This work was submitted at the
Institute for Railway Vehicles and Transport Systems

Master Thesis

Lidar-based Detection of UIC-Hook using Point Cloud Library in ROS

submitted by

Santamaria, Alvaro

INSTITUTE FOR RAILWAY VEHICLES AND TRANSPORT SYSTEMS
UNIV.-PROF. DR.-ING. C. SCHINDLER

Master Thesis

Santamaria, Alvaro

Matriculation number: 433400

Subject:

Lidar-based Detection of UIC-Hook using Point Cloud Library in ROS

Abstract: This thesis presents the software necessary for the detection and recognition of objects within point clouds, with special attention to the position of the UIC-Hook in the railway environment. The procedures and methods necessary for cloud processing are explained and implemented in C++ using the libraries available in Point Cloud Library and Robot Operating System. Finally, the methods considered are verified and validated using data from the process simulation through the Digital Twin process.

This work was carried out the Institute for Railway Vehicles and Transport Systems in Aachen.

Faculty supervisor: Univ.-Prof. Dr.-Ing. Christian Schindler

Hyun-Suk Jung, M.Sc.

Place and date of submission: Aachen, 30th September 2022

Eidesstattliche Versicherung

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Master Thesis mit dem Titel

Lidar-based Detection of UIC-Hook using Point Cloud Library in ROS

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht.

Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Ort, Datum	Santamaria, Alvaro

Belehrung:

§ 156 StGB: Falsche Versicherung an Eides Statt

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

Ort, Datum	Santamaria, Alvaro

The present translation is for your convenience only.
Only the German version is legally binding

Statutory Declaration in Lieu of an Oath

I hereby declare in lieu of an oath that I have completed the present Master Thesis entitled

Lidar-based Detection of UIC-Hook using Point Cloud Library in ROS

independently and without illegitimate assistance from third parties. I have use no other than the specified sources and aids. In case that the thesis is additionally submitted in an electronic format, I declare that the written and electronic versions are fully identical. The thesis has not been submitted to any examination body in this, or similar, form.

ALVARO SANTAMARIA GOMEZ CARREÑO - NIF:48608820E

Firmado digitalmente por ALVARO SANTAMARIA GOMEZ CARREÑO - NIF:48608820E
Fecha: 2022.10.19 16:07:39 +02'00'

Location/City, Date

Santamaria, Alvaro

Official Notification:

Para. 156 StGB (German Criminal Code): False Statutory Declarations

Whosoever before a public authority competent to administer statutory declarations, falsely makes such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment not exceeding three years or a fine.

Para. 161 StGB (German Criminal Code): False Statutory Declarations Due to Negligence

(1) If a person commits one of the offences listed in sections 154 to 156 negligently the penalty shall be imprisonment not exceeding one year or a fine.

(2) The offender shall be exempt from liability if he corrects his false testimony in time. The provisions of section 158 (2) and (3) shall apply accordingly.

I have read and understood the above official notification:

City, Date

Santamaria, Alvaro

Index

List of abbreviations

CV: Computer Vision.

DoG: Difference of Gaussians.

DT: Digital Twin.

EVD: EigenValue Decomposition.

FLANN: Fast Library for Approximate Nearest Neighbors.

GHT: Generalized Hough Transform.

GRF: Global Reference Frame.

HT: Hough Transform.

HV: Hough Voting.

ICP: Iterative Closest Point.

ISS: Intrinsic Shape Signatures.

LRF: Local Reference Frame.

NSS: Nearest Neighbor Search.

PC: Point Cloud.

PCL: Point Cloud Library.

PDT: Product Digital Twin.

QIP: Quality Improvement Methodology.

ROS: Robot Operation System.

RViz: ROS Visualizer.

SIFT: Scale Invariant Feature Transform.

SHOT: Signature of Histograms of OrienTations.

1. Introduction

Nowadays, rail transport is one of the leading transport methods for raw materials, e.g., in the chemical industry or metallurgy. This is because it is still the most efficient method of transport for moving heavy and difficult transport goods over long distances, where the associated transport cost represents a high percentage of the product's total price [1]. This is possible due to combining different goods in the same shipment and the availability of specialized containers for each type of goods to be transported.

The last few years have seen a revolution in machine vision, allowing the automation of tasks that were previously impossible to automate. The fields of application of these new techniques range from the industrial area to medicine, even opening up new areas of application such as agriculture [2].

Machine vision analyzes images or point clouds using different techniques to help solve industrial problems with a high visual component [2]. Within machine vision there are two terms, computer vision and artificial vision. It is important not to confuse both, which are part of machine vision. According to the Automated Imaging Association (AIA), the first one focuses on the deep analysis of images or point clouds to extract as much detail as possible. On the other hand, computer vision does not need to analyze the details, i.e. computer vision in an autonomous car only needs to detect obstacles on the road to avoid collisions [3].

In short, the introduction of machine vision in the different processes has as its main objective the reduction of costs (increasing efficiency and reducing errors). In addition the possibilities of machine vision to free workers from dangerous, repetitive or tiring, activities must also be taken into account [2].

The aim of this thesis is to develop a vehicle that performs fully automatic coupling and transport of wagons, this vehicle, inevitably, needs the ability to detect the availability of the wagon to be coupled and the characteristics of the wagon. For this purpose, in the case of the robot studied in this thesis, a LiDAR sensor located at the front of the vehicle and a point cloud processing algorithm will be used. The algorithm must be able to properly identify the wagon elements and transmit this information to the rest of the system. Point cloud processing and recognition is a relatively new technology, but nevertheless, a multitude of articles have already been published concerning the application of computer vision (CV) in transportation fields. In particular, the goal is to analyze the rear ends of the wagons which are conformed by elements standardized in DIN, such as the Berne rectangle and the UIC hook. In addition, it is expected that the software used for the processing of the Point Cloud Library (PCL) will be able to communicate the state of the wagon to the software in charge of the general handling of the robot, Robot Operating System (ROS). This will result in a fully integrated and functional system. To achieve this, numerous point cloud processing algorithms will be analyzed to evaluate the suitability of each one for the task. With them, the criteria and reasoning for the use or not of the different possibilities will be raised.

2. State of the art

2.1 Automation of railway shunting operations

Process automation contributes to higher productivity rates, more efficient use of resources, better product quality and, most importantly, increased safety by reducing the interaction between robots and machinery [4].

Currently, handling goods wagons and locomotives at railway hubs and maintenance facilities is often carried out with the help of remote-controlled robots. There are two main ways of operating robots for wagon distribution and transport tasks. The most classic and basic one is the robot shunting operated by a human through a cabin incorporated in the robot itself, with human intervention being necessary not only for driving but also for coupling with the elements to be transported, which means a safety risk for the workers operating the robot [5]. On the other hand, in recent years remotely controlled robots have been developed, that meaning considerable reduction in human intervention in the docking process [6]. With this new technology, only one manual step is required to engage and disengage the braking system, with all other necessary steps being carried out automatically. Nevertheless, the technology still requires the presence of a human in relatively proximity to the dangerous shunting process [7].

Nowadays, automation in this sector continues to develop, mainly for two reasons. Firstly, to eliminate the presence of human workers in a dangerous environment, with the robots being fully autonomous in performing the task of coupling with the wagons. Secondly, the elimination of the human variable from the process, allows to reduce the coupling times and to know them exactly in advance [8].

2.2 Standard for the UIC-Hook

The components of a railway carriage are standardized, i.e. their dimensions, surfaces and materials to be used are predefined by a DIN or EN standard. Therefore, the variability of any component of a German railway carriage will be covered by the correspondent standard. This makes it possible to know in advance all the possible states in which the wagon may be at the time of shunting. The elements that contain the most significant variability and importance for the state of the wagon is the coupling hook as shown in the central part of the Figure 1.

Figure 1: Visual example of the elements that make up the rear of a train wagon [7].

The element that defines the availability for the shunting operation is the UIC hook (UIC, French: *Union Internationale des Chemins de fer*, International Union of Railways). The standard defining the shape of the hook is EN 15566. This standard, as seen in Figure 2, specifies both the dimensions of the hook, which are the same for all European rail wagons.

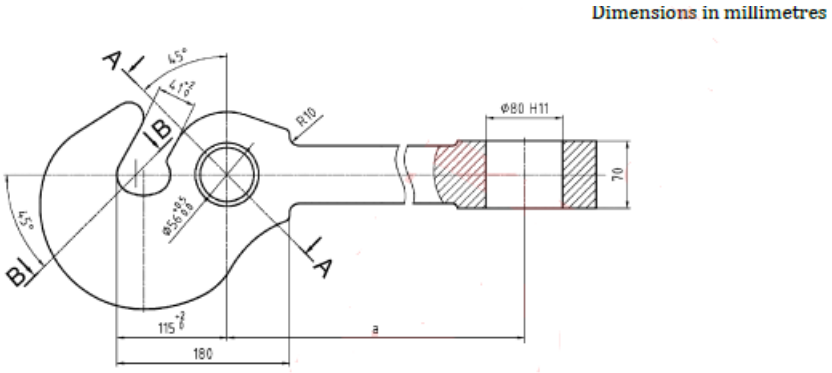


Figure 2: UIC-Hook dimensions EN 15566 [9].

A chain is attached to the UIC-Hook, which is used for coupling with other train wagons. The dimensions of this chain are defined in the same standard.

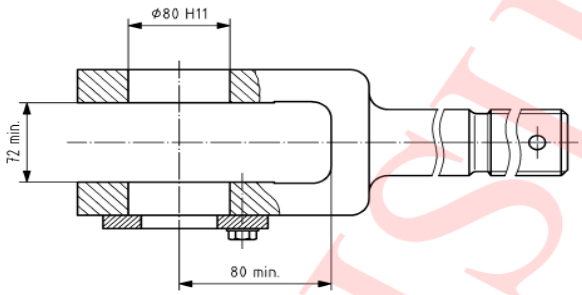


Figure 2: Chain dimensions EN 15566 [9].

2.3 Computer Vision

The purpose of Computer Vision (CV) is to mimic the behaviour of the human sense of vision. By duplicating this sense, it enables machines to have the ability to process images and point clouds and recognize elements in them.

Nowadays, CV is widely used in a multitude of fields, from autonomous driving of cars and trucks (recognizing signs, other cars, pedestrians and road layout) to the medical field (recognizing the results of radiological studies). Computer vision is therefore becoming increasingly important in the automation of all kinds of processes.

Within the CV, there are four different approaches for processing and interpreting images. One or a combination of them can be used in a project.

1. Recognition: The algorithm identifies and interprets elements of the images. For example, the recognition of a stop signal in an autonomous car application.

2. Reconstruction: Using data from visual sensors, the computer is able to recognize different perspectives in an image.

3. Registration: The computer puts by means of a homogeneous transform different sets of data in the same reference frame.

4. Reorganizing: This last approach labels the different elements. This prevents the possible interaction between the elements detected in the image from affecting the processes.[17]

2.3.1 Image based

Image-based Computer Vision is based on photogrammetry. This technique consists of obtaining a 3D scale model from a set of photographs taken from different positions. These images are processed to get a 3D mesh. This grid consists of a three-dimensional representation obtained through the "fusion" of the photos.

Figure 3: 3D mesh obtained through photogrammetry [10]

The main application of photogrammetry is the processing of large surfaces, such as the one shown Figure 4. In addition, 3D meshes have a high quality in the textures of the elements that make up the images. However, the accuracy in detecting object surfaces is limited [11].

2.3.2 Point Cloud Based

A point cloud is a replica of the physical elements created from a set of data measured in the cartesian space. It is a set of thousands of points defined by the three Cartesian dimensions and, additionally, other attributes such as RGB values (indicators of the colour of each point) or the light intensity at the point. To obtain the cartesian coordinates of the cloud, the detection point of the sensor is used as the origin of coordinates [12].

Point clouds are, as the name implies, composed of point information. Therefore, they are not continuous information but, as shown in the figure 5, discontinuous. Logically, as the density of the cloud increases, so does the quality of the representation.

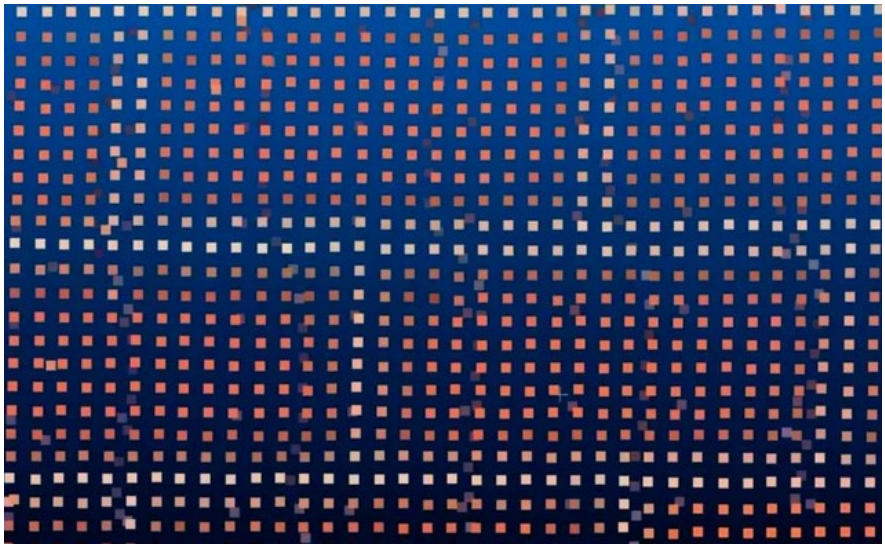


Figure 4: Example of a zoomed Point Cloud [13]

Light Detection and Ranging (LiDAR) technology is used for cloud detection. This sensor, using a pulsed laser, measures the distance between the sensor and the surrounding objects. In addition to measuring distance, it can also measure other characteristics, such as those mentioned above. Scanning can be done in different ways [12, 13].

Firstly, the capture of the point cloud can be done dynamically, i.e. the LiDAR is mounted on a mobile platform. This scanning has the advantage that it can be used in applications such as road surveillance by helicopter. However, the quality and accuracy of the cloud decrease.

Secondly, when quality and accuracy are crucial factors, the scanning can be performed statically. This information capture methodology is used for engineering applications.

2.4 Process simulation

When an engineering project involves heavy machinery, expensive equipment and a hazard environment to both people and objects, simulation should be considered before experimentation. Simulation is used to create virtual models that can predict the behavior of a process, without having to interact with the actual process itself. The following are some of the advantages of process simulation:

- The behavior of a model can be obtained without having to interact with the real world. In this way, possibly dangerous situations and risks can be avoided.
- It is possible to observe how a process works before its implementation.
- Via simulation, rapid changes can be made once it is available. This advantage allows the continuous improvement of processes.
- It speeds up the experimentation process by being able to test a multitude of possibilities by simply modifying parts of the software.

Despite the many advantages of simulation, there are also disadvantages to be considered:

- A considerable amount of time is required to create and program the model to carry it out.
- The quality of the simulation will depend on the correct setting of boundaries and input data.
- The results will never be 100% accurate.

As mentioned above, the simulation allows the results of the process to be obtained without the need to interact with the real equipment. This, together with the other advantages described, is more than enough reason to justify the use of simulation in the process studied in this master thesis, since expensive machinery is used with a high degree of danger for both

equipment and people. On the other hand, due to the lack of absolute precision in the simulation, the process should be tested in reality in order to verify the results of the simulation and to consider possible situations that have not been taken into account in the previous simulation [14].

2.4.1 Digital Twin

A digital twin (DT) is a virtual representation of physical elements or processes. DTs are used to predict the behaviour of the physical counterpart by simulating them prior to investing in prototypes and physical components.

Figure 5: Illustration of a DT of a car production line [15].

Through the use of multi-physics simulation and data analysis, DTs are able to obtain the impact on the process of design changes, usage scenarios, environmental conditions and [39]many other external variables. This avoids the need for testing and experimentation, which can be hazardous to equipment and people during early stages of development.

The digital twin corresponding to described above is the Product Digital Twin (PDT). This PDT is used to validate the performance of the product while showing how it behaves in the simulated world. With this information, the performance can be analyzed, and any necessary adjustments can be made to program its behaviour as desired. PDT allows the simulation and tuning of complex systems in a simpler and faster way by permitting faster iteration processes. As a result, the overall cost and time required for product development are reduced [16].

The advantages outlined in the previous paragraphs have motivated the creation of a Digital Twin of the process for this project. To achieve this, the two main components of the process, the train carriage and the coupling robot have been created.

On the one hand, as shown in Figure 6, the train carriage has been designed to be as similar as possible to the Real Twin. For this reason, all the elements that make it up, apart from the UIC-Hook, have been included.

On the other hand, for the modelling of the shunting robot, has been designed, like the wagon, in a faithful way to the real robot. The element that is most interesting for this project, however, is the LiDAR sensor, which is located at the front of the robot, below the coupling element. This sensor will obtain the information used to carry out the detection. As explained in previous sections, the information will be obtained as point clouds.

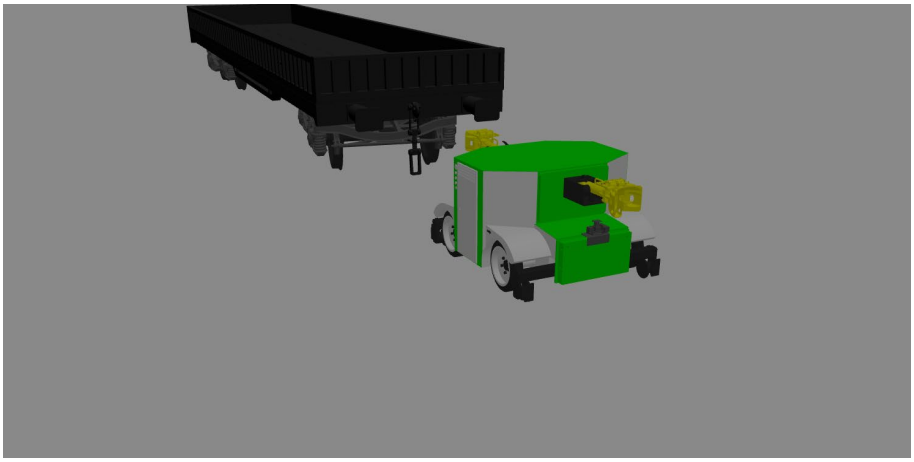


Figure 6: Digital Twin of the coupling process.

2.5 Point Cloud Processing

The point cloud obtained by the sensor must be processed to obtain the output, i.e. the recognition of the desired parts. There are three main phases: pre-processing, registration of the two point clouds and element recognition.

1. **Pre-Processing**: the aim is to simplify the input as much as possible to obtain a higher degree of success and to reduce the processing power required to perform the rest of the steps. As can be seen in the following figure, it is composed of Outlier Removal, Downsampling, Segmentation and Cluster Extraction [17]

2. **Point Cloud Registration**: this section describes how to obtain the homogeneous transform necessary to align the elements of the pre-processed point cloud in the previous section with the different models considered. This alignment is essential to be able to correctly recognize the state of the elements of the point cloud. This makes this section one of the most important, as the quality of the result depends on the precision with which this section is carried out.

   In this section, as described above, two point clouds are processed, one from the LiDAR and the other from the database. However, the steps applied to each of them

are similar. They are, in order of application, Keypoint extraction, Features Estimation, Correspondence Estimation, Correspondence Rejection Method/s and Homogeneous Transform estimation [18].

3. **Point Cloud Recognition**: once the cloud obtained by sensing and the one obtained from the available database are aligned, the recognition of the state of the different elements expected in the wagon is carried out. To do this, a recognition algorithm must be applied to the two aligned clouds. In terms of types of algorithms available to perform the recognition, there are those that require artificial intelligence and those that do not. The ones that use artificial intelligence, require a training phase with a database to "learn" which are the characteristics that define an object. After this phase, the algorithm will be able to distinguish the objects captured by the sensor. Algorithms that do not use artificial intelligence, on the other hand, base the object recognition on the comparison of the point cloud with the different cases stored in the database, thus determining the different correlations [17]. This project will only use algorithms that do not use artificial intelligence since, as the objects that make up the clouds are normalized, significant changes between situations are not contemplated. Therefore, it is sufficient to consider the possible configurations that may occur in the process to compare models with each observed case.

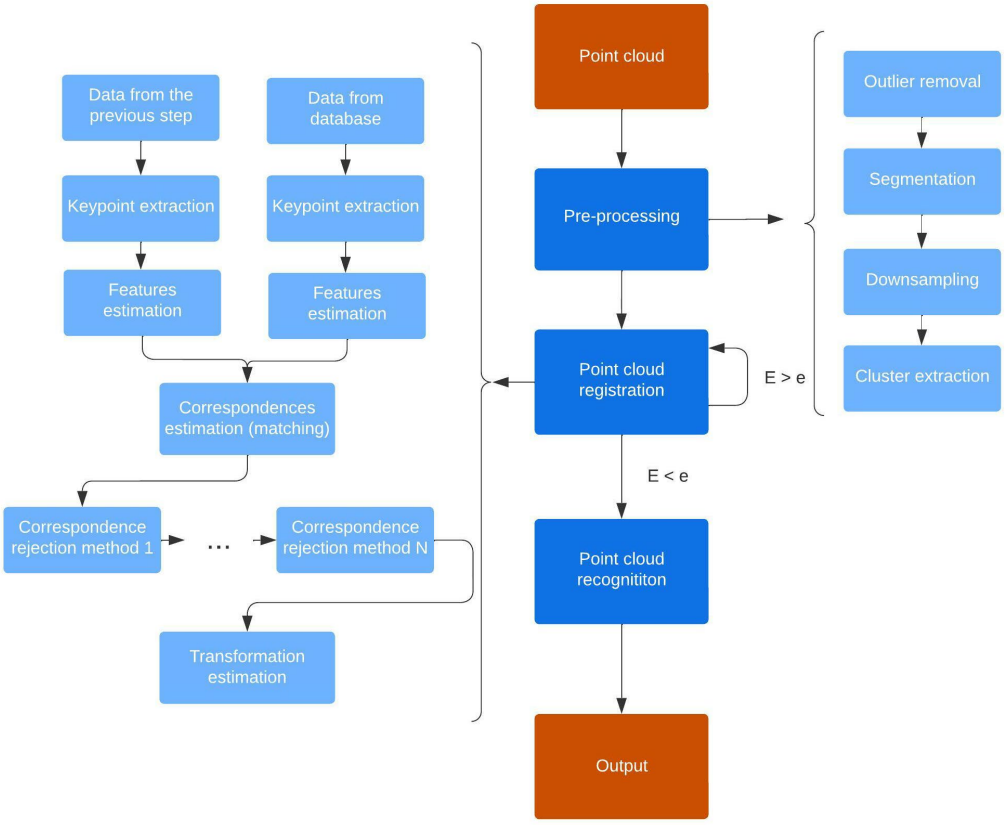


Figure 7: General procedure for cloud point processing [18, 19]

Finally, the purpose of the thesis is not to develop a methodology for the implementation of a particular method. Therefore, all the algorithms selected below fulfil the condition of being currently implemented as a library within PCL.

2.5.1 Pre-processing

First, the point cloud must be simplified and prepared for the subsequent phases. This is done by reducing the number of points to be analysed. In addition, it is also necessary to eliminate those points that do not provide any information either because they are points obtained due to noise in the measurements or because they are points that are not in the place where the relevant information is.

2.5.1.1 Outlier removal

The aim of the outlier removal is primarily to eliminate points that are considered as not interesting for the analysis. These points can be those generated due to noise during measurements or minor elements that are desired to be removed from the point cloud [20]. This process generates a "cleaned" point cloud of unimportant or misleading elements, giving a higher probability of success and reducing processing time for the subsequent steps. The existence of incorrect points can cause errors in the estimation of local cloud characteristics such as surface normals or curvature changes, which can cause a fatal error in the registration of point clouds.

In figure 8, it can be observed the result of applying the outlier removal process to the point cloud plotted in blue. The output is the pink cloud where all the erroneous points (collected in the green point cloud) have been eliminated.

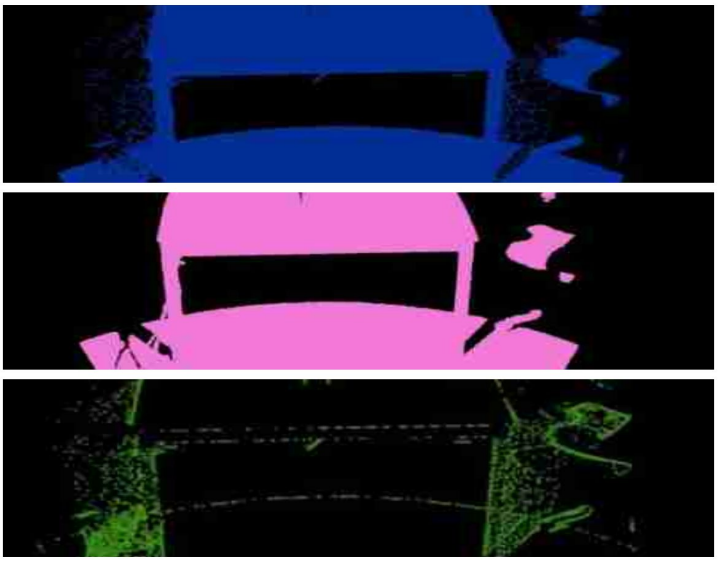


Figure 8: Example of application of the Outlier Removal process to a point cloud [21]

Below are presented the possible algorithms selected to perform the outlier removal:

1. **Radius Outlier Removal**: this technique filters the points in the cloud based on the number of neighbours that each point has. It is composed of two functions, the first one is the Radius Outlier Removal Background. It iterates through the input cloud and obtains the number of neighbours for each point considering a certain radius. For a point to be considered an outlier, it must have less than a certain number of points in the radius established. Secondly, with the Conditional Removal Background function those points considered as Outlier will be discarded from the cloud [22].

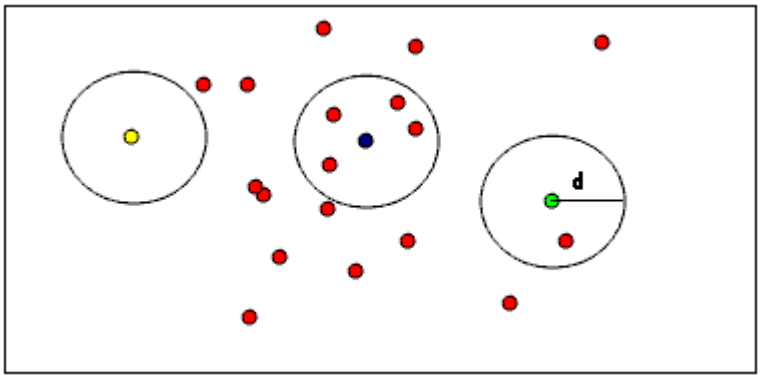


Figure 9: Illustration of Radius Outlier Removal [23]

The figure above shows the application of the method to an example 2D point cloud. The points removed will depend on the minimum number of neighbours set. As an example, in the case of setting a minimum of one neighbour, the point coloured yellow would be removed.

2. **Statistical Outlier Removal**: this procedure consists of performing a statistical analysis in the neighbourhood of each one of the points. More precisely, it is based on obtaining the statistical distribution of the distances between a point and its neighbours. This is done by obtaining the mean of the distances between each of the points and their vicinity points. Once the means have been calculated, a Gaussian distribution is created with one mean and one standard deviation [24]. Then, all those points whose means are outside an interval defined by the mean and standard deviation of the Gaussian distribution will be considered as Outliers [25].

$$Dist_{threshold} = mean + Stand_{dev\_mult} \times Stand_{dev} \quad (1)$$

Being the variables:

- $mean$: mean of the Gaussian distribution.
- $Stand_{dev\_mult}$: multiplicative parameter of the standard deviation (determined by experimentation).

- $Stand_{dev}$: standard deviation of the Gaussian distribution.

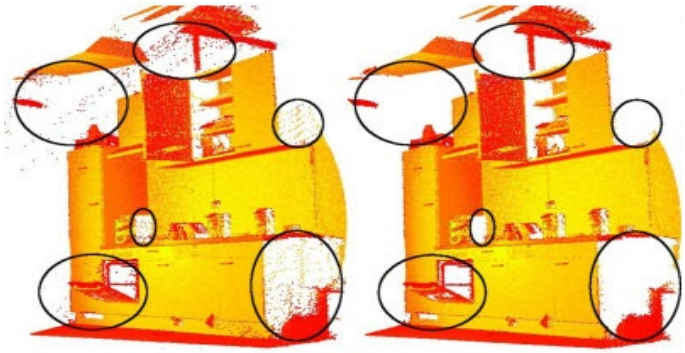


Figure 10: Example of application of the Statistical Outlier Removal [24].

2.5.1.2 Segmentation

The aim of segmentation is to divide the point cloud into the different planes that compose it. Using this method, the areas that are not relevant for the recognition of the elements and which, nevertheless, represent a high percentage of the total number of points in the cloud could be removed. Specifically, as in most point clouds, the planes to be eliminated are the ground plane and the plane considered as "air". As indicated in the section on outlier removal, the purpose of the previous method is equivalent, therefore, the segmentation phase is a second filter to obtain a noise-free point cloud [26].

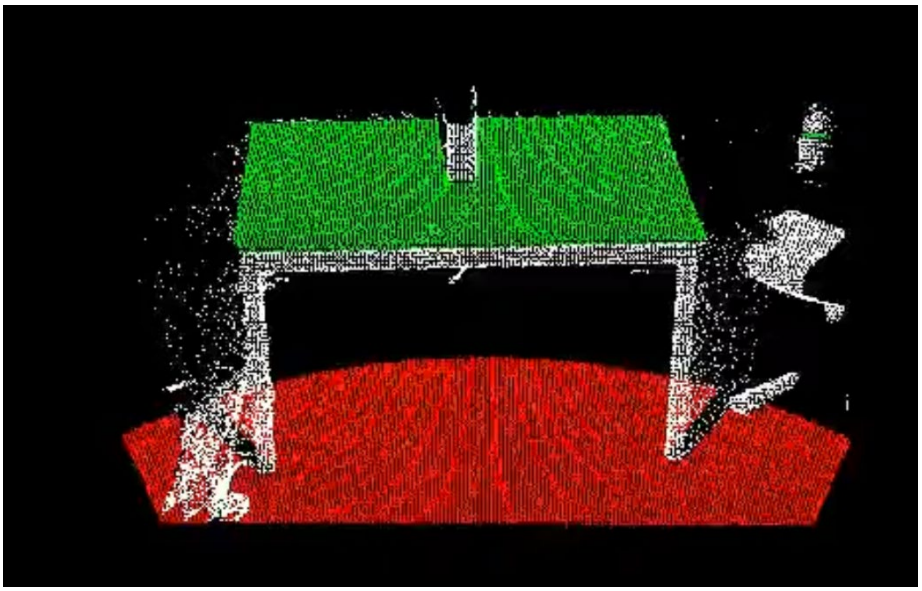


Figure 11: Output resultant after applying RANSAC to the point cloud [27].

The RANdom SAmple Consensus (RANSAC) method is used to perform the segmentation process. This algorithm was developed in 1981 as a model fitting method; however, nowadays it is used to detect geometric shapes such as lines to divide point clouds [28].

Inputs

- Set of data points U = {xi}.
- Function to obtain the model parameters from the data set f (S): S → θ.
- Cost function to evaluate the different models obtained.

Algorithm

Repeat until P{better solution exists} < η (a function of $C*$ and no. of steps k)

k := k + 1

I. Hypothesis

(1) select randomly set $Sk \subset U$, $|Sk| = s$

(2) compute parameters θk = f (Sk)

II. Verification

(3) compute cost $Ck = \sum x \in U \rho(\theta k, x)$

(4) if $C < Ck$ then $C* := Ck$, $\theta* := \theta k$*

end

As an example, the detection of a line in the 2D space is introduced.

1. Random selection of two points, the minimum number of points that form a line.
2. The hypothesis of the model, in this case, corresponds to the line passing through the two selected points.
3. The cost function on this occasion to evaluate the different models is the distance from the points to the model.
4. Logically, the optimal model will be the one that has the most points close to it. With the detection, this line divides the space into two subspaces.

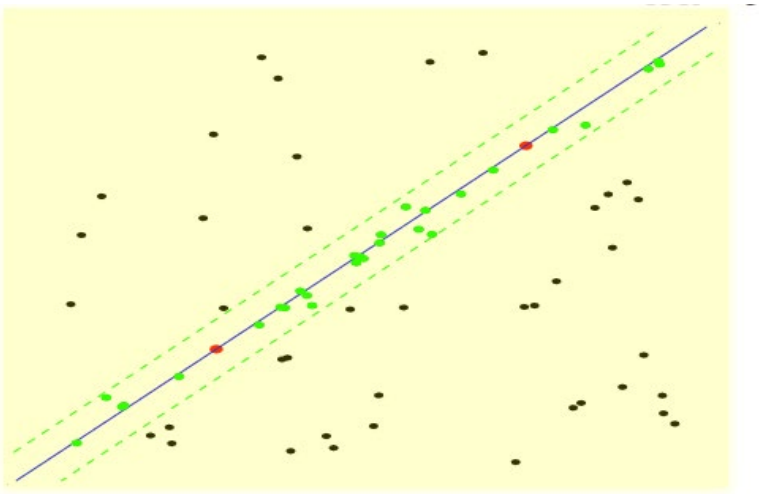


Figure 12: Line detection in 2D point cloud using RANSAC algorithm [28].

Finally, to estimate the number of iterations needed to find the desired shape with probability p in a set of points with s points with a percentage of outliers e [26].

$$T = \frac{\log(1-p)}{\log(1-(1-e)^s)} \quad (2)$$

2.5.1.3 Downsampling

The purpose of downsampling is to reduce the total number of points to be processed. This should be done affecting as little as possible the distribution of points present in each of the clouds [58]. Because, if the distribution and density of the points in the cloud is significantly changed, the result obtained may not correspond to the reality originally captured. It is also noteworthy to mention the impact of this method on clouds with a large total density of points.

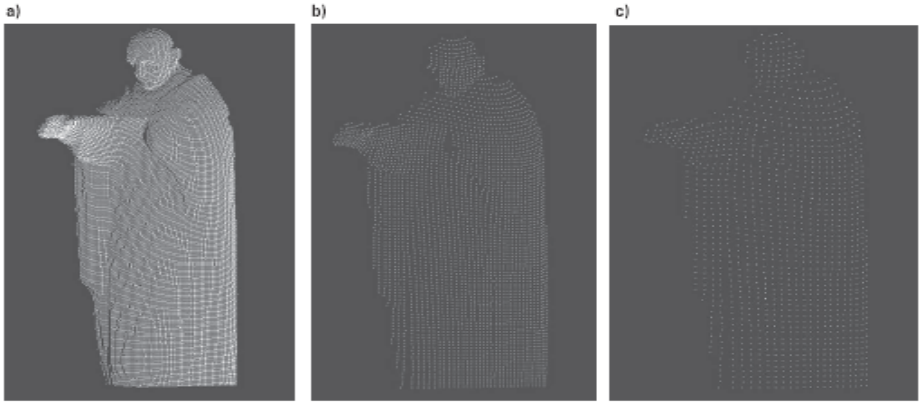


Figure 13: Point cloud visualization: a) input point cloud, b) downsampled point cloud – 0.01m and c) down sampled point cloud – 0.04m [29]

As observed in the previous figure, the reduction of points to be processed is substantial, but always trying to maintain the overall shape of the object.

Voxel Grid filter downsampling: the Voxel Grid method consists on subdividing the space into small cubes (3D) or squares (2D). Then, the centroid of the points in each of the subdivisions is obtained, leaving only this point as a representation of all the points in the partition [30].

The centroid of each of the voxels in the 3D case is obtained from a set S of n points.

$$S = \{(x_1, y_1, z_1), (x_2, y_2, z_2), \dots \ (x_n, y_n, z_n)\} \quad (3)$$

It is calculated by dividing the sum of the coordinates of the points of the set by the number of those points.

$$(\bar{x}, \bar{y}) = (\frac{1}{n}\sum_{i=0}^{n} x_i, \frac{1}{n}\sum_{i=0}^{n} y_i, \frac{1}{n}\sum_{i=0}^{n} z_i) \quad (4)$$

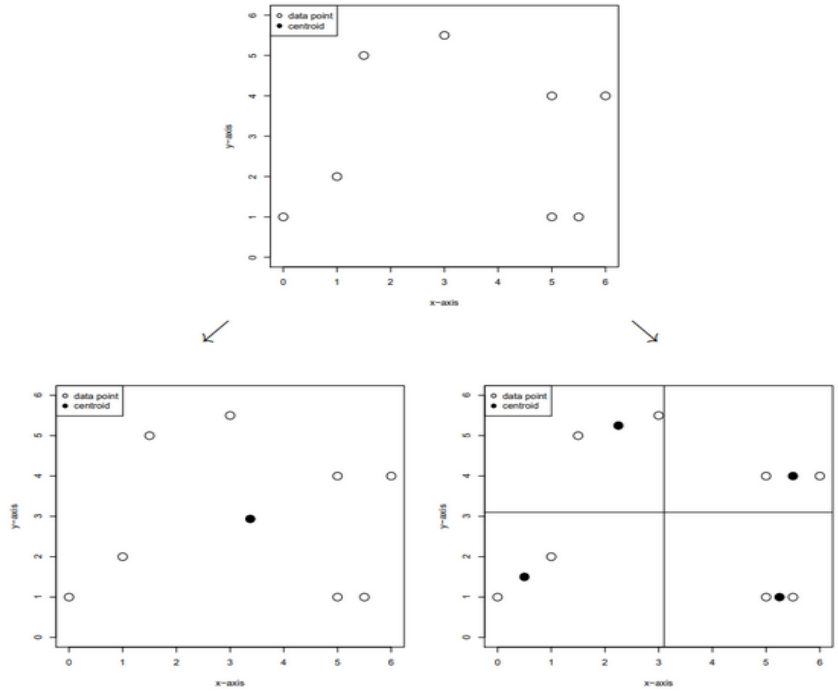


Figure 14: Example of a Voxel Grid downsampling [30]

It can be deduced from the figure that, logically, the larger the size of the Voxel Grid, the greater the simplification that will be carried out. This is because a greater number of points are replaced by the single centroid. The relationship between the size of the Voxel Grid and the ratio of points, as shown in Figure 15, is linear [30].

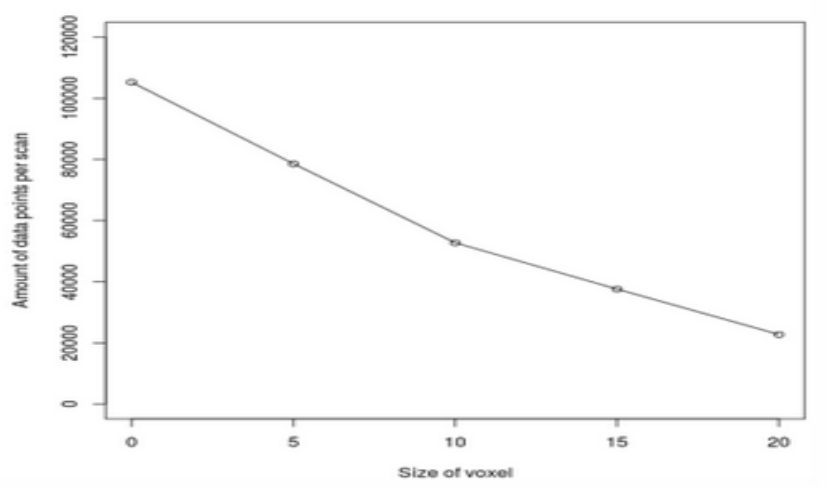


Figure 15: Relation between the size of the voxel and the number of points [30].

Lastly, it should also be noted that since this method is based on obtaining the centroid of the different grids, the result obtained maintains the point density of the original cloud. Consequently, the distortion of the cloud and therefore the loss of information is avoided to a greater extent.

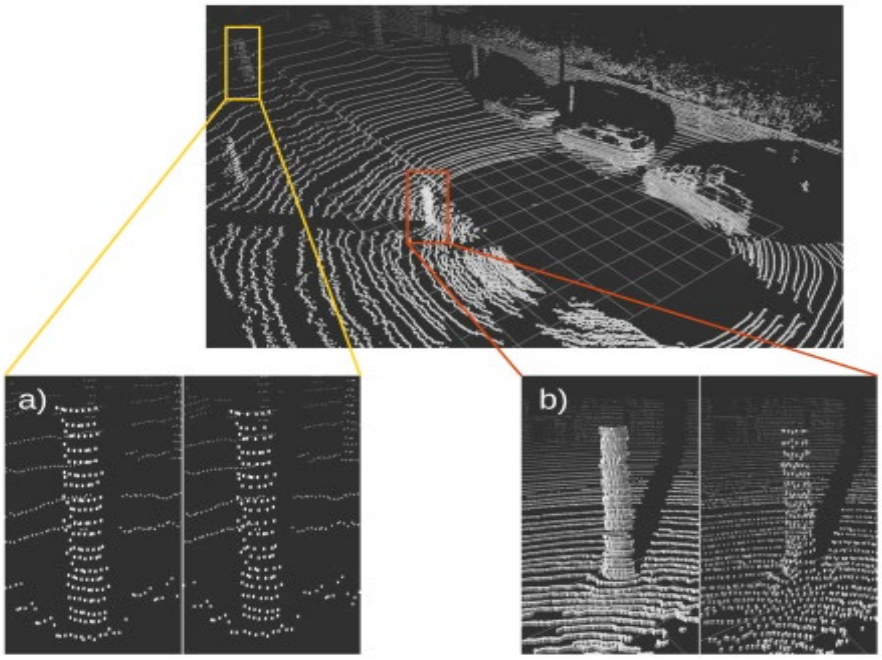


Figure 16: Example of application of a Voxel Grid to a Point Cloud [30].

Can be noticed that element a), which initially has a lower density of points, maintains most of them after obtaining the centroids. However, element b), which initially has more points when the Voxel Grid filter is applied, logically, the total number of points is reduced to a greater extent.

2.5.1.4 Cluster Extraction

First of all, clustering consists of classifying a disorganized point cloud based on some criteria. Such as belonging to the same object or similarities between different point clouds. For this purpose, the concepts of distance and similarity are mainly used.

- Distance between two points
- Distance between a point and a cluster.
- Distance between two clusters.

In this project, the Euclidean distance typology is used, as in most projects. This distance is defined as the straight line joining two points in Euclidean space [32].

Once the two properties have been obtained for each point, the boundaries of the clusters must be defined. Two points that are close together must belong to the same cluster, however, if they are considered far apart, they must be separated into different clusters. As can be inferred, this decision will be given by the parameterization to be performed [31].

As a result of the fact that in most cases the objects to be recognized are located on planar elements (floor, table...) it is interesting to remove the parts that correspond thin unsignificant planar parts. The interest in this lies mainly in the elimination of points to accelerate the processing. Moreover, by deleting these points, it is also possible to have more precise clusters in their delimitation.

Once the clustering process has been explained, the algorithm used to implement it will be explained [32]:

1. create a Kd-tree representation for the input point cloud dataset P.
2. Set up an empty list of clusters C, and a queue of the points that need to be checked Q.
3. then for every point $\boldsymbol{p}_i \in P$, perform the following steps:
   - add $\boldsymbol{p}_i$ to the current queue Q.
   - for every point $\boldsymbol{p}_i \in Q$ do:
     - search for the set P_i^k of point neighbours of $\boldsymbol{p}_i$ in a sphere with radius $r < d_{th}$.
     - for every neighbour $\boldsymbol{p}_i^k \in P_i^k$, check if the point has already been processed, and if not add it to Q.
   - when the list of all points in Q has been processed, add Q to the list of clusters C, and reset Q to an empty list.
4. the algorithm terminates when all points $\boldsymbol{p}_i \in P$ have been processed and are now part of the list of point clusters C

First, a Kd-Tree is created to obtain the distances between a point and its neighbours in a faster way than calculating the distance between a point and the rest in the classical way. The Kd-Tree is created by incorporating hyperplanes that divide the space into different cubes (as the point cloud considered is in 3D).

1. One of the three dimensions is taken randomly.
2. The average of the selected points is calculated.
3. A hyperplane chosen_dimension = mean in the selected dimension is created.
4. The same process is repeated for each of the subdivisions until all dimensions have been analysed in each of the partitions.

The illustration below shows the result of the process for a 2D cloud and for a 3D cloud.

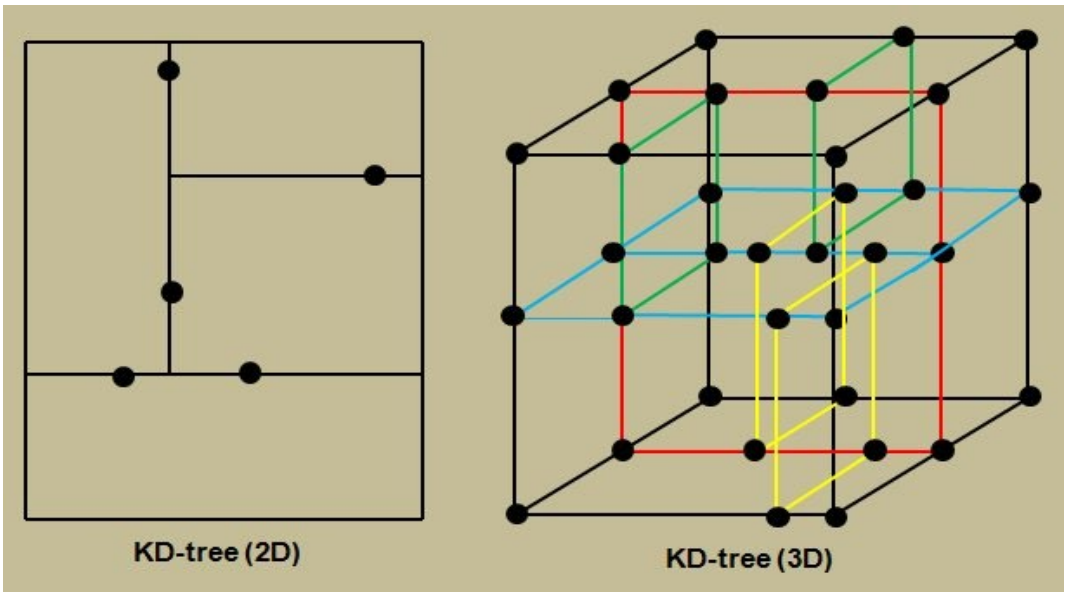


Figure 17: Representation of a 2D-KdTree and a 3D-KdTree [25].

The following steps are explained in the algorithm outlined above. After applying all these stages, the subdivision of the different elements or objects that make up the point cloud is obtained.

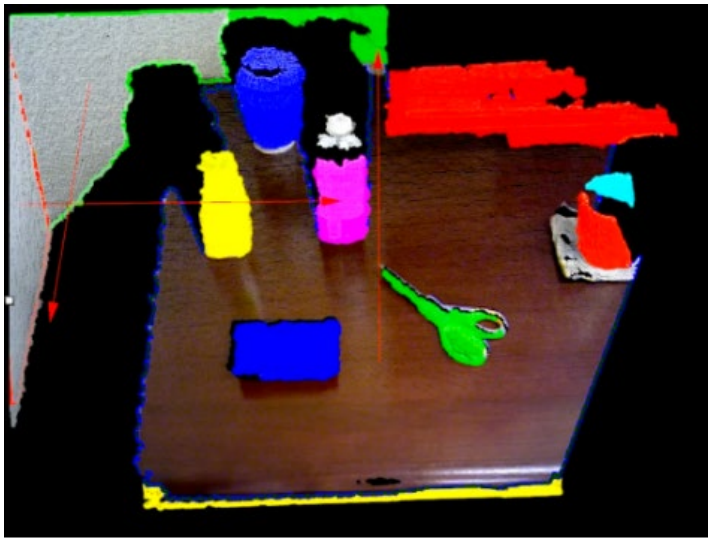


Figure 18: Example of the result of clustering a point cloud [33].

On the one hand, this procedure will be used to obtain the models of the elements to be recognized from point clouds. To obtain the different models, the cluster creation procedure is followed as explained in this section.

On the other hand, clustering will also be used in the recognition phase to divide the elements of the point cloud to find the desired model in the cloud in a simpler way.

2.5.2 Point Cloud Registration

2.5.2.1 Keypoint extraction

A keypoint is also called a point of interest. These are the points in a point cloud that are descriptive, i.e., that define the main characteristics or outstanding characteristics of the objects captured in the point cloud. To be a keypoint, the point must also be stable, in other words, they must be invariant to image rotation, shrinkage, translation, distortion, and so on.

On the other hand, the criteria for extracting keypoints differ according to the selected technique. This makes it important to experiment for each of the situations, since the efficiency and effectiveness of the method will depend on the typology of the cloud. This will lead to a highly variable execution time and accuracy when estimating keypoints.

Typically, the number of keypoints is considerably smaller than the total number of points in the pre-processed cloud. In combination with the local features in each of the key points, they can be used as a really compact, but still descriptive, descriptor for the representation of the original data [8, 12].

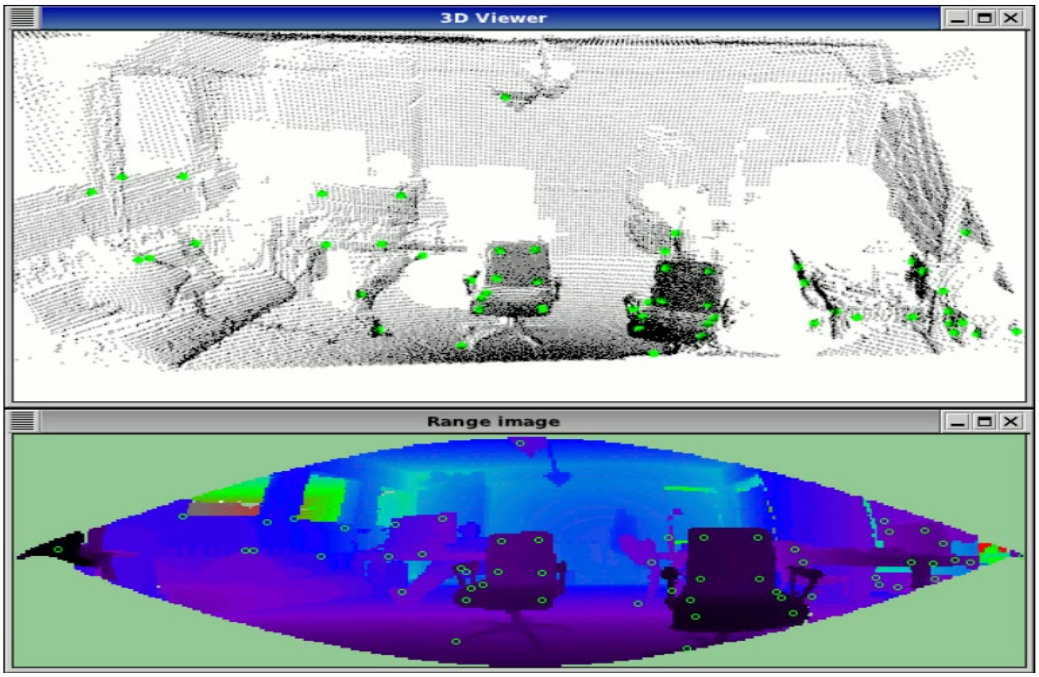


Figure 19: Example of keypoints estimation [34].

1. Uniform Sampling

Consists of the creation of a 3D square mesh on the given point cloud. Equivalently to the Voxel Grid downsampling, all the points of the voxel are replaced with the one closest to the midpoint.

Similarly to downsampling, the uniform sampling method maintains the proportions in cloud density while preserving the original cloud shapes. Therefore, the points resulting from this process will be a good representation of the points describing the object [35].

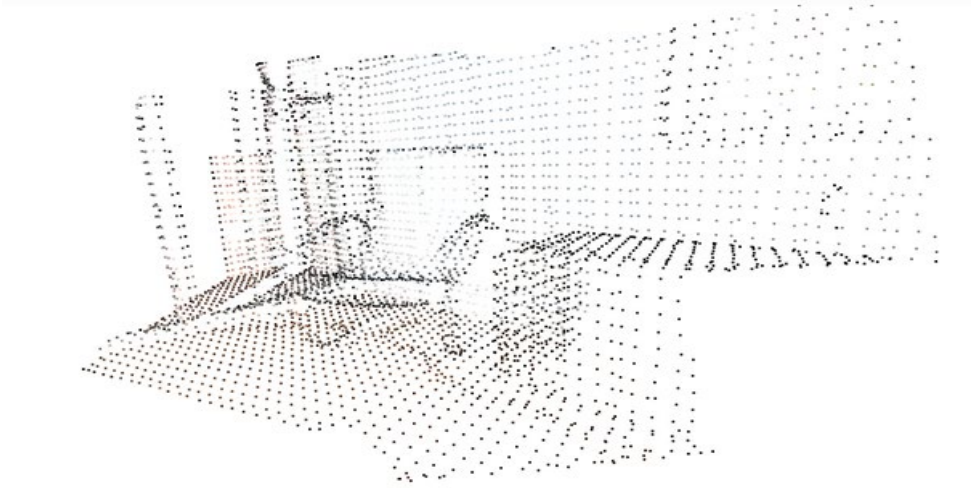


Figure 20: Uniformly sampled point cloud [34].

In conclusion, the method is a very rapid method of extracting keypoints due to the simplicity of the process. Moreover, as mentioned in the previous paragraphs, the keypoints obtained reliably describe the surface of the objects. This makes it an interesting method for real-time applications. However, it is vulnerable to changes, such as luminosity or orientation change in the input point cloud. Therefore, the flexibility of environments where this method can be applied with the same parameterization is restricted [36].

2. SIFT (Scale Invariant Feature Transform)

Created by Lowe in 2004 for 2D images, was adapted by the PCL community (Rusu and Cousins) to work specifically with 3D point clouds, the 3D-SIFT algorithm extracts keypoints based on the Gaussian scale-space. Going further, the Gaussian scale-space is created by downsampling with Voxel Grid filters of different sizes and a blur filter that serves to perform a radial search of the near neighbours. From these, the new intensity is calculated for each of the points as a weighted sum of the nearest points [37].

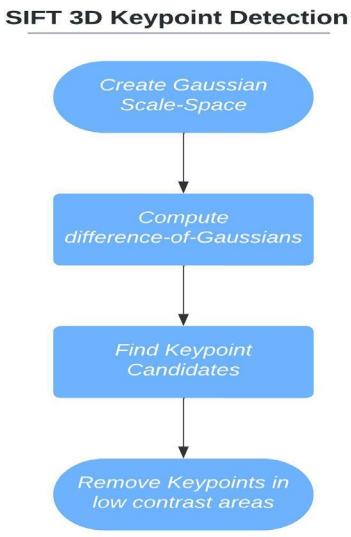


Figure 21: SIFT Keypoint extraction procedure [37].

1. Gaussian Scale-Space: the original point cloud is convolved with Gaussian filters at different scales. The convolutions consist of scaling and blurring of the original point cloud. Convolutions are divided into octaves and within each octave into scales. Each octave is defined by the scale that is applied to the input point cloud. On the other hand, the number of scales defines the number of times the Gaussian filter will be applied to achieve different levels of blurring [38].

$$L\,(x, y, k\sigma) = G(x, y, k\sigma)\ \times I(x, y) \qquad (5)$$

Where:

- $I(x, y)$ represents the initial point cloud.

- $G(x, y, k\sigma)$ is the Gaussian filter to be applied for the scale of kσ.

- $L\,(x, y, k\sigma)$ is the convolution of the original point cloud applying the Gaussian filter corresponding to the octave and cycle.

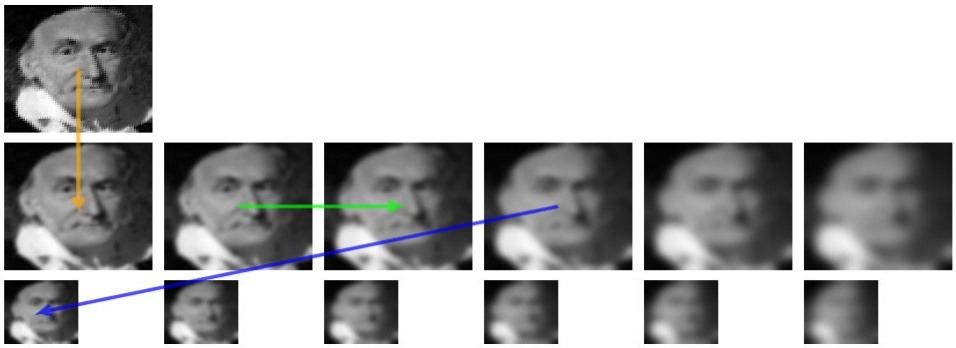


Figure 22: Example of Gaussian Scale-Space of 2 octaves with 6 cycles each [39]

2. Difference of Gaussians (DoG): Consists in the subtraction of Gaussians belonging to the attached cycles within an octave [38].

$$D(x, y, \sigma) = L(x, y, k_i\sigma) - L\big(x, y, k_j\sigma\big) \qquad (6)$$

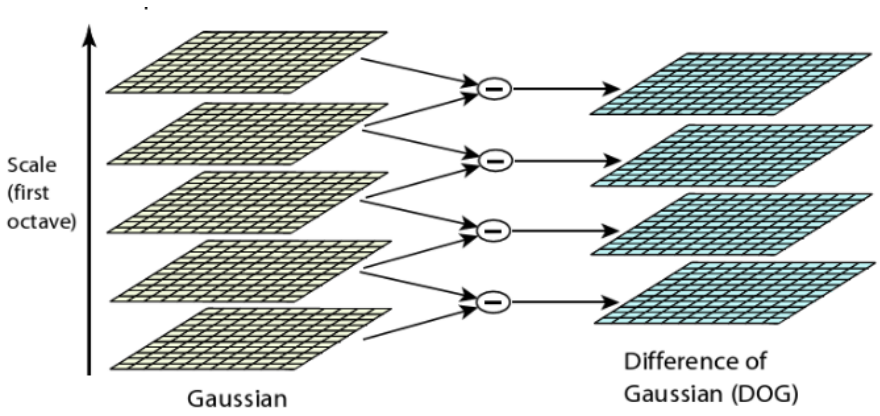


Figure 23 : Example of application of DoG [38].

3. Keypoint candidates

First, the extreme scale-space is calculated from the DoG across the scales. The keypoints are defined as the local minima/maximums of this process, i.e., whether the point value in the Gaussian is the maximum or minimum among all compared points is selected as the keypoint.

There can be a problem of precision caused by imprecise localization of the extreme points due to the sampling procedure.

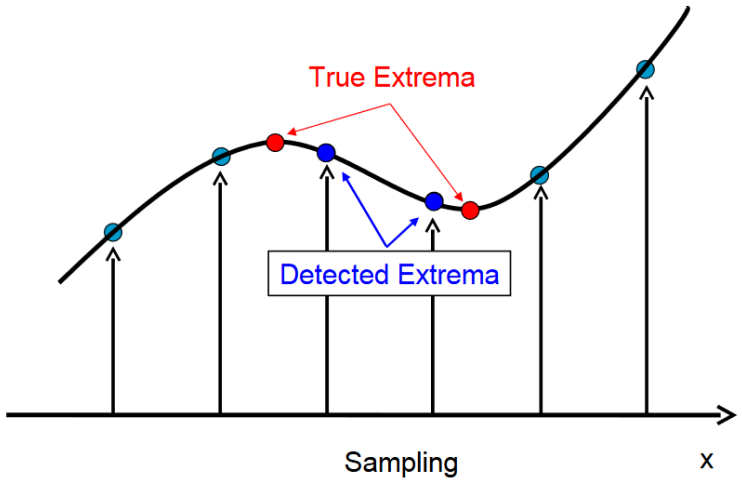


Figure 24: Precision problem in the estimation of local extrema [38]

Therefore, using the Taylor series, it is interpolated with the contiguous data to obtain a more accurate position of these endpoints.

$$D(x) = D + \frac{\partial D^T}{\partial x}x + \frac{1}{2}x^T\frac{\partial^2 D}{\partial x^2}x \qquad (7)$$

Where:

- $x = (x, y, \sigma)^T$: represents the offset parameters for the studied point.

Ultimately, the real position of the extreme points will be determined by obtaining the first derivative of the previous expression with respect to x and equalling it to 0, $\hat{x}$. In the case that the result of this operation is greater than 0.5, the real extreme is closer to another point, being this point the new candidate for keypoint. Otherwise, the calculated offset is added to the candidate point to estimate the location of the extreme point accurately.

4. Keypoints removal: Points with poor contrast should be removed, as they do not provide information of interest. Dots with a D($\hat{x}$) of less than 0.03 are considered to have bad contrast.

   In addition, to increase stability, points with a sharp change in the edges in one direction are also eliminated. For those points with poorly defined peaks in the DoG function, the main curvature at the edge will be much larger than the main curvature along it. These principal curvatures correspond to the eigenvalues of the second order Hessian matrix:

$$H = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{yx} & D_{yy} \end{bmatrix} \quad (8)$$

   Once the Hessian matrix is obtained, the candidate keypoints that do not meet the ratio, calculated from a threshold called r, are eliminated [38].

$$\frac{Tr(H)^2}{Det(H)} < \frac{(r+1)^2}{r} \quad (9)$$

Finally, in different experiments, several objects scanned by different technologies and from different angles have been analyzed. After the experiments, the data indicate that for a point cloud of about 112000 points on average. Regarding the number of keypoints, an average of 2838 keypoints was obtained. This quantity of keypoints is considerable, being an accurate representation of the original cloud. Due to this high precision of the keypoints, obtained from different scales and blurred, more successful results are achieved in the registration of point clouds and greater invariance of the position of the keypoints due to changes in detection angles and luminosity. However, it took an average of 1.891 seconds to obtain the keypoints. This makes the 3D-SIFT algorithm not entirely suitable for real-time applications. However, if during pre-processing/downsampling the point cloud is reduced, the processing time could be reduced because the number of primitive points to be analyzed by the method are reduced [40].

3. ISS (Intrinsic Shape Signatures)

This method is an adaptation made by Federico Tombari and Samuele Salti especially for PCL. Their adaptation is based on the original work of Yu Zhong ("Intrinsic shape signatures: A shape descriptor for 3D object recognition"). This being said, the ISS is based on the EigenValue Decomposition (EVD) of the scatter matrix Σ(p) of the points belonging to the environment of p, being p the point under study [41].

$$\Sigma(\mathrm{p}) = \frac{1}{N}\sum_{q\in\aleph(p)}(q-\mu_p)(q-\mu_p)^T \quad (10)$$

$$\text{with: } \mu_p = \frac{1}{N}\sum_{q\in\aleph(p)} q$$

Once the eigenvalues are obtained, they are ordered from highest to lowest, λ1, λ2, λ3. Then, during the pruning stage, the points whose ratio between two successive eigenvalues is below a limit are retained. This is done to avoid extracting keypoints that have similar characteristics in the three main directions.

$$\frac{\lambda_2(p)}{\lambda_1(p)} < \tau_{12} \quad \wedge \quad \frac{\lambda_3(p)}{\lambda_2(p)} < \tau_{23} \quad (11)$$

$$\rho(\mathrm{p}) \doteq \lambda_3(\mathrm{p}) \quad (12)$$

This procedure would prevent in later stages the establishment of a reference frame that is canonical, a point being non-descriptive of its environment. For points that have sufficient variability in their environment, their saliency must be calculated. Specifically, the saliency of a point is exactly the value of its lowest eigenvalue (λ3) [41].

After the assignment of the saliency, the keypoints must be computed. It follows that the keypoint of a specific environment will be the point with the highest saliency value.

Regarding the characteristics of ISS, both relative and absolute repeatability are remarkably good, as points with considerable changes between their main directions are selected. Because of this, descriptive keypoints are also obtained, which will increase the repeatability of the method further, thereby increasing the accuracy of the registration. Moreover, the ISS method is efficient, since, as seen in the short theoretical introduction, only points that have a considerable variation between their three main directions are considered. For this reason, all those points not suitable to be keypoints due to lack of describability will not be considered in the subsequent steps, increasing the efficiency of ISS.

On the other hand, as observed in Tombari's paper, ISS is considerably affected by the degree of noise in the point cloud. Therefore, filtering would be an important pre-processing step for the application of this method. Moreover, as for the processing time of the keypoints by this method, it has an average time of about 2 seconds, which would make it not applicable for real time [42].

2.5.2.2 Feature Descriptors Estimation

The keypoint concept has been introduced as a representative point of the point cloud. However, despite their relevance in describing the image, keypoints are only specific points within the point cloud. This creates a problem for registration because a small difference in sensor position or other small deviations can lead to a difference in the position of the keypoints in the point cloud and thus to an error in the subsequent registration phase.

For the reason introduced above, feature descriptors have been developed. These structures summarise the local structures that surround the previously extracted keypoints. This makes it easier and more accurate to recognize similarities between keypoints. In this way, the relationships between two point clouds with different capture conditions can be established.

Normally, the features descriptors consist of vectors that store the various parameters, which, depending on the methodology, define the keypoint environment. The dimensions of the vector vary according to the method used. However, the vector dimension will always be maintained.

SHOT Descriptor

One of the most serious problems of feature descriptors is the definition of a single, unambiguous and stable local coordinate system for each of the keypoints (Tombari et al., 2010). Therefore, the Signature of Histograms of OrienTations (SHOT) method is proposed [43].

First, a local coordinate system is created to solve the problem proposed in the previous paragraph. The first three steps consist of the computation of this system at a point p.

1. Covariance Matrix computation: For the computation of the matrix, the n neighbors of point p_i are considered.

$$C = \frac{1}{n}\sum_{i=1}^{n}(r - \|p_i - p\|) \cdot (p_i - p) \cdot (p_i - p)^T \quad (13)$$

   where r is the search radius of the neighboring points.

2. Extraction of the eigenvectors: In this step, the eigenvectors are obtained from the eigenvalues.

3. Reorient Eigenvectors: From the covariance matrix, three orthogonal eigenvectors have been obtained, defining the coordinate system at the study point. However, to obtain a good result, these vectors must be reoriented. First, the eigenvectors are arranged in descending order, i.e. v1 will have the largest value and v3 the smallest, thus representing the X, Y and Z axes. The direction of the X-axis is determined by the orientation of the vectors from p to the neighboring points:

$$X = \begin{cases} v_1, & if \ |S_x^+| \geq |S_x^-| \\ v_1, & otherwise \end{cases}$$

$$S_x^+ = \{p_i | (p_i - p) \cdot v_i \geq 0\} \quad (14)$$

$$S_x^- = \{p_i | (p_i - p) \cdot v_i < 0\} \quad (15)$$

The direction of the Z-axis is obtained in an identical way to the X-axis. Finally, since the three eigenvectors are orthogonal, the Y-axis is obtained by the cross product of the X and Z axes.

4. Creation of the spherical grid: once the coordinate system for each of the p points has been obtained, the isotropic spherical grid is created. The grid, by using the local coordinate system, divide the space of p.

5. Creation of the histograms for the grid cells: For each of the points within a cell, the angle between the normal of the point and the normal of p (study point) is calculated $\varepsilon_i = p_i \cdot p$.

   There are two possibilities for the storage of the local distribution of angles, signatures and histograms.

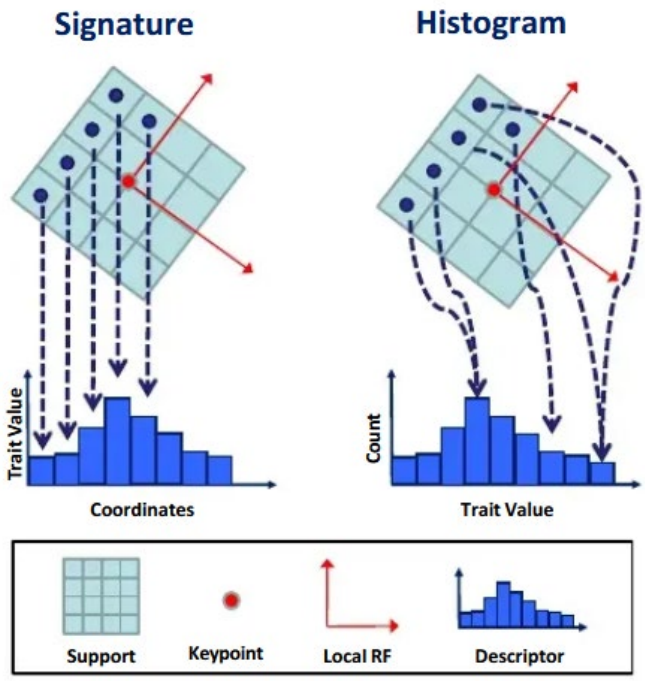


Figure 25: Comparison between signature and histogram [44].

As can be seen in the image above, the signatures are highly descriptive as the information is stored considering the spatial distribution. However, small errors in

defining the Local Reference Frame (LRF) can modify the descriptor considerably. However, histograms do not have this problem. Therefore, histograms are chosen as a storage method [44].

6. Group cell histograms: The histograms of the individual cells are added together to obtain the total histogram of the point. This histogram will contain a total of k*b values, where k is the number of cells to consider, and b is the number of bins per cell histogram. These values are normalized to sum to 1 to handle different point densities in different cloud types.

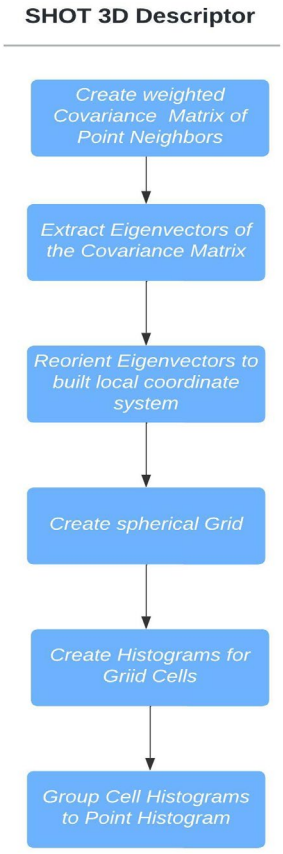


Figure 26: Process for SHOT features obtention [43].

2.5.2.3 Correspondence estimation

This section describes the registration process. In general terms, the registration of two point clouds is based on estimating the homogeneous transformation to be applied to one of the two clouds. By applying this transformation matrix, the clouds are placed in the same reference frame. This implies that a particular point in the two clouds will occupy the same place in 3D space, in the case where the transform is ideal.

In order to calculate the homogeneous transform, it is assumed that the point clouds are rigid solids. This means that the deformations that may appear between the internal points of the same cloud are not considered, with the distances between two points of the same solid being constant. With this simplification, the homogeneous transform consists of a rotation matrix and a translation vector.

Once the two sets of points have the same reference, methods for object recognition based on comparison with databases can be applied. This procedure is depicted in the figure below.

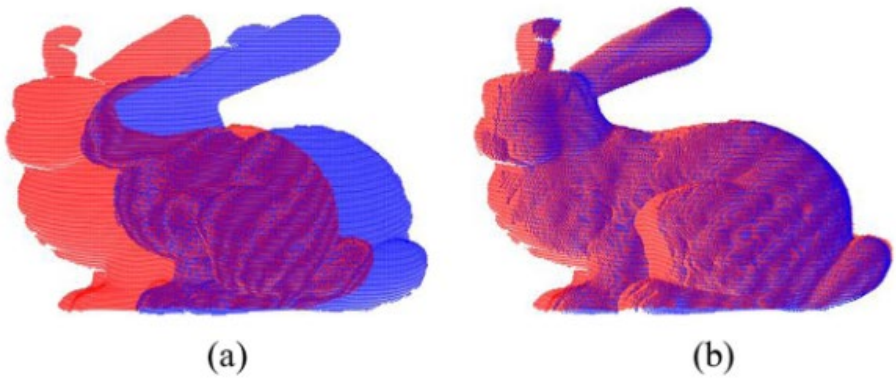


Figure 27: Registration of two point clouds, being a) initial position of the sets b) position after applying the homogeneous transform [45]

As can be inferred from the previous paragraph, this process has a great importance as its precision will determine the success of the subsequent processes. This makes accuracy one of the main requirements to be fulfilled.

In the following subsections, the operations that must be carried out to register the two clouds will be described.

The registration process requires, as is logical, a series of correspondences to calculate the homogeneous transform. In other words, the interconnections between two different points in the two clouds must be created. However, in most real processes the correct correspondences are not available and must be obtained by a heuristic process.

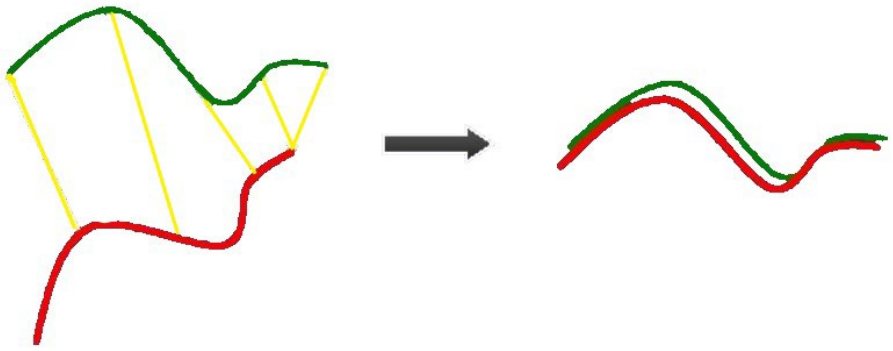


Figure 28: Example of correspondence estimation

The process with which the search for these correspondences is carried out is the Closest Point Algorithm. This algorithm, as its name indicates, consists of detecting the closest point in a cloud of points with respect to a point in the other cloud that is to be registered. It is important to mention that, as it is a heuristic method, in the case of obtaining correspondences that are not sufficiently precise to register the clouds correctly, an iterative process must be carried out to improve their precision (known as ICP), however, due to the characteristics of the Hough3D algorithm will not need. Having made this important observation, the stages that make up this algorithm are set out below [46]:

1. Select the points of a point cloud from which the correspondences are to be obtained. In the case of this thesis, the points from which the correspondences are to be obtained are the keypoints and the features associated to them, since they are the points that describe the most important characteristics of the objects to be recorded.

2. Obtaining the closest points in the other cloud of each of the points selected in the first cloud of points. This will be done using the Kd-TreeFLANN algorithm, which is an optimised version of Kd-Tree.

   (From this point on, the method must be applied if iterations are necessary to obtain the correspondences).

3. Obtaining the transform and applying it to "bring closer" the two point clouds.

4. Iterate.

The optimization of the KdTree has been carried out by Marius Muja and David Lowe and collected in the FLANN library (Fast Library for Approximate Nearest Neighbours). The importance of this library is the reduction in the time used in the NNS, since this process is one of the most important in applications such as object recognition or image recognition [47].

This algorithm is based, as can be inferred, on the Kd-Tree algorithm that has already been introduced in previous sections. However, the Kd-Tree algorithm does not provide a significant improvement in the search in high-dimensional spaces as it has a search time similar to that of the standard brute-force search algorithm. Because of this, the NNS will be performed in an approximate manner as it is sufficient for most practical applications. The benefit of performing NNS in an approximate manner over the exact manner is the reduction by orders of magnitude of the time required to perform the search [48]. Finally, it should be mentioned that the library itself will automatically choose the best algorithm and parameters for the cloud input [49].

2.5.2.4 Correspondence rejection

In the previous subsection it has been explained which method is followed to obtain the correspondences between the points of the two clouds to be studied. Now, one or more procedures must be established to determine whether the correspondences estimated

previously are considered valid or not and thus eliminate those that would affect the quality of the result.

There is a multitude of methods to consider correspondence erroneous. These are based on various characteristics, some of which are listed below:

- Distance between correspondences.
- Based on the characteristics of the features.
- Statistical parameters such as the average of the distances between matches.
- Elimination of duplicate indices in the creation of correspondences.

On the other hand, the use of correspondence rejectors is subject to the following two principles:

1. Use of computational capacity: One should logically try to reduce as much as possible the use of resources when processing the point clouds. Therefore, the number of rejection algorithms used should be reduced as much as possible. In addition, priority should be given to those algorithms that are simpler and faster to apply.
2. Accuracy of the result: Always maintain an accurate registration process in as few stages or iterations as possible. The goodness of the result will be given by the ability of the homogeneous transform to perform the movement between the two coordinate bases to be registered.

Once the two principles governing the selection of the algorithms have been set out, it is emphasised that this selection will be made experimentally by simulating a series of scenarios. Based on the results obtained, it will be decided which methods are necessary to filter the correspondences.

2.5.2.5 Transformation estimation

Once the correspondences have been estimated and those that did not provide information or did so in an erroneous manner have been properly eliminated, it is time to determine the homogeneous transform.

First of all, the concept of homogeneous transform must be defined. It is based on a 4x4 square matrix representing the rotational and translational movements that are necessary to register the two point clouds. The equation below shows the components of the homogeneous transform matrix.

$$T = \begin{pmatrix} R_{3x3} & Trans_{3x1} \\ 0_{1x3} & 1 \end{pmatrix} \quad (16)$$

First, the rotation matrix will be defined. This matrix, for the three dimensions, is a 3x3 square matrix representing the rotation that must be performed on each of the coordinate axes for two coordinate systems to have the same orientation in space. Since this is operating in three dimensions, it is obvious that a maximum of three rotations (in the x, y and z axes) will be needed to have the same orientation in two coordinate systems. The rotation matrices to be applied to each of the axes are shown below.

- Rotation on the x-axis

$$R_x = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) \\ 0 & \sin(\alpha) & \cos(\alpha) \end{pmatrix} \quad (17)$$

- Rotation on the y-axis

$$R_y = \begin{pmatrix} \cos(\beta) & 0 & \sin(\beta) \\ 0 & 1 & 0 \\ -\sin(\beta) & 0 & \cos(\beta) \end{pmatrix} \quad (18)$$

- Rotation on the z-axis

$$R_z = \begin{pmatrix} \cos(\gamma) & -\sin(\gamma) & 0 \\ \sin(\gamma) & \cos(\gamma) & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad (19)$$

To combine a series of rotations, simply must perform the product of the rotations. However, it is very important to emphasise the NON-commutativity of the product. That is, a rotation on the y-axis and a rotation on the z-axis will not give the same result as a rotation on the z-axis and a rotation on the z-axis. The commutative property is only fulfilled in the case of multiple rotations on the same axis, which is not the case in this project. That said, the order in which the rotations are applied depends exclusively on the method chosen, in our case, the Levenberg-Marquardt method.

The LM method provides a solid platform for the calculation of homogeneous transforms. With this method, the computation time is reduced with respect to other methods, without losing the robustness of the algorithm and the reliability of its results.

Once the rotation matrix that orients the two coordinate axes in the same direction has been calculated, it is time to calculate the translation that must be carried out to place the two systems at the same point in space. The translation is defined by a vector of dimension 3x1.

This vector indicates the movements that must be carried out on each of the three axes to position the two origins at the same point. As with the successions of rotations, the commutative property between rotations and translations is also not fulfilled.

Finally, as mentioned above, the homogeneous matrix is obtained from the composition of the rotation and translation matrices.

2.5.3 Recognition

Object recognition in 3D scenes has its main problem in scenes with clutter and occlusions. In these scenes, methods must be applied to discard those features correspondences that are caused by nuisance factors, as well as to determine a subset of correspondences that can detect the objects in the scene and determine their pose. State-of-the-art approaches are mainly two. First, starting from initial correspondences, registration is performed iteratively by aggregating those correspondences that satisfy the correspondence rejection conditions, i.e. applying the ICP algorithm.

On the other hand, the other approach is based on clustering pose hypotheses in a 6-dimensional pose space, obtaining from each correspondence a pose hypothesis (i.e. a rotation and translation) based on the Local Reference Frame (LRF) associated with each descriptor correspondence. Once reliable correspondences have been selected, the number of models in the scene can be recognized as well as the final homogeneous transform.

The Hough Transform (HT) is mostly used to detect lines in 2D clouds as well as circles and ellipses. Its main idea is to vote the cloud features (such as edges or corners) into the shape to be detected. These votes are stored in an accumulator that has the same dimension as the number of parameters that define the shape of the object to be recognized. Therefore, this classical approach is not suitable for the recognition of complex surfaces, such as the UIC hook proposed in this Thesis, since the large size of the accumulator would make the method too slow and require excessive memory.

The same problem that occurs in the 2D HT, is transferred to the Generalized Hough Transform (GHT) for the 3D case, with the added difficulty in recognizing elements with arbitrary shapes.[50]

For the reasons presented in the previous paragraphs, Federico Tombari and Luigi Di Stefano have proposed the 3D Hough Voting Algorithm [50].

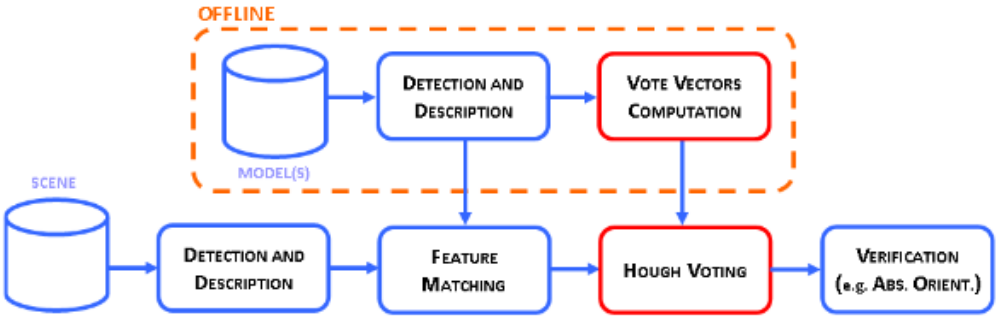


Figure 29: Hough voting scheme (red blocks) in a 3D object recognition pipeline [50]

As seen in the image above, after the registration phase, where the keypoints, their associated features and the correspondences between the features have been obtained, the Hough Voting (HV) stage follows.

The HV method is based on accumulating evidence of the presence of the model in the scene. In the case that enough features vote for the presence of the model in a given position, the object is detected and its location is estimated from the correspondences between these features.

First, when the algorithm is initialized, a single reference point is chosen. The selection of this point is arbitrary and does not affect the performance of the algorithm. Secondly, as the method should be invariant with respect to rotations and translations, it is necessary to create a LRF for each of the features that have been extracted above. To obtain the LRFs, the three eigenvectors are extracted using EVD from the distance-weighted covariance matrix of a local neighbourhood of the feature. Once the frames are obtained, a process to make them unique and unambiguous is applied.

As shown in figure 29, the Hough method consists of both an offline and an online step. On the one hand, the offline step represents the initialization of the Hough accumulator. Assuming that, at the beginning, all points are referenced with respect to the Global Reference Frame (GRF), for each of the features, the vector between it and the initialization point is obtained:

$$V_{i,G}^{M} = C^{M} - F_{i}^{M} \tag{20}$$

Then, to make this vector invariant for rotations and translations, it is transformed into the corresponding LRF.

$$V_{i,L}^{M} = R_{GL}^{M} \cdot V_{i,G}^{M} \tag{21}$$

where · represents the product of matrices and R_{GL}^{M} is the rotation matrix. In this matrix, each line represents a vector of the LRF of each feature.

$$R_{GL}^{M} = \left[L_{i,x}^{M}, L_{i,y}^{M}, L_{i,z}^{M}\right]^{T}$$

Finally, in the offline stage, each feature is associated with its corresponding $V_{i,G}^{M}$ vector.

On the other hand, in the online stage, from the correspondences obtained during the registration, each feature with a correspondence found casts a vote for the position of the reference point in the scene. As previously explained, the LRF is invariant to rotations and translations, therefore, the transformation between the LRF of the scene and the model can be performed $V_{i,L}^{S} = V_{i,L}^{M}$. To finally obtain V in the GRF of the scene [50].

$$V_{i,G}^{S} = R_{LG}^{S} \cdot V_{i,L}^{S} + F_{j}^{S} \qquad (22)$$

where R is the rotation matrix of the scene.

$$R_{GL}^{S} = \left[L_{j,x}^{S}, L_{j,y}^{S}, L_{j,z}^{S}\right]^{T}$$

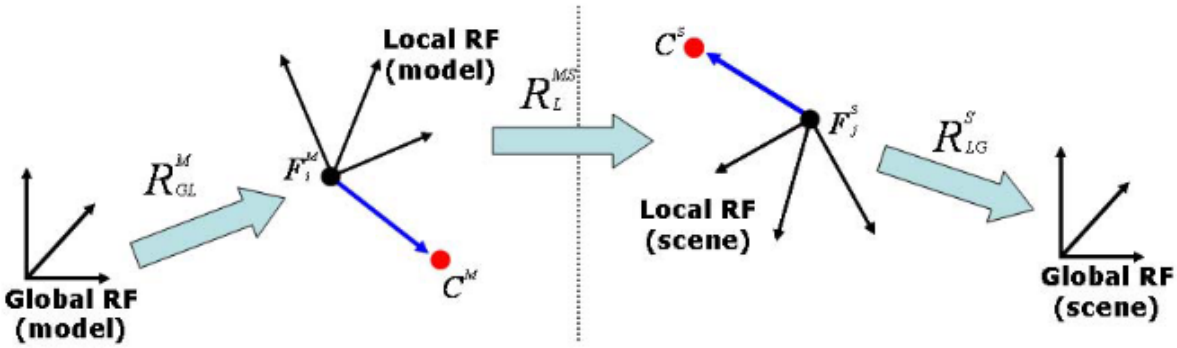


Figure 30: Transformations induced using LRF [50]

With these transforms, the feature F can be cast into a small 3D Hough space by means of the vector V.

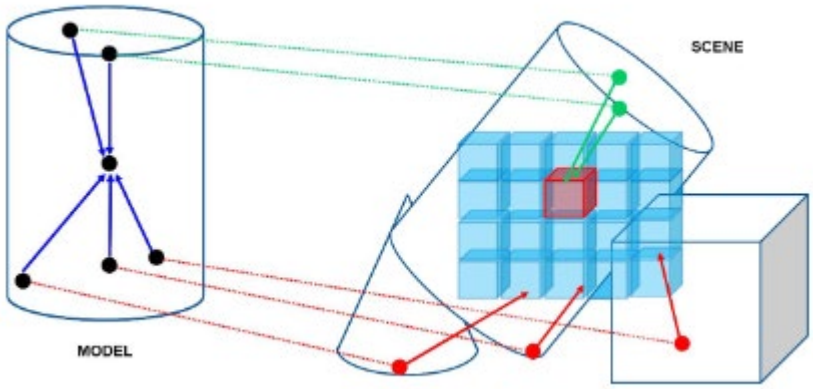


Figure 31: Example of the 3D Hough Voting scheme [50].

Ultimately, the detection of the presence of the model in the scene is evaluated by thresholding peaks in the Hough space. Multiple peaks in the space represent multiple locations of the

model. Once the model is located in the scene, the transform to be performed between the two point clouds is obtained.

In conclusion, the Hough3D method clearly outperforms, as demonstrated in the experiments carried out by Federico Tombari and Luigi Di Stefano, the other two classically used algorithms for recognition, Clustering and GC. Therefore, this algorithm is the most suitable for object recognition in Real-Time applications in complex scenes.

3. Approach

The programming of the functionalities necessary to recognize objects in the point clouds will be carried out using four software packages. These open-source software packages allow the use of their functionalities through the introduction of libraries in the header of the programs. Interaction with ROS/PCL-based programs, Gazebo simulations and the Rviz viewer will be done through the Ubuntu terminal. For this, a series of specific commands are used. In addition, a pre-assessment will also be included to compare, in those procedures where more than one method is available, the characteristics of each method. The purpose of the comparison is a better understanding of the methodologies when implementing and simulating them.

Firstly, using the Point Cloud Library (PCL), the methods and objects necessary to process and recognize objects are implemented. Therefore, as it is the most widely used library in the project, most of the libraries and functionalities implemented come from here.

Secondly, Robot Operation System (ROS) is used for communication between programs and functionalities by creating topics. The software allows communication between the different parts of the cloud processing algorithm as well as communication with the rest of the robot.

Thirdly, Gazebo is used to simulate the process and the possible situations to obtain the initial point clouds.

Finally, Rviz, in conjunction with the integrated PCL viewer, will be used to check visually the state of the initial point clouds. In addition, by using visual inspection, the quality of the result obtained by some methods will be checked.

3.1 PCL

Point Cloud Library (PCL) is an open-source library dedicated to providing algorithms for point cloud processing. The library contains the necessary methods to implement each of the steps necessary for object recognition. The available algorithms are divided into modules or objects according to the functionality for which they have been designed. Specifically, the procedures that have been proposed during State-of-the-art are implemented in the following functions which can be found in PCL modules or objects.

Preprocessing

Outlier removal

- **Statistical Outlier Removal**: Module Filters pcl::StatisticalOutlierRemoval <pcl::PCLPointCloud2>.

- **Radius Outlier Removal:** Module Filters pcl::RadiusOutlierRemoval <pcl::PCLPointCloud2>.

Segmentation

- **SACSegmentation:** Module Segmentation pcl::SACSegmentationFromNormals <PointT, PointNT>.

Downsampling

- **Voxel Grid Downsampling:** Module Filters pcl::VoxelGrid <pcl::PCLPointCloud2>.

Euclidean Cluster Extraction

- **Kd-Tree**: Module Search pcl::search::KdTree <PointT, Tree>.
- **Extract indices**: Module Filters pcl::ExtractIndices <pcl::PCLPointCloud2>.
- **SACSegmentation**: Module Segmentation pcl:: SACSegmentationFromNormals <PointT, PointNT>.
- **Euclidean Cluster Extraction**: Module Segmentation pcl::EuclideanClusterExtraction <PointT>.

Registration

Keypoint Extraction

- **Uniform Sampling:** Module Filtering pcl::UniformSampling< PointT >
- **SIFT Keypoint**: Module Keypoints pcl::SIFTKeypoint< PointInT, PointOutT >
- **ISS Keypoint**: Module Keypoints pcl::ISSKeypoint3D< PointInT, PointOutT, NormalT >

Features Extraction

- **SHOT_OMP Features**: Module Features pcl::SHOTEstimationOMP <PointInT, PointNT, PointOutT, PointRFT>.
- **Normal Estimator OMP**: Module Features pcl::NormalEstimationOMP <PointInT, PointOutT>.

Correspondence Estimation

- **CorrespondencesPtr:** Object without a specific module pcl::CorrespondencesPtr.
- **KdTreeFLANN**: Module KdTree pcl::KdTreeFLANN <PointT, Dist>.

Recognition

- **LRF Estimator**: Module Features pcl::BOARDLocalReferenceFrameEstimation <PointInT, PointNT, PointOutT>.
- **Hough 3D grouping**: Module Recognition pcl::Hough3DGrouping <PointModelT, PointSceneT, PointModelRfT, PointSceneRfT>.

On the other hand, to make use of the functions and objects present in PCL, the libraries containing the functions and objects required must be included in the head of the programs. The libraries necessary for each procedure are explained in the implementation section. Therefore, in this section, emphasis will be placed on those general libraries needed for the operation of any of the programs.

- #include <pcl/point_cloud.h>: required to include basic PCL capabilities.
- #include <pcl_conversions/pcl_conversions.h>: includes, among others, conversions between ROS messages and point clouds.
- #include <sensor_msgs/PointCloud2.h>: allows the creation of PointCloud2 messages.
- #include <pcl/visualization/pcl::visualizer.h>: allows the use of the PCL software's built-in viewer to observe internal variables or output without the need to create a ROS topic.

3.2 ROS

Robot Operation System (ROS) is a software that defines the interfaces, components and tools for controlling robots. As is well known, robots can be divided into different subfields. In this case, they will be divided into actuators, sensors and the robot processors. For a robot to operate effectively and efficiently, the different parts must be connected and coordinated. For this purpose, ROS employs what are defined as nodes, topics and messages.

First of all, it will be defined what is meant by a node in ROS. A node is an executable file, in charge of performing an action, a service or publishing or subscribing to some information. A fundamental part of the functioning of ROS is the communication between these nodes. This occurs with subscribers, publishers and messages.

- Publisher: the node designated as publisher is committed to send an information periodically, to which the rest of the nodes will be able to access. A publisher node sends the information regardless of whether other nodes are receiving it.

- Subscriber: This type of node is the "complement" of the publisher as it is a node in charge of reading the information that has been published by another node.

- Services: This method is similar to the one used in the Publisher/Subscriber duality. However, in contrast to the publisher, it sends information when the client node sends it a message called "request". At this point, the server node performs the pertinent operations on the request and sends a message known as "response". In this way, information is only transmitted when there is a request message. Making an analogy with the two types of nodes described above, the publisher node would be in this case the server node, while the client node would be the subscriber node.

In addition, it should be mentioned that the messages in ROS can be stored in a multitude of formats and be used in simulators such as Rviz, which allow experimentation with Digital Twins of real robots [51].

Secondly, topics are the way nodes communicate with each other. Through them, information flows between, for example, a publisher node and a subscriber node. Information in ROS, as explained in the next paragraph, is transmitted in the form of messages. Therefore, each topic will act as the transmission channel for the messages. It is important to mention that each topic can only transmit one type of messages.

Thirdly, messages, as just mentioned, are the way in which information is structured. In the case of a message generated by a service, it will have a part called request and a part called response [52].

In order to use the full capabilities of ROS, the corresponding software library must be included:

- #include <ros/ros.h>: necessary to include the basic ROS functionalities.

3.3 Gazebo

Gazebo is an open-code software used to simulate robots and the processes they are involved in. For this purpose, the elements must be modelled both physically and by defining the necessary variables obtained using sensors. The use of Gazebo in this thesis is limited only to obtaining point clouds considering the different cases of study. The models of both objects and sensors have been provided.

The models change for the different case studies is done by substituting them in the dedicated folder.

3.4 Rviz

Rviz (ROS visualizer) is used as a tool to check the goodness of the results of the different methods. This is done by observing the resulting point cloud and its characteristics in this viewer. The use of the visualizer to fine-tune the values is of great value as it allows to see the effects on the output when a change in the parameters is made. This allows obtaining in a fast and effective way correct values for the parameters, besides being totally understandable as it is in a visual format.

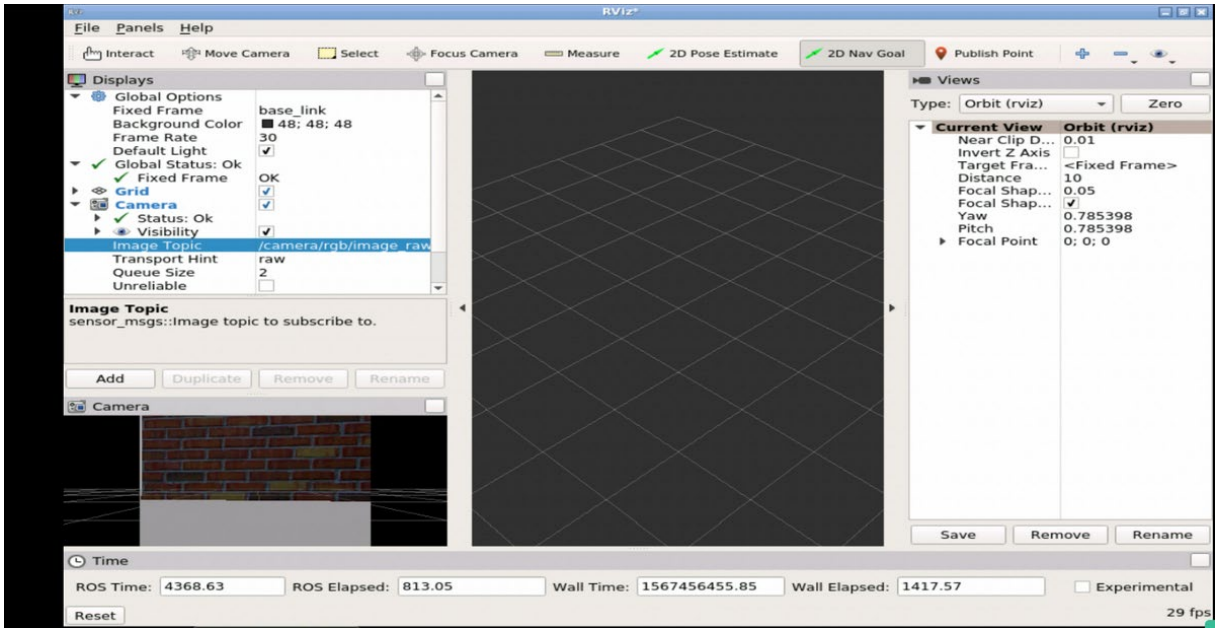


Figure 32: Example of Rviz control panel [53].

The right panel, as can be seen in the figure, is where the configuration of the visualizer is carried out. In the case of this thesis, the general frame is called Odom, therefore, to be able to visualize the different topics, this general frame must be selected.

On the other hand, when selecting the topic to be displayed, an object of the same type as the message to be displayed must first be created. In other words, the type of message for which the topic has been created. After having created this object, the topic to visualize must be selected in the tab "Make sure tab name".

3.5 Pre-assessment

3.5.1 Outlier removal

This section provides a comparison of the two proposed methods for outlier removal, Statistical and Radius. Table 1 summarizes and compares the attributes of the two methods. First, the parameters chosen for the comparison between the different methods are explained, using a numerical scale from 1 to 5 that qualifies the performance of each method with respect to the other.

1. Precision in the elimination of outliers: the aim is to establish the precision in the elimination of noisy points, i.e., the greater the precision, the higher the score.
2. Higher total filtering: another positive characteristic of an outlier removal method is the number of points that it eliminates since it decreases the processing time in later stages.
3. Ease of implementation compares in both methods the simplicity of implementation, as well as its understandability and parameterization.

Table 1: Qualitative comparison between possible methods for outlier removal.

	Precision in the elimination of outliers	Higher total filtering	Ease of implementation
Radius Outlier Removal	4	5	4
Statistical Outlier Removal	5	3	3

3.5.2 Keypoint Extraction

Similarly to the Outlier removal step, more than one method has also been proposed for the extraction of Keypoints. Therefore, a comparison of them is presented in table 2.. The characteristics to be evaluated in the table are:

1. Processing speed: Indicates the processing time that the method requires to extract the keypoints from the input cloud. Logically, the shorter the processing time, the better.
2. Accuracy in the selection of keypoints: The selected keypoints are consistent and representative of the input point cloud despite differences in the capture position.
3. Ease of implementation: Rates the ease of implementation, comprehension and parameterization.

Table 2: Qualitative comparison between possible methods for keypoint extraction.

	Processing speed	Accuracy in the selection of keypoints	Ease of implementation
Uniform Sampling	5	3	5
SIFT	2	5	2
ISS	1	4	2

4. Implementation

This section will explain how each of the functionalities necessary to process the point cloud have been implemented. To do so, the operation of each of the algorithms and the functions and libraries used in the programs will be explained.

In order to facilitate the implementation of the different functionalities, a program will be designed for each of them. In this way, it is possible to have greater control over the program in the initial phases of development and, in this way, to independently evaluate each of the proposed methods. Once each of the proposed methods has been tested, they will all be integrated into a single program. This program is available in the appendix.

The implementation part and the simulation part are closely linked because, depending on the results observed in the simulation part, it can be decided to modify, extend or remove a method in the implementation part. As an example, it is the case of the Segmentation process, which has not been implemented because with the outlier removal the desired results have been achieved for both methods.

4.1 Extraction of point cloud from the simulation

In this section the algorithms and programs used to extract the point cloud from the simulation of the desired situation will be explained. This extraction is going to be carried out in a file with the format used for the storage of point clouds in PCL and ROS, point cloud data format, pcd.

4.1.1 Description of a pcd file

The pcd file has a specific structure that is divided into two main fields, the header and the set of points. The header contains all the information relating to the type of points contained in the document. On the other hand, the set of points contains the information necessary to represent these points in space.

Now, the different elements used to classify the points in the document will be described.

```
# .PCD v0.7 - Point Cloud Data file format
VERSION 0.7
FIELDS x y z
SIZE 4 4 4
TYPE F F F
COUNT 1 1 1
WIDTH 13985
HEIGHT 1
VIEWPOINT 0 0 0 1 0 0 0
POINTS 13985
DATA ascii
1.4804899 -3.5304341 -0.65766329
1.5113562 -3.5173316 -0.6576634
```

Figure 33: Header of a generic pcd file.

Now, the different elements used to classify the points in the document will be described. The different fields shown in the figure above will now be described:

- Version: indicates the version of pcd being used.
- Fields: specifies the name of each dimension/field that a point is allowed to have. In the case of the example, the Cartesian coordinates x y z are considered, however other information such as rgb, normals to each direction, colours... could be considered.
- Size: Specifies the size of each of the dimensions. It can be 1 byte (unsigned char), 2 bytes (unsigned short), 4 bytes (unsigned int) or 8 bytes (double).
- Type: Informs about the type of variable in each dimension. The accepted types are I (integer signed), U (integer unsigned) and F (float).
- Count: Indicates how many elements each dimension has. Basically, this is a way of treating each of the dimensions as a single continuous block of memory. In the case of Cartesian coordinates, they have a single element.
- Width: Specifies the width of a point cloud. In the case of an unorganised dataset, it will coincide with the number of points in the cloud.
- Height: Informs about the height of a point cloud. In case of having an unorganized dataset, it has a value of one. Otherwise, its value will be the number of rows in the document.
- Viewpoint: Specifies the acquisition point of the point cloud. This is of great value for later construction of homogeneous transforms between different coordinate systems or to help features to have a consistent orientation. The viewpoint information is expressed as a translation (tx ty tz) and a quaternion for the orientation (qw qx qx qy qz).
- Points: Expresses the total number of points held in the cloud.

- Data: Informs about the type of data stored in the dataset, usually stored in ascii format.

4.1.2 Pcd file extraction from the simulation

Once the pcd file format and the information it contains have been introduced, the next step is to describe how this file has been obtained from the robot simulation. To extract the file, the ROS environment already has a built-in command to save the file from a PointCloud2 type topic.

By running this command in the Ubuntu terminal, the pcd file would be obtained from the topical. However, a problem arises when getting it from the available process simulation. This is because in the simulation the point cloud is contained in a PointCloud format, which is deprecated for the latest versions of ROS and PCL. As can be deduced, this difference in formats will stop the command from working.

Because of this, a format conversion from PointCloud to PointCloud2 must be made. This change will be done through a function from the previous version of ROS, Diamondback. This function created by Radu Bogdan Rusu, under BSD license, is available in the public repository at www.ros.org.

The function, called converter.cpp, is listed in the appendix of this document. It basically uses a series of functions belonging to the PointCloud library <sensor_msgs/point_cloud_conversion.h> to perform the relevant conversions between PointCloud and PointCloud2. As for the interaction with the rest of the algorithm, as extracted from the code, this function consists of a subscriber and a publisher. The function of the subscriber is to obtain the information of the topic to be transformed. Then, the format transformation between PointCloud and PointCloud2 is performed. Finally, the point cloud is published in PointCloud2 format from a Publisher under the topic /Points2_out.

4.1.3 Reading from pcd file

Once the point cloud has been stored in the pcd file, the pcd file must be read in order to have the information available in a topic. This task will be performed by the code contained in read_pcd_node.cpp, available in the read_pcd package. The main functions used to perform this functionality are described below. The complete code can be found in the appendix.

In first place, the necessary libraries must be added to implement the desired function. In this case, the function used to read pcd files is found in the pcd_io.h library. Therefore, in the head of the program, it must be added:

```
#include <pcl/io/pcd_io.h>
```

As the information extracted from the file must be shared, it is necessary to create a publisher node that publishes the information extracted from the file. This is done with the instruction below, where "output" is the name of the topic being published. The message type supported by the topic will be of type sensor_msgs::PointCloud2.

```
ros::Publisher pub = nh.advertise <sensor_msgs::PointCloud2> ("output", 1);
```

This creates a message of the type mentioned in the topic.

```
sensor_msgs::PointCloud2 output;
```

Finally, an object of type PointCloud must also be created to store the information contained in the document.

```
pcl::PointCloud<pcl::PointXYZ> point_cloud;
```

Once all the necessary nodes, messages and variables/objects have been created, the file is read. This is done using the <pcl/io/pcd_io.h> library function. The loadPCDFile function reads the file and stores it in a PointCloud variable.

```
pcl::io::loadPCDFile ("6304000.pcd", point_cloud);
```

It is of vital importance that the file is located in the folder specified in the current path. Normally, following the given recommendations, a folder called data should be created inside the corresponding package.

Finally, the PointCloud information must be transformed into ROS message so that it can be sent through the created topic. For this purpose, the following command will be used:

```
pcl::toROSMsg (point_cloud, output);
```

4.1.4 Saving pcd file

Similar to the code required to read the information from a .pcd file and use it in a variable of the PointCloud type, it is also necessary to save the point cloud obtained after applying the different processes. This way, it is possible to modularize the other procedures to be applied by holding the result obtained after the execution of each one of them to use in subsequent operations.

In the same way, as in the pcd file reading program, a library is required to access the functions for operating with pcd files.

```
#include <pcl/io/pcd_io.h>
```

In contrast to the previous section, the aim now is to record the information generated after a process in a file. Therefore, a subscriber node is needed to "read" the information obtained from the corresponding output topic.

```
sub = nh.subscribe("PC_radius_removal", 10, cloudCB);
```

This instruction is indicating to subscribe to the "PC_radius_removal" topic, with a refresh rate of 10, and the cloudCB void function is called.

In the following paragraphs, a void function CloudCB is described, where the information obtained by the subscription is saved. In the first place, the head of the process is declared. In addition to indicating the type of function, the inputs of the operation must be introduced. In this case, a variable of type PointCloud2 is declared using a pointer called input.

```
void cloudCB(const sensor_msgs::PointCloud2 &input)
```

Secondly, following the same reasoning as in the previous section, a conversion from ROSMsg to PointCloud must be performed to use the variable created in the function header for storing it in a pcd file.

```
pcl::fromROSMsg(input, cloud);
```

Finally, the point cloud obtained is saved in a pcd file using the following command.

```
pcl::io::savePCDFileASCII ("PC_outlier_radius.pcd", cloud);
```

4.2 Pre-processing

In this section the content of the different programs and functions to implement each of the functionalities required for each of the pre-processing steps will be explained. For this purpose, the methods considered in the pre-assessment part will be implemented in each of the preprocessing steps. With this, it is intended, by means of experimentation, to see which of the methods considered has a better result for the situation available.

4.2.1 Outlier removal

This section explains the programs designed for each of the methods selected to develop the outlier removal. As a reminder, the outlier removal procedure is carried out to remove the points that represent the noise in the measurement, i.e. those that are not representative of the object to be analyzed.

The first method to be implemented is statistical outlier removal. After this, the program related to radius outlier removal will be explained.

4.2.1.1 Statistical Outlier Removal

The statistical outlier removal removes those points that do not fall within the specified Gaussian distribution. The Gaussian distribution, as discussed earlier in this document, is created from the mean of the distances from a point to several neighboring points and the standard deviation obtained from the previous points, multiplied by a factor. Once this information has been recalled, it will proceed to explain the most important lines of code.

In first place, the necessary libraries must be added to implement the desired function. In this case, the function used to perform the outlier removal using the statistical method. Therefore, in the head of the program must be added:

```
#include <pcl/filters/statistical_outlier_removal.h>
```

This function, as can be deduced, will require as input the initial point cloud previously obtained from the simulation. Therefore, it is required to create a subscriber node to be able to read the information of the corresponding topic. In addition, the point cloud resulting from the outlier removal must also be published. For this purpose, a publisher will be created to publish this message.

```
sub = nh.subscribe("output", 10, &cloudHandler::cloud_filtering, this);

pub = nh.advertise<sensor_msgs::PointCloud2> ("PC_outlier_removal_statistical", 1);
```

In this case, in contrast to the programme designed for reading from the pcd file, two point clouds will be needed. One of them as input for the method and the other as output. The way to create them is identical to the one used in the read_pcd_node.cpp, therefore, it is no necessary to comment it once again here.

Afterwards, the StatisticalOutlierRemoval object type is created to be able to apply the procedure to the input point cloud. To do so, using the function provided in the included library, the code is written:

```
pcl::StatisticalOutlierRemoval<pcl::PointXYZ> statitistical_Filter;
```

Once the object has been created, it needs to be parameterized, following the PCL guide as follows:

- **PointCloud input**: In this case, the function makeShared() is being used since the PointCloud object is being passed to the Void function as a pointer.

  ```
  statitistical_Filter.setInputCloud(initial_cloud.makeShared());
  ```

- **Number of neighbouring points** used to create the Gaussian distribution.

  ```
  statitistical_Filter.setMeanK(50);
  ```

- **Multiplie**r of the standard deviation:

  statitistical_Filter.setStddevMulThresh(0.3);

- **Point cloud output**: In this case the point cloud has not been given as an input to the function in the form of a pointer. Therefore, the auxiliary function makeShared() is not required.

  statitistical_Filter.filter(outrem_statistical_cloud);

Finally, as in previous programs, the output point cloud is transformed into a ROS message to be published by the publisher node that was created earlier.

4.2.1.2 Radius Outlier Removal

The Radius Outlier Removal method removes the points that do not have a minimum number of neighboring points within a sphere (in the 3D case) of a particular radius. By means of this geometric approach it is possible to eliminate the noisy points or the ones that have no relevant information.

In the same way as with Statistical Outlier Removal, it is necessary to add the library where the function is placed. Therefore, it is added in the function header:

#include <pcl/filters/radius_outlier_removal.h>.

According to the same reasoning as before, it is also created a subscribe node, in order to read the input information of the corresponding topic, and a publisher node, to publish the information obtained after applying the method.

sub = nh.subscribe("output", 10, &cloudHandler::cloud_filtering, this);

pub = nh.advertise<sensor_msgs::PointCloud2> ("PC_radius_removal", 1);

Subsequently, as has been done in the function to store point clouds, the variable that is also the input of the same function is created in the head of the void function, created to perform the outlier of the point cloud, the variable that is also the input of the same function.

void cloud_filtering (const sensor_msgs::PointCloud2& input) {

Then, inside this function, the different point clouds necessary for the application of the method are created and the necessary conversions are performed in the same way as described before. After this, an object of type RadiusOutlierRemoval must be created to be able to use the method.

pcl::RadiusOutlierRemoval<pcl::PointXYZ> radius_Filter;

Once the object has been created, a value must be assigned to its parameters:

- **Determination of the radius of the sphere** that is created at each of the points to obtain the number of neighbors.

  radius_Filter.setRadiusSearch(0.07);

- **Minimum number of neighbors** that must be in the sphere to consider the point as valid.

  radius_Filter.setMinNeighborsInRadius(8);

- **Maintain the same order as in the cloud**, i.e. keep the original indices for those points considered as valid.

  radius_Filter.setKeepOrganized(false);

Finally, the application of the filter is also performed using the RadiusOutlierRemoval object created earlier. As in previous functions, the result has to be prepared to be communicated through a node of the Publisher type.

radius_Filter.filter(radius_removal_cloud);

4.2.2 Downsampling

Once the outlier removal process has been executed, in which those points considered to be invalid have been eliminated, the downsampling procedure is applied. In this procedure a percentage of points are removed from the cloud to increase the processing speed of the cloud without losing relevant information to then carry out the rest of the steps.

The Voxel Grid method consists of replacing all the points within a voxel by its centroid. In the case of this project, as the work dimension in 3D, the voxels will have the appearance of a cube or prism.

In the same way as before, the library where the function is found is included.

#include <pcl/filters/voxel_grid.h>

Afterwards, the two corresponding nodes are created. On the one hand, the subscriber node, which in this case will not read the information coming from the reading of the initial cloud but will read the output of the previous step, in other words, from the outlier removal. On the other hand, the publisher node will publish the result of the application of the method.

sub = nh.subscribe("PC_radius_removal",10, &cloudHandler::cloudCB, this);

pub = nh.advertise<sensor_msgs::PointCloud2> ("PC_downsampled_voxel",1);

The implementation of the function is done in a separate void function from the main function, following the same style of declaration as in the previous section.

void cloudCB (const sensor_msgs::PointCloud2& input){

Now, the necessary clouds to apply the method should be created, as well as the necessary conversions to be able to operate with the information obtained through the subscriber node. After this, the object for the application of the Voxel Grid method is created.

pcl::VoxelGrid<pcl::PointXYZ> voxel_downsampling;

After the object has been created, it needs to be parameterized to be used for the processing of the input cloud.

- **Definition of the Voxel dimensions**: by defining this size, the density of points will be reduced to a greater or lesser extent.

  voxel_downsampling.setLeafSize(0.02f, 0.02f, 0.02f);

After setting the only necessary parameter the application of the method is reapplied and the result is saved in the created cloud as output.

voxel_downsampling.filter (downsampled_cloud);

4.2.3 Cluster extraction

This method is used in two different phases of the processing of the point cloud. This section is going to focus on explaining the procedure necessary to extract the models. It is extremely important to mention that this Pre-Processing step is performed only once to obtain each of the desired models and, logically, it will not be included in the main program that contains the total processing of the point cloud.

Once this has been made clear, it proceeds to explain the different algorithms and procedures that must be applied to obtain the different clusters. It is very important to mention that, to obtain the clusters, procedures that have already been described previously, such as Downsampling or reading from a pcd file, must be applied. In order to reduce the number of active Subscriber and Publisher nodes, some of these programs have been integrated in the same program.

On the other hand, as this program is considerably more complex than those mentioned earlier in this document, it is considered useful to interact with the user by displaying important information on the screen relating to the output of the method.

Firstly, the initial cloud is read, which in this case will be the cloud output from applying the selected outlier removal procedure using the function explained above. Once the reading has been done, the initial number of points of the cloud is published to give the user an idea of the initial density.

std::cout << "PointCloud before filtering has: " << cloud->size () << " data points." << std::endl;

The cloud obtained after applying downsampling has not been chosen as the initial cloud because the correct obtaining of the clusters is closely related to the density of points. Therefore, it has been considered that, despite having optimized the downsampling process, flexibility would be given to the recognition algorithm by placing the procedure shown in the program. The way to implement this functionality has already been described in the downsampling section. In addition, the number of points after downsampling is displayed on the screen.

Planar Segmentation and Extraction

It now applies the procedure to eliminate the planar elements and thus obtain a greater precision and speed of processing of the clusters. To do this, the library containing the functions to perform the segmentation of planar elements must be included.

```
#include <pcl/segmentation/sac_segmentation.h>
```

Then all the necessary objects to perform the segmentation of planar elements and the elimination of points must be created. In addition to these objects, it is also necessary to create the necessary indices for the points as well as an object for the segmentation of planar elements. The necessary number of points as well as an object (write_pcd) to save the results are also created.

- Creation of the planar segmentation object.

  ```
  pcl::SACSegmentation<pcl::PointXYZ> segmentation;
  ```

- Creation of the object that will contain the point indices of the cloud.

  ```
  pcl::PointIndices::Ptr inliers (new pcl::PointIndices);
  ```

- Creation of the object that defines the model of segmentation to be performed.

  ```
  pcl::ModelCoefficients::Ptr coefficients (new pcl::ModelCoefficients);
  ```

Once the necessary objects have been created, they are parametrized by program.

- **Enable coefficients refinement** by setting the Boolean input to true.

  ```
  segmentation.setOptimizeCoefficients (true);
  ```

- **Set the type of geometric model** that will be searched.

  ```
  segmentation.setModelType (pcl::SACMODEL_PLANE);
  ```

- **Set the type of sampling consistency method** used. RANSAC method has been adopted as it is the simplest and most effective one. Therefore, the corresponding library must be included in the header.

segmentation.setMethodType (pcl::SAC_RANSAC);

#include <pcl/sample_consensus/method_types.h>

#include <pcl/sample_consensus/model_types.h>

- **Indicates the maximum number of iterations** allowed.

  segmentation.setMaxIterations (100);

- This function cooperates with the model specified by the user and **sets the threshold of the distance** from the point to the model. If the distance from the point to the model does not exceed this distance threshold, the point is considered an inside point, otherwise, it is considered an outlier point.

  segmentation.setDistanceThreshold (0.02);

Having defined the object to perform the segmentation of the planar surface, the extraction of these surfaces from the point cloud is performed. For that purpose, as can be extracted from line 42 of the program cluster_extraction.cpp, a loop is applied. Firstly, the planar element will be obtained using the object that has just been parameterized and, secondly, the indices of the points that make up the element will be extracted to eliminate them from the original point cloud.

1. **Obtention of the planar element.**

   segmentation.setInputCloud (cloud_filtered);

   *segmentation.segment (*inliers, *coefficients);*

2. **Extraction of indices**: For the extraction of the planar element, an object of type pcl::ExtractIndices<pcl::PointXYZ> is used, therefore, the corresponding library must be included.

   #include <pcl/filters/extract_indices.h>

   Afterwards, as has been explained several times in this section, the corresponding object is created and parameterized.

   pcl::ExtractIndexes<pcl::PointXYZ> extraction;

   extraction.setInputCloud (cloud_filtered);

   - **Indices of the points** to be extracted.

     extraction.setIndexes(inliers);

   - **Elimination of the points extracted**.

extraction.setNegative (true);

After the parameterization, the result of the index extraction is showed in a cloud of points. In this case, as the element extract.setNegative is set to true, the cloud of points resulting from having eliminated those points to be extracted from the input cloud will be extracted.

*extraction.filter (*cloud_f);*

After finishing the extraction process, the value of the loop condition is updated.

Cluster Extraction

To perform the extraction of clusters, first, a Kd_tree is created to calculate the distances between the points.

pcl::search::KdTree<pcl::PointXYZ>::Ptr KdTree (new pcl::search::KdTree<pcl::PointXYZ>);

KdTree ->setInputCloud (cloud_filtered);

After obtaining the distances, the libraries, the object and the parameters to extract the clusters are included.

#include <pcl/segmentation/extract_clusters.h>;

pcl::EuclideanClusterExtraction<pcl::PointXYZ> euclidean_extraction;

- **Cluster tolerance** indicates the tolerance (in meters) for cluster creation.

  euclidean_extraction.setClusterTolerance (0.13);

- **Minimum and maximum sizes** (number of points) of each cluster.

  euclidean_extraction.setMinClusterSize (80);

  euclidean_extraction.setMaxClusterSize (10000);

- **Method to compute distances** between a point and its neighbours, in this case, the previously computed Kd-Tree is introduced as input.

  euclidean_extraction.setSearchMethod (KdTree);

- **Point cloud input**.

  euclidean_extraction.setInputCloud (cloud_filtered);

Once the object has been configured, it is executed. Due to the type of output provided by this library, a vector must be created to store the indices of the points that make up the different

clusters. It is important to consider that each of the i-entries of the vector (cluster_indices[i]) will correspond to a cluster.

```
std::vector<pcl::PointIndexes> cluster_indices;

euclidean_extraction.extract (cluster_indices);
```

Finally, a loop is created (lines 90-105) to separate each of the clusters that have been extracted into independent point clouds.

4.2.4 Point Cloud Registration and Recognition

After all the Pre-processing steps have been done, the registration of the point clouds and then the recognition of objects in the input cloud is carried out. Although these two processes have been described independently previously, when it comes to their implementation, due to their close relationship in terms of variable sharing, it has been decided to implement a single program that encompasses both procedures. This way, possible problems derived from many variables to be communicated between processes are avoided.

4.2.4.1 Point Cloud Registration

KeyPoint Extraction

The first step in the registration of two point clouds is to extract the keypoints. For this purpose, the necessary objects must be created for both, the model and the scene. Below is shown the code for each of the keypoint extraction methods that have been researched. Only the extraction of keypoints for the model is shown, as the extraction for the scene is done in the same manner.

1. **Uniform Sampling**

   ```
   #include <pcl/filters/uniform_sampling.h>

   pcl::UniformSampling<PointType> keypoints_uniform;
   ```

   - **Input point cloud** provides the input point cloud as a pointer to the input dataset.

     ```
     keypoints_uniform.setInputCloud (model);
     ```

   - **Set the 3D grid** leaf size.

     ```
     keypoints_uniform.setRadiusSearch (0.02f);
     ```

   - **Call to the filtering method** obtaining the filtered dataset as an output.

*keypoints_uniform.filter (*model_keypoints);*

2. **SIFT Keypoint extraction**

#include <pcl/keypoints/sift_keypoint.h>

pcl::SIFTKeypoint <PointType, pcl::PointWithScale> keypoints_SIFT;

Creation of the search Kd-Tree object needed for the application of the SIFT keypoint extraction method.

pcl::search::KdTree <PointType>::Ptr tree_SIFT (new pcl::search::KdTree<PointType>());

- **Input point cloud** provides the input point cloud as a pointer to the input dataset.

  keypoints_SIFT.setInputCloud (model);

- **Search object** used for the detection of the closest points.

  keypoints_SIFT.setSearchMethod (tree_SIFT);

- **Range of scales** over to which search for keypoints, including the minimum scale, the number of octaves and the number of scales per octave.

  keypoints_SIFT.setScales(min_scale_mod,number_oct_mod,number_scales_octave_mod);

- **Minimum contrast** as a threshold to limit detection of keypoints without sufficient contrast.

  keypoints_SIFT.setMinimumContrast (min_contrast_mod);

- **Application of the SIFT keypoint detection method** for all points given as an input.

  keypoints_SIFT.compute (result_model);

3. **ISS Keypoint extraction**

The ISS method for keypoint extraction requires the resolution of the analyzed point clouds. Therefore, a program is created to estimate the resolution of the cloud as the minimum distance between the points that make up the cloud. This is done using a search tree to obtain this distance.

#include <pcl/search/kdtree.h>

pcl::search::KdTree<pcl::PointXYZ> tree;

```
tree.setInputCloud (cloud);
```

The smallest distance between points in the cloud is obtained by iterating through the cloud using a for loop.

```
for (std::size_t i = 0; i < cloud->size (); ++i) {
```

Using the KdTree object previously created the closest point to the current point of study is calculated.

```
nres = tree.nearestKSearch (i, 2, indices, sqr_distances);
```

Then, after verifying that a point associated to the studied one has been found, the resolution is updated with the distance between the pair of points.

```
if (nres == 2) {
    res += sqrt (sqr_distances[1]);
    ++n_points;
}
```

After the calculation of the model and scene resolution, the keypoints can be extracted using Intrinsic Shape Signatures method.

```
#include <pcl/keypoints/iss_3d.h>
pcl::ISSKeypoint3D <PointType, PointType>  keypoints_ISS;
pcl::search::KdTree<PointType>::Ptr tree_ISS (newpcl::search::KdTree
<PointType>());
```

- **Input point cloud** provides the input point cloud as a pointer to the input dataset.

  ```
  keypoints_ISS.setInputCloud (model);
  ```

- **Search object** used for the detection of the closest point.

  ```
  keypoints_ISS.setSearchMethod (tree_ISS);
  ```

- **Radius of the spherical neighbourhood** used to compute the scatter matrix.

  ```
  keypoints_ISS.setSalientRadius (1 * cloud_resolution_mod);
  ```

- **Non maxima suppression algorithm** radius activation threshold.

  ```
  keypoints_ISS.setNonMaxRadius (1 * cloud_resolution_mod);
  ```

- **Upper bound on the ratio** between the **second** and the **first** eigen value.

  keypoints_ISS.setThreshold21 (set_Threshold21_mod);

- **Upper bound on the ratio** between the **third** and the **second** eigen value.

  keypoints_ISS.setThreshold32 (set_Threshold32_mod);

- **Minimum numbers of neighbours** needed while applying the non-maxima **suppression algorithm**.

  keypoints_ISS.setMinNeighbors (min_neigh_mod);

- **Initialization of the scheduler and setting of the number of threads** to use.

  keypoints_ISS.setNumberOfThreads (num_threads_mod);

- **Application of the ISS keypoint detection method** for all points given as an input.

  *keypoints_ISS.compute (*model_keypoints);*

Features extraction

Continuing the workflow, the descriptors in each of the keypoints are now obtained to enhance the information known around the keypoints. As in the extraction of keypoints, here should also be done for both, the model and the scene.

SHOT Estimation OMP: As described in the State-of-the-art section, to obtain these features, the point cloud normal must be computed.

#include <pcl/features/normal_3d_omp.h>

pcl::NormalEstimationOMP<PointType, NormalType> norm_est;

- **Number of the k neighbours**, number of neighbours used during the feature estimation.

  norm_est.setKSearch (20);

- **Input point cloud** provides the input point cloud as a pointer to the input dataset.

  norm_est.setInputCloud (model);

- **Computation of normals** based on the method for all the points provided in the input.

  *norm_est.compute (*model_normals);*

After the normal has been computed for the point cloud and, considering also the keypoints obtained in the previous step, the features are extracted.

```
#include <pcl/features/shot_omp.h>

pcl::SHOTEstimationOMP<PointType, NormalType, DescriptorType> SHOT_descr;
```

- **Input point cloud** provides the input point cloud as a pointer to the input dataset.

  ```
  SHOT_descr.setInputCloud (model_keypoints);
  ```

- **Input normal** provides the calculated normal for the keypoints as a pointer to the input dataset.

  ```
  SHOT_descr.setInputNormals (model_normals);
  ```

- **Setting of the sphere radius** used to determine the **nearest neighbours** for the feature estimation.

  ```
  SHOT_descr.setRadiusSearch (descr_rad_);
  ```

- Provide a pointer to the **pre-processed dataset** to add additional information to estimate the features in the keypoints entered as an input point cloud.

  ```
  SHOT_descr.setSearchSurface (model);
  ```

- **Application of the SHOT features extraction method** for all points given as an input.

  ```
  SHOT_descr.compute (*model_descriptors);
  ```

Correspondences estimation

When the features of each of the keypoints have been obtained, it is the moment to obtain the correspondences between the different features to carry out the registration of both clouds. Following the ICP method it is necessary to obtain the point of the opposite cloud nearest to the point that is being analyzed. For this purpose, a Kd-TreeFLANN search tree, specially designed to speed up the nearest neighbour search process, will be used.

```
#include <pcl/kdtree/kdtree_flann.h>

pcl::KdTreeFLANN<DescriptorType> match_search;

#include <pcl/correspondence.h>

match_search.setInputCloud (model_descriptors);
```

After creating the search object and specifying the input cloud, the matches between the indices of each of the clouds are obtained. This task is performed by means of a loop that goes through all the points (logically those keypoints with their corresponding features) of the scene cloud and obtains the points of the other cloud that are at a shorter distance.

```
found_neighs = match_search.nearestKSearch (scene_descriptors->at (i), 1, neigh_indices, neigh_sqr_dists);
```

Correspondences rejection

Straight after estimating the correspondences between features, it is necessary to perform filtering of these correspondences. This is known as the correspondence rejection process. To apply this process, it will be checked if the correspondences fulfil the decided conditions. The process is implemented by means of a conditional if that is included in the loop described in the previous section. In this conditional, it is checked if the condition is fulfilled to consider the correspondence as valid, in the case of the SHOT Descriptor:

```
if(found_neighs == 1 && neigh_sqr_dists[0] < 0.25f){
```

In the case of not fulfilling the condition, the correspondence must be removed from the vector where it is stored.

```
pcl::Correspondence corr (neigh_indices[0], static_cast<int> (i), neigh_sqr_dists[0]);

model_scene_corrs->push_back (corr);
```

Homogeneous Transform Estimation

In order to extract the homogeneous transform matrix, it is used the Hough3D method, the same as the one used to perform the recognition. First, to apply this method, the object containing the homogeneous transform is created.

```
#include <pcl/common/transforms.h>

std::vector<Eigen::Matrix4f, Eigen::aligned_allocator<Eigen::Matrix4f> > rototranslations;
```

Once the object has been created, applying the method automatically obtains the transform.

```
clusterer.recognize (rototranslations, clustered_corrs);
```

4.2.4.2 Point Cloud Recognition

In the Hough3D method, as has already been inferred from the reading of the previous section, the image recognition is performed in parallel to the estimation of the homogeneous transform.

```
#include <pcl/recognition/cg/hough_3d.h>
#include <pcl/features/board.h>
```

As explained in the Recognition section, in order to apply Hough's method, the Local Reference Frames (LRF) of each of the keypoints must first be calculated.

```
pcl::BOARDLocalReferenceFrameEstimation<PointType, NormalType, RFType> LRF_est;
```

- **Input point cloud** provides the input point cloud as a pointer to the input dataset.

  ```
  LRF_est.setInputCloud (model_keypoints);
  ```

- **Input normal** provides the calculated normal for the keypoints as a pointer to the input dataset.

  ```
  LRF_est.setInputNormals (model_normals);
  ```

- Provide a pointer to the **pre-processed dataset** to add additional information to estimate the features in the keypoints entered as an input point cloud.

  ```
  LRF_est.setSearchSurface (model);
  ```

- Setting of **whether the holes in the margin of the support**, for each point, are **searched and accounted** for in the estimation to the LRF or not.

  ```
  LRF_est.setFindHoles(true);
  ```

- **Setting of the sphere radius** used to determine the **nearest neighbours** for the LRF estimation.

  ```
  LRF_est.compute (*model_rf);
  ```

When the object necessary to obtain the LRFs has been created and parameterized, the method is applied:

```
LRF_est.compute (*model_rf);
```

Now that the LRFs have been obtained for the keypoints of both the model and the scene, the Hough3D algorithm can be applied to perform the recognition (and, as introduced in the previous section, also the estimation of the homogeneous transform). As has already been done and explained many times in the document, the object that executes the method is created and parameterized.

pcl::Hough3DGrouping<PointType, PointType, RFType, RFType> Hough_recognition;

- **Input point cloud** provides the input model point cloud as a pointer to the input dataset.

  Hough_recognition.setInputCloud (model_keypoints);

- **Input Reference Frame** provides the input points local reference frames for the model, obtained in the previous step.

  Hough_recognition.setInputRf (model_rf);

- **Setting of the size of each bin** into the Hough3D space.

  Hough_recognition.setHoughBinSize (cg_size_);

- **Parametrization of the minimum number of votes in the Hough Space** needed to infer the presence of a model instance in the scene cloud.

  Hough_recognition.setHoughThreshold (cg_thresh_);

- **Setting of the use of interpolation method** for the score between neighbouring bins of the Hough space.

  Hough_recognition.setUseInterpolation (true);

- **Use of the correspondence distance** as a score for the voting procedure.

  Hough_recognition.setUseDistanceWeight (false);

- **Scene cloud** provides the scene point cloud as a pointer.

  Hough_recognition.setSceneCloud (scene_keypoints);

- **Scene Reference Frame** provides the input points local reference frames for the scene, obtained in the previous step.

  Hough_recognition.setSceneRf (scene_rf);

- **Correspondences between model and scene,** obtained during the registration phase.

  Hough_recognition.setModelSceneCorrespondences (correspondence_array);

Then, the command is executed to obtain the recognition of the model in the scene and, if applicable, the transform between the model and the scene cluster that corresponds to the model.

Hough_recognition.recognize (rototranslations, clustered_corrs);

5. Results and discussion

5.1 Simulation

The principal aim of the implemented algorithm, as explained in the introduction, is to recognize the position of the UIC Hook to distinguish between wagons ready to couple and those that are not. Therefore, through simulation, the algorithms will be parameterized and tested for each of the possible cases.

Firstly, model 0 represents, as shown in the picture, the case where the chain is supported by a bracket underneath the UIC-Hook. The wagon would be available for coupling in this case.

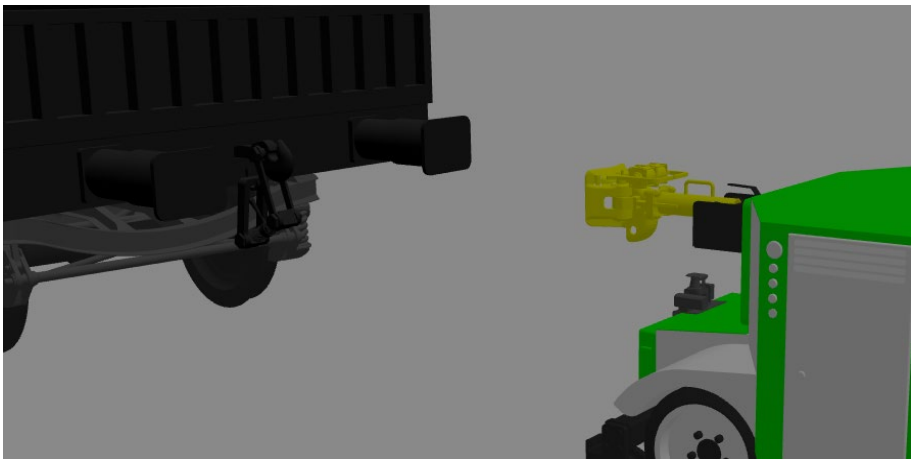


Figure 34: model 0 shown in Gazebo.

Secondly, model 1 represents, as can be seen in the picture, the case where the chain is in the UIC-Hook, this situation means that the coupling is not possible.

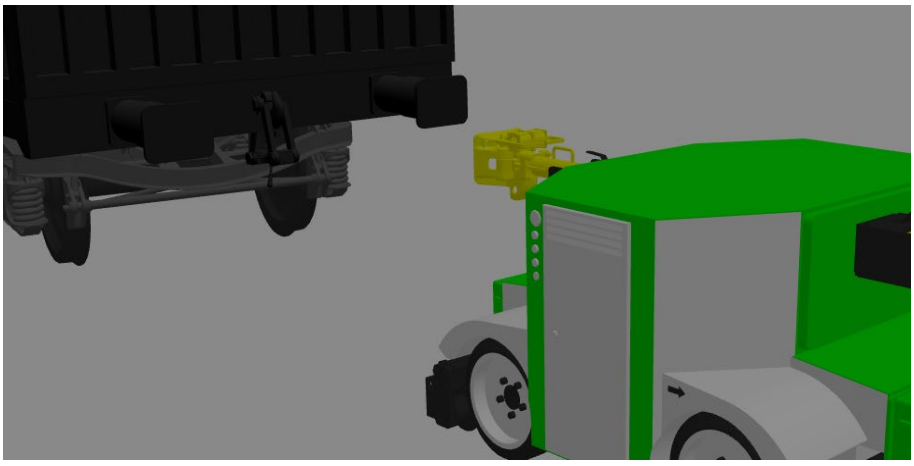


Figure 35: model 1 shown in Gazebo.

Thirdly, model 2 corresponds to the case where the UIC Hook is available for coupling because the chain is dangling.

Figure 36: model 2 shown in Gazebo.

5.1.1 Process for method and parameters selection

The process for testing algorithms and parameters in the model that has been created for simulation is empirical. In other words, the validation and improvement of the programs will be done through iterations based on analysis and observation of the results. Therefore, it is especially necessary to follow an empirical method to obtain the best outcome.

After a consultation of empirical methods followed to improve the quality of processes, it has been decided to follow the Quality Improvement Paradigm (QIP) [54]. This decision is due, on the one hand, to its specialization and wide use for continuous improvement in the field of software development. On the other hand, for the simulation process to be improved, it is the most suitable process because it does not include any phase that requires physical equipment, as well as fully covers the necessary steps for software iteration.

The QIP consists of six steps that are repeated iteratively until the desired result is achieved:

1. **Characterize**: In this step, the current situation of the problem and how it will be affected by a possible change of parameters must be understood.
2. **Set Goals**: Quantifiable objectives for improvement are indicated.
3. **Choose method/parameters**: Based on the characterization and the goals, a new method or parameters are proposed.
4. **Execute**: Carry out the simulation changing what was decided in the previous section.
5. **Analyze**: The result obtained is studied to make further changes in the first three stages. New methods or values for the parameters are proposed in this step.

6. **Package**: Store the experience and conclusions obtained during the experimentation to be used as a basis for further improvements.

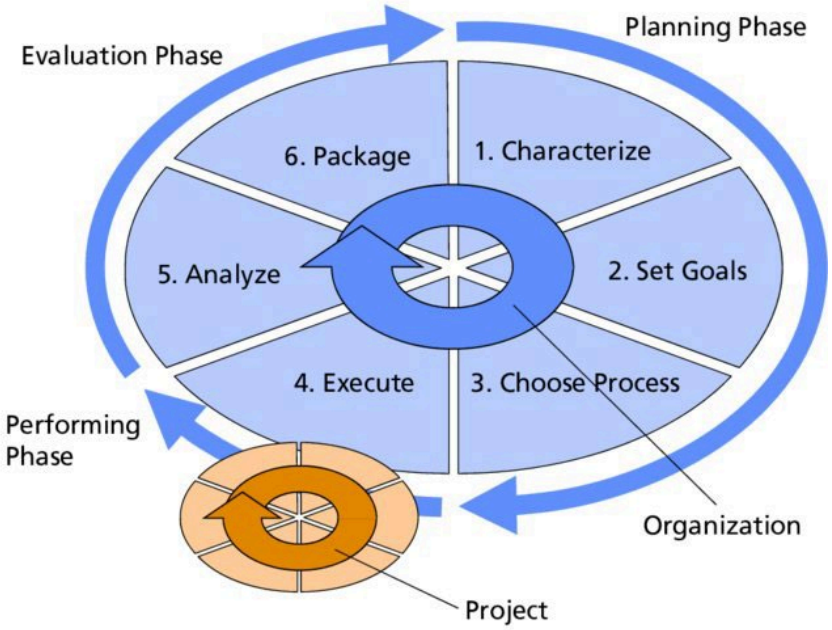


Figure 37: QIP methodology [55].

This section contains the various commands and procedures needed to simulate the algorithms, programs and procedures explained in the implementation section. To do so, it will be shown the way in which the commands required for the simulation are executed. In addition, the relationships between the different nodes and which of them are active at each moment of the process are also described.

5.2 Pcd file extraction

The pcd file is extracted from the process simulation previously designed in the Gazebo software. Therefore, the topic /rotrac_e2/laser_rear/scan, which contains the point cloud generated by the simulation, must be active. To do so, first, the information must be loaded into Gazebo. Running the following commands from the Ubuntu terminal:

$ roslaunch train_gazebo simple_world.launch

$ roslaunch train_gazebo spawn_flachwagen.launch

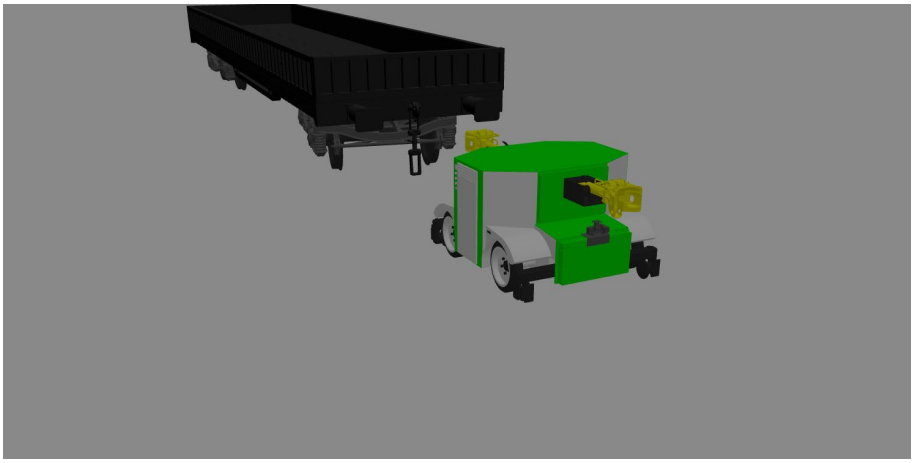


Figure 38: Simulation of the process in Gazebo.

As can see in the image above, the objects that make up the process have been loaded into Gazebo and are being simulated.

For storing the point cloud obtained, the command provided for write_pcd program must be executed. However, the point cloud must first be transformed into PointCloud2 format. To accomplish this, the command is executed in another terminal window:

$ rosrun point_cloud_converter point_cloud_converter points_in:=/my_input_pointcloud_topic points2_out:=/my_output_pointcloud2_topic

It follows, at the beginning, the typical structure of the rosrun command, $ rosrun name_pkg name_executable_cpp. In addition, in this case, it also includes the topic to which the function will subscribe in points_in, and the topic that the publisher will create in points2_out.

It should be noted that to execute the command following the rosrun instruction it is necessary that the ROS master node is running (this is executed automatically in the case of using a launch file to execute the programs, but not with rosrun instruction). This is done by typing in a terminal tab the command $ roscore.

Executed the command for the transformation the point cloud is now in the PointCloud2 format. Therefore, without stopping the execution of any of the previous nodes, the command to obtain the pcd file is executed:

$ rosrun write_pcd write_pcd_node

After executing this command, the .pcd file could be obtained from the simulation.

5.3 Reading pcd file

To perform the simulation of the pcd file reading function, it is only necessary to execute the read_pcd_node. This node will read the previously obtained pcd file and publish it in the

selected by program topic. To do this, the following command must be typed in a terminal tab, considering that the path must be in the folder where the pcd file to be read is stored:

$ rosrun read_pcd read_pcd_node

This function does not require any other function to be running at the same time, since it is a function to extract information from a file.

5.4 Saving pcd file

To save the point cloud resulting from some operation, the write_pcd_node is executed. This node will subscribe to a topic from which it will extract a message that will be saved in a file of type pcd. To do this, the command is executed in the terminal:

$ rosrun write_pcd write_pcd_node

From the observation of the initial pcd file, it is important to mention that it has a total of 29700 points.

5.5 Outlier removal

This section describes the steps and interdependencies of the programs necessary to perform the outlier removal step.

5.5.1 Model 0

5.5.1.1 Statistical outlier removal

The first step to simulate the statistical_outlier_filter is to activate the node associated with the program that performs this operation. To accomplish this, the following command is executed in the Ubuntu terminal.

$ rosrun outlier_removal outlier_removal_node

With the node now operational, the empirical experimentation using QIP of this process can begin. It is important to emphasize that, in this case, as there are no changes in the characterize and set goals phases, the iteration will be carried out from phase three onwards.

1. **Characterize**

   The method consists of the elimination of outliers using statistical elements of those points that are not within the distribution. To improve the performance of the method,

the only possibility is to optimize the value of the parameters. Therefore, in this first section, the consequences of their modification are analyzed:

- setMeanK: As this parameter is increased, more points will be considered to calculate the mean of the distribution, therefore, less possibilities that a greater number of points remain within the distribution and are not eliminated. Therefore, when the goal is to eliminate a greater number of points this parameter should be increased.

- setStddevMulThresh: The higher the value of the standard deviation multiplier, the larger the area covered by the distribution, therefore, the greater the chances that the points will remain within it. For this reason, to eliminate a larger number of points, this parameter must be reduced.

The image below shows a normal distribution where the dependence of the shape of the distribution on the change of the mean and the standard deviation can be appreciated.

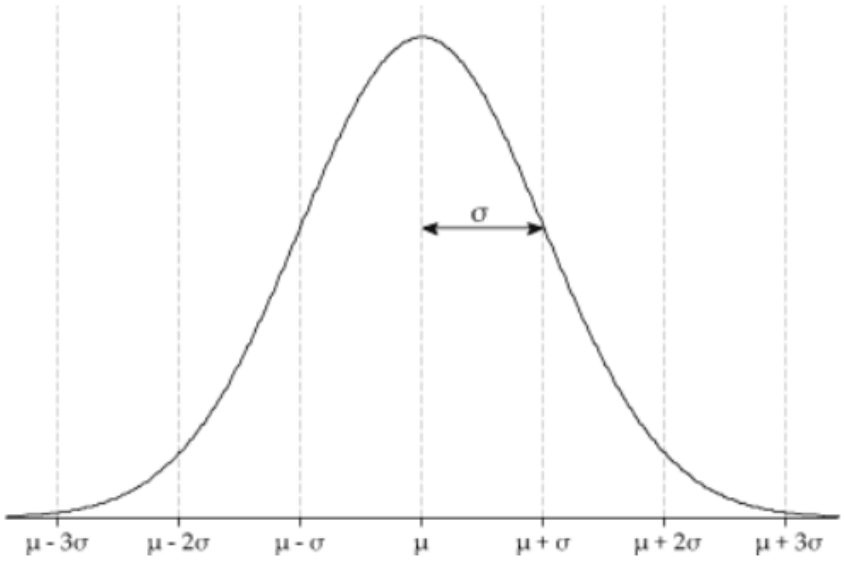


Figure 39: Normal Distribution [56].

2. Set Goals

There are two antagonistic objectives in this process, i.e. the improvement of one represents the worsening of the other. Therefore, a compromise solution where both objectives are satisfied to the greatest extent possible needs to be found.

- Reduction of points: The greater the point reduction of the initial cloud, the more effective and efficient the subsequent processes will be.

- Maintaining information on the key elements.

3. Choose parameters

From this phase onwards, the results obtained in each of the iterations are shown. The first iteration is carried out with the default values proposed in PCL due to these values have been tested in situations like the point cloud studied, so is considered a good start point for iteration.

Iteration 1: setMeanK (10), setStddevMulThresh (0.2).

Iteration 2: setMeanK (10), setStddevMulThresh (0.1).

Iteration 3: setMeanK (20), setStddevMulThresh (0.05).

Iteration 4: setMeanK (20), setStddevMulThresh (0.01).

Iteration 5: setMeanK (50), setStddevMulThresh (0.01).

Iteration 6: setMeanK (20), setStddevMulThresh (0.001).

4. Execute

Each iteration is executed in the following order:

- Interrupt the execution of the command that activates the outlier_removal_node command by using Ctrl+C in the terminal.
- Update and save the values of the parameters in the .cpp file associated with the node.
- Compilation of the program modified.
- Activation of the node by typing the command indicated at the beginning of the subsection.

5. Analysis

The first objective is analyzed through the observation of the number of points remaining in the .pcd file. The second objective is carried out by examining the resulting point cloud in the viewer. In order not to overload the document with images, the images for the iterative process are displayed in the appendix. For more details of each phase, please refer to the corresponding appendix.

Iteration 1: The noisy points caused by the interaction of the sensor with the "air" surrounding the train wagon have been eliminated. However, the points representing the ground, also considered noise, have not been eliminated. Therefore, the value of

the standard deviation multiplier is reduced by 50% to reduce the area of the distribution. In this iteration the number of points after the procedure is 14145.

Iteration 2: The number of points has been reduced to 13497. Nevertheless, the ground is still detected in the resulting cloud. Due to this, the number of points for the calculation of the mean is increased to 20 (100%), and the multiplier for the standard deviation is reduced to 0.05 (50%). The aim, as can be inferred, is to further reduce the area of the statistical distribution.

Iteration 3: After implementing the changes proposed in iteration 2, there is a reduction in the number of points to only 16 points. The value of setStddevMulThresh is updated to 0.01 (-80%).

Iteration 4: The resulting cloud has a size of 13261 points; however, the ground is still detected in that cloud. Due to this fact, the setMeanK parameter is updated to a 50 value (150%).

Iteration 5: The size of the obtained cloud is of 13985 points. In this case, this increase in the setMeanK parameter has a negative effect on the filtering of the point cloud. As a result, due to this and the fact that the ground is still being detected, the setMeanK parameter is returned to 20 and the multiplier value is updated to 0.001.

Iteration 6: The number of points in the resulting cloud is 13272, a considerable reduction compared to the previous case. Nevertheless, the ground plane is still not eliminated. Although this objective has not been achieved, this parametrization is established as the result of the iterative process as it is the one that results in a cloud of smaller size.

6. Package

The iterative process shows that this method of outlier removal is partially effective because, although it eliminates most of the noise without affecting the quality of the important elements, it is not able to eliminate the points belonging to the ground detection. This method will require, for this type of cloud, an additional segmentation process to eliminate the ground plane.

5.5.1.2 Radius Outlier Removal

The first step to simulate the Radius Outlier Removal method is to activate the node that launches the program associated. This is done by entering the following command in the Ubuntu terminal.

```
$ rosrun radius_removal radius_removal_node
```

With the node running, the QIP steps are now specified.

1. Characterize

The purpose of the Radius Removal method is to eliminate those points considered as noise. Tuning the parameters involved is carried out for the method to work correctly. This first section explains the expected effect on the cloud as a result of modifying them.

- setMinNeighborsInRadius(): An increase in the value of this parameter increases the minimum number of "neighbouring" points that a point must have in the vicinity to avoid being considered noise. Therefore, increasing this value will make the conditions to consider a point as noise more restrictive, increasing the filtering of the cloud.
- setRadiusSearch(): By modifying this parameter, the area where the points are considered "neighbours" is defined. Logically, a smaller size will imply more significant filtering of the cloud, i.e. greater reduction in the number of points.

2. Set Goals

The process carried out by this method is equivalent to the one performed for the Statistical Outlier Removal. Therefore, the goals are the same.

3. Choose Parameters

Iteration 1: setMinNeighborsInRadius (2), setRadiusSearch (0.8).

Iteration 2: setMinNeighborsInRadius (2), setRadiusSearch (0.4).

Iteration 3: setMinNeighborsInRadius (2), setRadiusSearch (0.2).

Iteration 4: setMinNeighborsInRadius (2), setRadiusSearch (0.1).

Iteration 5: setMinNeighborsInRadius (4), setRadiusSearch (0.1).

Iteration 6: setMinNeighborsInRadius (6), setRadiusSearch (0.1).

Iteration 7: setMinNeighborsInRadius (8), setRadiusSearch (0.1).

4. Execution

The execution is performed the same way as before except that the radius_removal_node has to be activated this time.

5. Analysis

Iteration 1: The algorithm's execution with the proposed initial values results in a cloud with 29696 points, i.e. practically equivalent to the initial cloud (29700 points). Therefore, more aggressive filtering should be performed. For this purpose, the search radius is reduced to 0.4 meters.

Iteration 2: The point reduction is almost imperceptible, with a result cloud of 29688 points, thus reducing the search radius of neighboring points to 0.2.

Iteration 3: A significant reduction in the number of points in the resultant process cloud is observed, with 20896 points. As seen in the corresponding image in the appendix, most of the points belonging to the noise generated by the process have been eliminated. The radius in the following iteration is reduced to 0.1.

Iteration 4: With the previous modification, all the noisy points created by the surroundings have been eliminated, leaving only those representing the ground. In an attempt to eliminate them, the minimum number of neighbors required is increased to 4.

Iteration 5: A reduction of about 1700 points has occurred compared to the previous iteration. However, in the resulting cloud, there are remaining points corresponding to the ground plane. Therefore, the minimum number of neighbors is increased to 6.

Iteration 6: The setMinNeighborsInRadius parameterized with a value of 6 has succeeded in eliminating the ground plane by removing all the points considered as noise. The point cloud has a dimension of 11515 points. The next iteration increases the minimum number of neighbors to 8 to achieve the best possible result.

Iteration 7: Despite the reduction of 95 points with respect to the previous parameterization, as seen in the corresponding image in the appendix, information relating to the wagon is beginning to be lost. Therefore, it was decided to stop iterating since, as stated in section 2 of the process, the noise should be eliminated without affecting the wagon.

6. Package

The radius outlier removal method has proven to be able to remove all the points considered noisy (environment and ground plane) without affecting the quality of the relevant elements (buffers and hook/chain).

For this reason, and as seen in the appendix, by applying this outlier removal method, a PC is obtained, to which it is not necessary to use a segmentation process since all the noisy points have been eliminated. The final parametrization is setMinNeighborsInRadius (6), setRadiusSearch (0.1).

In conclusion, two different methods of outlier removal have been analyzed. The Statistical method has yielded positive results, eliminating most of the noisy points. However, for this type

of cloud, it was not able to remove the ground plane. On the other hand, the Radius method was able to eliminate the ground plane. Therefore, because this second method has eliminated all the noisy points (removing the need for further pre-processing), without affecting the quality of the parts of interest, Radius is selected as the best method to perform the outlier removal. The result obtained is also applicable to the remaining models, as the only difference between them is the position of the hook/chain. Therefore, as the cloud typology and distribution are practically identic, both the method and its parameterisation will also be similar.

5.5.2 Model 1

The goodness of the method and parameterization derived from model 0 is checked. To perform this check, the radius_removal_node is launched with the final parameters, i.e. setMinNeighborsInRadius(6) and setRadiusSearch(0.1).

After the program's execution, the result is shown in the appendix. As can be deduced, the outlier removal has worked correctly with this model as it has eliminated the noisy points. Moreover, the number of points in the output cloud is 11483. This demonstrates that the Radius method and its setting work correctly.

5.5.3 Model 2

The figure representing the input cloud of model 2 shows that the chain to be recognized, due to its configuration, is joined to the sub-cloud of points that represents most of the wagon. Therefore, to facilitate the cluster extraction and recognition steps, both point clouds are separated through outlier removal. This separation will affect the geometry and the surface of the point cloud; however, due to the characteristics of the subsequent algorithms such a procedure is required.

In order to remove a larger number of points to separate the chain from the rest of the wagon, the number of neighbours must be increased and/or the search radius must be reduced. The minimum configuration to achieve the objective is a radius of 0.06 metres and a minimum number of neighbours of 9.

5.6 Downsampling

The iterative process to determine the parameterization of the downsampling algorithm is carried out in this section. In this case the downsampling is performed through the VoxelGrid method.

As in the previous cases, the node related to the program that implements the method must be activated. To do this, it is written in the Ubuntu terminal:

$ rosrun downsampling_voxel downsampling_voxel_node

It is essential that the method node for the outlier removal is active since the output of this method is the input of the downsampling.

5.6.1 Model 0

1. Characterize

The purpose of the downsampling process is to reduce the number of dots by reducing the density of the cloud. To achieve this goal the following parameter is modified:

- voxel_downsampling.setLeafSize(x f, y f, z f): represents, in meters, the size of the voxel. In this voxel, the points that compose it will be replaced by the centroid. Therefore, it is decided to create square voxels to distort the shape of the cloud as minimally as possible. An increase will mean a larger voxel size and therefore a larger point substitution.

2. Set Goals

There are two objectives with opposing behaviour, so a compromise solution must be reached in which both objectives are achieved without neglecting either.

- Reduction of the number of points: The main objective of downsampling is to reduce the number of dots by lowering the density. In this way, cloud processing is accelerated.

- Maintaining the original shape of the cloud: The shape of the elements that make up the cloud must be maintained in order to allow recognition in subsequent processes.

3. Choose parameters

Iteration 1: voxel_downsampling.setLeafSize(0.001f, 0.001f, 0.001f).

Iteration 2: voxel_downsampling.setLeafSize(0.01f, 0.01f, 0.01f).

Iteration 3: voxel_downsampling.setLeafSize(0.02f, 0.02f, 0.02f).

Iteration 4: voxel_downsampling.setLeafSize(0.04f, 0.04f, 0.04f).

4. Execution

The execution will be performed in an analogous way as described in the sections before.

5. Analysis

Iteration 1: A downsampling method execution error is reported. In detail, it is indicated that, for the study cloud, the voxel size is too small so that the integer indices would overflow. The voxel size is increased to 0.01 metres per side.

Iteration 2: The number of cloud points has been reduced to 5897, which is a reduction of 49% compared to the cloud input. On the other hand, the shapes of the components have not been affected by the downsampling. The next iteration will increase the voxel value to 0.02.

Iteration 3: The output cloud has a total of 5408 points, reduced by 500 points from the previous iteration. The shape of the elements is maintained, although a loss of resolution is beginning to show. In the next iteration the voxel value is increased to 0.04.

Iteration 4: As can be appreciated in the corresponding image, despite obtaining a cloud with only 2670 points, it is no longer possible to clearly distinguish the elements that make up the cloud. Therefore, it was decided to interrupt the process at this moment.

6. Package

After taking a closer look at the output images, it was decided to use 0.01f as the voxel size parameter. It has been chosen because it provides a considerable reduction in the number of points without affecting the shape and resolution of the cloud. This is especially relevant for the further recognition process since the recognition is based on the geometrical shape. Additional filtering is not carried out because, as can be seen in the corresponding figure, with a size of 0.02f, the resolution is already becoming lower without providing a significant reduction in the number of points.

5.6.2 Model 1

Applying the setting used for model 0, a point cloud with a total of X points is obtained, representing a considerable reduction in the number of points with respect to the cloud input to the process.

As observed in the figure, the downsampling process has not caused a considerable modification in the geometric distribution of the point cloud. Therefore, the parameterization is accurate.

5.6.3 Model 2

The parameters are the same as those used for models 0 and 1. In the figure, the application of the method, once again, has not caused a significant distortion of the geometry and surface of the point cloud studied. Therefore, the parameters are accurate for the downsampling of this model.

5.7 Cluster Extraction

The purpose of this section is to extract the models of the components to be recognized. For this purpose, Euclidean Cluster Extraction is applied.

In this case, as it is a process outside the general cloud processing pipeline, the cloud input will be accessed from a .pcd file. This file contains the cloud resulting from the pre-processing. To activate the node in this way, the following commands must be entered in the terminal.

First, set the path to the location of the input file.

$ cd ~catkin_ws/src/cluster_extraction/data

Secondly, activate the node.

$ rosrun cluster_extraction cluster_extraction

5.7.1 Model 0

1. **Characterize**

   The purpose of this algorithm is to cluster the point cloud. That is, to divide the original point cloud into subsets. To determine these subdivisions, the following parameter is mainly modified:

   - setClusterTolerance (): Determines the spatial tolerance (m) of the cluster, i.e. the distance range in which all vertices and boundaries in a shapefile or feature dataset are considered identical or coincident. As its value increases, the number of clusters reduces.

2. **Set Goals**

   In this side-process, there is only one goal:

   - Obtain the object in a cluster: The purpose of this is to be able to use the cluster later as a model for recognition. The only condition is that the object is in a single cluster without losing any relevant parts.

3. Choose parameters

Iteration 1: setClusterTolerance (0.02).

Iteration 2: setClusterTolerance (0.02).

Iteration 3: setClusterTolerance (0.04).

Iteration 4: setClusterTolerance (0.05).

Iteration 5: setClusterTolerance (0.07).

Iteration 6: setClusterTolerance (0.1).

Iteration 7: setClusterTolerance (0.15).

Iteration 8: ec.setClusterTolerance (0.18).

4. Execution

The execution will be performed in an analogous way as described in the previous methods.

The only difference is the extraction of the models. Each of the clusters created is stored in the indicated path, from where they are obtained.

5. Analysis

Iteration 1: A Euclidean Cluster Extraction algorithm execution error is displayed, informing that the input cloud is empty. The cause of this is the elimination of the planar elements, the first operation implemented in the program. Although this operation is recommended to eliminate uninteresting points, the elements and the planar geometry of the studied cloud do not allow the application of this process. This first step is eliminated and iterated with the original tolerance value for the next iteration.

Iteration 2: The algorithm works correctly now. However, only a few unrecognisable elements appear as independent clusters, being the rest of them to small to be considered (minimum size is 80 points). In order to divide it into bigger units, the tolerance value is increased to 0.04.

Iteration 3: The division of the cloud is still too detailed, i.e. objects have been divided into different clusters. Furthermore, the cluster representing the hook/chain is incomplete, as the hook is not recognized as part of the cluster. To deal with this, the tolerance value is increased to 0.05.

Iteration 4: This new iteration has solved the excess of clusters. However, the hook/chain set is incomplete as the hook is still not fully recognized. To counteract this, the tolerance is increased to 0.07.

Iteration 5: With this tolerance value, the intended elements, the two buffers and the hook/chain set have been adequately recognized. Therefore, this parameter value is considered the correct one, as the objective has been achieved. However, further iterations are performed to determine the flexibility of the parameter for this configuration, in other words, the range of values that correctly cluster the cloud.

Iteration 6: With the value set in this iteration, the clustering of the elements is still correctly visualized.

Iteration 7: With the value set in this iteration, the clustering of the elements is still correctly visualized.

Iteration 8: The value of 0.18 metres for the spatial tolerance shows that the hook/chain assembly has been associated with the same cluster as the main part of the wagon. Above this value, it is considered that the clustering no longer works correctly, as the primary target has been missed.

6. Package

From the iterations presented above, it is found that for this configuration, there is a wide range of tolerances within a correct result. Even though all the values within this range are considered valid, the lower limit is chosen as the initial value for the rest of the models. This decision is motivated by the small size of the most important element to be recognized (hook/chain). Due to this characteristic, the smallest possible value is taken to avoid, in the other configurations, its union with other elements.

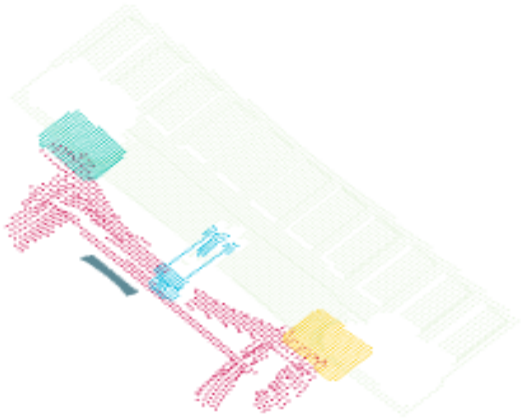


Figure 40: Clustering output for the model 0.

5.7.2 Model 1

The clustering of model 1 using the last parameterization has been successful, i.e., it has been possible to correctly isolate the chain - UIC Hook set from the rest of the cloud.

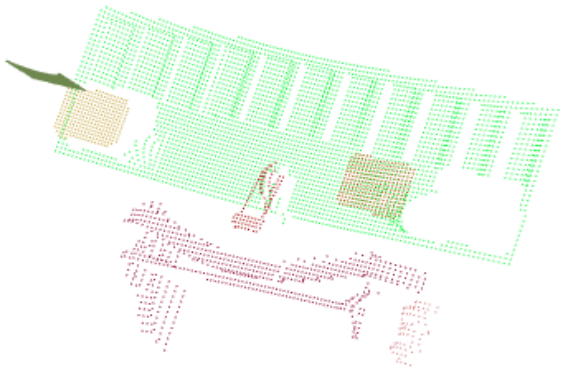


Figure 41: Clustering output for the model 1.

5.7.3 Model 2

The clustering of model 2 has also been successful using the setting proposed for the other two models.

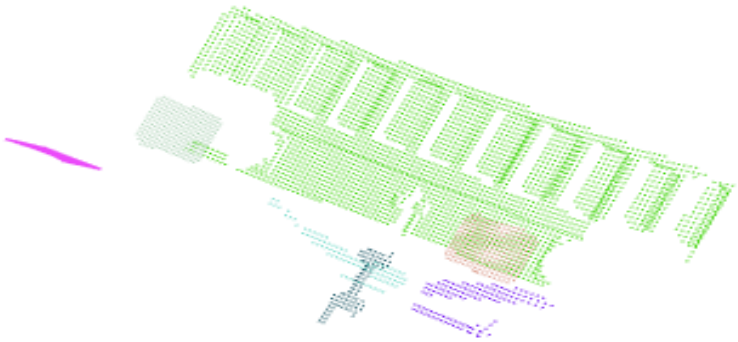


Figure 42: Clustering output for the model 2.

5.8 Registration and Recognition

The registration and recognition processes have been implemented in a single program. In addition, to facilitate testing with different models and scenes, the cloud inputs are introduced

through the terminal. These clouds must be saved in .pcd format in the path indicated in the terminal where the node is executed.

$ cd ~catkin_ws/src/correspondence_grouping/data.

$ rosrun correspondence_grouping correspondence_grouping correspondence_grouping model_filename_.pcd scene_filename_.pcd.

5.8.1 Model 0

1. Characterize

This algorithm aims to check the model's existence in the scene under study. In addition, the necessary transformation to be performed between the model and the scene object recognised as such is also obtained in the process. To adapt the algorithm to the cloud type, the following parameters are modified:

Keypoint extraction

Uniform sampling

- setRadiusSearch: Corresponds to the search radius for obtaining keypoints using the Uniform Sampling method. As it is a process of substituting points similar to downsampling, it is parameterized with the values that have already been tested, for the model, a radius of 0.01f and for the scene of 0.02f.

SIFT

- min_scale: Indicates the smallest possible scale for creating scales in the different octaves. The scale must be adaptive to the size of the point cloud; that is to say, it will be more prominent in those clouds with a larger total size.
- num_oct: number of octaves created for image processing. As the number of octaves increases, the number of keypoints and their precision increases, and the processing time increases.
- num_scales: number of scales calculated per octave. As with the number of octaves, the higher the number of scales, the higher the number of keypoints and the higher the precision, but also the longer the processing time.
- setMinimumContrast: Minimum contrast is required to consider a keypoint as valid; as it is increased, the conditions are relaxed, and the number of keypoints

obtained increases. However, it also reduces the quality of the keypoints obtained.

ISS

- setSalientRadius: Radius used for the creation of the scatter matrix, as the radius increases the number of points considered for the matrix calculation will increase.
- setNonMaxRadius: Radius for the application of the non-maximum suppression algorithm, as the radius is increased, the number of points that are considered for the calculation of the matrix will be reduced.
- setThreshold21: Upper limit to the ratio of the eigenvalues 2 and 1. As this ratio increases, more keypoints are allowed to be created by relaxing the condition to consider them representative.
- setThreshold32: Upper bound on the ratio of eigenvalues 3 and 2, following the same logic as in Threshold21.
- setMinNeighbors: Minimum number of neighbours to activate the suppression algorithm.

Feature extraction

SHOT

- setRadiusSearch: Determines the search area for the creation of the SHOT Descriptors, in other words, the area from which the information contained in these features is obtained. An insufficient radius will create features that do not have enough information to perform registration and recognition.

Hough3D recognition method

- setRadiusSearch: Determines the radius to obtain the LRF for the Hough3D method. As the radius increase, the number of points considered for the creation of the LRF will increase. This parameter should be adapted depending on the size and density of the point cloud considered.

2. Set Goals

- Model recognition: The sole purpose of this step is to perform unambiguous model recognition in the scene in case there is a matching component. Considering also the processing time and the accuracy of the keypoints obtained.

3. Choose parameters

Registration

Uniform Sampling and SHOT

Iteration 1: setRadiusSearch (0.02), uniform_model.setRadiusSearch(0.01), uniform_scene.setRadiusSearch (0.02).

Iteration 2: setRadiusSearch (0.03), uniform_model.setRadiusSearch(0.01), uniform_scene.setRadiusSearch (0.02).

Iteration 3: setRadiusSearch (0.05), uniform_model.setRadiusSearch(0.01), uniform_scene.setRadiusSearch (0.02).

Iteration 4: setRadiusSearch (0.06), uniform_model.setRadiusSearch(0.01), uniform_scene.setRadiusSearch (0.02).

Iteration 5: setRadiusSearch (0.10), uniform_model.setRadiusSearch(0.01), uniform_scene.setRadiusSearch (0.02).

Iteration 6: setRadiusSearch (0.15), uniform_model.setRadiusSearch(0.01), uniform_scene.setRadiusSearch (0.02).

SIFT and SHOT

The parameterization of the SHOT method for the processing of features will remain the same since the point cloud is the same, only the parameters of the new method for obtaining keypoints must be modified.

On the other hand, regarding the parameterization of the SIFT algorithm, a distinction must be made between the model and the scene, as the difference in size and configuration are significant.

Model

Iteration 1: min_scale (0.01), num_oct (6), num_scales (5), setMinimumContrast (0.001).

Iteration 2: min_scale (0.001), num_oct (6), num_scales (5), setMinimumContrast (0.005).

Iteration 3: min_scale (0.0005), num_oct (6), num_scales (5), setMinimumContrast (0.005).

Iteration 4: min_scale (0.0005), num_oct (12), num_scales (10), setMinimumContrast (0.005).

Iteration 5: min_scale (0.0005), num_oct (20), num_scales (18), setMinimumContrast (0.005).

Scene

Iteration 1: min_scale (0.01), num_oct (6), num_scales (5), setMinimumContrast (0.001).

Iteration 2: min_scale (0.0005), num_oct (20), num_scales (18), setMinimumContrast (0.005).

Iteration 3: min_scale (0.001), num_oct (20), num_scales (18), setMinimumContrast (0.001).

Iteration 4: min_scale (0.01), num_oct (20), num_scales (18), setMinimumContrast (0.001).

Iteration 5: min_scale (0.02), num_oct (10), num_scales (16), setMinimumContrast (0.001).

ISS and SHOT

This method, as explained in the package section, is not functional for the point clouds proposed in this project.

Recognition

Iteration 1: descr_est.setRadiusSearch (0.15), rf_est.setRadiusSearch (0.025).

Iteration 2: descr_est.setRadiusSearch (0.15), rf_est.setRadiusSearch (0.035).

Iteration 3: descr_est.setRadiusSearch (0.15), rf_est.setRadiusSearch (0.065).

Iteration 4: descr_est.setRadiusSearch (0.15), rf_est.setRadiusSearch (0.085).

4. Execution

The execution will be performed in an analogous way as described in the previous methods.

5. Analysis

Uniform Sampling and SHOT

Iteration 1: An error occurs in the creation of the features descriptors. The message: [pcl::SHOTEstimation::computeFeature] The local reference frame is not valid. This error affects all points in the model and in the scene. The most likely cause is the lack

of information for its creation. Therefore, the value of descr_est.setRadiusSearch is increased to 0.03.

Iteration 2: There have been many errors in the creation of features, however, the number has been reduced compared to the previous iteration. In addition, we have started to create valid matches between model and scene. These results validate the direction taken in the parameterization.

Iteration 3: An improvement in the feature creation process has been recorded due to a reduction in descriptor creation errors. On the other hand, a higher number of matches have been obtained, illustrating an improvement in the registration process.

Iteration 4: Improvement in the feature creation process has been noticed due to a reduction in descriptor creation errors. On the other hand, a higher number of correspondences have been obtained, illustrating an improvement in the registration process.

With the algorithm running, the values of the selected parameters are verified to obtain the keypoints. This is done in two steps:

- The keypoints obtained are valid for the creation of the features. This is proven as the number of errors is reduced and the number of matches is increased.

- The number of points is reduced proportionally for the model and the scene.

Iteration 5: The proposed parameterization of this iteration has further reduced the number of errors in the creation of features. The number of matches found has been maintained.

Iteration 6: The configuration of the parameters proposed for this iteration has on the one hand, practically eliminated the faults in the creation of features. On the other hand, the number of matches found has also increased slightly. By the value of these indicators, the point cloud registration process is up and running. The values of this iteration concerning the registration process are considered final.

SIFT and SHOT

Model

Iteration 1: The number of points obtained is insignificant. Therefore, the min_scale is increased to adapt the minimum scale to the size of the point cloud of the model. On the other hand, the minimum contrast is increased to increase the creation of keypoints.

Iteration 2: The number of keypoints has increased to 20, however the number is still insufficient. Therefore, the minimum scale size is further decreased.

Iteration 3: The number of keypoints extracted has increased to 31, which is still insufficient as the model was not detected. To increase the number, both the number of octaves and the number of scales are increased.

Iteration 4: After the last iteration the number of extracted points is increased to 74, however, the model is still not recognized.

Iteration 5: With the last parameter setting the model is recognized.

Scene

Iteration 1: The initial parameters selected extract an insufficient number of keypoints in the scene cloud. Therefore, the latest parameter settings that have been entered in the model are tested.

Iteration 2: An error has occurred due to the minimum scale size selected, as it is too small for the size of the scene cloud. This error is due to the large difference between the size of the model and the scene, therefore, the value of min_scale is increased.

Iteration 3: The proposed increase of the min_scale is not sufficient, so this value will be increased again.

Iteration 4: This setting has considerably increased the number of keypoints, however, it has not yet created a sufficient number of keypoints to obtain adequate recognition. In this iteration, in addition to modifying the min_scale parameter, the execution time will also be optimized through the reduction of num_oct and num_scales.

Iteration 5: A significant number of keypoints has been obtained, allowing the correct recognition of the model in the scene with the least possible number of operations.

Recognition

Iteration 1: No instance of the model has been detected in the scene even though the model of the hook/chain corresponds to the one present in the scene.

Iteration 2: An instance of the model has been found in the scene. With this value of rf_est.setRadiusSearch a fully functional algorithm for model recognition is already implemented. However, further iterations are performed to test the range of values for which the recognition works in this scenario.

Iteration 3: The recognition algorithm is still working correctly.

Iteration 4: The recognition algorithm is still working correctly.

6. Package

In this occasion, as there are different possible configurations of methods to obtain the keypoints and features, the results of each combination will be described separately.

As a conclusion, the most advantageous method for the application of the thesis and its reasons will be indicated.

Uniform Sampling and SHOT

As previously mentioned, this method has been successful in recognizing the different models proposed in a time close to real time. The parametrization has been particularly simple, since this method is based on the same concept as downsampling, the final parameters being 0.02 for the model and 0.03 for the scene.

In short, the method is functional, quick to execute and easy to implement. However, the quality of the keypoints obtained is not optimal, being vulnerable to positional changes in their capture.

SIFT and SHOT

This combination of methods has also been successful in recognizing the position of the UIC Hook. The simulation time estimated for the process simulation is less than one second, which is close to the real time. On the other hand, the parameterization is considerably more difficult than in the case of uniform sampling. The final setting is indicated in the last iterations.

In summary, it is a method with an execution time close to the real time and the keypoints obtained are more resilient to changes in the scene acquisition positions. However, both the parameterization and the understanding of the method are considerably more complex.

ISS and SHOT

After the testing of different parameterizations, it is concluded that this method is not functional for the considered point clouds. This is deduced from the long execution time required for the application of the method, close to 30 seconds, and the insignificant number of keypoints extracted from both the model and the scene. Consequently, the complete parameterization process for the method has not been carried out.

Therefore, it has been decided not to consider the ISS method as a feasible method for keypoint extraction in this project. Thus, the method will not be considered.

Due to the fact that with the combination of Uniform Sampling and SHOT both accuracy in the detection of the UIC-Hook states and speed have been achieved, these methods are chosen to obtain keypoints and features.

5.8.2 Model 1

The simulation with the model and the scene corresponding to scenario 1 with the combinations and configurations proposed for scenario 0 was successful. In other words, with the parameter values determined for model 0, the position of the UIC Hook has been recognised in the configuration corresponding to model 1.

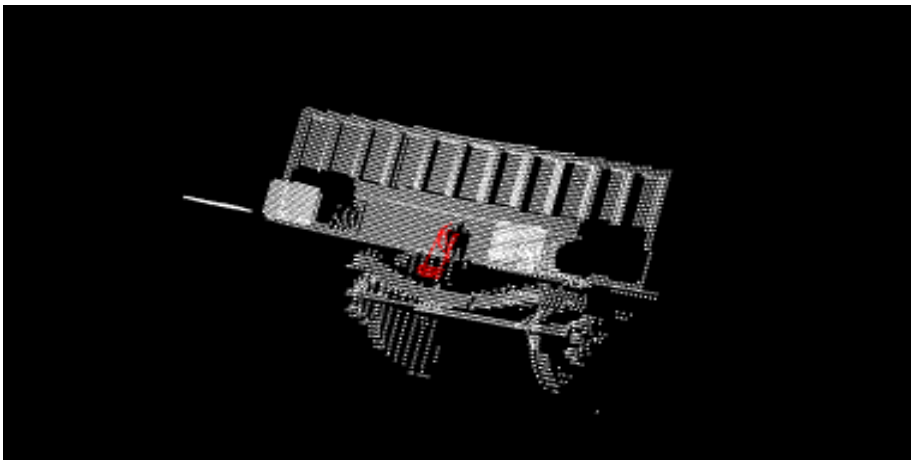


Figure 43: Result of the recognition algortihm for model 1.

5.8.3 Model 2

Similarly to model 1, after the corresponding simulations and processes, it is concluded that the combinations and parameterization of model 0 are successful and accurate for the recognition of the UIC Hook in model 2.

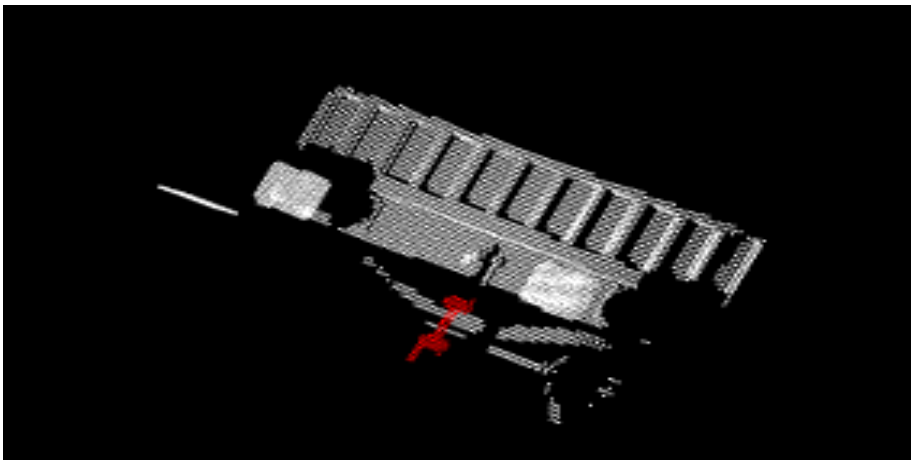


Figure 44: Result of the recognition algortihm for model 2.

Finally, to identify the position in which the UIC-Hook is positioned and, ultimately, to check whether the wagon is available for coupling, the presence of the models in the scene is checked. In other words, depending on which of the three models is identified in the scene, one coupling situation or the other will be determined. As it can be deduced, for the identification of the position it is only necessary to check the presence of two models, because if they are not present, having a limited number of cases, it means that the third model is present. On the other hand, the presence of the model is obtained with the variable clustered_corrs[i].size. If this variable has a value equal or greater than one, it means that the model is present in the scene captured by the sensor. As mentioned at the beginning of the simulation section, the case in which the wagon is not ready for coupling is the one in which the chain is inside the UIC-Hook, that is, model 1. Therefore, when this model is detected, through the variable clustered_corrs[i].size, coupling cannot be performed, being any other case valid for coupling.

6. Conclusion

This research reveals the large number of open-source methods and software available for processing and detecting elements in point clouds. The algorithm, designed from libraries available in open-source repositories, has successfully recognized the different states in which the target object, the UIC-Hook, can be found. This algorithm has been verified and validated by simulating the states through the free software Gazebo. This simulation program has been used to simulate, by means of a Digital Twin of the process, the possible configurations in which the hook can be and from which the input point clouds for the process have been obtained.

The methods that have been chosen for the processing of the point clouds are shown in Figure 45. In this figure, the procedures finally chosen have been substituted in the general scheme presented previously in the thesis.

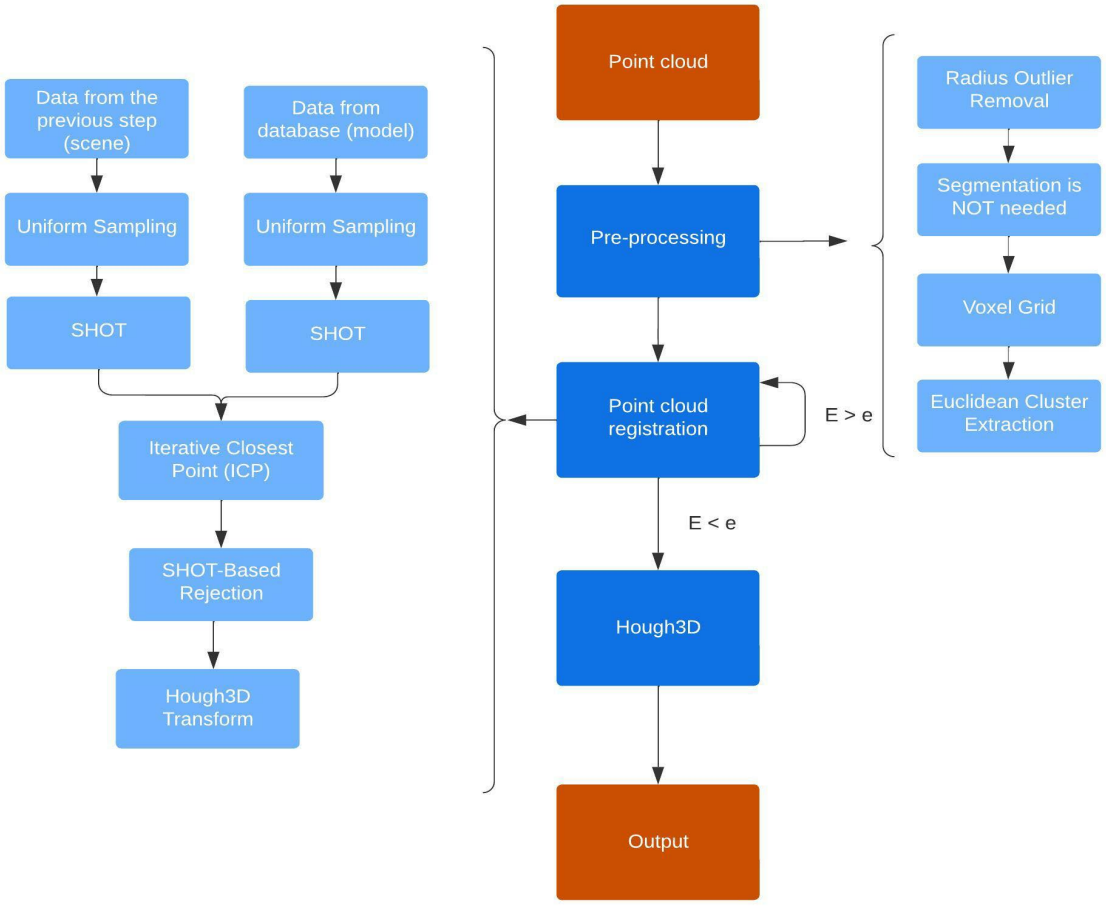


Figure 45: Methods used for point cloud processing.

On the other hand, the parameterization of the chosen methods is also very important, as the performance and the accuracy of the applied methods will depend on it. Therefore, the parameters chosen for the methods shown in figure 45 are listed below:

Radius Outlier Removal

- Models 0 and 1: setMinNeighborsInRadius(6); setRadiusSearch(0.1).
- Model 2: setMinNeighborsInRadius(9); setRadiusSearch(0.06).

Voxel Grid Downsampling

- All models: setLeafSize(0.01f, 0.01f, 0.01f).

Euclidean Cluster Extraction

- All models: setClusterTolerance (0.07).

Uniform Sampling

- Model (All models): setRadiusSearch (0.01).
- Scene: setRadiusSearch (0.02).

SHOT

- setRadiusSearch(0.15).

SHOT-Based correspondences rejection

- Value of the SHOT descriptor < 0.25f; neigh_sqr_dists[0] < 0.25f.

Hough3D Transform

- descr_est.setRadiusSearch (0.15).
- rf_est.setRadiusSearch (0.035).

Even though the algorithm is rigorously tested by simulation, it is recommended to test it also on the real process in question, as small differences in brightness or in the environment can affect the performance of the algorithm. In addition, after testing the algorithm in the real environment, the parameters may be slightly adjusted to optimize the performance of the algorithm in the real environment. On the other hand, small differences that may arise between the simulation and the real situation would, in most cases, be covered either by the flexibility of the chosen methods or by the variety of the methods. Lastly, although the algorithm has been optimised to try to run in close to real time, the complexity of the real scene can significantly affect the execution time.

Finally, it should be noted that the implementation of the algorithm in the shunting operation process in the railway environment would allow the complete automation of the process.

Bibliography

[1] 3DCollective community (2018): fotogrametria-aerea-ejemplo - 3DCollective. Available online at https://3dcollective.es/en/fotogrametria-aerea-ejemplo-2/, updated on 9/7/2022, checked on 9/7/2022.

[2] Smith, Melvyn L.; Smith, Lyndon N.; Hansen, Mark F. (2021): The quiet revolution in machine vision - a state-of-the-art survey paper, including historical review, perspectives, and future directions. In Computers in Industry 130 (1), p. 103472. DOI: 10.1016/j.compind.2021.103472.

[3] CORPORATIVA, IBERDROLA: What is artificial vision and what are its applications? (2). Available online at https://www.iberdrola.com/innovation/artificial-vision, checked on 6/10/2022.

[4] Encyclopedia Britannica (2022): automation - Advantages and disadvantages of automation, 9/7/2022. Available online at https://www.britannica.com/technology/automation/Advantages-and-disadvantages-of-automation, checked on 9/7/2022.

[5] Vollert Anlagenbau GmbH (2022): Shunting robot. Available online at https://www.vollert.de/en/product-areas/solutions-for-shunting-systems/shunting-robot, updated on 6/11/2022, checked on 6/11/2022.

[6] Railway Technology Company (2022): Rail Manteinance Vehicles and Shuntign Equipment. Available online at https://www.railway-technology.com/contractors/track/steelwheel/, checked on 6/11/2022.

[7] Zasiadko, Mykola (2020): Digital coupling reduces human exposure to risks. In RailTech.com, 7/24/2020. Available online at https://www.railtech.com/digitalisation/2020/07/24/digital-coupling-reduces-human-exposure-to-risks/, checked on 9/7/2022.

[8] Matthias Blumenschein; Daniela Wilbring; Katharina Babilon; Bernd D. Schmidt: Concept for autonomous shunting with an intelligent and self-actuating freight wagon. Available online at https://wagon40.com/Publikationen/2021_ICRT_AutonomousShunting_MB-DW-KB-BS.pdf, checked on 9/7/2022.

[9] CEN members. 2016. European standard EN 15566.

[10] FreightWaves Staff (2019): Why are railroads still important in the current era? FreightWaves. Available online at https://www.freightwaves.com/news/why-are-railroads-still-important-in-the-current-era, updated on 12/28/2019, checked on 6/11/2022.

[11] Lynn Puzo (2021): LiDAR vs Photogrammetry: Which is better for point cloud creation? -. Mosaic. Available online at https://www.mosaic51.com/technology/lidar-vs-photogrammetry-which-is-better-for-point-cloud-creation/, updated on 12/12/2021, checked on 9/7/2022.

[12] Argenis Pelayo (2021): ¿Qué son las nubes de puntos y cómo se pueden generar? - JP Global Digital. JP Global Digital. Available online at https://jpglobaldigital.com/es/2021/08/what-are-point-clouds-and-how-can-you-generate-them/#, updated on 7/13/2022, checked on 9/7/2022.

[13] Rack, Swann (2020): What is a Point Cloud? In Holocreators, 6/12/2020. Available online at https://holocreators.com/blog/what-is-a-point-cloud/, checked on 9/7/2022.
[14] Unknown (2021): Simulación en ingeniería industrial. In Industrias GSL, 8/30/2021. Available online at https://industriasgsl.com/blogs/automatizacion/simulacion-ingenieria-industrial, checked on 8/28/2022.
[15] Schluse, M. *The Digital Twin Concept*. RWTH Aachen University.
[16] Siemens Digital Industries Software (2019): Digital Twin | Siemens. Available online at https://www.plm.automation.siemens.com/global/en/our-story/glossary/digital-twin/24465, updated on 10/17/2019, checked on 9/7/2022.
[17] Pablo Gil; Carlos Mateo; Jorge Pomares; Gabriel J. Garcia; Fernando Torres (2016): Conceptos y Métodos en Visión por Computador. Available online at https://www.researchgate.net/publication/304073891_Reconocimiento_de_Objetos_3D_con_Descriptores_de_Superficie, checked on 6/11/2022.
[18] Andreas Nuechter, Udo Frese Giorgio Grisetti (2013): Large-Scale 3D Point Cloud Processing Tutorial at the 16th International Conference on Advanced Robotics, Montevideo, Uruguay. Juluis-Maximillians Universität Würzburg. Available online at http://kos.informatik.uni-osnabrueck.de/icar2013/, updated on 12/5/2013, checked on 9/7/2022.
[19] Unknown (2022): The PCL Registration API - Tutorial. Point Cloud Library. Available online at https://pcl.readthedocs.io/projects/tutorials/en/master/registration_api.html#registration-api, updated on 9/7/2022, checked on 9/7/2022.
[20] Unknown (Open3D): Point cloud outlier removal - Open3D 0.12.0 documentation. Available online at http://www.open3d.org/docs/0.12.0/tutorial/geometry/pointcloud_outlier_removal.html, updated on 1/4/2022, checked on 6/13/2022.
[21] Unknown (2022): [PCL of point cloud processing technology] filter -- outlier filter (statistical outlier removal, conditional removal and radius outlier removal). Available online at https://chowdera.com/2022/02/202202041614419220.html, updated on 2/4/2022, checked on 6/13/2022.
[22] Radu Bogdan Rusu (2022): Point Cloud Library (PCL): pcl::RadiusOutlierRemoval< PointT > Class Template Reference. PCL Point Cloud Library. Available online at https://pointclouds.org/documentation/classpcl_1_1_radius_outlier_removal.html#details, updated on 6/12/2022, checked on 6/13/2022.
[23] Unknown (2022): Removing outliers using a Conditional or RadiusOutlier removal - Point Cloud Library 0.0 documentation. Point Cloud Library. Available online at https://pcl.readthedocs.io/projects/tutorials/en/master/remove_outliers.html#remove-outliers, updated on 6/12/2022, checked on 6/13/2022.
[24] Radu Bogdan Rusu; Nico Blodow; Zoltan Marton; Alina Soos; Michael Beetz (2022): Towards 3D Object Maps for Autonomous Household Robots. Intelligent Autonomous Systems, Technische Universität München. Available online at https://pcl.readthedocs.io/projects/tutorials/en/master/statistical_outlier.html#statistical-outlier-removal, updated on 6/12/2022, checked on 6/13/2022.
[25] Radu Bogdan Rusu (2022): Point Cloud Library (PCL): pcl::RadiusOutlierRemoval< PointT > Class Template Reference. PCL Point Cloud Library. Available online at https://pointclouds.org/documentation/classpcl_1_1_radius_outlier_removal.html#details, updated on 6/12/2022, checked on 6/13/2022.

[26] Prof. Dr. Cyrill Stachniss: RANSAC. Available online at https://www.youtube.com/watch?v=9D5rrtCC_E0, checked on 9/22/2022.
[27] Unknown (2022): Plane model segmentation - Tutorial. Point Cloud Library. Available online at https://pcl.readthedocs.io/en/latest/planar_segmentation.html#planar-segmentation, updated on 9/20/2022, checked on 9/21/2022.
[28] Tomás Svoboda (2009): RANSAC. RANdom SAmple Consensus. Available online at https://cw.fel.cvut.cz/b172/_media/courses/bxaro/2009svobodaransac.pdf, checked on 9/21/2022.
[29] Zygmunt, Maria (2013): The testing of PCL: an open-source library for point cloud processing. In GLL 3, pp. 105-115. DOI: 10.15576/GLL/2013.3.105.
[30] Hunjung, Lim (2022): Downsampling a PointCloud using a VoxelGrid filter-PCL-Cpp · PCL. Available online at https://adioshun.gitbooks.io/pcl/content/Tutorial/Filtering/pcl-cpp-downsampling-a-pointcloud-using-a-voxelgrid-filter.html, updated on 6/12/2022, checked on 6/12/2022.
[31] Unknown. 2022. *How to use a KdTree to search — Tutorial*. https://pcl.readthedocs.io/projects/tutorials/en/master/kdtree_search.html, checked on 21 August 2022.
[32] Radu Bogdan Rusu (2021): Semantic 3D Object Maps for Everyday Manipulation in Human Living Enviorments. Point Cloud Library. Available online at https://pcl.readthedocs.io/projects/tutorials/en/latest/cluster_extraction.html?highlight=euclidean%20clustering, updated on 1/29/2021, checked on 8/21/2022.
[33] Alexander J. B. Trevor; Suat Gedikli; Radu Bogdan Rusu; Henrik I. Christensen: Efficient Organized Point Cloud Segmentation with Connected Components. Available online at https://cs.gmu.edu/~kosecka/ICRA2013/spme13_trevor.pdf, checked on 8/22/2022.
[34] David G. Lowe (2004): Distinctive Image Features from Scale-Invariant Keypoints. Available online at https://www.cs.ubc.ca/~lowe/papers/ijcv04.pdf, checked on 6/13/2022.
[35] Ludwig Dickmanns; Felix Dietrich; Daniel Lehmberg; Philipp Schuegraf (2021): Uniform subsampling of point cloud manifold - datafold 1.1.6 documentation. Datafold. Available online at https://datafold-dev.gitlab.io/datafold/tutorial_02_basic_pcm_subsampling.html, updated on 12/2/2021, checked on 9/12/2022.
[36] Khalid Yousif; Alireza Bab-Hadiashar; Reza Hoseinnezhad: Real-time RGB-D registration and mapping in texture-less environments using ranked order statistics. Available online at https://www.researchgate.net/publication/286549369_Real-time_RGB-D_registration_and_mapping_in_texture-less_environments_using_ranked_order_statistics, checked on 9/12/2022.
[37] Hänsch, R.; Weber, T.; Hellwich, O. (2014): Comparison of 3D interest point detectors and descriptors for point cloud fusion. In ISPRS Ann. Photogramm. Remote Sens. Spatial Inf. Sci. II-3, pp. 57-64. DOI: 10.5194/isprsannals-II-3-57-2014.
[38] Münch, Jürgen; Armbrust, Ove; Kowalczyk, Martin; Soto, Martín (Eds.) (2012): Software Process Definition and Management. Berlin, Heidelberg: Springer Berlin Heidelberg (The Fraunhofer IESE Series on Software and Systems Engineering).
[39] Prof. Dr. Edmund Weitz (2021): SIFT - Scale-Invariant Feature Transform. Available online at http://weitz.de/sift/, updated on 7/28/2021, checked on 9/12/2022.

[40] Unknown (2022): Intrinsic Shape Signatures (ISS). Open3D. Available online at http://www.open3d.org/docs/latest/tutorial/Advanced/iss_keypoint_detector.html, updated on 6/16/2022, checked on 6/16/2022.

[41] Tombari, Federico; Salti, Samuele; Di Stefano, Luigi (2013): Performance Evaluation of 3D Keypoint Detectors. In Int J Comput Vis 102 (1-3), pp. 198-220. DOI: 10.1007/s11263-012-0545-4.

[42] Hänsch, R.; Weber, T.; Hellwich, O. (2014): Comparison of 3D interest point detectors and descriptors for point cloud fusion. In ISPRS Ann. Photogramm. Remote Sens. Spatial Inf. Sci. II-3, pp. 57-64. DOI: 10.5194/isprsannals-II-3-57-2014.

[43] Samuele Salti; Federico Tombari; Luigi Di Stefano (2014): SHOT: Unique signatures of histograms for surface and texture description. In Computer Vision and Image Understanding 125, pp. 251-264. Available online at https://www.academia.edu/22339569/SHOT_Unique_signatures_of_histograms_for_surface_and_texture_description.

[44] Wan, Teng; Du, Shaoyi; Xu, Yiting; Xu, Guanglin; Li, Zuoyong; Chen, Badong; Gao, Yue (2019): RGB-D point cloud registration via infrared and color camera. In Multimed Tools Appl 78 (23), pp. 33223-33246. DOI: 10.1007/s11042-019-7159-6.

[45] Prof. Dr. Cyrill Stachniss (2022): ICP & Point Cloud Registration - Part 2: Unknown Data Association. Universität Bonn - Photogrammetry & Robotics Lab, 8/22/2022. Available online at https://www.ipb.uni-bonn.de/html/teaching/msr2-2020/sse2-03-icp.pdf, checked on 8/22/2022.

[46] Marius Muja; David G. Lowe; Mohammad Sadegh Riazi (2022): FLANN Fast Library for Approximate Nearest Neighbors. Rice University, 8/22/2022. Available online at https://slideplayer.com/slide/5383891/, checked on 8/22/2022.

[47] Stack Overflow: algorithm - Millions of 3D points: How to find the 10 of them closest to a given point? - Stack Overflow. Available online at https://stackoverflow.com/questions/2486093/millions-of-3d-points-how-to-find-the-10-of-them-closest-to-a-given-point, checked on 8/22/2022.

[48] Marius Muja; David Lowe (2009): FLANN - Fasr Library for Approximate Nearest Neighbors. Available online at https://www.fit.vutbr.cz/~ibarina/pub/VGE/reading/flann_manual-1.6.pdf, checked on 8/22/2022.

[49] Tombari, Federico; Di Stefano, Luigi (2010): Object Recognition in 3D Scenes with Occlusions and Clutter by Hough Voting. In : PSIVT 2010. Proceedings. 2010 Fourth Pacific-Rim Symposium on Image and Video Technology (PSIVT). Singapore, Singapore, 14/11/2010 - 17/11/2010. Los Alamitos, CA: IEEE, pp. 349-355. Available online at http://vision.deis.unibo.it/fede/papers/psivt10.pdf, checked on 9/13/2022.

[50] Javier Hualde (2020): Nodos, Topics y Mensajes. Turtlesim. In ROSTutorial, 8/25/2020. Available online at http://rostutorial.com/4-nodos-topics-y-mensajes-turtlesim/, checked on 7/16/2022.

[51] Javier Hualde (2020): Servicios. In ROSTutorial, 10/6/2020. Available online at http://rostutorial.com/7-servicios/, checked on 7/16/2022.

[52] Bayode Aderinola (2019): What is Rviz? The Construct. Available online at https://www.theconstructsim.com/ros-5-mins-025-rviz/, updated on 10/7/2019, checked on 9/26/2022.

[53] Wohlin, Claes; Martin Höst; Kennet Henningsson: Empirical Research Methods in Web and Software Engineering. Available online at https://www.wohlin.eu/web05.pdf, checked on 8/29/2022.

[54] Münch, Jürgen; Armbrust, Ove; Kowalczyk, Martin; Soto, Martín (2012): Process Improvement. In Jürgen Münch, Ove Armbrust, Martin Kowalczyk, Martín Soto (Eds.): Software Process Definition and Management. Berlin, Heidelberg: Springer Berlin Heidelberg (The Fraunhofer IESE Series on Software and Systems Engineering), pp. 139-176. Available online at https://link.springer.com/chapter/10.1007/978-3-642-24291-5_5

[55] Education Services Australia; Australian Mathematical Sciences Institute (2019): Normal distribution. Available online at https://amsi.org.au/ESA_Senior_Years/SeniorTopic4/4f/4f_2content_3.html, updated on 8/2/2019, checked on 9/26/2022.

[56] Radu Bogdan Rusu: Point Cloud Converter: Willow Garage Inc. Available online at https://github.com/pal-robotics-forks/point_cloud_converter/blob/hydro-devel/src/converter.cpp, checked on 7/17/2022.

[57] Unknown (2022): Point cloud - Basic usage Tutorial. Open3D. Available online at http://www.open3d.org/docs/latest/tutorial/geometry/pointcloud.html, updated on 9/12/2022, checked on 9/12/2022.

List of figures

List of tables

Appendix

This section introduces the reader to the different programs used to implement the different functionalities required by the project. The programs are classified according to the functionality that they implement.

Extraction of the .pcd file

- Converter.cpp: Used to transform the PointCloud message into a PointCloud2 message [31].

```
1: /* Software License Agreement (BSD License)
2: *
3: * Copyright (c) 2009, Willow Garage, Inc.
4: * All rights reserved.
5: *
6: * Redistribution and use in source and binary forms, with or without
7: * modification, are permitted provided that the following conditions
8: * are met:
9: *
10: * * Redistributions of source code must retain the above copyright
11: * notice, this list of conditions and the following disclaimer.
12: * * Redistributions in binary form must reproduce the above
13: * copyright notice, this list of conditions and the following
14: * disclaimer in the documentation and/or other materials provided
15: * with the distribution.
16: * * Neither the name of Willow Garage, Inc. nor the names of its
17: * contributors may be used to endorse or promote products derived
18: * from this software without specific prior written permission.
19: *
20: * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
21: * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
22: * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
23: * FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
24: * COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
25: * INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
26: * BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
27: * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
28: * CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
29: * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
30: * ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
31: * POSSIBILITY OF SUCH DAMAGE.
32: *
33: * $Id: converter.cpp 33249 2010-03-11 02:09:24Z rusu $
34: *
35: */
36:
37: #include <ros/ros.h>
38: #include <pcl/point_cloud.h>
39: #include <pcl_ros/point_cloud.h>
40: #include <pcl_conversions/pcl_conversions.h>
41: #include <sensor_msgs/PointCloud2.h>
42: #include <sensor_msgs/PointCloud.h>
43: #include <sensor_msgs/point_cloud_conversion.h>
44:
45: using namespace std;
46:
47: class PointCloudConverter {
48:
49:
50: private:
51:
52: //boost::mutex m_mutex_; NOT USEFUL FOR THE APPLICATION IN THE PROJECT
53: ros::NodeHandle nh_;
54: ros::Subscriber sub_points_, sub_points2_;
55: ros::Publisher pub_points_, pub_points2_;
```

```
56:
57: int queue_size_;
58: string points_in_, points2_in_, points_out_, points2_out_;
59:

60: public:

61: PointCloudConverter () : nh_ ("~"), queue_size_ (100),
62: points_in_ ("/points_in"), points2_in_ ("/points2_in"),
63: points_out_ ("/points_out"), points2_out_ ("/points2_out")
64: {
65: // Subscribe to the cloud topic using both the old message format and the new
66: sub_points_ = nh_.subscribe (points_in_, queue_size_, &PointCloudConverter::cloud_cb_points
67: sub_points2_ = nh_.subscribe (points2_in_, queue_size_, &PointCloudConverter::cloud_cb_points2
68: pub_points_ = nh_.advertise<sensor_msgs::PointCloud> (points_out_, queue_size_);
69: pub_points2_ = nh_.advertise<sensor_msgs::PointCloud2> (points2_out_, queue_size_);
70: ROS_INFO ("PointCloudConverter initialized to transform from PointCloud (%s) to PointCloud2 71:
ROS_INFO ("PointCloudConverter initialized to transform from PointCloud2 (%s) to PointCloud 72: }
73:
74:
75: inline std::string
76: getFieldsList (const sensor_msgs::PointCloud2 &cloud)
77: {
78: std::string result;
79: for (size_t i = 0; i < cloud.fields.size () - 1; ++i)
80: result += cloud.fields[i].name + " ";
81: result += cloud.fields[cloud.fields.size () - 1].name;
82: return (result);
83: }
84:
85:
86: inline std::string
87: getFieldsList (const sensor_msgs::PointCloud &points)
88: {
89: std::string result = "x y z";
90: for (size_t i = 0; i < points.channels.size (); i++)
91: result = result + " " + points.channels[i].name;
92: return (result);
93: }
94:
95:
96: void
97: cloud_cb_points2 (const sensor_msgs::PointCloud2ConstPtr &msg)
98: {
99: if (pub_points_.getNumSubscribers () <= 0)
100: {
101: //ROS_DEBUG ("[point_cloud_converter] Got a PointCloud2 with %d points on %s, but no su
102: return;
103: }
104:
105: ROS_DEBUG ("[point_cloud_converter] Got a PointCloud2 with %d points (%s) on %s.", msg->width
106:
107: sensor_msgs::PointCloud output;
108: // Convert to the new PointCloud format
109: if (!sensor_msgs::convertPointCloud2ToPointCloud (*msg, output))
110: {
111: ROS_ERROR ("[point_cloud_converter] Conversion from sensor_msgs::PointCloud2 to sensor_112:
return;
113: }
114: ROS_DEBUG ("[point_cloud_converter] Publishing a PointCloud with %d points on %s.", (int)
115: pub_points_.publish (output);
116: }
117:
118:
119: void

120: cloud_cb_points (const sensor_msgs::PointCloudConstPtr &msg)

121: {
122: if (pub_points2_.getNumSubscribers () <= 0)
123: {
124: //ROS_DEBUG ("[point_cloud_converter] Got a PointCloud with %d points on %s, but no sub
```

```
125: return;
126: }
127:
128: ROS_DEBUG ("[point_cloud_converter] Got a PointCloud with %d points (%s) on %s.", (int)msg
129:
130: sensor_msgs::PointCloud2 output;
131: // Convert to the old point cloud format
132: if (!sensor_msgs::convertPointCloudToPointCloud2 (*msg, output))
133: {
134: ROS_ERROR ("[point_cloud_converter] Conversion from sensor_msgs::PointCloud to sensor_msgs::135:
return;
136: }
137: ROS_DEBUG ("[point_cloud_converter] Publishing a PointCloud2 with %d points on %s.", output
138: pub_points2_.publish (output);
139: }
140:
141: };
142:
143:
144: int
145: main (int argc, char** argv)
146: {
147: // ROS init
148: ros::init (argc, argv, "point_cloud_converter", ros::init_options::AnonymousName);
149:
150: PointCloudConverter p;
151: ros::spin ();
152:
153: return (0);

154: }
```

Read_pcd_node.cpp

```
1: #include <ros/ros.h>
2: #include <pcl/point_cloud.h>
3: #include <pcl_conversions/pcl_conversions.h>
4: #include <sensor_msgs/PointCloud2.h>
5: #include <pcl/io/pcd_io.h>
6:
7: main (int argc, char **argv) {
8:
9: //Initialization of the node
10: ros::init (argc, argv, "read_pcd_node");
11:
12: ros::NodeHandle nh;
13:
14: //Creation and parametrization of the publisher
15: ros::Publisher pub = nh.advertise <sensor_msgs::PointCloud2> ("output", 1);
16: //Creation of the PointCloud2 message
17: sensor_msgs::PointCloud2 output;
18:
19: //Creation of an PointCloud object
20: pcl::PointCloud<pcl::PointXYZ> point_cloud;
21:
22: //Function for loading the .pcd file into a PointCloud object
23: pcl::io::loadPCDFile ("milk_cartoon_all_small_clorox.pcd", point_cloud);
24: //Conversion into a RosMsg to be able to publish it
25: pcl::toROSMsg (point_cloud, output);
26:
27: //Output frame, for Rviz visualization of the node to check that the information stored
28: output.header.frame_id = "odom";
29:
30: ros::Rate loop_rate(1);
31:
32: while (ros::ok()) {
33:
34: //Publishing of the information stored in the file
35: pub.publish(output);
36:
37: ros::spinOnce();
```

```
38: loop_rate.sleep();
39: }
40:
41: return 0;
42:
43: }
```

Write_pcd_node.cpp

```
1: #include <ros/ros.h>
2: #include <pcl/point_cloud.h>
3: #include <pcl_conversions/pcl_conversions.h>
4: #include <sensor_msgs/PointCloud2.h>
5: #include <pcl/io/pcd_io.h>
6:
7: void cloudCB(const sensor_msgs::PointCloud2 &input)
8: {
9:
10: //Creation of the PointCloud obejct
11: pcl::PointCloud<pcl::PointXYZ> cloud;
12: pcl::fromROSMsg(input, cloud);
13:
14: //Command for storing the point cloud in a pcd file
15: pcl::io::savePCDFileASCII ("PC_outlier_radius.pcd", cloud);
16: }
17:
18: main (int argc, char **argv)
19: {
20:
21: //Activation of the node
22: ros::init (argc, argv, "write_pcd_node");
23:
24: //Creation of node handler
25: ros::NodeHandle nh;
26:
27: //Creation of the subscriber
28: ros::Subscriber sub;
29:
30: //Subscription to the topic that is stored
31: sub = nh.subscribe("PC_radius_removal", 10, cloudCB);
32: ros::spin();
33:
34: return 0;
35: }
```

Outlier removal

- Statitical outlier removal: outlier_removal.cpp.

```
1: #include <ros/ros.h>
2: #include <pcl/point_cloud.h>
3: #include <pcl_conversions/pcl_conversions.h>
4: #include <sensor_msgs/PointCloud2.h>
5: #include <pcl/filters/statistical_outlier_removal.h>
6:
7: class cloudHandler {
8:
9: public:
10:
11: //Creation of the node, Subscriber and Publisher
12: ros::NodeHandle nh;
13: ros::Subscriber sub;
14: ros::Publisher pub;
15:
16: cloudHandler() {
17: //Parametrization of a subscriber node to obtain the initial cloud
18: sub = nh.subscribe("output", 10, &cloudHandler::cloud_filtering, this);
19: //Parametrization of the publisher for publishing the output of the Statistical Outlier
```

```
20: pub = nh.advertise<sensor_msgs::PointCloud2> ("PC_outlier_removal_statistical", 1);
21: }
22:
23: void cloud_filtering (const sensor_msgs::PointCloud2& input) {
24: //Creation of the PointCloud objects for the initial cloud and the filtered one
25: pcl::PointCloud<pcl::PointXYZ> initial_cloud;
26: pcl::PointCloud<pcl::PointXYZ> outrem_statistical_cloud;
27:
28: //Creation of the topic output for publishing the output
29: sensor_msgs::PointCloud2 output_outrem_statistical;
30:
31: pcl::fromROSMsg (input, initial_cloud);
32:
33: //Creation of the StatisticalOutlierRemoval object
34: pcl::StatisticalOutlierRemoval<pcl::PointXYZ> statitistical_Filter;
35:
36: //Parametrization of the StatisticalOutlierRemoval object
37: statitistical_Filter.setInputCloud(initial_cloud.makeShared());
38: statitistical_Filter.setMeanK(50);
39: statitistical_Filter.setStddevMulThresh(0.01);
40: statitistical_Filter.filter(outrem_statistical_cloud);
41:
42: //Transformation of the PointCloud again to a RosMsg
43: pcl::toROSMsg(outrem_statistical_cloud, output_outrem_statistical);
44: pub.publish(output_outrem_statistical);
45:
46: }
47: };
48:
49:
50: main (int argc, char** argv) {
51:
52: //Initialization of the node
53: ros::init(argc, argv, "outlier_removal_node");
54:
55: // Calling to the class -- Application of the method
56: cloudHandler handler;
57: ros::spin();
58:
59: return 0;

60: }
```

- Radius_removal_node.cpp

```
1: #include <ros/ros.h>
2: #include <pcl/point_cloud.h>
3: #include <pcl_conversions/pcl_conversions.h>
4: #include <sensor_msgs/PointCloud2.h>
5: #include <pcl/filters/radius_outlier_removal.h>
6:
7: class cloudHandler {
8:
9: public:
10:
11: //Creation of the node handler, subscriber and publisher
12: ros::NodeHandle nh;
13: ros::Subscriber sub;
14: ros::Publisher pub;
15:
16: cloudHandler() {
17: //Parametrization of the Subscriber for obtaining the initial PointCloud
18: sub = nh.subscribe("output", 10, &cloudHandler::cloud_filtering, this);
19: //Parametrization of the Publisher
20: pub = nh.advertise<sensor_msgs::PointCloud2> ("PC_radius_removal", 1);
21: }
22:
23: void cloud_filtering (const sensor_msgs::PointCloud2& input) {
24: //Creation of the PointCloud objects for the initial cloud and the filtered
25: pcl::PointCloud<pcl::PointXYZ> initial_cloud;
26: pcl::PointCloud<pcl::PointXYZ> radius_removal_cloud;
```

```
27:
28: //Creation of the topic output for the expression of the PC
29: sensor_msgs::PointCloud2 output_radius_removal;
30:
31: pcl::fromROSMsg (input, initial_cloud);
32:
33: //Creation of the RadiusOutlierRemoval object
34: pcl::RadiusOutlierRemoval<pcl::PointXYZ> radius_Filter;
35:
36: //Parametrization of the Radius removal filter
37: radius_Filter.setInputCloud(initial_cloud.makeShared());
38: radius_Filter.setRadiusSearch(0.07);
39: radius_Filter.setMinNeighborsInRadius (8);
40: radius_Filter.setKeepOrganized(false);
41:
42: //Application of the filter
43: radius_Filter.filter (radius_removal_cloud);
44:
45: pcl::toROSMsg(radius_removal_cloud,output_radius_removal);
46: pub.publish(output_radius_removal);
47:
48: }
49:
50: };
51:
52: main(int argc, char** argv) {
53:
54: //Initialization of the node
55: ros::init(argc, argv, "radius_removal_node");
56:
57: cloudHandler handler;
58: ros::spin();
59:

60: return 0;

}
```

Downsampling

- Downsampling_voxel_node.cpp

```
1: #include <ros/ros.h>
2: #include <pcl/point_cloud.h>
3: #include <pcl_conversions/pcl_conversions.h>
4: #include <sensor_msgs/PointCloud2.h>
5: #include <pcl/filters/voxel_grid.h>
6: #include <iostream>
7:
8: class cloudHandler {
9:
10: public:
11: //Creadtion of node handler, subscriber and publisher
12: ros::NodeHandle nh;
13: ros::Subscriber sub;
14: ros::Publisher pub;
15:
16: cloudHandler(){
17: //Parametrization of the subscriber for obtaining the output of the outlier_removal
18: sub = nh.subscribe("PC_radius_removal",10, &cloudHandler::cloudCB, this);
19: //Parametrization of the subscriber for output communication
20: pub = nh.advertise<sensor_msgs::PointCloud2> ("PC_downsampled_voxel",1);
21: }
22:
23: void cloudCB (const sensor_msgs::PointCloud2& input){
24:
25: //Creation of the initial and output clouds
26: pcl::PointCloud<pcl::PointXYZ> outlier_cloud;
27: pcl::PointCloud<pcl::PointXYZ> downsampled_cloud;
```

```
28:
29: sensor_msgs::PointCloud2 output_downsampling_voxel;
30:
31: //Conversion from ROSmsg to PointCloud to apply the method
32: pcl::fromROSMsg(input, outlier_cloud);
33:
34: //Creation of the VoxelGrid object
35: pcl::VoxelGrid<pcl::PointXYZ> voxel_downsampling;
36:
37: //Parametrization of the VoxelGrid object
38: voxel_downsampling.setInputCloud (outlier_cloud.makeShared());
39: voxel_downsampling.setLeafSize(0.01f, 0.01f, 0.01f);
40: voxel_downsampling.filter (downsampled_cloud);
41:
42: pcl::toROSMsg (downsampled_cloud, output_downsampling_voxel);
43: pub.publish (output_downsampling_voxel);
44: }
45: };
46:
47: main (int argc, char **argv) {
48:
49: //Initialization of the node
50: ros::init(argc, argv, "downsampling_voxel_node");
51:
52: //Class calling
53: cloudHandler handler;
54: ros::spin();
55:
56: return 0;

57: }
```

Cluster extraction

- Cluster_extraction.cpp

```
1: #include <pcl/ModelCoefficients.h>
2: #include <pcl/point_types.h>
3: #include <pcl/io/pcd_io.h>
4: #include <pcl/filters/extract_indices.h>
5: #include <pcl/filters/voxel_grid.h>
6: #include <pcl/features/normal_3d.h>
7: #include <pcl/search/kdtree.h>
8: #include <pcl/sample_consensus/method_types.h>
9: #include <pcl/sample_consensus/model_types.h>
10: #include <pcl/segmentation/sac_segmentation.h>
11: #include <pcl/segmentation/extract_clusters.h>
12:
13:
14: int main () {
15:
16: //Creation of a PCDReader object
17: pcl::PCDReader reader;
18: //Creation of the initial cloud
19: pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_initial (new pcl::PointCloud<pcl::PointXYZ>), cloud_20:
21: //Reading information from file
22: reader.read ("radius_outlier_pcd.pcd", *cloud_initial);
23: //Feedback of the number of points the initial cloud
24: std::cout << "Initial PointCloud has: " << cloud->size () << " points." << std::endl;
25:
26: //Creation of the VoxelGrid object
27: pcl::VoxelGrid<pcl::PointXYZ> voxel_downsampling;
28:
29: //Downsampling output cloud
30: pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_filtered (new pcl::PointCloud<pcl::PointXYZ>);
31:
32: //Parametrization and output of VoxelGrid Object
33: voxel_downsampling.setInputCloud (cloud_initial);
34: voxel_downsampling.setLeafSize (0.01f, 0.01f, 0.01f);
35: voxel_downsampling.filter (*cloud_filtered);
```

```
36: //Feedback number of points in the PC after downsampling
37: std::cout << "PointCloud after filtering has: " << cloud_filtered->size () << " data points."
38:
39: //Creation of a PCD writer object for storing the cloud clusters
40: pcl::PCDWriter writer;
41:
42: /* NOT USED IN THE TYPE OF THE POINT CLOUD IN THIS PROJECT - NEVERTHELESS IMPORTANT FOR OTHER
TYPOLOGIES OF POINT CLOUD
43:
44: // Create the segmentation object
45: pcl::SACSegmentation<pcl::PointXYZ> segmentation;
46: //Calculation of the points indices in the Point Cloud
47: pcl::PointIndices::Ptr inliers (new pcl::PointIndices);
48: //Calculation of coefficients of PC
49: pcl::ModelCoefficients::Ptr coefficients (new pcl::ModelCoefficients);
50: //Segmentation Output PC
51: pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_plane (new pcl::PointCloud<pcl::PointXYZ> (
52:
53: //Parametrization of segmentation object
54: segmentation.setOptimizeCoefficients (true);
55: segmentation.setModelType (pcl::SACMODEL_PLANE);
56: segmentation.setMethodType (pcl::SAC_RANSAC);
57: segmentation.setMaxIterations (100);
58: segmentation.setDistanceThreshold (0.02);
59: // Obtaining the number of points

60: int nr_points = (int) cloud_filtered->size ();


61:
62: //Iterate while the filtered cloud size is greater then the number of points of the
63: while (cloud_filtered->size () > 1 * nr_points) {
64: // Segment the largest planar component from the remaining cloud
65: segmentation.setInputCloud (cloud_filtered);
66: segmentation.segment (*inliers, *coefficients);
67:
68: //Check if the indices are obtained correctly
69: if (inliers->indices.size () == 0) {
70: //Feedback Error in the segmentation
71: std::cout << "Error in the segmentation, the indices vector is empty." << std::e
72: break;
73: }
74:
75: // Extract the planar inliers from the input cloud
76: pcl::ExtractIndices<pcl::PointXYZ> extraction;
77: //Parametrization of the extraction object
78: extraction.setInputCloud (cloud_filtered);
79: extraction.setIndices (inliers);
80: extraction.setNegative (false);
81:
82: // Get the points associated with the planar surface
83: extraction.filter (*cloud_plane);
84:
85: std::cout << "Number of points in the planar element: " << cloud_plane->size () <<
86:
87: // Remove the planar inliers and extraction of the rest
88: extract.setNegative (true); // Elimitation of the
89: //Extraction of the not planar elements
90: extract.filter (*cloud_f);
91: *cloud_filtered = *cloud_f;
92: }*/
93:
94: // Creation of KdTree as search methodology
95: pcl::search::KdTree<pcl::PointXYZ>::Ptr KdTree (new pcl::search::KdTree<pcl::PointXYZ>);
96: //Setting input for KdTree
97: KdTree->setInputCloud (cloud_filtered);
98:
99: //Creation of array for storing (in each entry separately) the indices of each cluster
100: std::vector<pcl::PointIndices> cluster_indices;
101: //Creation ofEuclideanClusterExtraction object
102: pcl::EuclideanClusterExtraction<pcl::PointXYZ> euclidean_extraction;
103:
104: //Parametrization of cluster tolerance
```

```
105: euclidean_extraction.setClusterTolerance (0.07); //meters
106:
107: //Parametrization of the EuclideanClusterExtraction object
108: euclidean_extraction.setMinClusterSize (80);
109: euclidean_extraction.setMaxClusterSize (10000);
110: euclidean_extraction.setSearchMethod (KdTree);
111: euclidean_extraction.setInputCloud (cloud_filtered);
112:
113: //Storing the indices of the extracted clusters in the vector created
114: euclidean_extraction.extract (cluster_indices);
115:
116: //Loop for storing each cluster in a separate point cloud
117: int j = 0;
118: for (std::vector<pcl::PointIndices>::const_iterator it = cluster_indices.begin (); it !=
cluster_119:

120: //Creation of PointCloud object for cluster points storage

121: pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_cluster (new pcl::PointCloud<pcl::PointXYZ>);
122:
123: //Storing of all points of the cluster in a PointCloud object
124: for (const auto& idx : it->indices)
125: cloud_cluster->push_back ((*cloud_filtered)[idx]);
126: cloud_cluster->width = cloud_cluster->size ();
127: cloud_cluster->height = 1;
128: cloud_cluster->is_dense = true;
129:
130: //Feedback of number of points per cluster
131: std::cout << "Size of the Cluster: " << cloud_cluster->size () << "points." << std::endl;
132:
133: //Creation of the name of the cluster of the current interation for pcd file creation
134: std::stringstream ss;
135: ss << "cloud_cluster_" << j << ".pcd";
136:
137: //Storing of the point in the appropiate cluster
138: writer.write<pcl::PointXYZ> (ss.str (), *cloud_cluster, false);
139: j++;
140: }
141:
142: return (0);

143: }
```

Registration and Recognition

- Correspondence_grouping.cpp

```
1: #include <pcl/io/pcd_io.h>
2: #include <pcl/point_cloud.h>
3: #include <pcl/correspondence.h>
4: #include <pcl/features/normal_3d_omp.h>
5: #include <pcl/features/shot_omp.h>
6: #include <pcl/features/board.h>
7: #include <pcl/filters/uniform_sampling.h>
8: #include <pcl/recognition/cg/hough_3d.h>
9: #include <pcl/recognition/cg/geometric_consistency.h>
10: #include <pcl/visualization/pcl_visualizer.h>
11: #include <pcl/kdtree/kdtree_flann.h>
12: #include <pcl/kdtree/impl/kdtree_flann.hpp>
13: #include <pcl/common/transforms.h>
14: #include <pcl/console/parse.h>
15: #include <pcl/keypoints/narf_keypoint.h>
16: #include <pcl/keypoints/sift_keypoint.h>
17: #include <pcl/keypoints/iss_3d.h>
18:
19: //typedef declaration of objects to save time in the declarations in the program
```

```
20: typedef pcl::PointXYZ PointType;
21: typedef pcl::Normal NormalType;
22: typedef pcl::ReferenceFrame RFType;
23: typedef pcl::SHOT352 DescriptorType;
24:
25: //Model filename name, for charging it later from the terminal
26: std::string model_filename_;
27: //Scene filename name, for charging it later from the terminal
28: std::string scene_filename_;
29:
30: // Definition of parameters for the algorithms implicated in the process
31: bool show_keypoints_ (false);
32: bool show_correspondences_ (false);
33: bool use_cloud_resolution_ (false);
34: bool use_hough_ (true);
35:
36: // Parameters Uniform Sampling Keypoint extraction
37:
38: float model_ss_ (0.02f); //keypoint sampling dimensions model
39: float scene_ss_ (0.03f); //keypoint sampling dimensions scene
40:
41: //Parameters SIFT Keypoint extraction
42:
43: //Model
44: float min_scale_mod (0.01f); //Standard deviation of the smallest scale
45: int number_oct_mod (10); //Number of groups in the Gaussian pyramid
46: int number_scales_octave_mod (16); //Number of scales per group
47: float min_contrast_mod (0.00001f); //Threshold for Keypoint detection
48:
49: //Scene
50: float min_scale_sce (0.01f); //Standard deviation of the smallest scale
51: int number_oct_sce (18); //Number of groups in the Gaussian pyramid
52: int number_scales_octave_sce (24); //Number of scales per group
53: float min_contrast_sce (0.00001f); //Threshold for Keypoint detection
54:
55: // Parameters ISS Keypoint extraction
56: double cloud_resolution_mod (0.0226285);
57: float set_Threshold21_mod (0.975f);
58: float set_Threshold32_mod (0.975f);
59: int min_neigh_mod (2);

60: int num_threads_mod (1);


61:
62: double cloud_resolution_sce (0.0126568);
63: float set_Threshold21_sce (0.975f);
64: float set_Threshold32_sce (0.975f);
65: int min_neigh_sce (2);
66: int num_threads_sce (1);
67:
68:
69: //Recognition and Registration parameters
70: float rf_rad_ (0.035f); // Hough3D algorithm
71: float descr_rad_ (0.15f); //features SHOT extraction
72: float cg_size_ (0.01f); //Size of the Hough3D subdivisions
73: float cg_thresh_ (5.0f); //In the researcher document the
74:
75:
76: //void function for files obtention from the terminal
77:
78: void parseCommandLine (int argc, char *argv[]) {
79:
80: //Model & scene filenames, vector for taking the names from the terminal -- More flexib
81: std::vector<int> docs;
82:
83: //Termination (.pcd) required for the files of the input
84: docs = pcl::console::parse_file_extension_argument (argc, argv, ".pcd");
85:
86: //If you don't enter 2 files show error and quit the function
87: if (docs.size () != 2) {
88: std::cout << "Filenames missing.\n";
89: exit (-1);
```

```
90: }
91:
92: //Storing of the filenames intp variables
93: model_filename_ = argv[docs[0]];
94: scene_filename_ = argv[docs[1]];
95:
96: }
97:
98: //For creation of SIFT Keypoints
99:
100: //The SIFT keypoints need to have a value of the intensity, in this case the depth will
101: namespace pcl {
102: template<>
103: struct SIFTKeypointFieldSelector<PointXYZ>
104: {
105: inline float
106: operator () (const PointXYZ &p) const
107: {
108: return p.x;
109: }
110: };
111: }
112:
113:
114: //Starting point for the registration // Recognition
115:
116: int main (int argc, char *argv[]) {
117:
118: parseCommandLine (argc, argv);
119:

120: //Declaration of variables needed for the algorithms below

121:
122: pcl::PointCloud<PointType>::Ptr model (new pcl::PointCloud<PointType> ());
123: pcl::PointCloud<PointType>::Ptr model_keypoints (new pcl::PointCloud<PointType> ());
124: pcl::PointCloud<PointType>::Ptr scene (new pcl::PointCloud<PointType> ());
125: pcl::PointCloud<PointType>::Ptr scene_keypoints (new pcl::PointCloud<PointType> ());
126: pcl::PointCloud<NormalType>::Ptr model_normals (new pcl::PointCloud<NormalType> ());
127: pcl::PointCloud<NormalType>::Ptr scene_normals (new pcl::PointCloud<NormalType> ());
128: pcl::PointCloud<DescriptorType>::Ptr model_descriptors (new pcl::PointCloud<DescriptorType> (
129: pcl::PointCloud<DescriptorType>::Ptr scene_descriptors (new pcl::PointCloud<DescriptorType> (
130:
131: //PC load from files obtained in the first function -- The load is done in the conditio
132:
133: if (pcl::io::loadPCDFile (model_filename_, *model) < 0)
134: {
135: std::cout << "Error loading model cloud." << std::endl;
136: return (-1);
137: }
138:
139: if (pcl::io::loadPCDFile (scene_filename_, *scene) < 0)
140: {
141: std::cout << "Error loading scene cloud." << std::endl;
142: return (-1);
143: }
144:
145: //Computation of the normals for each of the points
146:
147: //Creation of the Normal estimation object
148: pcl::NormalEstimationOMP<PointType, NormalType> normal_est;
149:
150: normal_est.setKSearch (20); //Number of points used for the Kd-Tree search
151:
152: //Model
153: normal_est.setInputCloud (model);
154: normal_est.compute (*model_normals);
155:
156: //Scene
157: normal_est.setInputCloud (scene);
158: normal_est.compute (*scene_normals);
159:
```

```
160:
161: //KEYPOINTS EXTRACTION
162:
163: /* UNIFORM KEYPOINT EXTRACTION
164:
165: //Creation of the UniformSampling object
166: pcl::UniformSampling<PointType> keypoints_uniform;
167:
168: //paramerization of the UniformSampling object
169: keypoints_uniform.setInputCloud (model);
170: keypoints_uniform.setRadiusSearch (model_ss_);
171: keypoints_uniform.filter (*model_keypoints);
172:
173: //Feedback number of keypoints extracted respect the initial point cloud
174: std::cout << "Model total points: " << model->size () << "; Selected Keypoints: " <<
175:
176: //Keypoints extraction for the scene
177: keypoints_uniform.setInputCloud (scene);
178: keypoints_uniform.setRadiusSearch (scene_ss_);
179: keypoints_uniform.filter (*scene_keypoints);

180:


181: std::cout << "Scene total points: " << scene->size () << "; Selected Keypoints: " <<
182:
183: // SIFT KEYPOINT EXTRACTION
184: /*
185: //Creation of the bool use_hough_ (tr uSeI)F;TKeypoint object
186: pcl::SIFTKeypoint <PointType, pcl::PointWithScale> keypoints_SIFT;
187: //Creation of the search object KdTree
188: pcl::search::KdTree <PointType>::Ptr tree_SIFT (new pcl::search::KdTree<PointType>())
189:
190: //The SIFT output requires a PointWithScale PointCloud
191: pcl::PointCloud<pcl::PointWithScale> result_model;
192:
193: //Parametrization of the SIFT object
194: keypoints_SIFT.setInputCloud (model);
195: keypoints_SIFT.setSearchMethod (tree_SIFT);
196: keypoints_SIFT.setScales (min_scale_mod, number_oct_mod, number_scales_octave_mod);
197: keypoints_SIFT.setMinimumContrast (min_contrast_mod);
198: keypoints_SIFT.compute (result_model);
199:
200: //Conversion of the PointWithScale to PointXYZ pointcloud
201: copyPointCloud(result_model, *model_keypoints);
202:
203: //Feedback number of keypoints extracted respect the initial point cloud
204: std::cout << "Model total points: " << model->size () << "; Selected Keypoints: " <<
205:
206: //SIFT keypoint extraction for the scene
207: pcl::PointCloud<pcl::PointWithScale> result_scene;
208: keypoints_SIFT.setInputCloud (scene);
209: keypoints_SIFT.setSearchMethod (tree_SIFT);
210: keypoints_SIFT.setScales (min_scale_sce, number_oct_sce, number_scales_octave_sce);
211: keypoints_SIFT.setMinimumContrast (min_contrast_sce);
212: keypoints_SIFT.compute (result_scene);
213: copyPointCloud(result_scene, *scene_keypoints);
214: std::cout << "Model total points: " << scene->size () << "; Selected Keypoints: " <<
215:
216: // ISS KEYPOINT EXTRACTION
217:
218: //Creation of the ISSKeypoint3D objet
219: pcl::ISSKeypoint3D <PointType, PointType> keypoints_ISS;
220: //Creation of the search object KdTree
221: pcl::search::KdTree <PointType>::Ptr tree_ISS (new pcl::search::KdTree<PointType>());
222:
223: //Parametrization of the ISSKeypoint3D
224: keypoints_ISS.setSearchMethod (tree_ISS);
225: keypoints_ISS.setSalientRadius (1 * cloud_resolution_mod);
226: keypoints_ISS.setNonMaxRadius (1 * cloud_resolution_mod);
227: keypoints_ISS.setThreshold21 (set_Threshold21_mod);
228: keypoints_ISS.setThreshold32 (set_Threshold32_mod);
229: keypoints_ISS.setMinNeighbors (min_neigh_mod);
```

```
230: keypoints_ISS.setNumberOfThreads (num_threads_mod);
231: keypoints_ISS.setInputCloud (model);
232: keypoints_ISS.compute (*model_keypoints);
233:
234: //Feedback number of keypoints extracted respect the initial point cloud
235: std::cout << "Model total points: " << model->size () << "; Selected Keypoints: " <<
model_keypoints
236:
237: //ISS keypoint extraction for the scene
238: keypoints_ISS.setSearchMethod (tree_ISS);
239: keypoints_ISS.setSalientRadius (1 * cloud_resolution_sce);

240: keypoints_ISS.setNonMaxRadius (2 * cloud_resolution_sce);

241: keypoints_ISS.setThreshold21 (set_Threshold21_sce);
242: keypoints_ISS.setThreshold32 (set_Threshold32_sce);
243: keypoints_ISS.setMinNeighbors (min_neigh_sce);
244: keypoints_ISS.setNumberOfThreads (num_threads_sce);
245: keypoints_ISS.setInputCloud (scene);
246: keypoints_ISS.compute (*scene_keypoints);
247:
248: std::cout << "Model total points: " << scene->size () << "; Selected Keypoints: " <<
scene_keypoints
249:
250: //Features extraction
251:
252: //The method used is the SHOTEstimator, based in the OMP, the same type of the normal e
253: pcl::SHOTEstimationOMP<PointType, NormalType, DescriptorType> SHOT_descr;
254:
255: //Set of parameter -- Remember that all the parameters are defined at the begining
256: SHOT_descr.setRadiusSearch (descr_rad_);
257:
258: //Feature extraction for the model
259:
260: //Parametrization of SHOTEstimationOMP object
261: SHOT_descr.setInputCloud (model_keypoints);
262: SHOT_descr.setInputNormals (model_normals);
263: SHOT_descr.setSearchSurface (model);
264: SHOT_descr.compute (*model_descriptors);
265:
266: //Feature extraction for the scene
267: SHOT_descr.setInputCloud (scene_keypoints);
268: SHOT_descr.setInputNormals (scene_normals);
269: SHOT_descr.setSearchSurface (scene);
270: SHOT_descr.compute (*scene_descriptors);
271:
272:
273: //Registration -- Obtention of correspondences
274:
275: //Correspondences Object Creation
276: pcl::CorrespondencesPtr correspondence_array (new pcl::Correspondences ());
277:
278: //Creation of the Kd-Tree for NN search
279: pcl::KdTreeFLANN<DescriptorType> NN_search;
280:
281: NN_search.setInputCloud (model_descriptors);
282:
283: // For each scene keypoint descriptor, find nearest neighbor into the model keypoints
284: for (std::size_t i = 0; i < scene_descriptors->size (); ++i)
285: {
286: std::vector<int> neigh_indices (1); //Array for neighbours indices
287: std::vector<float> neigh_sqr_dists (1); //Array for the distances between neighbours
288: if (!std::isfinite (scene_descriptors->at (i).descriptor[0])) //skipping NaNs
289: {
290: continue;
291: }
292: //Search neighbours with the KdTreeFLANN object
293: int found_neighs = NN_search.nearestKSearch (scene_descriptors->at (i), 1, neigh_indices,
neigh_294: if(found_neighs == 1 && neigh_sqr_dists[0] < 0.25f) //Correspondences rejection based in the
295: {
296: //If its applicable store of the indice in the correspondence vector
297: pcl::Correspondence corr (neigh_indices[0], static_cast<int> (i), neigh_sqr_dists[0]);
```

```
298: correspondence_array->push_back (corr);
299: }

300: }

301: // Information about the number of correspondences
302: std::cout << "Correspondences found: " << correspondence_array->size () << std::endl;
303:
304:
305: //Actual clustering
306: //Creation of the object for storing the Homogeneous Transform matrix
307: std::vector<Eigen::Matrix4f, Eigen::aligned_allocator<Eigen::Matrix4f> > Homogeneous_transform
308: std::vector<pcl::Correspondences> clustered_corrs;
309:
310: //Conditional dependand of the method selected
311: if (use_hough_) {
312: // Compute (Keypoints) Reference Frames
313: //Creation of the point clouds for LRF (Local Reference Frame)
314: pcl::PointCloud<RFType>::Ptr model_rf (new pcl::PointCloud<RFType> ());
315: pcl::PointCloud<RFType>::Ptr scene_rf (new pcl::PointCloud<RFType> ());
316:
317: //Creation of BOARDLocalReferenceFrame for LRF of keypoints estimation
318: pcl::BOARDLocalReferenceFrameEstimation<PointType, NormalType, RFType> LRF_est;
319: LRF_est.setFindHoles (true);
320: LRF_est.setRadiusSearch (rf_rad_);
321:
322: //Model object parametrization
323: LRF_est.setInputCloud (model_keypoints);
324: LRF_est.setInputNormals (model_normals);
325: LRF_est.setSearchSurface (model);
326: LRF_est.compute (*model_rf);
327:
328: //Scene object parametrization
329: LRF_est.setInputCloud (scene_keypoints);
330: LRF_est.setInputNormals (scene_normals);
331: LRF_est.setSearchSurface (scene);
332: LRF_est.compute (*scene_rf);
333:
334: //Hough3D method application
335: //Creation of Hough3DGrouping object
336: pcl::Hough3DGrouping<PointType, PointType, RFType, RFType> Hough_recognition;
337:
338: //General parametrization
339: Hough_recognition.setHoughBinSize (cg_size_);
340: Hough_recognition.setHoughThreshold (cg_thresh_);
341: Hough_recognition.setUseInterpolation (true);
342: Hough_recognition.setUseDistanceWeight (false);
343:
344: //Model parametrization
345: Hough_recognition.setInputCloud (model_keypoints);
346: Hough_recognition.setInputRf (model_rf);
347:
348: //Scene parametrization
349: Hough_recognition.setSceneCloud (scene_keypoints);
350: Hough_recognition.setSceneRf (scene_rf);
351:
352: //Give correspondences obtained before
353: Hough_recognition.setModelSceneCorrespondences (correspondence_array);
354:
355: //recognition step
356: Hough_recognition.recognize (Homogeneous_transform, clustered_corrs);
357: }
358: /* METHOD B IN CASE THE FIRST ONE DOES NOT WORK
359: else // Using GeometricConsistency -- aplicacion del otro algoritmo que se tiene

360: {'

361: pcl::GeometricConsistencyGrouping<PointType, PointType> gc_clusterer;
362: gc_clusterer.setGCSize (cg_size_);
363: gc_clusterer.setGCThreshold (cg_thresh_);
364:
365: gc_clusterer.setInputCloud (model_keypoints);
```

```
366: gc_clusterer.setSceneCloud (scene_keypoints);
367: gc_clusterer.setModelSceneCorrespondences (model_scene_corrs);
368:
369: //gc_clusterer.cluster (clustered_corrs);
370: gc_clusterer.recognize (rototranslations, clustered_corrs);
371: }*/
372:
373:
374: // Output results
375:
376: std::cout << "Model instances found: " << Homogeneous_transform.size () << std::endl;
377: for (std::size_t i = 0; i < Homogeneous_transform.size (); ++i)
378: {
379: std::cout << "\n Instance " << i + 1 << ":" << std::endl;
380: std::cout << " Correspondences belonging to this instance: " << clustered_corrs[i].size
381:
382: // Print the rotation matrix and translation vector
383: Eigen::Matrix3f rotation = Homogeneous_transform[i].block<3,3>(0, 0);
384: Eigen::Vector3f translation = Homogeneous_transform[i].block<3,1>(0, 3);
385:
386: printf ("\n");
387: printf (" | %6.3f %6.3f %6.3f | \n", rotation (0,0), rotation (0,1), rotation (0
388: printf (" R = | %6.3f %6.3f %6.3f | \n", rotation (1,0), rotation (1,1), rotation (1,2);
389: printf (" | %6.3f %6.3f %6.3f | \n", rotation (2,0), rotation (2,1), rotation (2,2);
390: printf ("\n");
391: printf (" t = < %0.3f, %0.3f, %0.3f >\n", translation (0), translation (1), translation (2);
392: bool use_hough_ (true);
393:
394: // Visualization of results
395: pcl::visualization::PCLVisualizer viewer ("Correspondence Grouping");
396: viewer.addPointCloud (scene, "scene_cloud");
397:
398: pcl::PointCloud<PointType>::Ptr off_scene_model (new pcl::PointCloud<PointType> ());
399: pcl::PointCloud<PointType>::Ptr off_scene_model_keypoints (new pcl::PointCloud<PointType> ())
400:
401:
402: while (!viewer.wasStopped ())
403: {
404: viewer.spinOnce ();
405: }
406:
407: return (0);

408: }
```

Point Cloud Resolution

- Cloud_resolution.cpp

```
1: #include <pcl/io/pcd_io.h>
2: #include <pcl/point_cloud.h>
3: #include <pcl/console/parse.h>
4: #include <pcl/search/kdtree.h>
5:
6: int main () {
7:
8: //Initialization of variables
9: double res = 0.0;
10: int n_points = 0;
11: int nres;
12: std::vector<int> indices (2);
13: std::vector<float> sqr_distances (2);
14:
15: //Creation of the PointCloud object
16: pcl::PointCloud<pcl::PointXYZ>::Ptr cloud (new pcl::PointCloud<pcl::PointXYZ>);
17: //Creation of the PCDReader object
18: pcl::PCDReader reader;
```

```
19:
20: //Read from file and store in the desired cloud
21: reader.read ("scene_filename_2.pcd", *cloud);
22: //Creation of KdTree for search
23: pcl::search::KdTree<pcl::PointXYZ> tree;
24: tree.setInputCloud (cloud);
25:
26: //Iteration trough point cloud
27: for (std::size_t i = 0; i < cloud->size (); ++i){
28: if (! std::isfinite ((*cloud)[i].x))
29: {
30: continue;
31: }
32: //Considering the second neighbor since the first is the point itself for resolution es
33: nres = tree.nearestKSearch (i, 2, indices, sqr_distances);
34: if (nres == 2)
35: {
36: res += sqrt (sqr_distances[1]);
37: ++n_points;
38: }
39: }
40: if (n_points != 0)
41: {
42: res /= n_points;
43: }
44:
45: //Feedback trough screen indicating the point cloud resolution in meters
46: std::cout << "Model resolution: " << res<< std::endl;
47:
48: return 0;

49: }
```

Final program

```
1: #include <pcl/io/pcd_io.h>
2: #include <ros/ros.h>
3: #include <pcl/point_cloud.h>
4: #include <pcl_conversions/pcl_conversions.h>
5: #include <sensor_msgs/PointCloud2.h>
6: #include <pcl/correspondence.h>
7: #include <pcl/features/normal_3d_omp.h>
8: #include <pcl/features/shot_omp.h>
9: #include <pcl/features/board.h>
10: #include <pcl/filters/uniform_sampling.h>
11: #include <pcl/recognition/cg/hough_3d.h>
12: #include <pcl/recognition/cg/geometric_consistency.h>
13: #include <pcl/visualization/pcl_visualizer.h>
14: #include <pcl/kdtree/kdtree_flann.h>
15: #include <pcl/kdtree/impl/kdtree_flann.hpp>
16: #include <pcl/common/transforms.h>
17: #include <pcl/console/parse.h>
18: #include <pcl/keypoints/sift_keypoint.h>
19: #include <pcl/keypoints/iss_3d.h>
20: #include <pcl/filters/radius_outlier_removal.h>
21: #include <pcl/filters/voxel_grid.h>
22:
23: //typedef declaration of objects to save time in the declarations in the program
24: typedef pcl::PointXYZ PointType;
25: typedef pcl::Normal NormalType;
26: typedef pcl::ReferenceFrame RFType;
27: typedef pcl::SHOT352 DescriptorType;
28:
29: //Model filename name, for charging it later from the terminal
30: std::string model_filename_;
31: //Scene filename name, for charging it later from the terminal
32: std::string scene_filename_;
33:
34: // Definition of parameters for the algorithms implicated in the process
35: bool show_keypoints_ (false);
36: bool show_correspondences_ (false);
37: bool use_cloud_resolution_ (false);
```

```
38: bool use_hough_ (true);
39:
40: // Parameters Uniform Sampling Keypoint extraction
41:
42: float model_ss_ (0.02f); //keypoint sampling dimensions model
43: float scene_ss_ (0.03f); //keypoint sampling dimensions scene
44:
45: //Parameters SIFT Keypoint extraction
46:
47: //Model
48: float min_scale_mod (0.01f); //Standard deviation of the smallest scale
49: int number_oct_mod (10); //Number of groups in the Gaussian pyramid
50: int number_scales_octave_mod (16); //Number of scales per group
51: float min_contrast_mod (0.00001f); //Threshold for Keypoint detection
52:
53: //Scene
54: float min_scale_sce (0.01f); //Standard deviation of the smallest scale
55: int number_oct_sce (18); //Number of groups in the Gaussian pyramid
56: int number_scales_octave_sce (24); //Number of scales per group
57: float min_contrast_sce (0.00001f); //Threshold for Keypoint detection
58:
59: // Parameters ISS Keypoint extraction
60: double cloud_resolution_mod (0.0226285);
61: float set_Threshold21_mod (0.975f);
62: float set_Threshold32_mod (0.975f);
63: int min_neigh_mod (2);
64: int num_threads_mod (1);
65:

66: double cloud_resolution_sce (0.0126568);

67: float set_Threshold21_sce (0.975f);
68: float set_Threshold32_sce (0.975f);
69: int min_neigh_sce (2);
70: int num_threads_sce (1);
71:
72:
73: //Recognition and Registration parameters
74: float rf_rad_ (0.035f); // Hough3D algorithm
75: float descr_rad_ (0.15f); //features SHOT extraction
76: float cg_size_ (0.01f); //Size of the Hough3D subdivisions
77: float cg_thresh_ (5.0f); //In the researcher document the
78:
79:
80: //void function for files obtention from the terminal
81:
82: void parseCommandLine (int argc, char *argv[]) {
83:
84: //Model & scene filenames, vector for taking the names from the terminal -- More flexibility for
85: std::vector<int> docs;
86:
87: //Termination (.pcd) required for the files of the input
88: docs = pcl::console::parse_file_extension_argument (argc, argv, ".pcd");
89:
90: //If you don enter 2 files show error and quit the function
91: if (docs.size () != 2) {
92: std::cout << "Filenames missing.\n";
93: exit (-1);
94: }
95:
96: //Storing of the filenames intp variables
97: model_filename_ = argv[docs[0]];
98: scene_filename_ = argv[docs[1]];
99:
100: }
101:
102: //For creation of SIFT Keypoints
103:
104: //The SIFT keypoints need to have a value of the intensity, in this case the depth will be used a
105: namespace pcl {
106: template<>
107: struct SIFTKeypointFieldSelector<PointXYZ>
108: {
109: inline float
110: operator () (const PointXYZ &p) const
111: {
112: return p.x;
113: }
114: };
115: }
116:
117:
```

```
118: //Starting point for the registration // Recognition
119:
120: int main (int argc, char *argv[]) {
121:
122: parseCommandLine (argc, argv);
123:
124: //Declaration of variables needed for the algorithms below
125:
126: pcl::PointCloud<PointType>::Ptr model (new pcl::PointCloud<PointType> ());
127: pcl::PointCloud<PointType>::Ptr model_keypoints (new pcl::PointCloud<PointType> ());
128: pcl::PointCloud<PointType>::Ptr scene (new pcl::PointCloud<PointType> ());
129: pcl::PointCloud<PointType>::Ptr scene_prepro (new pcl::PointCloud<PointType> ());
130: pcl::PointCloud<PointType>::Ptr scene_keypoints (new pcl::PointCloud<PointType> ());
131: pcl::PointCloud<NormalType>::Ptr model_normals (new pcl::PointCloud<NormalType> ());

132: pcl::PointCloud<NormalType>::Ptr scene_normals (new pcl::PointCloud<NormalType> ());

133: pcl::PointCloud<DescriptorType>::Ptr model_descriptors (new pcl::PointCloud<DescriptorType> ());
134: pcl::PointCloud<DescriptorType>::Ptr scene_descriptors (new pcl::PointCloud<DescriptorType> ());
135:
136: //PC load from files obtained in the first function -- The load is done in the condition field of
137:
138: if (pcl::io::loadPCDFile (model_filename_, *model) < 0)
139: {
140: std::cout << "Error loading model cloud." << std::endl;
141: return (-1);
142: }
143:
144: //Final program, lecture from the initial cloud topic
145: //sub = nh.subscribe("initial_PC2", 10, *scene);
146:
147: if (pcl::io::loadPCDFile (scene_typedef pcl::Normal NormalType; filename_, *scene)
148: {
149: std::cout << "Error loading scene cloud." << std::endl;
150: return (-1);
151: }
152:
153: //Outlier removal
154:
155: pcl::RadiusOutlierRemoval<pcl::PointXYZ> radius_Filter;
156:
157: radius_Filter.setInputCloud(scene);
158: radius_Filter.setRadiusSearch(0.1);
159: radius_Filter.setMinNeighborsInRadius (6);
160: radius_Filter.setKeepOrganized(false);
161:
162: radius_Filter.filter (*scene_prepro);
163:
164: //Downsampling - Use only in the case where uniform_sampling is not the method for keypoint extra
165:
166: pcl::VoxelGrid<pcl::PointXYZ> voxel_downsampling;
167:
168: voxel_downsampling.setInputCloud (scene_prepro);
169: voxel_downsampling.setLeafSize(0.01f, 0.01f, 0.01f);
170: voxel_downsampling.filter (*scene_prepro);
171:
172:
173:
174: //Computation of the normals for each of the points
175:
176: //Creation of the Normal estimation object
177: pcl::NormalEstimationOMP<PointType, NormalType> normal_est;
178:
179: normal_est.setKSearch (20); //Number of points used for the Kd-Tree search
180:
181: //Model
182: normal_est.setInputCloud (model);
183: normal_est.compute (*model_normals);
184:
185: //Scene
186: normal_est.setInputCloud (scene_prepro);
187: normal_est.compute (*scene_normals);
188:
189:
190: //Keypoints extraction
191:
192: //Creacion del objeto para hacer el keypoints extraction
193: pcl::UniformSampling<PointType> keypoints_uniform;
194:
195: //Keypoints extraction for the model
196:
197: //UNIFORM SAMPLING
```

```
198: keypoints_uniform.setInputCloud (model);

199: keypoints_uniform.setRadiusSearch (model_ss_);
200: keypoints_uniform.filter (*model_keypoints);
201: std::cout << "Model total points: " << model->size () << "; Selected Keypoints: " << model_keypoints->size
202:
203: //Keypoints extraction for the scene
204: keypoints_uniform.setInputCloud (scene_prepro);
205: keypoints_uniform.setRadiusSearch (scene_ss_);
206: keypoints_uniform.filter (*scene_keypoints);
207: std::cout << "Scene total points: " << scene->size () << "; Selected Keypoints: " << scene_keypoints->size
208:
209: // SIFT KEYPOINT EXTRACTION
210: /*
211: pcl::SIFTKeypoint <PointType, pcl::PointWithScale> keypoints_SIFT;
212: pcl::search::KdTree <PointType>::Ptr tree_SIFT (new pcl::search::KdTree<PointType>());
213:
214: pcl::PointCloud<pcl::PointWithScale> result_model;
215: keypoints_SIFT.setInputCloud (model);
216: keypoints_SIFT.setSearchMethod (tree_SIFT);
217: keypoints_SIFT.setScales (min_scale_mod, number_oct_mod, number_scales_octave_mod);
218: keypoints_SIFT.setMinimumContrast (min_contrast_mod);
219: keypoints_SIFT.compute (result_model);
220: copyPointCloud(result_model, *model_keypoints);
221:
222: std::cout << "Model total points: " << model->size () << "; Selected Keypoints: " << model_keyp
223:
224: pcl::PointCloud<pcl::PointWithScale> result_scene;
225: keypoints_SIFT.setInputCloud (scene);
226: keypoints_SIFT.setSearchMethod (tree_SIFT);
227: keypoints_SIFT.setScales (min_scale_sce, number_oct_sce, number_scales_octave_sce);
228: keypoints_SIFT.setMinimumContrast (min_contrast_sce);
229: keypoints_SIFT.compute (result_scene);
230: copyPointCloud(result_scene, *scene_keypoints);
231: std::cout << "Model total points: " << scene->size () << "; Selected Keypoints: " << scene_keyp
232:
233: // ISS KEYPOINT EXTRACTION
234:
235: /*pcl::ISSKeypoint3D <PointType, PointType> keypoints_ISS;
236: pcl::search::KdTree <PointType>::Ptr tree_ISS (new pcl::search::KdTree<PointType>());
237:
238: keypoints_ISS.setSearchMethod (tree_ISS);
239: keypoints_ISS.setSalientRadius (1 * cloud_resolution_mod);
240: keypoints_ISS.setNonMaxRadius (1 * cloud_resolution_mod);
241: keypoints_ISS.setThreshold21 (set_Threshold21_mod);
242: keypoints_ISS.setThreshold32 (set_Threshold32_mod);
243: keypoints_ISS.setMinNeighbors (min_neigh_mod);
244: keypoints_ISS.setNumberOfThreads (num_threads_mod);
245: keypoints_ISS.setInputCloud (model);
246: keypoints_ISS.compute (*model_keypoints);
247: std::cout << "Model total points: " << model->size () << "; Selected Keypoints: " << model_keyp
248:
249: keypoints_ISS.setSearchMethod (tree_ISS);
250: keypoints_ISS.setSalientRadius (1 * cloud_resolution_sce);
251: keypoints_ISS.setNonMaxRadius (2 * cloud_resolution_sce);
252: keypoints_ISS.setThreshold21 (set_Threshold21_sce);
253: keypoints_ISS.setThreshold32 (set_Threshold32_sce);
254: keypoints_ISS.setMinNeighbors (min_neigh_sce);
255: keypoints_ISS.setNumberOfThreads (num_threads_sce);
256: keypoints_ISS.setInputCloud (scene);
257: keypoints_ISS.compute (*scene_keypoints);
258: std::cout << "Model total points: " << scene->size () << "; Selected Keypoints: " << scene_keyp
259:
260: //Features extraction
261:
262: //The method used is the SHOTEstimator, based in the OMP, the same type of the normal estimator
263: pcl::SHOTEstimationOMP<PointType, NormalType, DescriptorType> SHOT_descr;

264:

265: //Set of parameter -- Remember that all the parameters are defined at the begining
266: SHOT_descr.setRadiusSearch (descr_rad_);
267:
268: //Feature extraction for the model
269: SHOT_descr.setInputCloud (model_keypoints);
270: SHOT_descr.setInputNormals (model_normals);
271: SHOT_descr.setSearchSurface (model);
272: SHOT_descr.compute (*model_descriptors);
273:
274: //Feature extraction for the scene
275: SHOT_descr.setInputCloud (scene_keypoints);
276: SHOT_descr.setInputNormals (scene_normals);
```

```
277: SHOT_descr.setSearchSurface (scene_prepro);
278: SHOT_descr.compute (*scene_descriptors);
279:
280:
281: //Registration -- Obtention of correspondences
282:
283: //Correspondences Object Creation
284: pcl::CorrespondencesPtr correspondence_array (new pcl::Correspondences ());
285:
286: //Creation of the Kd-Tree for NN search
287: pcl::KdTreeFLANN<DescriptorType> NN_search;
288:
289: NN_search.setInputCloud (model_descriptors);
290:
291: // For each scene keypoint descriptor, find nearest neighbor into the model keypoints descriptor
292:
293: for (std::size_t i = 0; i < scene_descriptors->size (); ++i)
294: {
295: std::vector<int> neigh_indices (1);
296: std::vector<float> neigh_sqr_dists (1);
297: if (!std::isfinite (scene_descriptors->at (i).descriptor[0])) //skipping NaNs
298: {
299: continue;
300: }
301: int found_neighs = NN_search.nearestKSearch (scene_descriptors->at (i), 1, neigh_indices, neigh_sqr_dists
302: if(found_neighs == 1 && neigh_sqr_dists[0] < 0.25f) //Correspondences rejection based in the SHOT descriptors
303: {
304: pcl::Correspondence corr (neigh_indices[0], static_cast<int> (i), neigh_sqr_dists[0]);
305: correspondence_array->push_back (corr);
306: }
307: }
308: // Information about the number of correspondences -- IMP for iteration of algorithms
309: std::cout << "Correspondences found: " << correspondence_array->size () << std::endl;
310:
311:
312: //Actual clustering
313: std::vector<Eigen::Matrix4f, Eigen::aligned_allocator<Eigen::Matrix4f> > rototranslations;
314: std::vector<pcl::Correspondences> clustered_corrs;
315:
316:
317: if (use_hough_)
318: {
319: //
320: // Compute (Keypoints) Reference Frames only for Hough
321: pcl::PointCloud<RFType>::Ptr model_rf (new pcl::PointCloud<RFType> ());
322: pcl::PointCloud<RFType>::Ptr scene_rf (new pcl::PointCloud<RFType> ());
323:
324: pcl::BOARDLocalReferenceFrameEstimation<PointType, NormalType, RFType> rf_est;
325: rf_est.setFindHoles (true);
326: rf_est.setRadiusSearch (rf_rad_);
327:
328: rf_est.setInputCloud (model_keypoints);
329: rf_est.setInputNormals (model_normals);

330: rf_est.setSearchSurface (model);

331: rf_est.compute (*model_rf);
332:
333: rf_est.setInputCloud (scene_keypoints);
334: rf_est.setInputNormals (scene_normals);
335: rf_est.setSearchSurface (scene_prepro);
336: rf_est.compute (*scene_rf);
337:
338: // Clustering
339: pcl::Hough3DGrouping<PointType, PointType, RFType, RFType> clusterer;
340: clusterer.setHoughBinSize (cg_size_);
341: clusterer.setHoughThreshold (cg_thresh_);
342: clusterer.setUseInterpolation (true);
343: clusterer.setUseDistanceWeight (false);
344:
345: clusterer.setInputCloud (model_keypoints);
346: clusterer.setInputRf (model_rf);
347: clusterer.setSceneCloud (scene_keypoints);
348: clusterer.setSceneRf (scene_rf);
349: clusterer.setModelSceneCorrespondences (correspondence_array);
350:
351: //clusterer.cluster (clustered_corrs);
352: clusterer.recognize (rototranslations, clustered_corrs);
353: }
354:
355: // Output results
356:
```

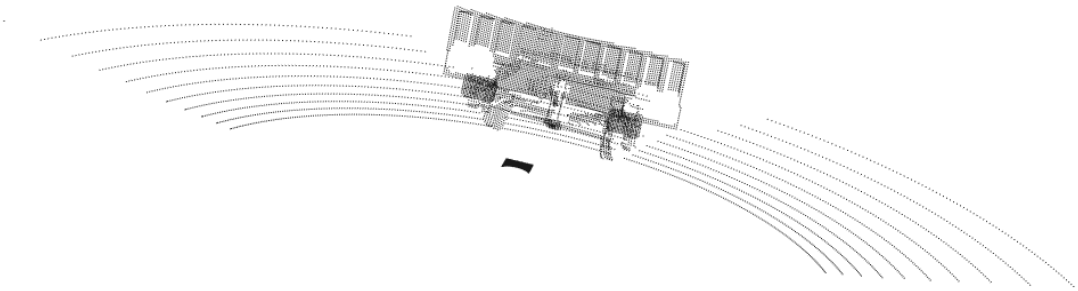
```
357: std::cout << "Model instances found: " << rototranslations.size () << std::endl;
358: for (std::size_t i = 0; i < rototranslations.size (); ++i)
359: {
360: std::cout << "\n Instance " << i + 1 << ":" << std::endl;
361: std::cout << " Correspondences belonging to this instance: " << clustered_corrs[i].size () << std
362:
363: // Print the rotation matrix and translation vector
364: Eigen::Matrix3f rotation = rototranslations[i].block<3,3>(0, 0);
365: Eigen::Vector3f translation = rototranslations[i].block<3,1>(0, 3);
366:
367: printf ("\n");
368: printf (" | %6.3f %6.3f %6.3f | \n", rotation (0,0), rotation (0,1), rotation (0,2));
369: printf (" R = | %6.3f %6.3f %6.3f | \n", rotation (1,0), rotation (1,1), rotation (1,2));
370: printf (" | %6.3f %6.3f %6.3f | \n", rotation (2,0), rotation (2,1), rotation (2,2));
371: printf ("\n");
372: printf (" t = < %0.3f, %0.3f, %0.3f >\n", translation (0), translation (1), translation (2));
373: }
374:
375: // Visualization
376: //
377: pcl::visualization::PCLVisualizer viewer ("Correspondence Grouping");
378: viewer.addPointCloud (scene, "scene_cloud");
379:
380: pcl::PointCloud<PointType>::Ptr off_scene_model (new pcl::PointCloud<PointType> ());
381: pcl::PointCloud<PointType>::Ptr off_scene_model_keypoints (new pcl::PointCloud<PointType> ());
382:
383:
384: for (std::size_t i = 0; i < rototranslations.size (); ++i)
385: {
386: pcl::PointCloud<PointType>::Ptr rotated_model (new pcl::PointCloud<PointType> ());
387: pcl::transformPointCloud (*model, *rotated_model, rototranslations[i]);
388:
389: std::stringstream ss_cloud;
390: ss_cloud << "instance" << i;
391:
392: pcl::visualization::PointCloudColorHandlerCustom<PointType> rotated_model_color_handler (rotated_model,
393: viewer.addPointCloud (rotated_model, rotated_model_color_handler, ss_cloud.str ());
394: }
395:

396: while (!viewer.wasStopped ())

397: {
398: viewer.spinOnce ();
399: }
400:
401: return (0);

402: }
```

Image Appendix

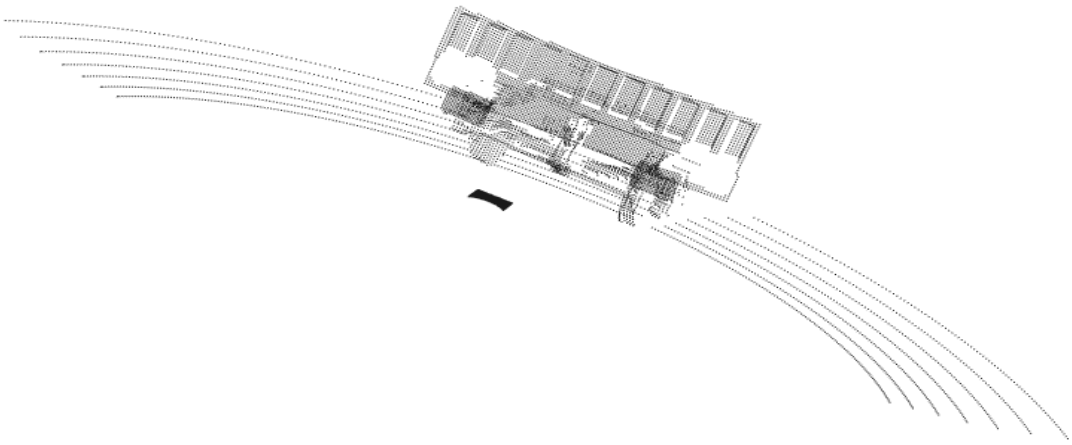
Index

1. Statistical Outlier Removal

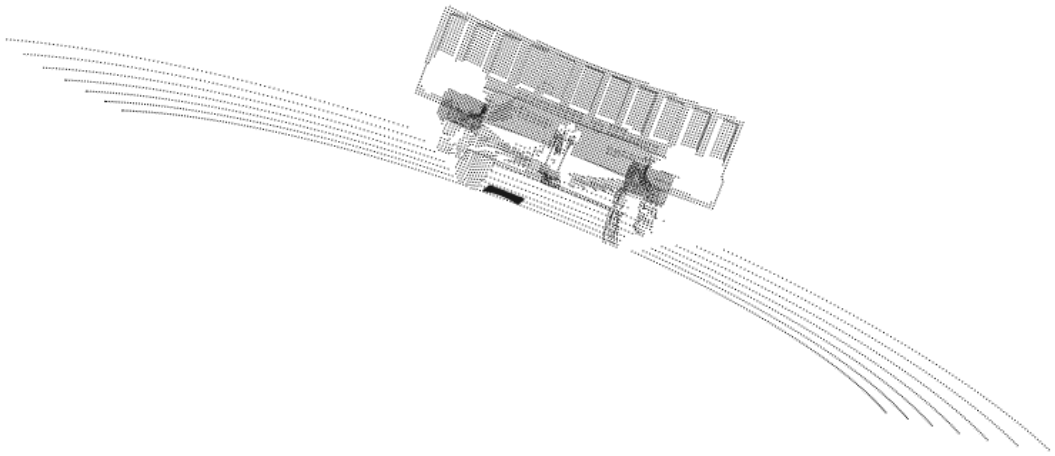
1.1 Model 0



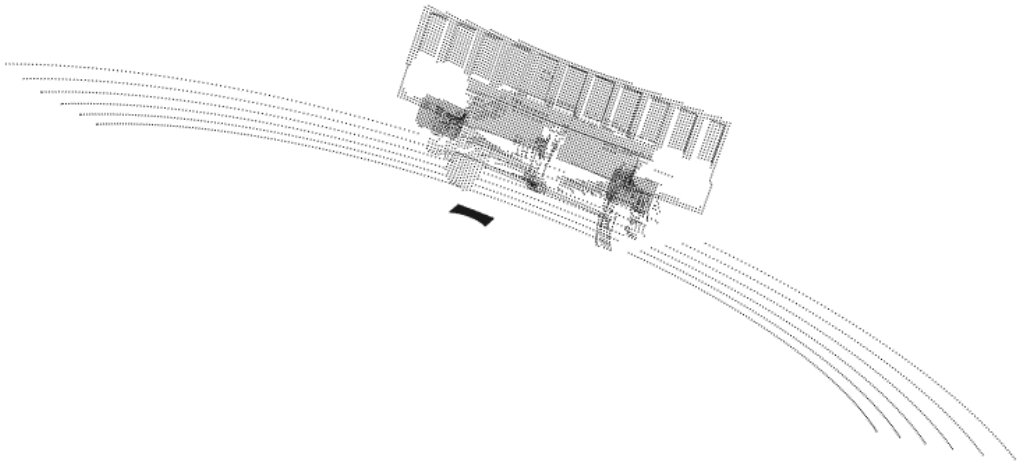
Point cloud 1: Statistical Outlier Removal setMeanK = 10 setStddevMulTresh = 0.2



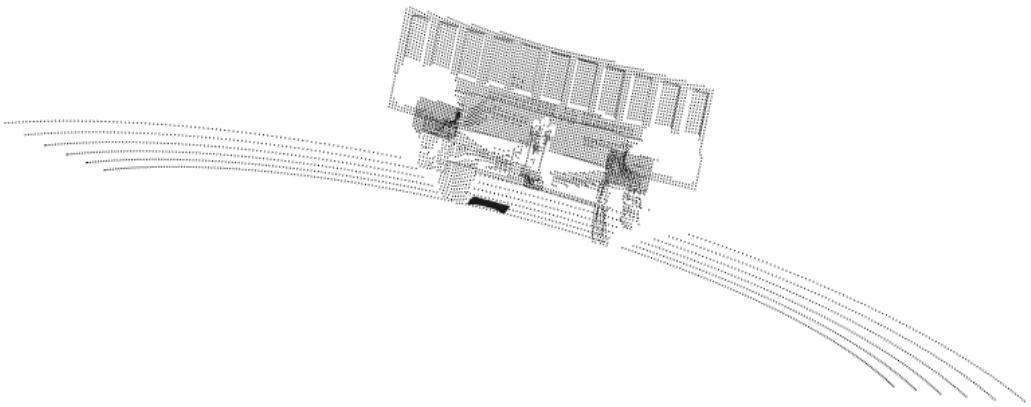
Point cloud 2: Statistical Outlier Removal setMeanK = 10 setStddevMulTresh = 0.1



Point cloud 3: Statistical Outlier Removal setMeanK = 20 setStddevMulTresh = 0.05



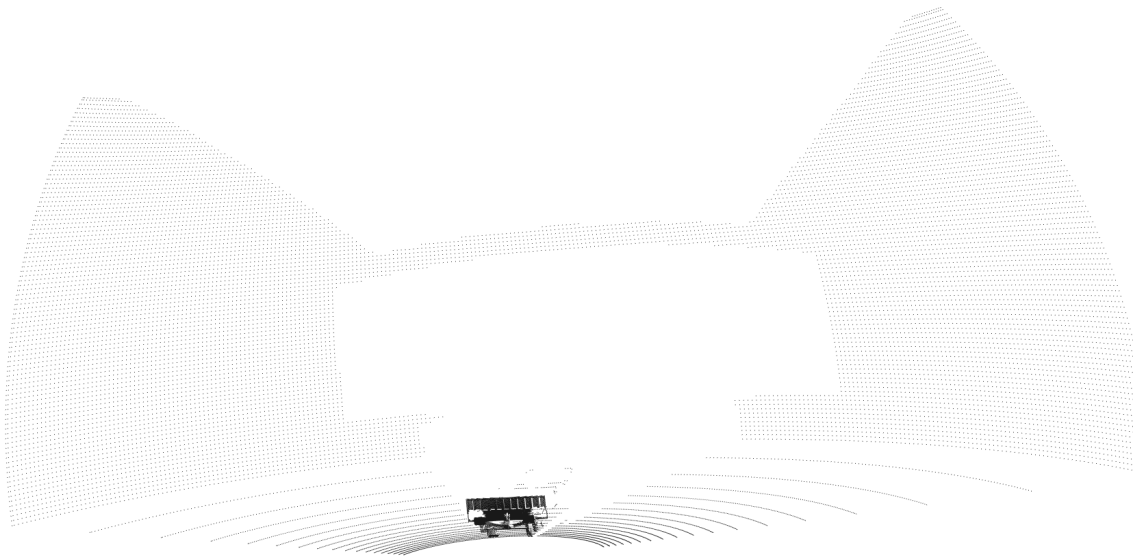
Point cloud 4: Statistical Outlier Removal setMeanK = 20 setStddevMulTresh = 0.01



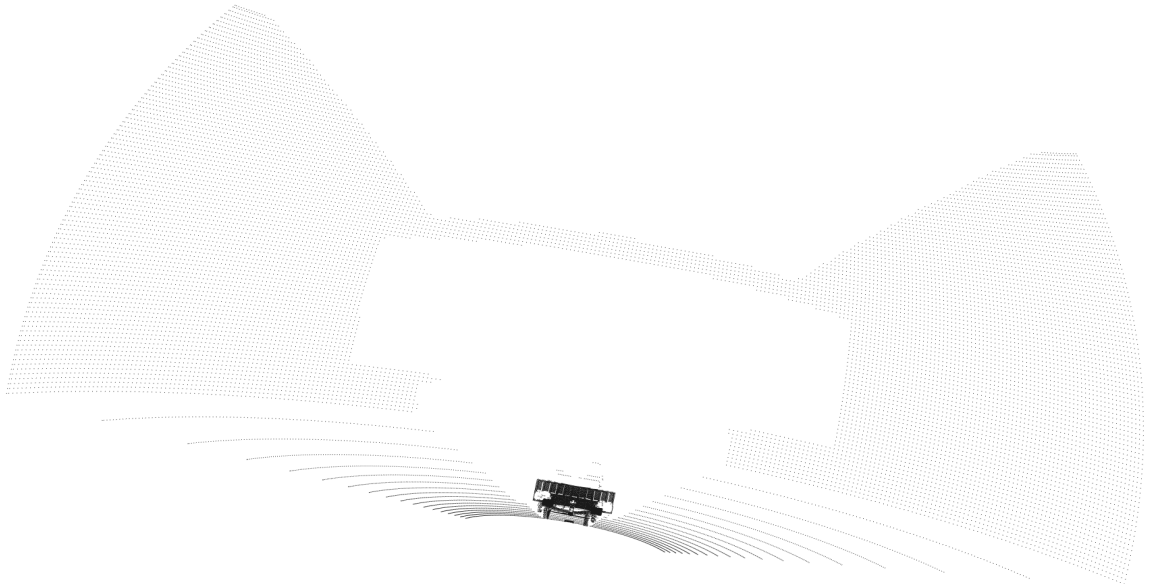
Point cloud 5: Statistical Outlier Removal setMeanK = 20 setStddevMulTresh = 0.001

2. Radius Outlier Removal

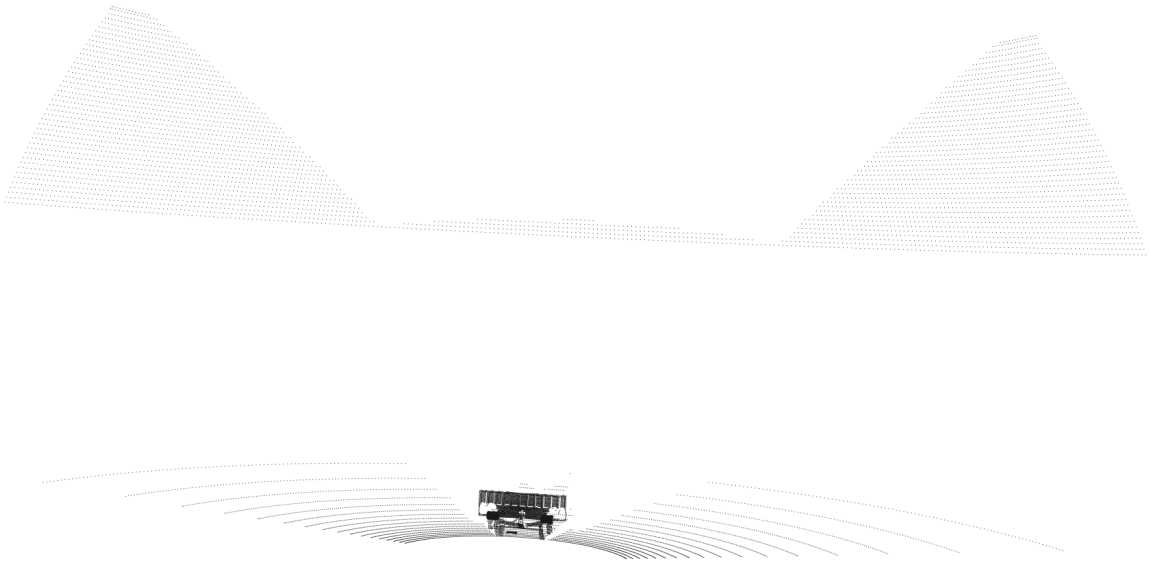
2.1 Model 0



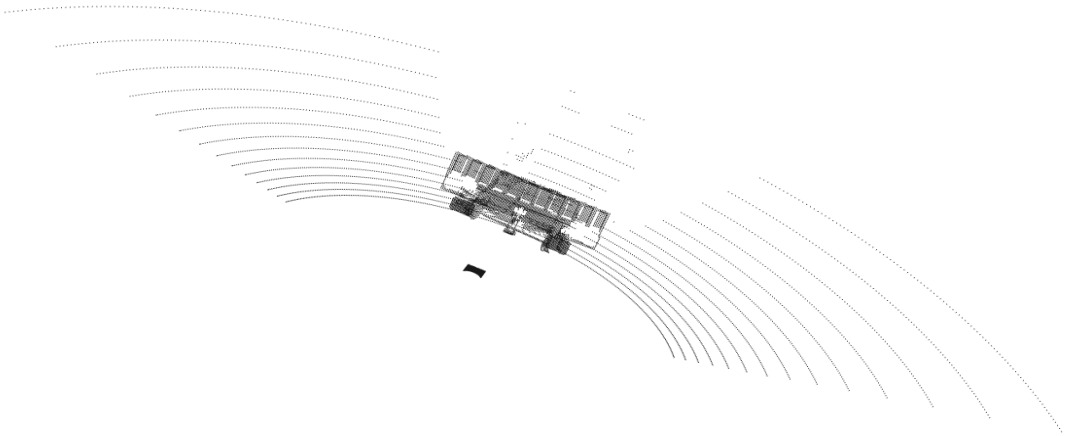
Point cloud 6: Radius Outlier Removal setMinNeighborsInRadius = 2 setRadiusSearch = 0.8



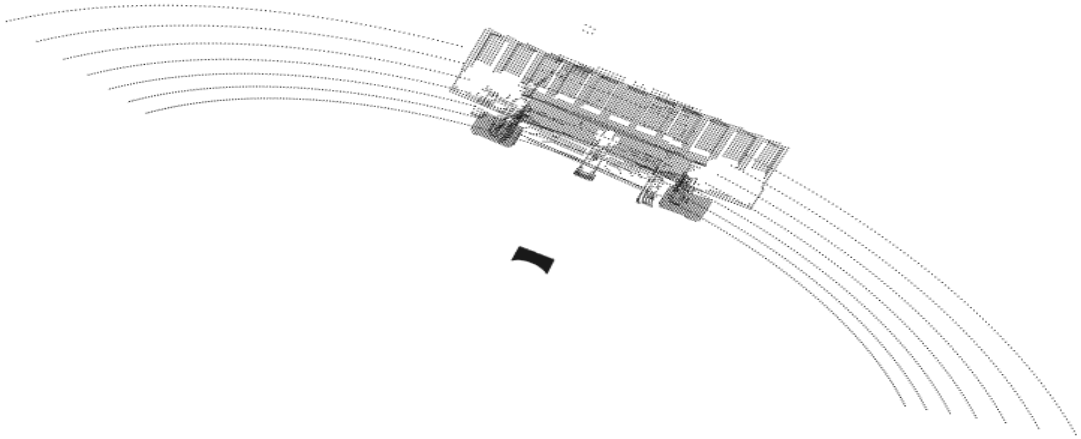
Point cloud 7: Radius Outlier Removal setMinNeighborsInRadius = 2 setRadiusSearch = 0.4



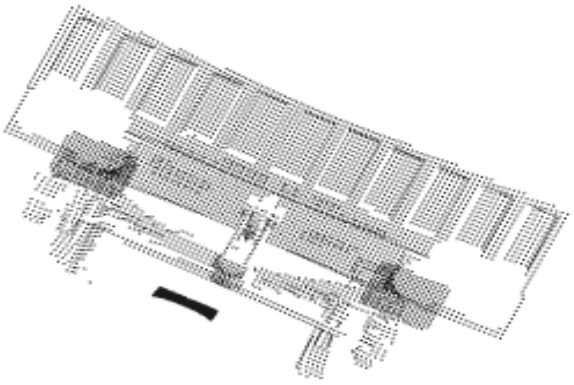
Point cloud 8: Radius Outlier Removal setMinNeighborsInRadius = 2 setRadiusSearch = 0.2



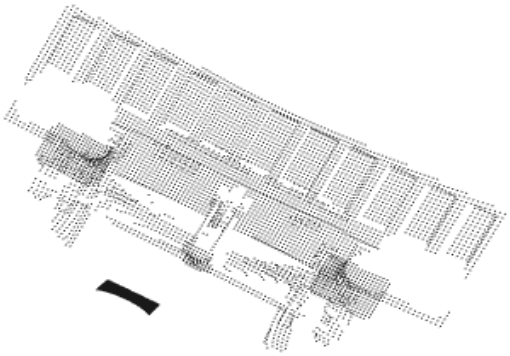
Point cloud 9: Radius Outlier Removal setMinNeighborsInRadius = 2 setRadiusSearch = 0.1



Point cloud 10: Radius Outlier Removal setMinNeighborsInRadius = 4 setRadiusSearch = 0.1

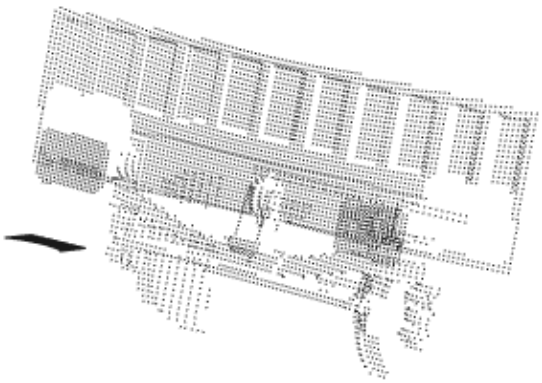


Point cloud 11: Radius Outlier Removal setMinNeighborsInRadius = 6 setRadiusSearch = 0.1



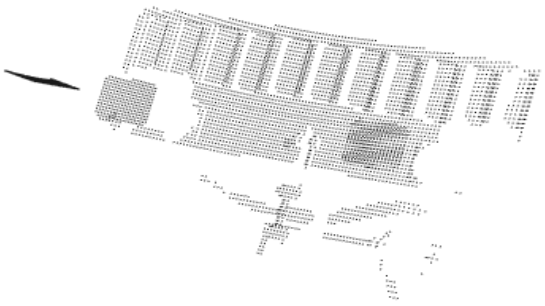
Point cloud 12: Radius Outlier Removal setMinNeighborsInRadius = 8 setRadiusSearch = 0.1

2.2 Model 1



Point cloud 13: Radius Outlier Removal setMinNeighborsInRadius = 6 setRadiusSearch = 0.1

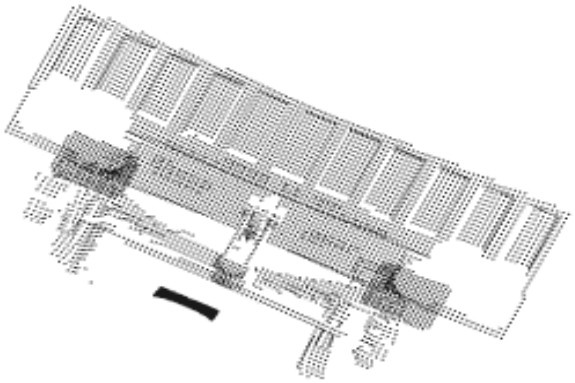
2.3 Model 2



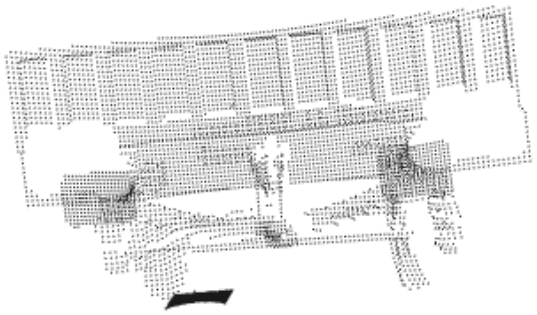
Point cloud 14: Point cloud 15: Radius Outlier Removal setMinNeighborsInRadius =9 setRadiusSearch = 0.06

3. Voxel Downsampling

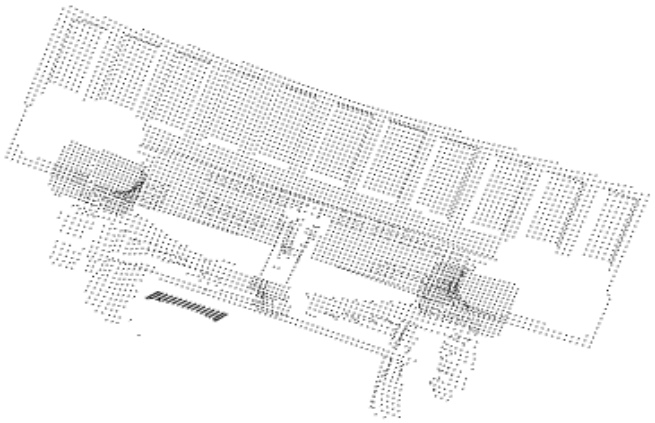
3.1 Model 0



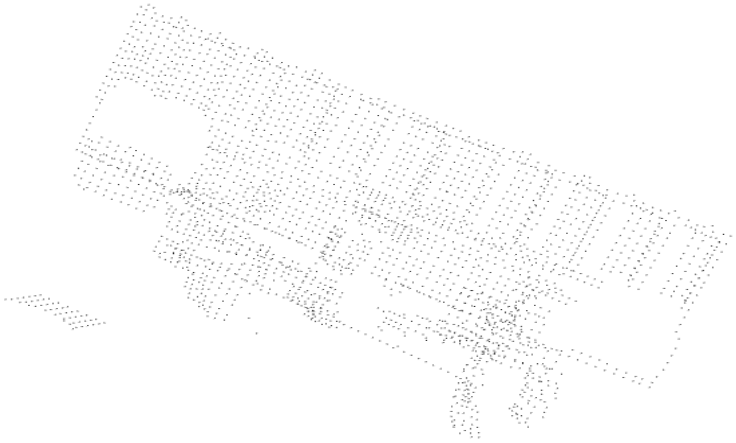
Point cloud 16: VoxelGrid Downsampling setLeafSize = 0.001f



Point cloud 17: VoxelGrid Downsampling setLeafSize = 0.01f

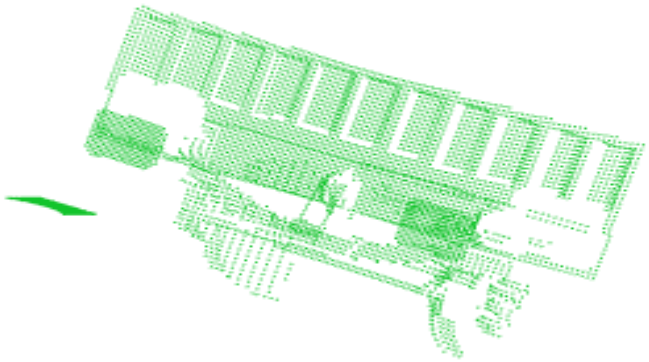


Point cloud 18: VoxelGrid Downsampling setLeafSize = 0.02f



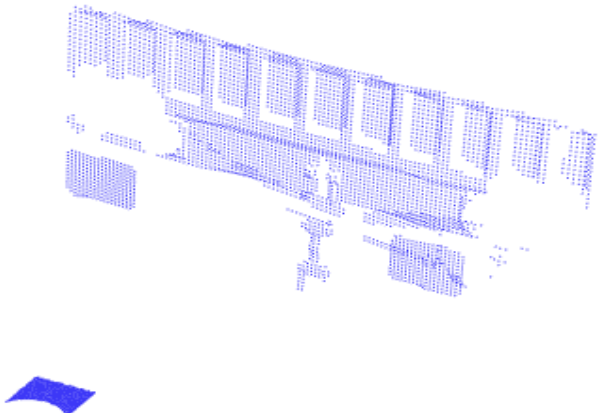
Point cloud 19: VoxelGrid Downsampling setLeafSize = 0.04f

3.2 Model 1



Point cloud 20: VoxelGrid Downsampling setLeafSize = 0.01f

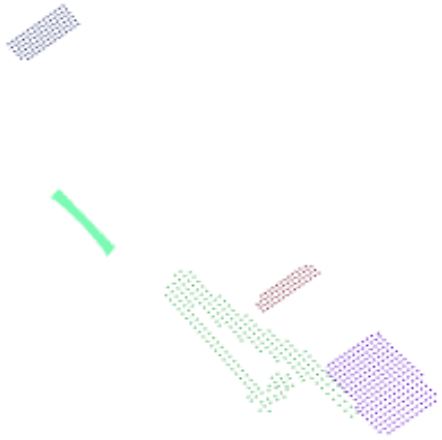
3.3 Model 2



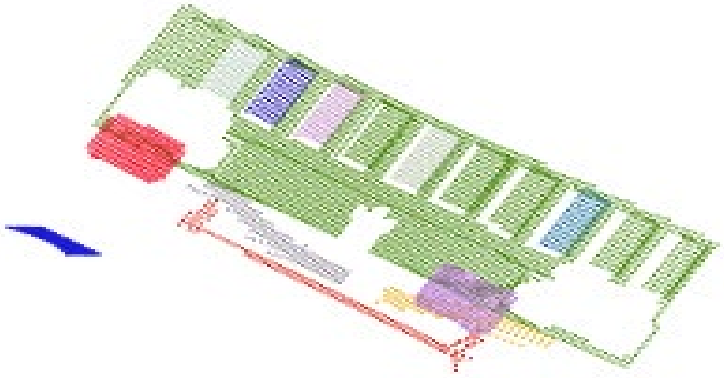
Point cloud 21. VoxelGrid Downsampling setLeafSize = 0.01f

4. Cluster Extraction

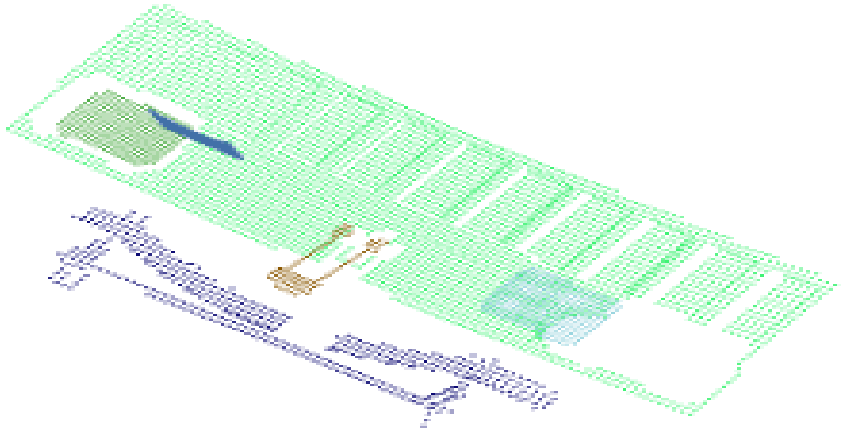
4.1 Model 0



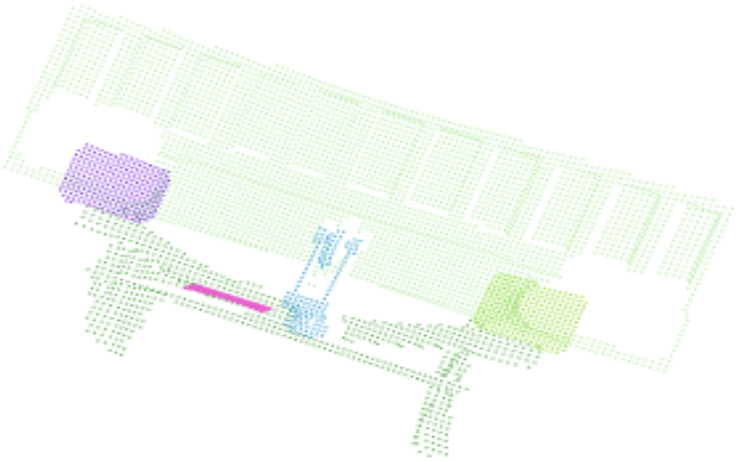
Point cloud 22: Euclidean Cluster Extraction setClusterTolerance = 0.02



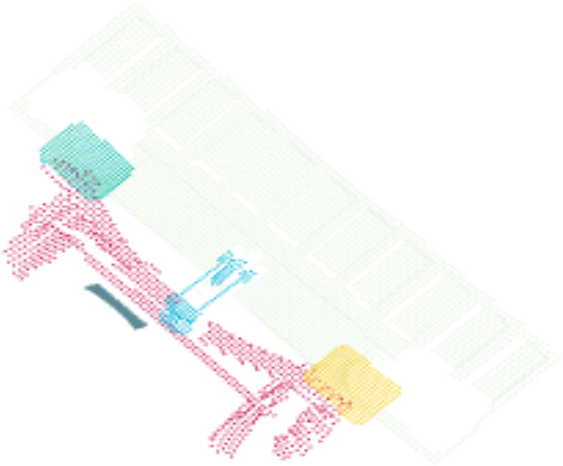
Point cloud 23: Euclidean Cluster Extraction setClusterTolerance = 0.04



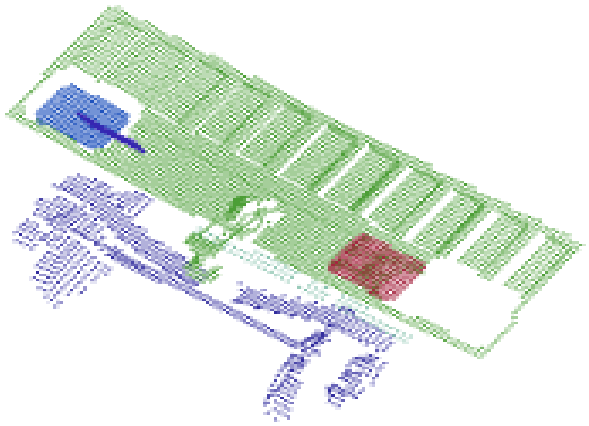
Point cloud 24: Euclidean Cluster Extraction setClusterTolerance = 0.05



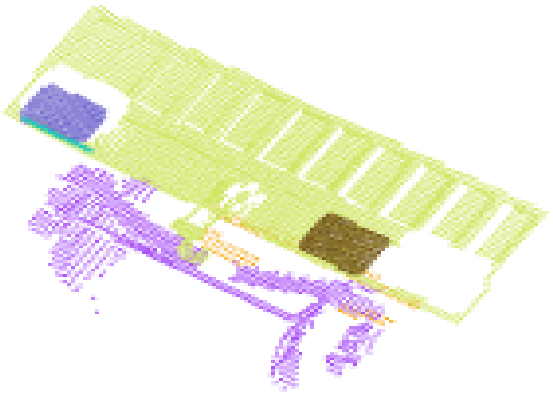
Point cloud 25: Euclidean Cluster Extraction setClusterTolerance = 0.07



Point cloud 26: Euclidean Cluster Extraction setClusterTolerance = 0.1

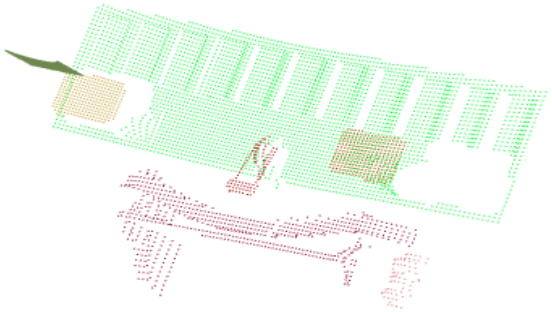


Point cloud 27: Euclidean Cluster Extraction setClusterTolerance = 0.15



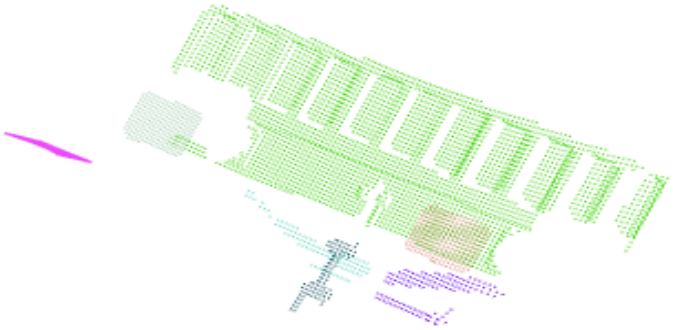
Point cloud 28: Euclidean Cluster Extraction setClusterTolerance = 0.18

4.2 Model 1



Point cloud 29: Euclidean Cluster Extraction setClusterTolerance = 0.07

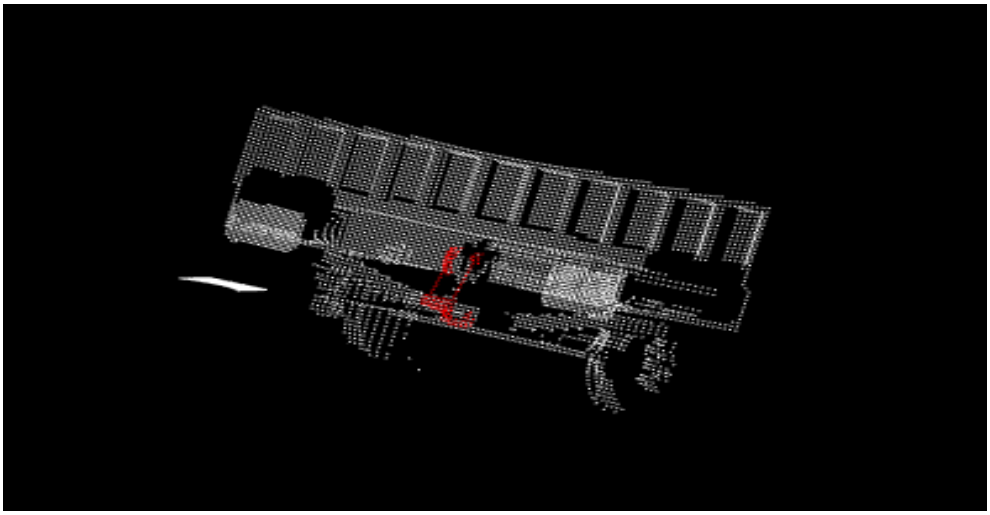
4.3 Model 2



Point cloud 30: Euclidean Cluster Extraction setClusterTolerance = 0.07

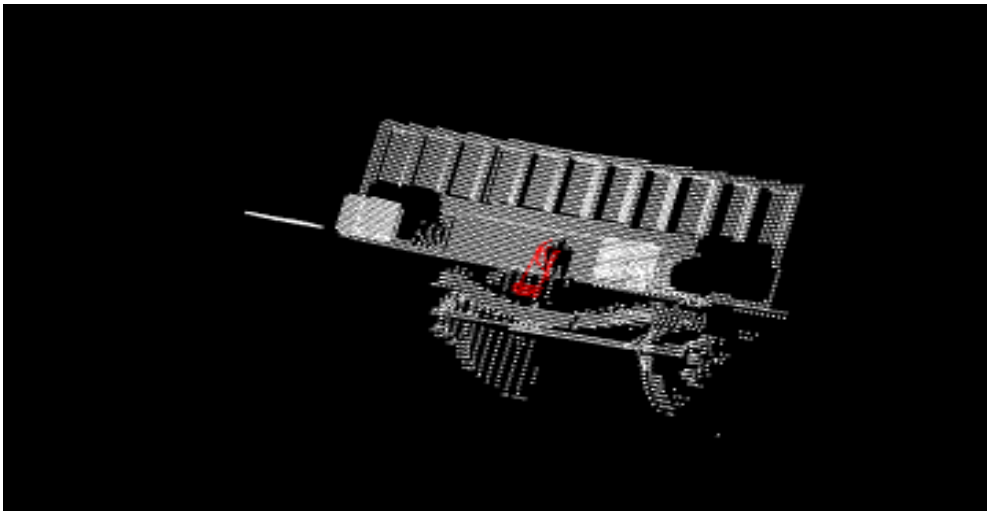
5. Recognition

5.1 Model 0



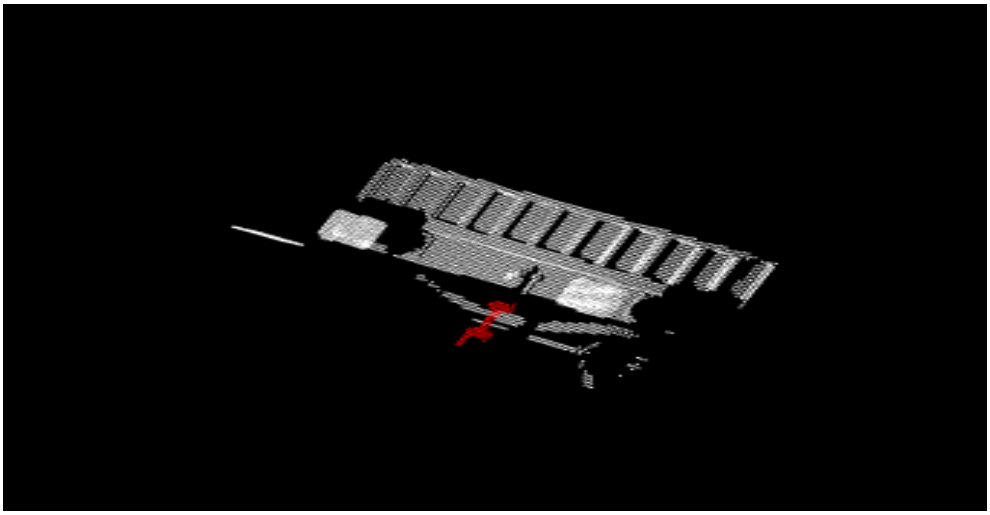
Point cloud 31: Recognition of UIC Hook in Model 0

5.2 Model 1



Point cloud 32: Recognition of UIC Hook in Model 1

5.3 Model 2



Point cloud 33: Recognition of UIC Hook in Model 2