $\mathbf{H}_{WNED}^L$ may then transform points in $\{L\}$ to $\{WNED\}$ to create a mesh in this reference frame.

7.3.3 Quadrotor Frame and Sensor Package Integration

We utilize the M330 quadrotor designed at the University of Michigan Autonomous Aerospace Systems (A2Sys) Lab for all flight experiments [207]. The frame measures 33cm diagonally between each pair of motors and is powered by a 4S 3000mAh LiPo battery. Markers are placed

on the quadrotor frame and tracked by a Motion Capture System (MCS). The MCS sends pose estimates of the quadrotor to the flight controller using a wireless serial radio. The quadrotor is controlled by a custom autopilot running on a BeagleBone Blue that uses these ground truth pose estimates for position and yaw control. The sensor package is mounted directly underneath the quadrotor as shown in Figure 7.3. The vehicle body frame $\{B\}$ is defined by the MCS. The total mass of the quadrotor and sensor package is 1744 grams.

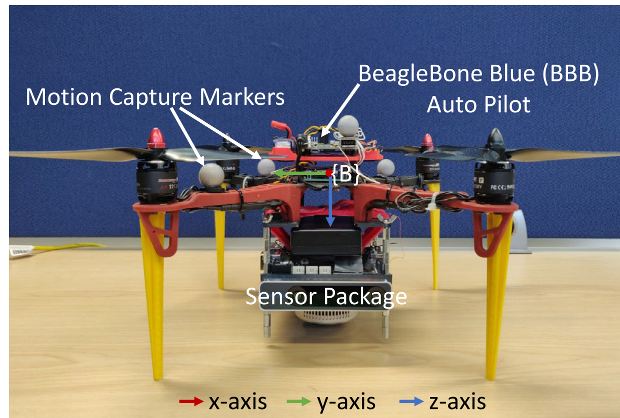


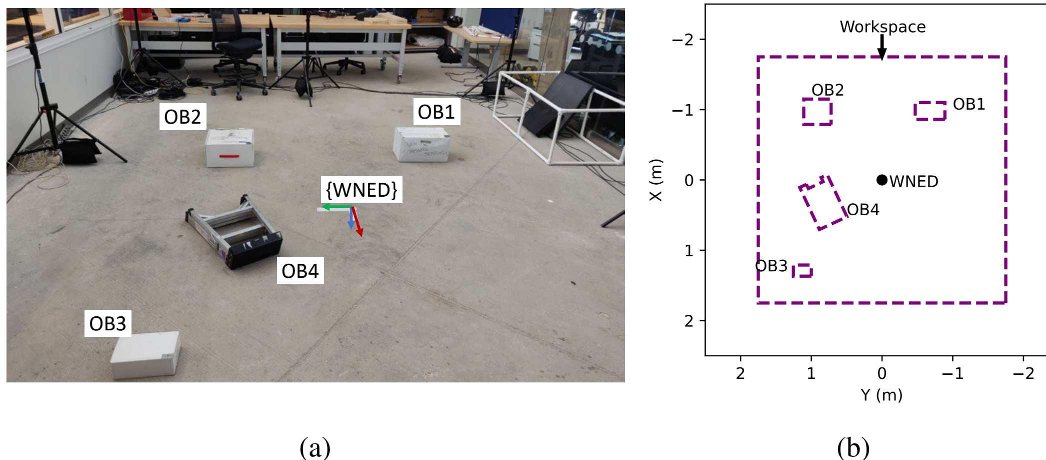
Figure 7.3: Sensor package quadrotor integration. The sensor package is mounted directly underneath the quadrotor. The Beagle Bone Blue autopilot, motion capture markers, and quadrotor body frame $\{B\}$ are indicated.

We continue to use the MCS for quadrotor position control because the T265 sensors accuracy, precision, and reliability have not been fully tested for flight experiments. The T265 is only used to generate a mesh of the environment to give a final touchdown point command to the quadrotor. Note that the T265 sensor initializes the $\{$WNED$\}$ frame during startup. Therefore the quadrotor is positioned such that the MCS coordinate frame is closely aligned with $\{$WNED$\}$ before every flight. However, there is a marginal height offset ($<$ 10cm) between these frames because the T265 is mounted higher than the ground plane. All commanded touchdown points are given in the $\{$WNED$\}$ frame.

7.3.4 Environment

All experiments were performed inside the University of Michigan's Ford Robotics Building Fly Lab. Four obstacles were placed on the floor including three boxes and one small ladder. The environment, obstacle labelling, and origin frame are shown in Figure 7.4a. Obstacle dimensions were measured using a ruler and placed at known positions within the environment. An overlay of the environment and obstacles is shown in Figure 7.4b. The workspace was limited to a $3.5m \times 3.5m$ box centered at the MCS origin and represents the safely navigable region for the drone also within MCS view. Together the workspace and obstacles form a ground truth polygon to assess the

accuracy of any proposed landing area created by Polylidar3D. The obstacles were chosen to have a mix of convex and non-convex shapes to challenge our polygon extraction methods. The size and placement of the obstacles were selected to prevent overloading the workspace and to provide an open area for landing. In future experiments we will diversify obstacles and their placement to further challenge our proposed methods.



(a) (b)

Figure 7.4: Flight lab setup. (a) Photo of flight lab and obstacle placement. (b) Ground truth graph of the environment.

7.3.5 Hardware and Software Integration

The data streams and frequencies for each sensor are shown in Table 7.1. Because of the real-time computational demand of integrating RGBD frames into a volume the frequency of the L515 was reduced to 6Hz. However, the authors noticed no degradation in mesh quality in comparison to running at full speed (30Hz) on a desktop computer. The resolution of the depth stream and RGB stream were set at the recommended levels from Intel to further reduce computational demand.

Table 7.1: Sensor package details

Sensor	Stream	Frequency	Description
L515 LiDAR Camera	Depth	6 Hz	640X480, Depth Stream
	Color	6 Hz	1280X720, RGB Stream
T265 Tracking Camera	6DOF	100 Hz	Position and Orientation

Figure 7.5 shows a diagram of the devices and software used in the experiments. The top left of the diagram displays the hardware interfaces, the top right is a legend, and the bottom is the software architecture. The Odyssey SBC communicates to the quadrotor Beaglebone Blue flight controller

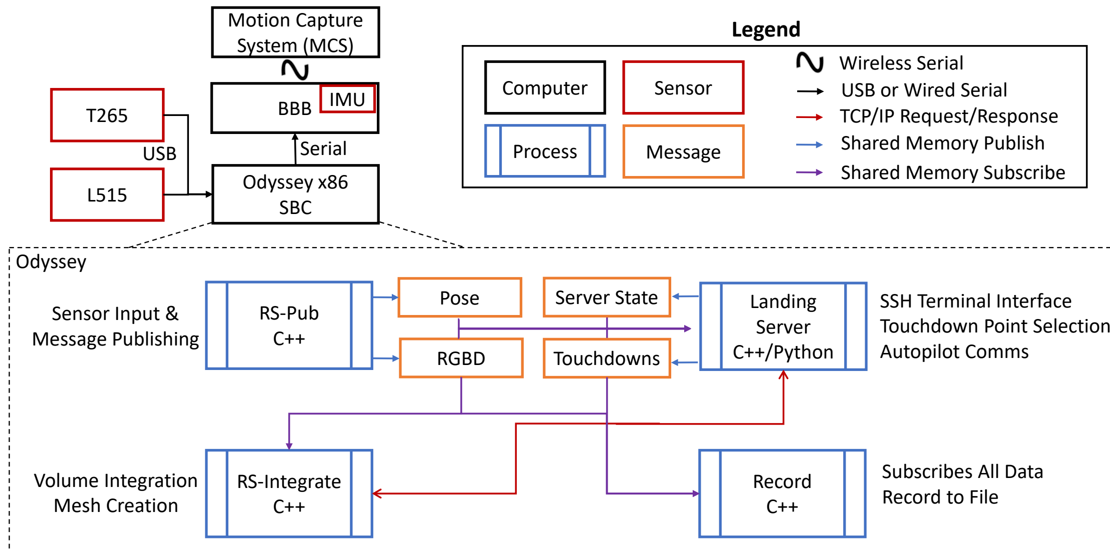


Figure 7.5: Overview of hardware interfaces and software architecture.

through a serial connection. The communication is one-way and allows the emergency landing software to command a landing position. Currently the command is issued only once, cannot be cancelled, and the quadrotor will immediately fly a constant altitude path to the position and then land. More sophisticated emergency landing logic is beyond the scope of this work.

The urgent landing software is composed of four main programs: `RS-Pub`, `RS-Integrate`, `Landing Sever`, and `Record`. Programs communicate with each other with messages using Enhanced Communication and Abstraction Library (ECAL), a fast publish-subscribe middleware that manages inter-process communication [208]. The program `RS-Pub` is responsible for configuring and gathering data from the RealSense devices and publishes shared memory messages containing RGBD frames with pose information. The program `RS-Integrate` subscribes to these messages and will integrate them into a cohesive voxel volume using Open3D [83]. It also runs a TCP/IP server that upon request will extract a mesh from the volume. The program `Landing-Sever` contains the algorithms and software previously presented in this dissertation. This contains our own software including mesh smoothing, polygon extraction, polygon filtering, and touchdown point selection. This program also runs an interactive terminal user interface, shown in Figure 7.6, which allows a user to activate the emergency landing protocols. Finally, the program `Record` efficiently records all messages with synchronized timestamps. All code for this work is open-source and freely available [209].

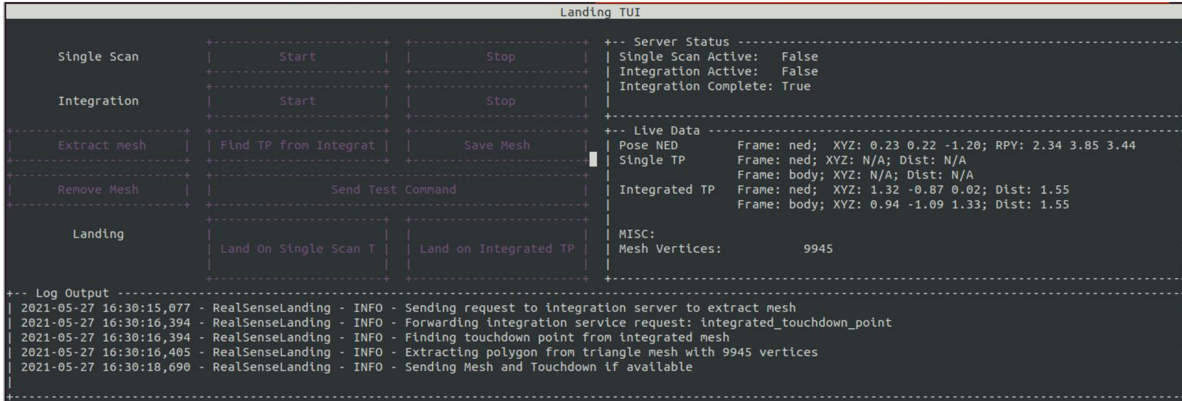


Figure 7.6: Picture of terminal user interface for urgent landing.

7.4 Experimental Results for Touchdown Point Selection

Sections 7.4.1 and 7.4.2 present results for hand-carry and flight tests for our touchdown point selection algorithm, respectively. Section 7.4.3 provides execution timings statistics for our algorithms. Section 7.4.4 presents a trajectory evaluation error analysis for the Intel T265 sensor.

7.4.1 Hand Carry Test

Five hand-carry tests were performed in the Flight Lab. An identical software suite as presented in Section 7.3.5 was running except landing commands to the quadrotor were disabled. To simulate flight, we attached an extendable pole to the sensor package. The package was then picked up and moved around within the environment. The landing software automatically created meshes of the environment and used Polylidar3D to extract flat surfaces for landing areas. A risk-optimal touchdown point was then selected by finding the greatest inscribed circle within the polygon. Figures 7.7a,b show qualitative results for the mesh, polygon, and touchdown point for two of the hand-carry tests. The green and orange lines represent the exterior shell of the polygon and interior holes (obstacles), respectively. The center of the blue circle denotes the risk-optimal touchdown point.

Figures 7.7c,d display the same polygons shown in (a,b) but projected to the XY plane alongside the ground truth workspace previously shown in Figure7.4b. The landable area and obstacles are accurately captured in both examples. Our methods intentionally slightly exaggerate the size of obstacles to provide a safety buffer for landing. We provide quantitative accuracy results by calculating the Intersection over Union (IOU) of each extracted polygon and the ground truth workspace polygon. The mean IOU for all five tests was 94.2% which indicates that the surface extraction was highly accurate. There seems to be a small positional bias ($< .05m$) in obstacle placement which may be from inaccurate localization in the T265 tracking sensor. The is investigated

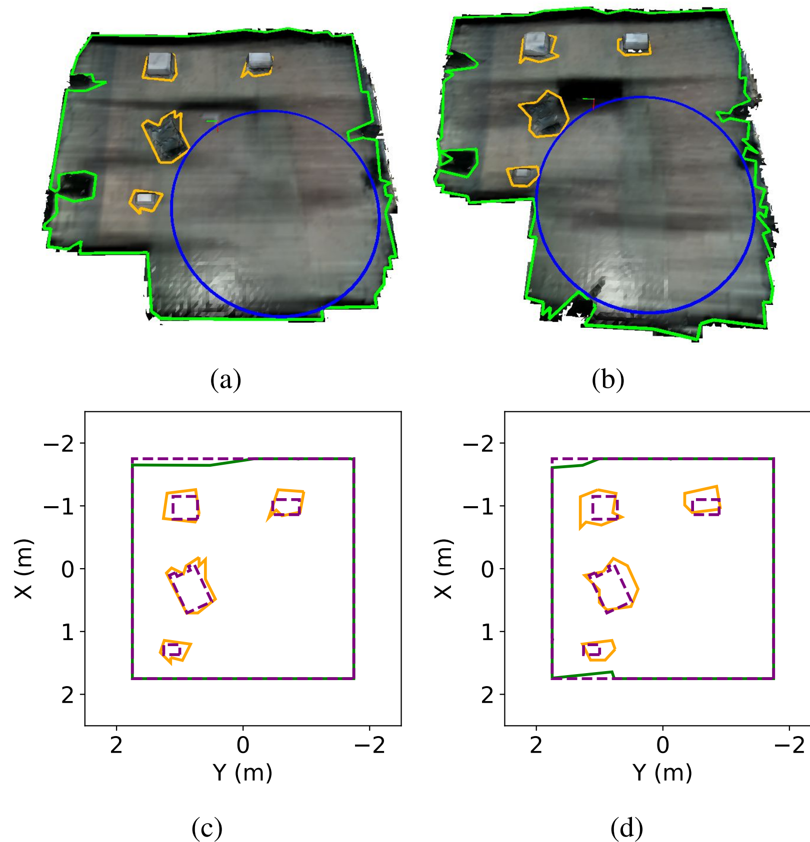


Figure 7.7: Real-time constructed meshes and polygons during a hand carry test. (a,b) Meshes, polygons, and touchdown points from two hand carry tests. Polygons are in green/orange and the touchdown circle in blue. (c,d) Comparisons of the ground truth polygon (dashed purple) versus extracted polygons.

more thoroughly in Section 7.4.4.

7.4.2 Flight Tests

Three flight tests were conducted with the sensor package payload. In all experiments a University of Michigan graduate student with remote piloting experience acted as Pilot in Command (PIC) and manually controlled the quadrotor to execute the flight path shown in Figure 7.8. This box pattern allowed full coverage of the workspace by the L515 LiDAR during the integration process. A flow diagram of the experiment protocol is shown in Figure 7.9. After scanning the environment, the urgent landing protocol was activated which extracted a mesh of the environment, found the optimal touchdown point, and began autonomous trajectory generation, navigation and landing procedures. In all flight tests the quadrotor successfully found the touchdown point and landed. Each flight experiment took approximately 75 seconds.

Figures 7.10a,b show the meshes generated in real-time from the flight experiments. We can see

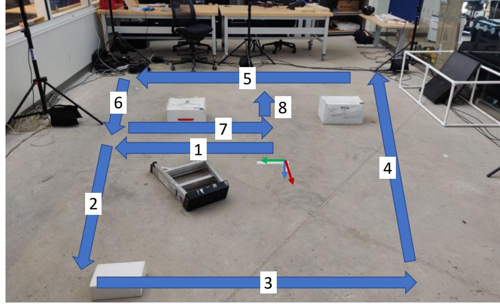


Figure 7.8: Visualization of flight path used in all experiments.

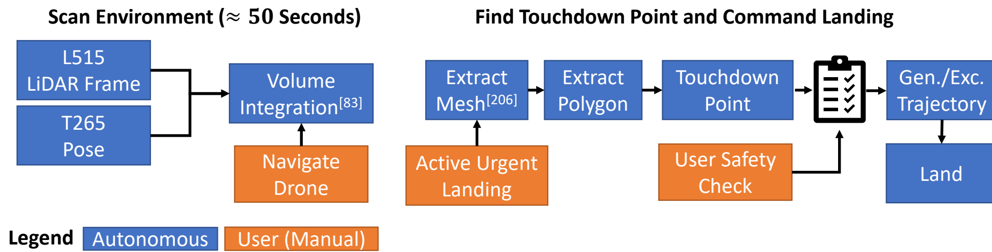


Figure 7.9: Flow diagram of flight experiment protocol.

that the mesh, polygons, and touchdown point are accurate of the environment. Figures 7.7c,d show the extracted polygons projected to the XY plane and compared with the ground truth polygon of the workspace. The mean IOU for all three flight tests was 92.7%, slightly lower than the 94.3% accuracy of the hand carry tests. This may indicate that the T265 struggled with more precise localization in flight versus being hand carried.

7.4.3 Execution Time

The computation time for the major steps in our methods are calculated and presented in Table 7.2. The table shows the mean execution time and standard deviation for all experiments conducted on both the low-power Odyssey x86 SBC as well as a desktop computer. Both computers ran the same software over recorded data for all experiments. The Odyssey board has an Intel J4105 processor with 4 Cores and 8 GB of RAM while the desktop is configured with a 12 core AMD 3900X processor with 32 GB of RAM. In both systems a maximum of 2 threads were used for parallelization in Polylidar3D for polygon extraction. All other steps are single threaded.

For each test approximately 111 RGBD frames were integrated into a voxel volume. Once the volume is created, a triangular mesh of the environment is extracted. Polylidar3D then extracts a polygon of the mesh, filters/simplifies it, and an optimal touchdown point is found. The Odyssey SBC executes volume integration sufficiently fast to match the real-time 6 Hz stream of the LiDAR

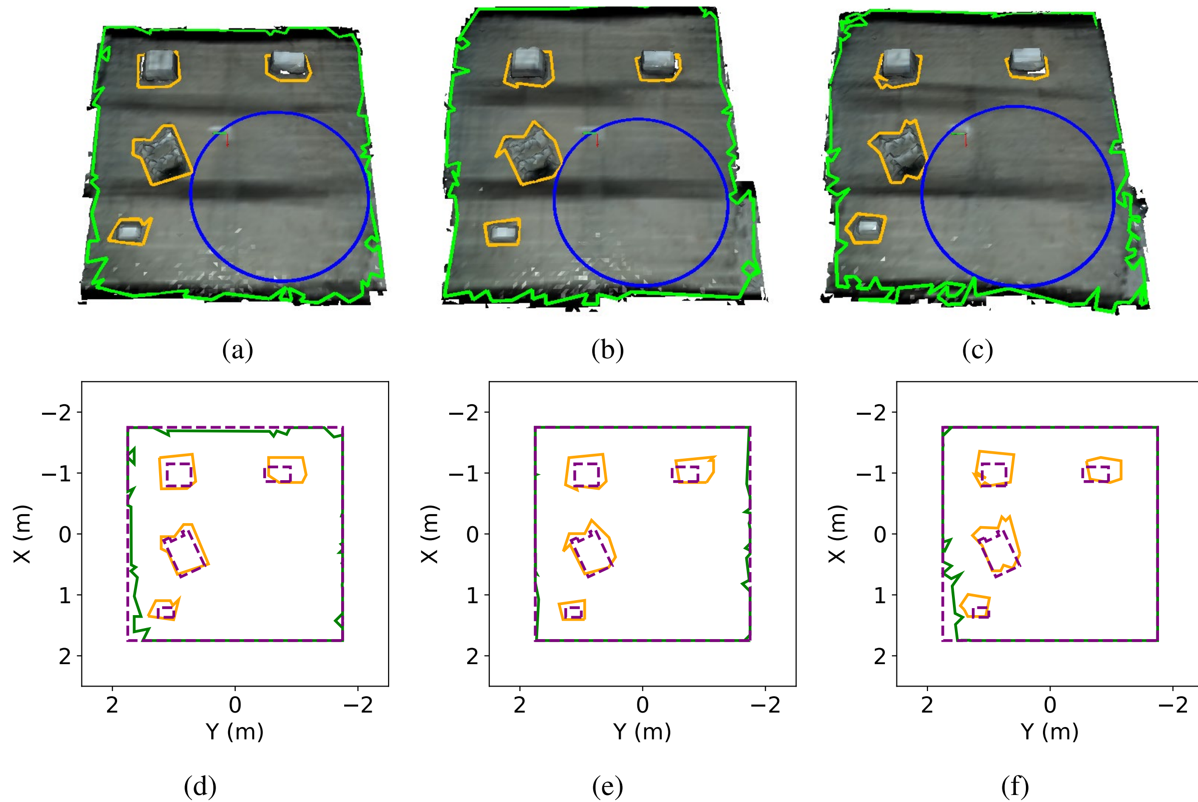


Figure 7.10: Real-time constructed meshes and polygons during flight test. (a,b,c) Meshes, polygons, and touchdown points from three flight tests, respectively. The polygon is shown in green/orange and touchdown circle in blue. (c,d,e) Comparisons of the ground truth polygon (purple) versus extracted polygons.

sensor. Additionally, our landing site selection software is able to find a safe landing site in less than 60 ms. In most cases the desktop computer is ≈ 3 times faster than the Odyssey SBC. However, frame integration has a 10X performance degradation when using the Odyssey board in comparison to the desktop computer. This can be seen not only from the average execution times but also the large standard deviation of 27ms.

Table 7.2: Mean and standard deviation of execution times (ms)

Computer	**Volume Integration**		**Touchdown Point Selection**		
	Integrate Frame	**Extract Mesh**	**Extract Polygon**	**Filter Polygon**	**Touchdown Point**
Odyssey SBC	19.6 ± 27.2	47.8 ± 8.4	18.3 ± 2.9	39.1 ± 9.2	0.2
Desktop	2.5 ± 0.5	16.9 ± 2.6	6.7 ± 0.9	14.0 ± 5.1	0.1

7.4.4 Trajectory Error Analysis of Intel RealSense T265

This section provides a trajectory error analysis of the T265 pose predictions in comparison to the ground truth MCS. The T265 predicted pose estimates are given an initial alignment to the ground truth trajectory following standard evaluation procedures in [210]. Figure 7.11 shows the 3D trajectory of the quadrotor in the first flight experiment using pose estimates from the MCS (orange) and the T265 (blue). The predicted T265 trajectory closely follows MCS but appears to drift away marginally after time. Figure 7.12 provides time response plots for the x, y, z, roll, pitch, and yaw estimates for all three flight experiments.

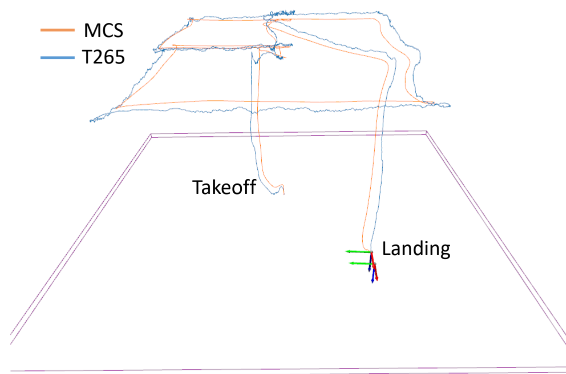
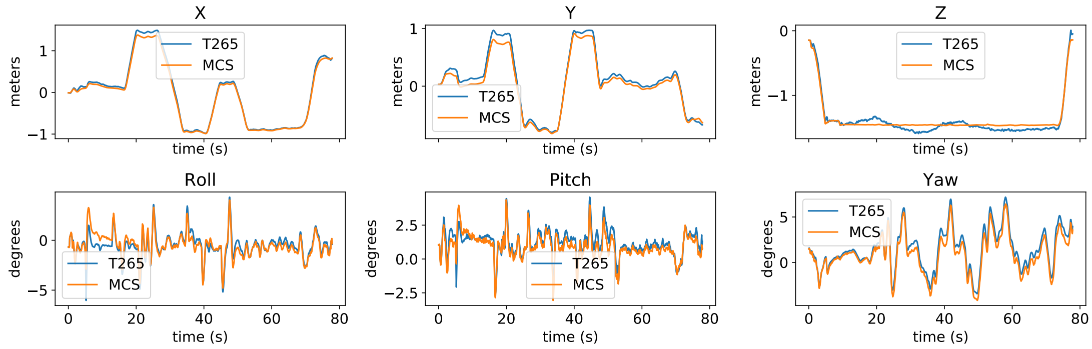


Figure 7.11: Trajectory of quadrotor from Flight #1. Trajectory of Motion Capture System (orange) and T265 (blue) shown.

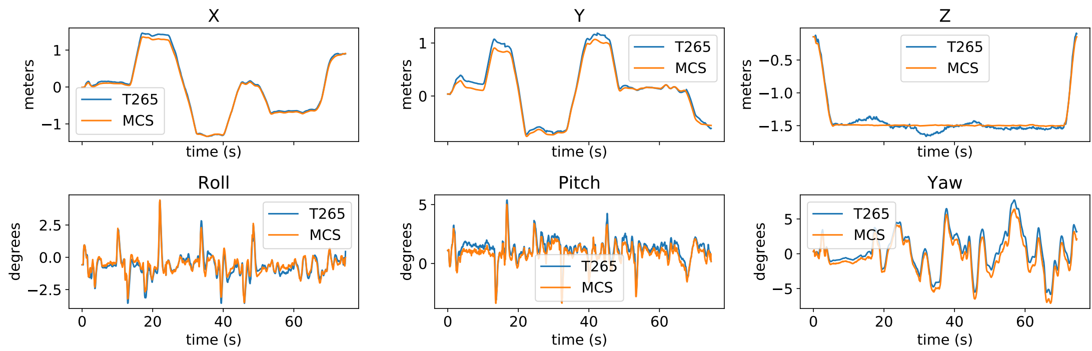
The mean absolute trajectory error is calculated for each flight and displayed in Table 7.3. The position and rotation error are computed following procedures in [210]. The average length of each flight experiment path is approximately 17.2m. The position and rotation error is low for the first and second flight but markedly higher in the third flight. During take-off of the third flight the T265 had strong deviations in altitude, roll, and pitch from the MCS as seen in Figure 7.12c. The estimated roll and pitch continued to track correctly but with a large bias of approximately 4 degrees. The positional altitude quickly recovered after 5 seconds when the T265 experienced a "pose jump" after a loop closure occurred. Currently "pose jumping" for the T265 is only implemented for translation and does not correct orientation error [211].

Table 7.3: Mean Absolute Trajectory Error (ATE) for T265

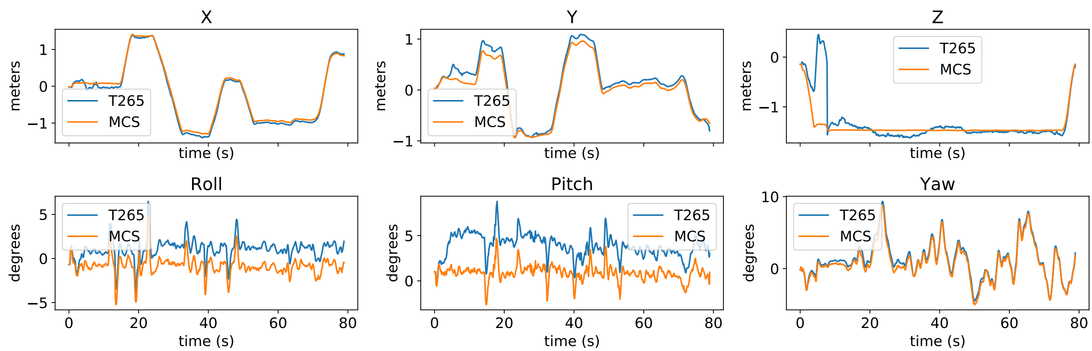
Trial	Length (m)	Mean ATE_{pos} (m)	Mean ATE_{rot} (deg)
Flight #1	17.1	0.11	0.98
Flight #2	16.6	0.10	1.03
Flight #3	17.9	0.35	3.7



(a) Flight #1



(b) Flight #2



(c) Flight #3

Figure 7.12: Comparison of Motion Capture System (MCS) versus T265.

7.5 Conclusion and Future Work

Multiple experiments were conducted to validate our proposed methods for real-time touchdown point selection. We developed a sensor package that holds an Intel RealSense L515 LiDAR, RealSense T265 Tracking Camera, and an Odyssey x86 Single Board Computer (SBC). Together they provide depth data, localization and mapping, and the computational power necessary for our landing software. The sensor package was both hand-carried and flown with a quadrotor in

an obstacle cluttered indoor environment. Accurate meshes of the environment were generated in real-time for which landing sites were extracted as polygons. The polygons representing safe landing areas were compared against the ground truth map and found to be accurate. In every experiment a safe landing zone was found which correctly minimized risk.

Future work should be performed to expand upon the diversity and placement of obstacles in the workspace. For example, adding larger more non-convex obstacles with holes in the center will further challenge our methods to more fully verify their efficacy. In addition, highly cluttered environments should be tested to verify our touchdown point selection algorithm will only select safe landing points. The size of the environment, the number of obstacles, and obstacle shape complexity will have an impact on the computation time needed for our landing software. Further work should be performed to quantify this relationship and determine its impact. Experimental work should also be conducted to integrate a semantic neural network with Polylidar3D as proposed in Chapter 6. Data of real-world and synthetic environments should be captured and labelled to train the proposed network. The Odyssey board should be swapped with a SBC with an on-board GPU to provide minimal inference time for semantic segmentation. Further flight experiments above real-world city rooftops can then be conducted to further validate our touchdown point selection algorithm.

CHAPTER 8

Conclusions

This dissertation has developed and integrated methods to enable safe and robust small UAS urgent landing capabilities in complex urban environments with focus on perception, mapping, and flight planning necessary to support unprepared rooftop landings. To identify and quantify the risk of landing sites we present methods integrating a multitude of data sources through novel computation geometry algorithms and deep learning.

Prior to this work, an sUAS needed to be near prepared sites or open terrain suitable for urgent landing. This dissertation has shown that within cities there may be hundreds of nearby flat rooftops available for landing. This drastic increase in landing site options requires on-board autonomy to be as *prepared*, *efficient*, and *lazy* as possible. First, preparedness comes from preprocessing as much data as possible to accurately assess the risk level of landing sites and their surrounding environments. Never leave work to be done online during an emergency that can instead be done offline. Second, it is essential to be efficient by utilizing appropriate data structures and algorithms while also exploiting parallelism. For example, using spatial indexes vastly reduces spatial query execution time, heuristic choices should be guided by the environment (3D octile distance for 3D grids), and work should be partitioned into as many isolated tasks as possible while distributed in parallel asynchronously. Finally, software should attempt to be as lazy as possible by only executing functions that are actually needed. This is especially true for expensive computations, such as path planning, that need only be executed when required. No amount of algorithmic optimization will ever outperform a no-op (no operation). These three principles have guided our research approach and contributed to numerous open source algorithms and datasets for the academic community.

8.1 Contributions

The contributions of this dissertation include:

- In Chapter 2, we proposed our algorithm Polylidar to extract non-convex polygons with interior holes from 2D point sets. The point set is firstly triangulated where the shape is extracted using efficient region growing of triangle simplices. Unlike other methods which

extract polygons by taking the union of triangle sets, Polylidar carefully walks the boundary of the set while accounting for interior holes. We benchmark our proposed algorithm on several state-of-the art algorithms showing more than 4X speed improvement and comparable accuracy.

- In Chapter 3, we extended Polylidar to extract polygons representing flat surfaces from a variety of 3D data sources such as unorganized 3D point clouds, organized 3D point clouds, and user-defined meshes. As part of this work we present a novel fast Gaussian Accumulator that can quickly identify dominant plane normals within a 3D scene. Flat surfaces of non-connected surfaces are extracted independently though our parallel region growing and polygon extraction routines. We evaluate our methods on five separate datasets showing the speed and versatility of our methods.

- In Chapter 4, we proposed a method for identifying the roof shape of buildings using deep learning from airborne LiDAR point clouds, satellite images, and building outline data. A prepossessing routine takes raw data and generates both an RGB and depth image for each rooftop. Over 4500 building roofs spanning three cities were manually classified and archived. This is the largest dataset for roof shape identification at the time of publication. A combination of a CNN for feature extraction and a random forest for classification gave the best results with an accuracy of 86% on test data sets. We show that confidence thresholding can lead to greater than 95% precision and 75% recall in labeling flat-like roofs.

- In Chapter 5, we proposed a framework for assimilating GIS data to identity and evaluate risk for emergency landing sites, uniquely including building rooftops. Our work not only identifies flat rooftops, but isolates obstacle-free flat surfaces on them and quantifies the *usable* landing space thereon for risk evaluation. We presented a multi-goal planner that efficiently selected the landing site/path pair guaranteed to minimize a weighted total risk function. Several case studies and Monte Carlo simulations are conducted showing that our planner finds risk-optimal landing sites in less than 50ms for 95% of all cases.

- In Chapter 6, we proposed a hybrid computational geometry and deep neural network algorithm to identify and select safe rooftop landing zones in real-time using a combination of LiDAR and camera sensors. For testing, we created a high-fidelity simulated city in the Unreal game engine with particular attention given to creating a statistically accurate representation of rooftop obstacles. Results showed that our fusion of geometric and semantic information improved landing site identification accuracy over 4%.

- In Chapter 7, we successfully evaluated our proposed methods for touchdown point selection with a drone platform. We used on-board solid state LiDAR and 6DOF tracking sensors to

create full environmental meshes of an indoor flight environment. Obstacle-free flat surfaces were extracted with Polylidar3D and optimal touchdown points selected for autonomous landing. In all three flights a landing site was found and the drone landed successfully.

8.2 Future Work

Although this dissertation has presented multiple contributions in computational geometry, machine learning, and urgent landing for sUAS, there are still many challenges to improve reliability of autonomous urgent landing in cities. Specific challenges are discussed below.

8.2.1 Robustly Segmenting Rooftop Point Clouds

Chapter 4 proposed a general framework for identifying rooftop shapes through RGB and depth images using CNNs. There have recently been many advances in deep learning which can operate more directly on 3D data [152, 153, 212]. These neural network architectures sample from the point cloud and directly learn global and local geometric features of the point cloud surface. These methods have been shown to be successful in shape classification, object detection and tracking, and point cloud segmentation [213]. Our methods on rooftop landing site detection could be improved if aerial LiDAR point clouds could be more accurately segmented and classified before being given to Polylidar3D. Polylidar3D could then be modified to take advantage of these segmentation classes to provide a more robust estimate of landing areas.

8.2.2 Improving Polylidar3D

Polylidar3D is currently designed for extracting dominant planes within scenes such as floors and walls. This focus allows Polylidar3D to be extremely fast at grouping triangles that may belong to the same continuous surface and performing region growing of disparate regions in parallel. However, this limits Polylidar3D's use in applications that require detailed extraction of smaller surfaces within a 3D scene. Future work should investigate integrating new techniques that use Spherical Convex Hulls to iteratively refine surface normal estimates during region growing [214]. There is potential to combine our proposed Gaussian Accumulator for an initial estimate of planes and the Spherical Convex Hull for refinement and extraction of the remaining small surfaces.

Polylidar3D was designed to be a versatile framework to take many forms of 3D input. This versatility expands its applicability but creates a challenge for creating unified and optimized software. For example, there are many ways to further increase and parallelize Polylidar3D when working with range images. The structure of a range image allows neighbor information to be implicitly computed and does not need require explicit neighborhood data structures such as the

half-edge neighbors of triangles. Optimized routines for each data input will allow further speedup and possibly improved accuracy with new techniques.

8.2.3 Extending Touchdown Point Definition to Fixed-Wing Aircraft

This work has focused on finding terrain and rooftop landing sites suitable for Vertical Take-of and Landing (VTOL) sUAS urgent landing within cities. Touchdown points on landing sites were found by calculating the largest inscribed circle of the polygonal representations of flat surfaces. These touchdown circles are ideal for VTOL aircraft but are not suitable for fixed-wing aircraft which require a level strip of smooth ground, i.e., an unprepared runway, to support safe deceleration to a full stop. The flat terrain-based landing sites identified in this work may be suitable for fixed-wing aircraft if alternative touchdown locations are defined. Future work should investigate finding the longest inscribed rectangle inside a non-convex polygon with sufficient width and length necessary for safe aircraft landing. This problem is related to prior work which finds a rotated rectangle with the largest area inside non-convex polygons [215]. However, the method cannot handle polygons with interior holes and optimizes for the area of the rectangle instead of its length. Future work should develop methods for finding multiple "runways" inside polygons that can optimize for the approach and roll-out of a fixed-wing aircraft needing immediate landing. Wind, nearby terrain, et al. must also be considered in future work.

8.2.4 Remembering the Human Factor

The risk models presented in Chapter 5 assume that the overall risk of a decision is the sum of the magnitude of each attribute multiplied by a risk score. The choice of attributes and their magnitudes are subjective and require thoughtful human determination [169]. Future work should be performed to gather expert pilot opinions on whether the proposed attributes are sufficient and if additional metrics are needed. Participant will also rank, categorize, and group the most important attributes. Scenarios similar to the presented case studies in Chapter 5 can be shown where pilots will choose a landing site/path pair. A fully integrated visualization of 2D maps, 3D environments, and risk graphs can be presented to allow research participants to make informed decisions. The result of this work will inform attribute and weight definitions in our final risk models.

In the event of an emergency, our proposed emergency planner can operate locally and autonomously; a data-link for remote operator action is not required. However, it may be desirable to give a time-limited opportunity for a remote human operator to participate in the emergency response process. Some failure scenarios may pose high-risk toward humans requiring an immediate decision for landing (i.e., sub-second). These situations do not allow elaborate human interaction with an emergency planner interface; we must prioritize the necessary speed of the autonomous

system against the value a human operator may provide. A simple interface displaying the optimal landing site/path pair with a confirmation or "go" button may be used in such situations. These confirmation displays are often used in high risk situations, e.g., autonomous weapon defense systems where a human operator has less than one second to confirm the launch of intercepting missiles against incoming short-range rockets [172]. However, many sUAS failures will not pose immediate high risk to overflown populations such that more in depth human feedback may be beneficial. Humans in this role should not focus on low level details such as trajectory planning but act in a supervisory role by choosing the best course of action that is presented [173].

For this purpose, an intuitive user interface for our emergency planner should be carefully defined. The type, amount, and form of information presented should be balanced with cognitive strain humans encounter during time-sensitive, high risk, and uncertain situations [170]. Research indicates that humans in this "problem-solving" mode look for cues from data, perform hypothesis generation and selection, and finally action selection [171]. Therefore our user interface should be limited to elements that successfully aid humans through this decision making process. First, our emergency planner interface should have separate *pre-takoff configuration* and *emergency action selection* screens. The pre-takeoff screen allows a user to configure the mission specific constraints and attribute rankings for risk minimization during landing site selection. Constraints such as flight altitude, flight time, landing site distance, and landing site type (e.g., prepared or unprepared) should be able to be removed and added through the interface. Additionally, users may select an option to bias landing site rank to those closer to a critical destination point (e.g., a hospital) rather than being near the sUAS itself. A slider can also be presented that changes the ranking of landing sites based on landing site or path risk metrics. This screen may be information dense as time is not a constraint during the pre-flight process. In contrast, during an urgent crisis the action selection screen must afford quick operator selection. Therefore, only the top ranked sites should be presented to the operator to bring attention to the most likely of choices. Concise summary views of each landing site and their paths should be available. The operator may then choose a final landing site/path pair. Future work should gather expert opinions of pilots and user interface design engineers to begin designing both interfaces.

8.2.5 Gaining Confidence in the Data

Our proposed methods have successfully identified thousands of additional landing sites within cities by finding rooftops suitable for landing. However, the maps we create are limited by the availability, accuracy, and resolution of the archived GIS data sources. The characteristics of the underlying data may change over time and by location in the world. Future work should be done to quantify data quality and uncertainty while determining their effects for use in urgent landing. Examples of important attributes include the date and time of recording, satellite image resolution,

point cloud spacing and noise, image occlusion and distortion, and many others. A common example of inaccurate data is when airborne LiDAR point clouds are several years old which may mislead our methods about the current state of the environment. Our methods attenuate these effects by fusing multiple data streams together (e.g., satellite and LiDAR data), but sometimes both modalities can be incorrect. Future work should be able to explicitly mark rooftops identified using low quality sources by attaching a "grade" that indicates its reliability.

Our methods for rooftop identification need not be limited to archived data sources. Future work should investigate methods of integrating real-time data streams from UAS into a remote cloud database. This data could then be processed in the cloud to significantly augment existing data and further reduce uncertainty. However, this does introduce new challenges as multiple datasets, possibly conflicting, must now be integrated. It is likely that this future work will require the use of distributed consensus algorithms to enforce data consistency [216]. Database changes could then be streamed to UAS with a datalink to provide the most up to date information.

8.2.6 Creating a More Complete Picture of Risk

The efforts in this dissertation have found hundreds of safe nearby landing zones by utilizing rooftops. Yet a complete quantification of the risks sUAS pose to themselves, people, and property are difficult to accurately calculate. For example, quantifying the effects of wind on sUAS trajectories is an important task left for future work. However, supporting a quick nearby landing that minimizes flight time, and knowing whether a rooftop is truly unoccupied *at the time of landing* are keys to risk reduction. Yet high resolution temporal population information is missing from public datasets. Large corporations, such as Google and Apple, have access to this data through location tracking on mobile devices. The transformation and packaging of this and other personal data becomes the corporation's property and is either used internally or sold to business partners. This is an example of a centralized tracking program. However, the recent COVID-19 pandemic has shown that a decentralized, user-friendly, and anonymous location tracking program is not only possible but beneficial for public health [217, 218]. These opt-in applications have been used successfully during the COVID-19 pandemic to allow more rapid contact tracing during outbreaks. This same technology can be used to enable real-time anonymized population density information within cities to inform decision making for autonomous safety systems. A drone with real-time access to such a data stream can make a more informed choice for landing site selection and path planning.

A future is soon coming where there will be hundreds of autonomous drones navigating the urban skies. This will present challenges as drones will need to cooperate with each other when executing their individual missions. These challenges are similar to those facing the autonomous vehicles (AV) industry. Vehicle to Vehicle (V2V) communication is proposed as a solution which can increase the safety of passengers by sending and receiving local omni-directional messages informing

vehicles of nearby potential accidents/crashes/threats. In addition, Vehicle to Infrastructure (V2I) communication can allow vehicles to send their position, velocity, and important observations to a centralized server. We critically need trusted datalink for next-generation air traffic management, system-wide. Human voice communication has served us well but simply cannot scale. A traffic management server can then inform others about traffic congestion and provide warnings for hazardous situations [219]. UAS can greatly benefit by utilizing V2V and V2I (V2I2V) techniques and NASA's proposed UAS Traffic Management (UTM) system would be an excellent fit for V2I functionality. A drone or future advanced air mobility taxi may ingest these local and cloud-based messages to create a more complete picture of local risks.

APPENDIX A

Source Code Summary

This dissertation presents numerous algorithms which aid in identifying safe rooftop landing sites in cities. However, many of our methods are general and can be applied to other domains. In particular, our Polylidar3D algorithm has applications to any problem domain that requires polygon extraction from 2D or 3D data. Such problems readily arise in fields such as computational geometry, GIS, and robotics. We have released the open-source software implementations of Polylidar3D and its supporting software under the permissive MIT license [220]. We have carefully optimized each software implementation and tested on both Windows and Linux platforms. All software is hosted online using GitHub, a software development and version control website. Our software is currently being used by academic researchers, software engineers, and geographic information system scientists. The three main software repositories of interest are:

1. Polylidar3D - Fast polygon extraction from 2D and 3D data.
   Hosted at: `https://github.com/JeremyBYU/polylidar`

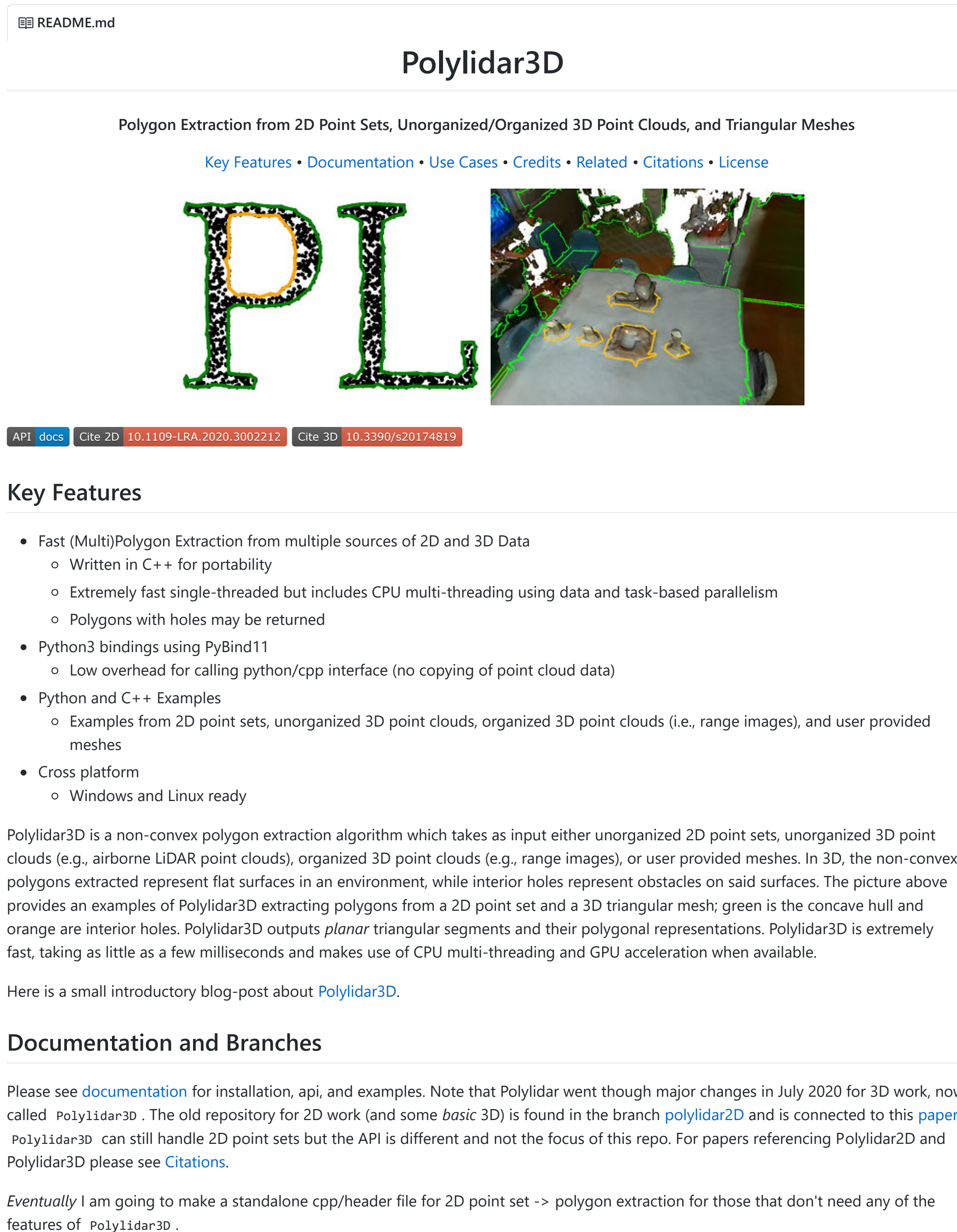
2. Fast Gaussian Accumulator (FastGA) - Dominant plane normal estimation for 3D scenes.
   Hosted at: `https://github.com/JeremyBYU/FastGaussianAccumulator`

3. Organized Point Filters (OPF) - Fast smoothing for organized 3D point clouds.
   Hosted at: `https://github.com/JeremyBYU/OrganizedPointFilters`

Each software repository is written in C++ for high performance but includes bindings to Python. We utilize CPU parallelism if available. The build system uses `CMake` and is tested with the `msvc` and `gcc` compilers. Each repository provides documentation through examples as well as the application programming interface (API). Bug reports may be filed at each individual repository. Each section below will provide multi-page screenshots of their respective source code repositories. The screenshots are of the `README.md` file which provides a summary of the software.

A.1 Polylidar3D Source Code Summary

Below is a multi-page screenshot of the `README.md` file for the Polylidar3D source code repository hosted at `https://github.com/JeremyBYU/polylidar`



README.md

Polylidar3D

Polygon Extraction from 2D Point Sets, Unorganized/Organized 3D Point Clouds, and Triangular Meshes

Key Features • Documentation • Use Cases • Credits • Related • Citations • License

API docs | Cite 2D 10.1109-LRA.2020.3002212 | Cite 3D 10.3390/s20174819

Key Features

- Fast (Multi)Polygon Extraction from multiple sources of 2D and 3D Data
  - Written in C++ for portability
  - Extremely fast single-threaded but includes CPU multi-threading using data and task-based parallelism
  - Polygons with holes may be returned
- Python3 bindings using PyBind11
  - Low overhead for calling python/cpp interface (no copying of point cloud data)
- Python and C++ Examples
  - Examples from 2D point sets, unorganized 3D point clouds, organized 3D point clouds (i.e., range images), and user provided meshes
- Cross platform
  - Windows and Linux ready

Polylidar3D is a non-convex polygon extraction algorithm which takes as input either unorganized 2D point sets, unorganized 3D point clouds (e.g., airborne LiDAR point clouds), organized 3D point clouds (e.g., range images), or user provided meshes. In 3D, the non-convex polygons extracted represent flat surfaces in an environment, while interior holes represent obstacles on said surfaces. The picture above provides an examples of Polylidar3D extracting polygons from a 2D point set and a 3D triangular mesh; green is the concave hull and orange are interior holes. Polylidar3D outputs *planar* triangular segments and their polygonal representations. Polylidar3D is extremely fast, taking as little as a few milliseconds and makes use of CPU multi-threading and GPU acceleration when available.

Here is a small introductory blog-post about Polylidar3D.

Documentation and Branches

Please see documentation for installation, api, and examples. Note that Polylidar went though major changes in July 2020 for 3D work, now called `Polylidar3D` . The old repository for 2D work (and some *basic* 3D) is found in the branch polylidar2D and is connected to this paper. `Polylidar3D` can still handle 2D point sets but the API is different and not the focus of this repo. For papers referencing Polylidar2D and Polylidar3D please see Citations.

Eventually I am going to make a standalone cpp/header file for 2D point set -> polygon extraction for those that don't need any of the features of `Polylidar3D` .

Figure A.1: Page 1 of the Polylidar3D repository

Polylidar Use Cases

- Polylidar-RealSense - Live ground floor detection with Intel RealSense camera using Polylidar
- Polylidar-KITTI - Street surface and obstacle detection from autonomous driving platform
- PolylidarWeb. An very old Typescript (javascript) version with live demos of Polylidar2D
- Concave-Evaluation - Evaluates and benchmarks several competing concavehull algorithms

Credits

This software is only possible because of the great work from the following open source packages:

- Delaunator - Original triangulation library
- DelaunatorCPP - Delaunator ported to C++ (used)
- parallel-hashmap - Fast hashmap library (used)
- marl - A parallel thread/fiber task scheduler (used)
- PyBind11 - Python C++ Binding (used)
- Robust Geometric Predicates - Original Robust Geometric predicates
- Updated Predicates -Updated geometric predicate library (used)

Related Methods

2D ConcaveHull Extraction

- CGAL Alpha Shapes - MultiPolygon with holes
- PostGIS ConcaveHull - Single Polygon with holes
- Spatialite ConcaveHull - MultiPolygon with holes
- Concaveman - A 2D concave hull extraction algorithm for 2D point sets

Contributing

Any help or suggestions would be appreciated!

Citation

2D

If are using Polylidar for 2D work please cite:

J. Castagno and E. Atkins, "Polylidar - Polygons From Triangular Meshes," in IEEE Robotics and Automation Letters, vol. 5, no. 3, pp. 4634-4641, July 2020, doi: 10.1109/LRA.2020.3002212. Link to Paper

```
@ARTICLE{9117017,
  author={J. {Castagno} and E. {Atkins}},
  journal={IEEE Robotics and Automation Letters},
  title={Polylidar - Polygons From Triangular Meshes},
  year={2020},
  volume={5},
  number={3},
  pages={4634-4641}
}
```

3D

If you are using Polylidar3D for 3D work please cite:

Figure A.2: Page 2 of the Polylidar3D repository

J. Castagno and E. Atkins, "Polylidar3D - Fast Polygon Extraction from 3D Data," in MDPI Sensors, vol. 20, no.17, 4819, September 2020, doi: 10.3390/s20174819 Link to Paper

```
@Article{s20174819,
author = {Castagno, Jeremy and Atkins, Ella},
title = {Polylidar3D-Fast Polygon Extraction from 3D Data},
journal = {Sensors},
volume = {20},
year = {2020},
number = {17},
article-number = {4819},
url = {https://www.mdpi.com/1424-8220/20/17/4819},
issn = {1424-8220}
}
```

License

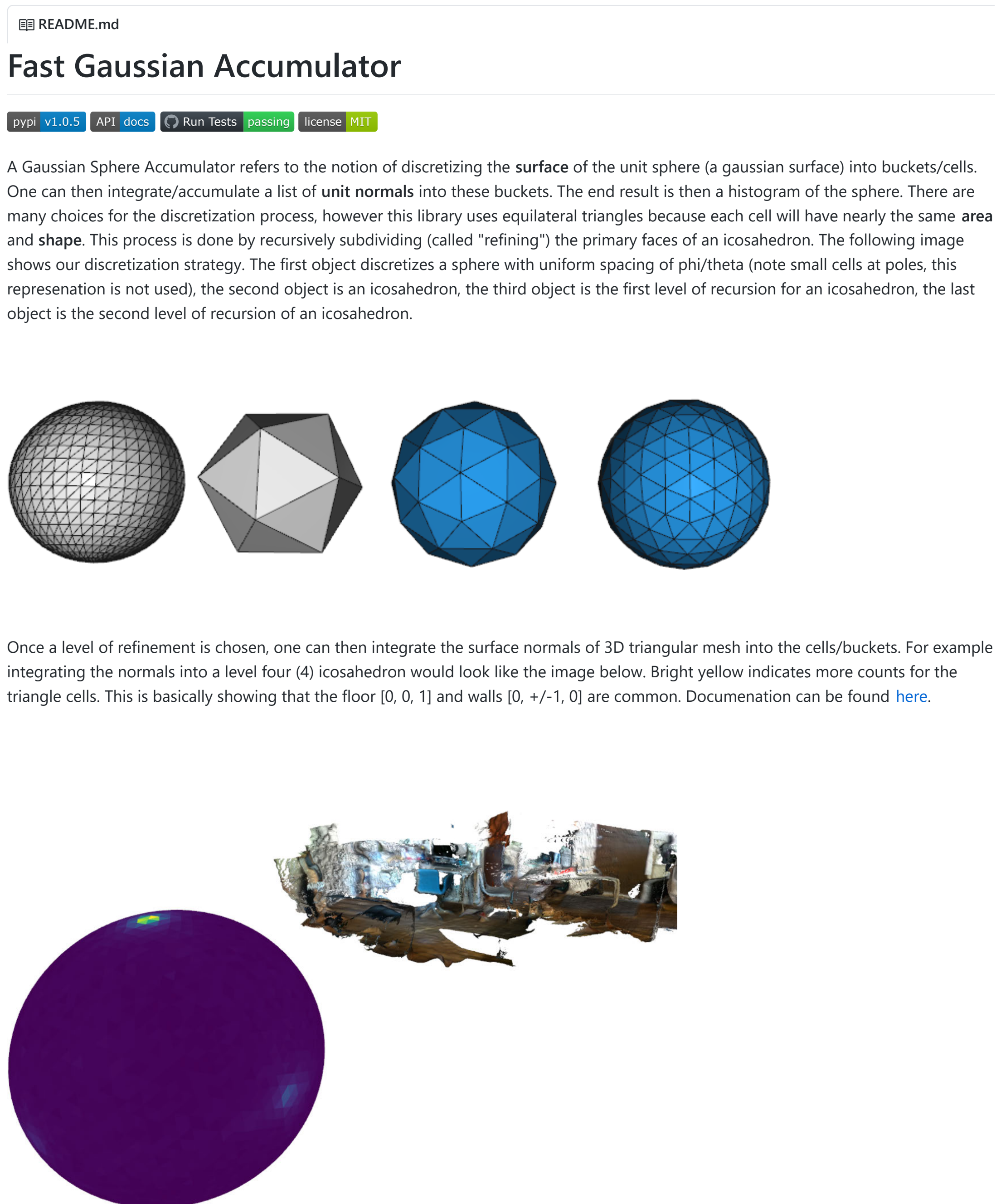
MIT

> GitHub @jeremybyu

Figure A.3: Page 3 of the Polylidar3D repository

A.2 Fast Gaussian Accumulator Source Code Summary

Below is a multi-page screenshot of the `README.md` file for the FastGA source code repository hosted at `https://github.com/JeremyBYU/FastGaussianAccumulator`.



README.md

Fast Gaussian Accumulator

pypi v1.0.5 | API docs | Run Tests passing | license MIT

A Gaussian Sphere Accumulator refers to the notion of discretizing the **surface** of the unit sphere (a gaussian surface) into buckets/cells. One can then integrate/accumulate a list of **unit normals** into these buckets. The end result is then a histogram of the sphere. There are many choices for the discretization process, however this library uses equilateral triangles because each cell will have nearly the same **area** and **shape**. This process is done by recursively subdividing (called "refining") the primary faces of an icosahedron. The following image shows our discretization strategy. The first object discretizes a sphere with uniform spacing of phi/theta (note small cells at poles, this represenation is not used), the second object is an icosahedron, the third object is the first level of recursion for an icosahedron, the last object is the second level of recursion of an icosahedron.

Once a level of refinement is chosen, one can then integrate the surface normals of 3D triangular mesh into the cells/buckets. For example integrating the normals into a level four (4) icosahedron would look like the image below. Bright yellow indicates more counts for the triangle cells. This is basically showing that the floor [0, 0, 1] and walls [0, +/-1, 0] are common. Documenation can be found here.

Figure A.4: Page 1 of the Fast Gaussian Accumulator repository

Integrating Normals into the Gaussian Accumulator

To integrate normals into the Gaussian Accumulator one must **find** the cell that corresponds to the normal. This is a search process that has been implemented in several fashions in this repo. The main ways are as follows:

- 3D KD Tree - Do a nearest neighbor search using a binary tree.
  - `GaussianAccumulatorKDPY` - One implementation using scipy kdtree.
  - `GaussianAccumulatorKD` One implementation uses C++ nanoflann.
- Global Index and Local Search - A 3D point is transformed to a unique integer id. The unique ids have the property that ids close to each other will be close to each other in 3D space. The closest id is found corresponding to a triangle cell. A local search of triangle neighbors is performed to find closest triangle cell to the point.
  - `GaussianAccumulatorOpt` - Works good on **only** on the top hemisphere. Projects 3D point to plane using Azimuth Equal Area projection. Convert 2D point to int32 index using Hilbert Curve. This implementation is severely limited and is not recommended.
  - `GaussianAccumulatorS2Beta` - Works on full sphere! Uses Googles S2 space filling curve (uint64). 3D point is projected to unit cube, assigned to a face of the cube, and then a Hilbert curve index is found for that cube face. This is recommended, and what I use.

Use GaussianAccumulatorS2Beta! Look at `python -m examples.python.run_normals`

Peak Detection

There are two (2) peak detection methods used within this repository. The user can choose which one best suit their needs.

2D Image Peak Detection

This method basically unwraps the icosahedron as a 2D image in a very particular way as described by Gauge Equivariant Convolutional Networks and the Icosahedral CNN. This unwrapping is hardcoded and fixed once a refinement level is chosen so it is very fast. The library then uses a 2D peak detector algorithm followed up with agglomerative hierarchial clustering (AHC) to group similar peaks. All of this is user configurable.

1D Signal Peak Detection

This performs peak detection on the 1D thread following the hilbert curve. This produces more peaks which are actually near each other or S2 and are then grouped with AHC. This actually works pretty well, but I recommend to use the 2D Image Peak Detector.

Installation

For python there are pre-built binary wheel on PyPI for Windows and Linux. You can install with `pip install fastgac` .

Below are instruction to build the C++ Package (and python package) manaully with CMake. Installation is entirely through CMake now. You must have CMake 3.14 or higher installed and a C++ compiler with C++ 14 or higher.

For C++ Users

1. `mkdir cmake-build && cd cmake-build` . - create build folder directory
2. `cmake ../ -DCMAKE_BUILD_TYPE=Release` . For windows also add `-DCMAKE_GENERATOR_PLATFORM=x64`
3. `cmake --build . -j4 --config Release` - Build FastGA

For Python Users (Requires CMake)

1. Install conda or create a python virtual environment (Why?). I recommend conda for Windows users.
2. `pip install .`

If you want to run the examples then you need to install the following (from main directory):

Figure A.5: Page 2 of the Fast Gaussian Accumulator repository

1. `pip install -r dev-requirements.txt`

Build and Install Python Extension and C++

Here building is entirely in CMake. You will build C++ Library and Python extension manually with CMake Commands.

1. Install conda or create a python virtual environment (Why?). I recommend conda for Windows users.
2. `cd cmake-build && cmake --build . --target python-package --config Release -j$(nproc)`
3. `cd lib/python_package && pip install -e .`

If you want to run the examples then you need to install the following (from main directory):

1. `pip install -r dev-requirements.txt`

Documentation

Please see documentation website for more details.

Citation

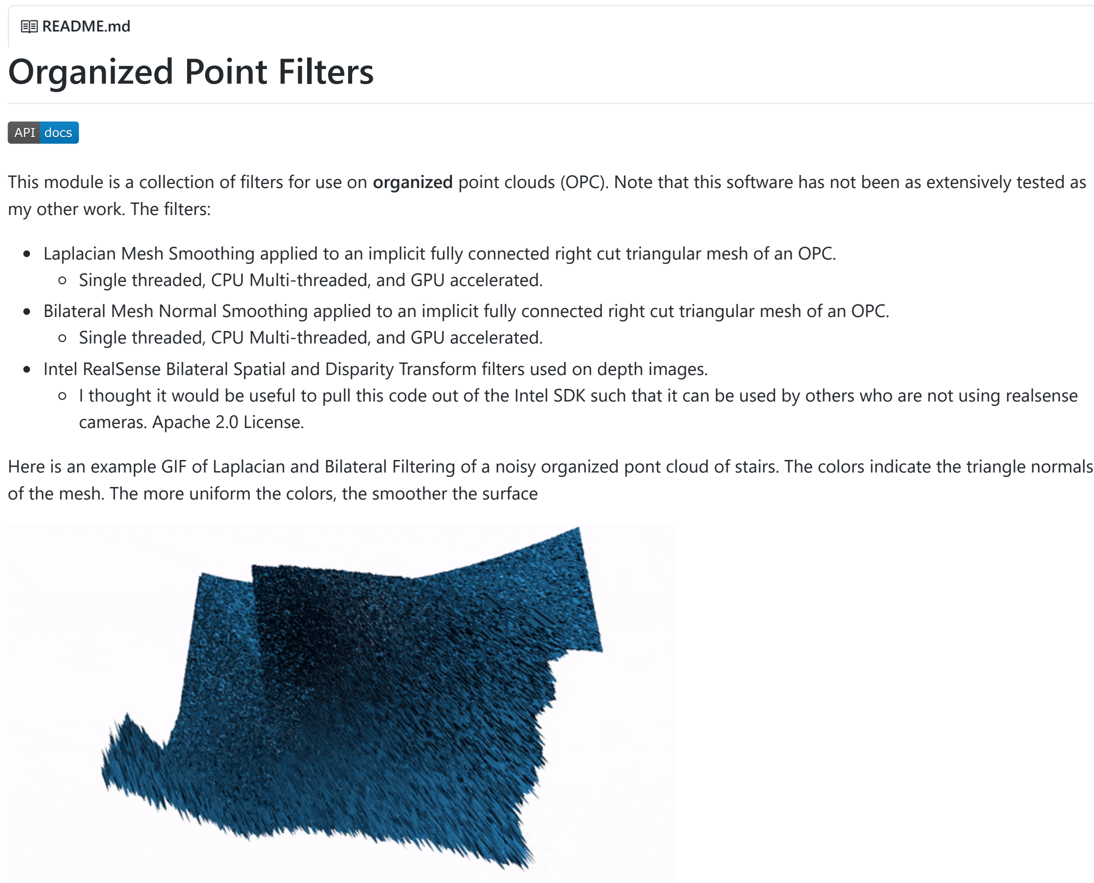
To support our work please cite:

```
@Article{s20174819,
author = {Castagno, Jeremy and Atkins, Ella},
title = {Polylidar3D-Fast Polygon Extraction from 3D Data},
journal = {Sensors},
volume = {20},
year = {2020},
number = {17},
article-number = {4819},
url = {https://www.mdpi.com/1424-8220/20/17/4819},
issn = {1424-8220}
}
```

Figure A.6: Page 3 of the Fast Gaussian Accumulator repository

A.3 Organized Point Filters Source Code Summary

Below is a multi-page screenshot of the `README.md` file for the OPF source code repository hosted at `https://github.com/JeremyBYU/OrganizedPointFilters`.



README.md

Organized Point Filters

API docs

This module is a collection of filters for use on **organized** point clouds (OPC). Note that this software has not been as extensively tested as my other work. The filters:

- Laplacian Mesh Smoothing applied to an implicit fully connected right cut triangular mesh of an OPC.
  - Single threaded, CPU Multi-threaded, and GPU accelerated.
- Bilateral Mesh Normal Smoothing applied to an implicit fully connected right cut triangular mesh of an OPC.
  - Single threaded, CPU Multi-threaded, and GPU accelerated.
- Intel RealSense Bilateral Spatial and Disparity Transform filters used on depth images.
  - I thought it would be useful to pull this code out of the Intel SDK such that it can be used by others who are not using realsense cameras. Apache 2.0 License.

Here is an example GIF of Laplacian and Bilateral Filtering of a noisy organized pont cloud of stairs. The colors indicate the triangle normals of the mesh. The more uniform the colors, the smoother the surface

Installation

Installation is entirely through CMake now. You must have CMake 3.14 or higher installed and a C++ compiler with C++ 14 or higher. No built binaries are included currently.

Build Project Library

1. `mkdir cmake-build && cd cmake-build` . - create build folder directory
2. `cmake ../ -DCMAKE_BUILD_TYPE=Release` . For windows also add `-DCMAKE_GENERATOR_PLATFORM=x64`
3. `cmake --build . -j$(nproc)` - Build OPF

Build and Install Python Extension

1. Install conda or create a python virtual envrionment (Why?). I recommend conda for Windows users.
2. `cd cmake-build && cmake --build . --target python-package --config Release -j$(nproc)`
3. `cd lib/python_package && pip install -e .`

If you want to run the examples then you need to install the following (from main directory):

Figure A.7: Page 1 of the Organized Point Filter repository

1. `pip install -r dev-requirements.txt`

You also need `cupy` to be installed with cuda device drivers if you want GPU acceleration. I cant vouch that this will always work:

1. `conda install cudatoolkit=10.1`
2. `pip install cupy-cuda101`

Documentation

Please see documentation website for more details.

Citation

To support our work please cite:

```
@article{s20174819,
author = {Castagno, Jeremy and Atkins, Ella},
title = {Polylidar3D - Fast Polygon Extraction from 3D Data},
journal = {Sensors},
volume = {20},
year = {2020},
number = {17},
article-number = {4819},
url = {https://www.mdpi.com/1424-8220/20/17/4819},
issn = {1424-8220}
}
```

Figure A.8: Page 2 of the Organized Point Filter repository

BIBLIOGRAPHY

[1] Winnefeld, J. A. and Kendall, F., "Unmanned Systems Integrated Roadmap FY 2011-2036," *Office of the Secretary of Defense. US*, 2011.

[2] DeGarmo, M. T., "Issues Concerning Integration of Unmanned Aerial Vehicles in Civil Airspace," Tech. rep., The Mitre Corporation, Sept. 2013.

[3] Atkins, E. M., Portillo, I. A., and Strube, M. J., "Emergency Flight Planning Applied to Total Loss of Thrust," *Journal of Aircraft*, Vol. 43, No. 4, July 2006, pp. 1205–1216.

[4] Ten Harmsel, A. J., Olson, I. J., and Atkins, E. M., "Emergency Flight Planning for an Energy-Constrained Multicopter," *Journal of Intelligent & Robotic Systems*, Vol. 85, No. 1, Jan. 2017, pp. 145–165.

[5] Ochoa, C. A. and Atkins, E. M., "Fail-Safe Navigation for Autonomous Urban Multicopter Flight," *AIAA Information Systems-AIAA Infotech @ Aerospace*, AIAA SciTech Forum, American Institute of Aeronautics and Astronautics, Jan. 2017.

[6] Sankararaman, S., "Towards A Computational Framework for Autonomous Decision-Making in Unmanned Aerial Vehicles," *AIAA Information Systems-AIAA Infotech @ Aerospace*, AIAA SciTech Forum, American Institute of Aeronautics and Astronautics, Jan. 2017.

[7] Stansbury, R. S. and Wilson, T. A., "Technology Surveys and Regulatory Gap Analyses of UAS Subsystems toward Access to the NAS," *Handbook of Unmanned Aerial Vehicles*, edited by K. P. Valavanis and G. J. Vachtsevanos, Springer Netherlands, Dordrecht, 2015, pp. 2293–2338.

[8] Mejias Alvarez, L., Fitzgerald, D., Eng, P., and Liu, X., "Forced Landing Technologies for Unmanned Aerial Vehicles: Towards Safer Operations," *Aerial Vehicles*, edited by T. M. Lam, InTech, Austria, 2009, pp. 415–442.

[9] Warren, M., Mejias, L., Yang, X., Arain, B., Gonzalez, F., and Upcroft, B., "Enabling Aircraft Emergency Landings Using Active Visual Site Detection," *Field and Service Robotics: Results of the 9th International Conference*, edited by L. Mejias, P. Corke, and J. Roberts, Springer Tracts in Advanced Robotics, Springer International Publishing, Cham, 2015, pp. 167–181.

[10] Jin, S., Zhang, J., Shen, L., and Li, T., "On-board vision autonomous landing techniques for quadrotor: A survey," *2016 35th Chinese Control Conference (CCC)*, July 2016, pp. 10284–10289, ISSN: 1934-1768.

[11] Yang, S., Scherer, S. A., Schauwecker, K., and Zell, A., "Autonomous Landing of MAVs on an Arbitrarily Textured Landing Site Using Onboard Monocular Vision," *Journal of Intelligent & Robotic Systems*, Vol. 74, No. 1, April 2014, pp. 27–43.

[12] Wu, X. and Mueller, M., "Towards a Consequences-Aware Emergency Landing System for Unmanned Aerial Systems," *2018 International Conference on Unmanned Aircraft Systems (ICUAS)*, IEEE, June 2018.

[13] Shen, Y.-F., Rahman, Z., Krusienski, D., and Li, J., "A Vision-Based Automatic Safe Landing-Site Detection System," *IEEE Transactions on Aerospace and Electronic Systems*, Vol. 49, No. 1, Jan. 2013, pp. 294–311.

[14] Forster, C., Faessler, M., Fontana, F., Werlberger, M., and Scaramuzza, D., "Continuous On-Board Monocular-Vision-Based Elevation Mapping Applied to Autonomous Landing of Micro Aerial Vehicles," *2015 IEEE International Conference on Robotics and Automation (ICRA)*, May 2015, pp. 111–118.

[15] Desaraju, V. R., Michael, N., Humenberger, M., Brockers, R., Weiss, S., Nash, J., and Matthies, L., "Vision-Based Landing Site Evaluation and Informed Optimal Trajectory Generation toward Autonomous Rooftop Landing," *Autonomous Robots*, Vol. 39, No. 3, Oct. 2015, pp. 445–463.

[16] Scherer, S., Chamberlain, L., and Singh, S., "Autonomous landing at unprepared sites by a full-scale helicopter," *Robotics and Autonomous Systems*, Vol. 60, No. 12, Dec. 2012, pp. 1545–1562.

[17] Theodore, C., Rowley, D., Ansar, A., Matthies, L., Goldberg, S., Hubbard, D., and Whalley, M., "Flight Trials of a Rotorcraft Unmanned Aerial Vehicle Landing Autonomously at Unprepared Sites," *Annual Forum Proc. American Helicopter Society*, Vol. 62, 2006, p. 1250.

[18] Papa, U., Ariante, G., and Del Core, G., "UAS Aided Landing and Obstacle Detection Through LIDAR-Sonar Data," *2018 5th IEEE International Workshop on Metrology for AeroSpace (MetroAeroSpace)*, June 2018, pp. 478–483.

[19] Bleier, M., Settele, F., Krauss, M., Knoll, A., and Schilling, K., "Risk Assessment of Flight Paths for Automatic Emergency Parachute Deployment in UAVs," *IFAC-PapersOnLine*, Vol. 48, No. 9, Jan. 2015, pp. 180–185.

[20] Garg, M., Kumar, A., and Sujit, P., "Terrain-based landing site selection and path planning for fixed-wing UAVs," *2015 International Conference on Unmanned Aircraft Systems (ICUAS)*, June 2015, pp. 246–251.

[21] Frantis, P., "Emergency and precautionary landing assistant," *2011 IEEE/AIAA 30th Digital Avionics Systems Conference*, Oct. 2011, pp. 6E2–1–6E2–6, ISSN: 2155-7209.

[22] Yan, L., Qi, J., Wang, M., Wu, C., and Xin, J., "A Safe Landing Site Selection Method of UAVs Based on LiDAR Point Clouds," *2020 39th Chinese Control Conference (CCC)*, July 2020, pp. 6497–6502, ISSN: 1934-1768.

[23] Maturana, D. and Scherer, S., “3D Convolutional Neural Networks for landing zone detection from LiDAR,” *2015 IEEE International Conference on Robotics and Automation (ICRA)*, May 2015, pp. 3471–3478, ISSN: 1050-4729.

[24] Di Donato, P. F. A. and Atkins, E. M., “Evaluating Risk to People and Property for Aircraft Emergency Landing Planning,” *Journal of Aerospace Information Systems*, Vol. 14, No. 5, 2017, pp. 259–278.

[25] Poissant, A., Castano, L., and Xu, H., “Mitigation of Ground Impact Hazard for Safe Unmanned Aerial Vehicle Operations,” *Journal of Aerospace Information Systems*, Vol. 17, No. 12, 2020, pp. 647–658, _eprint: https://doi.org/10.2514/1.I010797.

[26] Melnyk, R., Schrage, D., Volovoi, V., and Jimenez, H., “A Third-Party Casualty Risk Model for Unmanned Aircraft System Operations,” *Reliability Engineering & System Safety*, Vol. 124, April 2014, pp. 105–116.

[27] Li, K., Liu, P., Pang, T., Yang, Z., and Chen, B. M., “Development of an unmanned aerial vehicle for rooftop landing and surveillance,” *2015 International Conference on Unmanned Aircraft Systems (ICUAS)*, June 2015, pp. 832–838.

[28] Antenucci, J. C., Brown, K., Croswell, P. L., Kevany, M. J., and Archer, H., *GEOGRAPHIC INFORMATION SYSTEMS: A GUIDE TO THE TECHNOLOGY*, Springer US, 1991.

[29] Kumar, M., “World geodetic system 1984: A modern and accurate global reference frame,” *Marine Geodesy*, Vol. 12, No. 2, 1988, pp. 117–126.

[30] Galati, S. R., *Geographic Information Systems Demystified.*, Artech House Mobile Communications Series, Artech House, Inc, 2006.

[31] PROJ contributors, “PROJ Coordinate Transformation Software Library,” 2018.

[32] Hollings, T., Burgman, M., van Andel, M., Gilbert, M., Robinson, T., and Robinson, A., “How Do You Find the Green Sheep? A Critical Review of the Use of Remotely Sensed Imagery to Detect and Count Animals,” *Methods in Ecology and Evolution*, Vol. 9, No. 4, 2018, pp. 881–892.

[33] Shao, H., Song, P., Mu, B., Tian, G., Chen, Q., He, R., and Kim, G., “Assessing City-Scale Green Roof Development Potential Using Unmanned Aerial Vehicle (UAV) Imagery,” *Urban Forestry & Urban Greening*, Vol. 57, 2021, pp. 126954.

[34] Meuleau, N., Plaunt, C., Smith, D. E., and Smith, T., “An Emergency Landing Planner for Damaged Aircraft,” *Proc. of the 21st Innovative Applications of Artificial Intelligence Conf.*, 2009, pp. 71–80.

[35] Patterson, T., McClean, S., Morrow, P., Parr, G., and Luo, C., “Timely Autonomous Identification of UAV Safe Landing Zones,” *Image and Vision Computing*, Vol. 32, No. 9, Sept. 2014, pp. 568–578.

[36] Castagno, J., "Github - Polylidar," [Online] Available: https://github.com/JeremyBYU/polylidar, 2020, Accessed: 2019-01-05.

[37] Duckham, M., Kulik, L., Worboys, M., and Galton, A., "Efficient Generation of Simple Polygons for Characterizing the Shape of a Set of Points in the Plane," *Pattern Recognition*, Vol. 41, No. 10, Oct. 2008, pp. 3224–3236.

[38] Edelsbrunner, H., Kirkpatrick, D., and Seidel, R., "On the Shape of a Set of Points in the Plane," *IEEE Transactions on Information Theory*, Vol. 29, No. 4, July 1983, pp. 551–559.

[39] Furieri, A., "Spatialite," [Online] Available: https://www.gaia-gis.it/fossil/libspatialite/index, 2017, Visited on 2017-01-14.

[40] Hipp, D. R., Kennedy, D., and Mistachkin, J., "SQLite (Version 3.28) SQLite Development Team," [Online] Available: https://www.sqlite.org/index.html, 2020, Visited on 2020-06-14.

[41] Open Source Geospatial Foundation, "PostGIS," [Online] Available: https://postgis.net/docs/ST_ConcaveHull.html, 2019, Visited on 2019-01-14.

[42] The CGAL Project, *CGAL User and Reference Manual*, CGAL Editorial Board, 4th ed., 2019.

[43] Herring, J. R., "OpenGIS Implementation Specification for Geographic Information-Simple Feature Access- Part 1: Common Architecture," *Open Geospatial Consortium*, 2006, pp. 95.

[44] Mapbox, "Github - Delaunator," [Online] Available: https://github.com/mapbox/delaunator, 2018, Accessed: 2018-01-05.

[45] Bell, A., "Github - Port of Delaunator to C++," [Online] Available: https://github.com/abellgithub/delaunator-cpp, 2018, Accessed: 2018-01-05.

[46] Richard Shewchuk, J., "Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates," *Discrete & Computational Geometry*, Vol. 18, No. 3, Oct. 1997, pp. 305–363.

[47] Cao, R., Zhang, Y., Liu, X., and Zhao, Z., "Roof Plane Extraction from Airborne Lidar Point Clouds," *International Journal of Remote Sensing*, Vol. 38, No. 12, June 2017, pp. 3684–3703.

[48] de Berg, M., Cheong, O., van Kreveld, M., and Overmars, M., "Delaunay Triangulations: Height Interpolation," *Computational Geometry: Algorithms and Applications*, 2008, pp. 191–218.

[49] Feng, C., Taguchi, Y., and Kamat, V. R., "Fast Plane Extraction in Organized Point Clouds Using Agglomerative Hierarchical Clustering," *2014 IEEE International Conference on Robotics and Automation (ICRA)*, May 2014, pp. 6218–6225.

[50] Pham, T. T., Eich, M., Reid, I., and Wyeth, G., "Geometrically Consistent Plane Extraction for Dense Indoor 3D Maps Segmentation," *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Oct. 2016, pp. 4199–4204.

[51] Holz, D. and Behnke, S., "Fast Range Image Segmentation and Smoothing Using Approximate Surface Reconstruction and Region Growing," *Intelligent Autonomous Systems 12*, edited by S. Lee, H. Cho, K.-J. Yoon, and J. Lee, Vol. 194, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 61–73.

[52] Huang, T., Lin, C., Guo, G., and Wong, M., "Cpp-Taskflow: Fast Task-Based Parallel Programming Using Modern C++," *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2019, pp. 974–983.

[53] Pathak, K., Birk, A., Vaskevicius, N., Pfingsthorn, M., Schwertfeger, S., and Poppinga, J., "Online Three-Dimensional SLAM by Registration of Large Planar Surface Segments and Closed-Form Pose-Graph Relaxation," *Journal of Field Robotics*, Vol. 27, No. 1, 2010, pp. 52–84.

[54] Malihi, S., Valadan Zoej, M. J., Hahn, M., Mokhtarzade, M., and Arefi, H., "3D Building Reconstruction Using Dense Photogrammetric Point Cloud," *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, Vol. XLI-B3, Copernicus GmbH, June 2016, pp. 71–74.

[55] Lerma, J. L., Navarro, S., Cabrelles, M., and Villaverde, V., "Terrestrial Laser Scanning and Close Range Photogrammetry for 3D Archaeological Documentation: The Upper Palaeolithic Cave of Parpalló as a Case Study," *Journal of Archaeological Science*, Vol. 37, No. 3, March 2010, pp. 499–507.

[56] Balsa-Barreiro, J. and Fritsch, D., "Generation of Visually Aesthetic and Detailed 3D Models of Historical Cities by Using Laser Scanning and Digital Photogrammetry," *Digital Applications in Archaeology and Cultural Heritage*, Vol. 8, March 2018, pp. 57–64.

[57] Rusinkiewicz, S. and Levoy, M., "Efficient Variants of the ICP Algorithm," *Proceedings Third International Conference on 3-D Digital Imaging and Modeling*, May 2001, pp. 145–152.

[58] Schaefer, A., Vertens, J., Buscher, D., and Burgard, W., "A Maximum Likelihood Approach to Extract Finite Planes from 3-D Laser Scans," *2019 International Conference on Robotics and Automation (ICRA)*, IEEE, Montreal, QC, Canada, May 2019, pp. 72–78.

[59] Lee, T., Lim, S., Lee, S., An, S., and Oh, S., "Indoor Mapping Using Planes Extracted from Noisy RGB-D Sensors," *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Oct. 2012, pp. 1727–1733.

[60] Biswas, J. and Veloso, M., "Planar Polygon Extraction and Merging from Depth Images," *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Oct. 2012, pp. 3859–3864.

[61] Castagno, J., "Github - Fast Gaussian Sphere Accumulator," [Online] Available: `https://github.com/JeremyBYU/FastGaussianAccumulator`, 2020, Accessed: 2020-06-05.

[62] Castagno, J., "Github - Polylidard3D and KITTI," [Online] Available: `https://github.com/JeremyBYU/polylidar-kitti`, 2020, Accessed: 2020-06-05.

[63] Castagno, J., "Github - Polylidar3D with RealSense," [Online] Available: `https://github.com/JeremyBYU/polylidar-realsense`, 2020, Accessed: 2020-06-05.

[64] Castagno, J., "Github - Polylidard3D and SynPEB," [Online] Available: `https://github.com/JeremyBYU/polylidar-plane-benchmark`, 2020, Accessed: 2020-06-05.

[65] Castagno, J., "Github - Organized Point Filters," [Online] Available: `https://github.com/JeremyBYU/OrganizedPointFilters`, 2020, Accessed: 2020-06-05.

[66] Kaiser, A., Ybanez Zepeda, J. A., and Boubekeur, T., "A Survey of Simple Geometric Primitives Detection Methods for Captured 3D Data," *Computer Graphics Forum*, Vol. 38, No. 1, Feb. 2019, pp. 167–196.

[67] Trevor, A., Gedikli, S., Rusu, R., and Christensen, H., "Efficient Organized Point Cloud Segmentation with Connected Components," *3rd Workshop on Semantic Perception Mapping and Exploration (SPME), Karlsruhe, Germany*, 2013.

[68] Salas-Moreno, R. F., Glocken, B., Kelly, P. H. J., and Davison, A. J., "Dense Planar SLAM," *2014 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*, Sept. 2014, pp. 157–164.

[69] Oesau, S., Lafarge, F., and Alliez, P., "Planar Shape Detection and Regularization in Tandem: Planar Shape Detection and Regularization in Tandem," *Computer Graphics Forum*, Vol. 35, No. 1, Feb. 2016, pp. 203–215.

[70] Poppinga, J., Vaskevicius, N., Birk, A., and Pathak, K., "Fast Plane Detection and Polygonalization in Noisy 3D Range Images," *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, IEEE, Nice, Sept. 2008, pp. 3378–3383.

[71] Castagno, J., "Github - Benchmark Concave Hull," [Online] Available: `https://github.com/JeremyBYU/concavehull-evaluation`, 2020, Accessed: 2020-01-05.

[72] Taubin, G., "Curve and Surface Smoothing without Shrinkage," *Proceedings of IEEE International Conference on Computer Vision*, June 1995, pp. 852–857.

[73] Zheng, Y., Fu, H., Au, O. K.-C., and Tai, C.-L., "Bilateral Normal Filtering for Mesh Denoising," *IEEE transactions on visualization and computer graphics*, Vol. 17, No. 10, Oct. 2011, pp. 1521–1530.

[74] Sun, X., Rosin, P. L., Martin, R., and Langbein, F., "Fast and Effective Feature-Preserving Mesh Denoising," *IEEE Transactions on Visualization and Computer Graphics*, Vol. 13, No. 5, Sept. 2007, pp. 925–938.

[75] Borrmann, D., Elseberg, J., Lingemann, K., and Nüchter, A., "The 3D Hough Transform for Plane Detection in Point Clouds: A Review and a New Accumulator Design," *3D Research*, Vol. 2, No. 2, June 2011, pp. 3.

[76] Limberger, F. A. and Oliveira, M. M., "Real-Time Detection of Planar Regions in Unorganized Point Clouds," *Pattern Recognition*, Vol. 48, No. 6, June 2015, pp. 2043–2053.

[77] Cohen, T. S., Weiler, M., Kicanaoglu, B., and Welling, M., "Gauge Equivariant Convolutional Networks and the Icosahedral CNN," *arXiv:1902.04615 [cs, stat]*, May 2019, Comment: Proceedings of the International Conference on Machine Learning (ICML), 2019.

[78] Toony, Z., Laurendeau, D., and Gagné, C., "Describing 3D Geometric Primitives Using the Gaussian Sphere and the Gaussian Accumulator," *3D Research*, Vol. 6, No. 4, Nov. 2015, pp. 42.

[79] Paris, R., "Modified Half-Edge Data Structure and Its Applications to 3D Mesh Generation for Complex Tube Networks." *Electronic Theses and Dissertations*, May 2013.

[80] Kazhdan, M. and Hoppe, H., "Screened Poisson Surface Reconstruction," *ACM Transactions on Graphics*, Vol. 32, No. 3, July 2013, pp. 29:1–29:13.

[81] Bernardini, F., Mittleman, J., Rushmeier, H., Silva, C., and Taubin, G., "The Ball-Pivoting Algorithm for Surface Reconstruction," *IEEE Transactions on Visualization and Computer Graphics*, Vol. 5, No. 4, Oct. 1999, pp. 349–359.

[82] Zhou, Q.-Y. and Koltun, V., "Dense Scene Reconstruction with Points of Interest," *ACM Transactions on Graphics*, Vol. 32, No. 4, July 2013, pp. 112:1–112:8.

[83] Zhou, Q.-Y., Park, J., and Koltun, V., "Open3D: A Modern Library for 3D Data Processing," *arXiv preprint arXiv:1801.09847*, 2018.

[84] Castagno, J. and Atkins, E., "Polylidar - Polygons From Triangular Meshes," *IEEE Robotics and Automation Letters*, Vol. 5, No. 3, July 2020, pp. 4634–4641.

[85] Wenninger, M. J., *Spherical Models*, Vol. 3, Courier Corporation, 1999.

[86] Mokbel, M. F. and Aref, W. G., "Space-Filling Curves," *Encyclopedia of GIS*, edited by S. Shekhar and H. Xiong, Springer US, Boston, MA, 2008, pp. 1068–1072.

[87] Google, "S2 Geometry," [Online] Available: `https://github.com/google/s2geometry`, 2020, Accessed: 2020-06-05.

[88] Van Sandt, P., Chronis, Y., and Patel, J. M., "Efficiently Searching In-Memory Sorted Arrays: Revenge of the Interpolation Search?" *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, Association for Computing Machinery, Amsterdam, Netherlands, June 2019, pp. 36–53.

[89] Khuong, P.-V. and Morin, P., "Array Layouts for Comparison-Based Searching," *Journal of Experimental Algorithmics*, Vol. 22, May 2017, pp. 1.3:1–1.3:39.

[90] Dagum, L. and Menon, R., "OpenMP: An Industry Standard API for Shared-Memory Programming," *Computational Science & Engineering, IEEE*, Vol. 5, No. 1, 1998, pp. 46–55.

[91] Google, "Github - A Hybrid Thread/Fiber Task Scheduler," [Online] Available: `https://github.com/google/marl`, 2020, Accessed: 2020-06-05.

[92] Douglas, D. H. and Peucker, T. K., "Algorithms for the Reduction of the Number of Points Required to Represent a Digitized Line or Its Caricature," *Cartographica: The International Journal for Geographic Information and Geovisualization*, Vol. 10, No. 2, Dec. 1973, pp. 112–122.

[93] Agarwal, P. K., Flato, E., and Halperin, D., "Polygon Decomposition for Efficient Construction of Minkowski Sums," *Computational Geometry*, Vol. 21, No. 1-2, 2002, pp. 39–61.

[94] Gillies, S. et al., "Github - Shapely: Manipulation and Analysis of Geometric Objects," [Online] Available: `https://github.com/Toblerity/Shapely`, 2020, Accessed: 2020-06-05.

[95] Blanco, J. L. and Rai, P. K., "Nanoflann: A C++ Header-Only Fork of FLANN, a Library for Nearest Neighbor (NN) with KD-Trees," 2014.

[96] Google, "Github - Google Benchmark," [Online] Available: `https://github.com/google/benchmark`, 2020, Accessed: 2020-06-05.

[97] Van der Walt, S., Schönberger, J. L., Nunez-Iglesias, J., Boulogne, F., Warner, J. D., Yager, N., Gouillart, E., Yu, T., and scikit-image contributors, t., "Scikit-Image: Image Processing in Python," *PeerJ Inc.*, Vol. 2, June 2014, pp. e453.

[98] Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., Carey, C. J., Polat, I., Feng, Y., Moore, E. W., VanderPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E. A., Harris, C. R., Archibald, A. M., Ribeiro, A. H., Pedregosa, F., van Mulbregt, P., and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, Vol. 17, 2020, pp. 261–272.

[99] Open NRW, "Open Geo Data," [Online] Available: `https://www.opengeodata.nrw.de/produkte/geobasis/dom/dom1l/`, 2017, Accessed: 2017-09-05.

[100] Land NRW, "Witten County Portal," [Online] Available: `https://www.land.nrw/de/tags/open-data`, 2018, Data was retrieved through ArcGIS World Imagery.

[101] Geiger, A., Lenz, P., Stiller, C., and Urtasun, R., "Vision Meets Robotics: The KITTI Dataset," *International Journal of Robotics Research (IJRR)*, Vol. 32, No. 11, 2013, pp. 1231–1237.

[102] Castagno, Jeremy, "Polylidar3D Kitti Videos," [Online] Available: `https://drive.google.com/drive/folders/18R0alYprRYgwz5_MyzcdOQzf44496DOz`, 2020, Accessed: 2020-06-30.

[103] Ahn, M. S., Chae, H., Noh, D., Nam, H., and Hong, D., "Analysis and Noise Modeling of the Intel RealSense D435 for Mobile Robots," *2019 16th International Conference on Ubiquitous Robots (UR)*, June 2019, pp. 707–711.

[104] Intel, "Github - Intel RealSense Post Processing," [Online] Available: `https://github.com/IntelRealSense/librealsense/tree/master/examples/post-processing`, 2019, Accessed: 2019-01-05.

[105] Hoover, A., Jean-Baptiste, G., Jiang, X., Flynn, P., Bunke, H., Goldgof, D., Bowyer, K., Eggert, D., Fitzgibbon, A., and Fisher, R., "An Experimental Comparison of Range Image Segmentation Algorithms," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 18, No. 7, July 1996, pp. 673–689.

[106] Torr, P. and Zisserman, A., "MLESAC: A New Robust Estimator with Application to Estimating Image Geometry," *Computer Vision and Image Understanding*, Vol. 78, No. 1, April 2000, pp. 138–156.

[107] Amdahl, G. M., "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), Association for Computing Machinery, Atlantic City, New Jersey, April 1967, pp. 483–485.

[108] Balsa-Barreiro, J., Avariento, J. P., and Lerma, J. L., "Airborne Light Detection and Ranging (LiDAR) Point Density Analysis," *Academic Journals*, Vol. 7, 2012, pp. 3010–3019.

[109] Graham, L., "Random Points: How Dense Are You, Anyway?" http://localhost:10018/2014/07/05/random-points-how-dense-are-you-anyway/, July 2014.

[110] Bergelt, R., Khan, O., and Hardt, W., "Improving the Intrinsic Calibration of a Velodyne LiDAR Sensor," *2017 IEEE SENSORS*, Oct. 2017, pp. 1–3.

[111] Intel, "Github - Intel RealSense SDK," [Online] Available: `https://github.com/IntelRealSense/librealsense/`, 2020, Accessed: 2020-06-05.

[112] Pham, T. T., Do, T.-T., Sünderhauf, N., and Reid, I., "SceneCut: Joint Geometric and Object Segmentation for Indoor Scenes," *2018 IEEE International Conference on Robotics and Automation (ICRA)*, May 2018, pp. 3213–3220.

[113] Castagno, J., Ochoa, C., and Atkins, E., "Comprehensive Risk-Based Planning for Small Unmanned Aircraft System Rooftop Landing," *2018 International Conference on Unmanned Aircraft Systems (ICUAS)*, June 2018, pp. 1031–1040.

[114] OpenStreetMap contributors, "Planet Dump Retrieved from https://planet.osm.org," 2017.

[115] Castagno, J. and Atkins, E. M., "Automatic Classification of Roof Shapes for Multicopter Emergency Landing Site Selection," *2018 Aviation Technology, Integration, and Operations Conference*, AIAA AVIATION Forum, American Institute of Aeronautics and Astronautics, June 2018.

[116] Zhou, G. and Zhou, X., "Seamless Fusion of LiDAR and Aerial Imagery for Building Extraction," *IEEE Transactions on Geoscience and Remote Sensing*, Vol. 52, No. 11, Nov. 2014, pp. 7393–7407.

[117] Haala, N. and Kada, M., "An Update on Automatic 3D Building Reconstruction," *ISPRS Journal of Photogrammetry and Remote Sensing*, Vol. 65, No. 6, Nov. 2010, pp. 570–580.

[118] Jochem, A., Höfle, B., Rutzinger, M., and Pfeifer, N., "Automatic Roof Plane Detection and Analysis in Airborne Lidar Point Clouds for Solar Potential Assessment," *Sensors*, Vol. 9, No. 7, July 2009, pp. 5241–5262.

[119] Yan, Y., Su, N., Zhao, C., and Wang, L., "A Dynamic Multi-Projection-Contour Approximating Framework for the 3D Reconstruction of Buildings by Super-Generalized Optical Stereo-Pairs," *Sensors*, Vol. 17, No. 9, Sept. 2017, pp. 2153.

[120] Maset, E., Fusiello, A., Crosilla, F., Toldo, R., and Zorzetto, D., "Photogrammetric 3D Building Reconstruction From Thermal Images," *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences*, Vol. IV-2/W3, 2017, pp. 25–32.

[121] Yan, Y., Gao, F., Deng, S., and Su, N., "A Hierarchical Building Segmentation in Digital Surface Models for 3D Reconstruction," *Sensors*, Vol. 17, No. 2, Feb. 2017, pp. 222.

[122] Mohajeri, N., Assouline, D., Guiboud, B., Bill, A., Gudmundsson, A., and Scartezzini, J.-L., "A City-Scale Roof Shape Classification Using Machine Learning for Solar Energy Applications," *Renewable Energy*, Vol. 121, June 2018, pp. 81–93.

[123] Assouline, D., Mohajeri, N., and Scartezzini, J.-L., "Building Rooftop Classification Using Random Forests for Large-Scale PV Deployment," *Earth Resources and Environmental Remote Sensing/GIS Applications VIII*, Vol. 10428, International Society for Optics and Photonics, Oct. 2017, p. 1042806.

[124] Schmidhuber, J., "Deep Learning in Neural Networks: An Overview," *Neural Networks*, Vol. 61, Jan. 2015, pp. 85–117.

[125] Alidoost, F. and Arefi, H., "Knowledge Based 3D Building Model Recognition Using Convolutional Neural Networks From Lidar and Aerial Imageries," *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, Vol. XLI-B3, 2016, pp. 833–840.

[126] Partovi, T., Fraundorfer, F., Azimi, S., Marmanis, D., and Reinartz, P., "Roof Type Selection Based on Patch-Based Classification Using Deep Learning for High Resolution Satellite Imagery," *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, Vol. XLII-1/W1, 2017, pp. 653–657.

[127] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L., "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision*, Vol. 115, No. 3, Dec. 2015, pp. 211–252.

[128] Weiss, K., Khoshgoftaar, T. M., and Wang, D., "A Survey of Transfer Learning," *Journal of Big Data*, Vol. 3, No. 1, May 2016, pp. 9.

[129] Tang, Y., "Deep Learning Using Linear Support Vector Machines," *arXiv:1306.0239 [cs, stat]*, Feb. 2015, Comment: Contribution to the ICML 2013 Challenges in Representation Learning Workshop.

[130] Suveg, I. and Vosselman, G., "Reconstruction of 3D Building Models from Aerial Images and Maps," *ISPRS Journal of Photogrammetry and Remote Sensing*, Vol. 58, No. 3, Jan. 2004, pp. 202–224.

[131] Taillandier, F., "Automatic Building Reconstruction from Cadastral Maps and Aerial Images," *International Archives of Photogrammetry and Remote Sensing*, Vol. 36, No. Part 3, 2005, pp. W24.

[132] Tack, F., Buyuksalih, G., and Goossens, R., "3D Building Reconstruction Based on given Ground Plan Information and Surface Models Extracted from Spaceborne Imagery," *ISPRS Journal of Photogrammetry and Remote Sensing*, Vol. 67, Jan. 2012, pp. 52–64.

[133] Samberg, A., "An implementation of the ASPRS LAS standard," *ISPRS Workshop on Laser Scanning and SilviLaser*, 2007, pp. 363–372.

[134] Lindsay, J. B., "Whitebox GAT: A Case Study in Geomorphometric Analysis," *Computers & Geosciences*, Vol. 95, Oct. 2016, pp. 75–84.

[135] Samosky, J. T., *SectionView–A System for Interactively Specifying and Visualizing Sections through Three-Dimensional Medical Image Data*, PhD Thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1993.

[136] Iglewicz, B. and Hoaglin, D. C., *How to Detect and Handle Outliers*, ASQC Quality Press, 1993.

[137] Hartigan, P., "Algorithm AS 217: Computation of the Dip Statistic to Test for Unimodality," *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, Vol. 34, No. 3, 1985, pp. 320–325.

[138] Freeman, J. B. and Dale, R., "Assessing Bimodality to Detect the Presence of a Dual Cognitive Process," *Behavior Research Methods*, Vol. 45, No. 1, March 2013, pp. 83–97.

[139] He, K., Zhang, X., Ren, S., and Sun, J., "Deep Residual Learning for Image Recognition," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016, pp. 770–778.

[140] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z., "Rethinking the Inception Architecture for Computer Vision," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016, pp. 2818–2826.

[141] Szegedy, C., Ioffe, S., Vanhoucke, V., and Alemi, A. A., "Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning," *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, AAAI'17, AAAI Press, San Francisco, California, USA, Feb. 2017, pp. 4278–4284.

[142] Kingma, D. P. and Ba, J., "Adam: A Method for Stochastic Optimization," *arXiv:1412.6980 [cs]*, Jan. 2017, Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

[143] Chollet, F. et al., "Github - Keras," [Online] Available: `https://github.com/keras-team/keras`, 2015, Accessed: 2018-01-01.

[144] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E., "Scikit-Learn: Machine Learning in Python," *Journal of Machine Learning Research*, Vol. 12, 2011, pp. 2825–2830.

[145] New York State, "2016 Annual Lot New York County," [Online] Available: `http://gis.ny.gov/gateway/orthoprogram/lot16/new-york.htm`, 2018, Data was retrieved through ArcGIS World Imagery.

[146] USGS, "LIDAR Point Cloud NY CMPG 2013," [Online] Available: `ftp://rockyftp.cr.usgs.gov/vdelivery/Datasets/Staged/Elevation/LPC/Projects/USGS_Lidar_Point_Cloud_NY_CMPG_2013_LAS_2015/metadata`, 2018, Accessed: 2018-09-05.

[147] NYC Open Data, "Building Footprints," [Online] Available: `https://data.cityofnewyork.us/Housing-Development/Building-Footprints/nqwf-w8eh`, 2018, Visited on 2019-01-14.

[148] Microsoft, "Bing Maps," [Online] Available: `https://www.bing.com/maps`, 2018, Accessed: 2018-09-05.

[149] USGS, "Lidar Point Cloud; Washtenaw County, MI," [Online] Available: `ftp://rockyftp.cr.usgs.gov/vdelivery/Datasets/Staged/Elevation/LPC/Projects/MI_WashtenawCo_2009/laz`, 2018, Accessed: 2018-09-05.

[150] Canziani, A., Paszke, A., and Culurciello, E., "An Analysis of Deep Neural Network Models for Practical Applications," *arXiv:1605.07678 [cs]*, April 2017.

[151] Rottensteiner, F., Sohn, G., Gerke, M., Wegner, J. D., Breitkopf, U., and Jung, J., "Results of the ISPRS Benchmark on Urban Object Detection and 3D Building Reconstruction," *ISPRS Journal of Photogrammetry and Remote Sensing*, Vol. 93, July 2014, pp. 256–271.

[152] Qi, C. R., Yi, L., Su, H., and Guibas, L. J., "PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space," *Advances in Neural Information Processing Systems 30*, edited by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Curran Associates, Inc., 2017, pp. 5099–5108.

[153] Xu, Y., Fan, T., Xu, M., Zeng, L., and Qiao, Y., "SpiderCNN: Deep Learning on Point Sets with Parameterized Convolutional Filters," *Computer Vision – ECCV 2018*, edited by V. Ferrari, M. Hebert, C. Sminchisescu, and Y. Weiss, Lecture Notes in Computer Science, Springer International Publishing, Cham, 2018, pp. 90–105.

[154] Primatesta, S., Rizzo, A., and la Cour-Harbo, A., "Ground Risk Map for Unmanned Aircraft in Urban Environments," *Journal of Intelligent & Robotic Systems*, Vol. 97, No. 3, March 2020, pp. 489–509.

[155] Saha, M., Sanchez-Ante, G., and Latombe, J., "Planning Multi-Goal Tours for Robot Arms," *2003 IEEE International Conference on Robotics and Automation (Cat. No.03CH37422)*, Vol. 3, Sept. 2003, pp. 3797–3803 vol.3.

[156] Lim, K. L., Yeong, L. S., Ch'ng, S. I., Seng, K. P., and Ang, L., "Uninformed Multigoal Pathfinding on Grid Maps," *2014 International Conference on Information Science, Electronics and Electrical Engineering*, Vol. 3, April 2014, pp. 1552–1556.

[157] Sarne, D., Manisterski, E., and Kraus, S., "Multi-Goal Economic Search Using Dynamic Search Structures," *Autonomous Agents and Multi-Agent Systems*, Vol. 21, No. 2, Sept. 2010, pp. 204–236.

[158] Nash, A., Koenig, S., and Tovey, C., "Lazy Theta*: Any-Angle Path Planning and Path Length Analysis in 3D," *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, AAAI'10, AAAI Press, Atlanta, Georgia, July 2010, pp. 147–154.

[159] Garcia-Castellanos, D. and Lombardo, U., "Poles of Inaccessibility: A Calculation Algorithm for the Remotest Places on Earth," *Scottish Geographical Journal*, Vol. 123, No. 3, Sept. 2007, pp. 227–233.

[160] Mapbox, "Github - Polylabel," [Online] Available: `https://github.com/mapbox/polylabel`, 2018, Accessed: 2018-01-05.

[161] Stevenson, J. D., O'Young, S., and Rolland, L., "Estimated Levels of Safety for Small Unmanned Aerial Vehicles and Risk Mitigation Strategies," *Journal of Unmanned Vehicle Systems*, Sept. 2015.

[162] Ancel, E., Capristan, F. M., Foster, J. V., and Condotta, R. C., "Real-Time Risk Assessment Framework for Unmanned Aircraft System (UAS) Traffic Management (UTM)," *17th AIAA Aviation Technology, Integration, and Operations Conference*, American Institute of Aeronautics and Astronautics, June 2017.

[163] Nagle, N. N., Buttenfield, B. P., Leyk, S., and Spielman, S., "Dasymetric Modeling and Uncertainty," *Annals of the Association of American Geographers*, Vol. 104, No. 1, Jan. 2014, pp. 80–95.

[164] Dmowska, A. and Stepinski, T. F., "A High Resolution Population Grid for the Conterminous United States: The 2010 Edition," *Computers, Environment and Urban Systems*, Vol. 61, Jan. 2017, pp. 13–23.

[165] Federal Aviation Administration, "Code of Federal Regulations (14 CFR) Part 107." 2016.

[166] Prevot, T., Rios, J., Kopardekar, P., III, J. E. R., Johnson, M., and Jung, J., "UAS Traffic Management (UTM) Concept of Operations to Safely Enable Low Altitude Flight Operations," *16th AIAA Aviation Technology, Integration, and Operations Conference*, American Institute of Aeronautics and Astronautics, June 2016.

[167] Paul, S., Hole, F., Zytek, A., and Varela, C. A., "Flight Trajectory Planning for Fixed-Wing Aircraft in Loss of Thrust Emergencies," *arXiv:1711.00716 [cs]*, Oct. 2017, Comment: This work was accepted as a full paper and presented in the Second International Conference on InfoSymbiotics / DDDAS (Dynamic Data Driven Applications Systems) held at MIT, Cambridge, Massachusetts in August, 2017.

[168] Ngatchou, P., Zarei, A., and El-Sharkawi, A., "Pareto Multi Objective Optimization," *Proceedings of the 13th International Conference on, Intelligent Systems Application to Power Systems*, Nov. 2005, pp. 84–91.

[169] Wickens, C. D., Hollands, J. G., Banbury, S., and Parasuraman, R., *Engineering Psychology and Human Performance*, Psychology Press, Aug. 2015.

[170] Isaac, A., Shorrock, S., Kennedy, R., Kirwan, B., Andersen, H., and Bove, T., *Technical review of human performance models and taxonomies of human error in ATM (HERA)*, HRS/HSP-002-REP-01, European Organisation for the Safety of Air Navigation, 2003.

[171] Wickens, C. D. and Flach, J. M., "5 - Information Processing," *Human Factors in Aviation*, edited by E. L. Wiener and D. C. Nagel, Cognition and Perception, Academic Press, San Diego, Jan. 1988, pp. 111–155.

[172] Docherty, B., *Losing humanity: The case against killer robots*, Human Rights Watch, 2012.

[173] Gutzwiller, R. S., Lange, D. S., Reeder, J., Morris, R. L., and Rodas, O., "Human-Computer Collaboration in Adaptive Supervisory Control and Function Allocation of Autonomous System Teams," *Virtual, Augmented and Mixed Reality*, edited by R. Shumaker and S. Lackey, Lecture Notes in Computer Science, Springer International Publishing, Cham, 2015, pp. 447–456.

[174] Castagno, J. and Atkins, E., "Roof Shape Classification from LiDAR and Satellite Image Data Fusion Using Supervised Learning," *Sensors*, Vol. 18, No. 11, Nov. 2018, pp. 3960.

[175] Castagno, J. and Atkins, E., "Map-Based Planning for Small Unmanned Aircraft Rooftop Landing," *Handbook on Reinforcement Learning and Control*, Springer, 2021.

[176] Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H., "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," *arXiv:1704.04861 [cs]*, April 2017, arXiv: 1704.04861.

[177] Ronneberger, O., Fischer, P., and Brox, T., "U-Net: Convolutional Networks for Biomedical Image Segmentation," *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, edited by N. Navab, J. Hornegger, W. M. Wells, and A. F. Frangi, Lecture Notes in Computer Science, Springer International Publishing, Cham, 2015, pp. 234–241.

[178] Epic Games, "Unreal Engine," April 2019.

[179] Shah, S., Dey, D., Lovett, C., and Kapoor, A., "AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles," *Field and Service Robotics*, edited by M. Hutter and R. Siegwart, Springer Proceedings in Advanced Robotics, Springer International Publishing, Cham, 2018, pp. 621–635.

[180] Dotenco, S., Gallwitz, F., and Angelopoulou, E., "Autonomous Approach and Landing for a Low-Cost Quadrotor Using Monocular Cameras," *Computer Vision - ECCV 2014 Workshops*, edited by L. Agapito, M. M. Bronstein, and C. Rother, Lecture Notes in Computer Science, Springer International Publishing, Cham, 2015, pp. 209–222.

[181] Schönberger, J. L. and Frahm, J., "Structure-from-Motion Revisited," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016, pp. 4104–4113, ISSN: 1063-6919.

[182] Whalley, M., Takahashi, M., Tsenkov, P., Schulein, G., and Goerzen, C., "Field-testing of a helicopter UAV obstacle field navigation and landing system," *65th Annual Forum of the American Helicopter Society, Grapevine, TX*, 2009.

[183] Wei, W., Shirinzadeh, B., Nowell, R., Ghafarian, M., Ammar, M. M. A., and Shen, T., "Enhancing Solid State LiDAR Mapping with a 2D Spinning LiDAR in Urban Scenario SLAM on Ground Vehicles," *Sensors*, Vol. 21, No. 5, Jan. 2021, pp. 1773, Number: 5 Publisher: Multidisciplinary Digital Publishing Institute.

[184] Nam, D. V. and Gon-Woo, K., "Solid-State LiDAR based-SLAM: A Concise Review and Application," *2021 IEEE International Conference on Big Data and Smart Computing (BigComp)*, Jan. 2021, pp. 302–305, ISSN: 2375-9356.

[185] Intel, *Intel® RealSense™ LiDAR Camera L515 Datasheet*, 6 2020, Rev. 002. [Online] Available: `https://dev.intelrealsense.com/docs/lidar-camera-l515-datasheet`.

[186] Shuo Yang, Jiahang Ying, Yang Lu, and Zexiang Li, "Precise quadrotor autonomous landing with SRUKF vision perception," *2015 IEEE International Conference on Robotics and Automation (ICRA)*, May 2015, pp. 2196–2201, ISSN: 1050-4729.

[187] Pluckter, K. and Scherer, S., "Precision UAV Landing in Unstructured Environments," *Proceedings of the 2018 International Symposium on Experimental Robotics*, edited by J. Xiao, T. Kröger, and O. Khatib, Springer Proceedings in Advanced Robotics, Springer International Publishing, Cham, 2020, pp. 177–187.

[188] Castagno, J. and Atkins, E., "Polylidar3D - Fast Polygon Extraction from 3D Data," *Sensors*, Vol. 20, No. 17, Jan. 2020, pp. 4819.

[189] Shelhamer, E., Long, J., and Darrell, T., "Fully Convolutional Networks for Semantic Segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 39, No. 4, April 2017, pp. 640–651, Conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence.

[190] Badrinarayanan, V., Kendall, A., and Cipolla, R., "SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 39, No. 12, Dec. 2017, pp. 2481–2495, Conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence.

[191] Kendall, A., Badrinarayanan, V., and Cipolla, R., "Bayesian SegNet: Model Uncertainty in Deep Convolutional Encoder-Decoder Architectures for Scene Understanding," *Procedings of the British Machine Vision Conference 2017*, British Machine Vision Association, London, UK, 2017, p. 57.

[192] Chen, L.-C., Papandreou, G., Schroff, F., and Adam, H., "Rethinking Atrous Convolution for Semantic Image Segmentation," *arXiv:1706.05587 [cs]*, Dec. 2017, arXiv: 1706.05587.

[193] Chen, L., Papandreou, G., Kokkinos, I., Murphy, K., and Yuille, A. L., "DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 40, No. 4, April 2018, pp. 834–848, Conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence.

[194] Siam, M., Gamal, M., Abdel-Razek, M., Yogamani, S., Jagersand, M., and Zhang, H., "A Comparative Study of Real-Time Semantic Segmentation for Autonomous Driving," *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, June 2018, pp. 700–70010, ISSN: 2160-7516.

[195] Zhang, X., Zhou, X., Lin, M., and Sun, J., "ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices," *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, June 2018, pp. 6848–6856, ISSN: 2575-7075.

[196] Castagno, J., "Github - Unreal Landing," [Online] Available: `https://github.com/JeremyBYU/UnrealRooftopLanding/`, 2021, Accessed: 2021-06-01.

[197] PolyPixel, "Unreal Marketplace - Urban City," [Online] Available: `https://www.unrealengine.com/marketplace/urban-city`, 2018, Accessed: 2017-01-05.

[198] Croissant, "Unreal Marketplace - Urban Rooftop," [Online] Available: `https://www.unrealengine.com/marketplace/rooftop-pack`, 2018, Accessed: 2017-01-05.

[199] Koenig, N. and Howard, A., "Design and use paradigms for Gazebo, an open-source multi-robot simulator," *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, Vol. 3, Sept. 2004, pp. 2149–2154 vol.3.

[200] Glennie, C. and Lichti, D. D., "Static Calibration and Analysis of the Velodyne HDL-64E S2 for High Accuracy Mobile Scanning," *Remote Sensing*, Vol. 2, No. 6, June 2010, pp. 1610–1624, Number: 6 Publisher: Molecular Diversity Preservation International.

[201] Ouster, *OS0 - Ultra-Wide View High-Resolution Imaging Lidar*, 2 2020, Rev. 02/23/20220 [Online] Available: `https://data.ouster.io/downloads/OS0-lidar-sensor-datasheet.pdf`.

[202] Mohamed, S. A. S., Haghbayan, M., Westerlund, T., Heikkonen, J., Tenhunen, H., and Plosila, J., "A Survey on Odometry for Autonomous Navigation Systems," *IEEE Access*, Vol. 7, 2019, pp. 97466–97486, Conference Name: IEEE Access.

[203] Zhang, B., Zhang, X., Wei, B., and Qi, C., "A Point Cloud Distortion Removing and Mapping Algorithm based on Lidar and IMU UKF Fusion," *2019 IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM)*, July 2019, pp. 966–971, ISSN: 2159-6255.

[204] Efraimidis, P. and Spirakis, P., "Weighted random sampling," *Encyclopedia of algorithms*, edited by M.-Y. Kao, Springer US, Boston, MA, 2008, pp. 1024–1027.

[205] Hoh, R. H., Baillie, S., Kereliuk, S., and Traybar, J. J., "Decision-Height Windows for Decelerating Approaches in Helicopters - Pilot/Vehicle Factors and Limitations," Tech. Rep. ADA239610, SYSTEMS CONTROL TECHNOLOGY INC ARLINGTON VA, April 1991, Section: Technical Reports.

[206] Lorensen, W. E. and Cline, H. E., "Marching Cubes: A High Resolution 3D Surface Construction Algorithm," *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, Association for Computing Machinery, New York, NY, USA, 1987, pp. 163–169.

[207] Romano, M., Kuevor, P., Lukacs, D., Marshall, O., Stevens, M., Rastgoftar, H., Cutler, J., and Atkins, E., "Experimental Evaluation of Continuum Deformation with a Five Quadrotor Team," *2019 American Control Conference (ACC)*, July 2019, pp. 2023–2029.

[208] Continental, "Github - Enhanced Communcaton and Abstraction Library," [Online] Available: `https://github.com/continental/ecal`, 2021, Accessed: 2021-06-05.

[209] Castagno, J., "Github - RealSense Sensor Package," [Online] Available: `https://github.com/JeremyBYU/realsense-tracking`, 2020, Accessed: 2020-06-05.

[210] Zhang, Z. and Scaramuzza, D., "A Tutorial on Quantitative Trajectory Evaluation for Visual(-Inertial) Odometry," *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Oct. 2018, pp. 7244–7251.

[211] Intel, "Github - Intel RealSense T265 Tracking Camera Documentation," [Online] Available: `https://github.com/IntelRealSense/librealsense/blob/master/doc/t265.md`, 2020, Accessed: 2021-06-05.

[212] Liu, J., Ni, B., Li, C., Yang, J., and Tian, Q., "Dynamic Points Agglomeration for Hierarchical Point Sets Learning," *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, Oct. 2019, pp. 7545–7554, ISSN: 2380-7504.

[213] Guo, Y., Wang, H., Hu, Q., Liu, H., Liu, L., and Bennamoun, M., "Deep Learning for 3D Point Clouds: A Survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2020, pp. 1–1, Conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence.

[214] Möls, H., Li, K., and Hanebeck, U. D., "Highly Parallelizable Plane Extraction for Organized Point Clouds Using Spherical Convex Hulls," *2020 IEEE International Conference on Robotics and Automation (ICRA)*, May 2020, pp. 7920–7926, ISSN: 2577-087X.

[215] Molano, R., Rodríguez, P. G., Caro, A., and Durán, M. L., "Finding the largest area rectangle of arbitrary orientation in a closed contour," *Applied Mathematics and Computation*, Vol. 218, No. 19, June 2012, pp. 9866–9874.

[216] Ongaro, D. and Ousterhout, J., "In Search of an Understandable Consensus Algorithm," *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, USENIX Association, Philadelphia, PA, June 2014, pp. 305–319.

[217] Cohen, I. G., Gostin, L. O., and Weitzner, D. J., "Digital Smartphone Tracking for COVID-19: Public Health and Civil Liberties in Tension," *JAMA*, Vol. 323, No. 23, June 2020, pp. 2371–2372.

[218] Lee, U. and Kim, A., "Benefits of Mobile Contact Tracing on COVID-19: Tracing Capacity Perspectives," *Frontiers in Public Health*, Vol. 9, March 2021.

[219] Chitanvis, R., Ravi, N., Zantye, T., and El-Sharkawy, M., "Collision avoidance and Drone surveillance using Thread protocol in V2V and V2I communications," *2019 IEEE National Aerospace and Electronics Conference (NAECON)*, July 2019, pp. 406–411, ISSN: 2379-2027.

[220] Saltzer, J. H., "The Origin of the "MIT License"," *IEEE Annals of the History of Computing*, Vol. 42, No. 4, 2020, pp. 94–98, Publisher: IEEE Computer Society.