EPITA–École Pour l'Informatique et les Techniques Avancées
CSI–Calcul Scientifique et Image
LRDE–Laboratoire de Recherche et Développement de l'EPITA

Improving point cloud support of *ContextCapture*^{TM} : Scan Finder, Point Cloud Visibility and Point Cloud Compression

Alexandre Gbaguidi Aïsse

Supervised by Cyril Novel

Submitted in part fulfilment of the requirements for the degree of
Software Engineer of EPITA, Paris, August 2018.

Abstract

Text of the Abstract.

Acknowledgements

I would like to express (whatever feelings I have) to:

- My supervisor
- My second supervisor
- Other researchers
- My family and friends

Dedication

Dedication here.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The need of 3D realistic models is increasingly present in several fields such as architecture, digital simulation or civil and structural engineering. One way to obtain a 3D model might be to build it by hand using specialized modeling softwares. In this case, the realistic aspect of the model could be doubtful. A more reliable way is to use *photogrammetry* or *surface reconstruction* or both at the same time. *ContextCapture*^{TM} is a reality modeling software that can produce highly detailed 3D models. It creates models of all types or scales from simple photographs of a scene to point clouds or both of them thanks to its hybrid processing. The usage of point clouds gained wide popularity because of the emergence of devices such as optical laser-based range scanners, structured light scanners, LiDAR scanners, Microsoft Kinect, etc. Actually, it is only in the past two years that *ContextCapture*^{TM} has started to support point clouds and the common goal between each part of this internship is to improve *ContextCapture*^{TM} point cloud support.

The general problem being solved by *surface reconstruction* is: given a point cloud P assuming to lie near an unknown shape S, construct a digital representation D approximating S. In order to reconstruct point clouds acquired from static[1] LiDAR scanners, possibly multi-scan[2], *ContextCapture*^{TM} needs to know the position of each scanner in the scene, on the one hand, and the attribution of each point to a scanner on the other. The scanners location information is not always present in point clouds metadatas, for instance LAS file format does not provide it. Moreover, some users of *ContextCapture*^{TM} sometimes lost this information due to a prior export with no metadata. Detecting the positions of multiple scanners exclusively from a point cloud is a subject not identified as interesting by academics, and thus not tackled since this information is almost always available from the outset. The same holds true on the industry side, it is only recently that scanner position information is relevant for a few applications like *ContextCapture*^{TM} . This is the main contribution presented in this internship report: a method able to detect automatically multiple scanners in a single point cloud without prior

[1] As opposed to mobile LiDAR scanners.
[2] Point cloud made of several laser scans.

knowledge of the scene.

Knowing scanners position is one step toward supporting LAS point cloud format. *ContextCapture*^{TM} still needs to know for each point which scanner sees it best[3]. This enters into the realm of visibility of point clouds. One way to retrieve visibility of point clouds is to reconstruct the surface and use the underlying mesh to compute visibility. But to reconstruct the surface we need to orient the normals; a chicken-and-egg problem in our case. After some literature review on the subject, we did not find accurate method for LiDAR point clouds. Also, most papers try to find which points are visible from a precise viewpoint but in our case, as different scanners can see the same points, we want to know for each point which scanner best sees it. We introduce a custom point cloud visibility method that serves our purpose and works well with LiDAR point clouds, regardless of the sampling density.

ScanFinder and PointCloudVisibility are two contributions which serves mainly the same purpose: expand *ContextCapture*^{TM} input point cloud formats. Another improvement made in *ContextCapture*^{TM} is point cloud compression. *ContextCapture*^{TM} provides a cloud service which gives the opportunity for people not having any clusters or high-performance machine to do the job; reconstructing a large surface requires effective machines. The problem is that, point clouds can be very huge, up to one hundred (100) gibabyte and more. And if something happens while uploading, the upload restarts from scratch. Being able to divide by two point cloud sizes and then reduce uploading time is an interesting point for *ContextCapture*^{TM} cloud services. Point cloud compression can be adressed in two ways: geometric compression [GKIS05, SK06] or pure arithmetic compression regardless of the kind of file being compressed. We compared different compressor such as Brolti, LZMA (7Zip), Zip before integrating one of them into the product.

This report is organised as follows. Chapter 2 present Bentley Systems, Acute3D, *ContextCapture*^{TM} and how the achieved work is positioned in the company's business line. After introducing in Chapter 3 some useful definitions for a better understanding of the report, we describe the achieved work of Scan Finder, Point Cloud Visibility and Point Cloud Compression respectively in Chapter 4, Chapter 5 and Chapter 6. Note that in each chapter, we recall the context, the issue addressed and the expected result before going into details. Finally Chapter 7 summarizes all the work, evaluates my contributions to *ContextCapture*^{TM} and assess what this experience has brought to me.

[3]There is no need to know exactly which scanner generated it.

Chapter 2

The company

This chapter shed more light on *ContextCapture*$^{\boldsymbol{TM}}$, the software I contributed to during these six (6) months internship as well as Bentley Systems, the company.

2.1 Bentley Systems

Bentley Systems is an American-based software development company founded by Keith A. Bentley and Barry J. Bentley in 1984.

For a bit of history[1], they introduced the commercial version of PseudoStation in 1985, which allowed users of Intergraph's VAX systems to use low-cost graphics terminals to view and modify the designs on their Intergraph IGDS (Interactive Graphics Design System) installations. Their first product was shown to potential users who were polled as to what they would be willing to pay for it. They averaged the answers, arriving at a price of $7,943. A DOS-based version of MicroStation was introduced in 1986. Later the two other brothers joined them in the business. Today, Bentley Systems is considered to have four (4) founders: Greg Bentley (CEO), Keith A. Bentley (EVP, CTO), Barry J. Bentley, Ph.D. (EVP) and Raymond B. Bentley (EVP).

At its core, Bentley Systems is a software development company that supports the professional needs of those responsible for creating and managing the world's infrastructure, including roadways, bridges, airports, skyscrapers, industrial and power plants as well as utility networks. Bentley delivers solutions for the entire lifecycle of the infrastructure asset, tailored to the needs of the various professions – the engineers, architects, planners, contractors, fabricators, IT managers, operators and maintenance engineers – who will work on and work with that asset over its lifetime. Comprised of integrated applications and services built on an open platform, each

[1] Derived from [ben]

solution is designed to ensure that information flows between workflow processes and project team members to enable interoperability and collaboration.

Bentley's commitment to their user community extends beyond delivering the most complete and integrated software – it pairs their products with exceptional service and support. Access to technical support teams 24/7, a global professional services organization and continuous learning opportunities through product training, online seminars and academic programs define their commitment to current and future generations of infrastructure professionals.

With their broad product range, strong global presence, and pronounced emphasis on their commitment to their neighbors, Bentley is much more than a software company – they are engaged functioning members of the global community. Their successes are determined by the skills, dedication, and involvement of extraordinary Bentley colleagues around the world.

Bentley has more than 3,500 colleagues in over 50 countries, and is on track to surpass an annual revenue run rate of $700 million. Since 2012, Bentley has invested more than $1 billion in research, development, and acquisitions.

2.2 Acute3D

Acquired by Bentley Systems in February 2015, Acute3D is now developing *ContextCapture™* , as part of Bentley Systems' Reality Modeling solutions.

Acute3D is a technological software company created in January 2011 by Jean-Philippe Pons and Renaud Keriven, by leveraging on 25 man-years of research at two major European research institutes, École des Ponts ParisTech and Centre Scientifique et Technique du Bâtiment. It won the French "most innovative startup" Awards. In 2011, Acute3D signed an industrial partnership with Autodesk, while keeping to advance its R&D work on city-scale 3D reconstruction. In 2012, Acute3D signed industrial partnerships with other industry leaders, including Skyline Software Systems and InterAtlas. In parallel, it started to commercialize its own Smart3DCapture® (now replaced by *ContextCapture™*) standalone software solution, optimized for highly detailed and large-scale automatic 3D reconstruction from photographs. In 2015, Acute3D is acquired by Bentley Systems, and becomes part of their end-to-end Reality Modeling solutions.

2.3 *ContextCapture™*

With *ContextCapture™* , you can quickly produce even the most challenging 3D models of existing conditions for infrastructure projects of all types. Without the need for expensive, specialized equipment, you can quickly create and use these highly detailed, 3D reality meshes to provide precise real-world context for design, construction, and operations decisions for use throughout the lifecycle of a project.

Hybrid processing in ContextCapture enables the creation of engineering-ready reality meshes that incorporate the best of both worlds – the versatility and convenience of high-resolution photography supplemented, where needed, by additional accuracy of point clouds from laser scanning.

Develop precise reality meshes affordably with less investment of time and resources in specialized acquisition devices and associated training. You can easily produce 3D models using up to 300 gigapixels of photos taken with an ordinary camera and/or 500 million points from a laser scanner, resulting in fine details, sharp edges, and geometric accuracy.

Extend your capabilities to extract value from reality modeling data with ContextCapture Editor, a 3D CAD module for editing and analyzing reality data, included with ContextCapture. ContextCapture Editor enables fast and easy manipulation of meshes of any scale as well as the generation of cross sections, extraction of ground and breaklines, and production of orthophotos, 3D PDFs, and iModels. You can integrate your meshes with GIS and engineering data to enable the intuitive search, navigation, visualization, and animation of that information within the visual context of the mesh to quickly and efficiently support the design process.

Chapter 3

Background

This chapter introduces the necessary concepts and definitions for a better understanding of the report.

3.1 Point Cloud

Definition 3.1 *: Point*
A point p_i *is a tuple* $\langle x_i, y_i, z_i, \vec{n}_i \rangle$ *where:*

- i *is an unique integer identifying* p_i*,*
- x_i*,* y_i *and* z_i *are the coordinates of* p_i*,*
- $\vec{n}_i$ *is the normal at* p_i*.*

Definition 3.2 *: Point Cloud*
A point cloud P *of size* s *is a set* $P = \{p_i \mid i \in [0, s]\}$*.*

A point cloud is said to be whether *static* or *mobile* depending on the *static* nature of the scanner. Mobile scanner means that the instrument is either handheld (like some units used for LIDAR speed detection) or mounted on a vehicle (from SUVs to trailer-mounted instruments to airborne instruments). It generates a point cloud by following a trajectory. Static scanner instead means that the scanner is set up in a locationp and turns on itself.

3.2 Core algorithms

3.2.1 Principal Component Analysis (PCA)

PCA is a common algorithm of data analysis. It is used to project data with n dimensions into a space with reduced dimensions while keeping the most relevant information – the directions where there is the most variance, where the data is most spread out. As its name suggests, it finds the principal components from the most important to the less one. We say that the first principal component is a rotation of x-axis to maximize the variance of the data projected onto it and the last axis (component) is the one with less variance, with redundant information.

We are not going to explain in detail what a PCA is as it is very common and not the core of our work here. But let us remind the methodology in the specific case of 3D data.

Definition 3.3 *: Principal Component Analysis (PCA)*
Let P_a be a set of points of size s_a. To perform PCA(P_a):

- *we compute the centroid of P_a, noted $\bar{p_a}$*
- *we compute the covariance matrix C which is the symetric 3×3 positive semi-definite matrix:*

$$C = \sum_{p \in P_a} (p - \bar{p_a}) \otimes (p - \bar{p_a}), \text{ where } \otimes \text{ denotes the outer product vector operator.}$$

- *we compute the eigenvalues of C with their corresponding eigenvectors. The eigenvector $\vec{v}_1$ associated with the greatest eigenvalue λ_1 is the principal component axis. And the axis having less variation of data, when it is projected onto it is $\vec{v}_3$; the one associated with the lowest eigenvalue λ_3.*
- *we return a pair of $\langle \mu, \vec{v}_3 \rangle$ where $\mu = \frac{\lambda_3}{\lambda_1+\lambda_2+\lambda_3}$ can be used as a confidence level. The smaller it is, the more $\vec{v}_1$ and $\vec{v}_2$ explain on their own our data.*

In [HDD+92], Hoppe et al. use PCA to determine the normal of a tangent plane. Similarly in this report, we use PCA for two purposes: detect if a set of points are plannar by computing their normal's direction or simply find the normal's direction at a specific point using its neighbours. If a set of points forms a plane, we choose its normal to be either $\vec{v}_3$ or $-\vec{v}_3$. It is the direction with less data variation and in a perfect word, that is a perfect plane alignment, the projection of all points onto it gives the same value. This means that $\mu = 0$. Now if P_a is formed of p_i and its k-neighbouring points, then $\vec{v}_3$ or $-\vec{v}_3$ is the normal at p_i. We empirically deduced fifty (50) to be the ideal value for k.

3.2.2 Least Square Methods

Least Square solving is a usual approach of regression analysis to approximate the solution of overdetermined systems (with more equations than unknowns). The problem can be described as follows. Let us assume a model of the form

$$y = f(z; x_1; ..., x_n) \tag{3.1}$$

where f describes a relation between $x_1, ..., x_n$, z is the control variable and y is the expected response to z. After m experiments, $(m \geq n)$, m observed quantities $(z_i, y_i), i \in [1, m]$ are collected. The purpose therfore is to find the paramaters $x_1, ..., x_n$ so that all z_i, y_i satisfy at best (3.1) . "Least squares" means that the overall solution minimizes the sum of the squares of the residuals made in the results of every single equation. The most important application is in data fitting. The best fit in the least-squares sense minimizes the sum of squared residuals, a residual being: the difference between an observed value, and the fitted value provided by a model.

Least-squares fall into two categories: linear and non-linear least squares, depending on whether or not the residuals are linear in all unknowns.

Linear

To solve linear least square problems, we use the Eigen library [GJ+10]. Consider an overdetermined system of equations, say $Ax = b$. Although it has no precise solution, it makes sense to search for the vector x which is closest to being a solution, in the sense that the difference $Ax - b$ is as small as possible. This x is called the least square solution (if the Euclidean norm is used). Provide Eigen with A and b and it will approximate the solution x.

Non-linear

In case of non-linear least square problems, we use the Ceres Solver [AMO]. Ceres Solver is an open source C++ library for modeling and solving large, complicated optimization problems. Ceres can solve bounds constrained robustified non-linear least squares problems of the form:

$$\min_x \quad \frac{1}{2} \sum_i \rho_i \left(\| f_i(x_{i_1}, ..., x_{i_k}) \|^2 \right), \text{ such that } l_j \leq x_j \leq u_j$$

Ceres solver considers the expression $\rho_i \left(\| f_i(x_{i_1}, ..., x_{i_k}) \|^2 \right)$ as a *Residual Block*, where $f_i(.)$ is a *Cost function* that depends on the parameter blocks $[x_{i_1}, ..., x_{i_k}]$. We only need to specify the parameters, write the *Cost function* and Ceres does the job.

Function RANSAC(P, k, n, t)

Input:

- a set of points $P = \{p_i \mid i \in [0, s]\}$,
- the number k of iteration,
- the required sample number n for fitting parameters to the mathematical model,
- the tolerated error threshold t when testing parameters on one sample.

Output: A tuple $\langle m, l \rangle$ where m is the final parameter estimation and l the associated set of inliers.

```
m ← 0⃗ // parameter vector
l ← ∅ // inliers set
δ_n ← ∅ // set of extracted samples
for ( i = 0; i < k; i = i + 1 ) {
    δ_n ← random_sample(P, n) // Extract n random points from P
    m′ ← fit_parameters(n, δ_n) // Use an estimation method to fit m′ to the mathematical model
    l′ ← ∅
    for ( j = 0; j < s; j = j + 1 ) {
        e ← compute_error(m′, p_j) // get model error when applied on point p_j
        if e² ≤ t² then
            l′ ← l′ ∪ {p_j}
    /* If it is the most satisfied mathematical model up to now */
    if size_of(l′) ≤ size_of(l) then
        l ← l′
        m ← m′
return ⟨m, l⟩
```

Algorithm 1: A variation of the RANSAC algorithm used in this report

3.2.3 RANSAC

According to [ran]: RANSAC is the abbreviation of Random Sample Consensus. It is a general algorithm that can be used with other parameter estimation methods in order to obtain robust models with a certain probability when the noise in the data doesn't obey the general noise assumption.

Various versions of this algorithm exist, this is why it is said to be *general*. Algorithm 1 shows the version used in this report. Generally, RANSAC selects a subset of data samples and uses it to estimate model parameters. Then it determines the samples that are within an error tolerance of the generated model. These samples are considered as agreed with the generated model and called as consensus set of the chosen data samples. Here, the data samples in the consensus as behaved as inliers and the rest as outliers by RANSAC. If the count of the samples in the consensus is high enough, the new model is saved. Actually, it repeats this process for a number of iterations and returns the model which has the smallest average error among the generated models.

Chapter 4

Scan Finder

This chapter describes *Scan Finder*, an algorithm that can be used to retrieve all scanners position of a point cloud, if there are any, as long as it is static. Firstly, Section 4.1 reviews the context which brings this need and describes some characteristics of the expected solution. Then, Section 4.2 reminds that there is no previous work related to this subject. Finally, Section 4.3 describes a first step toward scanner location finding before showing and discussing the results of two different approaches described respectively in Section 4.4 and Section 4.5. To be clear, Section 4.3 is a common first step to both methods.

4.1 Specifications

4.1.1 Context

ContextCapture^{TM} started to support point cloud reconstruction two (2) years ago. Today it even provides a hybrid processing mode which gives the opportuniy to combine the best of both words, photos and point clouds, in order to have a better precision. However, a recurring problem observed among *ContextCapture*^{TM} users is the impossibility to use the software after losing some metadata, specifically, scanners location. This is particulary problematic because usually, *ContextCapture*^{TM} users subcontract point cloud production to private companies which can charge them again for new exports (provided that they still have the point clouds). To enable customers to use their *defective* point clouds, the graphic interface of *ContextCapture*^{TM} allows to specify a scanner location by hand by positioning it in a 3D representation of the point cloud. But, it does not work with merged scans; only one scanner location can be specified. Usually, users do not know the location and even if they know, the reconstruction is subjected to too much errors.

This is how the need for an algorithm to automatically find scanners positions in a point cloud is born. In addition, with such algorithm, *ContextCapture*^{TM} will have the possibility to enhance the set of supported file formats. Currently, it supports file formats such as *PTX*, *e57*, *PLY*, *POD*, each of them being able to store scanners positions. But not all file formats are able to do it. For instance, *LAS* file format is not currently accepted as input because it does not provide any means of storing scanners location in metadata. Supporting more input file formats is also a good point of interest for *ContextCapture*^{TM} users.

4.1.2 Objective

In summary, the purpose here is to improve *ContextCapture*^{TM} point cloud support in two ways: support more file formats and allow users who lose scanners location to still use their point clouds. To do so, the algorithm must:

- take as input any 3D point cloud captured from static scanners,
- be invariant to the number of scanners in the point cloud,
- be invariant to differences between scanners, such as: density, noise, rotation angle,
- find all scanners locations,
- have a reasonnable running time.

Let us emphasize here that as a first step, the algorithm is expected to work only with static point clouds. It would be difficult to have the same approach with static and mobile point clouds. Extending it to mobile point clouds is certainly the next step.

4.2 Related Work

As said in the introduction, to the best of our knowledge, there have been no previous work on the subject.

"Detecting the positions of multiple scanners exclusively from a point cloud is a subject not identified as interesting by academics, and thus not tackled since this information is almost always available from the outset. The same holds true on the industry side, it is only recently that scanner position information is relevant for a few applications like ContextCapture^{TM} *."*

The closest publications on the topic are [TM15, SQLG15, KGC15, KC15, NS17]. They use learning techniques to estimate the point of view of one particular photo based on other photos. But this belongs more to the photogrametry domain and therefore is not applicable in a point cloud context.

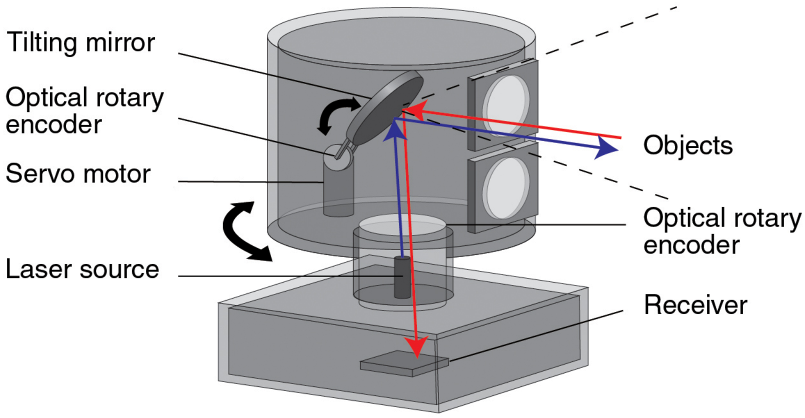


Figure 4.1: Example of a LiDAR scanner.

4.3 Clustering high-density area

This section explains a common first step to methods described in Section 4.4 and Section 4.5. For a better understanding, a quick introduction to LiDAR scanners is necessary.

4.3.1 LiDAR scanner and context

LiDAR[1] is an acronym of Light Detection and Ranging. It is a remote sensing technology which uses the pulse from a laser to collect measurements. The principle is simple: it works in a similar way to Radar and Sonar but uses light waves from a laser, instead of radio or sound waves. A LiDAR system calculates how long it takes for the light to hit an object or surface and reflect back to the scanner and then, uses the velocity of light[2] to calculate the distance. LiDAR systems can fire around 1,000,000 pulses per second.

When scanning, there are two kind of rotations that a LiDAR scanner performs. They are indicated by black arrows in Figure 4.1. Not only a LiDAR scanner has a constant rotation angle when turning on itself but it also has a constant vertical rotation angle when scanning the environment. Because of these constant rotation angles, the further we go, the lower the point density is. On the contrary, the closer to the scanner we are, the higher the point density is. This density variation can be observed in Figure 4.2. To conclude, extracting the most dense areas is interesting as they always are around scanner locations and then, reduce the global search area.

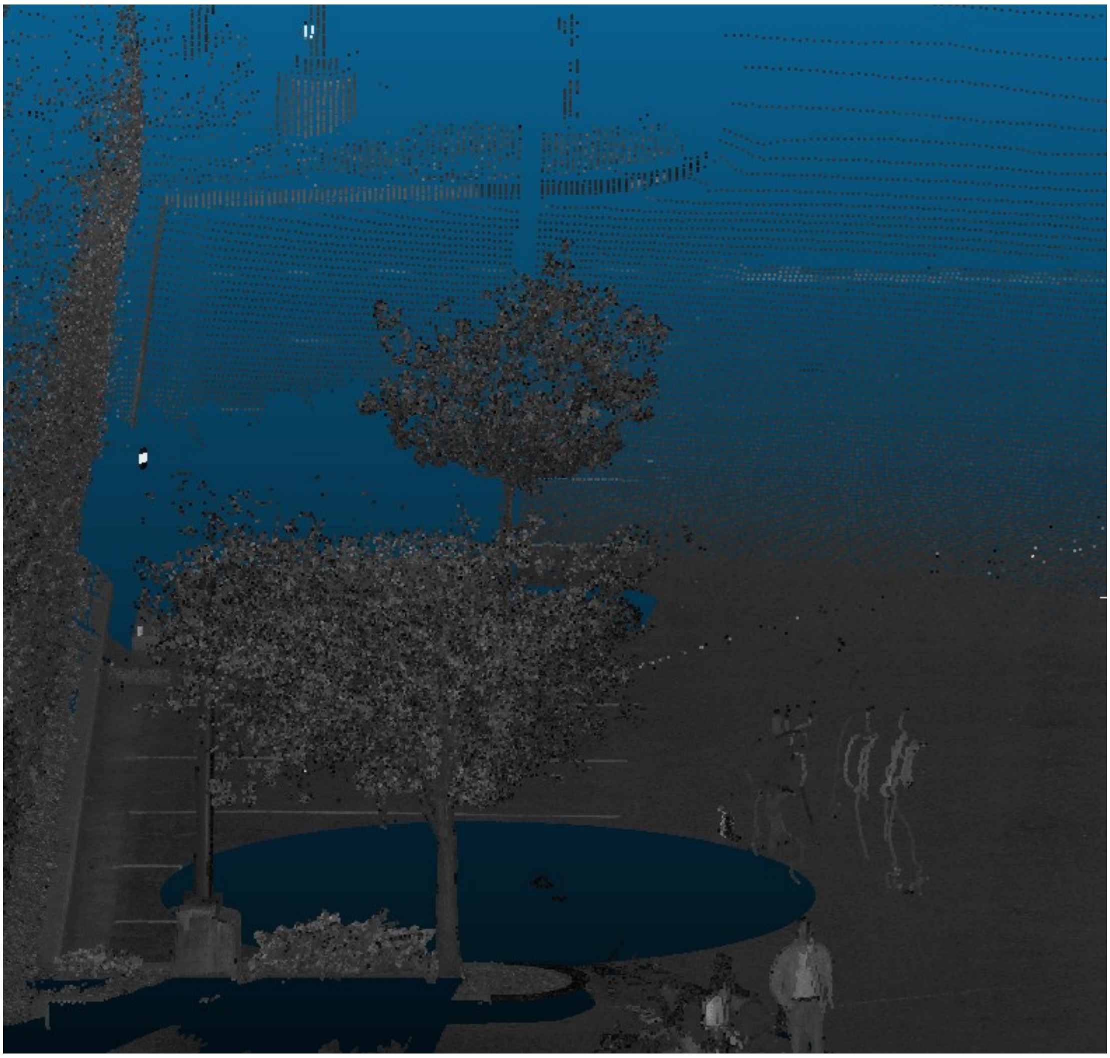


Figure 4.2: Variation of density as we go further away from the scanner source (the empty circular space).

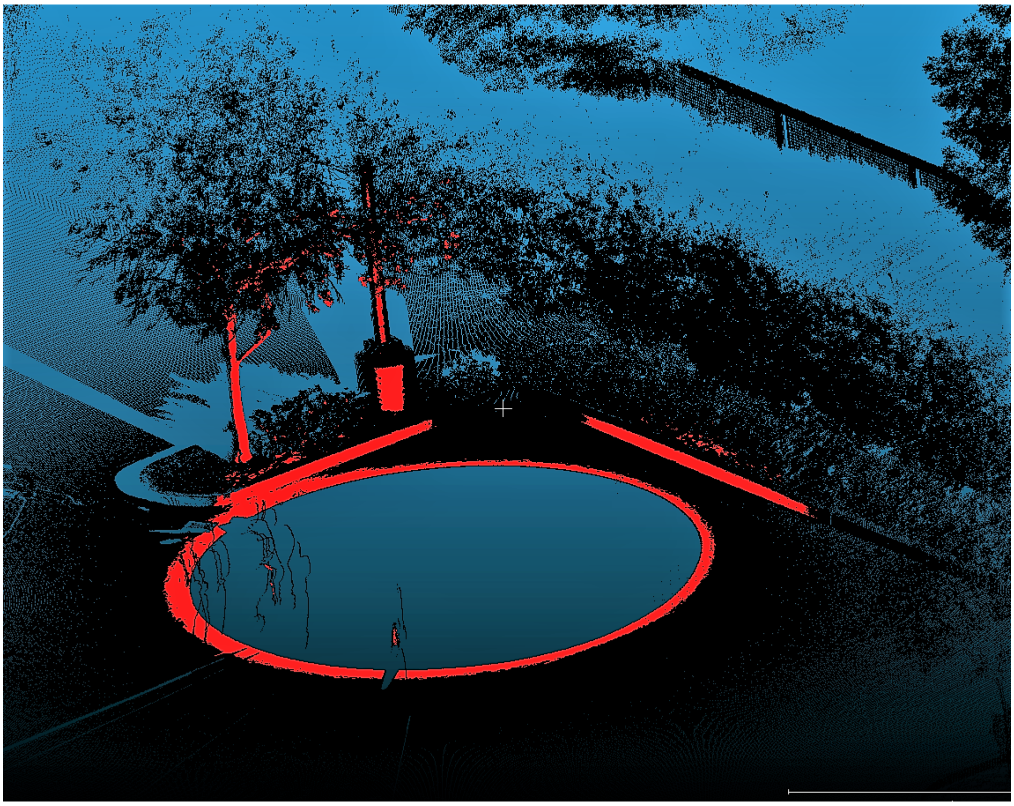


Figure 4.3: The result of the high density extraction of a point cloud.

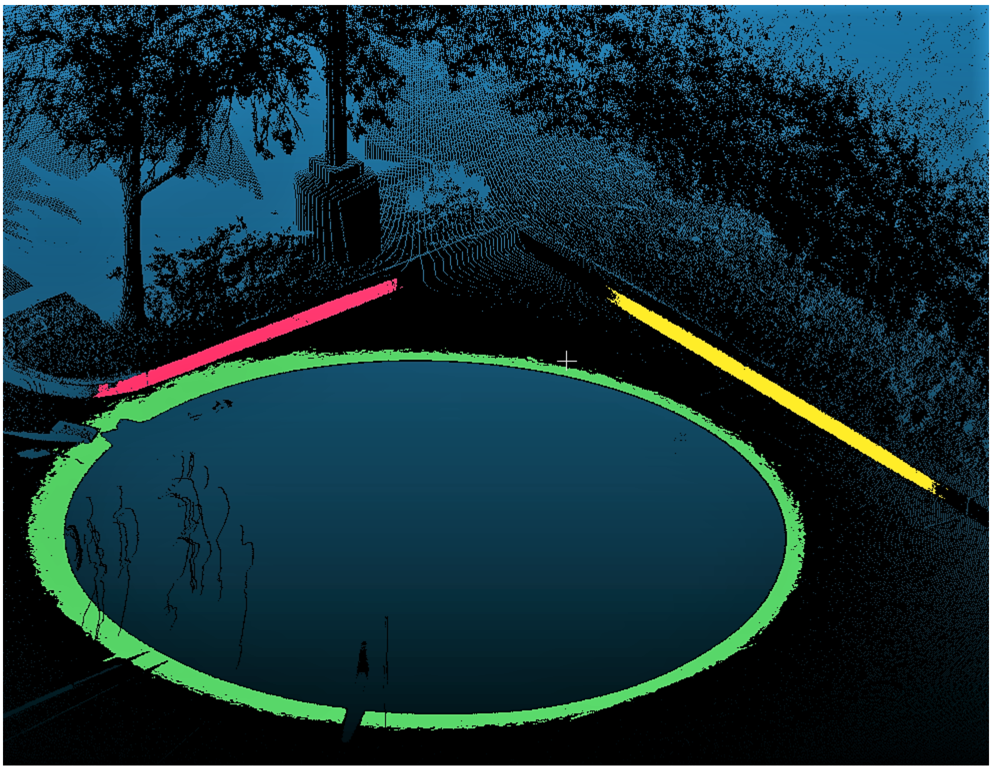


Figure 4.4: The result after clustering dense points of Figure 4.3.

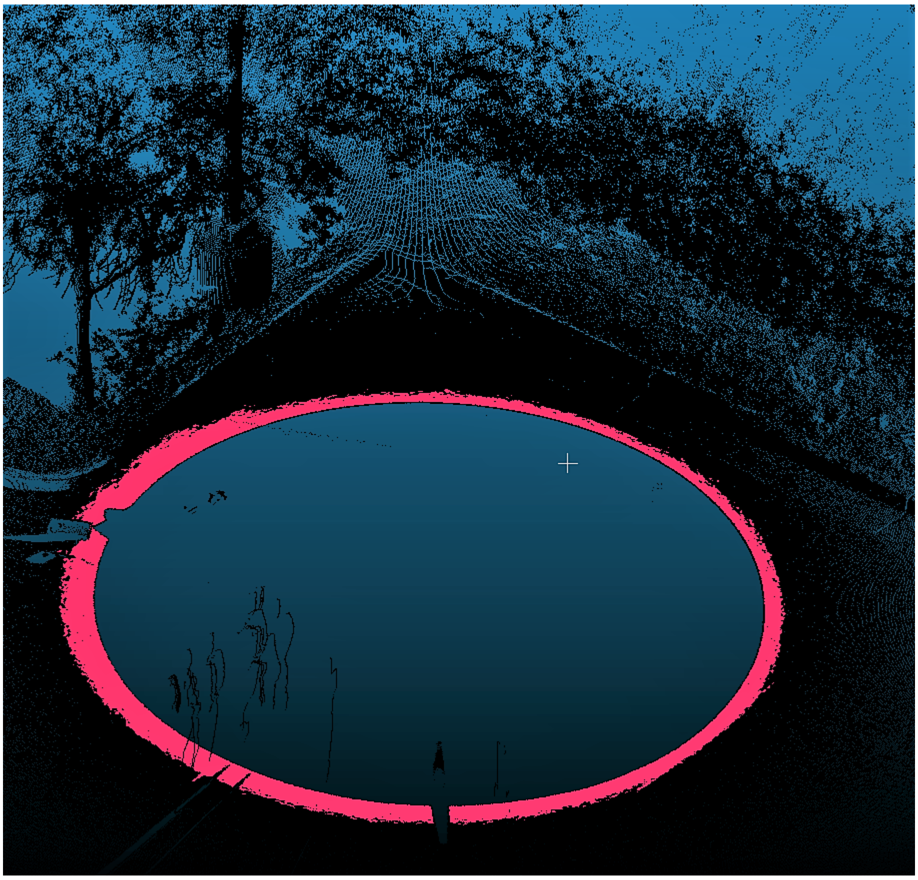


Figure 4.5: The result after trying to identify circular clusters of Figure 4.4.

4.3.2 Clustering and dense area extraction

The first step of the algorithm is to detect high density regions in the point cloud. For each point, we compute its density based on the mean distance to its k neighbors ($k \in [15, 50]$). We extract high density regions by selecting a percentage of the densest points (percentage between 1 and 5%). Figure 4.3 shows some results of this high density point extraction. As you can see, they always are around the scanner's location. Note that the scanner is not able to scan below himself leading to an elliptic empty form[3] appearing in all point clouds.

Once these points are extracted, a clustering step is applied. The clustering is needed so that points from the point clouds corresponding to potentially different parts of the scene around the scanner can be separated. For each point, we compute its non-oriented normal and the associated *confidence level* by performing a principal component analysis (PCA) on its local neighborhood. Section 3.2.1 gives more details on how this PCA works. Then, as long as points are not attributed to a cluster, we select the point on the flattest surface, and iteratively add its neighbors that share a coherent normal orientation (the dot product of the two normals must be superior to 0.8 in our experiments). Note that the flattest surface is determined using condifence levels, they tell which point belongs to a planar region. Figure 4.4 shows the result of such clustering.

[1]This paragraph is inspired by [lid].
[2]The velocity, or speed of light is 299,792,458 metres per second.
[3]This elliptic form is the key point of the method described in Section 4.5.

Once every point is clustered, we discard clusters with too few points (in our experiments, we use a threshold of 0.2% of the number of high density points). We then detect clusters that are circular. A cluster is considered circular if its centroid, i.e. the average of all the points it contains, is far away from the points of the cluster. Figure 4.5 shows a circular cluster identified.

At this point, we have computed a rough estimate of the scanner's location. Methods described in Section 4.4 and Section 4.5 use it in their own way in order to find scanner locations.

4.4 The grid-pattern method

This section explains one approach we tried in order to solve the problem. Before explaining the method in detail, let us first give the intuition behind this idea.

4.4.1 Intuition

As said in Section 4.3.1, LiDAR scanners perfom two kind of rotations. These constant vertical and horizontal rotation angles reveal some grid patterns on surfaces perpendicular to the ground[4]. Figure 4.6 shows an example of a grid pattern that can be found in point clouds. The purpose here is to find, for a single grid pattern, a relation between all constant rotation angles and the scanner's position. Therefore, this relation can be applied with all possible grid patterns in the point clouds, leading to a problem with a huge set of constraints to solve. The idea is that only the real scanner's location is able to explain in the best way these constant rotation angles.

This algorithm can be divided into two parts. The first step is to find *accurate* grid patterns in point clouds. Once this is done, the second step is to build the equation that will be solved. These two parts are explained in the following subsections. Note that this approach does not work in multiscan mode, compare to the other approach described in Section 4.5. It assumes there is only one scanner which explains all grid-patterns in the point cloud.

4.4.2 Grid-pattern matching

This subsection explains how to find *accurate* grid patterns in point clouds in order to reduce as much as possible the noise and its impact during the problem resolution described in Section 4.4.3. A set of point is considered as an *accurate* grid-pattern when:

- it is planar as much as possible[5],

[4]To be precise, perpendicular to the surface on which the scanner is installed.
[5]Because of the noise.

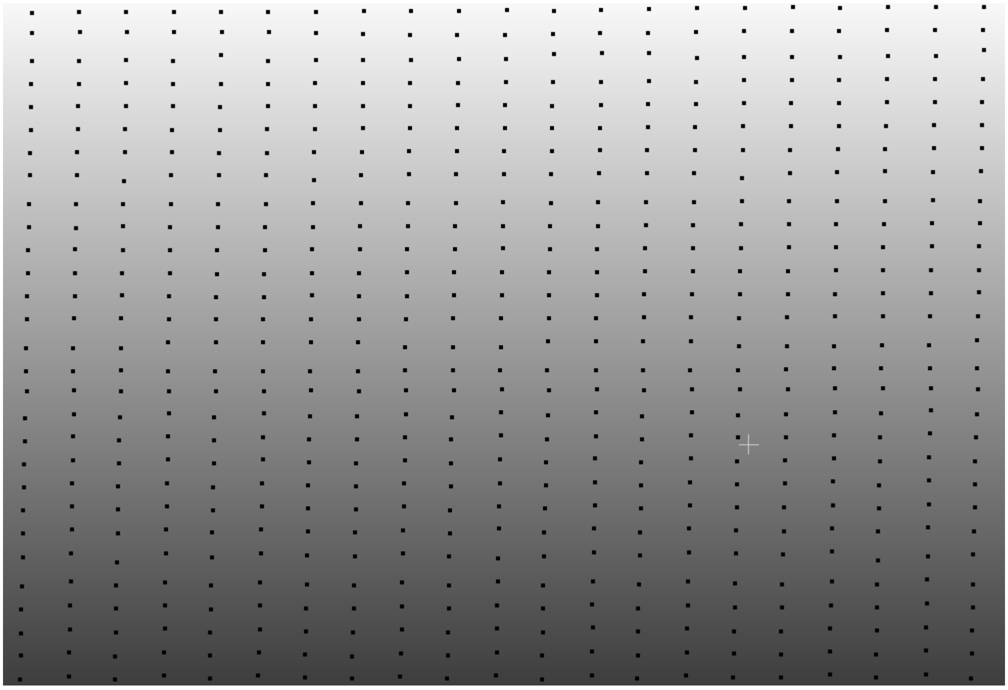


Figure 4.6: Example of a grid pattern. As you can see there is a constant offset between each vertical line and each horizontal lines. Note that the offset between columns is not necessary the same than between lines.

- the underlying planar surface is orthogonal to the surface of the ground,
- the gap between columns is consistent enough,
- the gap between lines is consistent enough.

Algorithm 2 shows how to extract some planar patches in a point cloud. A preprocessing step already done and explained in Section 4.3.2 is to compute for each point of the point cloud, the normal and its *confidence level*. This *confidence level* plays a key role in the algorithm as it tells how planar the region around each point is. By browsing all points, each time we find a point with a good *confidence level* (> 0.001), we start to build a patch around it. To do so, we extract its 50 closest neighbours and consider only those having their normal almost colinear to the normal of the starting point. If the size of the obtained set is bigger enough ($> 0.9 \times 200$), the set is kept and otherwise, not. Also, in order to avoid overlaping or patches too close, we keep track of visited points.

Although planar, this set of patches potentially contains: **(a)** patches whose underlying surfaces are not orthogonal to the surface of the ground and **(b)** patches without grid patterns. To filter out each **(a)** patch, we use a dot product between its normal and the normal of the circular cluster found in the high-density area[6] (see Section 4.3). For **(b)** patches, the objectif is to indentify grid-patterns. To do so, we use least square method in order to fit lines. Least-square is explained in Section 3.2.2. Fitting a line fall into linear least square problems. Figure 4.7 shows how to fit a line to a set of points using the Eigen library [GJ+10]. The purpose is to use

[6]The circular cluster desribes the surface on which the scanner is positioned.

```
Function GetPlanarPatches(P)
    Input: a pointcloud P = {p_i | i ∈ [0, s]}.
    Output: a set of potential grid-pattern patches.
    /* We compute the normal of each point and keep both confidence level and normal ⟨μ_i, v⃗_i⟩.   */
    /* See Section 3.2.1 for more details on PCA algorithm.   */
    /* From here, ∀i ∈ [0, s] we assume v⃗_i to be the normal at i and μ_i its confidence level.   */
    for ( i = 0; i < s; i = i + 1 ) {
        ⟨μ_i, v⃗_i⟩ ← PCA(50 closest neighbours of i)
    R ← ∅
    visited ← ∅
    for ( i = 0; i < s; i = i + 1 ) {
        /* Continue if already visited or is not planar enough.   */
        if i ∈ visited or μ_i > 0.001 then
            continue
        tmp ← {p_i}
        foreach j ∈ 50 closest neighbours of i do
            /* If v⃗_j and v⃗_i are almost colinear and j has not been seen yet.   */
            if abs(v⃗_j × v⃗_i) > 0.9 and j ∉ visited then
                tmp ← tmp ∪ p_j
                visited ← visited ∪ j
        /* If there is enough point for a grid-pattern patch.   */
        if size of tmp > 0.9 × 200 then
            R ← R ∪ tmp
    return R
```

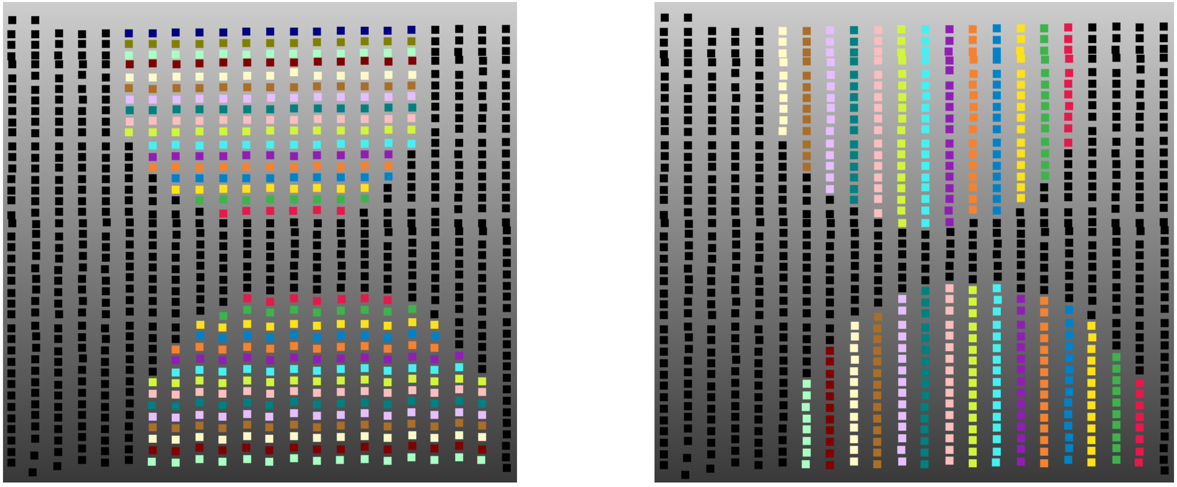
Algorithm 2: Find various not-overlaping planar patches in a point cloud.

this line fitting in order to cluster lines and columns. Figure 4.8 shows results of a clustering performed on two patches after several fittings and adjustements. We only keep firstly patches containing a grid-pattern and then, patches having a regular gap between columns and lines. The rest is discarded.

Once only *accurate* patches are kept, columns and lines are identified, the problem can be formalised.

```cpp
Eigen::VectorXf FitLine(std::vector<A3D::Point_3dPlus> const& points)
{
  Eigen::MatrixXf mat(points.size(), 2);
  Eigen::VectorXf vec(points.size());
  int count = 0;
  for (auto it = points.begin(); it != points.end(); ++it)
  {
    mat(count, 0) = static_cast<float>(it->x());
    mat(count, 1) = static_cast<float>(it->y());
    vec(count++) = 1.0;
  }
  return mat.fullPivHouseholderQr().solve(vec);
}
```

Figure 4.7: Function using least-square to fit a line to the given set of points.



(a) Lines clustering performed on two patch. (b) Column clustering performed on two patches.

Figure 4.8: Example of columns and lines clustering in order to identify grid-patterns.

4.4.3 Equation to solve

For a little recall of Section 4.4, the purpose here is to find a relation between the scanner's position (what we are looking for) and the regular gaps between lines and columns. At this point, we already have a reduced area of research which is around the circular cluster obtained in Section 4.3.2. What we need, is a set of constraints that will help to set the scanner at a specific position within this area.

Look at Figure 4.9. There are four (4) vertical lines viewed from a bird's eye: a, b, c and d. They are represented as points because of the point of view. A regular between them can be observed. As explained in the introduction to LiDAR scanners (Section 4.3.1) this regular gap is due to the consistent angle of rotation of the scanner $\alpha_1 = \alpha_2 = \alpha_3$.

These angles directly involve the scanner location in all equations. Here is how the problem is built: each time the scanner location is set, we minimize for each grid-pattern, and for each pair of two consecutive lines (or columns), the absolute difference between their respective angles. In our case me minimize:

$$d_1 = |\alpha_1 - \alpha_2|$$

$$d_2 = |\alpha_2 - \alpha_3|$$

This is a non-linear least-square problem which can be resolved using the Ceres Solver [AMO] (Section 3.2.2).

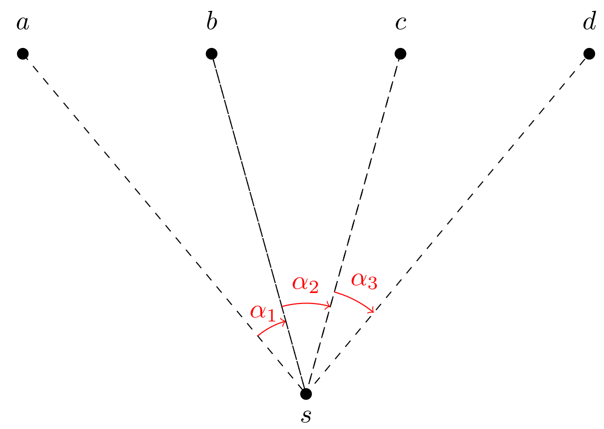


Figure 4.9: A top view over four (4) vertical grid lines (a, b, c and d) and a scanner (s).

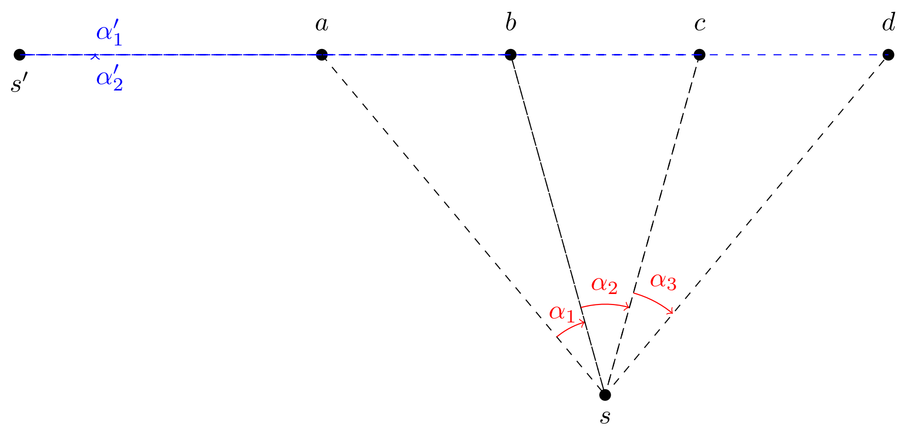


Figure 4.10: Illustration of the problem preventing the grid-pattern method from finding the scanner location. Note that this is a top view over four (4) vertical grid lines (a, b, c and d), the apprixmation (s') and the real scanner (s).

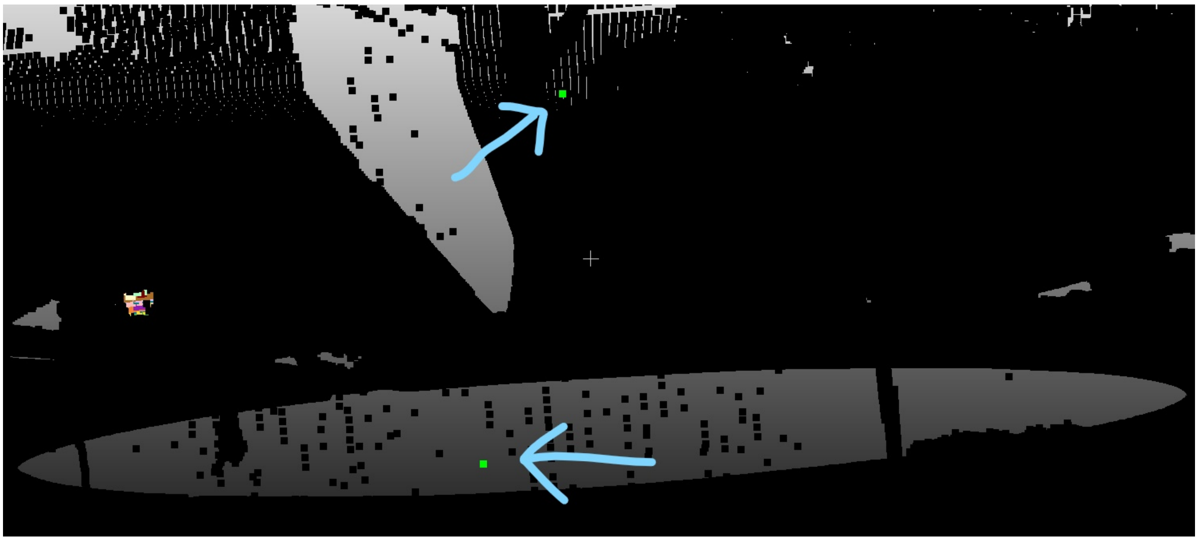


Figure 4.11: An approximation of the scanner location using the grid-pattern method. The real scanner location is highlighted by the upper arrow while the approximation is highlighted by the lower one.

4.4.4 Results and discussions

Figure 4.11 shows a result of grid-pattern approximation of scanner location. This method does not perform quite well. One would think that the approximation is not far from the real scanner location but reducing the area of research to a cube around the circular cluster can be misleading. We tried to remove the area constraints and observed that the approximation moves entirely away from the real location and even the point cloud itself. It can stoop pretty low, go to really high altitude, on the left, on the right. It completely depends on the distribution of the grid-patterns in the point cloud.

Let us take one grid-pattern in order to illustrate the problem: Figure 4.10. As our method tries to minimize the differences between all pairs of angles, here α_1, α_2 and α_3, the ideal case is if the scanner location belongs to the underlying surface of the grid-pattern in which case:

$$|\alpha_1 - \alpha_2| = 0$$

$$|\alpha_2 - \alpha_3| = 0$$

Therefore, with a huge set of equations involving several grid-patterns, the optimal solution for the solver is to put the scanner on the *mean plan of all grid-pattern's underlying planes.* This is why, without the reduced area constraint, the approximated scanner is far away from the point cloud as the solver tries to reduce all angle differences and then, satisfy all grid-patterns. Another problem found is that this method is too much subjected to noise. We are talking about really small values for angle differences that we want to minimize. A small noise can have huge effects during the solving.

To conclude, angles differences are not discriminating enough. We believe there is a way to express the problem, in order to bypass this behaviour but we deciced to try another approach presented in Section 4.5. The current approach is limited to one scanner whereas the other one works with multiple scaners, therefore, is more interesting for *ContextCapture*^{TM} .

4.5 The elliptic method

This section describes the second approach to solve the scanner location finding problem. This approach is more interesting than the one of Section 4.4 because it works natively with merged point clouds (multiple scanners within a point cloud). Let us have the intuition of this method.

4.5.1 Intuition

Recall that a first common step to both methods is performed. This step finds all circular clusters around all scanner locations of the point cloud. It is described in Section 4.3. As said in Section 4.3, these empty circles appears because the scanner is not able to scan below itself.

This method tries to find the scanner location in two steps:

- find the z axis on which the scanner is
- set the scanner on this axis by estimating its height

The z axis on which the scanner is is simply the axis perpendicular to the circular cluster which runs through its center. In order to find this axis, we fit an ellipse arround the circular cluster before finding its center. Once the axis is known, we try to estimate the scanners height.

The first key point of this method is that the scanner location is necessary on the axis perpendicular to this elliptic form and which runs through its center. The intuition is that the density of a local point is linked to its distance from the scanner. Figure 4.2 shows the variation of density as we go further away from the scanner location. Using multiple points in the high density area, we are able to triangulate the position of the scanner. The real scanner location is the best position able to explain the link between the local density of each point and its distance from the scanner.

In Section 4.5.2 we describe how the scanner's axis is found and in Section 4.5.3 we show how to formalise the non-linear least square problem to solve.

4.5.2 Fitting ellipse

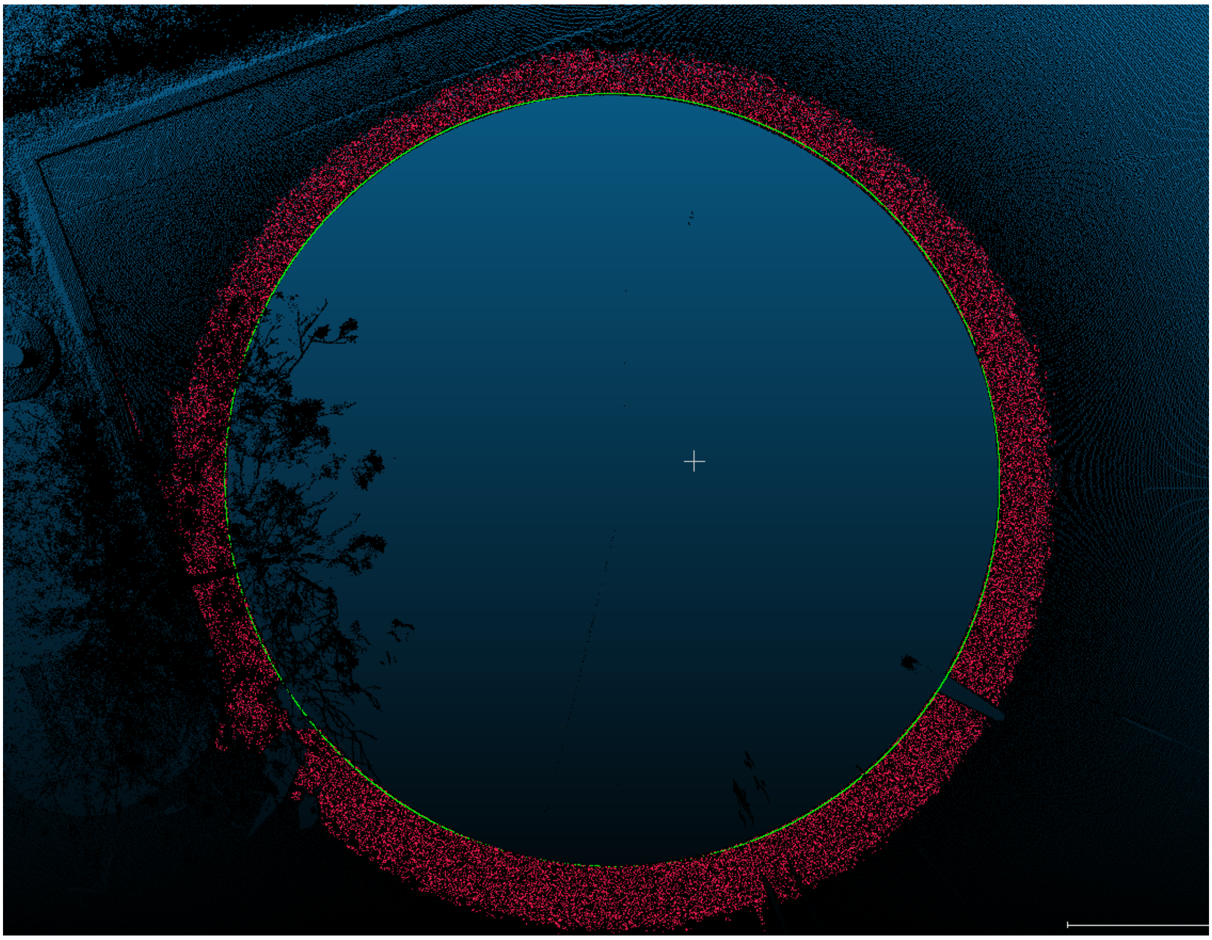


Figure 4.12: Ellipse fitted on a circular cluster.

This section describes how we fit an ellipse on a circular cluster before finding its center. We need the ellipse to be as close as possible to the cluster interior. For this, the following procedure is followed:

- Extract the closest points Δ to the cluster interior. We first compute the cluster's centroid c. Then, for each point p_i of the cluster, we extract its k nearest neighbors and compute their centroid c_i. If p_i is closer to c than c_i with a percentage of error, then it is considered as part of the ellipse edge.
- Perform a PCA on Δ to compute its normal. This normal is considered as the ellipse normal.
- Align the normal with the z axis using geometric transformation so that the future ellipse is completely horizontal. This is an important step as we need to use the elliptic two-dimensional equation:

$$Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0 \tag{4.1}$$

  This alignment allows us to ignore the last dimension while fitting the ellipse.
- Try to fit an ellipse using a RANSAC algorithm. This enables to filter out outlying points in order to

only keep *good* points. The RANSAC algorithm is described in Section 3.2.3. We call the algorithm with $k = 3000$ iterations, $n = 6$ unknowns and $t = d_e * 6$ where d_e is the mean density of Δ.

- As the equation of the ellipse in this 2D projection is known, we compute its center's location.

A result of such procedure can be observed in Figure 4.12. The circular cluster has a red color while the fitted ellipse is in green.

4.5.3 Equation to solve

To estimate the height of the scanner, we will use the slight variation in density of the points in the circular cluster. For a point p in the high density circular cluster, we note r the distance between the scanner and the point, d the density of the point, and α the angle between the normal of the point and the direction from the point to the scanner. Using the fact that point density decreases proportionally as moving away from the scanner, the following relation should be verified for any pair of points.

$$\frac{d_a \times r_a^2}{\cos(\alpha_a)} = \frac{d_b \times r_b^2}{\cos(\alpha_b)} \tag{4.2}$$

where d_a is the spatial density of a and d_b the spatial density of b. We normalize each spatial density by the cosinus of each normal's point and the optic beam in order to optain the scanner's point of view density which is different from the spatial density. Look at Figure 4.13 for a visual understanding. The local density of points a and b are known but of course their distances from the scanner are not because they depend on the scanner position. This is why knowing the axis on which the scanner is is interesting because it helps to restrain the area of research. Note that two solutions can be found but we easily discriminate the other one as it is under the point cloud.

For each circular cluster, we pick a sample of n points (in our experiments, $n = 6$ points) with different densities. We then build a system of (4.2) equations using all possible combinations of the n points in pairs. We then use Ceres [AMO] which is a non-linear least square solver to determine the position of the scanner. As a result, we retrieve an estimated position of the scanner location.

4.5.4 Results and discussions

This method happens to achieve good results. Figure 4.14 shows an approximation (light blue point) of the real scanner location (green) point. As you can see our approxiamtion is very close to the real scanner's location. By testing on various reference datasets, we found that the error between the correct position and our estimation ranges between 2 and 20 cm.

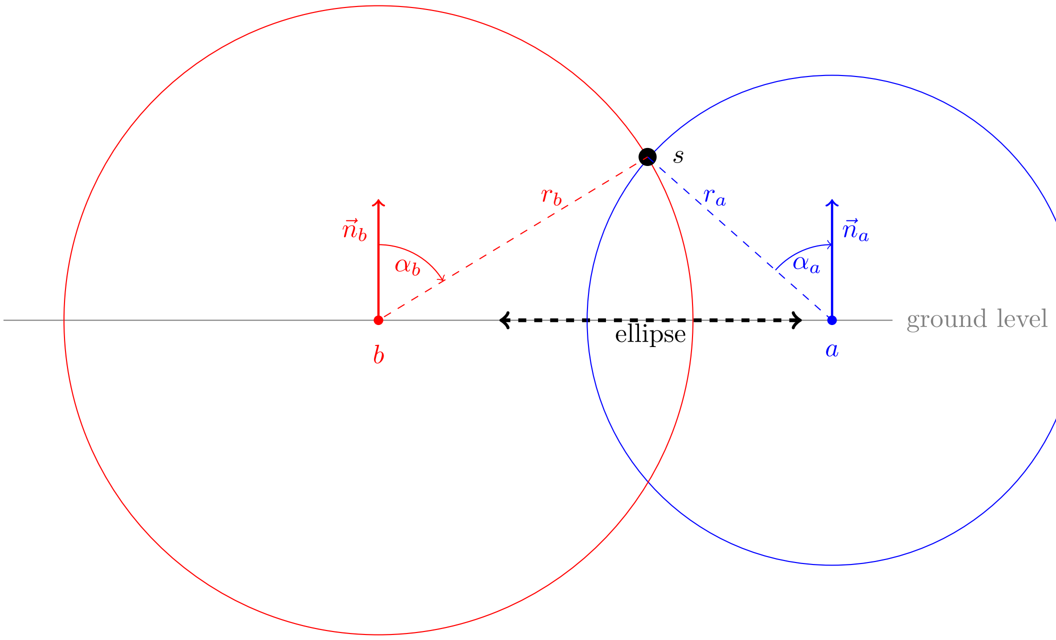


Figure 4.13: A side view of the ellipse and two spheres intersecting at the scanner position s and their respective centers a and b. We have the relation $\frac{d_a \times r_a^2}{\cos(\alpha_a)} = \frac{d_b \times r_b^2}{\cos(\alpha_b)}$ where d_a is the density of a and d_b the density of b.

This pipeline is particularly suited for multi-scan point clouds. Terrestrial scanners tend to generate most of the points close to their positions. If we consider that scans are not taken in the same close vicinity, high density clusters are disjoint. We confirm with hypothesis in our experiments. Therefore, the rough detection works well in the multi-scan hypothesis. The same idea applies for the fine position algorithm: most points in the circular cluster come from a single scanner. The few other points coming from different scanners don't have a significant impact on the local density of the points. That way, the scanner height is correctly detected in a multi-scan setting. Figure 4.15 and Figure 4.16 show some results of this approach on multiscan point clouds.

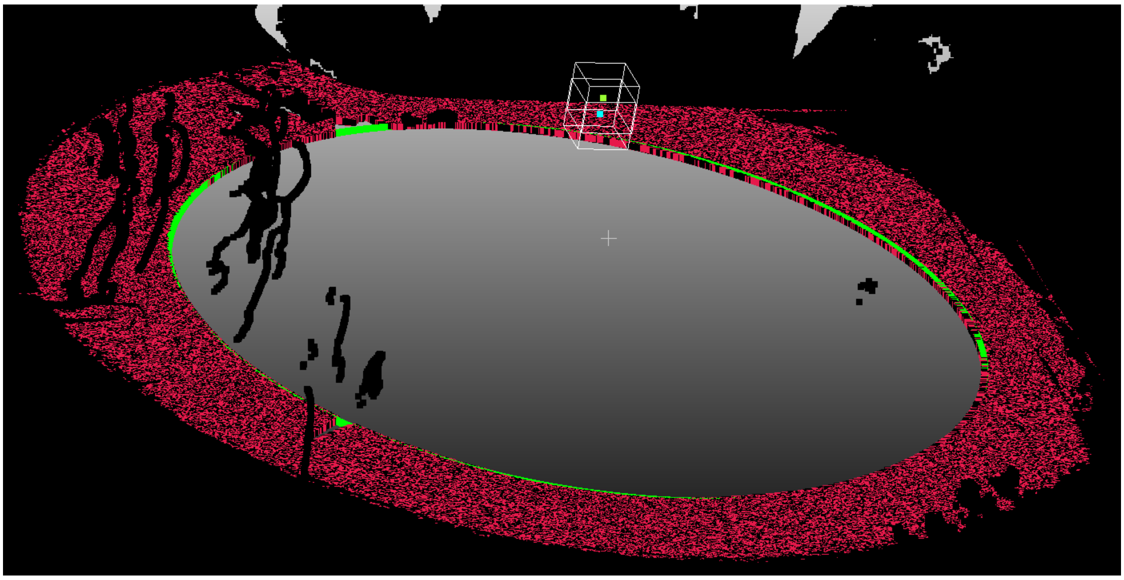


Figure 4.14: Green point in the box is the correct position, light blue point next to it is our estimation.

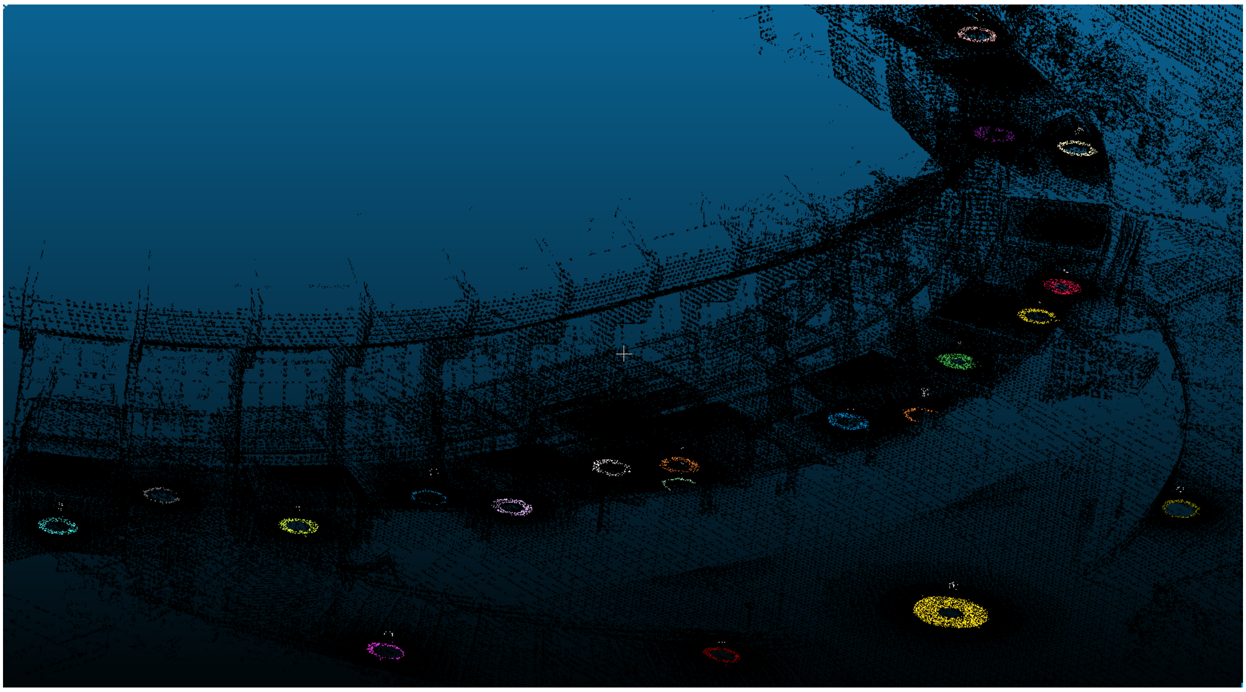


Figure 4.15: Multiple locations detected in a large point cloud.

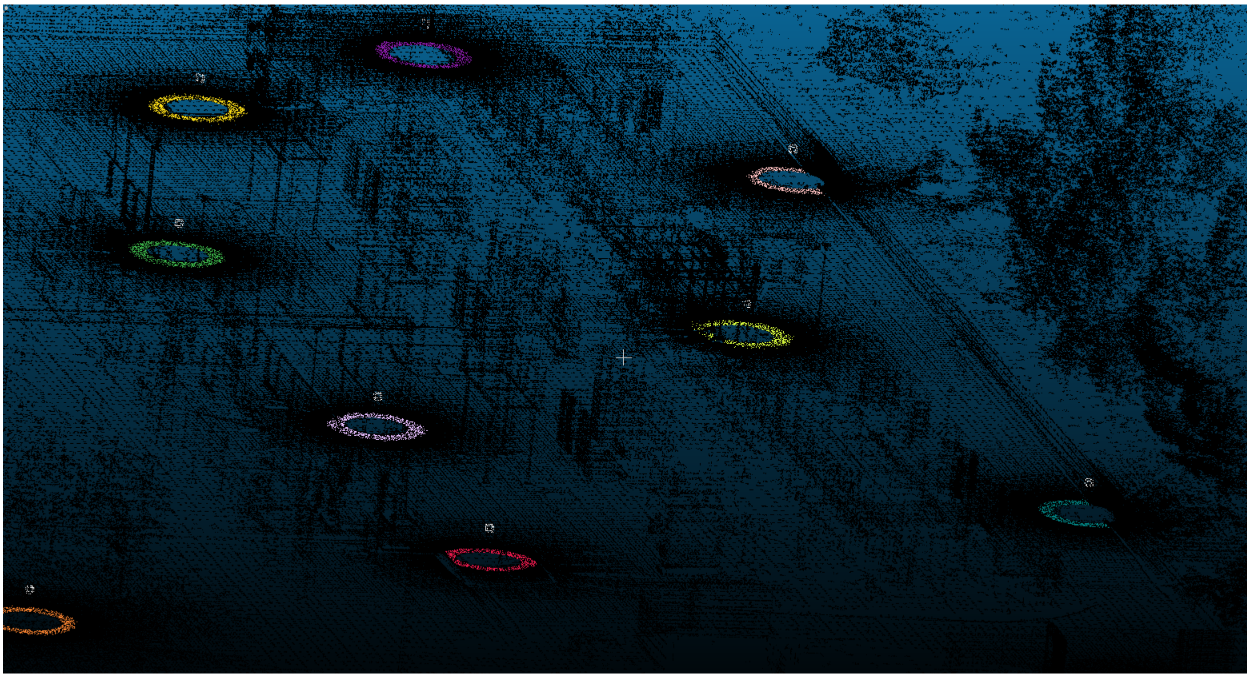


Figure 4.16: Multiple locations detected in another large point cloud.

Chapter 5

Point Cloud Visibility

This chapter introduces our custom point cloud visibility algorithm. It starts by giving in Section 5.1, a clear context for the presented work. Then, Section 5.2 gives an overview of the previous related work and describes some experimentations previously-published point cloud visibility algorithms. Finally, as nothing seems to be well adjusted for our specific case we implemented a custom point cloud visibility algorithm described in Section 5.3.

5.1 Specifications

5.1.1 Context

Currently in *ContextCapture™* , there is a way to still use a point cloud for reconstructing purposes even if the scanner's location information is lost. It gives the user the opportunity to manually set the scanners position on a 3D representation of the point cloud. But, only one position can be set. It requires that the point cloud contains points resulting from *one and only one* scanner[1]. This is because, during the reconstruction, *ContextCapture™* uses the scanner location to orient the normals at each point. In a monoscan point cloud case, it simply orients each normal toward the scanner. But in case of multiscan point cloud, it needs to know for each point, toward which scanner the normal must be oriented, which is difficult to find.

Therefore, even if *ScanFinder* (Chapter 4) is applied on a LAS format[2] multiscan point cloud in order to retrieve all scanner positions, the reconstruction is still infeasible for *ContextCapture™* . This is where a point cloud visibility algorithm is useful in order to attribute each point to a scanner. Have in mind that a point can be seen by two scanners, especially if there is no physical barrier between them. In this case it is difficult to exactly

[1] Also known as monoscan point cloud.

[2] A point cloud format that *ContextCapture™* wants to support and which does not store scanner locations in its metadata.

know which scanner produced the point. But, as this point-scanner attribution is only required for normal orientations, we only need to know which scanner best sees it.

5.1.2 Objective

As a final step toward supporting point clouds without scanner locations, the algorithm must:

- take as input any 3D point cloud captured from static scanners as well as the location of all scanners,
- be invariant to differences between scanners, such as: density, noise, rotation angle,
- find for each point the scanner which best sees it, not necessary the scanner which produced it,
- have a reasonnable running time.

As for *ScanFinder* described in Chapter 4, the algorithm is not expected to work with mobile point clouds.

5.2 Related work

This section highlights some previous work (Section 5.2.1) on the subject of *Point cloud visibility* and describes two previously-published algorithms and their results.

5.2.1 Previous work

Visibility in point clouds is a topic that experienced several publications since the 1960s [App68, SSS74, FST92, GKM93, BW03]. Some methods solve this problem in a 3D rendering context by estimating normals and reconstructing surfaces [SP04, SPL04, WS05, WK04]. This contrasts with our case as we need visible points in order to reconstruct surfaces. Another common approach is to use z-buffering techniques based on point depths [DVS03] to reconstruct surfaces in a real-time rendering context. Even if this approach can be adapted to compute visibility of all points from all scanner location viewpoints, it is not robust to noisy point clouds.

One elegant approach computes point cloud visibility without surface reconstruction: [KTB07]. This approach is invariant to point cloud density and only point coordinates are needed (no normal estimation). It uses simple operations: an inversion followed by a convex hull computation. However, it is not robust enough against point cloud noises. An improved version [MTSM10] has been published, it improves handling of noise and concave surfaces in point clouds. We tried both algorithms. They are described in Section 5.2.2 and Section 5.2.3 as well as the obtained results.

5.2.2 *Direct Visibility of Point Sets*

Overview

The purpose of [KTB07] is to discard directly from the point cloud, which point is *hidden*. The problem being solved is: Given a set of points P (considered a sampling of continuous surface S) and a viewpoint C, determine the points in P visible from C. They introduced the HPR (Hidden Point Removal) Operator. This operator uses two simple operations:

- an inversion
- a convex hull computation

A spherical inversion based on the depth of the points put the nearest points (must of them visible) further away. The second step on the right, based on a convex hull computation, detects visible points. A point lying on this convex hull is considered as visible. Even if a visible point is far in the real domain and closer in the inverted domain it should be on this convex hull, as long as it is visible – no points behind it, no points hidding it.

Results and discussions

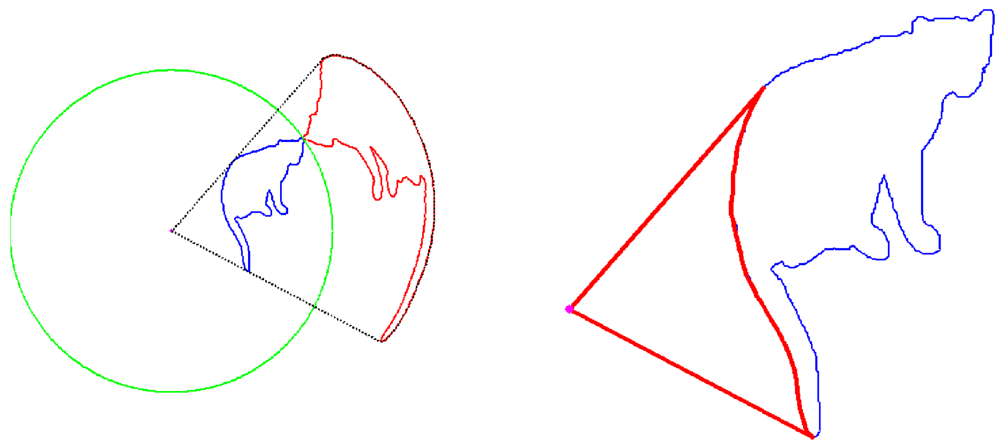


Figure 5.1: HPR operator: on the left a spherical flipping (in red) of a curve (in blue) using a sphere (in green), on the right: a back projection of the convex hull.

Figure 5.1 shows both operation results. On the left is performed a spherical flipping (the inversion) of the point cloud, centered at the viewpoint C. In this example, the ray R of the sphere seems to be $R_{\max}$: *the distance from C to the furthest point.* However, the HPR authors suggest to use a second viewpoint (the opposite of the current viewpoint, its reflection about center of mass of the point cloud) and vary R while maximizing the number of points considered visible by a unique viewpoint. Figure 5.2 shows on the same plot, for a particular

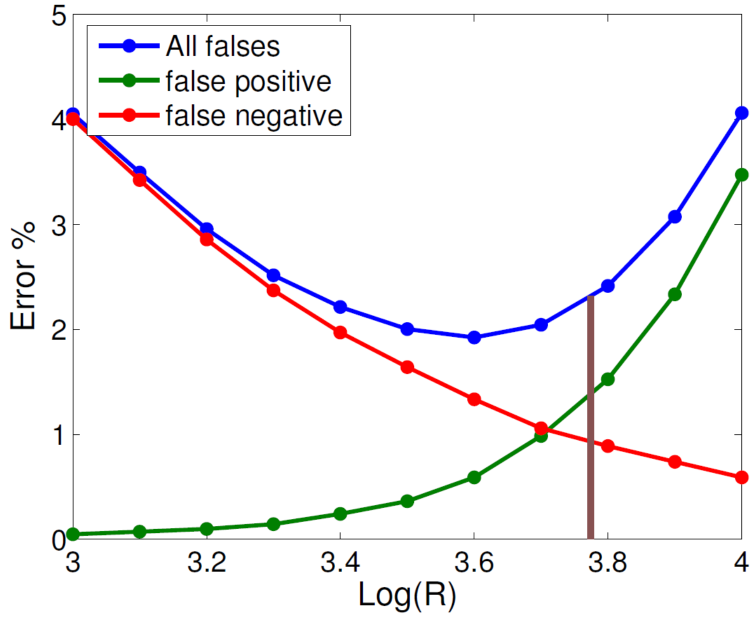


Figure 5.2: The variation of, in one hand the spherical inversion ray R and in the other, the percentage of visibility detection error.

point cloud, the estimated $R = R_{\text{opt}}$ (in brown) and the percentage or error of the method while varying R.

Although this method is simple, has only meaningful computations and an interesting complexity $\mathcal{O}(n \log n)$, it is not robust enough against noisy point clouds, in particular LiDAR ones. With a small R, visible points may be marked as non-visible by HPR. On the contrary, with a large R, non-visible points may be considered as visible. Moreover, this method handles poorly concave forms; it requirs a low curvature in case of concavity.

5.2.3 *Visibility of Noisy Point Cloud Data*

As previously said, the *Visibility of Noisy Point Cloud Data* [MTSM10] paper describes some improvements of *Direct visibility of point sets* [KTB07] for a better handling of noise and concave surfaces in point clouds.

Overview

Noise in point clouds can be observed through perturbations of the convex hull. A noise is a movement of a point toward a direction. A noisy point cloud can be expressed as:

$$P^{\sigma} = \{p_i + \sigma n_i | p_i \in P\}$$

where n_i is a unit vector oriented in a uniformy chosen random direction and σ a uniform random variable over the rand $[0, a]$. The [MTSM10] paper explains some experiences and mathematical proof leading to:

Figure 5.3: The guard zone (in yellow): a region in space from which the visibility cannot be reliably estimated.

- a guard band on the convex hull which helps to consider as visible all points closer than $2 \times \epsilon_{\text{max}}$ to the convex hull. The value of ϵ_{max} is the maximum noise of P^{σ} in the inverted domain.

$$\epsilon_{\text{max}} = \left(\frac{4R}{a_{\text{min}} - \sigma} - 1 \right) \sigma$$

  where a_{min} is the distance from C and the closest point in P.

- a boundary on the R value that must be respected:

$$m a_{\text{max}} \leq R \leq \left(\frac{\alpha D}{2\sigma} + 1 \right) \left(\frac{a_{\text{min}} - \sigma}{4} \right)$$

  where $D = a_{\text{max}} - a_{\text{min}}$ is the diameter of the point cloud and m a value describing the surface form. When $m = 1$ the local region is convex, if $m > 1$ it is concav.

- A guard zone around the noisy point cloud that rejects viewpoints closer than the treshold. This threshold can be observed in Figure 5.3. It depends on the point cloud, this is just a particular case.

$$a_{\text{min}} \geq \frac{\left(4m + \frac{\alpha}{2}\right) D + \alpha}{\left(\frac{\alpha D}{2\sigma} - (4m - 1)\right)}$$

- an iterative method for concavity robustness. It varies R value within the range. For each R value it computes points visibility and update points weight based on the number of times they are tagged as visible. The idea is that high curvatures becore visible at higher values while others (convex, oblique, planar) are consistently visible. Thi is the function f used in the Algorithm 3.

The algorithm is shown in Algorithm 3. Note that the function f (which computes the ray R) is not formally written. It is the last bullet point above on the improvements of [MTSM10] over [KTB07]. Instead it is describe above. Of course, the following algorithm is called for each scanner location C. When a point is seen by multiple scanners, we choose the closer one.

Function `RobustHPR`(P, C)**:**

Input:

- a set of points $P = \{p_i \mid i \in [0, s]\}$,
- the viewpoint C.

Output: a the set V of visible points

$P' \leftarrow \emptyset$ `// contains inverted points`
$\Delta' \leftarrow \emptyset$ `// contains inverted hidden points`
$R \leftarrow f(P, C)$`// Compute the spherical inversion ray` R

`// spherical inversion`
foreach $p_i \in P$ **do**
  $p'_i \leftarrow p_i + 2\,(R - \|p_i\|)\,\frac{p_i}{\|p_i\|}$
  $P' \leftarrow P' \cup \{p'_i\}$

`// compute the convex hull of` $P' \cup \{C\}$ `in order to put aside visible points`
$\Delta' \leftarrow \text{convex\_hull}(P' \cup \{C\})$

`// Catch all points within a` $2 \times \epsilon_{\text{max}}$ `distance form the convex hull points`
$\epsilon_{\text{max}} \leftarrow \left(\frac{4R}{a_{\text{min}} - \sigma} - 1\right)\sigma$
foreach $p'_i \in \Delta'$ **do**
  foreach $p'_j \in \Delta'$ belonging to the 50 nearest points of p'_i **do**
    if $\left\|p'_i - p'_j\right\| \leq 2\epsilon_{\text{max}}$ **then**
      $\Delta' \leftarrow \Delta' \cup \{p'_j\}$

`// Back projection of hidden points`
$\Delta \leftarrow \{p_i \in P \mid p'_i \in \Delta'\}$

return $P \setminus \Delta$

Algorithm 3: The robust HPR algorithm.

Results and discussions

Figure 5.4 shows some results comparing this method (Robust HPR) with the original methods, one using $R = R_{\text{max}}$ and the other $R = R_{\text{opt}}$. It may be observed that Robust HPR achieves better results than the others. In our case, this has also been observed. However, it is still not robust enough against LiDAR point cloud noises. This is what leads us toward a custom approach described in Section 5.3.

FIXME: Add pictures of the algorithm on Monday!!!

5.3 A custom disk-based approach

This section describes our custom *disk-based* approach to resolve visibility in point clouds.

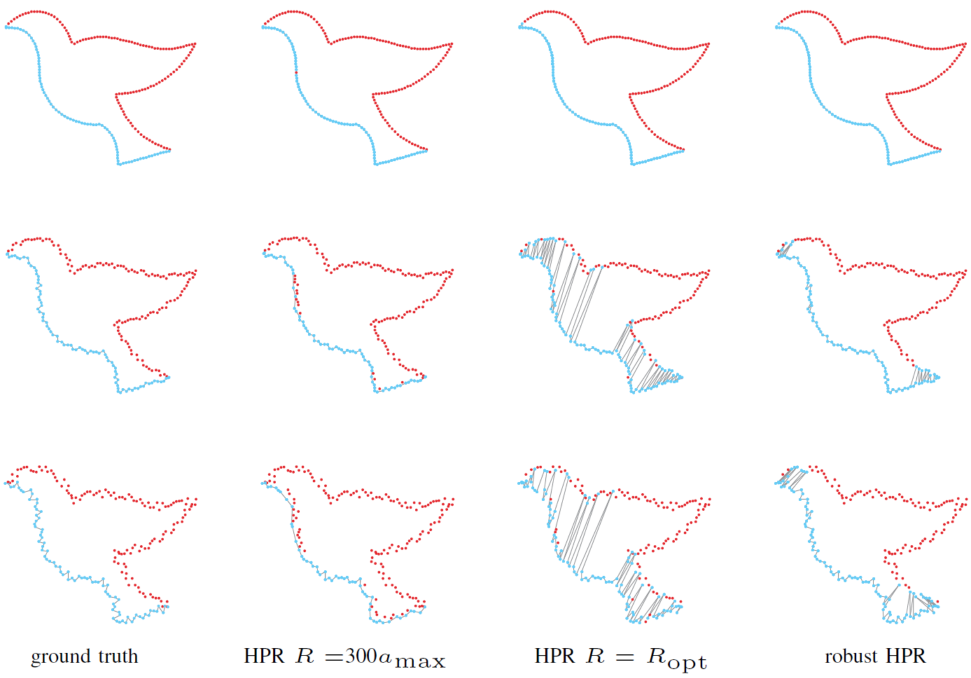


Figure 5.4: Result comparing HP$R_{\max}$, HPR_{opt} and Robust HPR

5.3.1 Overview

This method reduces the point cloud visibility problem to ray casting and intersection computations by assigning to each point p_i a *disk*. A *disk* is defined as a tuple $\langle p_i, r_d, \vec{n} \rangle$. For a specific point p_i, here is how the associated disk is computed:

- p_i is the center of the disk,
- r_d is the the mediane of the distances to the 50 nearest points of p_i,
- $\vec{n}$ is the normal at p_i computed via a PCA (Section 3.2.1) applied on the 50 nearest points of p_i, including itself.

These disks are computed as a preprocessing step. In what follows, all computations are made on these disks. Our custom approach can be compared to ray tracing as for each point p_i of P and each viewpoint c_k it casts a ray r_a, computes all intersections and use them to decide which scanner c_k must be associated to p_i. We use the Eigen library [GJ+10] for collisions computation. As our Disks are not in the basic Types of Eigen, a main contributer to ther Eigen library helped us to define our custom Disk data type and how the intersection should be computed using Eigen algorithms. Basically, an intersection of a ray r_a and a disk d_j happens when there exists a projection of the projection of r_a and when the distance of this projection from the disk center is smaller than its ray r_d. This ray casting is perfomed from each viewpoint $c_k \in C$ − scanner locations. Each time a ray is casted, to a specific point p_i, from each viewpoint c_k, a tuple $\langle v, n, d, k \rangle$ is kept, which contains:

- a boolean v telling if p_i is visible from c_k,
- the number n of collisions of r_a,
- the distance d from p_i and c_k,
- the scanner identifier k.

For each point, we sort those tuples. When a tuple $\langle v_1, n_1, d_1, k_1\rangle$ is compared to another tuple $\langle v_2, n_2, d_2, k_3\rangle$, if only one of them has a positive v, it comes before the other. Otherwise, if a tuple has at least two collisions less than the other, it comes before. Finally, if p_i is visible from multiple viewpoints which have almost the same number of collision when casting their respective ray, we associate the closest scanner to p_i. The sorting criteria have been found empirically. After sorting out the tuples for a specific p_i, the value k of the first tuple is the scanner identifier associated to p_i.

5.3.2 Implementation

Algorithm 4 shows our custom algorithm for computing visibility in point clouds, especially LiDAR ones. As explained in Section 5.3.1, we iterate over all points, all scanners, casts a ray between each point and scanner and compute collisions. Actually, we use an improved version of this algorithm which assign different disks to each point, varying the disk ray. The ray is casted with each disk sizes. This slows the algorithm in a significant way but helps to have better visibility results.

5.3.3 Results and discussions

Figure 5.5 and Figure 5.6 show two different views of the result of the custom visibility algorithm applied on a point cloud P_1. In general the method seems to work well. Points near one scanner are coloured with a unique color. Figure 5.6 shows a particular case where the method works well. Assume the current viewpoint as being at the scanner associated to the red color. The algorithm detects that a wall far away (in blue) is hidden by a closer wall (in red). Figure 5.7 and Figure 5.8 shows two other views of the result obtained on another point cloud P_2. Table 5.1 shows the percentage of error of our method applied on different point clouds. As you can see, it has less that 3 percent of error on those point clouds. In summary, this custom approach meets all of our requirements instead of the running time. For instance, it almost took 6 days to compute visibility of a point cloud named Big Stadium. Figure **??** (FIXME) shows a complete reconstruction of this point cloud. This algorithm was written as a prototype in order to validate our intuitions. Several improvements can be made on the algorithm, such as some tasks parallelization. The algorithm will certainly be optimized before being integrated to *ContextCapture*^{TM} .

```
Function DiskBasedVis(P, C):
    Input:
        • a set of points P = {p_i | i ∈ [0, s]},
        • a set of viewpoints C = {c_k | k ∈ [0, s_c]}.
    Output: a visibility Map ⟨i, j⟩ where i and j are respectively point and scanner unique identifiers.

    R ← ∅
    foreach p_i ∈ P do
        V_info ← ∅ /* Vector of ⟨v, n, d, k⟩ where n is the number of intersection, d the distance between p_i and
            c_k, and finally j the viewpoint identifier.                                                   */
        Δ ← ∅ /* Vector of disk collisions ⟨p_j, n⃗_j⟩ where p_j is the collided point− center of its associated
            disk and n⃗_j the disk's normal                                                                 */

        foreach c_k ∈ C do
            r_ray ← p_i − c_k // the ray/vector from c_k to p_i
            d_ik ← ||r_ray|| // distance between the point p_i and the viewpoint c_k
            v ← ⊤ // set to ⊤ as p_i is considered visible at the beginning
            Δ ← intersected_disks(r, P, C)

            foreach {p_j, n⃗_j} ∈ Δ do
                d_ij ← ||p_j − p_i|| // distance between the point p_i and the collided point p_j
                d_kj ← ||c_k − p_j|| // distance between the collided point p_j and the viewpoint c_k
                if v ∧ d_ij < d_ik ∧ ¬(d_kj ≤ 0.1 ∧ |n⃗_j × n⃗_i| > 0.6) then
                    v ← ⊥
                    break
            V_info ← V_info ∪ {⟨v, size_of(Δ), d_ik, j⟩}

        V_info ← sort(V_info) // sort V_info based on v, size_of(Δ) and d_ik
        v ← first element ofV_info
        R ← R ∪ {⟨i, v_j⟩}

    return R
```

Algorithm 4: A custom disk-based approach for visibility in point clouds.

5.3.4 Complete reconstruction using *ScanFinder* and *DiskBasedVisibility*

FIXME: Add pictures.

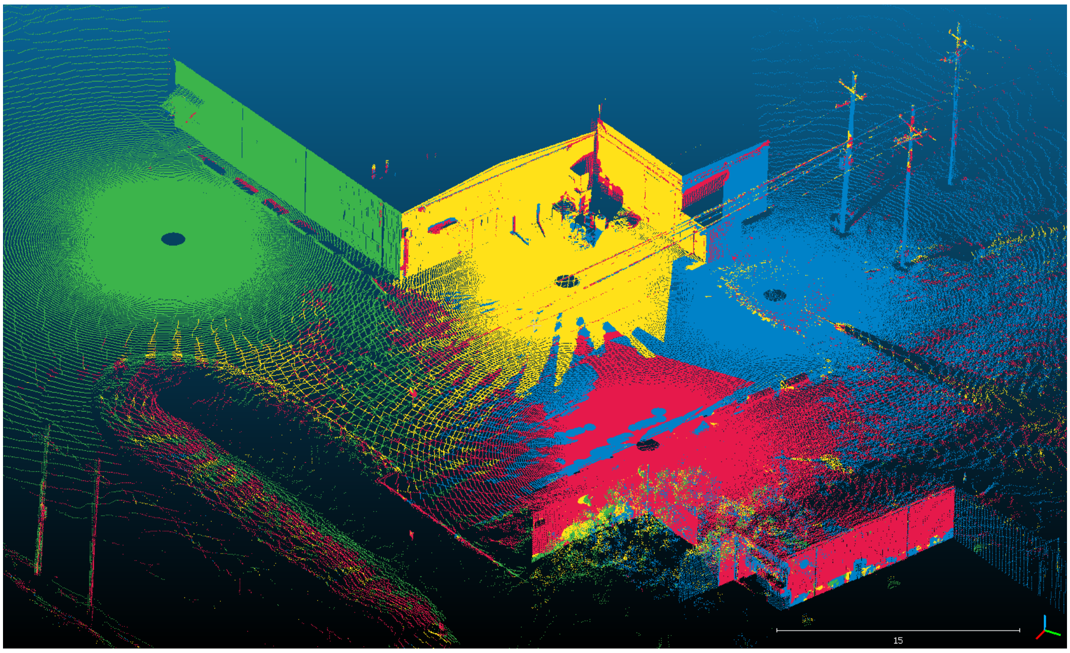


Figure 5.5: A view of the custom visibility algorithm result on a point cloud P_1: points colored with the same color are assigned to the same viewpoint–scanner location. There are four (4) colors, thus four (4) scanners in the point cloud.

Point Cloud	Number of points	Error (percent)
Pump:	604,638	0.19631
Parking:	5,188,752	2.70431
Road:	2,220,166	1.75293
Ashlan8+11	23,484,370	0.69309

Table 5.1: The percentage of error of our Disk-based approach on different point cloud sets with their respective sizes.

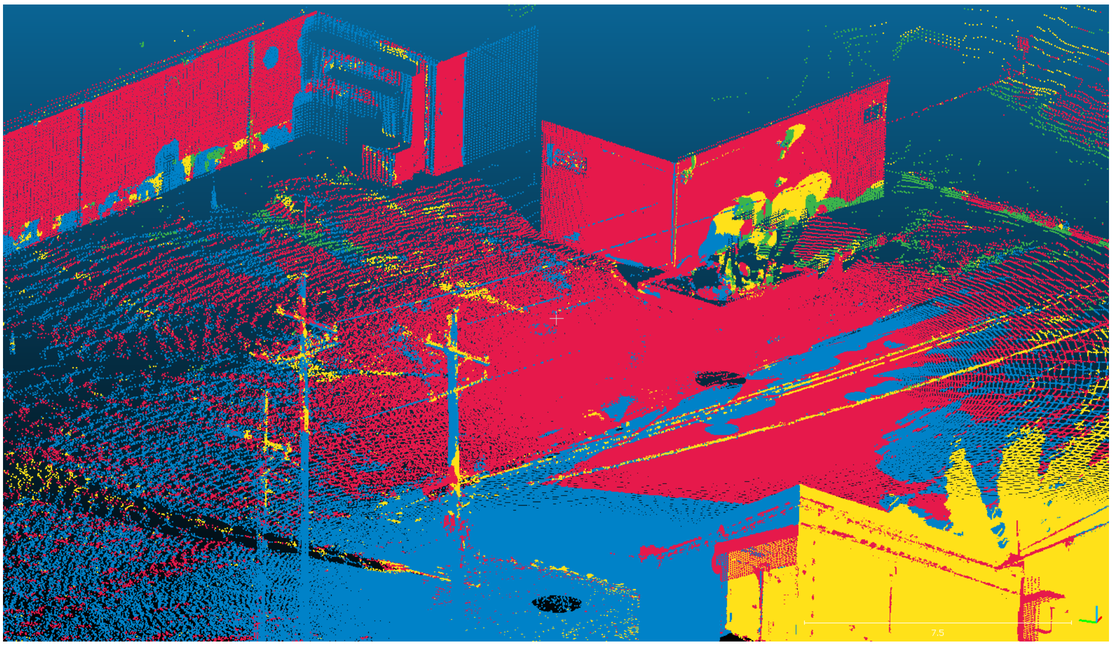


Figure 5.6: Another view of the custom visibility algorithm result on the same point cloud P_1: points colored with the same color are assigned to the same viewpoint–scanner location. There are four (4) colors, thus four (4) scanners in the point cloud.

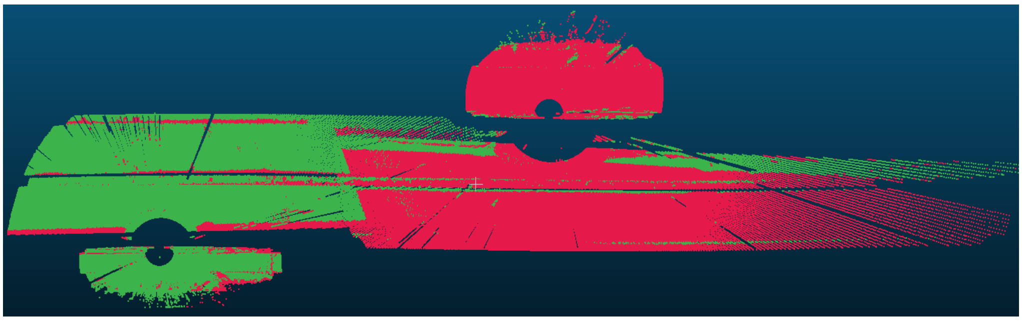


Figure 5.7: A view of the custom visibility algorithm result on a point cloud P_2: points colored with the same color are assigned to the same viewpoint–scanner location. There are two (2) colors, thus two (2) scanners in the point cloud.

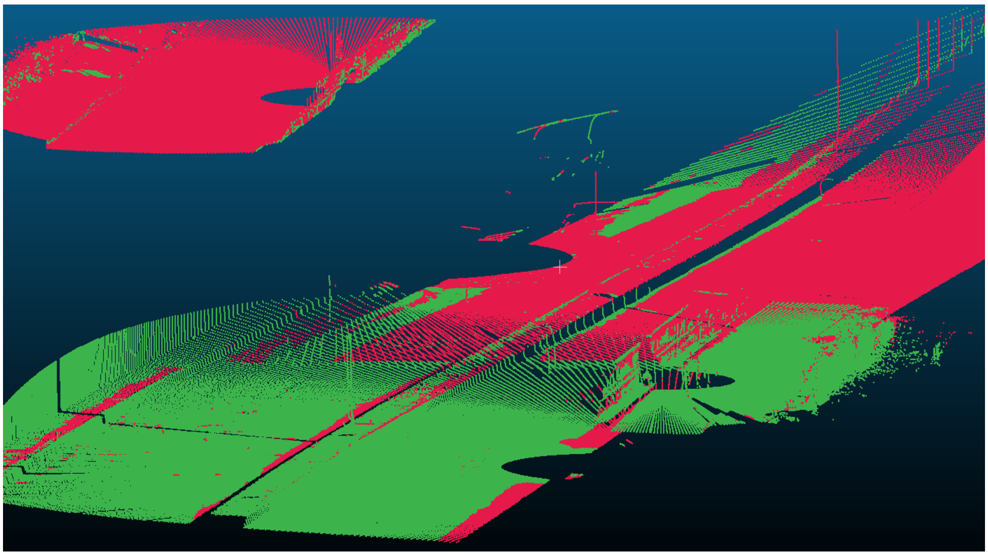


Figure 5.8: Another view of the custom visibility algorithm result on the same point cloud P_2: points colored with the same color are assigned to the same viewpoint–scanner location. There are two (2) colors, thus two (2) scanners in the point cloud.

Chapter 6

Point Cloud Compression

6.1 Specifications

6.1.1 Problem being addressed

6.1.2 Objective

6.1.3 Scope

6.2 Related work

6.3 A custom arithmetic approach

6.3.1 Overview

6.3.2 Implementation

6.3.3 Comparison with Brotli, 7Z and Zip

6.4 Integration

Chapter 7

Conclusion

7.1 Summary of Internship Achievements

Summary.

7.2 Applications

Applications.

7.3 Future Work

Future Work.

Bibliography

[AMO] Sameer Agarwal, Keir Mierle, and Others. Ceres solver. `http://ceres-solver.org`.

[App68] Arthur Appel. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 37–45. ACM, 1968.

[ben] Bentley systems by wikipedia. `https://en.wikipedia.org/wiki/Bentley_Systems#History`. Accessed: 2018-07-06.

[BW03] Jiří Bittner and Peter Wonka. Visibility in computer graphics. *Environment and Planning B: Planning and Design*, 30(5):729–755, 2003.

[DVS03] Carsten Dachsbacher, Christian Vogelgsang, and Marc Stamminger. Sequential point trees. In *ACM Transactions on Graphics (TOG)*, volume 22, pages 657–662. ACM, 2003.

[FST92] Thomas A Funkhouser, Carlo H Sequin, and Seth J Teller. Management of large amounts of data in interactive building walkthroughs. In *Proceedings of the 1992 symposium on Interactive 3D graphics*, pages 11–20. ACM, 1992.

[GJ+10] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[GKIS05] Stefan Gumhold, Zachi Kami, Martin Isenburg, and Hans-Peter Seidel. Predictive point-cloud compression. In *ACM SIGGRAPH 2005 Sketches*, page 137. ACM, 2005.

[GKM93] Ned Greene, Michael Kass, and Gavin Miller. Hierarchical z-buffer visibility. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 231–238. ACM, 1993.

[HDD+92] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. *Surface reconstruction from unorganized points*, volume 26. ACM, 1992.

[KC15] Alex Kendall and Roberto Cipolla. Modelling uncertainty in deep learning for camera relocalization. *arXiv preprint arXiv:1509.05909*, 2015.

[KGC15] Alex Kendall, Matthew Grimes, and Roberto Cipolla. Posenet: A convolutional network for real-time 6-dof camera relocalization. In *Proceedings of the IEEE international conference on computer vision*, pages 2938–2946, 2015.

[KTB07] Sagi Katz, Ayellet Tal, and Ronen Basri. Direct visibility of point sets. In *ACM Transactions on Graphics (TOG)*, volume 26, page 24. ACM, 2007.

[lid] What is lidar? `https://www.3dlasermapping.com/what-is-lidar-and-how-does-it-work/`. Accessed: 2018-07-06.

[MTSM10] Ravish Mehra, Pushkar Tripathi, Alla Sheffer, and Niloy J Mitra. Visibility of noisy point cloud data. *Computers & Graphics*, 34(3):219–230, 2010.

[NS17] Yoshikatsu Nakajima and Hideo Saito. Robust camera pose estimation by viewpoint classification using deep learning. *Computational Visual Media*, 3(2):189–198, 2017.

[ran] Ransac. `http://www.math-info.univ-paris5.fr/~lomn/Cours/CV/SeqVideo/Material/RANSAC-tutorial.pdf`. Accessed: 2018-07-06.

[SK06] Ruwen Schnabel and Reinhard Klein. Octree-based point-cloud compression. *Spbg*, 6:111–120, 2006.

[SP04] Miguel Sainz and Renato Pajarola. Point-based rendering techniques. *Computers & Graphics*, 28(6):869–879, 2004.

[SPL04] Miguel Sainz, Renato Pajarola, and Roberto Lario. Points reloaded: Point-based rendering revisited. *SPBG*, 4:121–128, 2004.

[SQLG15] Hao Su, Charles R Qi, Yangyan Li, and Leonidas J Guibas. Render for cnn: Viewpoint estimation in images using cnns trained with rendered 3d model views. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2686–2694, 2015.

[SSS74] Ivan E Sutherland, Robert F Sproull, and Robert A Schumacker. A characterization of ten hidden-surface algorithms. *ACM Computing Surveys (CSUR)*, 6(1):1–55, 1974.

[TM15] Shubham Tulsiani and Jitendra Malik. Viewpoints and keypoints. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1510–1519, 2015.

[WK04] Jainhua Wu and Leif Kobbelt. Optimized sub-sampling of point sets for surface splatting. In *Computer Graphics Forum*, volume 23, pages 643–652. Wiley Online Library, 2004.

[WS05] Ingo Wald and H-P Seidel. Interactive ray tracing of point-based models. In *Point-Based Graphics, 2005. Eurographics/IEEE VGTC Symposium Proceedings*, pages 9–16. IEEE, 2005.