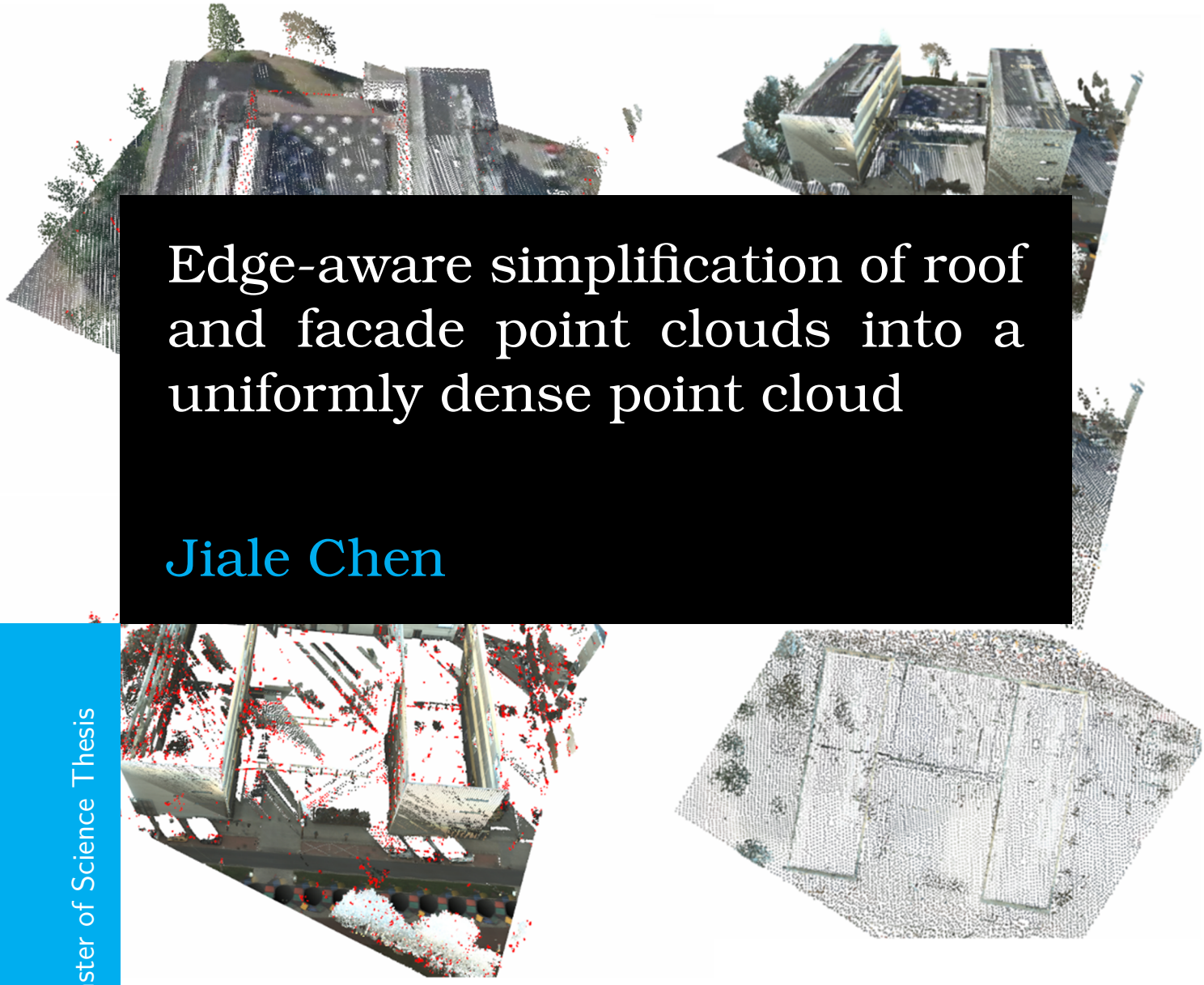
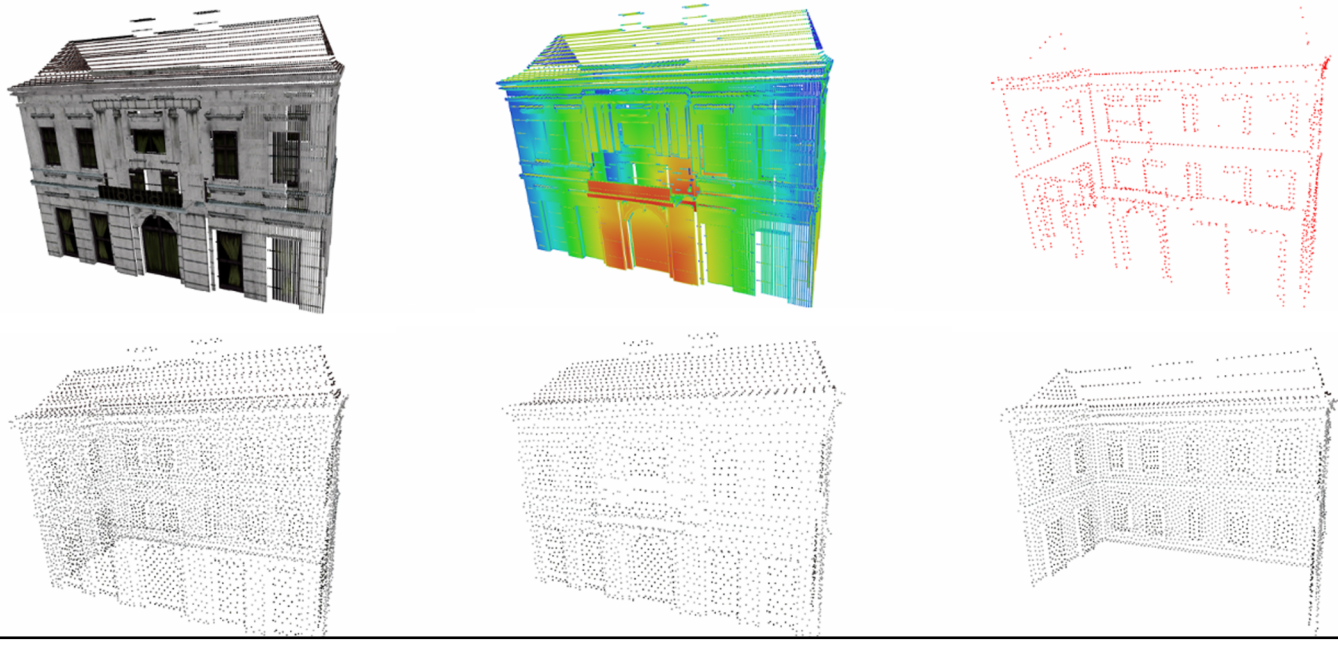


Edge-aware simplification of roof and facade point clouds into a uniformly dense point cloud

Jiale Chen

Master of Science Thesis



TU Delft Delft University of Technology

Architecture and the Built Environment (BK)

Edge-aware simplification of roof and facade point clouds into a uniformly dense point cloud

Master of Science Thesis

For the degree of Master of Science in Geomatics at Delft University of Technology

Jiale Chen

June 18, 2015

Faculty of Architecture and the Built Environment (BK) · Delft University of Technology

The work in this thesis was supported by the company Cyclomedia. Their cooperation is hereby gratefully acknowledged.

Abstract

Point clouds can be obtained by airborne or terrestrial Light Detection and Ranging (LiDAR) scanning or directly from street-view panoramic imagery. Point clouds obtained from airborne scanners cover most roof information but lack facade information. Similar but orthogonal to airborne scanning, street-view point clouds cover well of the facade information but lack roof points. Therefore, it is necessary to combine those point clouds in order to construct a complete building model. For this combination, simplification is needed in which edge preservation, outlier removal, noise smoothing and uniform density have to be considered as these properties are desired in many post-processing applications.

In this thesis, an algorithm pipeline is proposed that is able to take in an outlier-ridden, noisy and non-uniform roof and facade point clouds and generate an outlier-removed, edge-aware and uniform point cloud. The algorithm pipeline can be divided into two independent phases: outlier removal and simplification. The outlier removal algorithm can remove singly scattered and small cluster of outliers, whereas the simplification algorithm pipeline is able to generate a noise-reduced, uniform and edge-aware point cloud. The pipeline is validated to be able to achieve the objectives. Proof of efficiency in running-time is given, so that it can be used for processing large-size real-scene point clouds.

Table of Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my supervisors Ass.Prof.Dr.Sisi Zlatanova , Prof.Dr.Elmar Eisemann, Dr.Bert Buchholz for their guidance and support during the writing of this thesis. And I'm also very appreciate about the valuable opinions from Dr.Ir.Ben Gorteand other researchers who have given me suggestions. What is more, many thanks to company Cyclomedia for their support of the data and guidance as well.

For two years of MSc study in Netherlands, I have met really a lot of friends coming from many countries around the world. Especially my dear classmates in Geomatics, we've had a lot of fun together and I've learned really a lot from you during these two years. I also need to thank my good friend Dimitris Zervakis for his assistance in language check in the thesis.

I would also like to thank my family who always give me endless care and support when I stay abroad.

Delft, University of Technology Jiale Chen
June 18, 2015

Chapter 1

Introduction

Point clouds are an effective and popular representation of real world geographical information. Highly dense point clouds have been widely used in applications such as making digital elevation models of the terrain [1], reverse engineering of industrial sites [2], tree reconstruction [3] and creating 3D models of urban environment [4].

Point clouds are usually created from 3D optical scanners. LiDAR, an acronym for Light Detection and Ranging, has been popularly used to make high-resolution maps in a wide range of research fields [5]. When the laser scanners are deployed on an aircraft such as helicopter or UAV (Unmanned Aerial Vehicle), it is called airborne LiDAR. When the laser scanners are mounted on a moving car, van or boat the setup is usually referred to as terrestrial Mobile Mapping System. Traditionally, both airborne and terrestrial Mobile Mapping Systems are equipped with GNSS receiver, Inertial Measurement Unit and laser scanners [6]. Cameras may also be mounted in the systems but mainly only provide extra color information for LiDAR points.

Recently, due to the improved accuracy of the image capturing and matching techniques, new methods have emerged to retrieve point clouds directly by using only panoramic photographs. Compared to mobile LiDAR mapping systems, point clouds derived directly from images add extra color information and the scanning process is much faster and cheaper. Airborne scanning covers well the roof seen from above while street-view scanning covers the facade seen from ground. This research proposes a method for combining roof and facade point clouds.

This chapter starts by giving an introduction and research motivation in Section 1-1. Section 1-2 gives the research question, main objective and scope. Finally the overview of algorithms and the thesis structure follows in Section 1-3. Concluding, a summary of this thesis' contributions is provided in Section 1-4.

1-1 Motivation

In the current research, we consider the source of facade point cloud coming from either terrestrial LiDAR or panoramic images. Terrestrial LiDAR can produce a relatively uniform and accurate point cloud which can be very dense. As for point clouds generated from panoramic images, many approaches already exist for creating street level 360 degree spherical panoramic imagery with accurate positioning by using algorithms such as Structure from Motion (SfM) [7], Bundle Adjustment (BA) [8] and Iterative Closest Point (ICP) [9]. Based on the consolidated images, current researches are focusing on manufacturing a pipeline that is able to derive color point clouds directly from those street-view panoramic images, where both geometry and color information can be obtained. Point clouds obtained from photographs are less precise than terrestrial LiDAR point cloud. Raw point clouds without any processing are usually not clean contaminated with a lot of outliers, noise, redundancy and holes, which create problems for storage, surface reconstruction, visualization and analysis. A common problem for street-view point clouds is missing information from above, such as the roofs, which is necessary for a complete 3D building model.

The same type of problem but orthogonal to street-view point clouds is the fact that roof point clouds obtained by airborne laser scanning lack information of building facades. One famous airborne point cloud in Netherlands is the AHN2, which means "National Height model of the Netherlands version 2" in Dutch. It provides relatively uniform dense airborne LiDAR point clouds covering most areas of the country. Except for the common problem of lacking facade information, another limitation is that it does not contain color which is necessary for visualization and useful for other related analysis such as color-based segmentation.

A complete building model should be the combination of roof and facade point clouds. For both point clouds, outliers need to be removed respectively before merging since outliers are produced in the generation phase. Singly scattered points and points in small clusters are considered as outliers.

In point cloud simplification, edges indicate the geometry skeleton of the point cloud which is useful for applications such as point cloud surface reconstruction [10] and segmentation [2]. A uniform sampling distribution is desired in applications such as pyramid algorithms for multilevel smoothing [11] and texture synthesis [12]. Therefore the combined point cloud should be outlier removed, noise reduced, uniformly dense and edge-aware.

However, the integration of these data is not just as simple as transforming and registering them in the same coordinate system. Different point clouds derived from different methods expose the problems of large density variance, noise distribution difference and data overlapping. How to keep the edge features while at the same time remove outliers, smooth noise and reduce redundancy have been popular research topics recently. In outlier removal, none of the algorithms is able to remove both singly scattered outliers and small cluster of outliers without over-removing. In simplification, methods proposed by other researches focus on different properties separately. Some of them focus more on generating a uniform result and some focus on smoothing the noise and keeping edge points. Many of the algorithms are designed for processing small-size point clouds, but are slow and memory-inefficient to process large-size point cloud. None of the existing methods considers the point cloud spatial distribution from the data source perspective.

1-2 Research question, objective and scope

Based on the motivation mentioned above, the main research question of this thesis is as below:

- *Which algorithms are most appropriate to fuse roof and facade point clouds into an edge-aware and uniformly dense color point cloud?*

In order to answer the research question, the main objective of this thesis is:

- *Design and implement an algorithm that is able to efficiently integrate roof point clouds derived from airborne LiDAR with facade point clouds derived from street-view terrestrial LiDAR or panoramic images to create an edge-aware and uniformly dense color point cloud.*

The scope of this research includes designing two subsequent and independent algorithms, i.e. outlier removal and simplification, and how to integrate them into one unified algorithm pipeline. Our researched point clouds are derived from LiDAR scanning or panoramic imagery.

1-3 Overview

Our algorithm pipeline can take in outlier-ridden, noisy and non-uniform roof and facade point clouds and combine into an outlier-removed, noise-reduced, uniform and edge-aware point cloud. In general, the overview of our algorithm pipeline is mainly separated into two phases as shown in Figure 1-1. The first phase removes outliers in the input point clouds. The second phase simplifies the point cloud. The advantages of this approach is that uniform density, edge-preservation and noise-smoothing are considered at the same time with consideration of point cloud collection methods. What is more, this algorithm pipeline can be applied in large-scale processing. The detailed expansion of this pipeline is shown in Figure 5-1 and discussed in Chapter 3, Chapter 4 and Chapter 5.

The structure of this thesis is organized as follows: Chapter 2 first discusses the point clouds we use in this thesis and proposes our precision estimation models. Then, several popular outlier removal algorithms are discussed and analyzed. At the end of this chapter, we list several popular simplification algorithms and compare and discuss the efficiency and suitability for our use. Chapter 3 introduces our proposed outlier removal algorithm based on two existing ones. Our method can be applied to remove outliers in any point cloud. Chapter 4 is the key research of this thesis. We propose several subsequent algorithms that is able to take in several outlier-cleaned point clouds and simplify into an edge-aware, noise smoothed and uniformly-dense point cloud. Chapter 5 discusses the integration of outlier removal and simplification algorithms and gives our final complete and detailed algorithm pipeline. Later in this chapter we give our implementation details about different point clouds preparation, software prototypes and parameter tweaking strategies. Chapter 6 gives the simplified results prepared in Chapter 5 and validates them by checking with our objective list qualitatively or quantitatively. The limitations of our proposed method are discussed afterwards. Chapter 7 concludes this thesis and gives recommendations for future improvement.

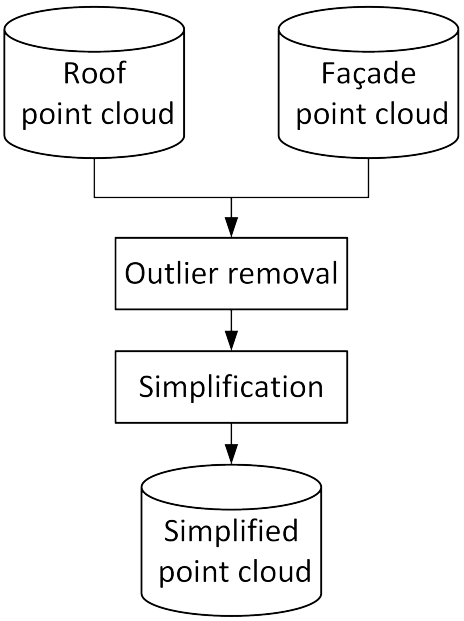


Figure 1-1: Overview of our algorithm pipeline.

1-4 Contributions

In this thesis, We propose a new outlier removal algorithm by integrating and improving two existing ones. Similar to other methods, our outlier removal algorithm removes outliers in a separate pre-processing step. But different from others, as far as we know, that none of which can remove both singly scattered outliers and small cluster of outliers without over-removing artifacts.

In simplification, our algorithm can be applied to process large-size point cloud in a short running time. Compared to other algorithms focusing on one or two objectives of noise-smoothing, edge preservation or uniform density, our proposed algorithm can achieve these properties at the same time. What is more, none of other simplification algorithms considers the point cloud precision distribution from a data collection perspective. Our method estimates precision according to scanning methods and attaches an importance value to each point so that a point with a low importance value has a higher probability to be smoothed out in simplification.

We finally integrate the proposed outlier removal and simplification algorithms into one unified algorithm pipeline that is able to take in several outlier-ridden, noisy, non-uniform point clouds and generate an outlier-cleaned, noise reduced, uniform and edge-aware point cloud.

Chapter 2

Background

According to the research objectives, this chapter first compares and discusses the spatial properties of different point clouds we use in this research. Then we give an overview about the current point cloud algorithm researches in outlier removal and simplification. For both outlier removal and simplification, listed methods are discussed with respect to efficiency and suitability for our use.

2-1 Point clouds spatial property and precision estimation

The spatial property of point cloud is determined by different acquisition methods. In this section we first give an overview about acquisition methods for airborne LiDAR, terrestrial LiDAR and street-view panoramic imagery point cloud. Then a rough and easy-to-implement precision estimation model has been put forward to compute the importance value used in simplification. Next we explore the possibility to get a mathematical rigorous precision estimation model based on error-source analysis. Lastly we make a short conclusion of our proposed precision estimation model.

2-1-1 Overview of point clouds acquisition methods

In this subsection different point cloud acquisition methods are briefly summarized.

Airborne LiDAR point clouds

The airborne LiDAR systems use laser scanners fixed on the bottom of the aircraft by rotating the beam mirror to scan the target objects. There are two types of beam mirrors available to choose in production, i.e. oscillating mirror or rotating mirror. Both scanners reflect laser beam emitted from the sensor. With fast rotating speed of the mirror, laser rays can hit a wide angle range of object surface. The position of the scanner is from a combination of GPS

position parameters (i.e. x, y and z) and INS rotation parameters (i.e. ω, κ and ϕ). The final object position is derived by combining scanner position, scan distance and attitude of each laser ray. An overview of LiDAR point collection process is shown in Figure 2-1.

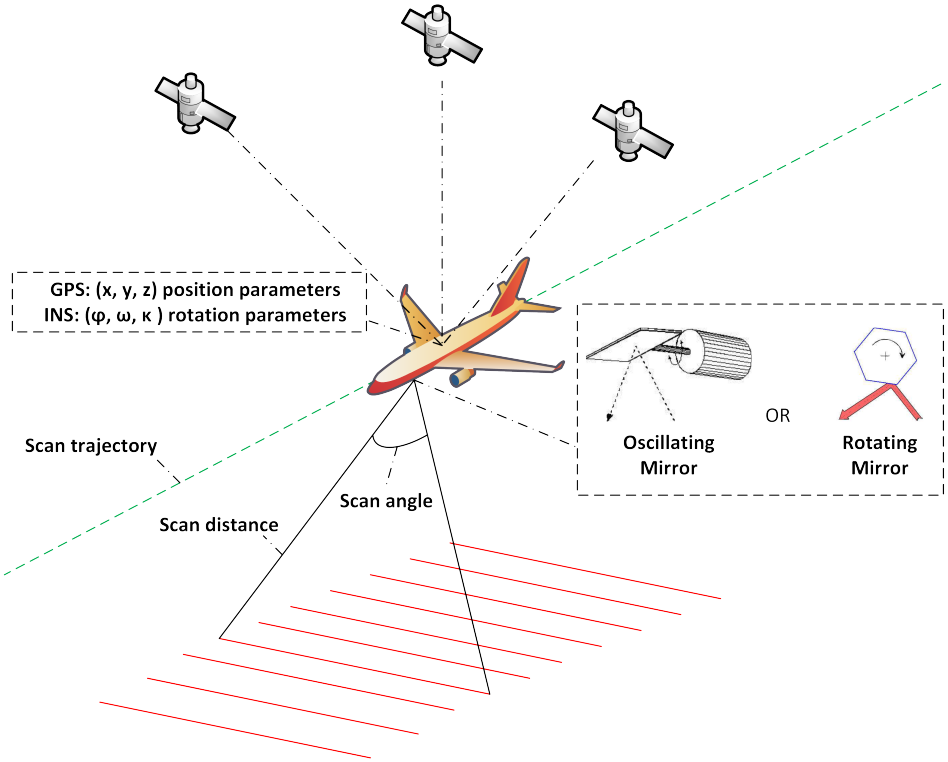


Figure 2-1: Airborne LiDAR scanning procedure. The position of each point is decided by two factors: (a).GPS and INS on the plane; (b).distance and attitude from scanner to the point.

Terrestrial LiDAR point clouds

Similar to airborne LiDAR systems, terrestrial LiDAR systems also use line scanners but scanned with both horizontal and vertical directions. According to the position of the scanners terrestrial LiDAR systems can be further divided into mobile and statically deployed LiDAR, in which statically deployed LiDAR has a higher precision. One example showing both laser scanners is shown in Figure 2-2.

Panoramic image point clouds

The acquisition of street-view point clouds from panoramic imagery is more complicated compared to LiDAR point cloud. General pipeline that is adopted by many vendors in producing point clouds from imagery is shown in Figure 2-3.

The feature points matched from pairs of stereo images are extracted by algorithms such as SIFT [40] and FAST [41]. Taking the matched feature points as control points, bundle adjustment is performed to find point positions and camera correction parameters that minimize the reprojection errors, i.e. the sum of least square distances from the projections of each

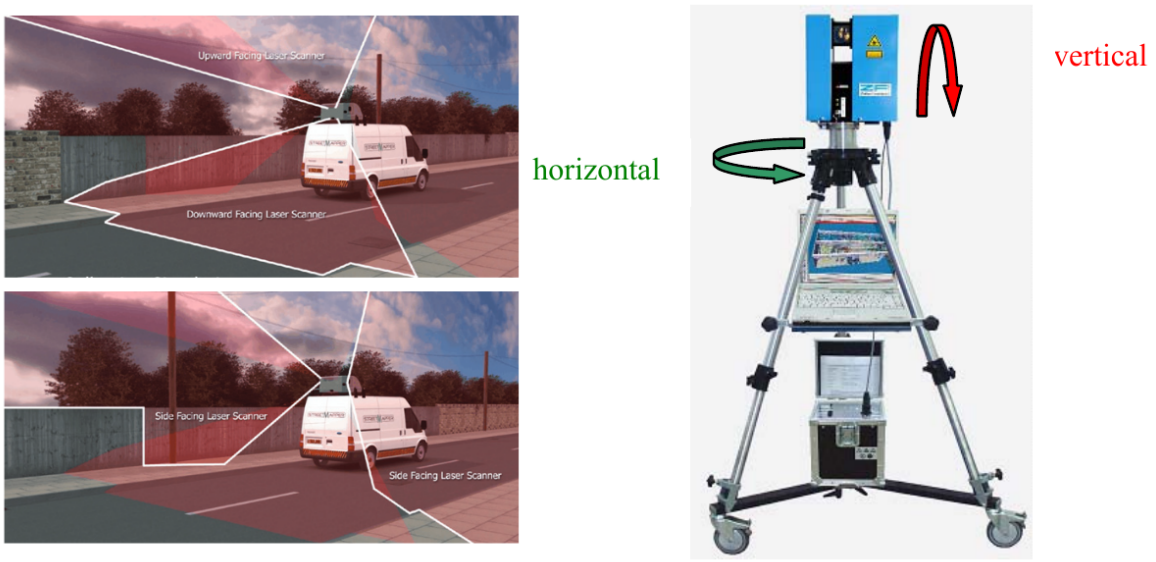


Figure 2-2: Left: terrestrial mobile mapping system whose position accuracy is determined by scanner position, distance and attitude from scanner to the point; Right: terrestrial static LiDAR scanner whose position is only decided by distance and attitude. Figure from Haala et al. [38] and Frohlich et al. [39].

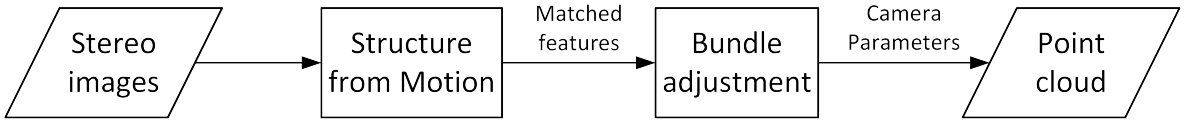


Figure 2-3: Street-view point clouds generation pipeline.

track to its corresponding image feature points [8]. Therefore each pixel can be calibrated and correlated to one point in the object coordinate system and thus point cloud from pixels can be derived.

2-1-2 Rough precision estimation model for all point clouds

Given an input point cloud with only position and color information, we propose a rough and easy-to-implement precision estimation model that is applicable for all point clouds discussed in our research.

LiDAR point cloud

Considering scanner position unknown for every point, the point positions of both airborne and terrestrial LiDAR point clouds are mainly determined by distance measurement. So the precision of each point is mainly determined by distance to the scanner. The distance accuracy mainly depends on the reflected laser intensity therefore directly determined by surface reflectivity. One step further, the surface reflectivity depends on the incidence angle and object material. To conclude the reliance propagation chain, the precision of each point

is determined by incidence angle. As shown in Figure 2-4, points with larger incidence angle from the scanner have lower precision. Given a uniform scanning angle interval, points with larger incidence angles are usually more dense than others. All in all, the density of point in local neighborhood can be an indicator about the precision.

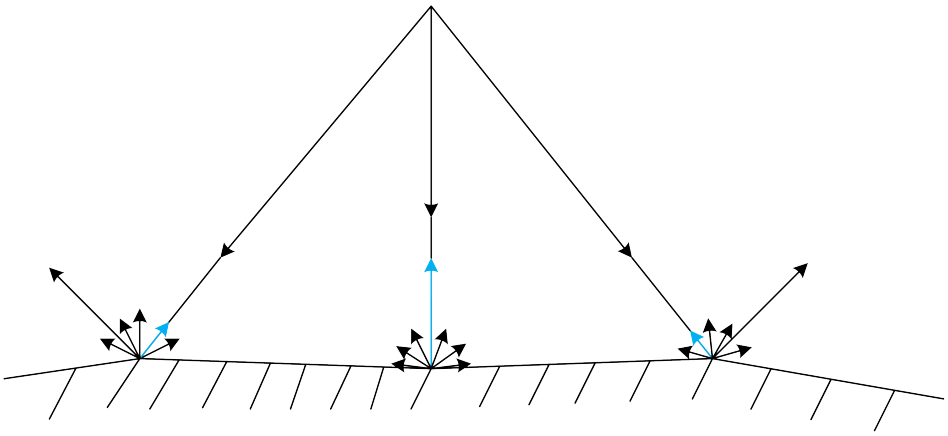


Figure 2-4: The precision of LiDAR points is mainly determined by incidence angle where a larger angle receives less reflection.

Panoramic imagery point cloud

Panoramic camera can be regarded as a point scanner passively receiving rays from the environment. We simulated the scanning by setting one airborne LiDAR scanner and one street-view camera in the scene.

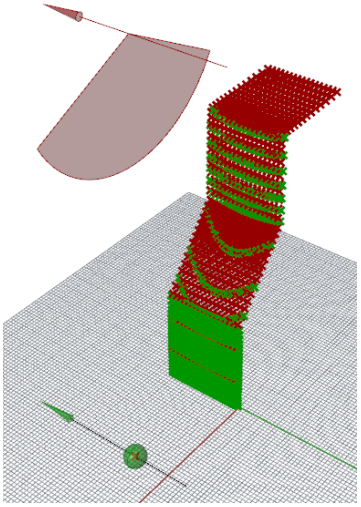


Figure 2-5: Scanning simulation. Airborne LiDAR point clouds are scanned by line scanner and colored in red in the figure; street-view point clouds can be regarded taken from a point scanner and colored in green in the figure.

As we can see from the scanning simulation in Figure 2-5. Similar to LiDAR point cloud,

the density of each point in the imagery point cloud is able to indicate the precision of the point. The points on sparse area have larger distortion and lower resolution compared to other points.

Density estimation operator

To conclude for both LiDAR and panoramic imagery point clouds, the density of each point in local area can roughly and qualitatively indicate the precision. For quantitative estimation, we choose the local density of each point as the metric for the relative accuracy of each point. Since in both LiDAR and panoramic imagery point clouds, all points are scanned from manifold surface. In our research we apply the surface density formula below:

$$\sigma_i = \frac{N}{\pi D^2} \tag{2-1}$$

where N is the number of neighbors and D is the search distance. In order to avoid estimation error for points located in sparse area, we use KNN search instead of radius search. N is replaced by neighbor search size K and D is the $K-th$ nearest neighbor distance. In order to scale the range of the importance value properly, we map the density value into $[0, 1]$ which is our final importance value by the formula below:

$$Iv_i = e^{-\frac{\bar{\sigma}}{\sigma_i}} \tag{2-2}$$

Figure 2-6 is one example we get from one simulated scan by method in Figure 2-5. From the result it can be seen that especially in overlapping area between two point clouds, the importance value well distinguishes two point clouds.

Potential threat in density estimation

We illustrate the potential threat to users who might need to deal with large scale processing using our importance operator. To guarantee a satisfactory coverage of point clouds, airborne LiDAR scanning always adopts strategies similar to aerial images by a certain amount of overlapping between neighboring scan strips. One such case can be seen from Figure 2-7 scanned in Zeeland province in Netherlands. It can seen that both strips have about 1/3 common area overlapped with each other.

If the processing is in city scale level, this artifact cannot be neglected in a rigorous model. The result will contain some dense area which is cause by overlapping instead of scanning. One such overlapping example is made by simply merging point clouds from different scan lines shown in Figure 2-8. We can see that points in overlapping area can even be more dense than those directly under one scanning line.

This artifact can be rectified by classifying points one scan id per scan line if it is known a priori. For example, in one famous LiDAR document *.LAS* file, this information is recorded in "Point Source ID" attribute documenting the source file of each point, i.e. which scan line it is generated.[43]. When this information is not documented, this artifact can only be eased in two cases: (a). when the simplification processing scale is smaller than per scan coverage width; (b). when post-processing algorithms such as ICP remove overlapping points. In this thesis all demos' sizes are smaller than per-scan coverage width so the artifacts can be ignored.

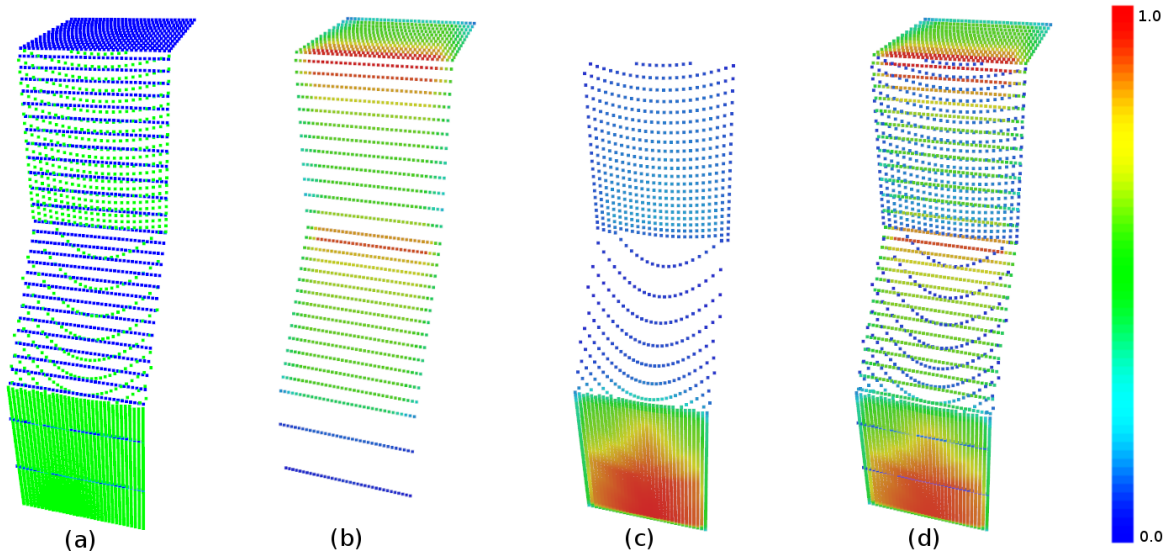


Figure 2-6: Importance value of each point in the point clouds. From left to right: (a). Original simulated point clouds with airborne point clouds colored in blue and street-view point clouds in green; (b). Density weight of airborne LiDAR point clouds; (c). Density weight of street-view point clouds; (d). Combined final importance value of the input point clouds.

2-1-3 Rigorous precision estimation model

The density of each point is a rough indicator about the precision. In the error propagation theory, a mathematical rigorous precision estimation should take all error sources into consideration. The result could be equations similar to the one below (suppose different error factors $\sigma_1, \sigma_2, ..., \sigma_n$ have no correlation and final error σ_{s_i} is the linear combination of all errors):

$$\sigma_{s_i}^2 = a_1\sigma_1^2 + a_2\sigma_2^2 + ... + a_n\sigma_n^2 \tag{2-3}$$

We explore the possibility of getting a mathematical rigorous precision estimation model for different point clouds we analyze. The noise can be well estimated if the precision of each point is known a priori. The result will be a precise uncertainty map depicting noise distribution of the point clouds.

Error sources of airborne and terrestrial LiDAR point clouds

From the LiDAR scanning procedures for both airborne and terrestrial point clouds, we list a table of error sources influencing the precision of scanned LiDAR points. The errors are divided into three main categories.

As can be seen from the analysis, it is difficult or even impossible to derive precision in most items. As far as we know, in current methods accuracy estimation is usually performed in post-processing without considering error sources from raw scan. Van der Sande et al. estimate the accuracy AHN2 point clouds by fitting points to local planes [42]. We conclude

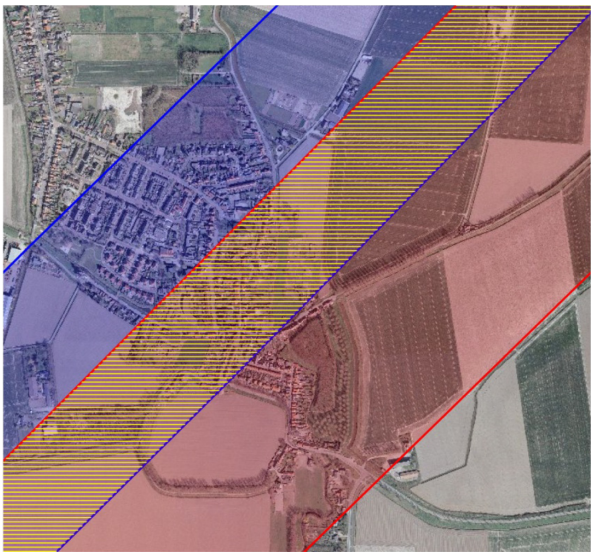


Figure 2-7: Overlapping neighbor strips scanned in Zeeland in 2007. The overlapping area between blue and red strips is colored in yellow. Figure from Van der Sande et al. [42]

that in current research it is not possible to estimate a rigorous uncertainty map from raw scan for LiDAR point clouds.

Error sources of panoramic imagery point cloud

Similar to LiDAR point clouds, many complicate cases generate errors that are not easily detected by normal geometry-based algorithms. Similar to Table 2-1 we list the main error sources in the street-view point clouds in four categories with respect to the steps in production pipeline.

As explained by Jalobeanu et.al. [44], both matching and modeling errors make it impossible to derive reliable uncertainty map for stereo imagery point clouds. In the table it can be seen that many error items exist and only bundle adjustment can be roughly estimated.

2-1-4 Conclusions for precision estimation

From the discussion in this section, we conclude that as lacking of thorough and rigorous error metric in current research, it is not possible to consider all the errors generated from source quantitatively. In our research, we put forward a rough and qualitative estimation model that can be applied in all point clouds. And we made a non-exhaustive list of rigorous error sources for both LiDAR and panoramic imagery point clouds. It is not implemented in this thesis but left as an open research question. A rigorous precision metric for uncertainty map need to be further refined in the future research.

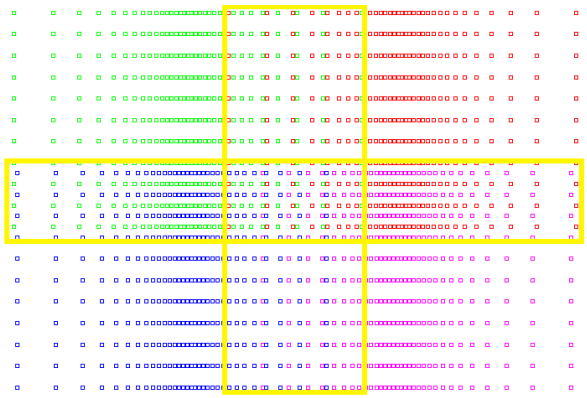


Figure 2-8: Simulated LiDAR scanning with merged points from different scan lines. Points are derived from four individual scans colored differently. Notice that points in the yellow rectangular region also can have falsely high importance value.

Category	Error source	Quantitative error estimation for each point
Scanner position	GPS position parameters; INS attitude parameters (not for static terrestrial systems)	Difficult to obtain for each scan
Coordinate difference from scanner to point	Incidence angle of the laser ray	Possible to get if recorded for each point in scanning.
Other factors	Laser reflection and refraction errors	In current methods not possible to derive

Table 2-1: Error sources of airborne and terrestrial LiDAR point cloud

2-2 Outlier removal

In point cloud processing, outliers may be generated from scanning, object reflection, preprocessing algorithms and so on. In this research, the purpose of outlier removal is to identify and remove outliers efficiently in large scale point clouds. In order to explain the chosen methods, the following two definitions are used throughout this section hence defined below:

Definition 1 (K-Nearest Neighbor (KNN) distance)**.** *Given a point cloud set* $P = \{p_1, p_2, ..., p_n\}$. *Let* N_p *be the set of the k-nearest neighbors of point* p_i *(excluding* p_i*). The KNN distance of a point* p_i *and its k-nearest neighbors. More specifically, let the distance between point* p_i *and* p_j *be defined as* $d(p_i, p_j) \geq 0$. *The KNN distance of a point is:*

$$\overline{d_{p_i}} = \frac{1}{k} \sum_{p_j \in N_p} d(p_i, p_j) \tag{2-4}$$

Category	Error source	Quantitative error estimation for each point
Camera position	GPS position parameters; INS rotation parameters	Difficult to obtain for each scan position
Feature points extraction	Object radiometric changes; non-uniform motion blur; non Gaussian-distributed camera structured noise etc.	Currently impossible to estimate precision for all those aspects.
Bundle adjustment	Difference between point positions before and after adjustment.	Possible to derive precision for each point.
Other error sources	Reflection and refraction of material such as glass.	In current methods not possible to derive.

Table 2-2: Error sources of street-view point cloud

Definition 2 (KNN inner distance)**.** *The K-Nearest Neighbor inner distance of a point p_i is the average KNN distance among points in N_p. Formally defined as:*

$$\overline{D_{p_i}} = \frac{1}{k(k-1)} \sum_{p_i, p_j \in N_p, i \neq j} d(p_i, p_j) \tag{2-5}$$

Three outlier removal methods are introduced in this section.

2-2-1 Local Distance-based Outlier Detection

Zhang et al [14] proposed the Local Distance-based Outlier Factor (LDOF) that is easy to set parameters. It is formally defined as follows:

Definition 3 (LDOF)**.** *The LDOF of a point p_i in point set $P = \{p_1, p_2, ..., p_n\}$ is the ratio between its KNN inner distance:*

$$LDOF_{p_i} = \frac{\overline{d_{p_i}}}{\overline{D_{p_i}}} \tag{2-6}$$

LDOF indicates the relative deviation of a point to its neighbors. To detect the outliers, the points are sorted in descending order according to their LDOF value and these with the highest LDOF values are the outliers. The overall running time complexity as claimed in the paper is $O(n \log n)$. An intuitive graphical explanation of LDOF is in Figure 2-9:

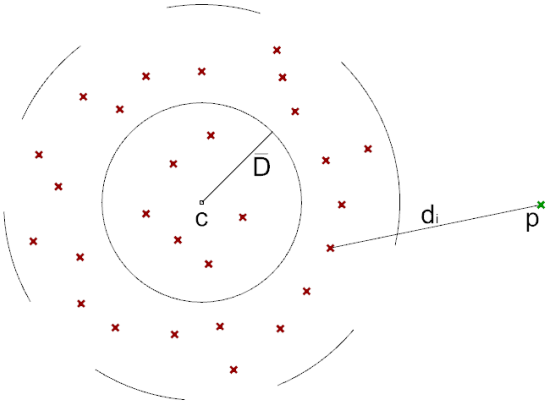


Figure 2-9: Local Distance-based Outlier Detection. c is the center of p 's neighbors. The dashed circle encloses k-nearest neighbors of p where each has distance d_i to p. The solid circle is the KNN inner distance circle of the point set with radius D. The ratio between those two values is the LDOF. In this case p is an outlier with LDOF $\gg 1$.

2-2-2 Outlier removal based on Distance-based Deviation Factor (DDF)

Based on the definitions of KNN distance and KNN inner distance, Wang et al. proposed a method to remove both sparse and dense outliers [13]. For sparse outliers, DDF is defined as:

$$DDF_{p_i} = |\frac{\overline{D_{p_i}} - \overline{d_{p_i}}}{\overline{D_{p_i}}}| \tag{2-7}$$

where $\overline{d_{p_i}}$ is the KNN distance and $\overline{D_{p_i}}$ is the KNN inner distance of point p_i. Assuming that pairwise distances at a small enough scale follow a single distribution, an outlier is classified based on the criterion below where θ is a threshold and $\sigma_{DDF_{p_i}}$ is the normalized standard deviation of DDF:

$$p_i = \begin{cases} \theta_{p_i} = \frac{\overline{DDF_{p_i}}}{\sigma_{DDF_{p_i}}} > \theta & p_i \text{ is an outlier} \\ \theta_{p_i} = \frac{\overline{DDF_{p_i}}}{\sigma_{DDF_{p_i}}} \leq \theta & p_i \text{ is an inlier} \end{cases} \tag{2-8}$$

Sparse outliers are removed using the classifier above. Small clusters of outliers may still survive after applying this algorithm. Thus a clustering-based method is proposed to remove small cluster of outliers by creating k-nearest neighbor graph for the point clouds and clustering points by region-growing. The clusters containing the lowest number of points are detected as outliers (See Figure 2-10).

2-2-3 Outlier removal by Nearest Neighbor Reciprocity (NNR) criterion

Weyrich et al. have proposed three novel methods to detect outliers including the plane fitting criterion, miniball criterion and nearest-neighbor reciprocity criterion [15]. The final outlier classifier is the linear combination of the criterion with user-defined weights. As shown by the experiment of the authors, the nearest-neighbor reciprocity criterion has the best result.

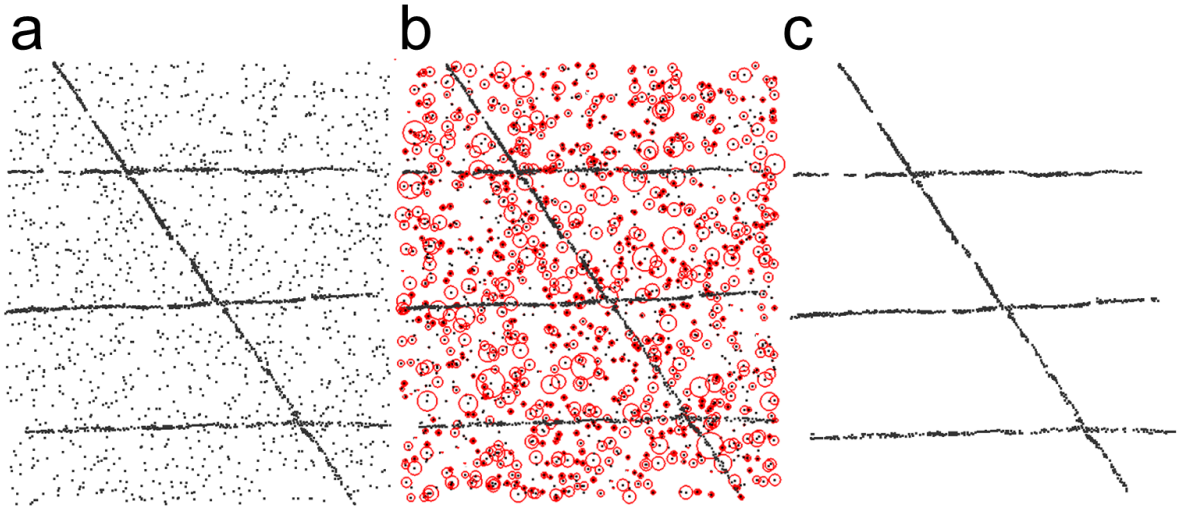


Figure 2-10: 2D illustration of outlier detection. (a) Original points with outliers; (b) Outliers associated with red circles whose radii represents each DDF value; (c) cleaned result. Figure from Wang et al.[13]

Furthermore in our research the plane fitting criterion smooth the edges while the mini-ball criterion is distance-based similar to the LDOF and DDF methods mentioned above. Therefore we do not explain these two methods in detail.

The central idea of the NNR criterion is that an inlier q may be in the set of the nearest neighbor set of an outlier p but p may not be in the nearest neighbor set of q. The uni-directional neighbors is defined as $N_{uni}(p) = \{q|q \in N_p, p \notin N_q\}$. while the bi-directional neighbors as $N_{bi}(p) = \{q|q \in N_p, p \in N_q\}$.

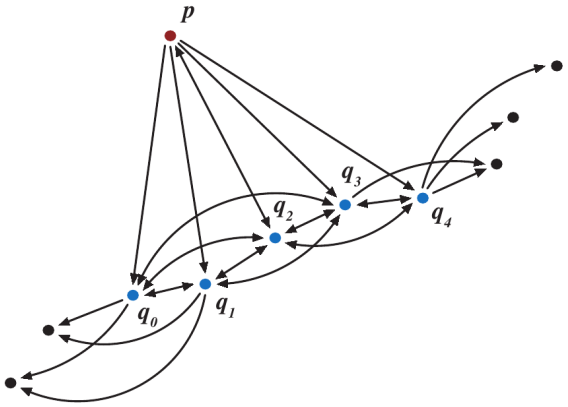


Figure 2-11: K-nearest neighbor graph of p. Note that in 5-NN relationship p only has one bi-directional neighbor q_2. The others are all uni-directional neighbors of p. Figure from Weyrich et al. [15]

The criterion considers the ratio between uni-directional and bi-directional neighbors of each point. The outlier classifier is defined as follows:

$$\chi(p) = \frac{|N_{uni}(p)|}{|N_{uni}(p)| + |N_{bi}(p)|} = \frac{|N_{uni}(p)|}{k} \tag{2-9}$$

For small cluster of outliers, they proposed to discard first l neighbors in KNN search but use $(l+1)st$ to $(l+k)th$ neighbor instead. Similar to LDOF, outliers are points with the highest classifier values.

2-2-4 Discussion of existing point cloud outlier removal algorithms

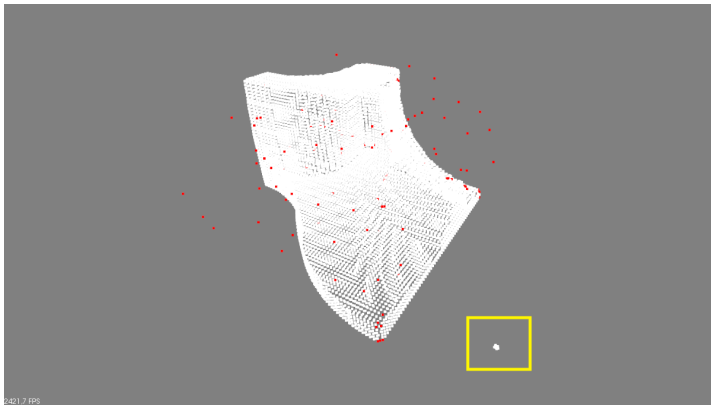


Figure 2-12: Poor performance of LDOF criterion on small cluster of outliers. Small cluster of outliers are not able to be detected . Note that outliers inside the yellow rectangle in the figure are labeled as inliers.

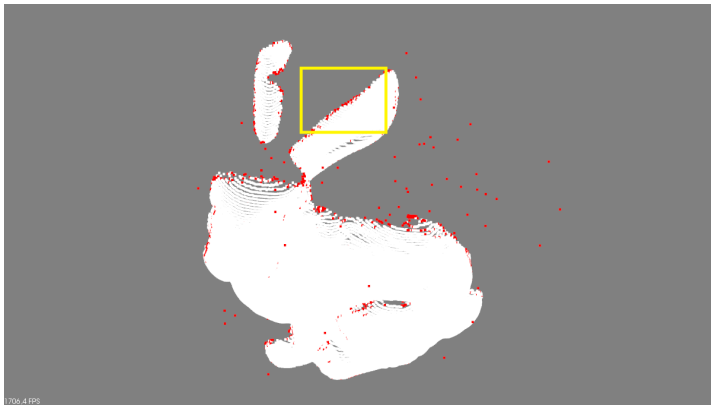


Figure 2-13: Poor performance of NNR criterion on points near manifold edges. Red points are labeled as outliers in the figure. Note that inliers near the border of the bunny ears are over-removed.

All methods mentioned above in Section 2-2 have the advantage of implementation friendly. Distance-based outlier removal algorithms including LDOF and DDF need to search for a large number of nearest neighbors to guarantee a satisfactory sparse outlier removal, which will increase the computation time. While concerning clusters of outliers: LDOF fails to remove those outliers because small clusters of outliers have relatively low LDOF values and will be labeled as inliers as shown in Figure 2-12; the DDF method is based on region growing that is computation inefficient to be applied in large scale processing. The NNR method is more robust in sparse outlier detection and removing small clusters of outliers. But points on the edges also have a high proportion of uni-directional neighbors and will be detected

as outliers. Another problem is over-removing points near the area of manifold borders as shown in Figure 2-13. A more robust outlier removal algorithm is needed in this research to guarantee robust point cloud simplification.

2-3 Simplification

As mentioned before when dealing with large scale point clouds, noise and redundancy create problems for point clouds storage, visualization, reconstruction, segmentation, analysis and so on. Therefore, it is necessary to simplify the point clouds for further processing. A uniformly dense point cloud is desired in some applications such as multilevel smoothing and texture synthesis. Edge points give geometric skeleton information used for applications such as surface reconstruction, object detection and segmentation that should be preserved.

A lot of researches have been done in feature preserving mesh simplification in the last decade. Fleishman et al. used anisotropic bilateral filtering for mesh smoothing [21]; Jones et al. derived a non-iterative feature-preserving filtering applicable for arbitrary mesh triangles [22]; Hildebrandt and Polthier presented mesh simplification using prescribed mean curvature flow [23]; Digne et al. iteratively simplified 3D Delaunay triangulation of the points through optimal transport[24]. However, mesh connectivity is obsolete in many cases such as point-based visualization. Mesh reconstruction is computationally expensive. It is better to simplify the point cloud first and then reconstruct the mesh than the other way around [25]. Therefore we focus on the simplification of large scale point clouds by meshless point clouds simplification methods, which will be discussed in the following sub-sections.

2-3-1 Clustering methods for point cloud simplification

Clustering methods split the point clouds into different subsets and pick one representative from each subset to consist the simplified point cloud. The usual way of clustering is by subdividing the bounding box into grid cells and replace all sample points by a common representative, as implemented in Point Cloud Library [19]. A drawback of this method is that it cannot adapt to the non-uniform distribution in the point clouds and sensitive to the global grid size. Pauly et al. had concluded three novel point-sampled surfaces simplification methods including two clustering methods [25]. They estimated point normals and surface variation parameter by Principle Component Analysis (PCA). The point cloud is split into different clusters either limited by maximum cluster size aimed for uniform point set or by surface variation aimed for feature-sensitive result. Finally one representative is picked from each cluster to consist the final simplified point cloud. They proposed two methods of clustering: A bottom-up incremental approach creates clusters by region-growing and a top-down hierarchical approach splits the point clouds into clusters using binary space partition (See Figure 2-14).

However, the clustering is computationally expensive because the whole original point cloud is needed for partitioning. Shi et al. proposed to first use k-means clustering to group initial clusters and then apply maximum normal vector deviation as criteria to subdivide the initial clusters [26]. This results in a uniform distribution of points in flat area and denser near edges.

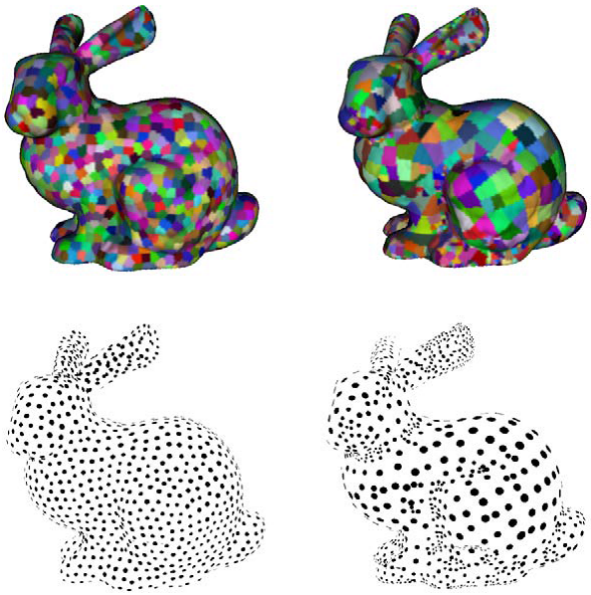


Figure 2-14: Left column: uniform incremental clustering; Right column: curvature-adaptive hierarchical clustering; Top column: clusters on original point clouds; Down column: simplified point clouds where size corresponds to cluster size. Figure from Pauly et al. [25]

2-3-2 Intrinsic point cloud simplification by geodesic Voronoi diagram

The geodesic distance between two points is a curve whose tangent vectors remain parallel if they are transported along it, which is the intrinsic shortest path between two points in real world. The edges are defined as the bisectors of neighboring cells with equal geodesic distance to their centers. Moening and Dodgson introduced a method for point cloud simplification in manifolds [27]. They suggested solving problems of simplification and re-sampling intrinsically by geodesic Voronoi diagrams. The Voronoi centroids consist the final simplified point cloud. It is constructed by embedding the point cloud in a Cartesian grid and propagating fronts simultaneously from an initial subset of input points outwards. The propagation procedure is achieved by fast marching algorithm, in which method computationally difficult intrinsic distance map has been transformed into easier-to-compute extrinsic distance map in a Euclidean manifold. By setting the propagation speed of each point to be the same weight, a uniform point cloud can be obtained. The propagation procedure and a simplification result are shown in Figure 2-15 and Figure 2-16 respectively.

2-3-3 Moving Least Squares (MLS) fitting with sharp features

Least-squares surface fitting is usually considered to be robust to noise but usually only used for reconstructing smooth surfaces. MLS was initially proposed by Lancaster and Salkauskas [29]. The central idea is to start from a weighted least squares formulation from an arbitrary point and move this point over the entire parameter domain to compute and evaluate the weighted least square for each point individually. Fleishman et al. proposed to resample point sets with feature preservation from noisy point cloud by MLS fitting [30]. They applied the forward-search paradigm: start from a robustly chosen subset of the original dataset without

Figure 2-15: Wave propagation procedure from left to right for computing geodesic Voronoi diagrams. Red points indicate the progressive intrinsic points. Right: Figure from Moenning and Dodgson [27].

Figure 2-16: Simplification result with user-controlled density. Figure from Moenning and Dodgson [28].

outliers and move forward with certain statistical estimates. Sharp features are well preserved by regarding points across the features as outliers. Instead of fitting surfaces locally they used iterative forward-search based refitting methods to classify a neighborhood to multiple local surfaces. Points close to more than one surfaces are projected on one of the smooth regions. Finally points are re-sampled on the piecewise local surfaces by MLS projection. An example of the procedure is shown in Figure 2-17 and a result in Figure 2-18.

2-3-4 Robust Implicit Moving Least Squares (RIMLS)

The Implicit Moving Least Squares (IMLS) method has been put forward by Shen et al. to reconstruct tangential implicit planes by standard MLS [31]. Using constant polynomials as the MLS basis in IMLS will generate a simple weighted average [32]. However this method has problems of expanding and shrinking effects. To solve this problem, Öztireli et al. suggested a Robust Implicit Moving Least Square method for effective meshless surface representation by combining IMLS with the robust LKR [33]. Spatial, residual and normal kernels are combined in one operator. The central function of their iterative minimization is below:

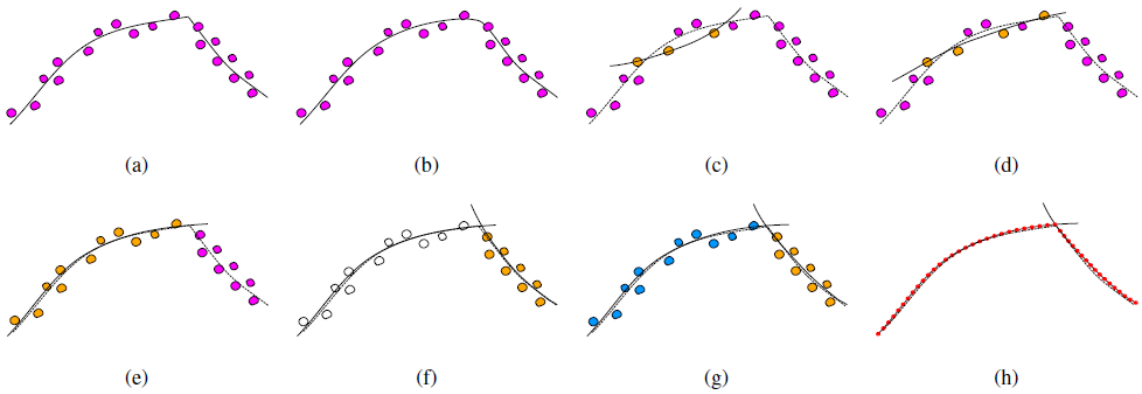


Figure 2-17: MLS simplification procedure. The noisy input point cloud can be regarded as two connected smooth surfaces (a) or one contact smooth surface (b). First, a surface is fit to a small reliable subset (c). Then incrementally add points with smallest residual and iteratively refit the surface (d). The final fitting result of one surface is shown in (e). Points across the edges are regarded as outliers to current surface but used for fitting another (f). Final result is in (g) with two surfaces. Uniform distance resampling is performed on the surfaces (h). Figure from Fleishman et al. [30].

$$f^k(x) = \frac{\sum \mathbf{n_i^T}(\mathbf{x} - \mathbf{x_i})\phi_i(\mathbf{x})\omega(r_i^{k-1})\omega_n(\Delta \mathbf{n_i^{k-1}})}{\sum \phi_i(\mathbf{x})\omega(r_i^{k-1})\omega_n(\Delta \mathbf{n_i^{k-1}})} \tag{2-10}$$

where n is the normal and x is the point. $\phi_i(x)$ is the spatial kernel. The kernel $\omega(r)$ handles the residual and $\omega_n(\Delta n)$ kernel for normals. This allows local control of sharpness around the edges. A result of RIMLS compared with IMLS is shown in Figure 2-19.

2-3-5 Edge-Aware point set Resampling (EAR)

Huang et al proposed a novel re-sampling approach to process a point cloud containing noise and even outliers to generate a noise-free and sharp edges preserving point cloud [16]. Their main idea is to start with a randomly sub-sampled point cloud and compute their normals by traditional PCA. Those initial points are consolidated using improved version of their Weighted Locally Optimal Projector (WLOP) [34] called anisotropic WLOP. The positions of points in each next iteration X^{k+1} is to minimize the formula below:

$$G(C) = argmin_{X=\{x_i\}i\in I}\{E_1(X,P,C) + E_2(X,C)\} \tag{2-11}$$

where E_1 term is the average term projecting points onto the original point cloud to approximate the geometry of points and smoothing out noise. It is further defined as:

$$[H]E_1(X,P,C) = \sum_{i\in I}\sum_{j\in J}||x_i - p_j||\theta(||c_i - p_j||) \tag{2-12}$$

And E_2 term in Equation 2-11 is the repulsion term punishing sample points getting too close to each other.

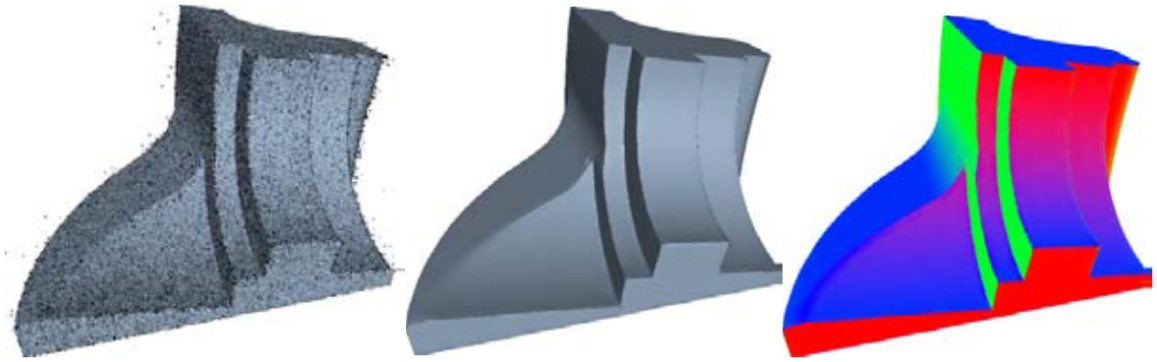


Figure 2-18: An example of the MLS fitting method. Left: noisy input cloud. Middle: re-sampling model; Right: middle model colored using normal direction. Figure from Fleishman et al. [30].

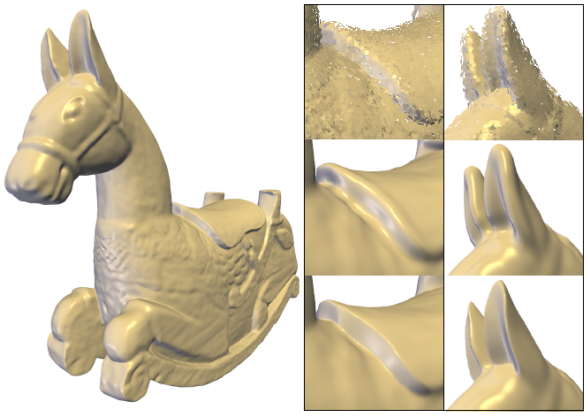


Figure 2-19: Left: RIMLS reconstruction result with 3% noise. Right: close views of sampling of the model (top), reconstruction with IMLS (middle) and RIMLS (bottom). Figure from Öztireli et al.[33]

$$[H]E_2(X,C) = \sum_{i' \in I} \lambda_{i'} \sum_{i \in I \setminus \{i'\}} \eta(||x_{i'} - c_i||)\theta(||c_{i'} - c_i||) \tag{2-13}$$

In both terms: I is the neighboring projected points; J is the neighboring original point cloud; $\theta(r)$ assigns decreasing smooth spatial weight; Φ re-samples away edges; η pushes projected points against each other; balancing term λ controls the repulsion force.

Regardless of iteration times, the initially sub-sampled point cloud distribution only determines number of points in the result and produces similar result compared with a same size but differently distributed starting sub-sample. The points are able to project uniformly onto the surface. To reduce the number of iterations and achieve a satisfactory result, a relatively uniform input sub-sample is preferred in practice. One example shows the projection from a crude initial non-uniform sub-sample to uniformly distributed result. (Figure 2-20).

WLOP does not need to know any extra information about the point cloud. Based on WLOP, the EAR needs to compute normals for both original and sampled point cloud and

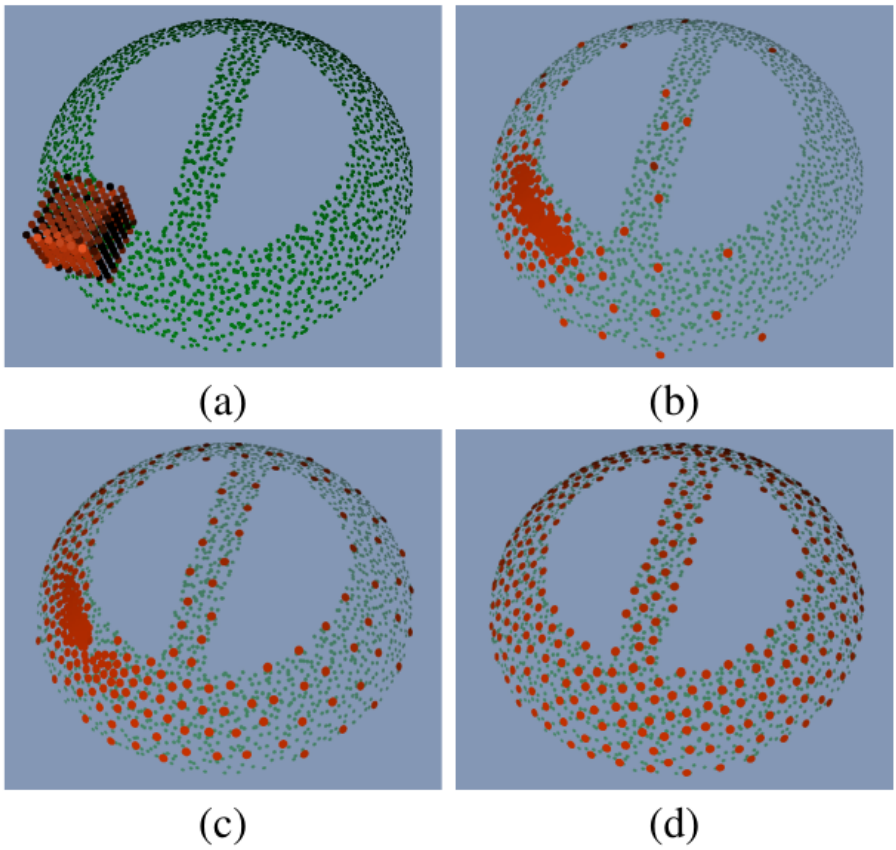


Figure 2-20: Starting from a crude initial sub-sampled point cloud in (a) where red points are projected onto green points. The WLOP operator iteratively from (b) to (d) distributes points uniformly onto the original point cloud. Figure from Lipman et al. [35]

adapt spatial operator $\theta(r)$ to a normal-spatial bilateral filter. Final sharp edges are created by up-sampling back to the edges. The whole pipeline is shown in Figure 2-21 and a result from real scanning data in Figure 2-22.

2-3-6 Discussion of existing point cloud simplification algorithms

Among the meshless point cloud simplification methods mentioned in Section 2-3, every method has its own advantages and disadvantages suitable in different applications. Pauly et al.'s clustering methods could produce uniform or feature-sensitive simplified point clouds efficiently. A drawback is that they estimated point normals by traditional PCA. PCA normal is unreliable especially near sharp edges which makes the method not suitable for satisfying uniform and feature-sensitive point cloud simplification at the same time. And this method does not take noise into consideration and not suitable to use in noisy point cloud.

Clustering methods are designed for sub-sampling the original point cloud. However their sub-sampling strategy considers every point with the same importance that is not able to keep the feature points. What is more, this strategy cannot smooth the noise.

Moening et al.'s intrinsic point cloud simplification is able to produce geodesically uniform

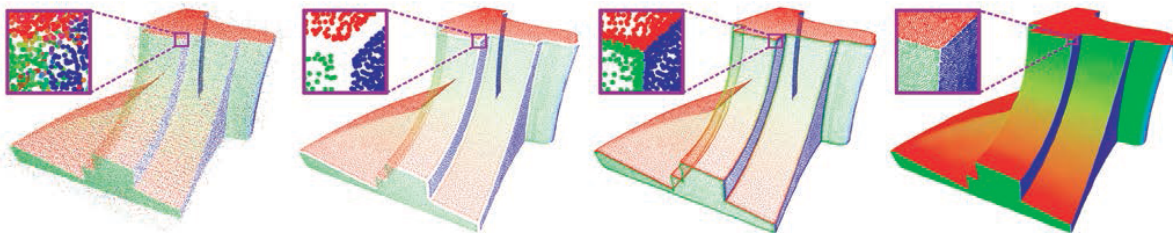


Figure 2-21: Overview of EAR pipeline. Input a noisy point cloud (a), points are first re-sampled away from edges creating gaps (b). Reliable normals can be obtained by the anisotropic WLOP operator for further up-sampling back near edges and filling the gaps (c). Point density is further increased by upsampling in dense area for point-based rendering (d). Figure from Huang et al. [16]

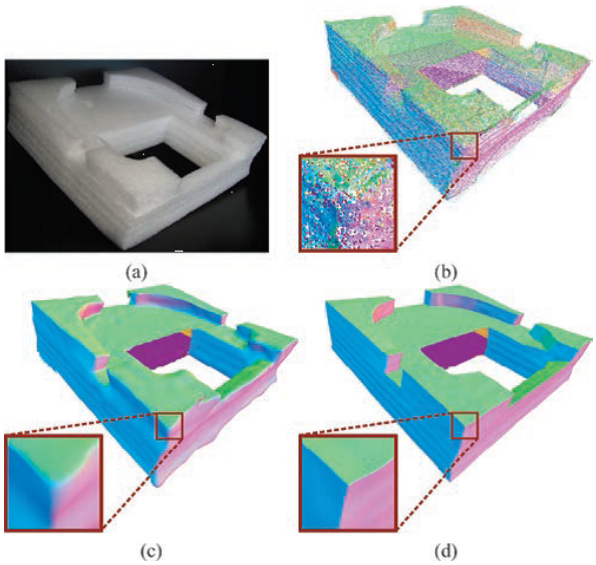


Figure 2-22: Laser scan of object (a) is contaminated with noise without normal information in (b). Traditional re-sampling without considering sharp edges (c). EAR result with edge preservation. Figure from Huang et al. [16].

point clouds by creating geodesic Voronoi diagrams on manifolds and they are re-sampled by the centroid of each Voronoi cell. But this method cannot distinguish between noise, outliers and feature points. Furthermore, this method is much harder to implement compared to the other methods and expensive in computation complexity due to the construction of the geodesic Voronoi diagram, which is obsolete in the final representation.

Fleishman et al.'s MLS method is able to refit surface iteratively that locally classify the samples across discontinuities. But their method requires a highly dense point cloud as input where regions near sharp features are always under-sampled in practice [36]. Local combination of different patches makes the approach slow. The global inconsistency of the classification will generate C^{-1} discontinuity and jagged edges [33]. Similar to Fleishman's MLS method, Öztireli et al.'s RIMLS method is more robust in classification and has controllable sharpness both locally and globally. But one shortage exists as indicated in Figure 2-23, a high noise-to-signal ratio will generate redundant features or over-smooth the result.

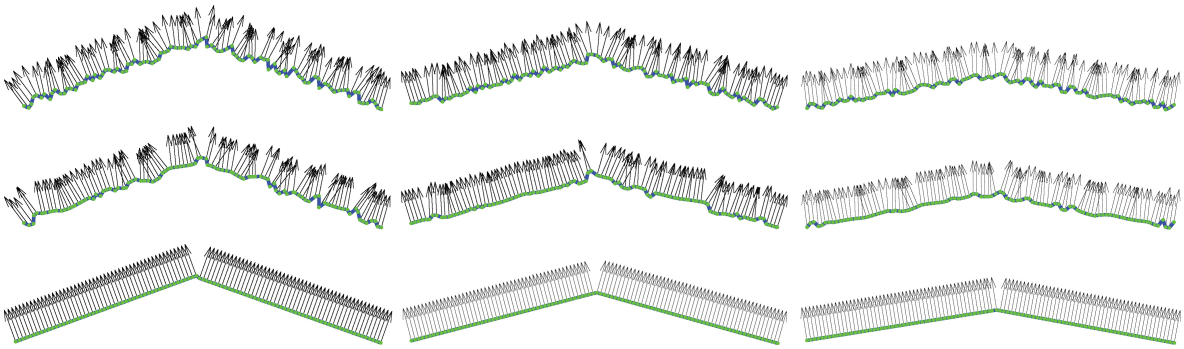


Figure 2-23: Problems in RIMLS. Top row: noisy input; middle row: fitting result after bilateral filtering in RIMLS where redundancy exists and the V-shaped edge cannot be correctly detected as an increase of angle across edges; bottom row: ideal result. Figure from [37].

Huang et al.'s EAR method produces quite satisfactory normal estimation and edge-aware resampling compared with the MLS and RIMLS methods. However the up-sampling generates too many points on the edges. The uniform property is achieved by up-sampling the flat area to be as dense as the edge area. This will generate more points than the original point cloud. Adaptation is needed to achieve the uniform density objective by reducing points in our research.

The methods listed in this section are popular in point cloud simplification. Different existing simplification algorithms have different focuses of processing. Some of them focus on generating a uniform result and some focus on reducing noise. Many of the simplification algorithms are designed to process small-size point cloud simplification that are slow and memory inefficient for large-size production. Of all the listed methods, none of them is able to consider keeping edge points, smoothing noise and generating uniform simplified result at the same time aiming at large-scale production. Inspired from these methods, our research focuses on how to adapt these existing general methods or create our own algorithm to deal with point cloud simplification in the specific scenario of fusing roof and facade point clouds in large scale.

Chapter 3

Outlier removal

This chapter gives our proposed algorithms in outlier removal based on the related research discussed in Section 2-2. A clean point cloud gives guarantee for later simplification. Outliers may exist singly without spatial connection with other points or in small clusters. In this chapter we derive our outlier removal algorithm by combining and improving the methods we discussed in Section 2-2. Our outlier removal algorithm can be applied in processing any point cloud.

3-1 Outlier removal by improved LDOF

Two proposed distance-based outlier removal algorithms LDOF and DDF factors have similar outlier detection strategy. LDOF requires less computation and easier to implement compared with DDF. So in our research we choose the LDOF algorithm to improve and adapt in our outlier removal algorithm,. As discussed in Section 2-2 and one example shown in 2-12, small cluster of outliers are unable to be detected in LDOF method. So we apply the technique proposed in NNR method by abandoning the first certain number of points in KNN search as outliers and then calculate for the KNN distance and inner KNN distance. This results in a new definition of cluster-adapted KNN distance and inner KNN distance with respect to Definition 1 and 2.

Definition 4 (Cluster-adapted K-Nearest Neighbor distance)**.** *Given a point cloud set* $P = \{p_1, p_2, ..., p_n\}$*. Let* N_p^{l+k} *be the set of the* $(l+k)$ *nearest neighbors of point* p_i *(excluding* p_i*) and* N_p^k *be the set abandoning the nearest* l *neighbors in* N_p^{l+k}*; the distance between point* p_i *and* p_j *be defined as* $d(p_i, p_j)$*. The cluster-adapted KNN distance of a point is:*

$$\overline{d'_{p_i}} = \frac{1}{k} \sum_{p_j \in N_p^k} d(p_i, p_j) \tag{3-1}$$

Definition 5 (Cluster-adapted K-Nearest Neighbor inner distance)**.** *The cluster-adapted K-Nearest Neighbor inner distance of a point* p_i *is the average KNN distance among points in*

N_p^k. *Formally defined as:*

$$\overline{D'_{p_i}} = \frac{1}{k(k-1)} \sum_{p_i,p_j \in N_p, i \neq j} d(p_i, p_j) \tag{3-2}$$

The cluster-adapted Local Distance-based Outlier Factor is defined as:

Definition 6. *The cluster-adapted LDOF of a point p_i in point set $P = \{p_1, p_2, ..., p_n\}$ is the ratio between its cluster-adapted KNN distance and cluster-adapted KNN inner distance:*

$$LDOF_{p_i} = \frac{\overline{d'_{p_i}}}{\overline{D'_{p_i}}} \tag{3-3}$$

The adapted LDOF is one of the factor considered in our final outlier removal algorithm. As a general problem in distance-based outlier removal algorithms, the outlier detection decides on the distance from the outlier to the group of inlier points. In Figure 3-1, the outlier is so close to the inlier points that its KNN distance and KNN inner distance are similar resulting a low LDOF value and will be labeled as inlier.

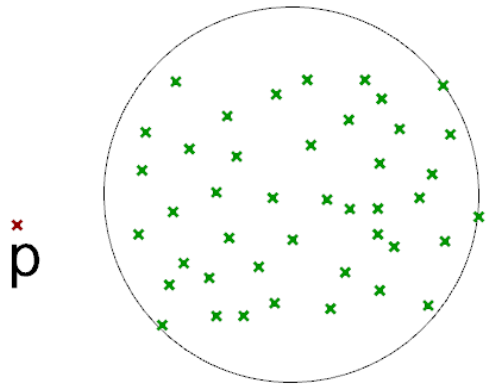


Figure 3-1: Problem of adapted LDOF algorithm. Notice that outlier p is unable to be detected because it is close to a group of inlier points.

NNR algorithm can avoid this problem because it does not need to consider how far the outlier deviates from the inliers. Next section we integrate LDOF into NNR algorithm and generate our final outlier removal algorithm.

3-2 Border-aware Nearest Neighbor Reciprocity outlier removal

As show in Figure 2-13, the NNR algorithm has the problem of over removing points on manifold borders which should be identified as inliers. Points on manifold border are close to the other points and have relatively low LDOF value. So we avoid this problem by integrating with our improved cluster-based LDOF algorithm. The final outliers are the set intersection of detected LDOF outlier point set and NNR outlier point set. The pseudo code of our final outlier removal algorithm is defined in Algorithm 1.

Algorithm 1 Outlier reduction algorithm

```
1: procedure OUTLIER REDUCTION(op, c)        ▷ outlier percent op; max cluster size c
2:     calculate cluster-adapted LDOF value for each point
3:     sort points using LDOF value in descending order
4:     outliersize ← inputsize ∗ op
5:     LDOFoutliers ← the first outliersize items in sorted result
6:     for p_i ∈ P do
7:         percent ← NNR unidirectional neighbor ratio of each point
8:         if percent ≤ op then
9:             if p_i ∉ LDOFoutliers then
10:                p_i is inlier
11:            else
12:                p_i is outlier
```

A comparison summary between results from different algorithms is shown in the example in Figure 3-2.

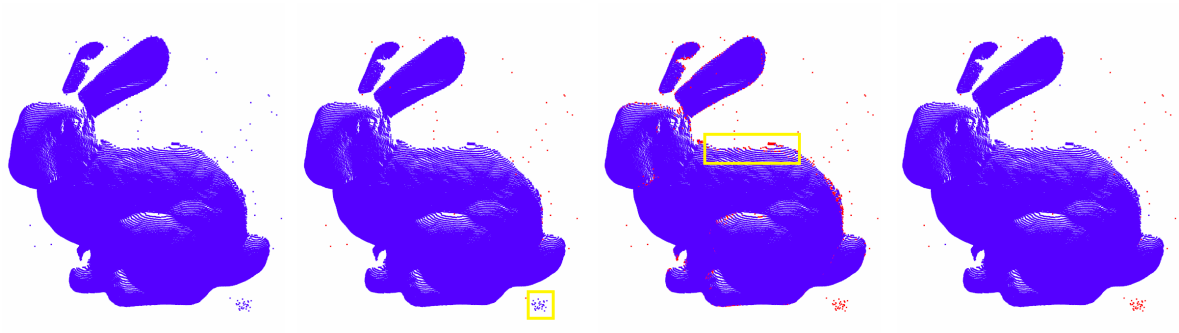


Figure 3-2: Comparison between different outlier removal algorithms where red points are labeled as outliers. From left to right: original point cloud, LDOF algorithm where cluster of outliers fails to be detected, NNR algorithm where inliers on manifold border are labeled as outliers, our cluster-adapted algorithm which solves both problems.

Starting from simplified single point clouds, we can further integrate and simplify point clouds. Simplification and integration will be discussed in next chapters.

Chapter 4

Simplification

Simplification is the key research in this thesis. In this chapter we start from an overview of the basic structure and sub-steps needed for simplification and then explain each step in the following sections.

4-1 Simplification overview

As already mentioned in section 1-3, in the simplification we aim at achieving the objectives of reducing noise, preserving edge points and controlling the uniform density at the same time. So in the design of the algorithm pipeline, different properties need to be considered. Besides, algorithm efficiency especially the running time is also a main issue for dealing with large-sized point cloud simplification. In order to adapt current general algorithms to our scenario, our strategy is mainly to search and optimize current general efficient simplification algorithms according to our case. Our final design of algorithm pipeline for the sub-steps in simplification is in Figure 4-1.

We start with outlier-cleaned roof and facade point cloud by algorithms proposed in Chapter 3. Point clouds need to be matched and merged for simplification. The standard merging procedure is using algorithms such as ICP by finding correspondences between neighboring point clouds and aligning by minimum least square distance fitting. Matching is closely related with the topic of gap filling between neighboring point clouds. This topic is beyond the research scope in this thesis and will only be mentioned in Chapter 7 as an open question for future research. In current proposed algorithm pipeline and implementation, point clouds are simply combined in the same coordinate system without other processing.

Based on different point cloud acquisition methods, we put forward a theory calculating importance value of each point in the point cloud as a metric used in later simplification. By merging point clouds from different scans, the feature points are extracted and preserved against later removal. After sub-sampling and smoothing the feature points, we keep them in the final simplified point cloud and apply the same sub-sampling strategy for other non-feature points. In order to smooth out the noise, we use WLOP operator and combine with

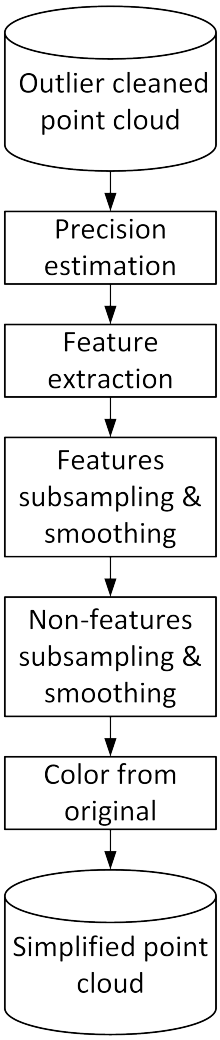


Figure 4-1: Algorithm pipeline of point cloud simplification.

our proposed importance value from different point clouds. Last colors are extracted from the original input point cloud to the simplified point cloud.

In the following sections each sub-step of simplification will be explained in the order of algorithm pipeline into detail.

4-2 Edge points extraction

The curvature gives the geometric importance of each point in its neighbors. Edge and salient points have a high curvature value given a proper curvature estimation operator. In our research we apply two curvature calculation methods from Pauly et al. [25] and Gumhold et al. [45]. Both methods calculate the curvature by eigen analysis. Assuming $0 \leq \lambda_0 \leq \lambda_1 \leq \lambda_2$ are all the eigen values of the local neighborhood of a point and v_0, v_1, v_2 are their corresponding eigen vectors, the plane defined by $T(x) = (x - \bar{p})v_0 = 0$ through point $\bar{p}$ minimizes the sum of squared distances to the neighbors of $\bar{p}$ (see Figure 4-2).

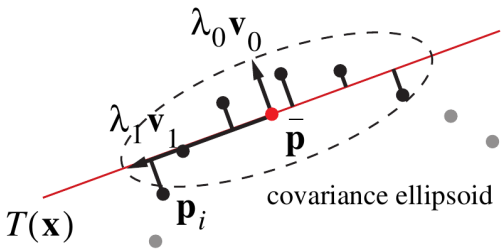


Figure 4-2: Curvature estimation from minimum eigen value λ_0. $T(x)$ represents the fitted plane of all the points by eigen analysis.

In Pauly's method, the curvature of each point is represented as one simple operator called surface variation:

$$\sigma_{sv} = \frac{\lambda_0}{\lambda_0 + \lambda_1 + \lambda_2} \tag{4-1}$$

While in Gumhold et al.'s method, the curvature estimation is more rigorous and accurate but more complicated. They deduct in KNN search to calculate the density of each point by using the K-th nearest neighbor distance D and point to plane distance λ_0 as shown in Figure 4-3.

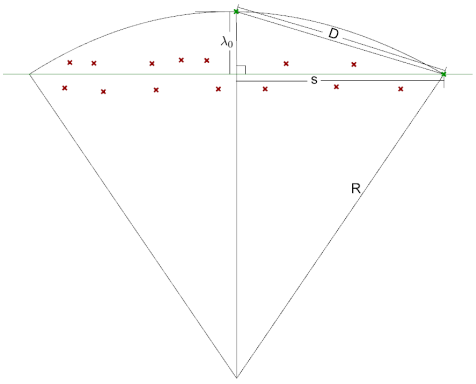


Figure 4-3: Curvature estimation from minimum eigen value λ_0 and K-th nearest neighbor distance D. Green line represents the fitted plane of all the points by eigen analysis.

The point-to-plane distance λ_0 is the same as the absolute of smallest eigen value λ_0 in equation 4-1. Thus the curvature is calculated by $\kappa = \frac{1}{R}$. With the other two constraint equations $s^2 = D^2 - \lambda_0^2$ and $s^2 = R^2 - (R - \lambda_0)^2$, the final curvature κ is deducted as below:

$$\kappa = \frac{2\lambda_0}{D^2} \tag{4-2}$$

In our test we found that both methods produce similar results. In case of simplicity and efficiency we use first method in our pipeline. One example colored by curvature is shown in Figure 4-4.

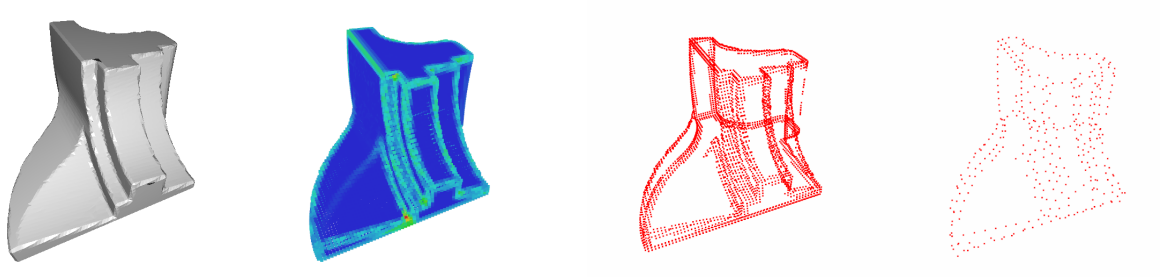


Figure 4-4: From left to right: original mesh; point cloud colored by curvature; feature points extraction with large curvature value; sub-sampled and smoothed feature points.

The extracted feature points are not uniform and may contain a lot of noise. So our solution is to sub-sample and smooth these points by adapting WLOP operator introduced in Huang et al. [34]. The sub-sampling and smoothing steps will be introduced in the next sections.

4-3 Sub-sampling

After extraction of the feature points, the original point cloud has been split into feature and non-feature points. For both types of points, sub-sampling is necessary to generate a simplified and uniform subset of the original point cloud. In our research we apply the same sub-sampling algorithm for extracted feature points and the non-features sub-sequentially. Those sub-sampled points will be combined in the step of Weighted Locally Optimal Projector (WLOP) in Section 4-4.

The strategy of simplification is by removing points falling inside the search radius. One example of sub-sampling process is shown in Figure 4-5.

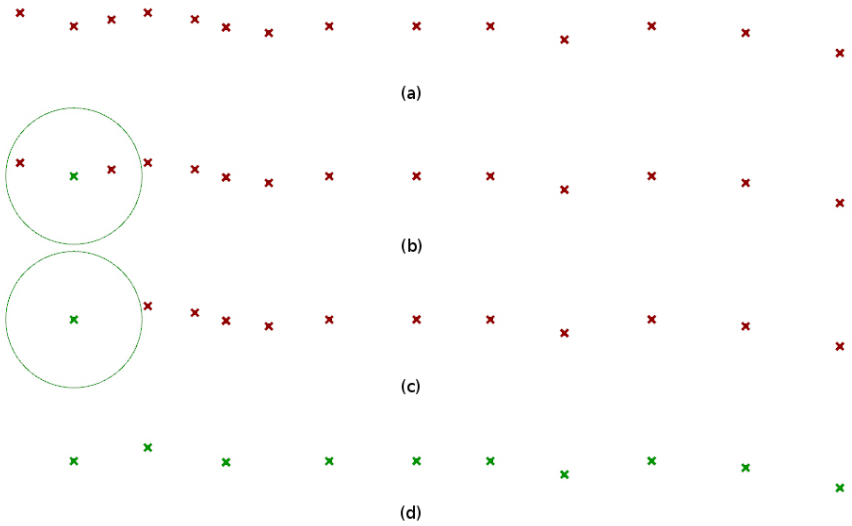


Figure 4-5: Sub-sampling algorithm. (a). original points; (b). for each point we remove the neighboring points inside the search radius with result in (c); (d). final result of sub-sampling.

A uniform subset of point cloud can be obtained in this step. Throughout this paper we use the standard deviation of density as the quantitative metric of the uniform property. One example of sub-sampling is shown in Figure 4-6.

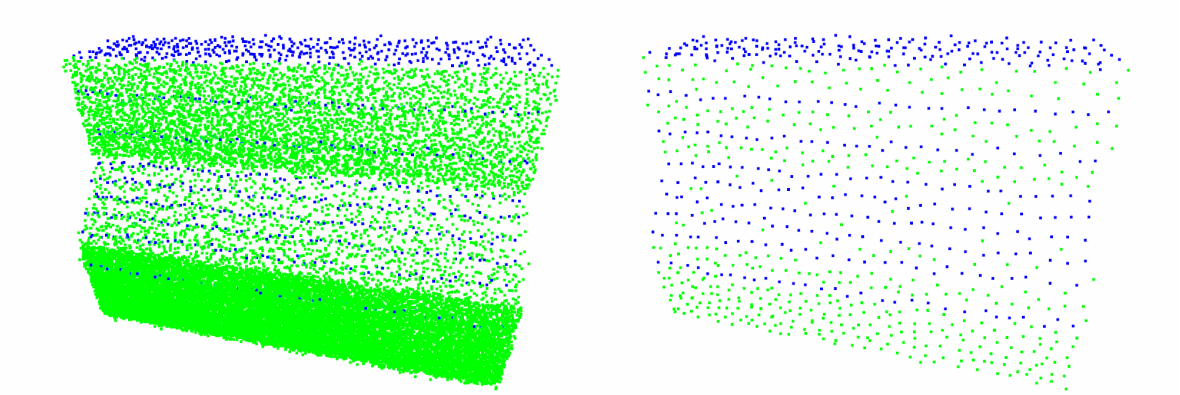


Figure 4-6: An example before (left) and after (right) sub-sampling.

4-4 Smoothing by adapted Weighted Locally Optimal Projector

Starting from a sub-sampled point cloud, we further smooth the point cloud to reduce the noise. In order not to destroy the uniform property already achieved in the sub-sampling step, we apply the operator Weighted Locally Optimal Projector to smooth the point cloud while still keep the uniform property. Here we explain deeper about the WLOP operator discussed in Equation 2-11 and adapt to our case.

As discussed in Section 2-3-5. The WLOP operator consists of two terms including the average term E_1 aiming at projection of points towards local distribution center and repulsion term E_2 aiming at punishing points getting too close to each other. In our adaptation, we combine into our weight value Iv_i derived from Equation 2-2. Because the weight value is calculated per original point. Term E_1 is the relation between points in the original point cloud and sample point cloud while term E_2 is between points inside sample point cloud. So we try to merge the density weight term Iv_i only in E_1 resulting the following our new E_1 term in the WLOP.

$$E_1(X,P,C) = \sum_{i\in I}\sum_{j\in J} ||x_i - p_j||\theta(||c_i - p_j||)Iv_{p_j} \tag{4-3}$$

while E_2 stays the same as in Equation 2-11. As a future improvement when the precision of each point is able to be derived, a noise weight WN_{p_j} can be appended after Iv_{p_j} resulting a weight map with consideration in both region and noise domains. Expanding the equation into details we derive our final weight adapted WLOP equation array with fixed point iterations as below:

$$x_i^{k+1} = \sum_{j\in J} p_j \frac{\alpha_{ij}^k / v_j}{\sum_{j\in J}(\alpha_{ij}^k / v_j)} + \mu \sum_{i'\in I\setminus\{i\}} \delta_{ii'}^k \frac{w_{i'}^k \beta_{ii'}^k}{\sum_{i'\in I\setminus\{i\}}(w_{i'}^k \beta_{ii'}^k)} \tag{4-4}$$

$$\theta = e^{-\frac{d^2}{R^2}} \tag{4-5}$$

$$\delta_{ii'}^k = x_i^k - x_{i'}^k \tag{4-6}$$

$$\alpha_{ij}^k = \frac{\theta(||x_i^k - p_j||)Iv_j}{||x_i - p_j||} \tag{4-7}$$

$$\beta_{ii'}^k = \frac{\theta(||x_i^k - x_{i'}^k||)}{||x_i^k - x_{i'}^k||} \tag{4-8}$$

$$v_j = 1 + \sum_{j' \in J \setminus \{j\}} \theta(||p_j - p_{j'}||)Iv_{j'} \tag{4-9}$$

$$w_i^k = 1 + \sum_{i' \in I \setminus \{i\}} \theta(||x_i^k - x_{i'}^k||) \tag{4-10}$$

One example showing the effect of WLOP operator is in Figure 4-7

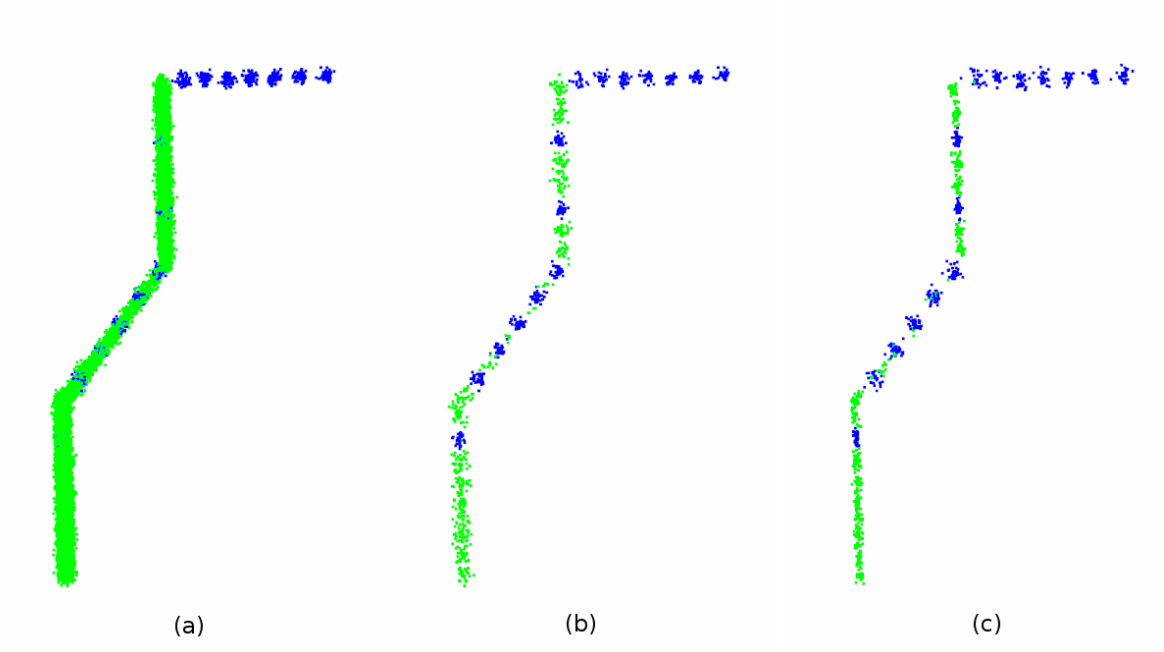


Figure 4-7: An example showing the effect of noise smoothing while keeping uniform property. The points are on four connected manifold surfaces and visualized in orthogonal view from side. (a)original point cloud with density standard deviation 0.0092; (b)initial sub-sampled point cloud with density standard deviation 0.0006; (c) WLOP smoothing result (3 iterations) with standard deviation 0.0006

The simplified point cloud is assigned color back by finding and averaging colors from the neighbors in the original point cloud in order to avoid aliasing artifacts. More examples and

applications of the algorithm pipeline in real-scene will be explained in Chapter 5 and Chapter 6.

Chapter 5

Integration and implementation

First in this chapter, the outlier removal and simplification algorithms are integrated into a unified algorithm pipeline. Then, virtual point clouds are simulated according to the scanner deployment for analysis and test purposes. Real-scene point clouds are pre-processed and prepared and finally algorithm prototypes are implemented for the entire algorithm pipeline. The result and analysis of the processing is discussed in Chapter 6.

5-1 Integration of outlier removal and simplification algorithms

With the independent algorithm components of outlier removal and simplification, we need to combine them into our final unified algorithm pipeline. There are two ways to order the sequence in the pipeline: first combine roof and facade point clouds and then perform outlier removal and simplification; or first remove outliers and then combine and do the simplification. We choose the second option because of two reasons: (a) The outliers are generated from scanning and should be excluded from the source not after merging; (b) The outliers have the lowest density value and will distort the importance value of other normal points. That is because the importance value of each point is calculated based on ratio with average density value as shown in Equation 2-2. Therefore we first apply our outlier removal algorithms for both roof and facade point clouds respectively then merge them to do the simplification. We extend Figure 1-1 and combine with Figure 4-1 into Figure 5-1 which is our final complete algorithm processing pipeline.

5-2 Implementation

In this implementation section, first our point clouds preparation procedure is introduced including generation of simulated point clouds in virtual scene and pre-processing of real-scene point clouds. Then, our software prototypes are discussed in a nutshell. Lastly the parameter tweaking strategies are given in the whole algorithm pipeline that require user interactions.

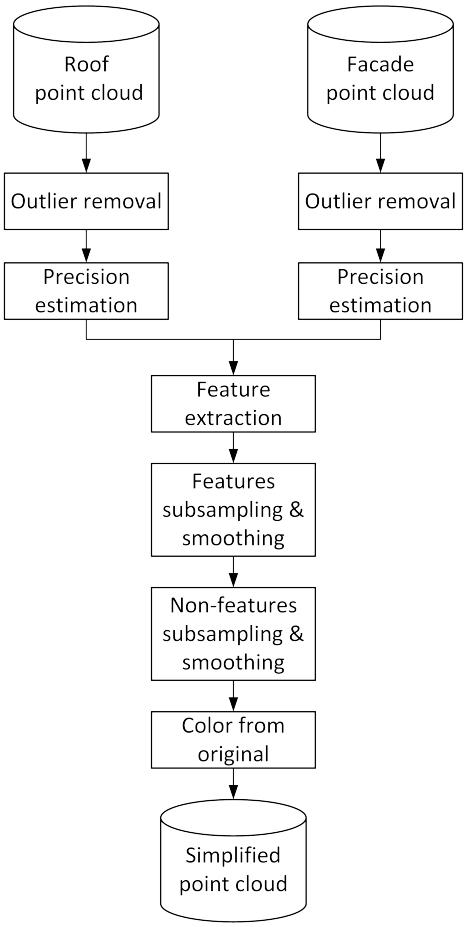


Figure 5-1: Complete algorithm pipeline.

5-2-1 Point clouds preparation

In this thesis, virtual scene point clouds are simulated for testing and analyzing algorithms. Later, the application on real-scene point clouds is presented.

Virtual point clouds simulation

Two virtual point clouds are simulated by intersecting laser rays with the virtual building mesh and assigning a color to each point from mesh texture images if available. The simulation is done in either *Rhino* software with *Grasshopper* plug-in for its easy-to-implement property or *Blender* software for its superiority in handling large-sized point clouds. The scanning simulation strategy is stated in Figure 2-5.

Figure 5-2 shows the scanner deployment and the original mesh of one demo that is created in *Rhino/Grasshopper*. For convenience, we call this demo "House". The "House" demo contains 252915 points.

Similar to the "House" demo another demo called "Villa" is generated in Blender. The "Villa" demo contains 1632047 points that are scanned by one airborne scanner and one street-view

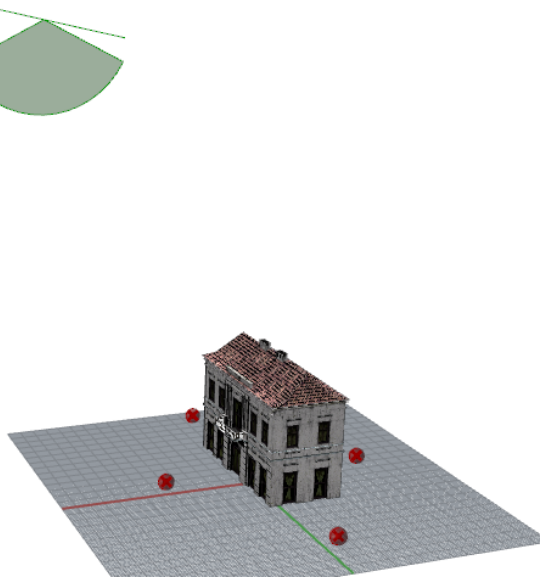


Figure 5-2: Simulated scan of demo "House". We put street-view cameras at four sides of the building and one airborne LiDAR scanner on top of the building. The scanner deployment strategy is the same as Figure 2-5.

scanner. Figure 5-3 gives the comparison of original mesh and our simulated point clouds seen from different views.

Real-scene point clouds preparation

For the real-scene point clouds, our roof point clouds are uniquely obtained from AHN2 which is an open-source dataset collected by airborne LiDAR scanners. Color is not encoded in the point cloud, thus aerial images are needed to add color information to the point clouds. We obtained the aerial images from *PDOK* service (Publieke Dienstverlening op de Kaart) in Netherlands. Facade point clouds come either from terrestrial LiDAR or panoramic imagery. Street-view terrestrial point clouds are provided by Fugro DRIVE-MAP system. Street-view panoramic imagery point clouds are provided by the company Cyclomedia. Using algorithms (i.e. SfM, BA, ICP) mentioned above, the street-view panoramic images are obtained and positioning consolidated. Based on the accurate panoramic images, sky pixels are removed using grab-cut algorithm [17] and final street-view point clouds are obtained by dense matching algorithm [18]. Figure 5-4 and Figure 5-5 are two demos with the same source of airborne point cloud but different source of street-view point clouds.

5-2-2 Software prototypes structure

The algorithm prototypes are implemented in *C++* and algorithm library from *Point Cloud Library (PCL)* is mainly used [19]. For convenience of user interaction with parameters tuning and point cloud visualization, a graphical interface is created using library *Qt* (see Figure 5-6).

In prototype implementation interface, algorithm and parameters are separated in three different classes (see Figure 5-7). The main data objects and algorithm functions are all stored in the *PointCleaner* class.

5-2-3 Parameters tweaking

We list all the parameters we use that require user input :

- Outlier removal

  K: neighbor search size (10 by default);

  L: denotes the minimum allowed cluster size in outlier removal (30 by default);

  P: percent of points in original point cloud regarded as outliers (0.01 by default);

- Simplification

  R: sampling radius (Adjust according to input)

  T: curvature threshold (Adjust according to input)

In outlier removal, users can input the parameters before processing and no interaction is needed in the processing. However in simplification, the parameters tweaking procedure needs user interactions in several sub-steps. The procedure is listed in Figure 5-8. Firstly, different outlier-cleaned point clouds are put into the pipeline and the average distance between points is calculated for each point cloud. Based on the average distance, the user can input sampling radius that has to be reasonably larger than the largest average distance of all the input point clouds. Next, the curvature of each point is estimated and visualized by color in the software. The user can compare the color from point cloud with the color legend in the software to give a proper curvature threshold. Finally, the automatic processing by the software is performed so that it generates the final simplified result.

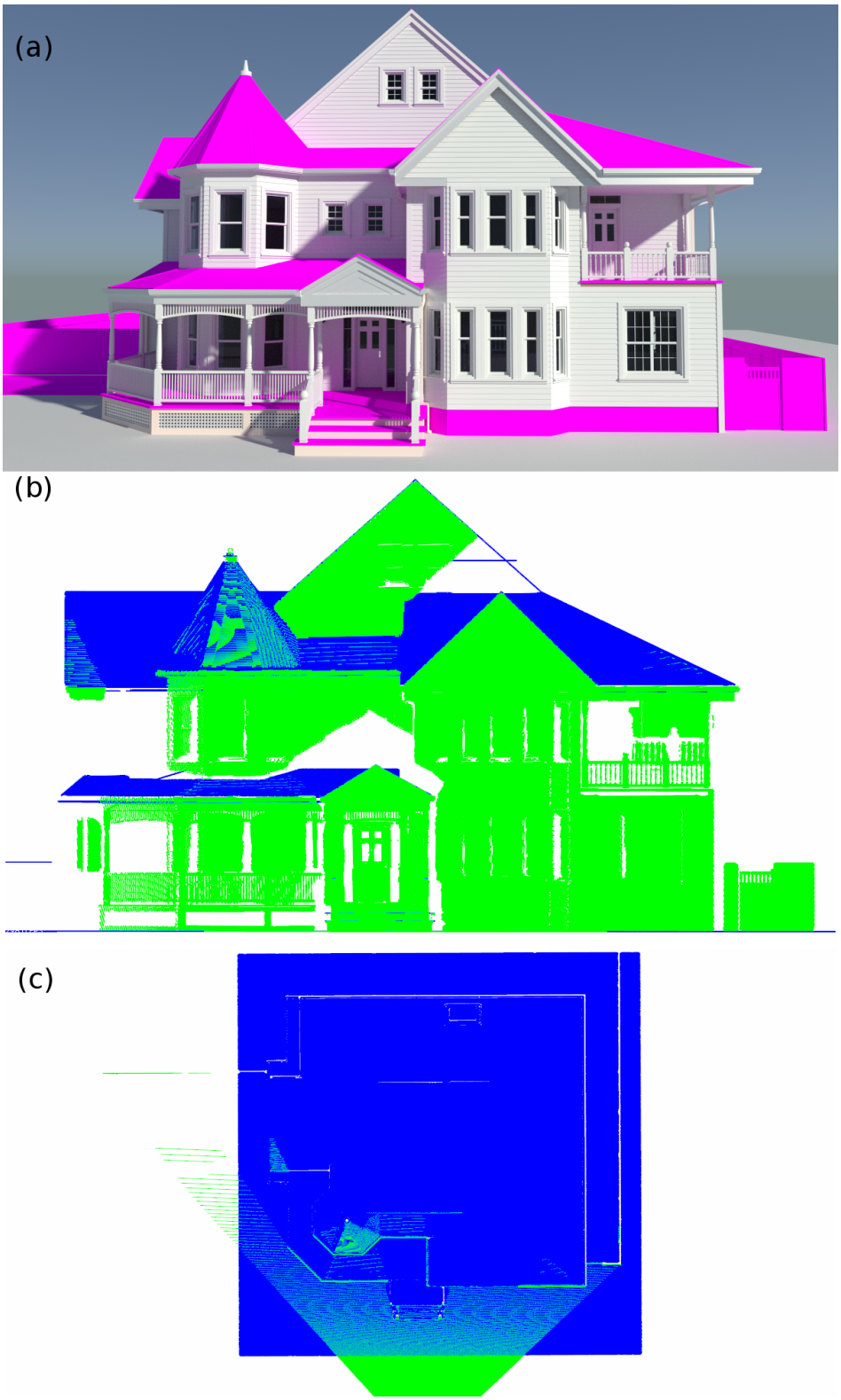


Figure 5-3: Simulated point cloud demo "Villa" (1668625 points). Blue points are from one airborne scanner and green points from one street-view scanner. (a) original mesh; (b) simulated point cloud seen from street-view ; (c) seen from airborne-view.

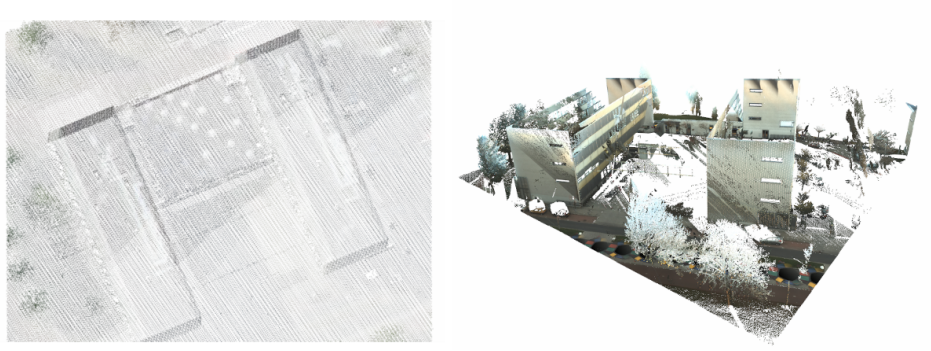


Figure 5-4: Demo "OTB building". Left: colored roof LiDAR point cloud; Right: facade terrestrial LiDAR point cloud.

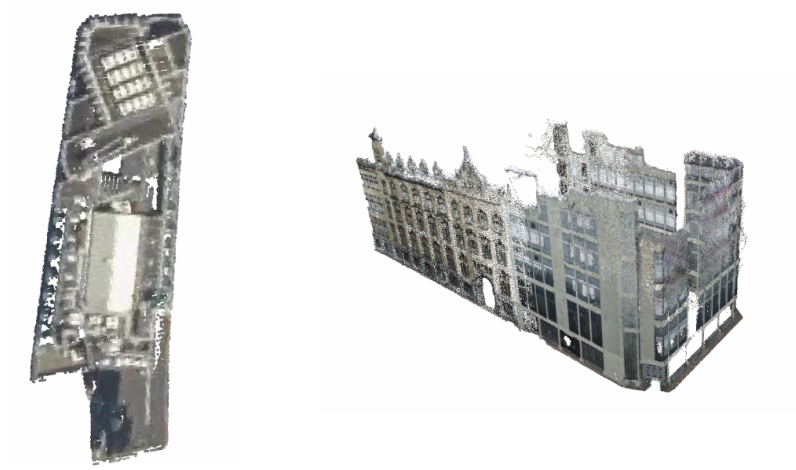


Figure 5-5: Demo "Amsterdam building". Left: colored roof LiDAR point cloud; Right: facade panoramic imagery point cloud.

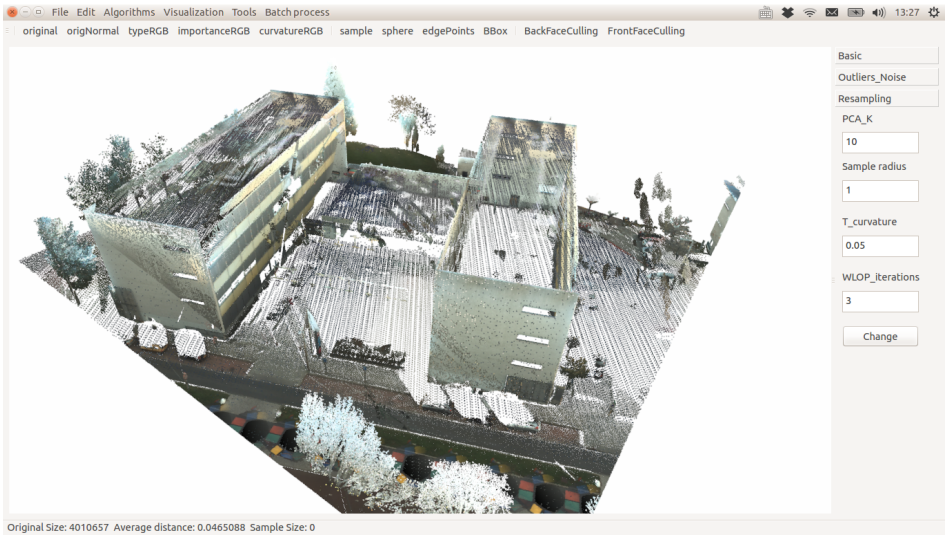


Figure 5-6: Software interface of our point cloud software.

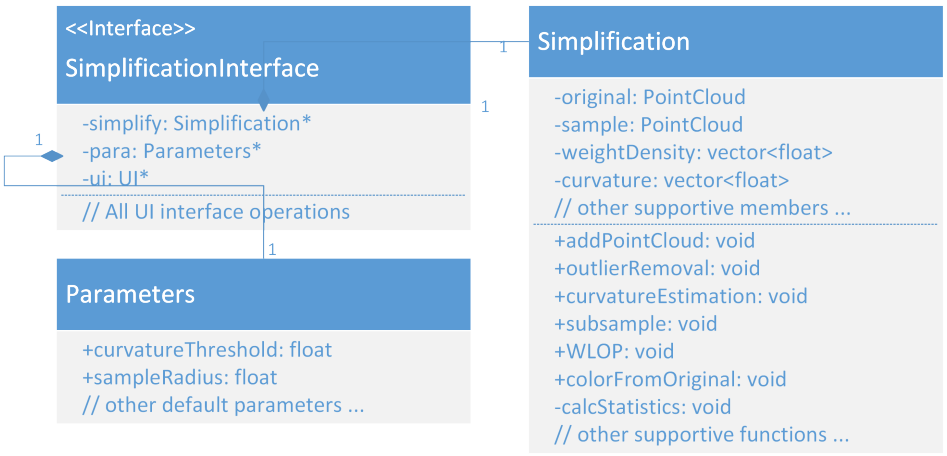


Figure 5-7: UML diagram of software prototype.

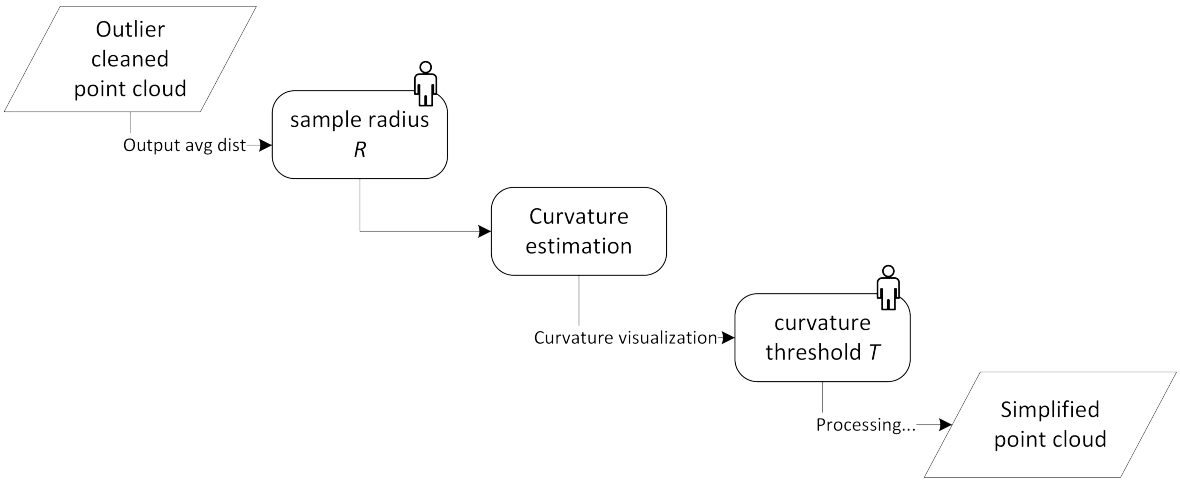


Figure 5-8: User-involved parameter tweaking procedure in point cloud simplification.

Chapter 6

Results, validation and discussion

This chapter presents the results of four demos pre-processed using the strategies and steps introduced in Chapter 5. The running time analysis of the algorithm is also listed. Then, the results are validated statistically by checking with our objective list. Finally, the results and limitation are discussed.

6-1 Results

6-1-1 Simplified point cloud demos

Demo 1: House

The result of each step in the simplification of "House" demo is shown in Figure 6-1.

Demo 2: Villa

The simplified result of demo "Villa" can be seen in Figure 6-2.

Demo 3: OTB building - Fusion of airborne LiDAR and terrestrial LiDAR point clouds

Figure 6-3 shows the effect of outlier removal for both point clouds.

A comparison between the outlier-cleaned point cloud and the final simplified point cloud is shown in Figure 6-4.

Demo 4: Amsterdam building - Fusion of airborne LiDAR and terrestrial panoramic imagery point clouds

Using the same processing pipeline as the methods mentioned above, we remove the outliers of Amsterdam building point cloud shown in Figure 6-5. And Figure 6-6 shows the simplified result with zoomed-out view.

6-1-2 Running time profiling

A summary of the running time for each demo is given in Table 6-1.

	House(252915)	Villa(1668625)	OTB(4204626)	Amst(833584)
Outlier removal	N/A	24.19	57.91	14.09
Weight calculation	0.61	4.11	12.24	2.15
curvature estimation	0.17	0.91	2.44	0.47
Subsample	5.02	6.82	1.01	0.30
WLOP initialization	12.25	155.71	525.57	27.81
WLOP iteration x 3	0.60	11.94	28.19	4.95
Color	0.01	0.08	0.19	0.06
Total	18.66	203.76	627.55	35.74

Table 6-1: CPU runtime of different point clouds processing (in seconds). Note that the house demo is clean no outlier removal is needed.

As we can see from the time statistics, the most time consuming sub-step in the processing is the WLOP initialization. This is because the radius search is necessary and unavoidable in order to compute the weight value v in the WLOP term for each point in the original point cloud. Radius search is running-time expensive in a dense point cloud.

6-2 Validation

Results are validated qualitatively or quantitatively with respect to our proposed three objectives: edge points preservation, uniform density and noise smoothing.

6-2-1 Edge points preservation

The decision about whether a point is an edge point depends on user input sampling radius and curvature threshold. The effect can only be verified qualitatively by checking the visualized result. As we can see from Figure 6-1 and Figure 6-2, the edge points are well preserved and sub-sampled given a set of proper parameters.

6-2-2 Uniform density

Uniform density can not only be directly verified qualitatively by visualization, but it is also possible to be estimated quantitatively by calculating the standard deviation of density. For the demos used in this chapter, we list in Table 6-2 of standard deviation changes between original point cloud, intermediate sub-sampled point cloud and result simplified point cloud.

	House	Villa	OTB	Amst
Original point cloud	0.0506	7.4009	2.7294	0.6384
After Sub-sampling	0.0016	0.0217	0.0044	0.0053
After WLOP smoothing	0.0016	0.0212	0.0047	0.0052

Table 6-2: Density standard deviation changes for different phases.

It can be seen from the result that the sub-sampling phase can greatly reduce the standard deviation of density value and the subsequent WLOP smoothing factor can well preserve or even improve the uniformity.

6-2-3 Noise smoothing

Noise smoothing effect can be verified visually and rigorously by estimating deviation from intrinsic manifold surface. One visual verification is already done in a simple demo in Figure 4-7. A rigorous validation is only possible to perform on the given virtual point cloud since the intrinsic manifold surface is unable to be derived from real-world scans. A noisy point cloud for the House demo is generated in Figure 6-1. We use the distance from the point cloud to the nearest point on the original mesh surface as a metric. Statistics for this distance value are listed in Table 6-3.

	Max	Avg	Std
Noisy input	0.478	0.077	0.059
Noisy input simplified result	0.434	0.055	0.054

Table 6-3: Noise smoothing effect analysis for House demo added with Gaussian noise.

6-3 Discussion and limitations

6-3-1 Results discussion

What can be derived from the results and validation of four demos is that the algorithm pipeline generates outlier-removed, noise-reduced and uniform results that can well preserve the edge points.

The *CPU* running time is satisfactory in a non-optimized implementation that can be applied in large-scale production. As we can see in Table 6-1, the step that takes most running time is the WLOP initialization which can be further traced to the reason of radius neighbor search for each point. As the neighbor search for each point has no conflict with each other, this problem can be solved either in *CPU* by multi-threading technology or adapting to *GPU* that has greater power in processing large-size data in parallel.

6-3-2 Limitations

In principle our method can be applied to fuse any point clouds that are registered properly in the same coordinate system. In general, our method only reduces points in dense area and

cannot up-sample points in sparse area, especially where large gaps or holes exist. This has limited that the user input sample radius has to be reasonably larger than the average distance in the sparsest input point cloud to achieve a satisfactory result. In other dense point clouds, abundant texture information may be excluded during simplification, which makes it difficult for point-based visualization. Due to this deficiency, too many existing holes in the input point cloud severely degrade the visualization result as can be seen from the church demo in Figure 6-7. All in all, adding points can be helpful in simplification but not considered in our research.

Another limitation lies in the fact that the edge points extraction is sensitive to severely-existing noise in the input point cloud. Our curvature estimation is performed locally. If the input point cloud contains severe noise, points in a local region will all either be marked as feature points or non-features. That will make the parameter of curvature threshold hard to choose and edge points difficult to be selected.

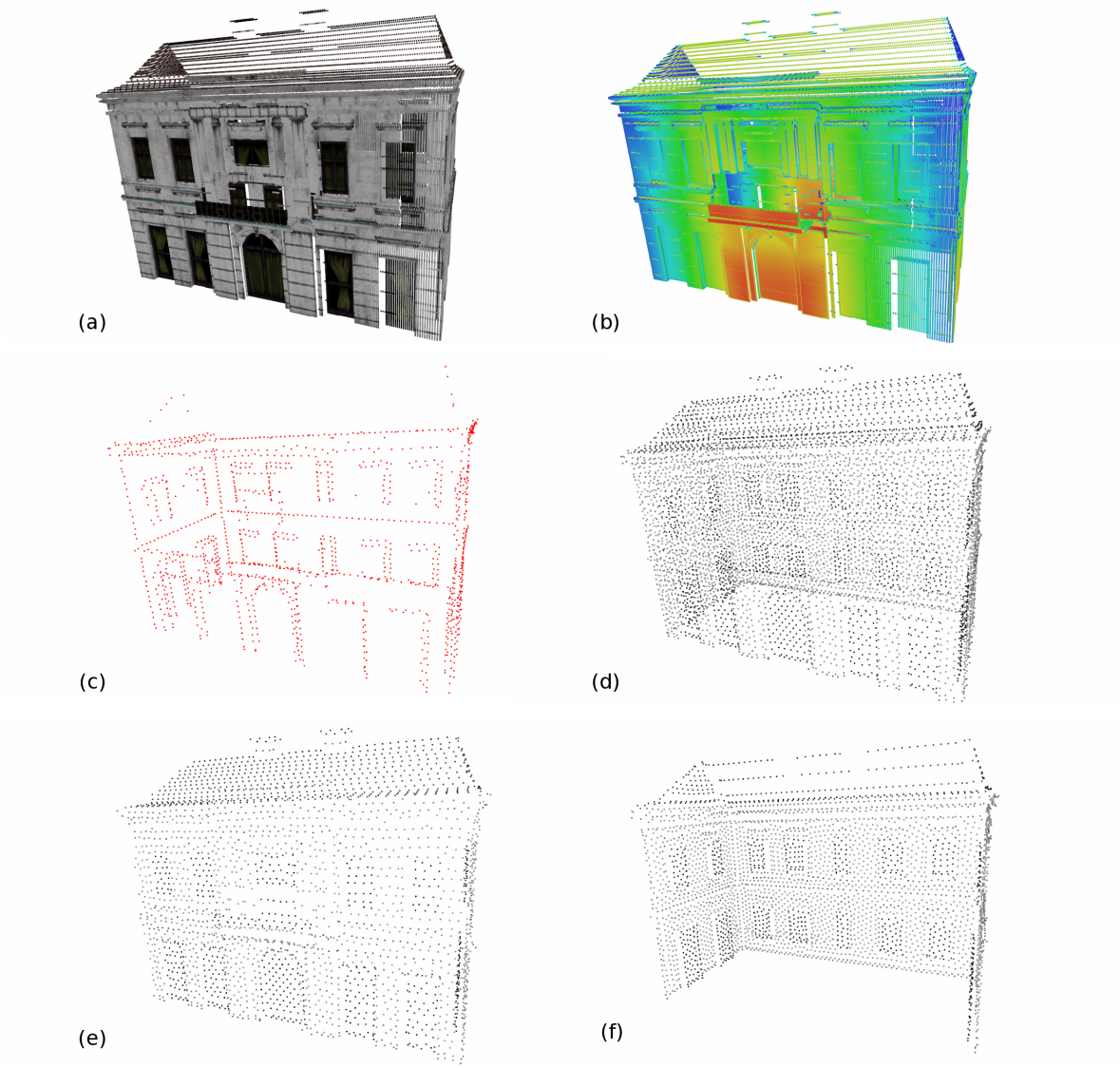


Figure 6-1: The result of each sub-step in our simplification algorithm pipeline. (a) original simulated point cloud; (b) weight value assigned to each point; (c) points with high curvature are extracted, smoothed, sub-sampled and preserved against later removal; (d) final simplified result; (e) front-face culling of the result; (f) back-face culling of the result. Parameter used $R = 2, T = 0.001$.

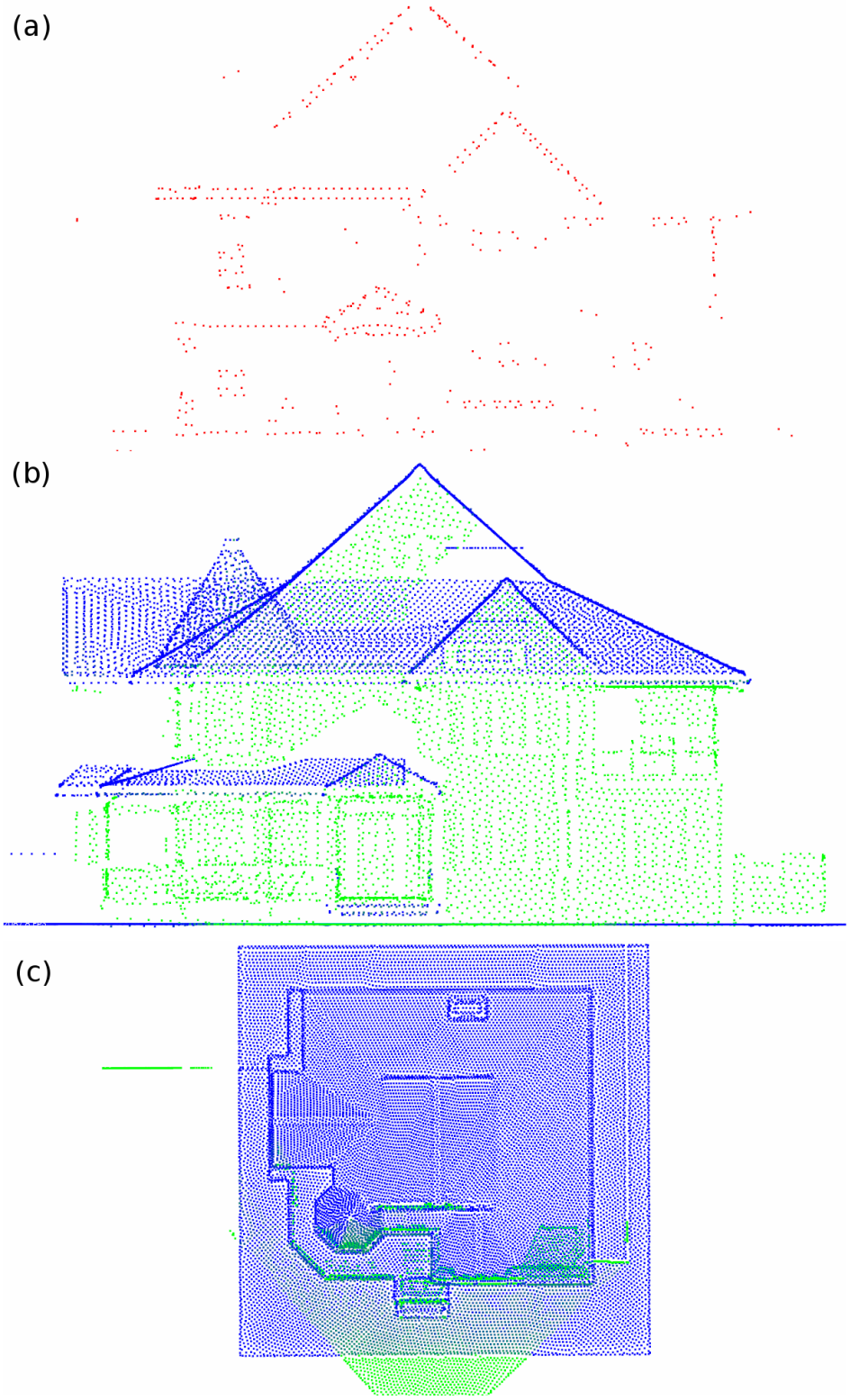


Figure 6-2: Simplification result of point clouds in Figure 5-3 (Sample radius $= 0.1$, curvature threshold $= 0.04$). (a) preserved edge points in the final simplified point cloud; (b) simplified point cloud seen from street-view level; (c) simplified point cloud seen from airborne level. Parameter used $R = 0.2, T = 0.04$

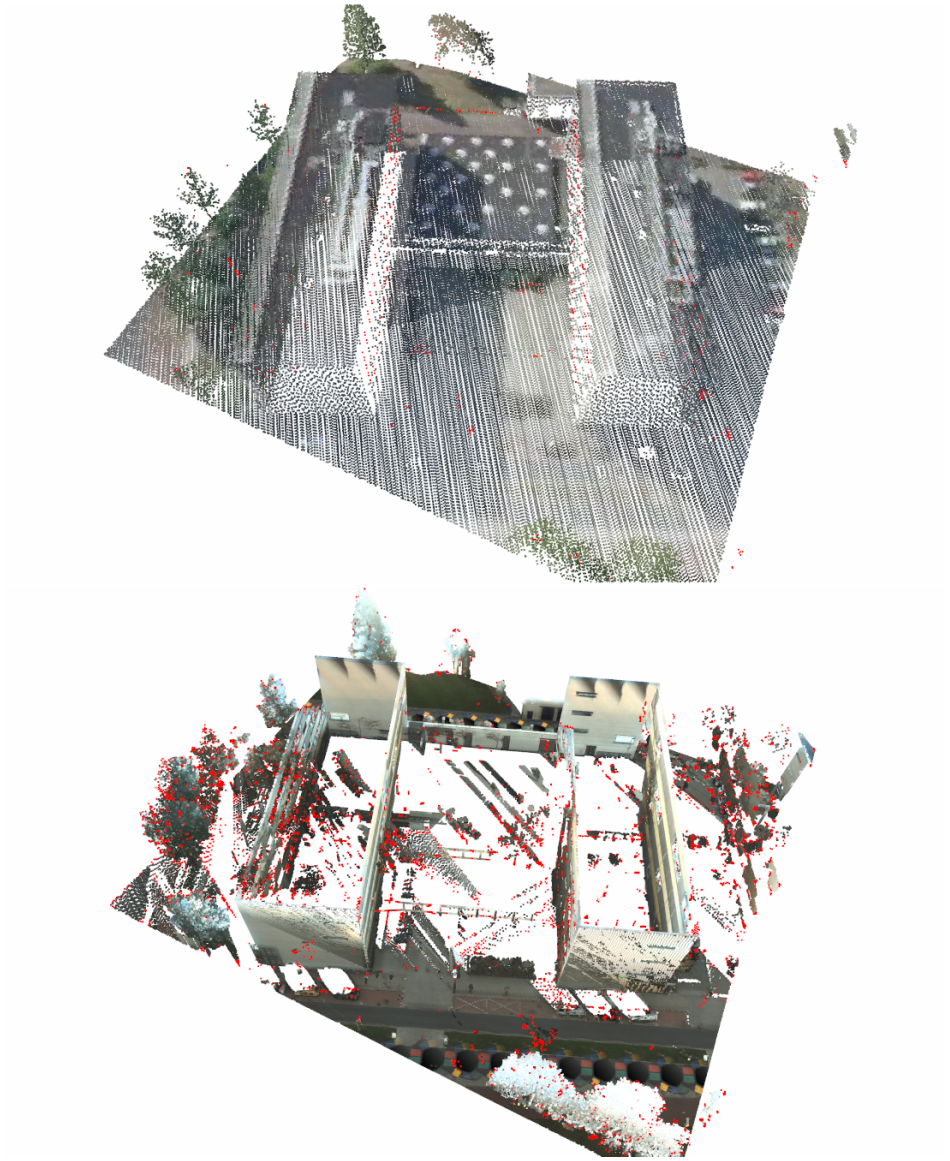


Figure 6-3: Outlier removal for OTB airborne (above $K = 10, L = 30, P = 0.005$) and terrestrial (below $K = 10, L = 30, P = 0.01$) point clouds with red points detected as outliers.

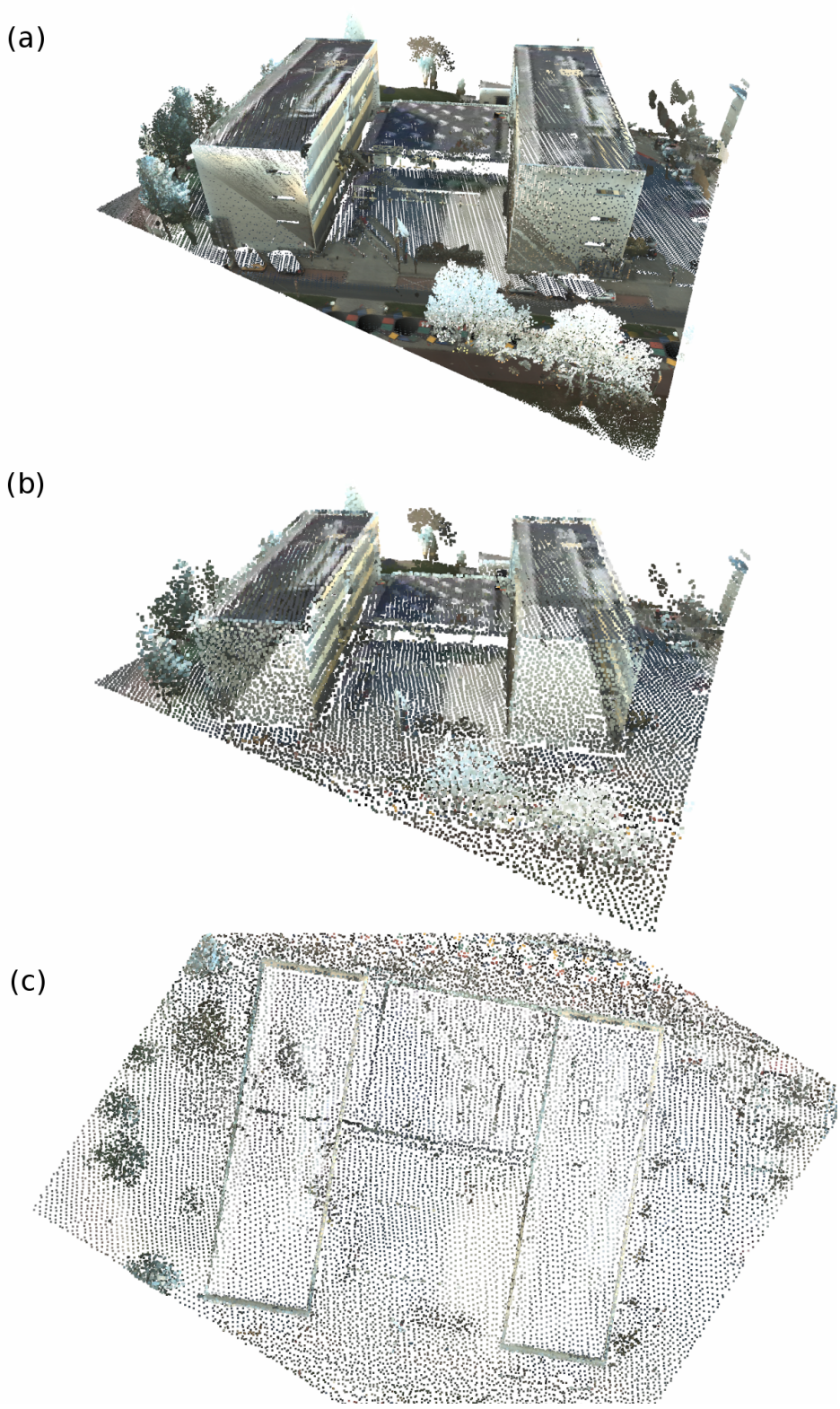


Figure 6-4: Simplification result for "OTB building" (With parameters $R = 0.5, T = 0.001$). Outlier-cleaned and merged point clouds as input in (a). Final simplified point cloud is shown in (b) and (c) in different view positions.

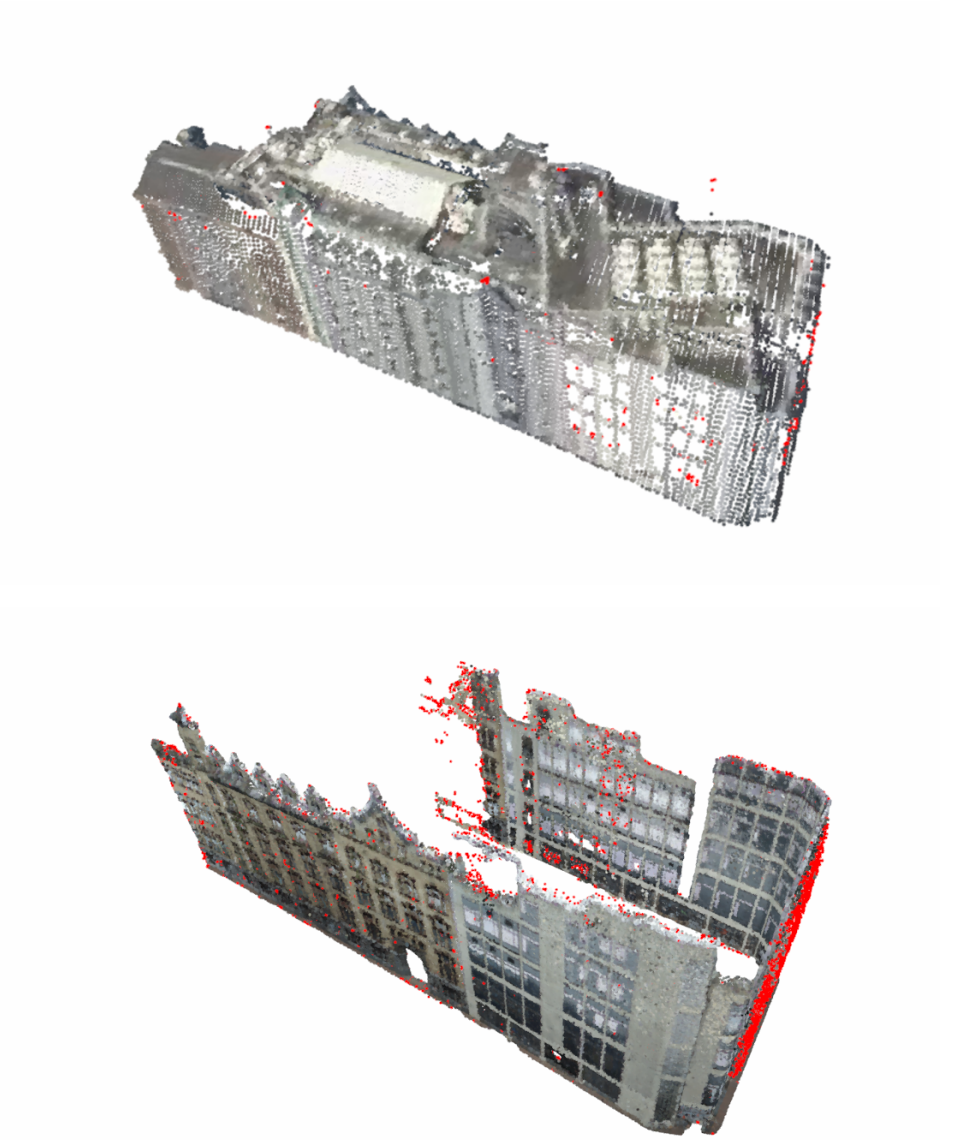


Figure 6-5: Outlier removal for Amsterdam building airborne (above. $K = 10, L = 30, P = 0.01$) and terrestrial (below. $K = 10, L = 50, P = 0.01$) point clouds with red color detected as outliers.

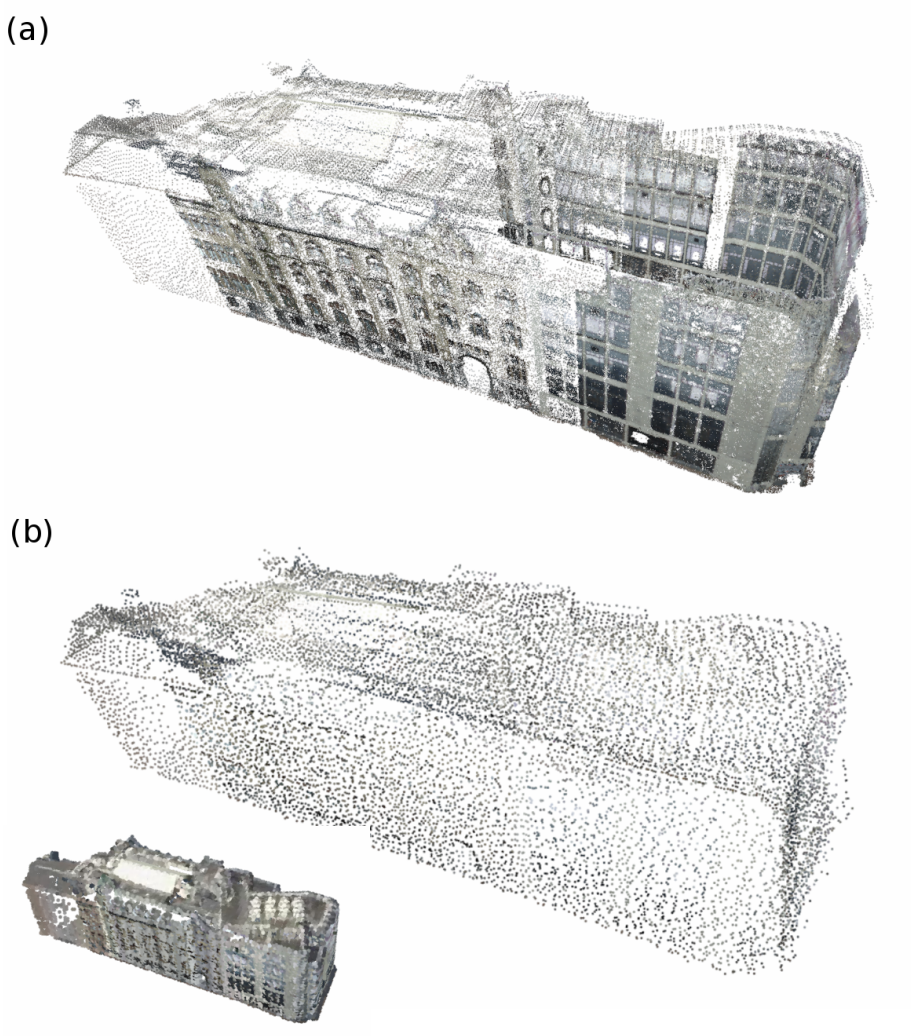


Figure 6-6: Panoramic imagery point cloud and LiDAR point cloud fusion and simplification. (With parameters $R = 0.5, T = 0.01$) Outlier-cleaned and merged point clouds as input in (a). Final simplified point cloud is shown in (b) with zoomed-out view in the left-bottom corner.

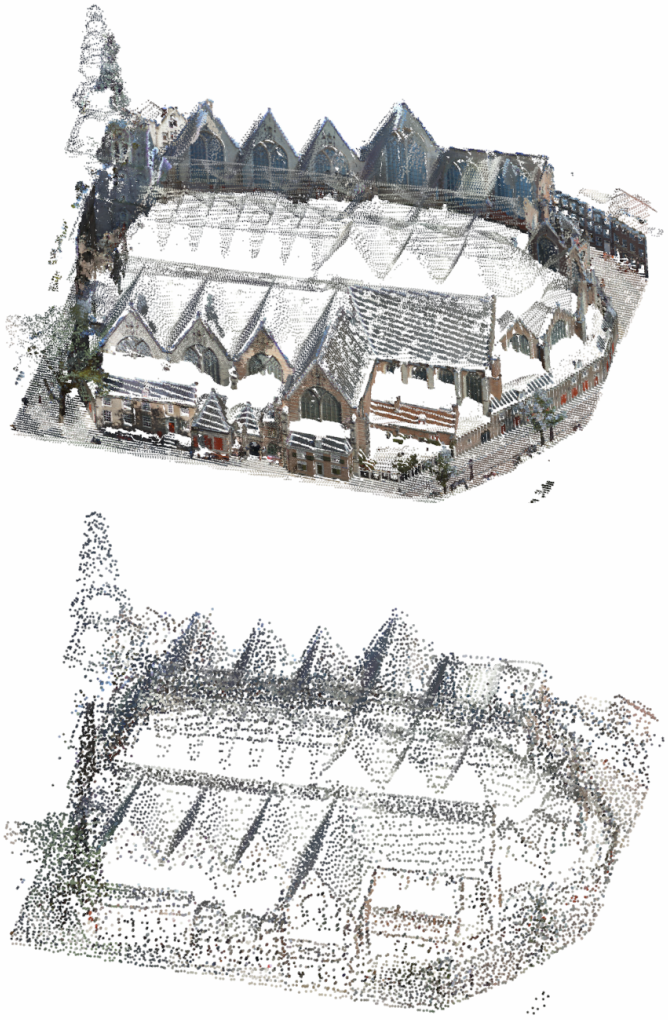


Figure 6-7: Simplification of a church point cloud. There are too many holes existing in both roof and facade point clouds. Our simplification only reduces points and thus rich texture information is lost in the result for point-based visualization.

Chapter 7

Conclusions and future work

In this chapter we conclude the thesis and provide recommendations for future improvement.

7-1 Conclusions

To fuse large-size roof and facade point clouds into a unified point cloud, aspects including algorithm correctness, algorithm efficiency and adaptability to different datasets need all be taken into consideration. In this thesis several algorithms are proposed and organized into a sequential pipeline for point cloud ourlier removal and simplification. The conclusions are made with respect to answer the research question:

- *Which algorithms are most appropriate to fuse roof and facade point clouds into an edge-aware and uniformly dense color point cloud?*

We answered as follows: we designed the most appropriate algorithms for fusing roof and facade point clouds either by developing algorithms ourselves or using and adapting existing algorithms and integrated into a unified algorithm pipeline, where outlier removal and simplification are performed and integrated that can produce an outlier-cleaned, noise-reduced, edge-aware and uniform point cloud. In outlier removal, different from other existing algorithms, our method can remove both singly scattered and small cluster of outliers without over-removing artifacts. Furthermore, in principle our outlier removal algorithm can be applied in any point cloud. In simplification, compared to other algorithms which focus on one or two objectives of uniform density, reducing noise and edge-awareness separately and having no consideration of data source, the presented approach can achieve all the mentioned objectives in one unified pipeline with consideration of precision distribution according to data source. The algorithm pipeline is efficient in running time that can be applied in large-scale production.

All proposed methods are implemented and tested first on virtually simulated environment and then on real-scene point clouds. For validation we made a checklist with respect to the objectives.

7-2 Recommendations for future work

According to the conclusions above and limitations in 6-3-2, we propose our recommendations for future work.

In precision estimation, a mathematical rigorous error estimation model is preferred for both LiDAR and panoramic imagery point clouds to construct a precision map used in simplification. In point cloud simplification, running time efficiency can be improved by implementing radius search in parallel using multi-threads in *CPU* or even adapt to run in *GPU*. Gaps and holes, either existing in original scanned point cloud or detected in the merging between point clouds, have to be filled. The gap between roof and facade point clouds can be filled in the process of registering point clouds in the same coordinate system using algorithms such as ICP. The holes need to be detected and distinguished between occlusion and real existing holes. The edge points extraction in simplification has to be more robust to existing severe-noise. This could be solved by adapting local curvature estimation to global.

Appendix A

Source code (part)

The central algorithms are all designed and implemented in the class *PointCloudCleaner* using C++ and Point Cloud Library (PCL). We select the most important function code in the class to be put in this appendix.

A-1 Header file of class *PointCloudCleaner*

```
1  #ifndef POINTCLOUDCLEANER_H
2  #define POINTCLOUDCLEANER_H
3
4  // basic for timer
5  #include <stdlib.h>
6  #include <time.h>
7  #include <sys/time.h>
8
9  // boost libraries for Gaussian random numbers
10 #include <boost/random.hpp>
11 #include <boost/random/normal_distribution.hpp>
12
13 // point cloud library
14 #include <pcl/common/common.h>
15 #include <pcl/point_cloud.h>
16 #include <pcl/point_types.h>
17 #include <pcl/features/normal_3d.h>
18 #include <pcl/octree/octree.h>
19 //#include <pcl/octree/octree_search.h>
20 #include <pcl/search/kdtree.h>
21 #include <pcl/filters/filter.h>
22
23 //#include <pcl/console/print.h>
24 //#include <pcl/gpu/containers/initialization.h>
25 //#include <pcl/gpu/octree/octree.hpp>
```

```
26 // project files
27 //#include "octree.h"
28 //#include "octree.cpp"
29
30 typedef pcl::PointXYZRGB Point;
31 typedef pcl::Normal Normal;
32 typedef pcl::PointCloud<Point> Cloud;
33 typedef pcl::PointCloud<Normal> CloudNormals;
34 //typedef pcl::gpu::Octree OctreeGPU;
35 typedef pcl::octree::OctreePointCloudSearch<Point> Octree;
36 typedef pcl::search::KdTree<Point>::Ptr KdtreePtr;
37
38 const float _ZERO = 1e-20f;
39 const float _BIG_FLOAT = 9999999.999f;
40 const float _PI = 3.14159265359f;
41
42 class PointCloudCleaner
43 {
44 public:
45     PointCloudCleaner():_octree_resolution(0.5f),
46                         _octree(_octree_resolution),
47                         _kdtreePtr(new pcl::search::KdTree<Point>()),
48                         _sample_kdtree(new pcl::search::KdTree<Point>()),
49                         _num_aerial(0),
50                         _num_street(0)
51     {
52         _cloud.clear();
53         _normals.clear();
54         /* Run octree radius search on GPU */
55 //        pcl::gpu::setDevice(0);
56 //        pcl::gpu::printShortCudaDeviceInfo(0);
57     }
58
59     /* Basic functions */
60     void add_cloud(const Cloud &cloud, bool is_aerial, int K = 5);
61     Cloud& get_cloud(){  return _cloud;}
62     float get_avg_distance(){  return _original_radius;}
63     CloudNormals& get_normals(){ return _normals;}
64     void set_octree_resolution(const float resolution){
           _octree_resolution = resolution;}
65     Octree& get_octree(){   return _octree;}
66
67     /* colors */
68     /* --- original */
69     void color_entire_original(int r, int g, int b);
70     void color_by_normal(Cloud &coloredCloud);
71     void color_by_type(Cloud &coloredCloud);
72     void color_by_curvature(Cloud &coloredCloud);
73     /* --- sample */
74     void color_edge_points(Cloud &coloredCloud);
75     void color_by_relative_density(Cloud &coloredCloud);
76
77     /* noise and outliers */
```

```
78      void add_noise(float noise_std);//add noise to the original point
            cloud based on normal (Gaussian) distribution
79      void add_outliers(int num);
80      void reset();
81      void remove_outliers_LDOF(int K = 10, int szSmallClusterOutliers =
            10, float outlier_percent = 0.07f, bool just_color = false);
82      void remove_outliers_DDF(int K = 10, int szSmallClusterOutliers = 10,
             float omega = 1.25f, bool just_color = false);
83      void remove_outliers_NNR(int K = 10, int szSmallClusterOutliers = 10,
             float outlier_percent = 0.07f, bool just_color = false);
84      void remove_outliers_combined(int K = 10, int szSmallClusterOutliers
            = 10, float outlier_percent = 0.07f, bool just_color = false);
85      void remove_outliers_voxel();
86
87      void build_octree(float resolution);                    //build octree
            for _cloud
88      void octree_neighbors();
89
90      /* point cloud simplification and resampling */
91      Cloud& get_sample(){    return _sample; }
92      CloudNormals& get_sample_normals(){ return _sample_normals; }
93
94      /* curvature calculated by k = 1 / r = 2 * lambda_0 / (miu ^ 2)
95      where lambda_0 is the eigen value pointing at normal direction
96      and miu is the search radius OR by k = lambda_0 / (lambda_0 +
            lambda_1 + lambda_2) */
97      void curvature_estimation(float search_radius);
98
99      void curvature_estimation_octree(float search_radius);
100
101     /* subsample while keep feature points */
102     void subsample(float sample_radius, float curvature_threshold, int
            num_iter, float miu);
103
104     /* With the same preprocessing as subsample() but give an extreme non
            -uniform initial subset for testing the WLOP effect */
105     void paranoid_subsample(float sample_radius, float
            curvature_threshold, float smooth_sigma_n);
106
107     void WLOP_non_features(float sample_radius, int num_iter, float miu);
108     void color_sample_from_original();
109
110 private:
111     /* basic */
112     void change_cloud(const Cloud &cloud, const std::vector<bool> &
            vec_is_aerial);
113     Cloud _cloud;
114     std::vector<bool> _vec_isaerial;
115     CloudNormals _normals;
116     KdtreePtr _kdtreePtr;
117     int _K_search;
118     float _original_radius;
119     Cloud _sample;
```

```
120     float _sample_radius;
121     int _num_edge_points_in_sample;    /* put the subsampled edge points
           at the beginning of the _sample vector
122        just keep record of the number of edge points */
123     std::vector<bool> _vec_sample_isaerial;
124     KdtreePtr _sample_kdtree;
125     CloudNormals _sample_normals;
126
127     Octree _octree;//maybe original point cloud doesn't need octree
           structure -> only sample need for hole filling
128     float _octree_resolution;
129 //    OctreeGPU _octree_gpu;
130
131     /* resampling */
132     float calc_theta(float dist, float radius_search);
133     float calc_normal_weight(float dot_product_n1_n2, float
           dot_product_n1_p12, float sigma_n);
134     float calc_phi(Normal ni, Point pi, Point pj, float original_radius,
           float sigma_n);
135     void calc_statistics(const Cloud &cloud, const KdtreePtr &tree, int
           K_search,
136                          std::vector<float> &vec_density, float &
                              std_density, float &avg_density, float &
                              min_density, float &max_density);
137     void WLOP_features(const Cloud &orig, const KdtreePtr &tree_orig,
           Cloud &sample, KdtreePtr &tree_sample,
138                        float sample_radius, int num_iter, float miu);
139
140     /* density members */
141     float _avg_density;
142     float _min_density;
143     float _max_density;
144     float _std_density;
145     float _avg_density_aerial;
146     float _avg_density_street;
147     int _num_aerial;
148     int _num_street;
149     std::vector<float> _vec_relative_density;
150
151
152     /* assistant functions */
153     inline float dot_product(Normal vec1, Normal vec2)
154     {
155         return (vec1.normal_x * vec2.normal_x + vec1.normal_y * vec2.
               normal_y + vec1.normal_z * vec2.normal_z);
156     }
157
158     inline float dot_product(Normal n, Point dp)
159     {
160         float scalar_dp = std::sqrt(dp.x * dp.x + dp.y * dp.y + dp.z * dp
               .z);
161         return (n.normal_x * dp.x + n.normal_y * dp.y + n.normal_z * dp.z
               ) / scalar_dp;
```

```
162     }
163 };
164
165 #endif // POINTCLOUDCLEANER_H
```

A-2 Source file of class *PointCloudCleaner* (part)

```
1  #include "pointcloudcleaner.h"
2
3  /* Add new point cloud if no point cloud existing yet
4     compute the density weight for each point */
5  void PointCloudCleaner::add_cloud(const Cloud &cloud, bool is_aerial, int
       K)
6  {
7      _K_search = K;
8      std::vector<float> vec_density;
9      KdtreePtr tempTree(new pcl::search::KdTree<Point>());
10     //temp tree for calculating relative density
11     tempTree->setInputCloud(cloud.makeShared());
12     int starttime, endtime;
13     starttime = std::clock();
14     float std_density, avg_density, min_density, max_density;
15     calc_statistics(cloud, tempTree, K, vec_density, std_density,
           avg_density, min_density, max_density);
16     if (is_aerial)
17     {
18         _num_aerial += cloud.size();
19         _avg_density_aerial = avg_density;
20     }
21     else
22     {
23         _num_street += cloud.size();
24         _avg_density_street = avg_density;
25     }
26
27     //density ratio
28     for (int i = 0; i != cloud.size(); i++)
29     {
30         float ratio = avg_density / vec_density[i];
31 //        float ratio = vec_density[i] / avg_density;
32         ratio = std::exp(- ratio); // scale to (0, 1] for WLOP
33         _vec_relative_density.push_back(ratio);
34     }
35     endtime = std::clock();
36     std::cout << "\n-------- Calculate weight value --------\n"
37               << "Time used: " << (endtime - starttime) / float(
                      CLOCKS_PER_SEC)
38               << " seconds\n"
39               << "Std density\t" << std_density << "\n"
40               << "Average density\t" << avg_density << "\n"
41               << "Minimum density\t" << min_density << "\n"
42               << "Maximum density\t" << max_density << "\n";
```

```
43
44      if (_cloud.size() > 0)
45      {
46          std::cout << "\n-------- Merge point clouds --------\n";
47          starttime = std::clock();
48          _vec_isaerial.resize(_cloud.size() + cloud.size(), is_aerial);
49
50          // merge two point clouds
51          _cloud += cloud;
52          _kdtreePtr->setInputCloud(_cloud.makeShared());
53          calc_statistics(_cloud, _kdtreePtr, K, vec_density, std_density,
                avg_density, min_density, max_density);
54          endtime = std::clock();
55          std::cout << "Time used: " << (endtime - starttime) / float(
                CLOCKS_PER_SEC)
56                    << " seconds\n";
57      }
58      else
59      {
60          _cloud = cloud;
61          _kdtreePtr = tempTree;
62  //          build_original_kdtree(K);
63          _vec_isaerial.resize(cloud.size(), is_aerial);
64      }
65  //    build_octree();
66      this->_original_radius = std::sqrt(K / (_PI * avg_density));
67      this->_avg_density = avg_density;
68      this->_min_density = min_density;
69      this->_max_density = max_density;
70      this->_std_density = std_density;
71  }
72
73  /* Combined LDOF and NNR method
74     for details please refer to the paper */
75  void PointCloudCleaner::remove_outliers_combined(int K, int
        szSmallClusterOutliers, float outlier_percent, bool just_color)
76  {
77      std::cout << "\n-------- NNR + LDOF outlier reduction --------\n";
78      int startTime, endTime;
79      startTime = std::clock();
80
81      std::vector<std::vector<int> > neighborVec;
82      std::vector<std::pair<int, float> > vec_pair_LDOF;
83      std::vector<int> tempNeighbors;
84      std::vector<float> tempDistances;
85
86      int szAll = K + szSmallClusterOutliers;
87      // KNN search
88      for (int i = 0; i != _cloud.size(); i++)
89      {
90          tempNeighbors.clear();
91          tempDistances.clear();
```

```
92          _kdtreePtr->nearestKSearch(_cloud.points[i], szAll, tempNeighbors
                , tempDistances);
93          std::vector<int> kNeighbors;
94          std::vector<float> kDistances;
95          float d, LDOF,D;
96          d = LDOF = D = 0.0f;
97          kNeighbors.clear();
98          kDistances.clear();
99          for (int j = szSmallClusterOutliers; j != szAll; j++)
100         {
101             kNeighbors.push_back(tempNeighbors[j]);
102             kDistances.push_back(tempDistances[j]);
103         }
104
105         for (int j = 0; j != kNeighbors.size(); j++)
106         {
107             d += kDistances[j];
108             int j_index = kNeighbors[j];
109             for (int m = j + 1; m != kNeighbors.size(); m++)
110             {
111                 int m_index = kNeighbors[m];
112                 float dx = _cloud.points[j_index].x -  _cloud.points[
                        m_index].x;
113                 float dy = _cloud.points[j_index].y -  _cloud.points[
                        m_index].y;
114                 float dz = _cloud.points[j_index].z -  _cloud.points[
                        m_index].z;
115                 D += 2.0 * std::sqrt(dx * dx + dy * dy + dz * dz);
116             }
117         }
118         d /= K;
119         D /= K * (K - 1);
120         LDOF = d / D;
121
122         std::pair<int, float> pair_LDOF(i, LDOF);
123         vec_pair_LDOF.push_back(pair_LDOF);
124         neighborVec.push_back(kNeighbors);
125     }
126
127     //LDOF constraint
128     std::sort(vec_pair_LDOF.begin(),vec_pair_LDOF.end(),
129               boost::bind(&std::pair<int, float>::second, _1) >
130               boost::bind(&std::pair<int, float>::second, _2));
131     int outlier_size = (int)(outlier_percent * _cloud.size());
132     std::set<int> LDOF_outlier_indices;
133     for (int i = 0; i != outlier_size; i++)
134     {
135         LDOF_outlier_indices.insert(vec_pair_LDOF[i].first);
136     }
137
138     pcl::PointCloud<pcl::PointXYZRGB>::Ptr resultCloud (new pcl::
            PointCloud<pcl::PointXYZRGB>);
139     std::vector<bool> is_aerial;
```

```
140     std::vector<float> vec_density;
141     uint32_t rgb = (static_cast<uint32_t>(255) << 16 |
142                     static_cast<uint32_t>(0) << 8 | static_cast<uint32_t
                            >(0));
143     // unidirectional neighbors search
144     for (int i = 0; i != _cloud.size(); i++)
145     {
146         int unidirectional_neighbors = 0;
147         for (int j = 0; j != K; j++)
148         {
149             int nb = neighborVec[i][j];
150             bool p_in_Nq = (std::find(neighborVec[nb].begin(),neighborVec
                    [nb].end(),i) != neighborVec[nb].end());
151             if (!p_in_Nq)
152                 unidirectional_neighbors += 1;
153         }
154         float percent = 1.0f - unidirectional_neighbors / K;
155         if (just_color)
156         {
157             //1. for changing color only
158             if (percent <= outlier_percent)
159             {
160                 bool is_in_LDOF_outliers = (LDOF_outlier_indices.find(i)
                        != LDOF_outlier_indices.end());
161                 if (is_in_LDOF_outliers)
162                 {
163                     _cloud.points[i].rgb = *reinterpret_cast<float*>(&rgb
                            );
164                 }
165             }
166         }
167         else
168         {
169             //2. change the cloud
170             if (percent > outlier_percent)
171             {
172                 bool is_in_LDOF_outliers = (LDOF_outlier_indices.find(i)
                        != LDOF_outlier_indices.end());
173                 if (!is_in_LDOF_outliers)
174                 {
175                     resultCloud->push_back(_cloud[i]);
176                     is_aerial.push_back(_vec_isaerial[i]);
177                     vec_density.push_back(_vec_relative_density[i]);
178                 }
179             }
180         }
181     }
182     endTime = std::clock();
183     std::cout << "Time used: "
184               << (endTime - startTime) / float(CLOCKS_PER_SEC)
185               << " seconds" << std::endl;
186     if (false == just_color)
187     {
```

```
188         change_cloud(*resultCloud, is_aerial);
189         _kdtreePtr->setInputCloud(_cloud.makeShared());
190 //        build_original_kdtree(_K_search);
191         _vec_relative_density = vec_density;
192     }
193 }
194
195 /* spatial kernel */
196 float PointCloudCleaner::calc_theta(float dist, float radius_search)
197 {
198     float result = std::exp(- dist * dist / (radius_search *
        radius_search));
199     return result;
200 }
201
202 /* curvature calculated by k = 1 / r = 2 * lambda_0 / (miu ^ 2)
203 where lambda_0 is the eigen value pointing at normal direction
204 and miu is the search radius OR by k = lambda_0 / (lambda_0 + lambda_1 +
    lambda_2) */
205 void PointCloudCleaner::curvature_estimation(float search_radius)
206 {
207     std::cout << "\n-------- Estimation curvature of the original point
        cloud --------\n";
208     int starttime, endtime;
209     starttime = std::clock();
210     _kdtreePtr->setSortedResults(false);// to speed up radius search
211     float factor = 2.0f / (search_radius * search_radius);
212     for (int i = 0; i != _cloud.size(); i++)
213     {
214         std::vector<int> indices;
215         std::vector<float> sqr_distances;
216         _kdtreePtr->radiusSearch(_cloud[i], search_radius, indices,
            sqr_distances);
217 //        _kdtreePtr->nearestKSearch(_cloud[i], _K_search, indices,
    sqr_distances);
218
219         Eigen::Matrix3f covariance_matrix;
220         Eigen::Vector4f xyz_centroid;
221
222         Normal n;
223         if (indices.size() < 3 || pcl::computeMeanAndCovarianceMatrix(
            _cloud, indices, covariance_matrix, xyz_centroid) == 0)
224         {
225             /* invalid neighbor size */
226             n.normal_x = n.normal_y = n.normal_z = n.curvature = std::
                numeric_limits<float>::quiet_NaN();
227             continue;
228         }
229
230 //        Eigen::Vector3f eigen_values;
231 //        Eigen::Matrix3f eigen_vectors;
232 //        pcl::eigen33(covariance_matrix, eigen_vectors, eigen_values);
```

```
233 //        float curvature_i = 2.0f * std::fabs(eigen_values[0]) /
    sqr_distances[_K_search - 1];
234         Eigen::Matrix3f::Scalar eigen_value;
235         Eigen::Vector3f eigen_vector;
236         pcl::eigen33(covariance_matrix, eigen_value, eigen_vector);
237 
238         n.normal_x = eigen_vector[0];
239         n.normal_y = eigen_vector[1];
240         n.normal_z = eigen_vector[2];
241 
242         /* method 1: Surface Variation from Pauly 2002 */
243 //        float eig_sum = covariance_matrix.coeff(0) + covariance_matrix.
    coeff(4) + covariance_matrix.coeff(8);
244 //        if (eig_sum != 0)
245 //            n.curvature = std::fabs(eigen_value / eig_sum);
246 //        else
247 //            n.curvature = 0.0;
248 
249         /* method 2: curvature from Gumhold 2001 */
250 //        n.curvature = 2.0f * std::fabs(eigen_value) / sqr_distances[
    _K_search - 1];
251         n.curvature = factor * std::fabs(eigen_value);
252         _normals.push_back(n);
253     }
254 
255     _kdtreePtr->setSortedResults(true);//turn on again
256     endtime = std::clock();
257     std::cout << "Time used: " << (endtime - starttime) / float(
        CLOCKS_PER_SEC)
258               << " seconds\n";
259 }
260 
261 void PointCloudCleaner::curvature_estimation_octree(float search_radius)
262 {
263     std::cout << "\n-------- Estimation curvature of the original point
        cloud --------\n";
264     int starttime, endtime;
265     starttime = std::clock();
266 
267     float factor = 2.0f / (search_radius * search_radius);
268     /* flag curvature estimation state */
269     std::vector<bool> flag;
270     flag.resize(_cloud.size(), false);
271     std::vector<int> indices;
272 
273     /* store curvature in _normals. Normal is one by-product */
274     _normals.clear();
275     Normal n;
276     n.normal_x = n.normal_y = n.normal_z = n.curvature = std::
        numeric_limits<float>::quiet_NaN();
277     for (int i = 0; i != _cloud.size(); i++)
278     {
279         _normals.push_back(n);
```

```
280     }
281
282     for (int i = 0; i != _cloud.size(); i++)
283     {
284         indices.clear();
285         if (!flag[i])
286         {
287             _octree.voxelSearch(i, indices);
288
289             /* covariance matrix */
290             Eigen::Matrix3f covariance_matrix;
291             Eigen::Vector4f xyz_centroid;
292             if (indices.size() < 3 || pcl::computeMeanAndCovarianceMatrix
                    (_cloud, indices, covariance_matrix, xyz_centroid) == 0)
293             {
294                 /* invalid neighbor size */
295                 for (int j = 0; j != indices.size(); j++)
296                 {
297                     flag[indices[j]] = true;
298                 }
299                 continue;
300             }
301
302             /* eigen values and eigen vectors */
303             Eigen::Matrix3f::Scalar eigen_value;
304             Eigen::Vector3f eigen_vector;
305             pcl::eigen33(covariance_matrix, eigen_value, eigen_vector);
306
307             n.normal_x = eigen_vector[0];
308             n.normal_y = eigen_vector[1];
309             n.normal_z = eigen_vector[2];
310
311             /* method 1: Surface Variation from Pauly 2002 */
312     //        float eig_sum = covariance_matrix.coeff(0) +
        covariance_matrix.coeff(4) + covariance_matrix.coeff(8);
313     //        if (eig_sum != 0)
314     //            n.curvature = std::fabs(eigen_value / eig_sum);
315     //        else
316     //            n.curvature = 0.0;
317
318             /* method 2: curvature from Gumhold 2001 */
319     //        n.curvature = 2.0f * std::fabs(eigen_value) / sqr_distances
        [_K_search - 1];
320             n.curvature = factor * std::fabs(eigen_value);
321             for (int j = 0; j != indices.size(); j++)
322             {
323                 int idx = indices[j];
324                 flag[idx] = true;
325                 _normals[idx] = n;
326             }
327         }
328     }
329
```

```
330     endtime = std::clock();
331     std::cout << "Time used: " << (endtime - starttime) / float(
            CLOCKS_PER_SEC)
332               << " seconds\n";
333 }
334
335 //#include <pcl/io/pcd_io.h>
336 /* subsample the points starting from feature points and smoothed by WLOP
        */
337 void PointCloudCleaner::subsample(float sample_radius, float
        curvature_threshold, int num_iter, float miu)
338 {
339     if (_normals.size() == 0)
340     {
341         std::cout << "Error: curvature not estimated yet!\n";
342         return;
343     }
344
345     std::cout << "\n-------- Subsample point cloud while keeping feature
            points --------\n";
346     int starttime = std::clock();
347
348     /* flag each original point about the state of whether finishing
            subsampling */
349     std::vector<bool> sample_flag;
350     sample_flag.resize(_cloud.size(), false);
351     Cloud extracted_feature_points;
352     CloudNormals extracted_feature_normals;
353     std::vector<int> extracted_indices;
354
355     for (int i = 0; i != _normals.size(); i++)
356     {
357         if (std::isnan(_normals[i].curvature))
358             continue;
359         if (curvature_threshold < _normals[i].curvature)
360         {
361             extracted_indices.push_back(i);
362             extracted_feature_points.push_back(_cloud[i]);
363             extracted_feature_normals.push_back(_normals[i]);
364         }
365     }
366 //    /* tmp write the extracted_feature_points into a pcd file in /tmp
        */
367 //    if (extracted_feature_points.size() > 0)
368 //        pcl::io::savePCDFileBinaryCompressed("/tmp/features.pcd",
        extracted_feature_points);
369
370     int endtime1 = std::clock();
371     std::cout << "\tFeature extraction\tTime used: "
372               << (endtime1 - starttime) / float(CLOCKS_PER_SEC)
373               << " seconds\n";
374
375     /* simple subsample the feature points */
```

```
376     KdtreePtr curvature_orig_tree(new pcl::search::KdTree<Point>());
377     curvature_orig_tree->setSortedResults(false);
378     curvature_orig_tree->setInputCloud(extracted_feature_points.
            makeShared());
379
380     Cloud subsampled_feature_points;
381     std::vector<bool> subsample_flag;
382     std::vector<int> indices;
383     std::vector<float> sqr_distances;
384     subsample_flag.resize(extracted_feature_points.size(), false);
385     for (int i = 0; i != extracted_feature_points.size(); i++)
386     {
387         if (!subsample_flag[i])
388         {
389             subsample_flag[i] = true;
390
391             indices.clear();
392             sqr_distances.clear();
393             curvature_orig_tree->radiusSearch(extracted_feature_points[i
                    ], sample_radius, indices, sqr_distances);
394             if (indices.size() > 1) // at least 1 neighbor otherwise
                    consider as outlier in features
395             {
396                 for (int j = 0; j != indices.size(); j++)
397                 {
398                     subsample_flag[indices[j]] = true;
399                 }
400                 subsampled_feature_points.push_back(
                        extracted_feature_points[i]);
401
402                 /* flag in the original point cloud this point has been
                        sampled */
403                 int index_in_orig = extracted_indices[i];
404                 sample_flag[index_in_orig] = true;
405             }
406         }
407     }
408
409     KdtreePtr subsample_tree(new pcl::search::KdTree<Point>());
410     subsample_tree->setSortedResults(false);
411     subsample_tree->setInputCloud(subsampled_feature_points.makeShared())
            ;
412
413     /* Remove salient points in the subsample */
414     Cloud changed_subsample;
415     float search_radius = 1.5f * sample_radius;
416     for (int i = 0; i != subsampled_feature_points.size(); i++)
417     {
418         indices.clear();
419         sqr_distances.clear();
420         subsample_tree->radiusSearch(subsampled_feature_points[i],
                search_radius, indices, sqr_distances);
421         if (indices.size() > 1) // at least one neighbor needed
```

```
422             changed_subsample.push_back(subsampled_feature_points[i]);
423     }
424
425     subsampled_feature_points = changed_subsample;
426     subsample_tree->setInputCloud(subsampled_feature_points.makeShared())
            ;
427 //    /* tmp write the subsampled feature points into a pcd file in /tmp
        */
428 //    if (subsampled_feature_points.size() > 0)
429 //        pcl::io::savePCDFileBinaryCompressed("/tmp/subsample.pcd",
        subsampled_feature_points);
430
431     int endtime2 = std::clock();
432     std::cout << "\tSubsample feature points\tTime used: "
433               << (endtime2 - endtime1) / float(CLOCKS_PER_SEC)
434               << " seconds\n";
435
436     /* WLOP smooth the subsampled feature points */
437     WLOP_features(extracted_feature_points, curvature_orig_tree,
            subsampled_feature_points, subsample_tree,
438                   sample_radius, num_iter, miu);
439
440     int endtime3 = std::clock();
441     std::cout << "\tFeature smoothing\tTime used: "
442               << (endtime3 - endtime2) / float(CLOCKS_PER_SEC)
443               << " seconds\n";
444 //    /* tmp write the extracted_feature_points into a pcd file in /tmp
        */
445 //    if (subsampled_feature_points.size() > 0)
446 //        pcl::io::savePCDFileBinaryCompressed("/tmp/smoothed.pcd",
        subsampled_feature_points);
447
448     extracted_feature_points.clear();
449     extracted_feature_normals.clear();
450
451     /* flag points < sample_radius in the original point cloud as sampled
            */
452     for (int i = 0; i != subsampled_feature_points.size(); i++)
453     {
454         indices.clear();
455         sqr_distances.clear();
456         _kdtreePtr->radiusSearch(subsampled_feature_points[i],
                sample_radius, indices, sqr_distances);
457         for (int j = 0; j != indices.size(); j++)
458         {
459             sample_flag[indices[j]] = true;
460         }
461     }
462     _sample = subsampled_feature_points;
463     _num_edge_points_in_sample = subsampled_feature_points.size();
464
465     /* do simple subsampling for the other points in original point cloud
            */
```

```
466     _kdtreePtr->setSortedResults(false);
467     for (int i = 0; i != _cloud.size(); i++)
468     {
469         if (!sample_flag[i])
470         {
471             sample_flag[i] = true;
472             indices.clear();
473             sqr_distances.clear();
474             _kdtreePtr->radiusSearch(_cloud[i], sample_radius, indices,
                    sqr_distances);
475             if (indices.size() > 1)
476             {
477                 for (int j = 0; j != indices.size(); j++)
478                 {
479                     sample_flag[indices[j]] = true;
480                 }
481                 _sample.push_back(_cloud[i]);
482             }
483         }
484     }
485     sample_flag.clear();
486     int endtime4 = std::clock();
487     std::cout << "\tSubsampling other points\tTime used: "
488               << (endtime4 - endtime3) / float(CLOCKS_PER_SEC)
489               << " seconds\n";
490
491     std::cout << "Total time used: "
492               << (endtime4 - starttime) / float(CLOCKS_PER_SEC)
493               << " seconds\n";
494
495     std::cout << "\n-------- Build sample tree and output subsampling
            statistics --------\n";
496     starttime = std::clock();
497     _sample_kdtree->setInputCloud(_sample.makeShared());
498
499     /* output statistics */
500     float std_density, avg_density, min_density, max_density;
501     std::vector<float> vec_density;
502     calc_statistics(_sample, _sample_kdtree, _K_search, vec_density,
            std_density, avg_density, min_density, max_density);
503     int endtime = std::clock();
504     std::cout << "Time used: "
505               << (endtime - starttime) / float(CLOCKS_PER_SEC)
506               << " seconds\n"
507               << "\nStatistics after subsampling: \n"
508               << "Std density\t" << std_density << "\n"
509               << "Average density\t" << avg_density << "\n"
510               << "Minimum density\t" << min_density << "\n"
511               << "Maximum density\t" << max_density << "\n";
512 }
513
514 void PointCloudCleaner::WLOP_features(const Cloud &orig, const KdtreePtr
        &tree_orig, Cloud &sample, KdtreePtr &tree_sample,
```

```
515                     float sample_radius, int num_iter, float miu)
516 {
517     /* calculate original and sample point cloud WLOP weight value */
518 //    std::cout << "\n-------- WLOP weight calculation -------- \n";
519     std::vector<int> indices;
520     std::vector<float> sqr_distances;
521     std::vector<float> vec_original_v;
522
523     tree_orig->setSortedResults(false);
524     tree_sample->setSortedResults(false);
525
526     /* weight value v in original point cloud */
527     float v = 1.0f;
528     for (int i = 0; i != orig.size(); i++)
529     {
530         v = 1.0f;
531         //use theta kernel (no sharp edge consideration)
532         indices.clear();
533         sqr_distances.clear();
534         tree_orig->radiusSearch(orig[i], sample_radius, indices,
                sqr_distances);
535
536         for (int j = 0; j != indices.size(); j++)
537         {
538             if (sqr_distances[j] < 1e-6) continue; // ignore points too
                    close
539             float theta = calc_theta(std::sqrt(sqr_distances[j]),
                    sample_radius);
540             v += theta;//
541         }
542         vec_original_v.push_back(v);
543     }
544
545 //    starttime = std::clock();
546     float sample_search_radius = 3.0f * sample_radius;
547     for (int i = 0; i != num_iter; i++)
548     {
549         /* The original_radius is limited only to neighbors for fast
                computation.
550         Theoretically the whole point cloud should be included in
                calculation*/
551         Cloud changed_sample_cloud;
552         Point average_term, repulsion_term;
553         float sum_alfa_v = 0.0f;
554
555         /* weight value of each sample point in repulsion term */
556         /* it is changed in each iteration */
557         std::vector<float> vec_sample_w;
558
559         for (int i = 0; i != sample.size(); i++)
560         {
561             float w = 1.0f;
562             indices.clear();
```

```
563             sqr_distances.clear();
564             tree_sample->radiusSearch(sample[i], sample_search_radius,
                    indices, sqr_distances);
565             for (int j = 0; j != indices.size(); j++)
566             {
567                 if (sqr_distances[j] < 1e-6)
568                     continue;
569                 w += calc_theta(std::sqrt(sqr_distances[j]),
                        sample_search_radius);
570             }
571             vec_sample_w.push_back(w);
572         }
573 
574         /* WLOP smooth feature points */
575         for (int i = 0; i != sample.size(); i++)
576         {
577             average_term.x = average_term.y = average_term.z = 0.0f;
578             repulsion_term.x = repulsion_term.y = repulsion_term.z = 0.0f
                    ;
579             average_term.rgb = repulsion_term.rgb = sample[i].rgb;
580             /// ----------------------
581             /// calculate average term
582             /// ----------------------
583             sum_alfa_v = 0.0f;
584             std::vector<float> vec_alfa_v;
585             std::vector<int> neighbors;
586             std::vector<float> sqr_distances;
587             tree_orig->radiusSearch(sample[i], sample_radius, neighbors,
                    sqr_distances);
588             /* term: alfa / v and SIGMA(alfa / v) */
589             for (int j = 0; j != neighbors.size(); j++)
590             {
591                 if (sqr_distances[j] < 1e-6)
592                 {
593                     //BUG HERE!!! VECTOR ITEM NOT MATCH
594                     //NEED TO PUSH A NAN VALUE TO VECTOR
595                     vec_alfa_v.push_back(std::numeric_limits<float>::
                            quiet_NaN());
596                     continue;
597                 }
598                 int idx = neighbors[j];
599                 float dist = std::sqrt(sqr_distances[j]);
600                 float theta = calc_theta(dist, sample_radius);
601                 float alfa_v = theta / (dist * vec_original_v[idx]);
602                 vec_alfa_v.push_back(alfa_v);
603                 sum_alfa_v += alfa_v;
604             }
605 
606             if (sum_alfa_v < _ZERO)
607             {
608                 /* no neighbor in the original point cloud -> outlier */
609 //                average_term = sample[i];
610                 continue;
```

```
611             }
612             else
613             {
614                 /* term: p_ij * (alfa_ij / v) / SIGMA(alfa / v) */
615                 for (int j = 0; j != neighbors.size(); j++)
616                 {
617                     if (sqr_distances[j] < 1e-6) continue;
618                     int idx = neighbors[j];
619                     float weight = vec_alfa_v[j] / sum_alfa_v;
620                     average_term.getVector3fMap() += orig[idx].
                            getVector3fMap() * weight;
621                 }
622             }
623 
624             vec_alfa_v.clear();
625             /// ------------------------
626             /// calculate repulsion term
627             /// ------------------------
628             std::vector<Point> vec_dp;
629             neighbors.clear();
630             sqr_distances.clear();
631             tree_sample->radiusSearch(sample[i], sample_search_radius,
                    neighbors, sqr_distances);
632 
633             /* no neighbor in given sample radius search. No need to push
                     away other points */
634             if (neighbors.size() == 0)
635             {
636                 changed_sample_cloud.push_back(average_term);
637                 continue;
638             }
639             //calculate beta
640             float sum_w_beita = 0.0f;
641             std::vector<float> vec_w_beita;
642 
643             for (int j = 0; j != neighbors.size(); j++)
644             {
645                 if (sqr_distances[j] < 1e-6)
646                 {
647                     //BUG HERE!!! VECTOR ITEM NOT MATCH
648                     //NEED TO PUSH A NAN VALUE TO VECTOR
649                     vec_w_beita.push_back(std::numeric_limits<float>::
                            quiet_NaN());
650                     vec_dp.push_back(std::numeric_limits<Point>::
                            quiet_NaN());
651                     continue;
652                 }
653                 int index = neighbors[j];
654                 Point dp;
655                 dp.getVector3fMap() = sample[i].getVector3fMap() - sample
                        [index].getVector3fMap();
656                 vec_dp.push_back(dp);
657                 float dist = std::sqrt(sqr_distances[j]);
```

```
658                 float theta = calc_theta(dist, sample_search_radius);
659                 float w_beita = theta / dist * vec_sample_w[index]; //;
660                 vec_w_beita.push_back(w_beita);
661                 sum_w_beita += w_beita;
662             }
663
664             for (int j = 0; j != neighbors.size(); j++)
665             {
666                 if (sqr_distances[j] < 1e-6) continue;
667                 float weight = miu * vec_w_beita[j] / sum_w_beita;
668                 repulsion_term.getVector3fMap() += vec_dp[j].
                        getVector3fMap() * weight;
669             }
670
671             vec_w_beita.clear();
672             Point changed_pt;
673             changed_pt.getVector3fMap() = average_term.getVector3fMap() +
                    repulsion_term.getVector3fMap();
674
675             changed_pt.rgb = sample[i].rgb;
676             changed_sample_cloud.push_back(changed_pt);
677         }
678         /* After each iteration change the sample point cloud */
679         /* Could be problem here??? Better change the tree for each point
                */
680         sample = changed_sample_cloud;
681         tree_sample->setInputCloud(sample.makeShared());
682     }
683 }
684
685 /* WLOP for smoothing projected (subsampled) point cloud while keep
    uniform */
686 void PointCloudCleaner::WLOP_non_features(float sample_radius, int
    num_iter, float miu)
687 {
688     if (_avg_density < _ZERO)
689     {
690         std::cout << "ERROR: average density is 0.0 probably not computed
                yet!" << std::endl;
691         return;
692     }
693
694     /* calculate original and sample point cloud WLOP weight value */
695 //    std::cout << "\n-------- WLOP weight calculation -------- \n";
696     std::vector<int> indices;
697     std::vector<float> sqr_distances;
698     int starttime = std::clock();
699     std::cout << "\n-------- WLOP initialization -------- \n";
700     std::vector<float> vec_original_v;
701     _sample_kdtree->setSortedResults(false);
702
703     /* weight value v in original point cloud */
704     float v = 1.0f;
```

```
705     for (int i = 0; i != _cloud.size(); i++)
706     {
707         v = 1.0f;
708         //use theta kernel (no sharp edge consideration)
709         indices.clear();
710         sqr_distances.clear();
711         _octree.radiusSearch(_cloud[i], sample_radius, indices,
                sqr_distances);
712 
713         for (int j = 0; j != indices.size(); j++)
714         {
715             if (sqr_distances[j] < 1e-6) continue; // ignore points too
                    close
716             float theta = calc_theta(std::sqrt(sqr_distances[j]),
                    sample_radius);
717             v += _vec_relative_density[indices[j]] * theta;//
718         }
719         vec_original_v.push_back(v);
720     }
721 
722     int endtime1 = std::clock();
723     std::cout << "Time used: " << (endtime1 - starttime) / float(
            CLOCKS_PER_SEC)
724               << " seconds\n";
725 
726     std::cout << "\n-------- WLOP iterations -------- \n";
727     std::cout << " Number of iterations\n " << num_iter << std::endl;
728     float sample_search_radius = 2.0f * sample_radius;
729     for (int i = 0; i != num_iter; i++)
730     {
731         /* The original_radius is limited only to neighbors for fast
                computation.
732         Theoretically the whole point cloud should be included in
                calculation*/
733         Cloud changed_sample_cloud;
734         Point average_term, repulsion_term;
735         float sum_alfa_v = 0.0f;
736 
737         /* Keep the first _num_edge_points_in_sample edge points from
                WLOP */
738         changed_sample_cloud.points.assign(_sample.begin(),_sample.begin
                () + _num_edge_points_in_sample);
739 
740         /* weight value of each sample point in repulsion term */
741         /* it is changed in each iteration */
742         std::vector<float> vec_sample_w;
743         for (int i = 0; i != _sample.size(); i++)
744         {
745             float w = 1.0f;
746             indices.clear();
747             sqr_distances.clear();
748             _sample_kdtree->radiusSearch(_sample[i], sample_search_radius
                    , indices, sqr_distances);
```

```
749             for (int j = 0; j != indices.size(); j++)
750             {
751                 if (sqr_distances[j] < 1e-6) continue;
752                 w += calc_theta(std::sqrt(sqr_distances[j]),
                        sample_search_radius);
753             }
754             vec_sample_w.push_back(w);
755         }
756 
757         /* WLOP smooth the other non-feature points */
758         for (int i = _num_edge_points_in_sample; i != _sample.size(); i
                ++)
759         {
760             average_term.x = average_term.y = average_term.z = 0.0f;
761             repulsion_term.x = repulsion_term.y = repulsion_term.z = 0.0f
                    ;
762             average_term.rgb = repulsion_term.rgb = _sample[i].rgb;
763             /// ----------------------
764             /// calculate average term
765             /// ----------------------
766             sum_alfa_v = 0.0f;
767             std::vector<float> vec_alfa_v;
768             std::vector<int> neighbors;
769             std::vector<float> sqr_distances;
770             _octree.radiusSearch(_sample[i], sample_radius, neighbors,
                    sqr_distances);
771 
772             /* term: alfa / v and SIGMA(alfa / v) */
773             for (int j = 0; j != neighbors.size(); j++)
774             {
775                 if (sqr_distances[j] < 1e-6)
776                 {
777                     //BUG HERE!!! VECTOR ITEM NOT MATCH
778                     //NEED TO PUSH A NAN VALUE TO VECTOR
779                     vec_alfa_v.push_back(std::numeric_limits<float>::
                            quiet_NaN());
780                     continue;
781                 }
782                 int idx = neighbors[j];
783                 float dist = std::sqrt(sqr_distances[j]);
784                 float theta = calc_theta(dist, sample_radius);
785                 /* v -> keep the same order */
786                 float alfa_v = theta * _vec_relative_density[idx] / (dist
                        * vec_original_v[idx]);
787                 vec_alfa_v.push_back(alfa_v);
788                 sum_alfa_v += alfa_v;
789             }
790 
791             if (sum_alfa_v < _ZERO)
792             {
793 //                  average_term = _sample[i];
794                 /* no neighbor in the original point cloud -> outlier */
795                 continue;
```

```
796             }
797             else
798             {
799                 /* term: p_ij * (alfa_ij / v) / SIGMA(alfa / v) */
800                 for (int j = 0; j != neighbors.size(); j++)
801                 {
802                     if (sqr_distances[j] < 1e-6) continue;
803                     int idx = neighbors[j];
804                     float weight = vec_alfa_v[j] / sum_alfa_v;
805                     average_term.getVector3fMap() += _cloud[idx].
                            getVector3fMap() * weight;
806                 }
807             }
808 
809             vec_alfa_v.clear();
810             /// ------------------------
811             /// calculate repulsion term
812             /// ------------------------
813             std::vector<Point> vec_dp;
814             neighbors.clear();
815             sqr_distances.clear();
816             _sample_kdtree->radiusSearch(_sample[i], sample_search_radius
                    , neighbors, sqr_distances);
817 
818             /* no neighbor in given sample radius search. No need to push
                     away other points */
819             if (neighbors.size() == 0)
820             {
821                 changed_sample_cloud.push_back(average_term);
822                 continue;
823             }
824             //calculate beta
825             float sum_w_beita = 0.0f;
826             std::vector<float> vec_w_beita;
827 
828             for (int j = 0; j != neighbors.size(); j++)
829             {
830                 if (sqr_distances[j] < 1e-6)
831                 {
832                     //BUG HERE!!! VECTOR ITEM NOT MATCH
833                     //NEED TO PUSH A NAN VALUE TO VECTOR
834                     vec_w_beita.push_back(std::numeric_limits<float>::
                            quiet_NaN());
835                     vec_dp.push_back(std::numeric_limits<Point>::
                            quiet_NaN());
836                     continue;
837                 }
838                 int index = neighbors[j];
839                 Point dp;
840                 dp.getVector3fMap() = _sample[i].getVector3fMap() -
                        _sample[index].getVector3fMap();
841                 vec_dp.push_back(dp);
842                 float dist = std::sqrt(sqr_distances[j]);
```

```
843                 float theta = calc_theta(dist, sample_search_radius);
844                 float w_beita = theta / dist * vec_sample_w[index]; //;
845                 vec_w_beita.push_back(w_beita);
846                 sum_w_beita += w_beita;
847             }
848             if (sum_w_beita < _ZERO)
849             {
850                 changed_sample_cloud.push_back(average_term);
851                 continue;
852             }
853             else
854             {
855                 for (int j = 0; j != neighbors.size(); j++)
856                 {
857                     if (sqr_distances[j] < 1e-6) continue;
858                     float weight = miu * vec_w_beita[j] / sum_w_beita;
859                     repulsion_term.getVector3fMap() += vec_dp[j].
                            getVector3fMap() * weight;
860                 }
861             }
862 
863             vec_w_beita.clear();
864             Point changed_pt, diff;
865             changed_pt.getVector3fMap() = average_term.getVector3fMap() +
                    repulsion_term.getVector3fMap();
866             changed_pt.rgb = _sample[i].rgb;
867             changed_sample_cloud.push_back(changed_pt);
868         }
869         /* After each iteration change the sample point cloud */
870         /* Could be problem here??? Better change the tree for each point
                */
871         _sample = changed_sample_cloud;
872         _sample_kdtree->setInputCloud(_sample.makeShared());
873     }
874     int endtime = std::clock();
875     std::cout << "Time used: " << (endtime - starttime) / float(
            CLOCKS_PER_SEC)
876               << " seconds\n";
877 
878     /// ----------output statistics-------------
879     float std_dev, avg_density, min_density, max_density;
880     std::cout << "\n Original point cloud nearest neighbor distance
            statistics\n std_dev\t avg_density\t min_density\t max_density\n "
881               << _std_density << "\t " << _avg_density << "\t " <<
                      _min_density << "\t" << max_density;
882 
883     std::vector<float> vec_density;
884     calc_statistics(_sample, _sample_kdtree, _K_search, vec_density,
            std_dev, avg_density, min_density, max_density);
885     std::cout << "\n Sample point cloud nearest neighbor statistics\n
            std_dev\t avg_density\t min_density\t max_density\n "
886               << std_dev << "\t " << avg_density << "\t " << min_density
                      << "\t" << max_density << "\n";
```

```
887 }
888
889 /* sample point cloud colored from input original point cloud */
890 void PointCloudCleaner::color_sample_from_original()
891 {
892     int starttime, endtime;
893     starttime = std::clock();
894     for (int i = 0; i != _sample.size(); i++)
895     {
896         std::vector<int> indices;
897         std::vector<float> sqr_distances;
898         _kdtreePtr->nearestKSearch(_sample[i], 10, indices, sqr_distances
                );
899         float sum_r, sum_g, sum_b;
900         sum_r = sum_g = sum_b = 0.0f;
901         float sum_weight;
902         sum_weight = 0.0f;
903         for (int j = 0; j != indices.size(); j++)
904         {
905             int idx = indices[j];
906             sum_r += _cloud[idx].r * _vec_relative_density[idx];
907             sum_g += _cloud[idx].g * _vec_relative_density[idx];
908             sum_b += _cloud[idx].b * _vec_relative_density[idx];
909             sum_weight += _vec_relative_density[idx];
910         }
911         sum_r /= sum_weight;
912         sum_g /= sum_weight;
913         sum_b /= sum_weight;
914         uint32_t rgb = (static_cast<uint32_t>(sum_r) << 16 |
915                         static_cast<uint32_t>(sum_g) << 8 | static_cast<
                            uint32_t>(sum_b));
916         _sample[i].rgb = *reinterpret_cast<float*>(&rgb);
917     }
918     endtime = std::clock();
919     std::cout << "\n Color sample point cloud from KNN in original point
            cloud: \n"
920               << (endtime - starttime) / float(CLOCKS_PER_SEC)
921               << " seconds\n";
922 }
923
924 /* avg, min, max and stdDev of the input point cloud */
925 void PointCloudCleaner::calc_statistics(const Cloud &cloud, const
        KdtreePtr &tree, int K_search,
926                                         std::vector<float> &vec_density,
                                            float &std_density, float &
                                            avg_density, float &
                                            min_density, float &
                                            max_density)
927 {
928     vec_density.clear();
929     min_density = _BIG_FLOAT;
930     max_density = - _BIG_FLOAT;
931     avg_density = 0.0f;
```

```
932     std_density = 0.0f;
933     float factor = K_search / _PI;
934     for (int i = 0; i != cloud.size(); i++)
935     {
936         std::vector<int> indices;
937         std::vector<float> sqr_distances;
938         tree->nearestKSearch(cloud[i], K_search + 1, indices,
                sqr_distances);// exclude itself
939         float sqr_dist = sqr_distances[K_search];
940         if (sqr_dist < 1e-6)
941         {
942             sqr_dist = 1e-6;
943         }
944         float density_i = factor / sqr_dist;
945         vec_density.push_back(density_i);
946         if (density_i < min_density) min_density = density_i;
947         if (density_i > max_density) max_density = density_i;
948         avg_density += density_i;
949     }
950     avg_density = avg_density / cloud.size();
951
952     for (int i = 0; i != cloud.size(); i++)
953     {
954         std_density += std::pow(vec_density[i] - avg_density, 2);
955     }
956     std_density = std::sqrt(std_density) / cloud.size();
957 }
```

Bibliography

[1] P. K. Agarwal, L. Arge, and A. Danner, *From point cloud to grid DEM: A scalable approach*. Springer, 2006.

[2] T. Rabbani, F. van den Heuvel, and G. Vosselmann, "Segmentation of point clouds using smoothness constraint," *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. 36, no. 5, pp. 248–253, 2006. 00212.

[3] Y. Livny, F. Yan, M. Olson, B. Chen, H. Zhang, and J. El-Sana, "Automatic reconstruction of tree skeletal structures from point clouds," in *ACM Transactions on Graphics (TOG)*, vol. 29, p. 151, ACM, 2010.

[4] S. Pu, G. Vosselman, and others, "Automatic extraction of building features from terrestrial laser scanning," *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. 36, no. 5, pp. 25–27, 2006.

[5] J. B. Campbell, *Introduction to remote sensing*. CRC Press, 2002.

[6] C. V. Tao and J. Li, *Advances in mobile mapping technology*. CRC Press, 2007. 00041.

[7] J. J. Koenderink, A. J. Van Doorn, and others, "Affine structure from motion," *JOSA A*, vol. 8, no. 2, pp. 377–385, 1991. 00743.

[8] B. Triggs, P. F. McLauchlan, R. I. Hartley, and A. W. Fitzgibbon, "Bundle adjustment-a modern synthesis," in *Vision algorithms: theory and practice*, pp. 298–372, Springer, 2000. 02231.

[9] P. J. Besl and N. D. McKay, "Method for registration of 3-D shapes," in *Robotics-DL tentative*, pp. 586–606, International Society for Optics and Photonics, 1992. 10561.

[10] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle, *Surface reconstruction from unorganized points*, vol. 26. ACM, 1992. 02867.

[11] L. Kobbelt, S. Campagna, J. Vorsatz, and H.-P. Seidel, "Interactive multi-resolution modeling on arbitrary meshes," in *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pp. 105–114, ACM, 1998. 00778.

[12] G. Turk, "Texture synthesis on surfaces," in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pp. 347–354, ACM, 2001. 00339.

[13] J. Wang, K. Xu, L. Liu, J. Cao, S. Liu, Z. Yu, and X. D. Gu, "Consolidation of Low-quality Point Clouds from Outdoor Scenes," in *Computer Graphics Forum*, vol. 32, pp. 207–216, Wiley Online Library, 2013.

[14] K. Zhang, M. Hutter, and H. Jin, "A new local distance-based outlier detection approach for scattered real-world data," in *Advances in Knowledge Discovery and Data Mining*, pp. 813–822, Springer, 2009.

[15] T. Weyrich, M. Pauly, R. Keiser, S. Heinzle, S. Scandella, and M. Gross, "Post-processing of scanned 3d surface data," in *Proceedings of the First Eurographics conference on Point-Based Graphics*, pp. 85–94, Eurographics Association, 2004.

[16] H. Huang, S. Wu, M. Gong, D. Cohen-Or, U. Ascher, and H. R. Zhang, "Edge-aware point set resampling," *ACM Transactions on Graphics (TOG)*, vol. 32, no. 1, p. 9, 2013. 00013.

[17] C. Rother, V. Kolmogorov, and A. Blake, "Grabcut: Interactive foreground extraction using iterated graph cuts," in *ACM Transactions on Graphics (TOG)*, vol. 23, pp. 309–314, ACM, 2004.

[18] E. Tola, V. Lepetit, and P. Fua, "A fast local descriptor for dense matching," in *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pp. 1–8, IEEE, 2008.

[19] R. B. Rusu and S. Cousins, "3d is here: Point cloud library (pcl)," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pp. 1–4, IEEE, 2011.

[20] F. E. Grubbs, "Procedures for detecting outlying observations in samples," *Technometrics*, vol. 11, no. 1, pp. 1–21, 1969.

[21] S. Fleishman, I. Drori, and D. Cohen-Or, "Bilateral mesh denoising," in *ACM Transactions on Graphics (TOG)*, vol. 22, pp. 950–953, ACM, 2003.

[22] T. R. Jones, F. Durand, and M. Desbrun, "Non-iterative, feature-preserving mesh smoothing," in *ACM Transactions on Graphics (TOG)*, vol. 22, pp. 943–949, ACM, 2003.

[23] K. Hildebrandt and K. Polthier, "Anisotropic Filtering of Non-Linear Surface Features," in *Computer Graphics Forum*, vol. 23, pp. 391–400, Wiley Online Library, 2004.

[24] J. Digne, D. Cohen-Steiner, P. Alliez, F. De Goes, and M. Desbrun, "Feature-preserving surface reconstruction and simplification from defect-laden point sets," *Journal of mathematical imaging and vision*, vol. 48, no. 2, pp. 369–382, 2014.

[25] M. Pauly, M. Gross, and L. P. Kobbelt, "Efficient simplification of point-sampled surfaces," in *Proceedings of the conference on Visualization'02*, pp. 163–170, IEEE Computer Society, 2002.

[26] B.-Q. Shi, J. Liang, and Q. Liu, "Adaptive simplification of point cloud using k-means clustering," *Computer-Aided Design*, vol. 43, no. 8, pp. 910–922, 2011.

[27] C. Moenning and N. A. Dodgson, "Intrinsic point cloud simplification," *Proc. 14th GrahiCon*, vol. 14, 2004.

[28] C. Moenning and N. A. Dodgson, "A new point cloud simplification algorithm," in *Proc. Int. Conf. on Visualization, Imaging and Image Processing*, pp. 1027–1033, 2003.

[29] P. Lancaster and K. Salkauskas, "Surfaces generated by moving least squares methods," *Mathematics of computation*, vol. 37, no. 155, pp. 141–158, 1981.

[30] S. Fleishman, D. Cohen-Or, and C. T. Silva, "Robust moving least-squares fitting with sharp features," in *ACM Transactions on Graphics (TOG)*, vol. 24, pp. 544–552, ACM, 2005.

[31] C. Shen, J. F. O'Brien, and J. R. Shewchuk, "Interpolating and approximating implicit surfaces from polygon soup," in *ACM Siggraph 2005 Courses*, p. 204, ACM, 2005.

[32] R. Kolluri, "Provably good moving least squares," *ACM Transactions on Algorithms (TALG)*, vol. 4, no. 2, p. 18, 2008.

[33] A. C. Öztireli, G. Guennebaud, and M. Gross, "Feature Preserving Point Set Surfaces based on Non-Linear Kernel Regression," in *Computer Graphics Forum*, vol. 28, pp. 493–501, Wiley Online Library, 2009.

[34] H. Huang, D. Li, H. Zhang, U. Ascher, and D. Cohen-Or, "Consolidation of unorganized point clouds for surface reconstruction," in *ACM Transactions on Graphics (TOG)*, vol. 28, p. 176, ACM, 2009.

[35] Y. Lipman, D. Cohen-Or, D. Levin, and H. Tal-Ezer, "Parameterization-free projection for geometry reconstruction," *ACM Transactions on Graphics (TOG)*, vol. 26, no. 3, p. 22, 2007.

[36] N. Salman, M. Yvinec, and Q. Merigot, "Feature preserving mesh generation from 3d point clouds," in *Computer graphics forum*, vol. 29, pp. 1623–1632, Wiley Online Library, 2010.

[37] H. Avron, A. Sharf, C. Greif, and D. Cohen-Or, "l1-Sparse reconstruction of sharp point set surfaces," *ACM Transactions on Graphics (TOG)*, vol. 29, no. 5, p. 135, 2010.

[38] N. Haala, M. Peter, J. Kremer, and G. Hunter, "Mobile LiDAR mapping for 3d point cloud collection in urban areas-a performance test," *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. 37, pp. 1119–1127, 2008.

[39] C. Fröhlich and M. Mettenleiter, "Terrestrial laser scanning-new perspectives in 3d surveying," *International archives of photogrammetry, remote sensing and spatial information sciences*, vol. 36, no. 8, p. W2, 2004.

[40] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International journal of computer vision*, vol. 60, no. 2, pp. 91–110, 2004.

[41] E. Rosten, R. Porter, and T. Drummond, "Faster and better: A machine learning approach to corner detection," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 32, no. 1, pp. 105–119, 2010.

[42] C. v. d. Sande, S. Soudarissanane, and K. Khoshelham, "Assessment of relative accuracy of AHN-2 laser scanning data using planar features," *Sensors*, vol. 10, no. 9, pp. 8198–8214, 2010.

[43] A. S. f. Photogrammetry and R. S. (ASPRS), "LAS Specification Version 1.4-R13," American Society for Photogrammetry and Remote Sensing Bethesda, Maryland, 2013.

[44] A. Jalobeanu and G. Gonçalves, "The Unknown Spatial Quality of Dense Point Clouds Derived From Stereo Images," 2014.

[45] S. Gumhold, X. Wang, and R. Macleod, "Feature Extraction from Point Clouds," in *In Proceedings of the 10 th International Meshing Roundtable*, 2001.

Glossary

List of Acronyms

AHN2	National Height model of the Netherlands version 2
BA	Bundle Adjustment
DDF	Distance-based Deviation Factor
EAR	Edge-Aware point set Resampling
GPS	Global Positioning System
ICP	Iterative Closest Point
IMLS	Implicit Moving Least Squares
INS	Inertial Navigation System
KNN	K-Nearest Neighbor
LDOF	Local Distance-based Outlier Factor
LiDAR	Light Detection and Ranging
LKR	Local Kernel Regression
MLS	Moving Least Squares
NNR	Nearest Neighbor Reciprocity
PCA	Principle Component Analysis
RIMLS	Robust Implicit Moving Least Squares
SfM	Structure from Motion
WLOP	Weighted Locally Optimal Projector